МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧЕРНІГІВСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНОЛОГІЧНИЙ УНІВЕРСИТЕТ
КАФЕДРА ІНФОРМАЦІЙНИХ ТА КОМП’ЮТЕРНИХ СИСТЕМ

ОСНОВИ ПРОГРАМУВАННЯ НА С/С++

Методичні вказівки
до виконання лабораторного практикуму
з дисципліни «Алгоритми та структури даних»
для студентів спеціальності
123 "Комп'ютерна інженерія"

Затверджено
на засіданні кафедри
інформаційних і комп’ютерних систем

Протокол № 1 від «27» серпня 2018 р.

Чернігів ЧНТУ 2018

Основи програмування на С/С++. Методичні вказівки до лабораторного практикуму з дисципліни «Алгоритми та структури даних» для студентів напрямів підготовки 123 – “Комп’ютерна інженерія”. /Укл.: Риндич Є.В., Солдатов А.Ю. – Чернігів: ЧНТУ, 2018. – 180 с.

Укладачі: Риндич Євген Володимирович, кандидат технічних наук, доцент, доцент кафедри інформаційних та комп’ютерних систем;
Солдатов Артем Юрійович, асистент кафедри інформаційних та комп’ютерних систем.

Відповідальний за випуск: С.В. Зайцев, зав. кафедрою інформаційних та комп’ютерних систем, д-р. техн. наук, доцент.

Рецензент: Є. В. Нікітенко, кандидат фізико-математичних наук, доцент, доцент кафедри інформаційних та комп’ютерних систем

ЗМІСТ

Лабораторна робота № 1 8
1.1 Короткі теоретичні відомості 8
1.1.1 Логічна структура пам’яті програми 8
1.1.2 Вказівники. Типізовані вказівники. 9
1.1.3 Змінні в статично-розподільчій пам'яті та динамічні змінні. Оператори та функції С ++ для управління динамічною пам'яттю 15
1.1.4 Стани вказівників 17
1.2 Приклади роботи з вказівниками 18
1.2.1 Два вказівники містять адресу однієї и тієї ж змінної 18
1.2.2 Ситуація витоку пам’яті 19
1.3 Завдання для самоперевірки 20
1.4 Створення проекту 23
1.5 Проведення досліджень по використанню типізованих вказівників в програмі 25
1.6 Зміст звіту 25
Лабораторна работа №2 27
2.1 Теоретичні відомості 27
2.1.1 Поняття «список» та правила його побудови 27
2.1.1 Реалізація зв'язаного односпрямованого списку. 28
2.1.2.1 Основні операції реалізації списку 28
2.1.2.2 Опис типів та оголошення змінних для побудови списку. ... 30
2.1.2.3 Ініціалізація списку 32
2.1.2.4 Функції додавання елементів до списку 32
2.1.2.4.2 Функція додавання елементу в кінець списку 35
2.1 Приклад створення проекту для обробки списку 40
2.1.2 Задача, вирішувана проектом 40
2.1.3 Інтерфейс проекту 40
2.1.4 Модуль типів та глобальних змінних проекту 41
2.1.5 Модуль функцій проекту 44
2.1.6 Основний модуль проекту 46
2.3 Завдання для отримання навичок «зв’язування» елементів в списку 49
2.4 Завдання для виповнення проекту 50
2.5 Тестування проекту 52
2.6 Зміст звіту 53
Лабораторна работа № 3 54
3.1 Теоретичні відомості 54
3.1.1 Додавання елемента в початок двоспрямованого списку 55
3.1.2 Додавання елемента у кінець двоспрямованого списку 56
3.1.3 Додавання елемента у відсортований двоспрямований список 56
3.1.4 Видалення елемента з двоспрямованого списку 58
3.1.5 Пошук елемента у двоспрямованому списку 59

3.1.6 Перевірка порожнечі двоспрямованого списку ... 60
3.1.7 Ініціалізація та створення двоспрямованого списка цілих чисел ... 60
3.1.8 Видалення двоспрямованого списку ... 61
3.2 Створення проекта по обробці двоспрямованого списку ... 61
3.2.1 Завдання, що вирішується проектом ... 61
3.2.2 Опис типів для проекта ... 61
3.2.3 Реалізація функції ініціалізації та створення списку випадкових чисел 62
3.2.4 Реалізація функції видалення елементу з двоспрямованого списку ... 66
3.3 Завдання для виконання проекта ... 67
3.4 Тестування проекта ... 67
3.5 Вміст звіта ... 68
Лабораторна работа № 4 ... 70
4.1 Теоретичні відомості ... 70
4.2 Створення проекту, якиий забезпечує створення та додавання елементів до мультисписку ... 72
4.2.1 Задача, вирішувана проектом ... 72
4.2.2 Інтерфейс проекту ... 72
4.2.3 Опис типів для проекту ... 73
4.2.4 Опис модулів проекту ... 73
4.2.5 Реалізація функції ініціалізації мультиспиcку ... 74
4.2.6 Реалізація функції додання елементів до мультиспиcку ... 74
4.2.7 Реалізація функції видалення елемента ... 78
4.2.8 Реалізація функції виводу вмісту мультиспиcку на екран ... 79
4.2.9 Реалізація функції видалення мультиспиcку ... 80
4.3 Завдання для виконання проекту ... 80
4.4 Тестування проекту ... 82
4.5 Вміст звіту ... 82
Лабораторна работа № 5 ... 83
5.1 Теоретичні відомості ... 83
5.3 Основні процедури для реалізації списку в двійковому файлі ... 85
5.3.1 Відкриття файла для списку ... 85
5.3.2 Ініціалізація списку ... 86
5.3.3 Вивід списку на екран ... 87
5.3.4 Пошук фізичного місця для розташування нового елемента у файлі ... 87
5.3.5 Вивільнення физичного місця в файлі після видалення елемента зі списку ... 88
5.3.6 Додавання нового елементу в віддсортований список ... 89
5.3.7 Пошук елемента у відсортованому списку ... 90
5.3.8 Видалення елементу із відсортованого списку ... 91
5.3.9 Видалення списку ... 91
5.4 Создание проекта, который обеспечивает создание и обработку списка, расположенного в двоичном файле ... 92
5.4.1 Задача, що вирішується проектом ... 92
5.4.2. Інтерфейс проекту ... 92

5.4.3 Опис типів для проекту 93
5.4.4 Реалізація основного модуля проекту 94
5.5 Завдання для виповнення проекту 96
5.6 Тестування проекту 98
5.7 Вміст звіту 98
Лабораторна робота №6 99
6.1 Теоретичні відомості 99
6.1.1 Поняття стек та основні операціїї при його обробці 99
6.1.2 Реалізація стеку 100
6.1.3 Поняття черги і основні операції по її обробці 105
6.1.4 Реалізація черги 108
6.1.5 Поняття дек і основні операції при його обробці 114
6.2 Завдяння для виконння проекту 115
6.3 Тестування проекту 120
6.4 Зміст звіту 120
Лабораторна работа №7 121
Реалізація рекурсивних підпрограм для обробки списків 121
7.1 Теоретичні відомості 121
7.1.1 Визначення понятть рекурсивний об'єкт та рекурсивна підпрограма 121
7.1.2 Особливості виконання рекурсивних підпрограм 122
7.1.3 Особливості рекурсивної обробки списків 125
7.1.4 Приклад аналізу та рішення задачі реалізації рекурсивної процедури добавки елементу в отсортований за зростанням список 126
7.1.5 Заміна рекурсії роботою з власним стеком 130
7.2 Завдання для виконання лабораторної роботи 131
7.3 Тестування проекту 132
7.4 Зміст звіту 132
Лабораторна работа №8 133
8.1 Теоретичні відомості 133
8.1.1 Поняття бінарного дерева. Види бінарних дерев 133
1.7 135
8.1.2 Способи подання бінарних дерев 138
8.1.3 Методи проходження бінарних дерев 141
8.1.4 Впорядковані бінарні дерева (або двійкові дерева пошуку) і основні операції з ними 146
8.1.5 Додавання елемента в бінарне дерево 147
8.1.6 Пошук елемента в бінарному дереві 147
8.1.7 Видалення елемента з бінарного дерева 148
8.2 Виконання проекту 149
8.3 Завдання для виконання лабораторної роботи 152
8.4 Завдання з виконання обходу бінарних дерев 153
8.5 Зміст звіту 153

Лабораторна робота №9 155
9.1.1 Короткі теоретичні відомості 155
9.1.2 Загальний опис STL 155
9.1.3 Контейнери STL 156
9.1.4 Ітератори STL 159
9.1.5 Шаблони в C++ 161
9.2 Зміст звіту 170
Лабораторна робота №10 171
10.1 Короткі теоретичні відомості 171
10.1.1 Загальні поняття і визначення 171
10.1.2 Приклад організації хеш-таблиці 172
10.1.3 Колізії і їх розширення 173
10.1.4 Типи алгоритмів хешування 173
10.1.5 Основні операції з хеш-таблицею 173
10.1.6 Функції хешування 174
10.1.7 Спискові алгоритми відкритого хешування 175
10.1.7.1 Основні функції спискових алгоритмів відкритого хешування 175
10.1.8 Алгоритми закритого хешування 176
10.1.8.1 Приклад реалізації алгоритма закритого хешування з лінійним рехешуванням 177
10.2 Завдання для виконання лабораторної роботи 178
10.3 Тестування проекту 179
10.4 Зміст звіту 179
Перелік використаної літератури 180

ВСТУП

Мова С++ надає ефективні та гнучкі засоби визначення нових типів. Використовуючи визначення нових типів, що відповідають поняттям проблемної області (з додатками), програміст може розбивати розроблювану програму на частини, якими легко контролювати, (принцип «розділяй і володарюй»).

Новий тип створюється користувачем для визначення поняття, якому серед вбудованих типів немає відповідності. Набагато легше зрозуміти програму, в якій створюються типи, що відповідають поняттям проблемної області. Добре визначені типи роблять програму ясною і короткої. Крім того, компілятор може виявити неприпустиме використання даних.

Такий метод побудови програм називають абстракцією даних. Інформація про типи міститься в об'єктах, тип яких визначається користувачем. Програмування з застосуванням об'єктів називають об'єктно-орієнтованим ([1,2]).

Мова С ++ є мовою об'єктно-орієнтованого програмування (ООП). Автор мови створював його з метою підтримки абстракції даних і об'єктно-орієнтованого програмування.

Об'єктно-орієнтована мова - це мова програмування, в якій програма задається описом поведінки сукупності взаємопов'язаних об'єктів. Об'єктно-орієнтоване програмування має справу з об'єктами. Об'єктно-орієнтовані мови включають такі основні риси: інкапсуляція даних, поліморфізм, спадкування.

Об'єкти можуть включати закриті (private) дані і правила їх обробки, доступні тільки об'єкту, захищені (protected), доступні об'єкту і його спадкоємцям, а також загальні (public) дані і правила, які доступні об'єктам і модулям в інших частинах програми.

Методичні вказівки містять опис виконання п’яти лабораторних робіт. Студент виконує лабораторну роботу в Qt Creator або Microsoft Visual Studio в залежності від варінта, який визначається останньою цифрою залікової книжки. По кожній роботі студент повинен оформити звіт. Звіти оформляються за допомогою текстового редактора Word на папері формату А4, у відповідності з вимогами стандартів на оформлення технічної документації. Звіт по роботі є розділом підсумкового документа. В кінці семестру звіти зшиваються в єдиний підсумковий документ з титульним листом, підписуються у керівника, після чого студент отримує допуск до іспиту.

Лабораторна робота № 1
Статичні і динамічні змінні.
Вказівники.

Цілі роботи:

- Засвоїти поняття «динамічний розподіл пам'яті», «статичний розподіл пам'яті», «робота зі змінними при статичному і динамічному розподілі пам'яті», «вказівники».
- Отримати практичні навички у створенні динамічних змінних різних типів.
- Створити найпростіший проект в системі QT Creator на мові C ++, в якому здійснюється робота зі змінними в динамічної пам'яті.

1.1 Короткі теоретичні відомості

1.1.1 Логічна структура пам'яті програми

Традиційно під час запуску програми, їй виділяється певна ділянка оперативної пам'яті, яка під час її роботи, в залежності від призначення і способу управління даними, поділяється на такі розділи:

Рисунок 1.1 – Розподіл пам'яті програми до початку її виконання

Область коду – це пам'ять, яка використовується для зберігання коду програми, функцій. Ця область пам'яті виділяється під час запуску програми, до початку її виконання, для розміщення в ній даних. Її розмір залишається незмінним до закінчення роботи програми. Тому всі функції мають глобальний час життя й існують протягом усього часу виконання програми.

Статичний розподіл пам'яті – це пам'ять, яка виділяється під час запуску програми, до початку її виконання, для розміщення в ній даних. Її розмір залишається незмінним до закінчення роботи програми.

У статичної пам'яті розміщуються глобальні, статичні змінні.

Стекова пам'ять використовується для зберігання локальних змінних під час виклику функцій.

Динамічна пам’ять — це сукупність блоків пам'яті, що виділяються з доступної вільної оперативної пам'яті («купи») безпосередньо під час виконання програми для розміщення конкретних об'єктів. У процесі роботи програми розмір динамічної пам'яті, займаний нею, може змінюватися.

«Купа» - частина вільної оперативної пам'яті комп'ютера, виділена програмі для розміщення в ній динамічно створюваних об'єктів.

Під час динамічного розподілу пам'яті потрібний простір пам'яті вибирається з «купи». В результаті розподіл пам'яті для програми набуває вигляду, зображеному на рисунку 1.2:

Рисунок 1.2 – Розподіл пам'яті програми під час її виконання

Деякі сучасні мови програмування, наприклад С ++, мають засоби для підтримки механізмів розподілу пам'яті засобами управління «купою». Для цього в них використовуються вказівники і вбудовані процедури і функції.

1.1.2 Вказівники. Типізовані вказівники.

Звернення до змінних в динамічно-розподільчій пам'яті (динамічним змінним) здійснюється через вказівник.

Як відомо, пам'ять комп'ютера розбита на клітинки розміром 1 байт. Всі вони пронумеровані. Номер клітинки пам'яті - це і є адреса цієї клітинки.

Якщо змінна займає декілька байтів пам'яті, то її адресою буде номер першого байта займаної ділянки.

Вказівник – це змінна, призначена для зберігання деякої адреси пам'яті.

У вказівнику можна зберігати адресу даних (змінної) або підпрограми (функції).

Змінна типу вказівник займає, на сьогоднішній день, в пам'яті 4 байта.

Крім значення конкретної адреси вказівник може містити значення

NULL. Це означає, що вказівник не вказує ні на яку комірку пам'яті.

Вказівники поділяються на **безтипові** і **типізовані**.

Безтиповий вказівник (вказівник на невизначений тип) – містить адресу деякого байта пам'яті. Якщо не використовувати деякий механізм перетворення типів, то за допомогою нього неможна звернутися до конкретної змінної. Безтипові вказівники розглядаються в наступному семестрі. Слід зазначити, що в мові програмування С ++ безтиповий вказівник оголошується наступним чином (див. рисунок 1.3):

void* ім’я_ вказівника;

Рисунок 1.3 – Приклад оголошення безтипового вказівника

При оголошенні такого вказівника замість типу адресованих їм даних вказується ключове слово **void**.

Типізований вказівник – це змінна, призначена для зберігання адреси змінної певного типу.

Для типізованого вказівника немає вбудованого типу, його повинен описувати сам програміст.

Використовуючи операцію **розіменування** вказівника, за допомогою типізованого вказівника можна звернутися до змінної в динамічній або статичній пам'яті.

Вказівник може містити як адресу змінної в статичній пам'яті, так і адресу змінної в динамічній пам'яті.

1.1.2.1 Оголошення вказівника на певний тип.

Оголошення змінної вказівника на певний тип в мові С ++ (далі таку змінну будемо називати просто «вказівник») в загальному вигляді має вид (див. рисунок 1.4):

тип* ім’я_змінної;

де <тип> - це тип змінної, адресу якого буде містити вказівник

Рисунок 1.4 – Оголошення вказівника в загальному вигляді

Особливістю оголошення вказівника є використання символу «*». В даному випадку символ «*» позначає, що наступна за нею змінна є вказівником.

Наведемо приклад оголошення типізованого вказівника, призначеного для зберігання адреси змінної цілого типу int (див. Рисунок 1.5).

```
int * p_i;
```

Рисунок 1.5 – Приклад оголошення змінної типізованого вказівника на змінну типу int

При формуванні назви змінної вказівника будемо дотримуватися наступного правила:

Назва будь-якої змінної типу вказівник буде починатися з літери p (pointer - вказівник).

Якщо для вирішення певної задачі необхідно оголосити змінну-вказівник на структуру або масив, то спочатку необхідно описати тип для запису або масиву, а після цього оголошувати вказівник.

1.1.2.2 Оголошення вказівників.

Приклад оголошення змінних типу «збірний вказівник» показано на рисунку 1.6:

```
   typedef  struct
{
char Name[30];
int    Age;
char Address[30];
}  TStruct; //*тип для структури

  TStruct  * pStruct;  // оголошення вказівник на структуру типу TStruct
   int  * pi;                // оголошення вказівник на змінну типу int
```

Рисунок 1.6 – Приклад оголошення вказівника на структуру і вказівника на змінну типу int

Для оголошення типізованого вказівника можна використовувати заздалегідь описаний для нього тип. Приклад оголошення двох типізованих вказівників, з використанням заздалегідь описаних типів показаний на рисунку 1.7.

```
typedef  struct
                {
                char Name[30];
                int    Age;
                char Address[30];
}  TStruct; //тип для структури

Typedef   TStruct  * TPStruct; // оголошення вказівникна структуру
Typedef   Tint *  TPInt;  // тип оголошення вказівник на змінну типу int
      TPStruct  *  pStruct;  // оголошення змінної-вказівника на структуру типу TStruct
       TPInt   p_i;  // оголошення змінної-вказівника на змінну типу int
```

Рисунок 1.7 – Приклад оголошення вказівника на структуру і вказівника на

змінну типу int з використанням заздалегідь описаних типів для них

Якщо вказівники мають різний тип, то не можна вказівником одного типу привласнити значення вказівника іншого типу. Іншими словами, якщо обидва вказівника, які беруть участь в операції привласнення, типізовані, то обидва вони повинні вказувати на об'єкти одного і того ж типу.

```
TStruct  * pStruct;  // оголошення вказівника на структуру типу TStruct
int  * pInt;  // оголошення вказівника на змінну типу int

pInt = pStruct;  // неприпустима операція
```

Рисунок 1.8 – Приклад неприпустимої операції з вказівником

У C ++ неможна вказівнику явно привласнити конкретне число. (Див. рисунок 1.9)

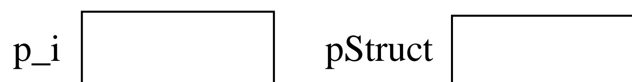
```
int  * pInt;  // оголошення вказівника на змінну типу int

pInt = 4895778;  // неприпустима операція
```

Рисунок 1.9 – Приклад неприпустимої операції з вказівником

При виконанні блоку програми з рисунку 1.6 в статично-розподіленій (або стековій) пам'яті з'являться дві змінні типу вказівник, які будуть займати по 4 байта кожна. При цьому слід зазначити, що в динамічно-розподільчій пам'яті ніяких змінних не з'явилося. (див. рисунок 1.10)

Статична пам’ять



Динамічна пам’ять

Рисунок 1.10 – Розміщення в пам'яті змінних після виконання програми на рисунку 1.6

1.1.2.3 Допустимі операції для вказівників на певний тип

У мові C ++ для типізованих вказівників припустимі наступні операції:

1. **Привласнення**

Вказівнику можна привласнити значення іншого вказівника такого ж типу

(рисунок 1.11):

```
int  * p1;
int * p3;

p1=p3;  // припустима операція, так як вказівники однакових типів
```

Рисунок 1.11 – Припустима операція - привласнення значення одного вказівника іншому

Вказівнику будь-якого типу можна привласнити значення константи NULL. Ця константа оголошується в одному із стандартних бібліотечних модулів і містить значення, яке не може бути адресою комірки пам'яті. Зазвичай це значення 0. Якщо вказівник містить значення NULL, то він називається нульовим вказівником, що означає, що він не вказує ні на яку комірку пам'яті. (Див. рисунок 1.12)

```
{
int  * p1;
int * p3;

p1=NULL;  // припустима операція

}
```

Рисунок 1.12 – Допустима операція - присвоювання значення константи NULL вказівнику

Вказівнику можна привласнити адресу іншої змінної, використовуючи оператор взяття адреси &.

Безтиповому вказівнику можна привласнити значення типізованого вказівника, але не навпаки. (Див. рисунок 1.13)

```
{
int  *   p1; // змінна - типізований вказівник
void * p3;  // змінна - вказівник на невизначений тип

int     d;  // змінна типу int

p1=&d;   // вказівнику p1 присвоюється адреса змінної d

p3=p1;   // безтиповому вказівнику p3 присвоюється значення типізованого вказівника p1

p1=p3; // заборонена операція

}
```

Рисунок 1.13 – Допустима операція - присвоювання вказівником адреси статичної змінної

2. Перевірка на рівність і нерівність

Значення вказівників можна порівнювати між собою.
Значення вказівника можна порівнювати зі значенням NULL.
Приклад таких операцій див. на рисунку 1.14.

```
{
int  *   p1; // змінна - типізований вказівник

void * p3;  // змінна - вказівник на невизначений тип

int     d;  // змінна типу int

p1=&d;  // вказівнику p1 присвоюється адреса змінної d

p3=&d; // вказівнику p3 присвоюється адреса змінної d

  // Приклад порівняння значень вказівників
 if  (p1 != p3)   {p3=NULL;} // якщо значення вказівників p1 і p3 не рівні, то значенню вказівника
p3 присвоюється NULL

 // Приклад порівняння значення вказівника зі значенням NULL
 if (p3 == NULL) {p1 = NULL;} // якщо значення вказівника p3 дорівнює NULL, то значенню
вказівника p1 присвоюється NULL

}
```

Рисунок 1.14 – Приклади операцій порівняння вказівників

3. Розіменування вказівника

Для доступу до змінної через вказівник використовується операція разіменування, яка в C ++ позначається як - «*».

Звернення до значення змінної через вказівник в загальному вигляді та приклад звернення має вигляд, показаний на малюнку 1.15.

Звернення до змінної через вказівник на неї в загальному вигляді:

*** Типізований вказівник =**

Приклад:

```
{
   int    d;  // змінна цілого типу в статично-розподільчій пам'яті
   int * p1;  // змінна типізований вказівник, яка знаходиться в статично-розподільчій пам'яті

 d = 14;  // в змінну d заносимо значення 14
 p1 = &d; // вказівнику p1 присвоюється адреса змінної d

 // Тепер можемо звернутися до змінної d не по імені, а через
// вказівник, використовуючи операцію разіменування вказівника p1
```

```
// Наприклад, збільшимо значення змінної d на 5

*p1 = *p1 + 5; // збільшення значення змінної d на 5
               // тепер в змінній d міститься значення 19

}
```

Рисунок 1.15 – Загальний вигляд та приклад використання операції разіменування вказівника.

Операція * p1 для комп'ютера означає наступне: «Взяти адресу в змінній p1 та звернутися до змінної за цією адресою».

При зверненні до змінної через вказівник замість її імені пишеться «* вказівник», в іншому ж робота з таким записом нічим не відрізняється від звичайної.

1.1.3 Змінні в статично-розподільчій пам'яті та динамічні змінні. Оператори та функції С ++ для управління динамічною пам'яттю

Динамічні змінні - змінні, які створюються в пам'яті під час виконання програми.

Динамічні змінні не мають власних імен, звернення до них здійснюється тільки за допомогою вказівників.

Для створення динамічних змінних в С ++ існують вбудований оператор new та функція malloc (). Для роботи з типізованим вказівником перевагу слід віддавати оператору new.

Робота з функцією malloc буде розглядатися в наступному семестрі.Для розміщення змінної в динамічній пам'яті використовується оператор new. (Див. рисунок 1.16):

```
вказівник  = new  тип;
```

Рисунок 1.16 – Використання оператора new в загальному вигляді

Оператор new запрошує вільну ділянку пам'яті розміром, залежним від типу змінної, на яку буде вказувати вказівник у виконавчої системи та адресу першого байта, виділеної ділянки пам'яті, заносить у вказівник.

В результаті роботи оператора new в динамічній пам'яті з'являється змінна - динамічна змінна.

Наведемо приклад створення динамічної змінної цілого типу int та занесення в неї значення, рівного 36. Приклад програми для вирішення такого завдання показаний на рисунку 1.17.

```
{
  int   * p1 ;   // оголошуємо змінну - типізований вказівник
                  // в статично-розподільчій пам'яті виділено 4 байта для
                            // вказівник p1 - в ньому «сміття»
   // Поки що в пам'яті немає динамічної змінної
  // Тепер використовуємо оператор new для запиту пам'яті для динамічної
 // змінної

  p1 = new int ;

 // Тепер в динамічній пам'яті з'явилася змінна типу int
 // і змінна-вказівник p1 містить її адресу (див. рисунок 1.17)

// Заносимо в змінну в динамічній пам'яті значення 36, шляхом
// разіменування вказівника P1

 *p1 = 36;
 }
```

Рисунок 1.16 – Приклад створення динамічної змінної і занесення в неї певного значення.

В результаті дій, виконаних програмою з малюнка 1.16, в пам'яті програми дані будуть представлені таким чином (Рисунок 1.17):

Для оператора і функції управління динамічною пам'яттю new і malloc () існують оператор і функція, які виконують протилежну дію - звільняють динамічну пам'ять.

Якщо пам'ять було запитано за допомогою оператора new, тоді її звільнення проводиться оператором delete:

delete вказівник;

Якщо ж пам'ять було отримано функцією malloc, то звільнить її виклик функції free ():

В результаті роботи обох механізмів звільнення пам'яті вказівник приймає невизначене значення, **але НЕ NULL.**

Розглянемо відмінності між змінними в статично-розподільчій пам'яті і динамічними змінними, які показані в таблиці 1.1.

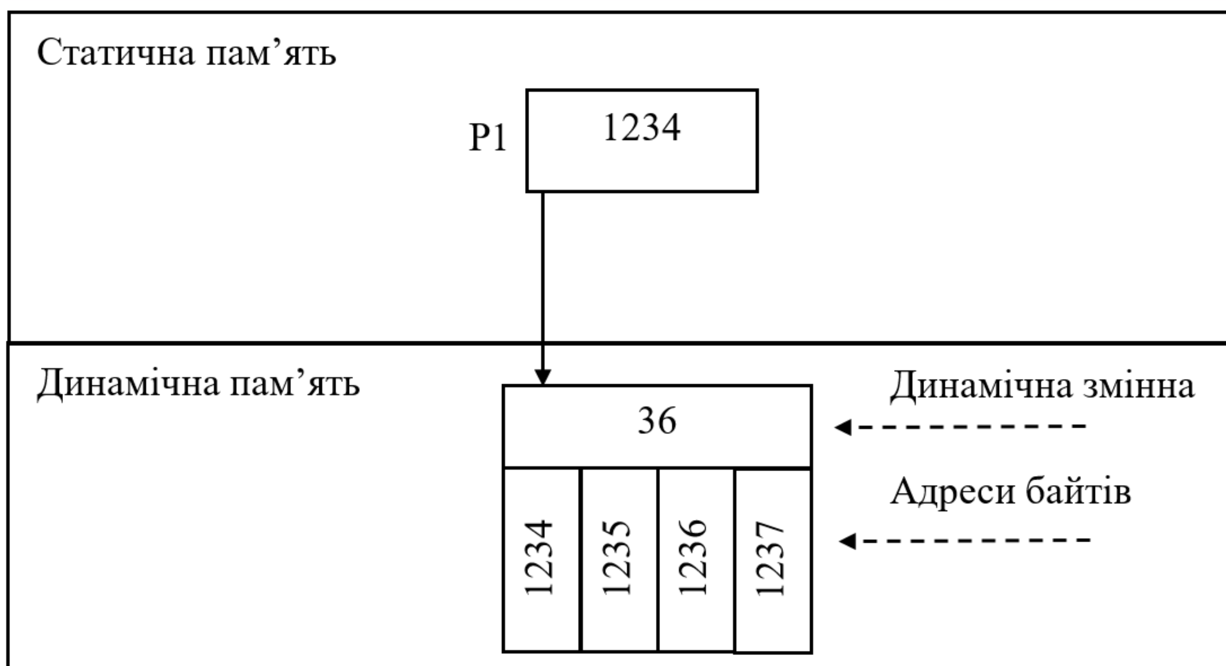


Рисунок 1.17 – Представлення даних в пам'яті після виконання програми рисунка 1.16

Таблиця 1.1 – Порівняльні характеристики змінної в статично-розподільчій пам'яті і динамічній

Параметр порівняння	Змінна в статичній пам'яті	Динамічна
Момент появи	З'являється в момент, коли компілятор обробляє рядок оголошення змінних	З'являється в результаті виконання оператора new або функції malloc ()
В який момент видаляється	У момент закінчення програми або підпрограми	У результаті виконання оператора звільнення пам'яті - delete або free ()

Крім перерахованих відмінностей, більш ніякої різниці немає при роботі зі змінними в статичної і динамічної пам'яті.

1.1.4 Стани вказівників

Вказівники можуть мати 3 стани:

1. Ініційований вказівник. Вказівник, який вказує на якісь дані.
2. Порожній вказівник. Вказівник, що містить порожнє значення - NULL.
3. Сміттєвий вказівник. Дуже небезпечний тип вказівника. Він містить якесь значення, але не вказує нікуди. Вказівник виявляється в такому стані відразу після оголошення і після того як пам'ять, на яку він вказує, вже звільнили. Помилки, засновані на спробах використання сміттєвих вказівників, доставляють найбільше клопоту. Тому рекомендується

своєчасно встановлювати невживаним вказівникам значення NULL і перевіряти їх при використанні на це значення.

1.2 Приклади роботи з вказівниками

1.2.1 Два вказівники містять адресу однієї и тієї ж змінної

Ситуація, коли два вказівника містять адресу однієї і тієї ж змінної може бути зображена на рисунку 1.18.

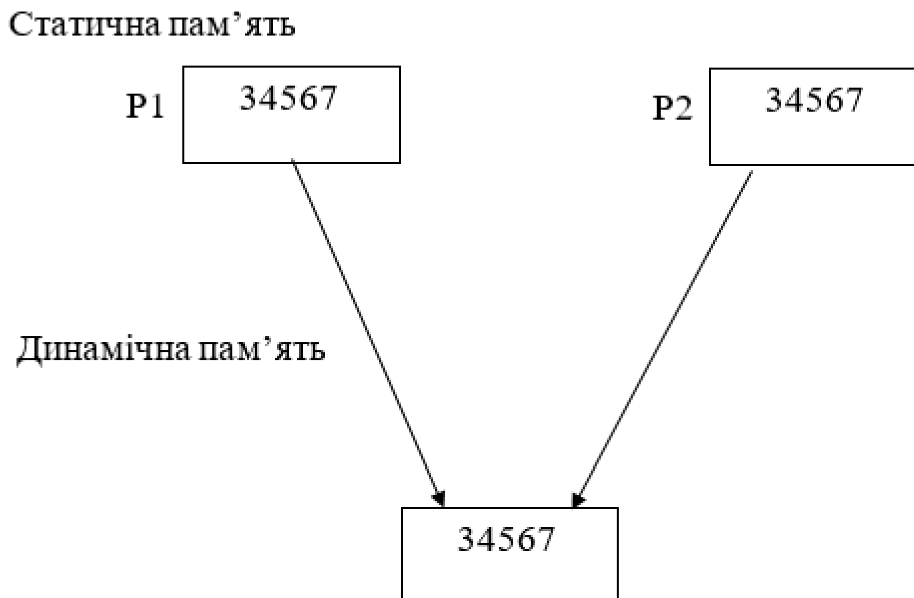


Рисунок 1.18 – Ситуація, коли два вказівника містять адресу однієї і тієї ж змінної

Для створення вищеописаної ситуації може бути виконана програма, проілюстрована на рисунку 1.19.

Слід звернути увагу на те, що на рисунку 1.18 зображена тільки одна динамічна змінна. Отже, тільки один раз в програмі буде викликано оператор New.

```
{
int  *p1,  *p2 ; // оголошуємо змінні - типізовані вказівники
                              // p1, p2 - в них «сміття»

    // Поки що в пам'яті немає динамічної змінної
   // Тепер викликаємо оператор new для запиту пам'яті для динамічної
  // змінної
 p1 = new int;

 // Тепер в динамічній пам'яті з'явилася змінна типу int
  // і змінна-вказівник p1 містить її адресу (див.рисунок 1.18)

 // Заносимо в змінну в динамічній пам'яті значення 36, шляхом
 // розіменування вказівника p1

*p1 = 36;
```

```
// Тепер необхідно в змінну p2 занести таке ж значення, як
 // в p1. Це дві статичні змінні. Шляхом присвоювання копіюємо
// значення однієї змінної в іншу. Вони однотипні.

p1 = p2;

}
```

Рисунок 1.19 – Програма, виконання якої призведе до розміщення даних в пам'яті, відповідно рисунку 1.18

1.2.2 Ситуація витоку пам’яті

Ситуація, коли в динамічній пам'яті знаходиться зайнята ділянка і жоден з наявних вказівників не містить її адресу, називається витоком пам'яті. До такої ділянки пам'яті вже не можливо отримати доступ, його неможливо звільнити до закінчення роботи програми і найгірше - він займає місце, яке не може бути виділено іншій змінній.

Витік пам'яті виникає тоді, коли програміст забуває звільняти пам'ять перед тим, як вказівник отримує адресу іншої ділянки пам'яті.

Така ситуація зображена на рисунку 1.20.

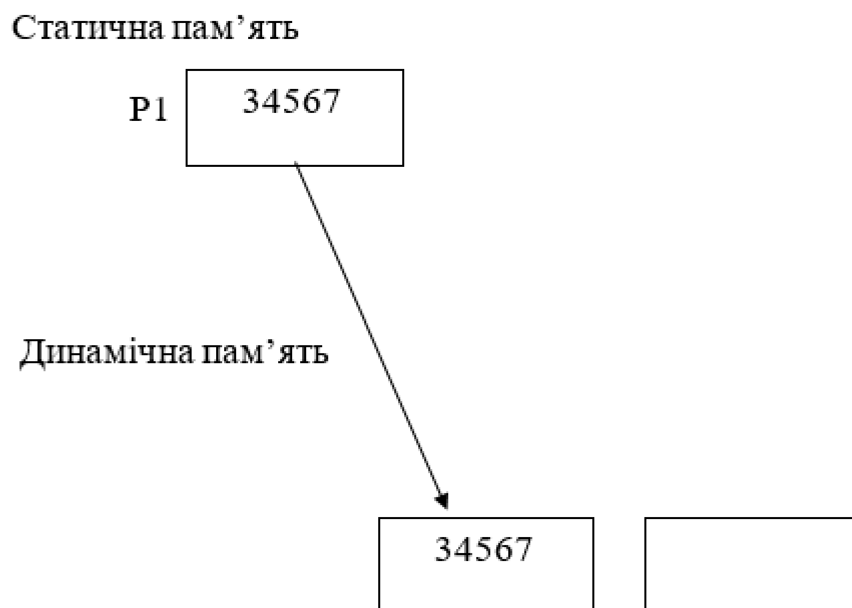


Рисунок 1.20 – Ситуація витоку пам’яті

Для створення вищеописаної ситуації може бути виконана програма, зображена на рисунку 1.21.

Слід звернути увагу на те, що на малюнку 1.20 зображено дві динамічні змінні. Отже, в програмі оператор new буде викликано двічі.

```
{
int  *p1;   // оголошуємо змінну - типізований вказівник
            //  p1 - в ній «сміття»

   // Поки що в пам'яті немає динамічної змінної
   // Тепер викликаємо оператор new для запиту пам'яті для динамічної
  // змінної

 p1 = new int;

 // Тепер в динамічній пам'яті з'явилася змінна типу int
  // і змінна-вказівник p1 містить її адресу (див. рисунок 1.18)

// Викликаємо ще раз процедуру new, в результаті чого p1 отримає адресу
// знову виділеної ділянки пам'яті, а попередня ділянка буде втрачена


p1 = new int;

// Виник витік пам'яті
}
```

Рисунок 1.21 – Приклад програми, виконання якої призведе до витоку пам'яті

1.3 Завдання для самоперевірки

Як завдання для засвоєння матеріалу виконайте наступне (можна без програмної реалізації) (Як приклад використовуйте рисунок 1.16):

Завдання 1 (виконати письмово).

1. Розмістіть в динамічній пам'яті дві змінні типу float і занесіть в одну з них значення кореня квадратного від 2.18, а в іншу - синус від значення, яке знаходиться в першій змінній.

2. Розмістіть в динамічній пам'яті дві змінні типу структура, яка містить інформацію про викладача: Прізвище, Кафедра, Посада, Зарплатня. Одну змінну заповніть наступною інформацією: Іванов, ІКС, Доцент, 5000.
Потім скопіюйте значення заповненої першої змінної в другу і тільки змініть значення поля Прізвище на «Петров».

3. Розмістіть в динамічній пам'яті змінну типу масив з трьох цілих чисел і заповніть її перший елемент значенням 5.
4. Розмістіть в динамічній пам'яті масив з 10 структур, що містять ту ж інформацію, що і записи в пункті 2 і заповніть третій елемент цього масиву

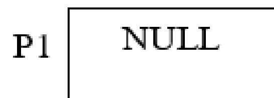
інформацією про вашого викладача.

Завдання 2

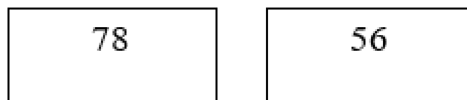
Напишіть програми, які забезпечать наступне розташування даних в пам'яті:

1.

Статична пам'ять

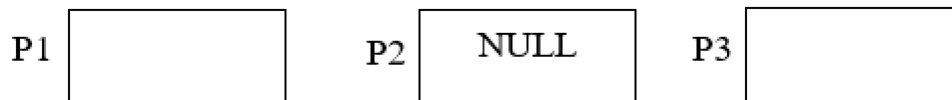


Динамічна пам'ять

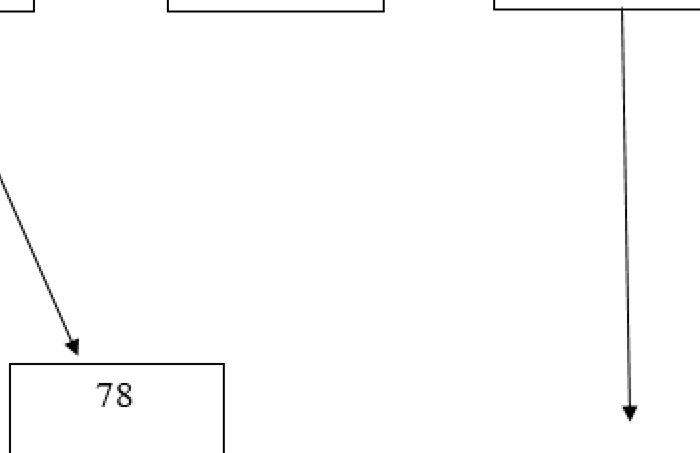


2.

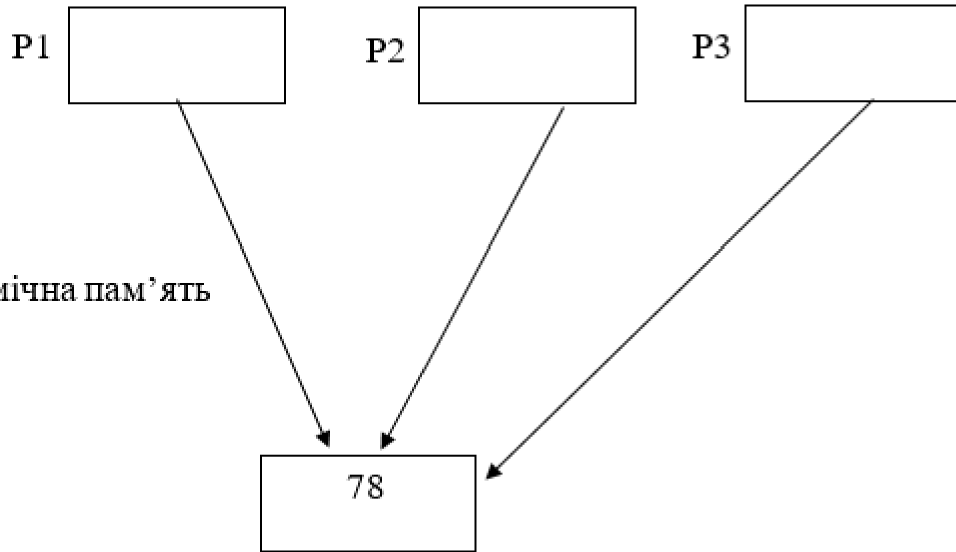
Статична пам'ять



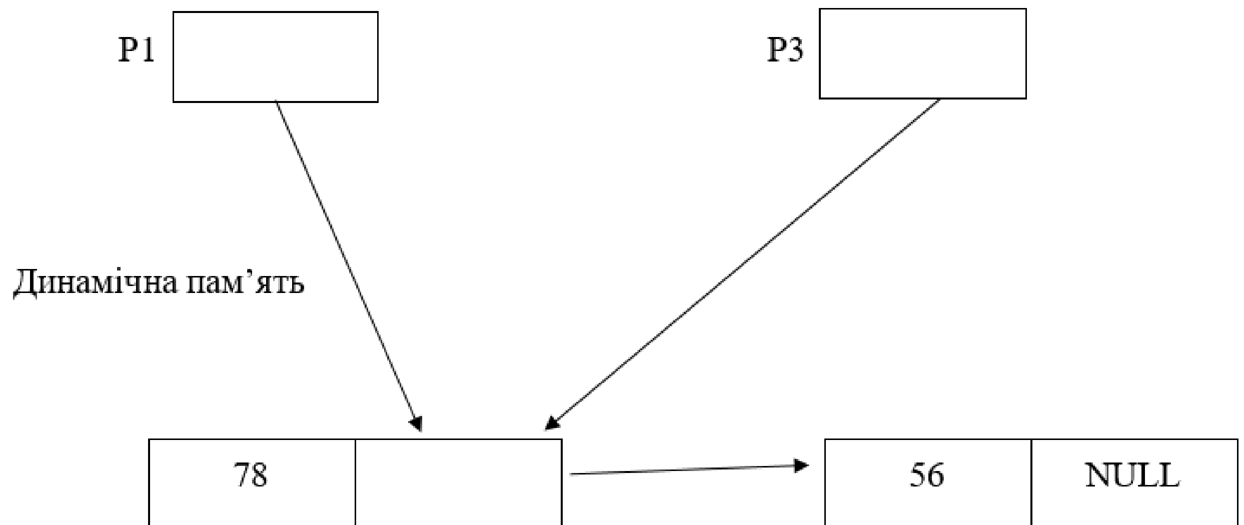
Динамічна пам'ять



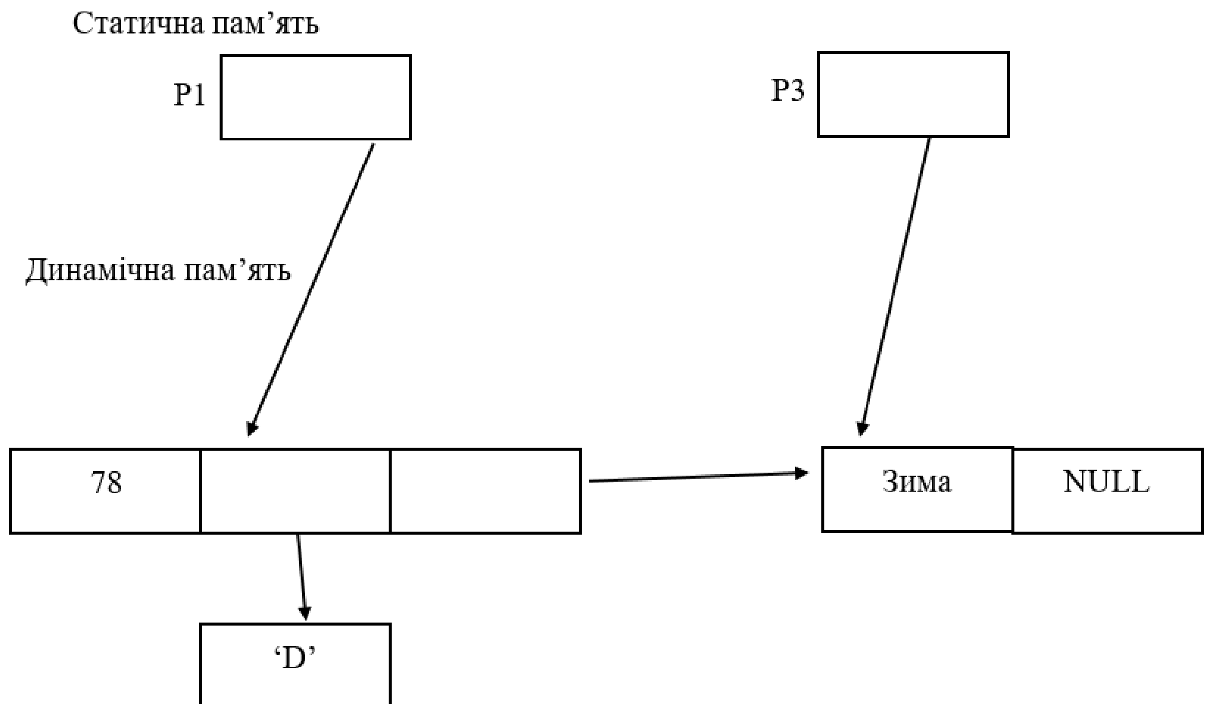
3.



4.



5.



1.4 Створення проекту

1.4.1 Мета створення проекту

Метою проекту є розміщення трьох змінних типу int, float, double в динамічній пам'яті і виведення їх значень на екран, а також аналіз значень вказівників на ці змінні до виділення пам'яті і після та проведення аналізу цих досліджень.

1.4.2 Створення інтерфейсу проекту

Інтерфейс програми повинен виглядати так як показано на рис.1.22.
Вихідні значення для розміщення в динамічній пам'яті:
int, float, double: 12, 3.12, 4.789

Значення вказівників до розміщення змінних у пам'яті:
P1 = …   ;
P2 = …   ;
P3= …   ;

Значення вказівників після розміщення в пам'яті:

P1 = …   ;
P2 = …   ;
P3= …   ;

Значення динамічних змінних:
*P1 = …   ;
*P2 = …   ;
*P3= …   ;

Рисунок 1.22 – Зображення інтерфейсу проекту

1.4.3 Створення функції розміщення змінних в динамічній пам'яті.

Створіть функцію, як показано на малюнку 1.23

```
void  varInDynamicMemory( ) {
p1 * int;
p2 * float;
p3 * double;

i int;
f float;
d double;

cout<<"Вихідні значення для розміщення в динамічній пам'яті:\n";
cout<<"int  float  double: ";
cin>>i>>f>>d;
cout"\n";

int iP1=(int)p1;
int iP2=(int)p2;
int iP3=(int)p3;


cout<<"Значення вказівників до розміщення змінних у пам'яті:\n";
cout<<"P1 ="<<iP1<<"\n";
cout<<"P2 ="<<iP2<<"\n";
cout<<"P3 ="<<iP3<<"\n";

   /* Звертаємося до пам'яті для змінних в динамічній пам'яті */
p1=new int;
p2=new float;
p3=new double;

  /* Беремо значення вказівників після виділення пам'яті під динамічні змінні */

int iP1=(int)p1;
int iP2=(int)p2;
int iP3=(int)p3;

     /* Заповнюємо значеннями динамічні змінні */

*p1 = i;
*p2 = f;
*p3 = d;
```

```
cout<<"Значення вказівників після розміщення змінних у пам'яті:\n";
cout<<"P1 ="<<iP1<<"\n";
cout<<"P2 ="<<iP2<<"\n";
cout<<"P3 ="<<iP3<<"\n";

cout<<"Значення динамічних змінних:\n";

i=*p1;
f=*p2;
d=*p3;

cout<<" *P1 ="<<i<<"\n";
cout<<" *P2 ="<<f<<"\n";
cout<<" *P3 ="<<d<<"\n";


}
```

Рисунок 1.23 – Функція розміщення динамічних змінних в пам'яті комп'ютера

1.4.4 Запуск проекту

Запустіть проект на виконання та результати перенесіть в звіт у вигляді зображення інтерфейсу проекту.

1.5 Проведення досліджень по використанню типізованих вказівників в програмі

1. За результатами роботи проекту проведіть аналіз початкових значень вказівників, до їх ініціалізації та виділення динамічній пам'яті.
2. Проведіть аналіз значень вказівників після розміщення змінних в динамічній пам'яті. Зверніть увагу, на скільки байт відрізняються значення адрес кожного наступного вказівника, та який розмір в байтах кожної динамічної змінної. Зробіть висновок про те, яка кількість байт виділяється під кожну змінну.
3. У режимі трасування подивіться значення вказівників та значення змінних, на які вони вказують до та після запиту пам'яті. Результати занесіть у таблицю та зробіть висновок про те, чи можна через вказівник звернутися до деякої ділянки пам'яті, якщо в ньому «сміття».
4. Оголосіть додатковий вказівник та надайте йому значення NULL. Потім спробуйте разіменувати його. Про результати виконання цієї ділянки програми напишіть висновки.

1.6 Зміст звіту

– 1. Назва роботи.

- 2. Мета роботи.
- 3. Мета проекта.
- 4. Тексти програм рішення завдань відповідно до пункту 1.3.
- 4. Рисунок с зображення інтерфейсу проекта.
- 5. Текст програми.
- 6. Висновки про результати проведених досліджень відповідно до пункту 1.5.

Лабораторна работа №2

Створення лінійного односпрямованого списку з вузловим представленням.

Алгоритми обробки списку з використанням одного вказівника.

Мета роботи:
- Засвоїти поняття «лінійно зв’язаний список».
- Ознайомитись з правилами створення лінійних списків.
- Отримати практичні навички з написання функцій обробки списків з використанням одного вказівника.
- Створити проект на мові C++, в якому здійснюється робота з однонаправленим списком та вивід його на екран.

2.1 Теоретичні відомості

2.1.1 Поняття «список» та правила його побудови.

Основне призначення вказівників - створення спискові динамічних структур даних: списків, дерев, графів.

Динамічна структура даних може змінювати свій розмір в ході роботи програми.

Список - структура даних, що складається з елементів, які називаються вузлами, пов'язаних між собою за допомогою вказівників.

Кожен елемент списку складається, як мінімум, з двох полів:

1. Поле інформації;

2. Поле, що містить адресу наступного елемента списку.

Одним з видів спискові структур є лінійний односпрямований пов'язаний список, який представляється в такий спосіб (див. рисунок 2.1). Такий список складається з однотипних елементів - структур, кожна з яких має додаткове поле – вказівник на наступний елемент списку.

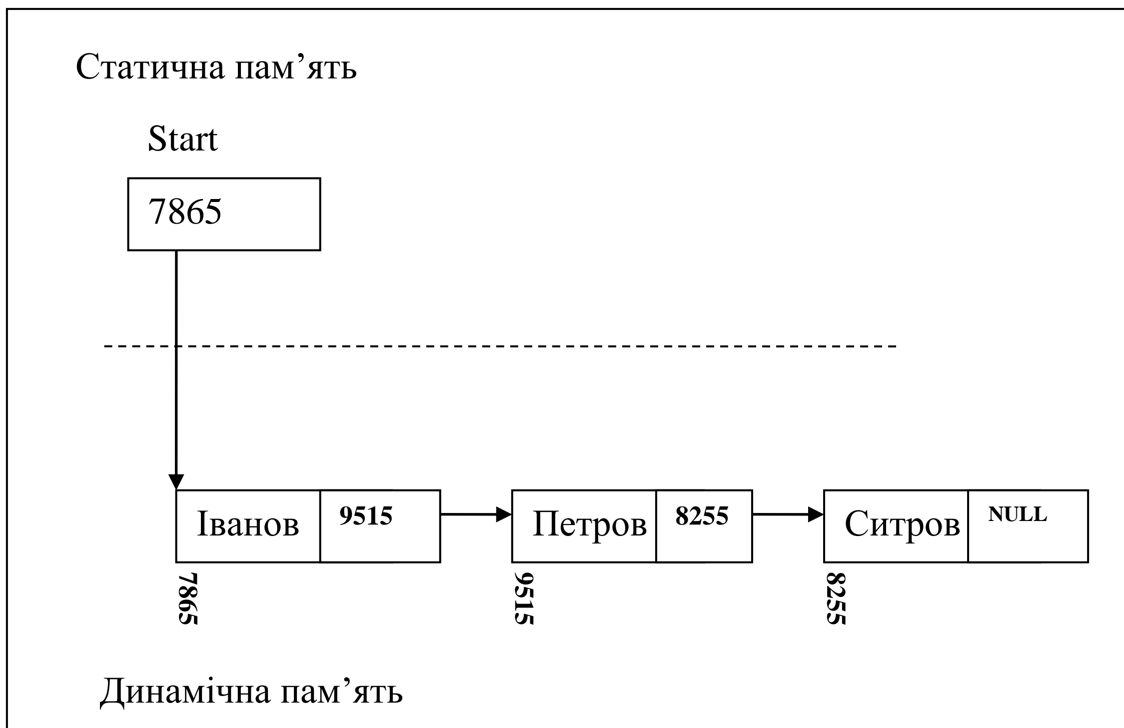


Рисунок 2.1 – Приклад зображення лінійного односпрямованого зв'язаного списку

Особливостями побудови облікової структури даних є:

1. Доступ до списку здійснюється через вказівник на його перший елемент. Зазвичай таку змінну-вказівник називають Start або Head;
2. Доступ до кожного наступного елементу списку можливий через поле-вказівник на наступний елемент;
3. В останньому елементі списку поле-вказівник на наступний елемент містить значення NULL.

2.1.1 Реалізація зв'язаного односпрямованого списку.

2.1.2.1 Основні операції реалізації списку

Для реалізації односпрямованого зв'язаного списку (надалі будемо говорити просто - список) необхідно забезпечити виконання наступних операцій:

1. Створення списку - ініціалізація списку;
2. Внесення елементу в список;
3. Видалення елемента зі списку;
4. Пошук елемента в списку;
5. Виведення списку на екран;

6. Видалення всього списку.

Перелік перерахованих операцій може бути розширено. Він залежить від завдання, яке вирішується на базі спискової структури.
Тепер уявімо деякі коментарі з приводу виконання кожної з перерахованих вище операцій.

Створення списку.

Ініціалізація списку є обов'язковою операцією. Ініціалізація списку полягає в присвоєнні значення NULL змінній-вказівнику на її початок.

Якщо список порожній, тобто в ньому немає жодного елемента, ця змінна повинна містити значення NULL. Якщо її не ініціалізувати таким значенням, в ній може знаходитися «сміття».

Внесення елементу в список.

Внесення елементів в список може відбуватися різними способами: їх може вводити користувач по одному в міру потреби; список може бути заповнений відразу декількома елементами, наприклад, при перенесенні в нього вмісту файлу або будь-якого списку даних на екрані.

У даній лабораторній роботі розглянемо ситуацію, коли новий елемент в список додається в міру необхідності, по одному. В такому випадку внесення елемента в список полягає у виклику підпрограми, яка отримує список і вказівник на новий елемент, і виробляє вставку елемента шляхом зв'язування його з елементами списку. Таке зв'язування полягає в зміні значень полів-вказівників на наступний у елементів, що вставляються, після яких здійснюється вставка в списку.

Підпрограма додавання елемента в список задає правило організації списку - кожен новий елемент буде весь час вставлятися чи в початок списку, чи в кінець, чи в відсортований список.

Видалення елемента зі списку.

Залежно від розв'язуваної задачі, видалення елемента зі списку може бути реалізовано по-різному. Тут може бути випадок, коли необхідно знайти елемент з певним значенням і видалити його. В такому випадку буде виконаний пошук елемента і видалення. Якщо завдання полягає у видаленні кількох елементів за певною ознакою, то пошук і видалення здійснюється в циклі проходу по списку.

Витяг елемента зі списку полягає в зміні значень полів зв'язку між елементами таким чином, щоб елемент, який видаляється, виявився поза списком, тобто не був пов'язаний ні з одним його елементів. А потім виконується звільнення пам'яті від цього елемента за допомогою оператора

delete.

Пошук елемента в списку.

Зазвичай, коли говорять про пошук в будь-якому списку, то мається на увазі рішення наступного завдання:

На вхід підпрограми надходить:
- спискова структура, в якій здійснюється пошук;
- «ключ пошуку».

Завдання полягає в тому, щоб знайти елемент, у якого певне поле рівне «ключу пошуку», і повернути «адресу» цього елемента. При пошуку в масиві під «адресою» розуміється індекс цього елемента, а в списковій структурі - вказівник на знайдений елемент.

Наприклад, якщо необхідно здійснити пошук елемента в списку студентів на прізвище «Кравченко», то ключем пошуку буде значення «Кравченко», а результатом пошуку буде вказваник, який містить адресу знайденого елемента списку.

Зазвичай, якщо пошук в списку не увінчався успіхом, вказівнику на знайдений елемент присвоюється значення NULL.

Виведення списку на екран.

Операція виведення списку на екран полягає в послідовному доступі до кожного елемента списку, починаючи з початку, і виводом значення його ключового поля або значень декількох його полів на екран.

Видалення всього списку.

При видаленні всього списку відбувається послідовне звільнення пам'яті, займаної кожним його елементом. В результаті виконання цієї операції список видаляється з пам'яті і вказівник на початок списку отримує знаечніе NULL.

2.1.2.2 Опис типів та оголошення змінних для побудови списку.

Оскільки список складається з елементів, які є структурами, то для його реалізації необхідно описати тип для структури. Кожна така структура містить крім інформаційних полів, ще й додаткове поле - вказівник на наступний елемент такого ж типу (зазвичай це поле називають «Next», що в перекладі з англійської означає «наступний»).

Приклад опису типів для списка студентів показано на рисунку 2.2:

```
Typedef
   struct Stud{

     string    fam;  /* поле для прізвища */
     int         bal;   /* поле для балу */
     Stud *    next;  /* додаткове поле-вказівник,
                              яке буде містити адресу наступного
                               елементу списку */

} TStud;
```

Рисунок 2.2 – Приклад опису типів для списку студентів (опис типу структури для елемента списку)

При такому описі типу для структури, його можна розмістити в окремому заголовку й використовувати в інших файлах програми, правильно підключивши даний заголовок.

Тип для вказівника на початок списку можна описувати при оголошенні змінної (див. нижче).

Для доступу до списку, як було зазначено раніше, використовується змінна, що містить адресу першого елемента списку. Зазвичай вона називається Start. Якщо для обробки списку використовуються кілька окремих функцій, то змінна-вказівник на початок списку повинна бути глобальною, для того, щоб не втрачати старт списку при виході з окремих підпрограм.

Якщо для списку описані типи, то для початку роботи з ним необхідно оголосити змінну, що містить адресу його першого елемента. Ця змінна буде типізованим вказівником на елемент списку. Приклад оголошення змінної-вказівника на початок списку наведено на малюнку 2.3.

```
typedef
   struct Stud{

     string    fam;  /* поле для прізвища */
     int         bal;   /* поле для бала*/
     Stud *    next;  /* додаткове поле-вказівник,
                              яке буде містити адресу
                             наступного елемента списку*/

} TStud;

 TStud  * Start;
```

Рисунок 2.3 – Приклад опису типів і оголошення змінної-вказівника на список студентів

Після того, як типи описані і оголошена змінна Start, можна здійснювати операції зі списком.

Слід звернути увагу на те, що, перш за все, необхідно виконати операцію

ініціалізації списку.

2.1.2.3 Ініціалізація списку

Підпрограма ініціалізації списку - це найперша підпрограма, яку необхідно викликати на початку роботи зі списком. Її завдання полягає в занесенні значення NULL на початок списку. Для програми це буде означати, що список порожній. Невиконання даної операції може спричинити за собою безліч помилок, адже, якщо вказівник містить «сміття», то це «сміття» може бути сприйняте системою як число - якась адреса. В результаті в програми буде спроба програми зробити звернення за цією адресою і здійснити роботу з даними, які насправді не належать списку, але сприймаються як його елементи.

Функція ініціалізації списку студентів буде виглядати наступним чином (див. рисунок 2.4):

```
void  initList (TStud * &Start)
{
  Start=NULL;
}
```

Рисунок 2.4 – Приклад функції, що ініціалізує список

При виконанні процедури ініціалізації списку розміщення змінної в пам'яті програми буде мати вигляд, показаний на рисунку 2.5:

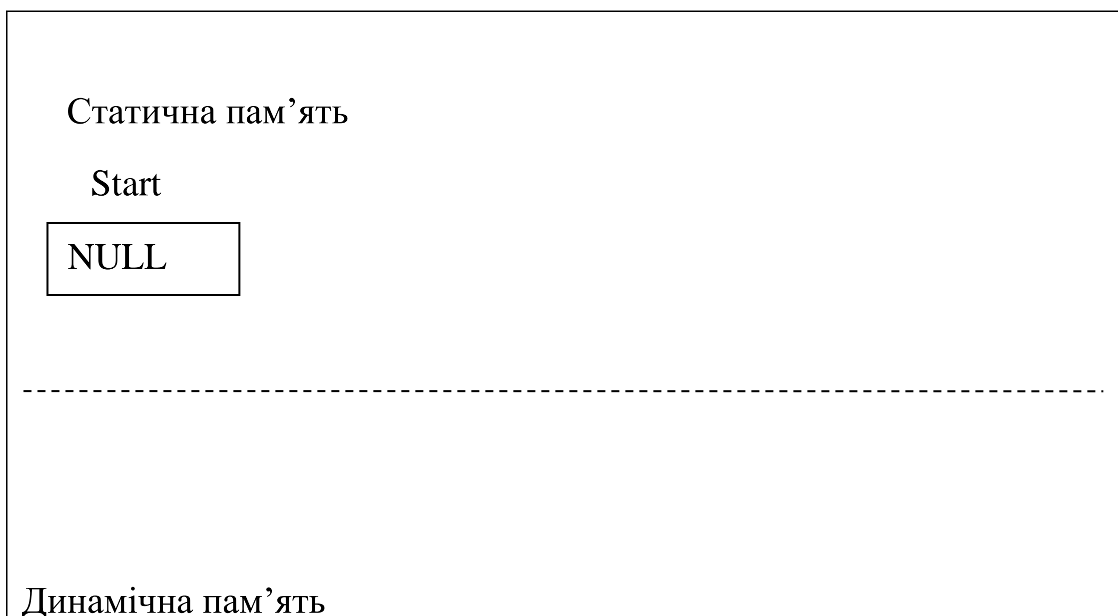


Рисунок 2.5 – Розміщення змінних програми в пам'яті після виконання процедури ініціалізації списку

2.1.2.4 Функції додавання елементів до списку

Будь-яка з функцій додавання нового елемента до списоку отримує вказівник на початок списку та новий елемент, який вже знаходиться в

динамічній пам'яті з заповненими полями. До виклику процедури додавання необхідно створити новий елемент в динамічної пам'яті і заповнити його поля певними значеннями.

Як було зазначено вище, можна створити функцію додавання, яка кожен новий елемент списку буде вставляти в його початок чи в кінець, або ж в середину списку згідно якійсь з умов вставки. У будь-якому з цих випадків список на початку буде порожнім і після вставки нового елемента адреса початку списку зміниться. Отже, будь-яка з функцій додавання елемента в список отримує параметр Start по посиланню.

2.1.2.4.1 Функція додавання елемента в початок списку

Функція додавання елемента в початок списку виконує наступне:

1. У поле Next нового елемента заносить адресу колишнього першого елемента списку, яку бере у змінної Start;
2. Змінна Start привласнює адресу нового елемента списку, так як тепер він буде першим.

Слід зазначити, що в будь-якому випадку на початковому етапі роботи зі списком, він - порожній. Функція додавання елемента в початок списку виконає додавання нового елемента і в порожній список. В такому випадку отримаємо список, що складається з одного елемента.

Функція додавання елемента в початок списку показана на рисунку 2.6.

```
void addToStart(TStud * &Start, TStud * PNew)
{

 PNew->next=Start; // В новий елемент в поле Next заноситься
              //адреса минулого першого елемента списку Start=PNew;     // Змінною Start присвоюється адреса
нового
          // элемента списку, так як тепер він буде першим
}
```

Рисунок 2.6 – Процедура додавання елементу в початок списку

У момент виклику процедури додавання розташування даних програми в пам'яті комп'ютера має вигляд, показаний на рисунку 2.7.

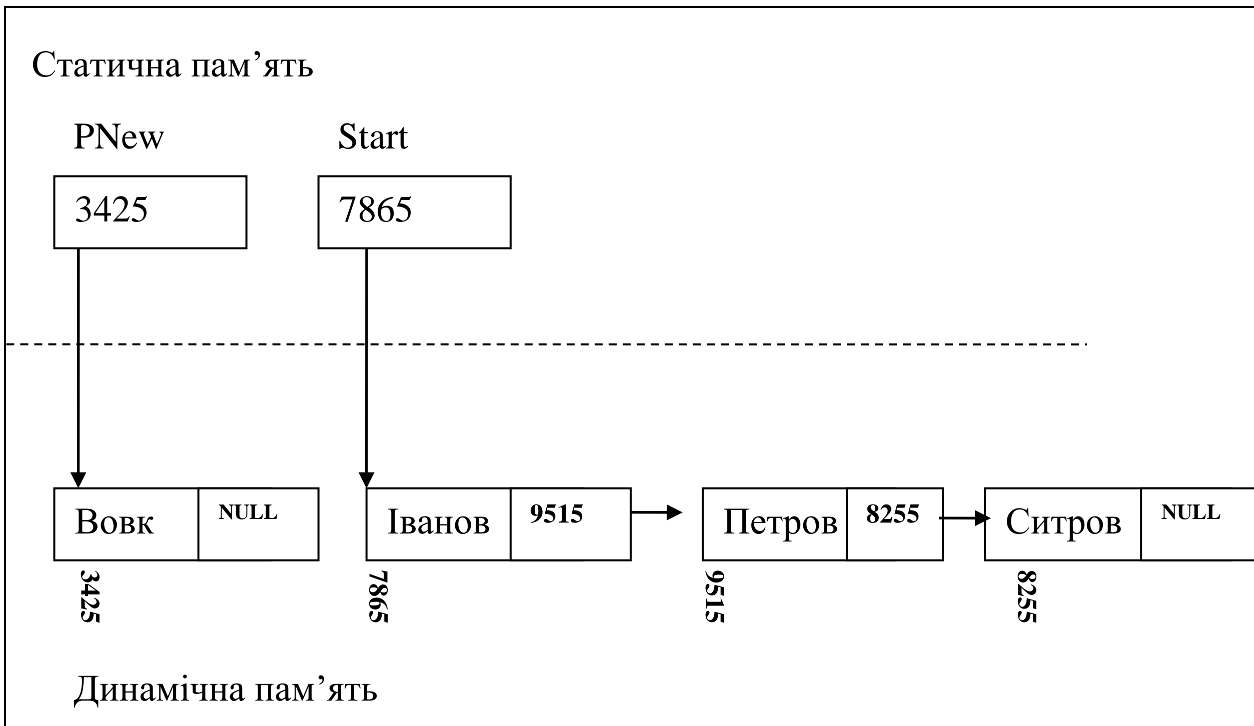


Рисунок 2.7 – Розташування даних програми в момент виклику функції, що додає новий елемент до список (в даному випадку список вже не порожній)

У момент, коли робота функції закінчується, закінчення роботи функції додавання розташування даних програми в пам'яті комп'ютера має вигляд, показаний на рисунку 2.8.

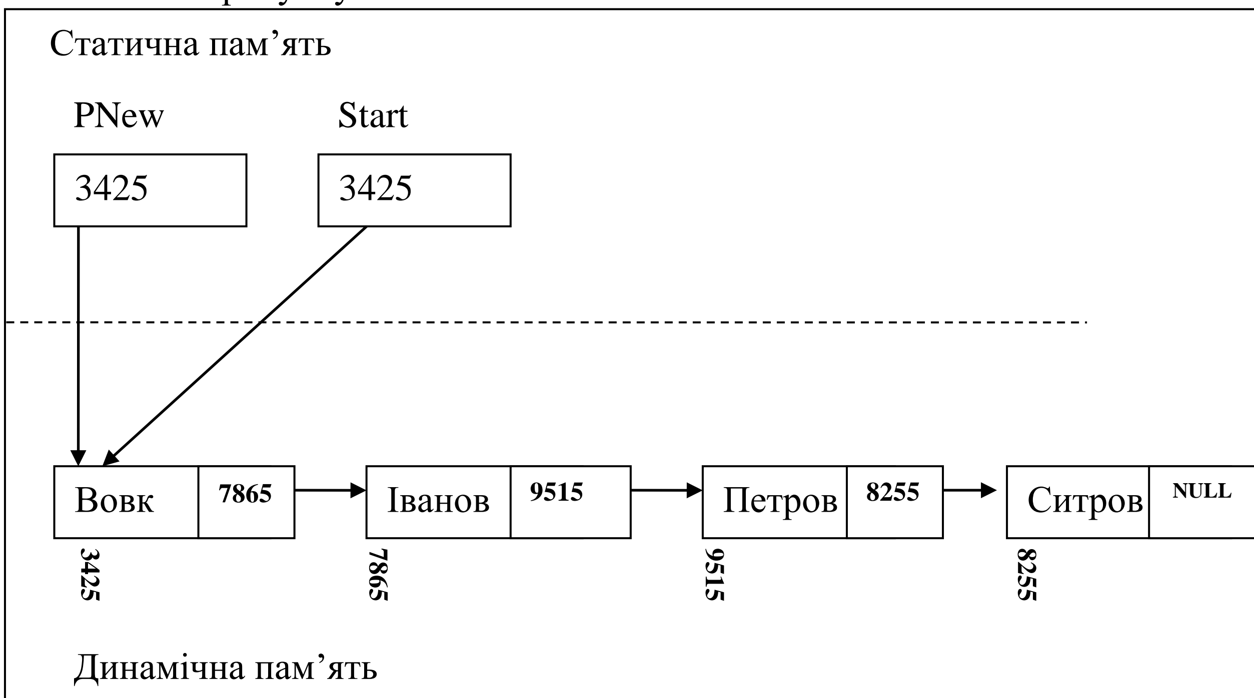


Рисунок 2.8 –Розташування даних програми в момент завершення роботи функції, що додає новий елемент в початок списку

2.1.2.4.2 Функція додавання елементу в кінець списку

Для того щоб додати новий елемент в кінець списку, необхідно отримати доступ до його останнього елемента, в поле Next якого занести адресу елементу, що додається.

До виклику даної функції, коли створюється новий елемент в динамічної пам'яті, необхідно в його поле Next занести значення NULL, так як в будь-якому випадку він буде останнім в списку.

Алгоритм функції, що додає елемент в кінець списку:

1. Перевірка списку на порожнечу. Якщо список порожній, то новий елемент стає першим – змінній Start присвоюємо адресу нового елемента і здійснюємо вихід з функції;
2. Якщо список ще порожній, то робочому вказівнику WP (work pointer, можна дати йому будь-яку іншу назву) присвоюємо адресу першого елемента списку. Таким чином робимо перший елемент списку поточним;
3. Організовуємо цикл проходу по списку з метою отримання доступу до його останнього елемента. Цикл виконується до тих пір, поки у поточного елемента поле Next не стане NULL, що означатиме, що поточний елемент - це останній елемент списку. У циклі здійснюється взяття значення в поле Next адреси наступного елемента у поточного, занесення адреси наступного елемента в змінну WP. Таким чином, забезпечується послідовний прохід по списку.
4. Після виходу з циклу змінна WP містить адресу останнього елемента списку. Тепер «прив'язуємо» новий елемент до останнього елемента списку шляхом занесення адреси нового елемента (який приходить в процедуру, як параметр) в поле Next останнього елемента списку, звертаючись до нього за допомогою вказівника WP.

Функція додавання елемента в кінець списку показана на рисунку 2.9.

```
void  addToEnd(TStud * &Start, TStud * PNew)
{
  TStud * WP;
  PNew->next=NULL;  // новий елемент в будь-якому випадку буде останнім  if
(Start == NULL) //перевірка списку на порожнечу     {
      Start=PNew;  //якщо список порожній, то елемент стає
//першим та останнім елементом списку
return;    //Вихід з процедури, елемент додано до списку      }
  WP=Start; // ініціалізація вказівника на поточний елемент
              // теперь поточним є перший
      // цикл проходу по списку до останнього елемента
    while(WP->next!=NULL)  // поки поточний – не останній           {
           WP=WP->next;  // в WP заносимо адресу наступного елементу
         }
       // тепер WP вміщує адресу останнього елемента списку
       // «прив'зуємо» новий елемент до останнього
   WP->next=PNew;
      // елемент включено до списку
}
```

Рисунок 2.9 – Функція додавання елемента в кінець списку

У момент виклику функції, що додає новий елемент до списку, розташування даних програми в пам'яті комп'ютера має вигляд, показаний на рисунку 2.10.

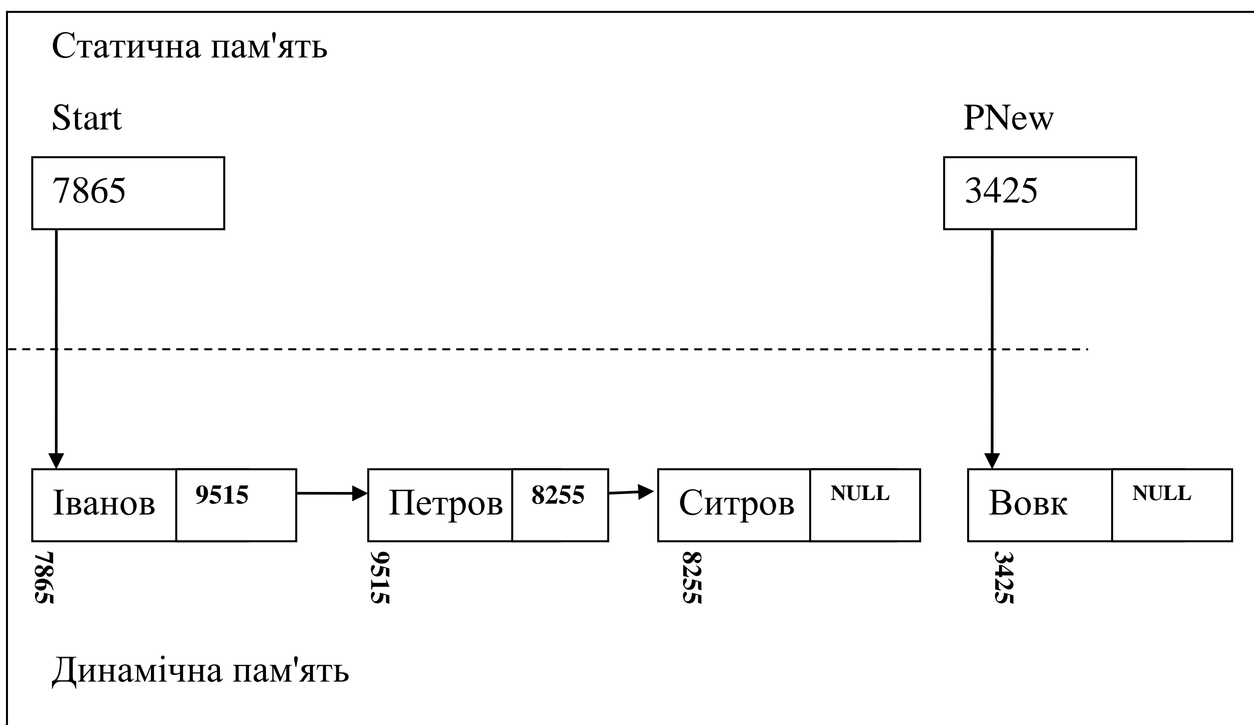


Рисунок 2.10 – Розташування даних програми в момент виклику функції додавання нового елемента в список

Момент виходу з циклу, коли робочий вказівник WP містить адресу останнього елемента списку, зображений на рисунку 2.11.

Статична пам'ять

Start WP PNew

7865 8255 3425

Іванов 9515 Петров 8255 Ситров NULL Вовк NULL

7865 9515 8255 3425

Динамічна пам'ять

Рисунок 2.11 – Розташування даних програми в момент закінчення циклу проходу по списку

Коли є вказівник, що містить адресу останнього елемента списку, ми можемо до нього «прив'язати» новий елемент. Результат роботи функції додавання елемента в кінець списку показано на рисунку 2.12.

Статична пам'ять

Start WP PNew

7865 8255 3425

Іванов 9515 Петров 8255 Ситров 3425 Вовк NULL

7865 9515 8255 3425

Динамічна пам'ять

Рисунок 2.12 – Розташування даних програми в момент закінчення її роботи - елемент додано в кінець списку

При виконанні функції додавання елемента в кінець списку в циклі здійснювався прохід за списком за допомогою одного допоміжного вказівника WP.

2.1.2.4.3 Функція додавання елемента в відсортований список

Функція додавання елемента в відсортований список використовує два допоміжних вказівника при проходженні по списку.
Використання двох вказівників для проходження по списку є темою наступної частини лабораторної роботи, що буде розглянута пізніше.

2.1.2.5 Процедури обробки всіх елементів списку

Часто доводиться вирішувати завдання обробки всіх елементів списку. Прикладами таких завдань можуть бути:
- виведення всіх елементів списку на екран;
- зміна значень всіх елементів списку певним чином;
- знаходження суми всіх елементів списку і т.д.

Підпрограми, які обробляють всі елементи списку, обов'язково містять цикл проходу по списку.

Відмінність такого циклу від циклу функцій додавання елемента в кінець списку полягає в тому, що в функції додавання останній елемент списку не обробляється і цикл закінчується, коли робочий вказівник містить адресу останнього елемента списку, тоді як цикл обробки всіх елементів списку закінчується тоді, коли робочий вказівник взяв у останнього елемента списку адресу наступного і отримав значення NULL.

Загальний вигляд циклу у будь-який функції обробки всіх елементів списку буде мати вигляд, показаний на рисунку 2.13

```
wp = Start; // ініціалізація вказівника на поточний елемент
                    // тепер поточним є перший

    // цикл проходу по списку задля обробки всіх його елементів

  while(wp !=NULL)  //поки поточний не отримав значення NULL
       {

            …  // тут знаходяться інструкції по обробці поточного
                 // елементу, до якого отримуємо доступ через вказівник wp

           wp = wp->next;  // в wp заносимо адресу наступного елементу

       }
```

Рисунок 2.13 - Організація циклу проходу по списку для обробки всіх його елементів

2.1.2.6 Функція виведення всіх елементів списку на екран

Для виведення всіх елементів списку на екран будемо використовувати консоль QT Creator.

Слід зазначити, що при виведенні списку на екран адреса елементів списку не змінюється, тому старт списку в процедуру надсилається за значенням.

Процедура виведення прізвищ списку студентів на екран показана на рисунку 2.14.

```
void showList(TStud * Start)
{
 TStud * wp;
 wp=Start; //ініціалізуємо рабочий вказівник значением Start
    // За допомогою wp послідовно отримуємо доступ
    // до кажного элемента списку
  while(wp != NULL)  // Доки wp не отримав значення null
    {
    cout<<wp->fam<<endl; //wp містить адресу поточного  элементу
                   // через нього отримуємо доступ до fam
    cout<<endl;
    wp=wp->next;  // wp отримує адресу наступного элементу
    }
}
```

Рисунок 2.14 – Функція виведення списку прізвища студентів на екран в консоль

У разі, коли змінна Start передається за значенням, а не за посиланням, можна використовувати замість робочого вказівника WP саму змінну Start для проходу по списку.

2.1.2.7 Функція видалення списку

Функція видалення списку послідовно отримує доступ до кожного його елемента і звільняє пам'ять, займану ним. В результаті роботи функції, Start списку отримує значення NULL, що означає - список порожній.

У функції використовується додатковий вказівник DelP, для того, щоб запам'ятати адресу елементу, що видаляється, і послати в оператор для звільнення пам'яті delete. Start буде містити в циклі адресу наступного елемента, що підлягає видаленню.

Текст зображення без списку наведено на рисунку 2.15

```
void delList(TStud * &Start){
 TStud * DelP;
 while (Start != NULL)  // доки не пройшли увесь список
   { DelP = Start; // DelP отримує адресу елемента,що видаляється
     Start = Start->next;  // в Start заносимо адресу наступного
           //элемента. Це робиться з ціллю
           // не втратити інші элементи списку
           // Таким чином, здобувається перший
```

```
        //элемент списка. І першим стає наступний.
    delete DelP; // Звільняємо пам'ять від елемента, що видаляємо
   }
}
```

Рисунок 2.15 – Функція видалення списку

2.1 Приклад створення проекту для обробки списку

2.1.2 Задача, вирішувана проектом

Завдання полягає в тому, щоб створити список студентів, кожен елемент якого містить наступну інформацію: прізвище студента, середній бал.

Забезпечити заповнення даних про кожного студента шляхом вибору операції «Додати елемент в початок списку» і введення даних про новий елемент. Висновок вихідного списку провести на екран в консоль.

Обчислити кількість студентів, які мають оцінку нижче 60 балів і вивести результат на екран.

2.1.3 Інтерфейс проекту

Інтерфейс проекту наведено на рисунку 2.16.

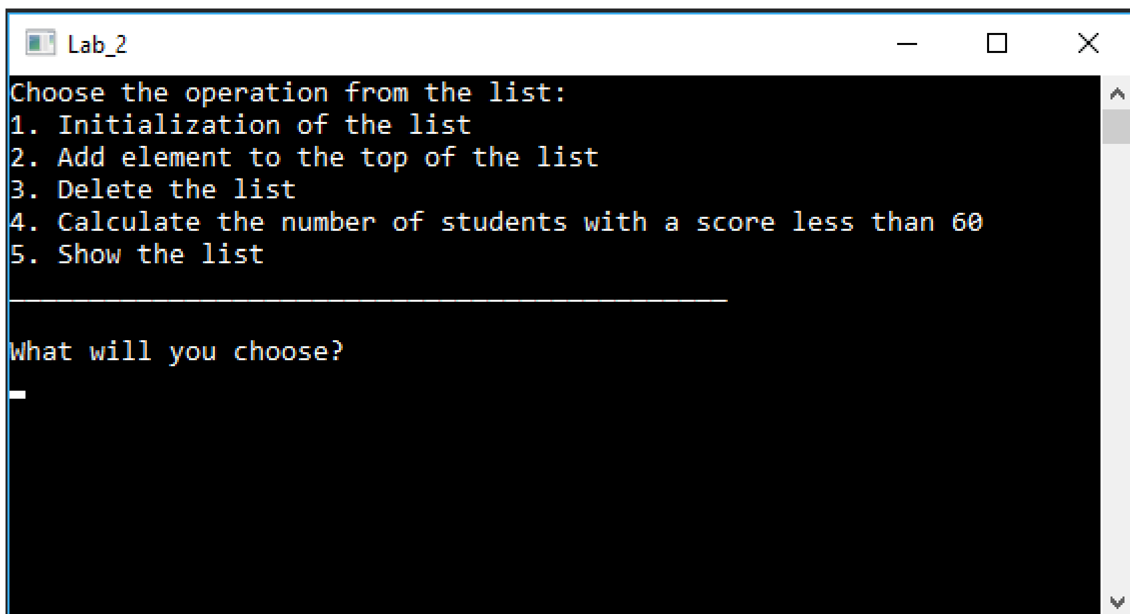


Рисунок 2.16 – Інтерфейс проекту

Для введення даних про нового студента при виборі операції «Додавання елемента в початок списку» на екрані з'являється пропозиція ввести прізвище та бал. (див. рисунок 2.17)

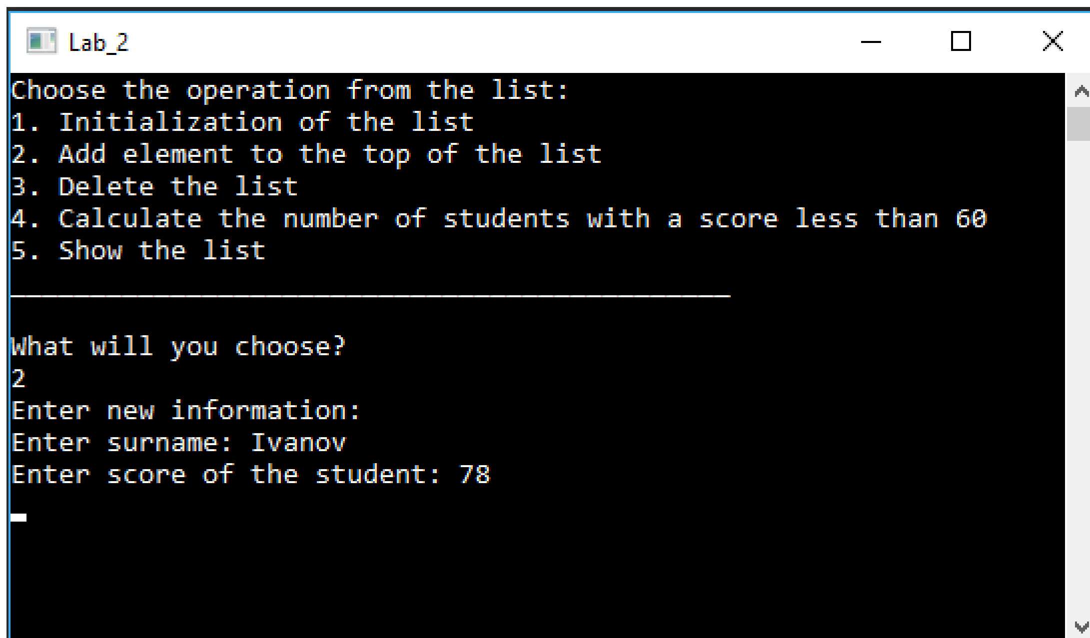


Рисунок 2.17 – Зображення консолі в момент вводу данних про нового студента

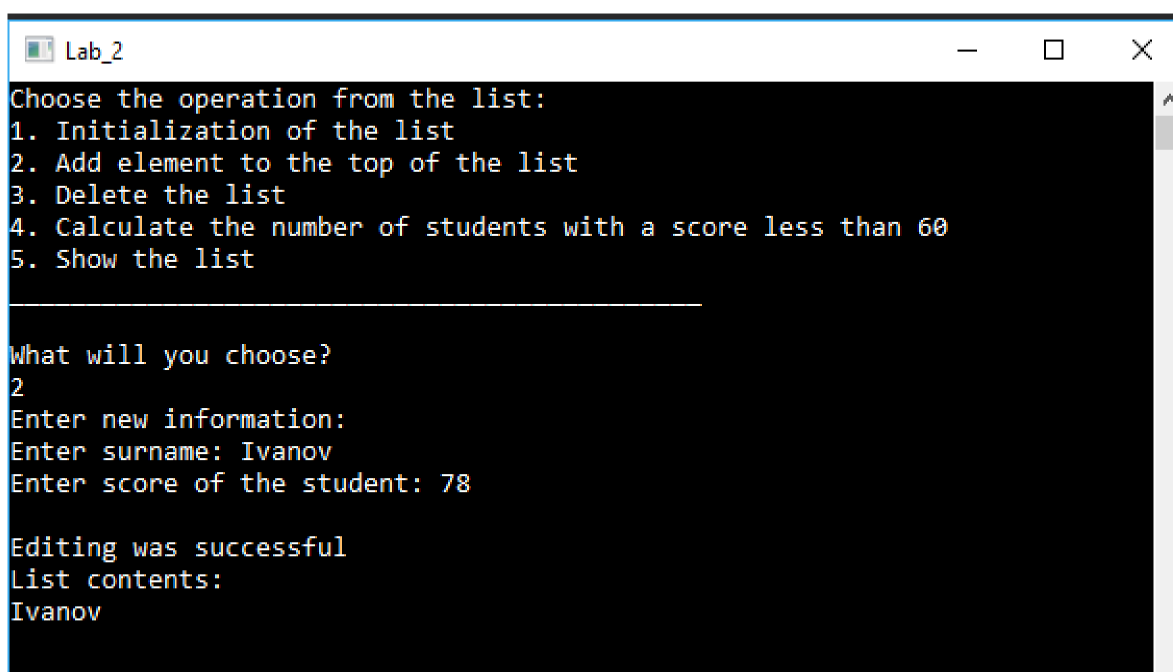


Рисунок 2.18 – Зображення консолі після виповнення додавання елементу до списку та виводу на екран

2.1.4 Модуль типів та глобальних змінних проекту

Для опису типів списку створимо модуль "Types.h". Описані типи будуть використовуватися в головному модулі і в модулі функцій зі списком,

тому їх необхідно описати в заголовки.

Порядок створення заголовного модуля наступний:
1. У середовищі QtCreator створити новий проект типу «Консольний додаток Qt». У ньому автоматично з'явиться файл «main.cpp» - це буде головний файл проекту.
2. Вибрати в головному меню Файл -> Новий файл або проект і вибрати тип файлу «C ++», заголовки, як показано на рисунку 2.19

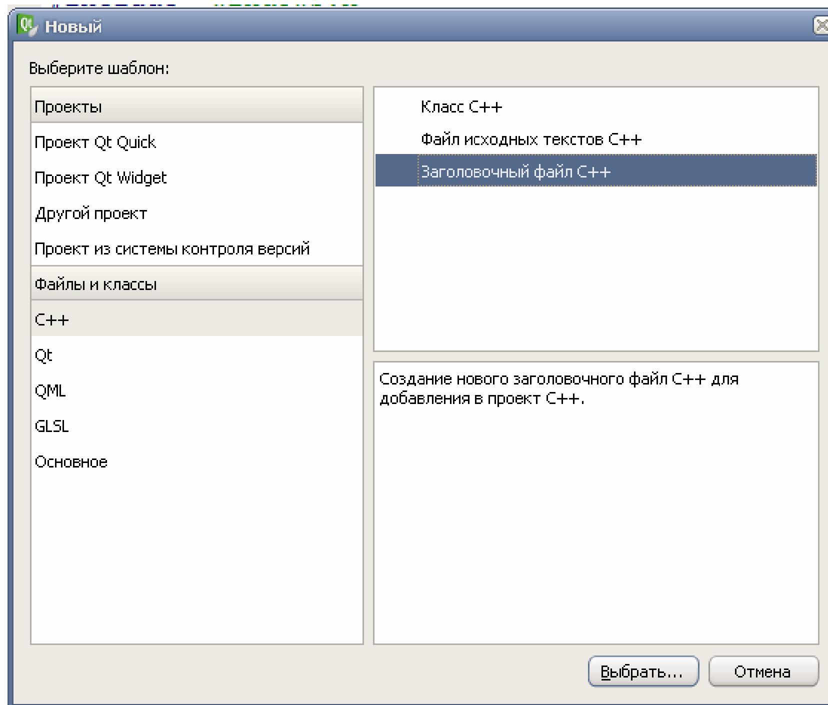


Рисунок 2.19 – Створення заголовочного файлу

У наступному вікні задати ім'я файлу - "types", як показано на рисунку 2.20.

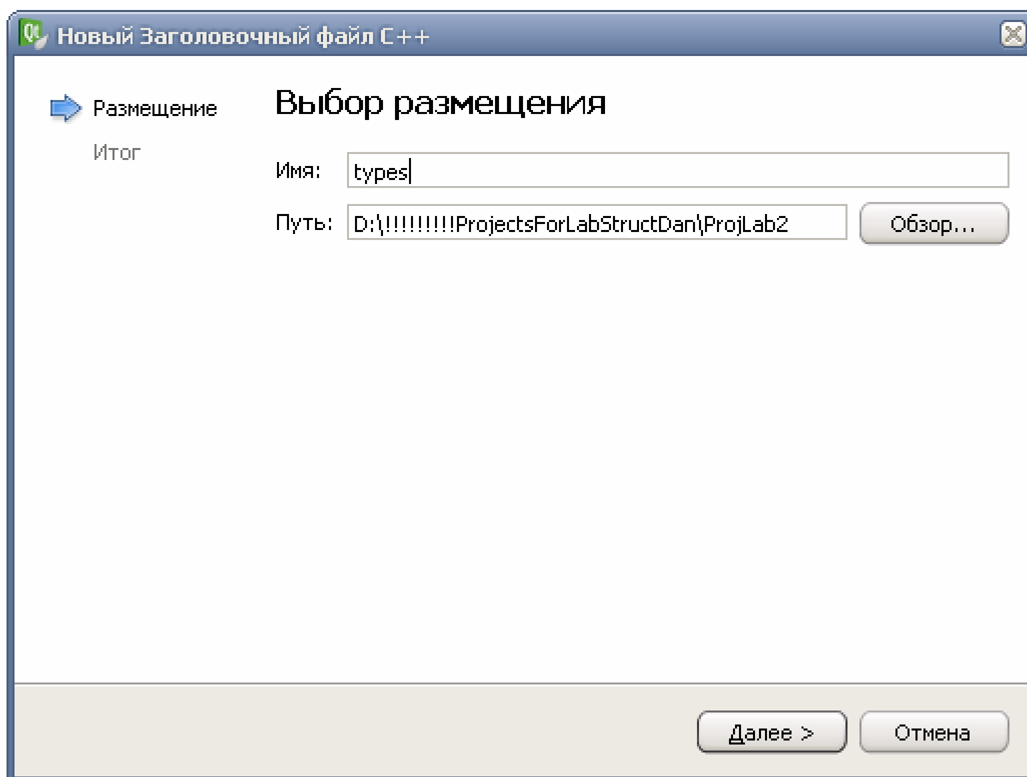


Рисунок 2.20 – Задання імені для заголовочного файлу

Вміст модуля "types.h" показано на рисунку 2.21.

```
#ifndef TYPES_H
#define TYPES_H
#include <string>

using namespace std;


typedef
    struct Stud{
      string   fam;
      int      bal;
      Stud *   next;

   } TStud;


#endif // TYPES_H
```

Рисунок 2.21 – Вміст модуля "types.h"

У модулі типів слід оголосити глобальну змінну Start, яка буде містити адресу початку списку.

Модуль типів слід підключити до модулів UnitMain, UnitAdd, UnitProc.

2.1.5 Модуль функцій проекту

Для реалізації проекту необхідні наступні функції:
- функція ініціалізації списку;
- функція видалення списку;
- функція виведення списку на екран;
- функція додавання елемента в початок або кінець списку;
- функція обчислення кількості студентів, з балом меншим 60.

Всі ці функції розміщуємо в окремому модулі "functions.cpp", який створюємо так само, як і заголовки в попередньому розділі. Лише з тією відмінністю, що вибираємо для файлу тип «.cpp». Заголовки створених функцій не забуваємо перенести в заголовки з такою ж назвою і розширенням «h» - "functions.h". Для того, щоб ці функції були доступні іншим модулям, необхідно заголовки модуля «functions.h» підключити до тих модулів, де використовуються ці функції.

У файлах «functions» використовується тип, описаний в модулі «types.h», тому не забудьте підключити його.

Вміст модуля «functions.cpp» представлено на рисунку 2.22.

Вміст заголовку модуля «functions.h» з заголовками функцій наведено на рисунку 2.23

```
#include "types.h"
#include <iostream>



void  initList (TStud * &Start)
{
   Start=NULL;
}






void addToStart(TStud * &Start, TStud * PNew)
{
    PNew->next=Start;

    Start=PNew;

}
```

```
void  addToEnd(TStud * &Start, TStud * PNew)
{
  TStud * WP;
  PNew->next=NULL;  // обавляемый элемент всегда будет последним
  if (Start == NULL) // если список пуст
     {
      Start=PNew;  // то новый элемент становится первым
                  // Start получает адрес нового элемента
                  // Элемент добавлен в список
      return;
     }
   WP=Start; // вспомогательному указателю wp
             // присваиваем адрес первого элемента списка
   // Проходим по списку до последнего элемента
   while(WP->next!=NULL)  // если элемент не последний
     {
       WP=WP->next;  // то WP получает адрес следующего элемента
     }
   // В данный момент wp содержит адрес последнего элемента списка
   // в поле next последнего элемента заносим адрес нового
   WP->next=PNew;
      //элемент добавлен в список
}
```

```
void showList(TStud * Start)
{
  TStud * wp=Start; //инициализируем рабочий указатель
    //значением Start. С помощью wp последовательно получаем доступ
    // к каждому элементу списка
   while(wp != NULL)  // Пока wp не получил значение null
       {
         cout<<wp->fam<<endl; //wp содержит адрес текущего элемент
                              // через него получаем доступ к полю fam
         cout<<endl;
         wp=wp->next;  // wp получает адрес следующего элемента
        }
}
//-----------------------------------------------------------------
  /* Удаление всего списка
     Входной параметр Start - указатель на начало списка,
                              возвращается со значением NULL */
void delList(TStud * &Start){
  TStud * DelP;
  while (Start != NULL)  // пока не прошли весь список
     { DelP = Start; // DelP получает адрес удаляемого элемента
       Start = Start->next;  // в Start заносим адрес следующего
            //элемента. Это делается с целью не потерять остальные
            // элементы списка. Таким образом, извлекается первый
            //элемент списка. И первым становится следующий.
      delete DelP; // Освобождаем  память от удаляемого элемента
     }
}
```

```
int countLess60(TStud * Start){
    TStud * wp;
    int cnt=0; // инициализация перед накоплением количества
    wp=Start; //инициализируем рабочий указатель значением Start


        // С помощью wp последовательно получаем доступ
        // к каждому элементу списка

     while(wp != NULL)  // Пока wp не получил значение null
        {
        if (wp->bal < 60) // если бал <60 увеличиваем счетчик
                          //студентов на 1
           {cnt++;}
        wp=wp->next;  // wp получает адрес следующего элемента
        }
    return cnt;
}
```

Рисунок 2.22 – Вміст модуля функцій “functions.cpp”

Вміст заголовку модуля «functions.h» з заголовками функцій наведено на рисунку 2.23

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

#include "types.h"

    void initList (TStud * &Start);
    void addToStart(TStud * &Start, TStud * PNew);
    void  addToEnd(TStud * &Start, TStud * PNew);
    void showList(TStud * Start);
    void delList(TStud * &Start);
    int countLess60(TStud * Start);

#endif // FUNCTIONS_H
```

Рисунок 2.23 – Вміст заголовочного модуля функцій “functions.h”

Слід звернути увагу, що в цьому модулі необхідно підключити модуль "types.h".
Даний модуль слід підключити до модуля "main.cpp".
Вміст заголовку модуля «functions.h» з заголовками функцій наведено на рисунку 2.23

2.1.6 Основний модуль проекту

Для правильної організації роботи проекту необхідно до головного модуля підключити модулі "functions.h", "types.h", "windows.h", <iostream>.

Головний модуль містить оголошення глобальної змінної Start і виконує вивід на екран переліку можливих операцій зі списком, а також обробляє вибір кожної операції. Крім того, цей модуль містить додаткову функцію add (), яка забезпечує обробку операції додавання елемента в список.

Вміст головного модуля «main.cpp» показано на рисунку 2.24.

```
#include <QtCore/QCoreApplication>
#include "functions.h"
#include "types.h"
#include "windows.h"
#include <iostream>

TStud * Start;  //оголошення змінної – вказівника на початок списку

void add(TStud* &Start);

int main(int argc, char *argv[])
{
   QCoreApplication a(argc, argv);
   char ch;
   do {
      system("cls");
      SetConsoleTitleA("Лабораторна №2");
      cout<<"Виберіть операцію зі списком: \n";
      cout<<endl;
      cout<<"1.Ініціалізація списку \n";
      cout<<endl;
      cout<<"2.Додавання елементів в початок списку \n";
      cout<<endl;
      cout<<"3.Видалення всього списку \n";
      cout<<endl;
      cout<<"4.Вирахувати квлькість студентів з балом нижчим ніж   60\n";
      cout<<endl;
      cout<<"5.Виведення списку на екран \n";
      cout<<"--------------------------------\n";
      cout<<endl;
      cout<<"Виберіть операцию для списку \n";
      cin>>ch;
      switch(ch){
         case '1': {initList(Start);
            cout<<endl;
            cout<<"Ініціалізація пройшла успішно\n";
            cout<<endl;
            }
```

```
            break;
          case '2': add(Start); break;
          case '3': {delList(Start);
                cout<<endl;
                cout<<"Видалення пройшло успішно \n";
                cout<<endl;
                }
                break;
         case '4': {int count=countLess60(Start);
              cout<<endl;
              out<<"Кількість двіяшників - "<<count;
              cout<<endl;
              }
              break;
         case '5': showList(Start);
                   break;
       }
       cout<<"\nДля продовження натисніть *";
       cin>>ch;
   }while(true);
   return a.exec();
}
    // Фунуція виповнення операції додавання
void add(TStud* &Start){
   TStud * PNew = new TStud; /*Запитуємо місце для нового елемента списку
*/
   /*Ввід інформації про новий елемент списку та заповнення його полів */
   cout<<"\N ВВІД ІНФОРМАЦІЇ ПРО НОВИЙ ЕЛЕМЕНТ: ";
   cout<<endl;
   cout<<"\nВведіть прізвище: ";
   cin>>PNew->fam;
   cout<<"\nВведіть бал: ";
   cin>>PNew->bal;
   cout<<"\n";
   addToStart(Start, PNew);
   cout<<"\nДодавання пройшло успішно! ";
   cout<<"\n";
   cout<<"\NЗМІСТ СПИСКУ:\n";
   cout<<"\n";
   showList(Start);
}
```

Рисунок 2.24 – Вміст головного модуля “main.cpp”

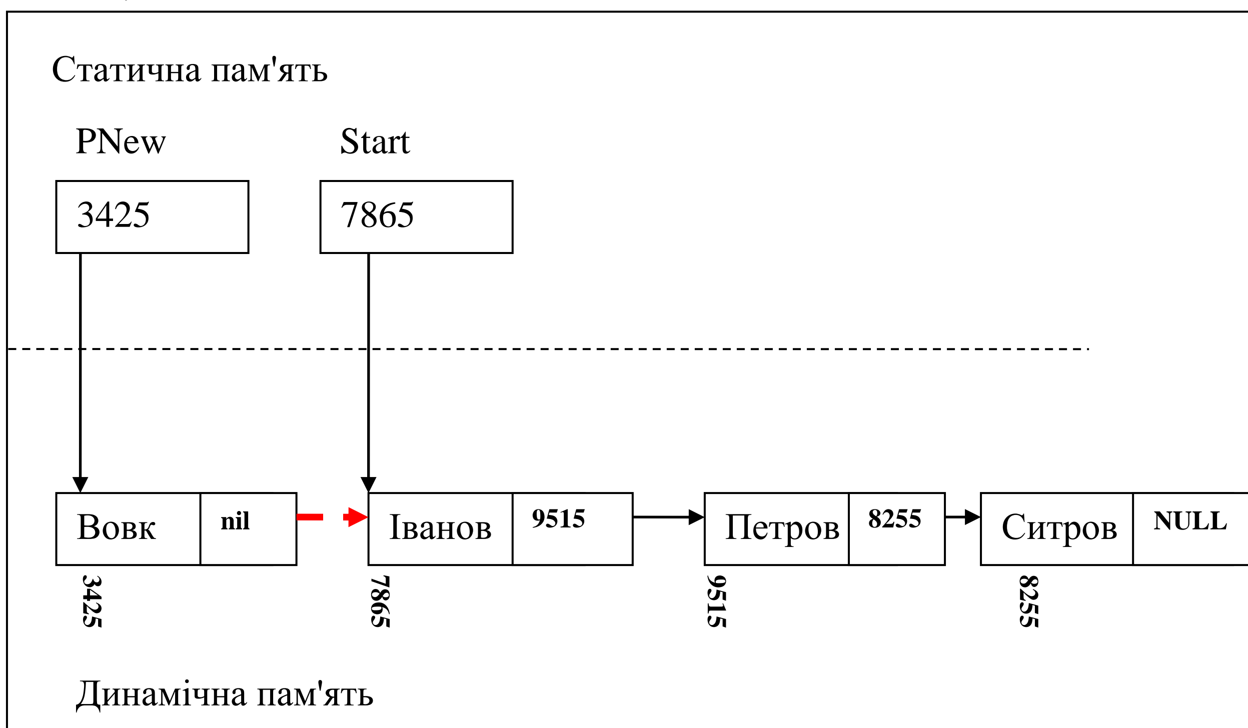
Перед додаванням елементів в список ОБОВ'ЯЗКОВО потрібно його спочатку ініціалізувати. Деякі системи автоматично ініціалізують глобальні вказівники значенням NULL.

2.3 Завдання для отримання навичок «зв'язування» елементів в списку

Як завдання для засвоєння матеріалу виконайте наступне (без програмної реалізації): напишіть інструкції, які забезпечать включення нового елемента до списоку, як показано пунктирною червоною стрілкою.

Можете використовувати ті змінні, які показані на рисунках.

1.



2.

Статична пам’ять

Start WP PNew

7865 8255 3425

Іванов 9515 Петров 8255 Ситров nil Вовк NULL

7865 9515 8255 3425

Динамічна пам’ять

3.

Статична пам’ять

Start PP PNew WP

7865 8255 3425 8255

Вовк nil

Іванов 9515 Петров 8255 Ситров NULL

7865 9515 8255

Динамічна пам’ять

2.4 Завдання для виповнення проекту

Створіть проект, який забезпечує ініціалізацію, створення, виведення на екран і видалення списка, як показано в прикладі п.2.2.

Крім того, реалізуйте завдання зі списком згідно з варіантом, наведеним в таблиці 2.1.

Таблиця 2.1 - Завдання для виконання проекту

Варіант	Вміст элемента списку	Реалізувати операцію зі списком:
0	Ціле число	1. Накопичити суму елементів списку 2.Парні елементи вивести на екран
1	- номер аудиторії - кафедра - кількість комп’ютерів	1. Обчислити кількість аудиторій кафедри ІКС 2. Вивести на екран аудиторії, де кількість комп'ютерів > 10
2	- прізвище спортсмена - розряд - вид спорту	1. Вивести на екран спортсменів, що мають перший розряд 2. Обчислити кількість спортсменів - футболістів
3	- назва книги - ціна - тираж	1. Для всіх елементів списку збільшити тираж на 2000 2. Вивести на екран назву книги і тираж, де ціна менше 100 грн.
4	- назва страви - кількість калорій - країна	1. Вивести на екран назви страв, де країна - Україна 2. Обчислити кількість страв, з числом калорій менше 300

5	- назва магазину - кількість працівників	1. Обчислити загальну кількість працівників по всіх магазинах 2. Вивести на екран назви магазинів, де кількість працівників менше 30
6	Ціле число	1.Знайти мінімальний елемент в списку і вивести його значення на екран 2. Вивести на екран числа, кратні 3-м
7	рядок	1.Знайти рядок «Машина» і вивести повідомлення про наявність такого рядка в списку 2. Вивести на екран довжину кожного рядка списку
8	- номер школи - директор - кількість вчителів	1. Якщо в школі кількість вчителів менше 100, то їх кількість збільшити на 10, в іншому разі ніяких змін не робити 2. Вивести на екран прізвища директорів шкіл, де кількість вчителів більше 200.
9	Ціле число	1.Зменьшіть значення кожного елемента списку на 20. 2.Вичісліть кількість негативних елементів списку.

2.5 Тестування проекту

Виконайте тестування проекту наступним чином:

1. У таблиці розмістіть значення елементів, доданих до списку.
2. Виконайте завдання 1, згідно варіанту, вручну і наведіть його результат в звіті;
3. Виконайте завдання 1 за допомогою проекту, зображення інтерфейсу з результатом виконання помістіть в звіт.
4. Аналогічно виконайте тестування другого завдання.

2.6 Зміст звіту

- 1. Найменування роботи.
- 2. Мета роботи.
- 3. Результат виконання завдання по «зв'язування» елемента в список.
- 4. Зображення інтерфейсу проекту.
- 5. Текст програми.
- 6. Результати тестування проекту.
- 7. Висновки про етапи створення і тестування проекту.

Лабораторна работа № 3

Двоспрямовані (Двонаправлені) списки

Мета роботи:
- ознайомитись з поняттям «двоспрямований список»;
- отримати практичні навички з написання процедур обробки двонаправлених списків;
- створити проект в системі програмування Qt Creator, в якому здійснюється робота з двоспрямованим списком.

3.1 Теоретичні відомості

Двоспрямованим називається список, кожний елемент якого містить два вказівники на наступний і попередній елементи списку (рисунок 3.1). наприклад:

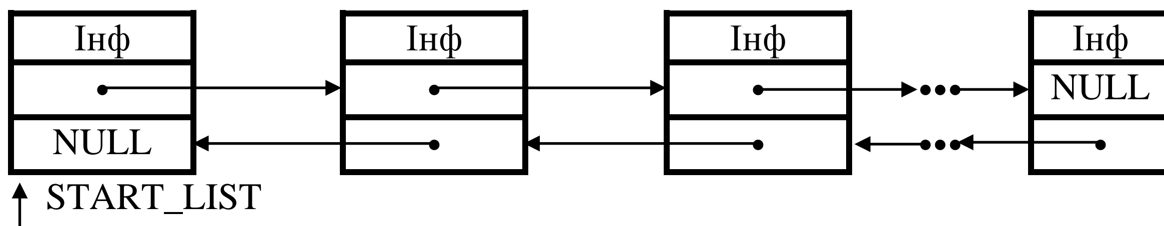


Рисунок 3.1 – Структура двоспрямованого списку

Двоспрямований список відрізняється двома основними перевагами.

По-перше, список може розглядатись в двох напрямках. Це не тільки спрощує сортування списку, але також дозволяє користувачам розглядати елементи в обох напрямках.

По-друге, список при порушенні одного зі зв'язків може бути відновлений за іншим зв'язком. Ця властивість є доречною при несправностях обладнання, що призводять до порушення списку. Зручною властивістю двоспрямованого списку є те, що якщо Point – вказівник на деякий елемент списку, то

Point->prev->next = Point = Point->next->prev

або

Point->next->prev = Point = Point->prev->next

Нижче наводиться приклад опису елемента двоспрямованого списку на C ++ (див. рисунок 3.2).

```
wp = Start; // ініціалізація вказівника на поточний елемент
                    // тепер поточним є перший
```

```
// цикл проходу по списку задля обробки всіх його елементів

while(wp !=NULL)  //поки поточний не отримав значення NULL
    {

         …  // тут знаходяться інструкції по обробці поточного
               // елементу, до якого отримуємо доступ через вказівник wp

        wp = wp->next;  // в wp заносимо адресу наступного елементу

    }
```

Рисунок 3.2 – Приклад опису типів для двуспрямованого списку та його ініціалізація

Над двоспрямованими списками можна виконати наступні операції:

- додавання елемента у початок списку;
- додавання елемента у кінець списку;
- додавання елемента у відсортований список;
- видалення елемента з списку;
- пошук елемента у списку;
- перевірка, чи порожній список;
- видалення списку;
- ініціалізація списку.

Розглянемо докладніше кожну з цих операцій.

3.1.1 Додавання елемента в початок двоспрямованого списку

При додаванні елемента у початок двоспрямованого списку необхідно розглянути випадки:

а) список порожній;

б) список містить елементи.

Якщо список порожній, то елемент стає першим і єдиним в списку. При цьому необхідно у нового елемента встановити поля next і prev в NULL і вказівником на початок списку привласнити вказівник на новий елемент. На рисунку 3.3 приведена структура списку, коли в ньому один елемент.

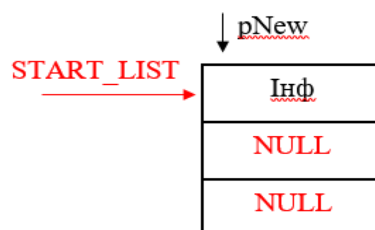


Рисунок 3.3 - Структура двоспрямованого лінійного списку з одним елементом

Якщо новий елемент додається у початок непорожнього списку, тоді необхідно полю next нового елемента привласнити вказівник першого елемента списку, поле prev встановити в значенні NULL, полю prev першого елемента початкового списку і вказівником на початок списку
привласнити вказівник на новий елемент (рисунок 3.4).

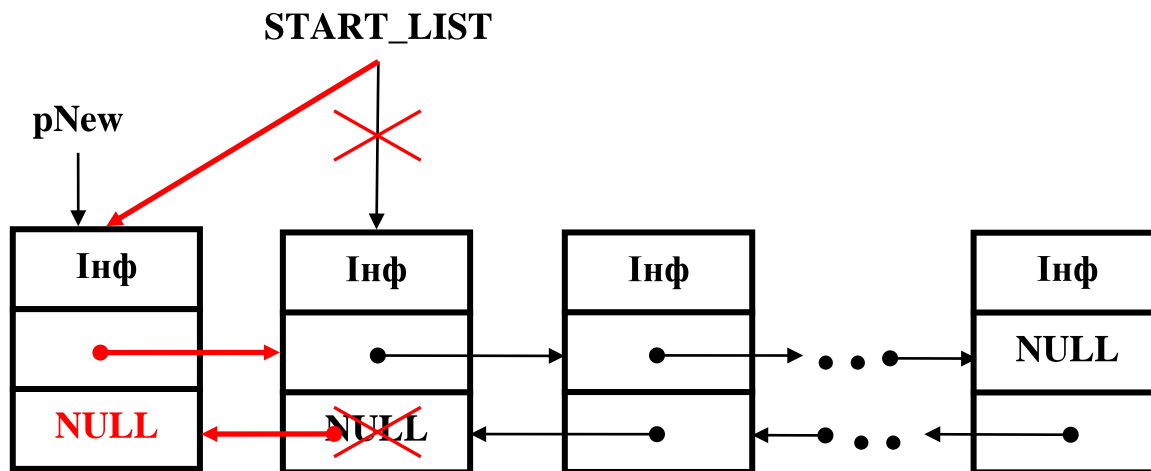


Рисунок 3.4 – Додавання елемента у початок двоспрямованого списку

3.1.2 Додавання елемента у кінець двоспрямованого списку

Перед додаванням елементу у кінець списку необхідно перевірити чи є в списку елементи. Якщо елементів немає, то для додавання стає першим. Якщо ж у списку є елементи, то необхідно переміститися в кінець списку і полю next останнього елемента привласнити вказівник на новий елемент. При цьому слід в поле prev нового елемента записати вказівник на останній елемент вихідного списку, а поле next нового елемента встановити в значення NULL (рисунок 3.5).

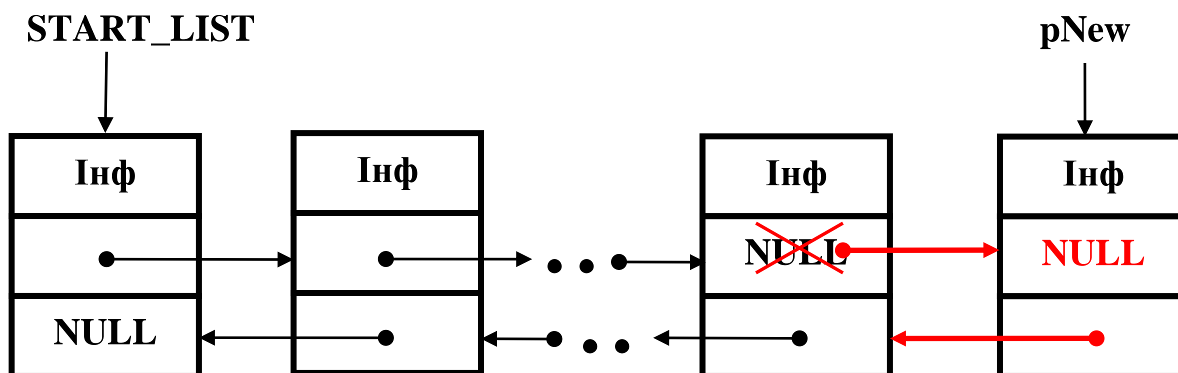


Рисунок 3.5– Додавання елементу у кінець двоспрямованого списку

3.1.3 Додавання елемента у відсортований двоспрямований список

Операція додавання елемента у відсортований список передбачає

обробку наступних ситуацій:

а) новий елемент становиться першим (аналогічно додаванню елемента у початок списку, рисунки 3.3 та 3.4);

б) новий елемент перевищує всі елементы списку (аналогічно додаванню елемента у кінець списку, рисунок 3.5);

в) новий елемент вставляється у середину списку.

Якщо новий елемент додається в середину початкогово списку, то необхідно відредагувати поля-вказівники у елементів після і перед якими він додається (рисунок 3.6).

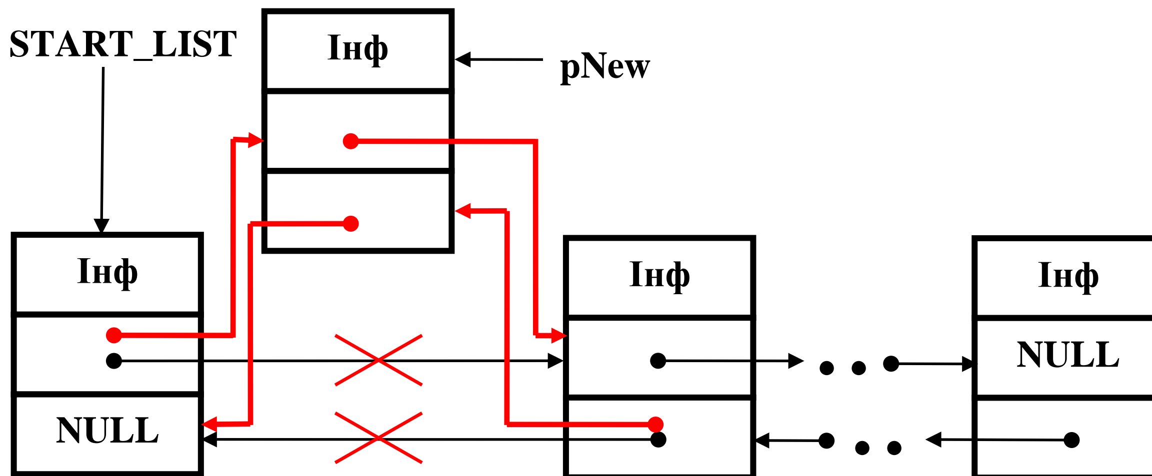


Рисунок 3.6 – Додавання елементу у відсортований двоспрямований список

У полі next нового елемента необхідно помістити вказівник на елемент перед яким він додається. В поле prev елемента перед яким додається новий елемент, потрібно помістити вказівник на новий елемент. Полю next елемента після якого додається новий елемент, необхідно присвоїти вказівник на новий елемент. Полю prev нового елемента привласнити вказівник на елемент, після якого він додається в список.

Нижче наводиться приклад процедури додавання елемента у відсортований двоспярмований список на мові C ++ (рисунок 3.7). Процедура в якості параметра отримує вказівник на початок списку і вказівник на елемент. Вказівник на початок списку передається по посиланню, і його нове значення буде повертатися з процедури в основну програму.

```
//Функція додавання елементу у відсортований двоспрямований список
//Параметри: Start –вказівник початку списка, pNew – вказівник на новий елемент void
addToSortDblList(TSNum * &Start,TSNum *pNew)
{
   TSNum *WP;  //Вказівник на поточний елемент списку
   TSNum *PP;  //Вказівник на попередній, відносно WP, елемент списку
   pNew->next=NULL;    //  Встановлює для нового елемента вказівник на наступний
   pNew->prev=NULL;    // Елемент next і вказівник на попередній елемент prev в NULL
   //Шукаємо у циклі потрібне місце для вузла
   PP=NULL;
```

```
    WP=Start;
    while ((WP!=NULL)&&(pNew->num>WP->num))
    {
      PP=WP;
      WP=WP->next;    //До наступного елемента
    }
    if (PP==NULL) {     //Якщо елемент стане першим у списку
      pNew->next=Start;
      if (Start!=NULL) Start->prev=pNew;   //Якщо список не порожній
      Start=pNew;
            }else { //Новий елемент після PP перед WP
      PP->next=pNew;
      pNew->prev=PP;
      if (WP!=NULL) { //Якщо елемент становиться НЕ останнім у списку
        WP->prev=pNew;
        pNew->next=WP;
              }//if
            }//if-else
}
```

Рисунок 3.7 – Процедура додавання елемента у відсортований двоспрямований список на мові C++

3.1.4 Видалення елемента з двоспрямованого списку

Операція видалення елемента з двоспрямованого списку здійснюється багато в чому аналогічно видалення з односпрямованого списку. Слід звернути увагу на два особливих випадки - видаляємий елемент є першим або останнім елементом списку, якщо видаляється елемент, який перший у списку (рисунок 3.8), то вказівник початку списку (start_list), переміщується на наступний за видаляємим елементом, в полі prev цього елемента заноситься null і звільняється пам'ять, яку займає елемент, що видаляється

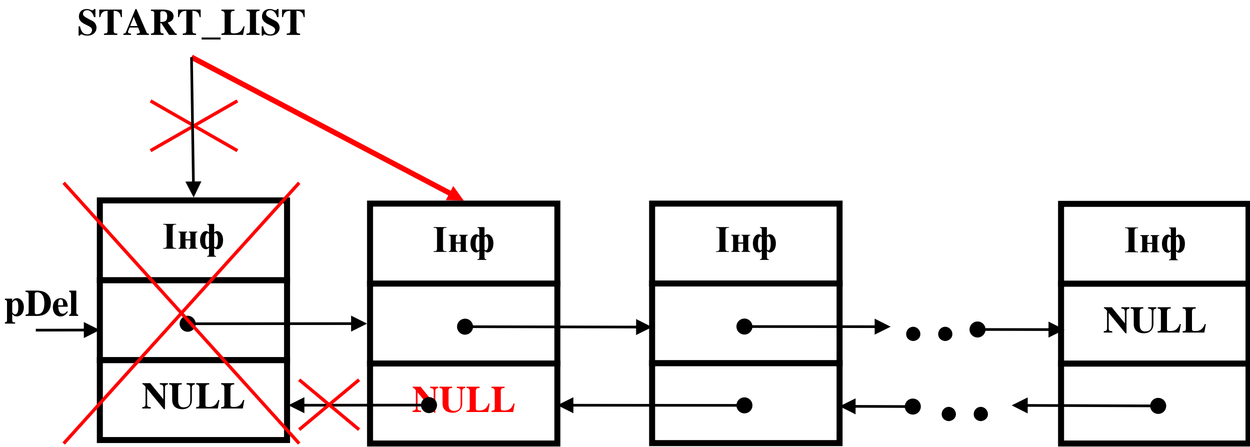


Рисунок 3.8 – Видалення першого елемента списка

Якщо видаляємий елемент - останній в списку (рисунок 3.9), то полю next передостаннього елемента присвоюється значення NULL і звільняється пам'ять, яку займає видаляємий елемент.

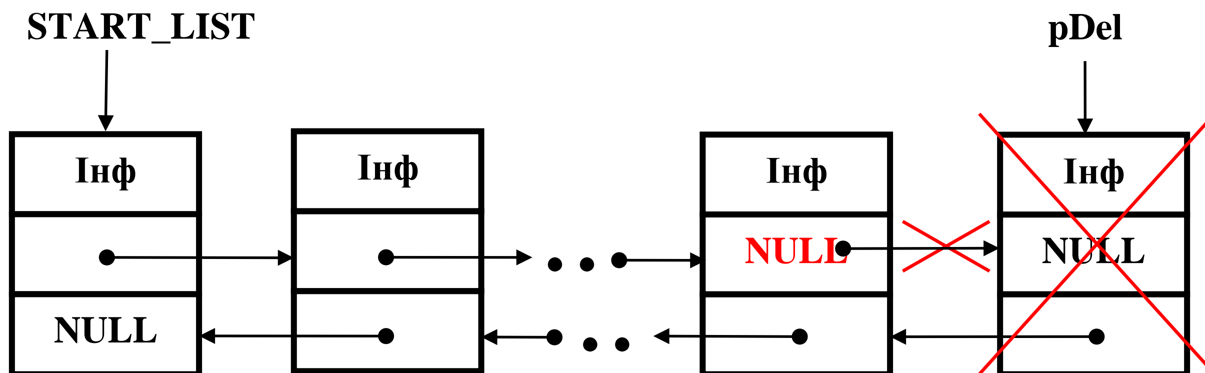


Рисунок 3.9 – Видалення останнього елемента списка

Третій випадок виникає, якщо видаляємий елемент знаходиться в середині списку. При цьому необхідно полю next попереднього елемента привласнити вказівник на наступний після видаляємого елемента, а полю prev наступного елемента привласнити вказівник на попередній елемент і звільнити пам'ять, займану видаляємим елементом (рисунок 3.10).

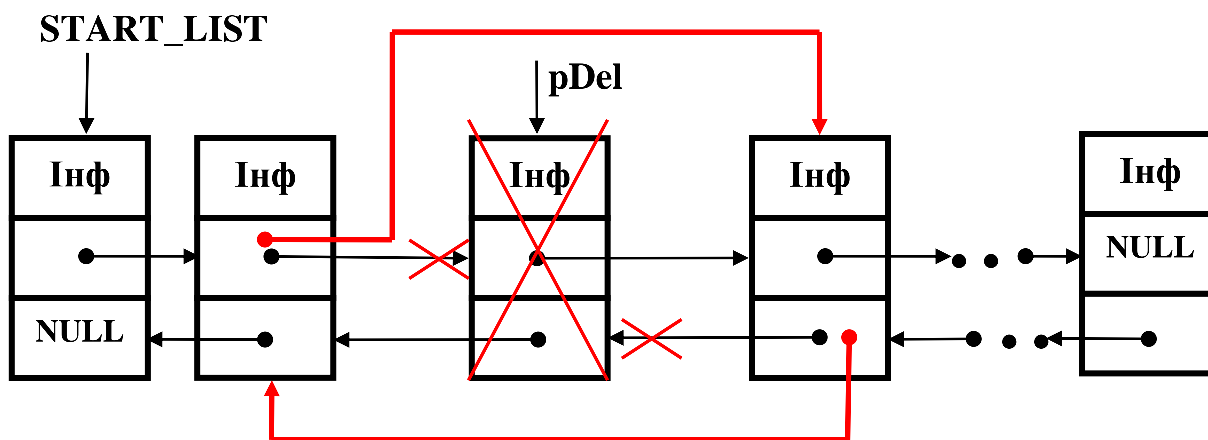


Рисунок 3.10 – Видалення елемента з середини списка

3.1.5 Пошук елемента у двоспрямованому списку

Операція пошуку елемента в двоспрямованому списку полягає в послідовному перегляді всіх елементів списку до тих пір, поки поточний елемент не буде містити задане значення або поки не буде досягнутий кінець списку. В останньому випадку фіксується відсутність шуканого елемента в списку

Пошук елемента у двоспрямованому списку можна вести:

а) переглядаючи елементи від початку до кінця списку;
б) переглядаючи елементи від кінця списку до початку;
в) переглядаючи список в обох напрямках одночасно: від початку до середини списку і від кінця до середини (беручи до уваги, що елементів у списку може бути як парна, так і непарна кількість)

3.1.6 Перевірка порожнечі двоспрямованого списку

Операція перевірки двоспрямованого списку на порожнечу відбувається порівнянням вказівника на початок списку (START_LIST) із значенням NULL. Якщо значення вказівника дорівнює NULL, то список пустий, в іншому випадку – у списку є елементи.

3.1.7 Ініціалізація та створення двоспрямованого списка цілих чисел

Для того, щоб створити список, необхідно послідовно виділяти пам'ять під нові елементи, а потім, за допомогою підпрограми додавання, додавати їх в список. Додавання може виконуватися у довільне місце списку: початок, кінець, середину.

Нижче наводиться приклад підпрограми ініціалізації і заповнення двоспрямовленого списку цілих чисел (рисунки 3.11 і 3.12).

```
//Функція ініціалізації списку
//Отримує вказівник початку списка Start і
//встновлює у NULL
 void initList(TSNum * &Start)
{
   Start = NULL;
}
```

Рисунок 3.11 – Підпрограма ініціалізації двоспрямованого списку

В якості парамаетра підпрограма отримує вказівник на початок списку і кількість елементів списку. Вказівник на початок списку передається по посиланню, і його нове значення буде повертатися з підпрограми в основну програму. Щоб додати елемент до списку використовується функція вставки елемента у відсортований список, розглянута раніше.

```
// Функція створення списку з випадкових чисел
// Параметри: Start - вказівник початку списку,
// cnt - кількість елементів списку
void createRndList(TSNum * &Start, int cnt)
{
   TSNum *pNew;    //Вказівник на новий елемент
   srand(time(NULL));  // "ініціалізіруємо" генератор випадкових чисел
                   // За зерно беремо кількість секунд з 1.01.1970 року    for (int i=1;i<=cnt;i++) {      //В циклі
заповнюємо список
      pNew=new TSNum;           //Створюємо новий елемент
//!!!!!!!!!!!!!!     УВАГА!      !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
// Поля next і prev нового елемента pNew не встановлює в NULL,
// так як це передбачено в функціях додавання ел. в список.
// Якби цього передбачено не було, то тут треба було б написати
// pNew-> next = NULL; pNew->prev=NULL;
//!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      pNew->num=rand()%100;      //Беремо ціле від ділення на 100        addToSortDblList(Start,pNew);//Заносимо
у кінець списку
   }//for
}
```

Рисунок 3.12 - Підпрограмма створення двоспрямованого списка випадкових чисел

3.1.8 Видалення двоспрямованого списку

Операція видалення списку полягає у звільненні динамічної пам'яті. Для даної операції організовується функція, в якій потрібно переставляти вказівник на наступний елемент списку до тих пір, поки вказівник не стане дорівнювати NULL, тобто не буде досягнутий кінець списку.

3.2 Створення проекта по обробці двоспрямованого списку

3.2.1 Завдання, що вирішується проектом

Завдання полягає в наступному: Створити невпорядкований двоспрямований список цілих чисел. Реалізувати можливість видалення будь-якого елементу зі списку. Виконати обробку двоспрямованого списку, відповідно до варіанту завдання і вивести початковий список і кінцевий список на екран. Для перевірки правильності рішення , кінцевий список необхідно вивести на екран два рази - при проході від початку до кінця списку і навпаки.

3.2.2 Опис типів для проекта

Зміст заголовного файлу, у якому описані типи, показано на рис. 3.13.

```
#ifndef TYPES_H
#define TYPES_H

using namespace std;

typedef
    struct SNum{

      int    num;
      SNum *   next;


      SNum *   prev;



} TSNum;

#endif // TYPES_H
```

Рисунок 3.13 – Зміст заголовного файлу, в кому описані типи

3.2.3 Реалізація функції ініціалізації та створення списку випадкових чисел

Для ініціалізації списку і створення списку випадкових чисел використовуються підпрограми, розглянуті раніше (рисунки 3.11, 3.12).

У функції createRndList, представленої на рисунку 3.11 раніше, викликається підпрограма додавання елемента у відсортований двоспрямований список. Вам необхідно використати підпрограму додавання в початок двоспрямованого списку або в кінець двоспрямованого списку. На рисунку 3.14 показана підпрограма додавання елемента в кінець двоспрямованого списку. Підпрограму додавання елемента в початок двоспрямованого списку необхідно написати самостійно.

```
void addToEndDblList(TSNum * &Start,TSNum *pNew)
{
    TSNum *WP;
    pNew->next=NULL;    //  Устанавливает для нового элемента указатель на следующий
    pNew->prev=NULL;    //элемент next и указатель на предыдущий элемент prev в NULL
    if (Start==NULL)    // Если список пустой
    {
        Start=pNew;     // Элемент становится первым и одновременно
        return;         // последним в списке
    }
    WP=Start;
    while (WP->next!=NULL)  //Ищем последний элемент списка в цикле
    {
        WP=WP->next;        //К следующему элементу
    };
    WP->next=pNew;          //Корректируем поле next последнего элемента
    pNew->prev=WP;          //и поле prev нового элемента
}
```

Рисунок 3.14 –Підпрограма додаванн елемента у кінець двоспрямованого списку

Після виведення початкового списку на екран (функцію виведення двоспрямованого списку на екран реалізувати самостійно), можна перевірити правильність написання процедури додавання елемента в початок або кінець двоспрямованого списку шляхом виклику підпрограми showControlOutputDblList, вона отримує вказівник на початок двоспрямованого списку і виводить його зміст в прямому і зворотному напрямку на екран. Підпрограма showControlOutputDblList представлена на рисунку 3.15.

```
void showControlOutputDblList(TSNum *Start)
{
//Вывод списка в прямом порядке
    cout<<endl<<"Содержимое списка в прямом порядке"<<endl;
    TSNum *WP;      //Указатель на текущий элемент списка
    TSNum *PP;      //Указатель на предыдущий, относительно WP, элемент списка

    WP=Start;       //Устанавливаем тнкущий указатель на начало списка
    PP=NULL;
    //Перебираем в цикле все элементы списка и выводим на экран
    while (WP!=NULL)
    {
        cout<<WP->num<<endl;
        PP=WP;
        WP=WP->next;    //К следующему элементу
    }
//Вывод списка в обратном порядке
        cout<<endl<<"Содержимое списка в обратном порядке"<<endl;

        WP=Start;       //Устанавливаем тнкущий указатель на начало списка
        //Перебираем в цикле все элементы списка и выводим на экран
        while (PP!=NULL)
        {
            cout<<PP->num<<endl;
            PP=PP->prev;    //К предыдущему элементу
        }
        cout<<endl;
}
```

Рисунок 3.15 – Підпрограма контрольного виведення двоспрямованого списку в обох напрямах на екран

Якщо список правильно виведений в обох напрямках, це означає, що операція по його обробці виконана правильно.

На рисунку 3.16 представлені заголовки з прототипами функцій проекту.

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

#include "types.h"

void initList(TSNum * &Start);
void createRndList(TSNum * &Start, int cnt);
void addToEndDblList(TSNum * &Start,TSNum *pNew);
void addToSortDblList(TSNum * &Start,TSNum *pNew);
void showList(TSNum *Start);
void showControlOutputDblList(TSNum *Start);
void multNeighb(TSNum *Start);

#endif // FUNCTIONS_H
```

Рисунок 3.16 – Заголовки з прототипами функцій проекту.

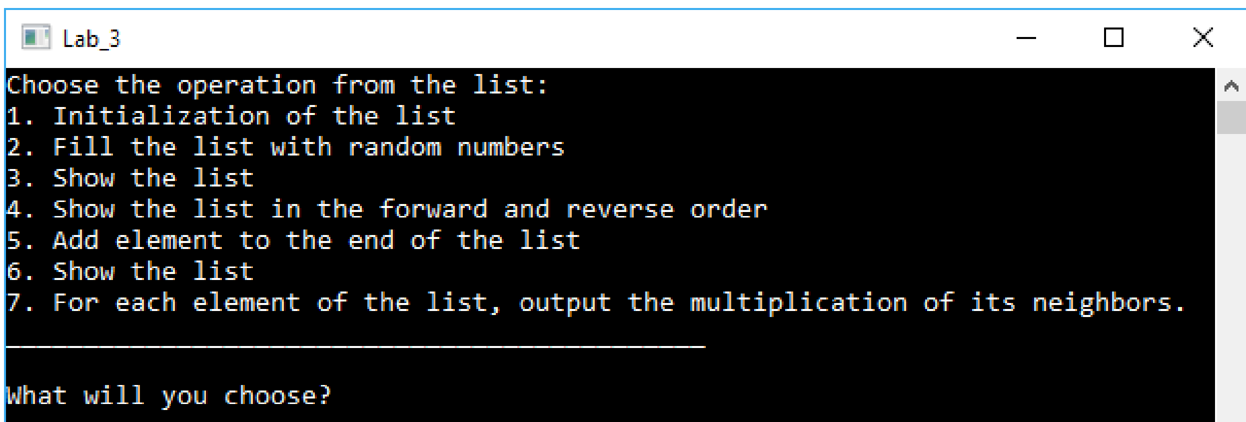
Як приклад обробки двоспрямованого списку на рисунку 3.17 представлена функція виведення на екран добутку лівого і правого сусідніх елементів списку.

```
void multNeighb(TSNum *Start)
{
    TSNum *WP;          //Указатель на текущий элемент списка
    int left,right;     //Числовые значения левого и правого соседа
                        //текущего элемента

    WP=Start;           //Текущий элемент на начало списка
    while (WP!=NULL)  { //Проход циклом по элементах списка
        //Если элемент в списке толко один - у него нет соседей
        if ((WP->next==WP->prev)&&(WP->next==NULL)) {
            cout<<"В списке только один элемент"<<endl;
            break;
        };//if
        //Если нет следующего элемента устанавливаем числовое значение
        //правого соседа в 1
        if (WP->next==NULL) right=1;
            else right=WP->next->num;
        //Если нет предыдущего элемента устанавливаем числовое значение
        //левого соседа в 1
        if (WP->prev==NULL) left=1;
            else left=WP->prev->num;
        //Вывод произведения правого и левого соседей элемента
        cout<<"для "<<WP->num<<":  "<<left<<"*"<<right<<"="<<left*right<<endl;

        WP=WP->next;   //Следующий элемент списка
    };//while
}
```

Рисунок 3.17 – Функція виведення добутку сусідніх елементів вузла списку. в



Базовий інтерфейс проекта представлений на малюнку 3.18, а текст основного модуля програми на малюнку 3.19

Рисунок 3.18 – Базовий інтерфейс проекта роботи із двоспрямованим списком

```
#include <QtCore/QCoreApplication>
#include "functions.h"
#include "types.h"
#include "windows.h"
#include <iostream>
TSNum * START_LIST;  // оголошення змінної - вказівника на початок списку
char getCommand();
int main(int argc, char *argv[])
{
   char ch;
   QCoreApplication a(argc, argv);
   setlocale(LC_ALL, "Russian");
  do {
   ch=getCommand();
   switch(ch){
      case '1': {initList(START_LIST);
         cout<<endl;
         cout<<"Ініціалізація пройшла успішно \n";
         cout<<endl;
         }
         break;
      case '2':  {
          int cnt;
          cout<<"Введіть кількість елментів ->";
          cin>>cnt;
          cout<<endl;
          createRndList(START_LIST,cnt);
          cout<<"Заповнення списку пройшло успішно \n";
          cout<<endl;
          showList(START_LIST);
          }
          break;
      case '3': showList(START_LIST);
          break;
      case '4': showControlOutputDblList(START_LIST);
          break;
      case '5': {
          TSNum *pNew;
          pNew=new TSNum;
          cout<<"Введіть число  - елемент списку->";
          cin>>pNew->num;
          // New-> next = NULL і New-> prev = NULL передбачено
         // в функції додавання
          cout<<endl;
          addToEndDblList(START_LIST,pNew);
          cout<<"Елемент доданий у список\n";
          cout<<endl;
          showList(START_LIST);
          }
          break;
      case '6': {
          multNeighb(START_LIST);
          cout<<endl;
          }
          break;
      case '7': exit(0);
          break;
   }
   cout<<"\n Для продовження натисніть * ->";
   cin>>ch;
  }while(true);
   return a.exec();
```

```
}

char getCommand()
{
   char ch;//номер обраного пункта меню (в символьному варианті)
      system("cls");
      SetConsoleTitleA("LAB №4");
      cout<<"Оберіть операцію для списка: \n";
      cout<<endl;
      cout<<"1.Ініціалізація списка \n";
      cout<<endl;
      cout<<"2.Заповнення списку випадковими числами \n";
      cout<<endl;
      cout<<"3.Виведення списку на екран\n";
      cout<<endl;
      cout<<"4.Контрольне виведення списка в прямому та обратному порядках\n";
      cout<<endl;
      cout<<"5.Додати елемент у кінець списку\n";
      cout<<endl;
      cout<<"6.Для кожного елемента списку вивести добуток його сусідів\n";
      cout<<endl;
      cout<<"7.Вихід із програми\n";
      cout<<"---------------------------------\n";
      cout<<endl;
      cout<<"Оберіть операцію для списку \n";
      cin>>ch;
      return ch;
          }
```

Рисунок 3.19 – Текст основного модуля програми

3.2.4 Реалізація функції видалення елементу з двоспрямованого списку

Функція видалення елемента з двоспрямованого списку подібна до підпрограми, що видаляє елемент в односпрямованому списку.

Процедура видалення елемента повинна бути реалізована самостійно. Також самостійно необхідно написати підпрограму видалення всього двоспрямованого списку.

При їх створенні слід врахувати, що в даному проекті список не впорядкований.

Крім того, в процедурі видалення для двоспрямованого списку необхідно, при вілученні елементу зі списку, подбати не тільки про поля next елементів, а й про поля prev.

При розгляданні наступного елемента списку через поле next або prev слід перевірити, чи не рівний цей вказівник NULL (в тих випадках, де така ситуація можлива).

За бажанням процедура може бути написана таким чином, щоб видалявся не перший знайдений по ключу елемент, а всі елементи із заданим ключем. Таке завдання складніше, але дозволить поліпшити навички у роботі зі списками.

3.3 Завдання для виконання проекта

1. Скопіюйте проект лабораторної роботи № 2;
2. Змініть інтерфейс згідно функціональності проекту;
3. В заголовочному файлі виконайте редагуванняя згідно п.3.2.2;
4. В заголовочному файлі з прототипами функцій внесіть зміни згідно рисунку 3.16;
5. Реалізуйте функції для роботи з двоспрямованим списком, розглянутим у рамках цієї лабораторної роботи;
6. Створіть власну процедуру додавання елемента у початок двоспрямованого списку;
7. Створіть власну процедуру видалення елементу з двоспрямованого списку;
8. Створіть власну процедуру видалення всього двоспрямованого списку;
9. Реалізуйте завдання по обробці двоспрямованого списку згідно варіанта, наведеного у таблиці 3.1.

3.4 Тестування проекта

Виконайте тестування проекту наступним чином:

1. Покажіть вид вікна програмі після того, як виповнена операція заповнення списку і його виведення на екран в обох напрямках;
2. Виконайте видаленная вибранного елемента і результат видалення покажіть у вигляді зображення вікна програмі із виведенням списку на екран в обох напрямках;
3. Виконайте завдання 1, згідно варіанту, вручну і зобразіть його результат у звіті;
4. Виконайте завдання 1 за допомогою проекту і зображення вікна програми із результатом зобразіть у звіті;
5. Аналогічно виконайте тестування другого завдання.

3.5 Вміст звіта

– 1. Назва роботи.
– 2. Мета роботи.
– 3. Текст програми.
– 4. Результати тестування проекту.

Таблиця 3.1 – Завдання для виконання проекту

Вариант	Реалізувати операцію з двоспрямованим списком:
0	1. Вивести на екран елементи списку, у яких лівий і правий сусід менше по значенню. 2. Видалити зі списку всі елементі, що кратні п’яти.
1	1. Визначити кількість елементів списку для яких виконується умова: сума попередніх елементів менша за суму наступних за ним елементів і вивести ці елементі на екран та накопленні з двох сторон суми. 2. Після кожного парного елемента вставити новий елемент, на один більше за поточний.
2	1. Вивести на екран елементи, для яких виконується умова: кількість парних елементів, що передують поточному, дорівнює кількості непарних елементів, наступних за поточним. 2. Перед кожним елементом, кратним 10, вставити два нових з таким самим значенням.
3	1. Якщо в списку йдуть підряд кілька парних елементів, то блоки поспіль парних замінити одним елементом, що дорівнює сумі цих парних. 2. Видалити зі списку всі елементи, що закінчуються на цифру 2.
4	1. Якщо зустрічається непарний елемент, то перед ним і після нього вставити нові елементи зі значенням 0. 2. Для кожного елемента обчислити суму попередніх і наступних елементів і вивести на екран таку інформацію: сума попередніх, елемент, сума наступних.
5	1. Знайти максимальний елемент в списку. Серед елементів, що знаходяться перед ним видалити всі кратні трьом, а для знаходяться після нього - видалити всі кратні п'яти. 2. Вивести на екран елементи, у яких сусіди праворуч і ліворуч парні.
6	1.Знайти середнє арифметичне елементів списку. Для першого знайденого елемента, кратного п'яти, виконати наступне: серед попередніх елементів вивести на екран всі, менші середнього арифметичного, а для наступних – всі, що більші за середнє арифметичне. 2. Видалити зі списку всі елементи кратні 5.
7	1. Видалити блоки непраних елементів, що зустрічаються поспіль. 2. Для кожного елемента вивести на екран суму його сусідів.
8	1.Вивести на екран елементи, для яких в списку попередніх елементів кількість парних дорівнює кількості парних в списку наступних елементів. 2. Після кожного елемента, кратного трьом, вставити новий елемент зі значенням 10.
9	1.Вивести на екран елементи, для яких сума двох попередніх елементів менше суми двох наступних. А також вивести на екран ці суми. 2.Видаліть ті елементи списку, після яких слідують непарні.

Лабораторна работа № 4

Реалізація мультиспискової структури ієрархічного типу на базі лінійних зв'язаних списків

Мета роботи:
- Ознайомитися з поняттям «мультисписок»;
- Ознайомитися з принципами обробки підсписків у мультисписках на прикладі проекту, у якому реалізована операція додавання елементів до мультисписку;
- Створити проект у системі QtCreator, у якому реалізовані основні операції обробки мультиспискової структури.

4.1 Теоретичні відомості

Мультисписком називають структуру даних, що включає в себе списки різних типів. У мультиспискові структурі списки, на базі яких вона створена, взаємопов'язані між собою за допомогою вказівників.

У даній лабораторній роботі розглядається робота з мультисписком ієрархічного типу, побудованого на базі лінійних зв'язаних односпрямованих списків.

Під ієрархічною структурою мається на увазі те, що між її елементами існує зв'язок підпорядкування: від елементів верхнього рівня є зв'язок до елементів більш низького рівня. Від елемента низького рівня неможливо отримати доступ до елемента верхнього рівня.

Мультисписки використовуються у тих випадках, коли декілька лінійних спискових структур необхідно об'єднати в одну взаємопов'язану структуру.

Будемо розрізняти верхній рівень мультисписку та інші – вкладені рівні мультисписку. Списки, розташовані на вкладених рівнях мультисписку, будемо називати підсписками. Наприклад, логічна структура даних Міністерства освіти, яка включає список вищих навчальних закладів, що входять у цей список та відповідні підсписки – списки факультетів та кафедр мають вид дерева (див. рис. 4.1), логічна структура даних відображає загальне представлення користувача про дані та їх взаємозв'язки, вона не пов'язана з реальним представленням даних у пам'яті комп'ютера.

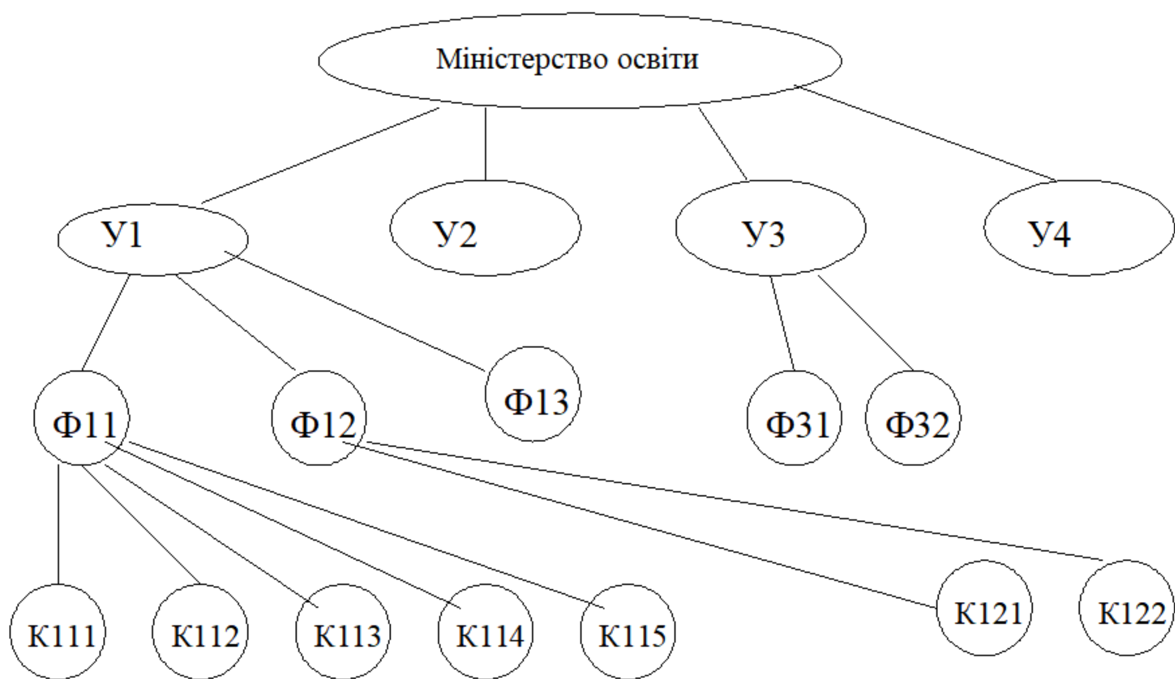


Рисунок 4.1 – Логічна структура даних «Міністерство освіти»

Фізична структура даних показує, як дані реально представлені у пам'яті комп'ютера та як вони взаємопов'язані між собою.

Фізична структура даних «Міністерство освіти» представляє собою мультисписок ієрархчного типу, побудований на основі лінійних односпрямованих зв'язаних списків і показана на рисунку 4.2.

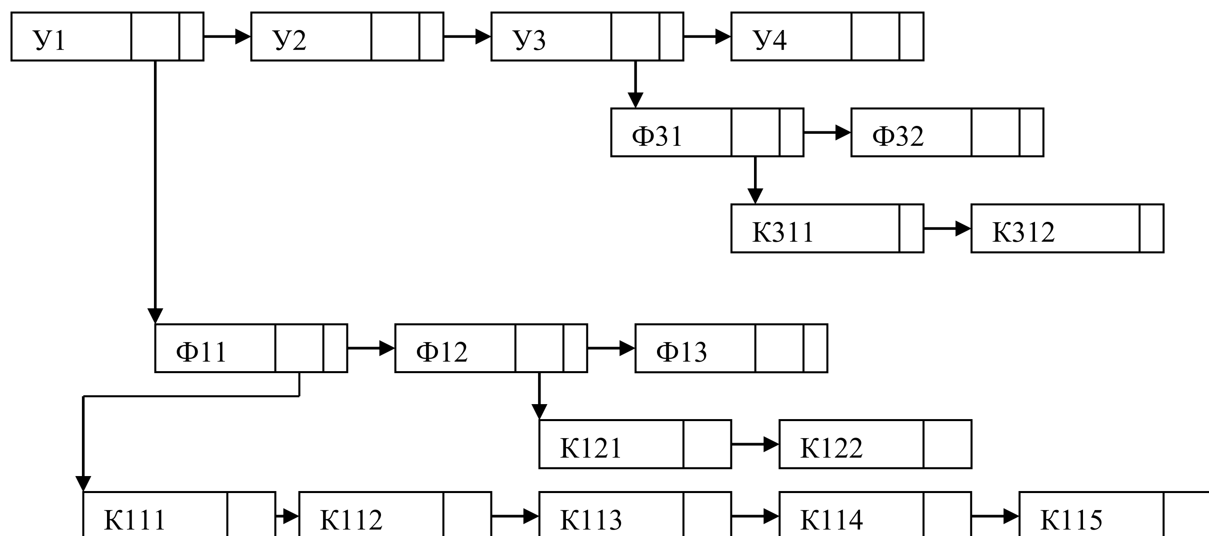


Рисунок 4.2 – Фізична структура даних «Міністерство освіти»

Для реалізації структури, представленої на рисунку 4.2 необхідно в

елементи мультисписку всіх рівнів, крім найнижчого, включити ще одне поле-вказівник, котре буде зберігати адрес першого елементу підсписку.

Мультисписок, зображений на рисунку 4.2 має три рівні. Обробка кожного його підсписку аналогічна обробці елементів звичайного односпрямованого лінійного списку.

4.2 Створення проекту, якиий забезпечує створення та додавання елементів до мультисписку

4.2.1 Задача, вирішувана проектом

Задача полягає в тому, що необхідно розробити програму, зо забезпечує створення та додання даних про Міністерство освіти. База даних «Міністерство освіти» представляє собою мультисписковy ієрархічну структуру даних, що містить інформацію про університети та факультети.

Фізична структура даних має структуру, схожу на структуру, представлену на рисунку 4.2, але для простоти виконання реалізуємо мультисписок, котрий матиме лише два рівні:

- Університети;
- Факультети.

4.2.2 Інтерфейс проекту

Початковий вигляд інтерфейсу проекту показано на рисунку 4.3. для виводу підсписків мультисписку використовується консоль.

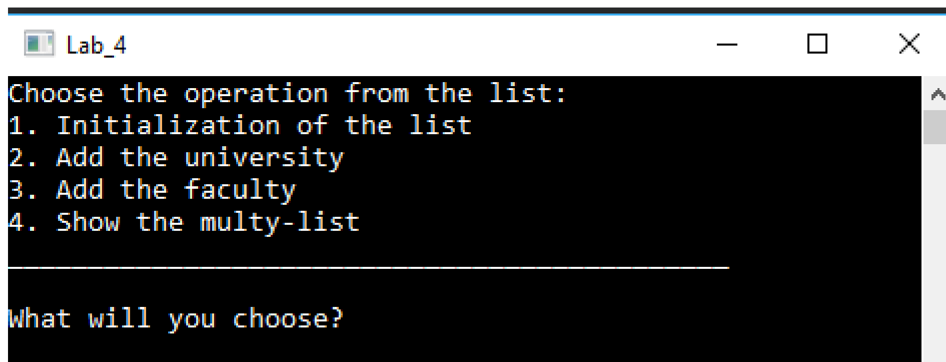


Рисунок 4.3 – Вид консолі при запуску проекту

Під час запуску проекту на екрані відображується перелік можливих операцій з обробки мультисписку та їх номери. Вибір необходної операції виконується через задання номеру операції.

Після виконання кожної операції з мультисписком, його вміст виводиться на екран.

Для виводу елементів другого рівня дометаніше робити деякий відступ вправо для зручності перегляду вмісту мультисписку.

Для додання нового елементу в мультисписок на консоль виводиться текст з пропозицією ввести дані про новий елемент.

4.2.3 Опис типів для проекту

Вміст модуля, у якому описані типи показано на рисунку 4.3.

```
#ifndef TYPES_H
#define TYPES_H
#include <string>

using namespace std;

  // описание типа для элемента списка - ФАКУЛЬТЕТ
typedef
    struct Facult{
      string   nameFacult;      /* название факультета */
      int      countSpec;    /* количество специальностей */
      string   decan;     /* фамилия декана */
      Facult * next;    /* поле-указатель на следующий факультет */
    } TFacult;


  // описание типа для элемента списка - УНИВЕРСИТЕТ
typedef
    struct Univ{
      string   nameUniv;      /* название университета */
      int      acredit;    /* уровень аккредитации */
      string   rector;     /* фамилия ректора */
      Univ *   next;    /* поле-указатель на следующий университет */
      Facult * sublevel; /* поле-указатель на подсписок факультетов */
    } TUniv;


#endif // TYPES_H
```

Рисунок 4.3 – Вміст модуля типів

Слід звернути увагу на те, що оскільки у елементів списку університетів є підсписки факультетів, то в записи університету добавлено поле-вказівник на початок списка факультетів.

У факультетів немає підсписків, а тому у їх елементів тільки одне поле-вказівник на наступний елемент.

Не дивлячись на те, що мультисписок складається з багатьох підсписків, для роботи з ним необхідно тільки одна глобальна змінна Start – вказівник на початок мультиспиcку (на перший елемент списку верхнього рівня, тобто на перший елемент списку університетів). Указівники на початки підсписків факультетів зберігаються в полях-вказівниках Sublevel відповідних університетів.

4.2.4 Опис модулів проекту

Для реализації проекту створюємо модулі:

1. Модуль «main.cpp» - головний модуль, котрий буде ініціалізувати необхідні операції з мультисписком.
2. Модуль «types.h» – модуль типів.
3. Модуль «functions.cpp», «functions.h» – модулі функций з обробки списків.

4.2.5 Реалізація функції ініціалізації мультиспискy

Ініціалізація мультиспиcка полягає узанесенні значення NULL у змінну – вказівник на початок мультиспиcка.

```
#include "types.h"
#include <iostream>

  /*  функция инициализации списка */
  /*  получает Start - и заменяет его значением  NULL*/

void  initList (TUniv * &Start)
{
   Start=NULL;
}
```

Рисунок 4.4 – Процедура ініціалізації мультиспискy

Процедура “showUniv” представлена на рисунку 4.5. Вона знаходиться в модулі “functions.cpp”, а її заголовок – у заголовному модулі з такою ж назвою.

```
void showUniv(TUniv * Start)
{
  TUniv * wp=Start; //инициализируем рабочий указатель
                    //значением Start.
                    //С помощью wp последовательно получаем доступ
                    // к каждому элементу списка
  while(wp != NULL)  // Пока wp не получил значение null
    {
     cout<<wp->nameUniv<<endl; //wp содержит адрес текущего элемента
                               // через него получаем доступ к полю nameUniv
     cout<<endl;
     wp=wp->next;              // wp получает адрес следующего элемента
    }
}
```

Рисунок 4.5 – Процедура выводу списку университетів на екран

4.2.6 Реалізація функції додання елементів до мультиспискy

Для збільшення швидкості обробки мультиспискy необхідно, щоб підсписки, на основі яких він організований, були відсортованими.

Додавання елементів у відсортовані списки аналогічно додаванню елементів в список, як виконувалося в лабораторній роботі № 3.

Процедура додавання нового університету у впорядкований список приведена на малюнку 4.6, а процедура додавання факультету - на малюнку 4.7.

```
void addToSortUniv(TUniv * &Start, TUniv * pNew)
{
  TUniv * pp, * wp;
  wp=Start;
  pp=NULL;
  while (wp != NULL  && pNew->nameUniv > wp->nameUniv)
    {
       pp = wp;
       wp = wp->next;
     }
  if (pp == NULL)
     { Start = pNew; }
  else
     { pp->next = pNew;}
  pNew->next = wp;
}
```

Рисунок 4.6 – Функція додання нового елемента в відсортований список університетів

Функція додавання нового факультету аналогічна функції додавання університету, тільки типи вказівників різні.

```
void addToSortFacult(TFacult * &Start, TFacult * pNew)
{
  TFacult * pp, * wp;
  wp=Start;
  pp=NULL;
  while (wp != NULL  && pNew->nameFacult > wp->nameFacult)
    {
       pp = wp;
       wp = wp->next;
     }
  if (pp == NULL)
     { Start = pNew; }
  else
     { pp->next = pNew;}
  pNew->next = wp;
}
```

Рисунок 4.7 – Функція додання нового елементу у відсортований список університетів

При додаванні нового університету ДУЖЕ ВАЖЛИВО в полі SubLevel записати значення NULL.

Так відбувається ініціалізація підсписку факультетів для цього університету. У нового елементу-університету немає доки факультетів, тому список факультетів для нього порожній, тобто вказівник на його початок повинен містити значення NULL.

Функція, що оброблює вибір операції "Додати університет" приведена на рисунку 4.8

Рисунок 4.8 – Функція, що оброблює вибір операції "Додати університет"

Новий елемент для списку університетів формується після введення

```
     // Функция выполнения операции добавления университета
void addUniv(TUniv* &Start){
    TUniv * pNew = new TUniv; //Запрашиваем место для нового
                              //элемента  списка

    //Ввод информации о новом университете и заполнение его полей
    cout<<"\nВВОД ИНФОРМАЦИИ О НОВОМ УНИВЕРСИТЕТЕ: ";
    cout<<endl;
    cout<<"\nВведите название университета: ";
    cin>>pNew->nameUniv;
    cout<<"\nВведите уровень аккредитации университета: ";
    cin>>pNew->acredit;
    cout<<"\nВведите фамилию ректора: ";
    cin>>pNew->rector;
    cout<<"\n";
    pNew->next = NULL;
    pNew->sublevel = NULL; // инициализация подсписка факультетов
    addToSortUniv(Start, pNew);
    cout<<"\nДобавление прошло успешно! ";
    cout<<"\n";
    cout<<"\nСОДЕРЖИМОЕ СПИСКА УНИВЕРСИТЕТОВ:\n";
    cout<<"\n";
       // Вывод всех названий университетов на экран
    showUniv(Start);

}
```

даних на консолі.

Для того, щоб додати факультет необхідно вибрати операцію "Додати факультет", після чого задати назву університету, до якого додається факультет. Якщо університет не заданий, то невідомо, в який підсписок додавати факультет. Отже, якщо не заданий університет, або заданого університету немає в списку університетів, то додавання факультету не відбувається.

Функція, що оброблює вибір операції "Додати факультет" просить назву університету і забезпечує введення даних про новий факультет і представлена на рисунку 4.9.

```
void addFct(TUniv* Start){

string univName;
TUniv * ppUniv, * wpUniv;
   //Ввод информации о новом университете и заполнение его полей
cout<<"\nВВОД ИНФОРМАЦИИ ОБ УНИВЕРСИТЕТЕ, ДЛЯ которого ВВОДИТСЯ ФАКУЛЬТЕТ: ";
cout<<endl;
cout<<"\nВведите название университета: ";
cin>>univName;
if (univName.length() == 0)
    {
        cout<<"\n             ВНИМАНИЕ! ";
        cout<<"\n   Название университета не задано!";
        cout<<"\nОсуществить ввод факультета невозможно";
        return;
   }

  // Поиск указателя на выбранный университет
  // в списке университетов по ключу

bool isFacult = findUniv(Start, univName, ppUniv, wpUniv);

if (!isFacult) // если факультет не найден в списке
    {
        cout<<"\n                         ВНИМАНИЕ! ";
        cout<<"\n   Название название университета введено не верно !";
        cout<<"\n         Осуществить ввод факультета невозможно";
        return;
    }


TFacult * pNew = new TFacult; //Запрашиваем место для нового факультета


//Ввод информации о новом факультете и заполнение его полей
cout<<"\nВВОД ИНФОРМАЦИИ О НОВОМ ФАКУЛЬТЕТЕ: ";
cout<<endl;
cout<<"\nВведите название факультета: ";
cin>>pNew->nameFacult;
cout<<"\nВведите количество специальностей факультета: ";
cin>>pNew->countSpec;
cout<<"\nВведите фамилию декана: ";
cin>>pNew->decan;
cout<<"\n";
pNew->next = NULL;
   // wpUniv содержит указатель на университет
   // для которого в подсписок добавляем факультет
addToSortFacult(wpUniv->sublevel, pNew);
cout<<"\nДобавление прошло успешно! ";
cout<<"\n";
cout<<"\nСОДЕРЖИМОЕ СПИСКА факультетов для университета: "<<univName<<endl;
cout<<"\n";
  // Вывод всех названий факультетов на экран
showFacult(wpUniv->sublevel);
}
```

Рисунок 4.9 – Функція, що оброблює вибір операції "Додати факультет"

У цій функції, передусім, робиться перевірка, чи заданий університет і чи існує він в списку. Якщо ні, то ніякі дії не виконуються. За бажанням можна видавати повідомлення про те, що університет не заданий або не існує.

Новий елемент для списку факультетів формується після введення даних на консолі.

Слід звернути увагу на відмінність цієї функції від функції додавання університету.

Перед викликом функції додавання факультету спочатку необхідно знайти Start списку факультетів, який знаходиться в полі Sublevel у вибраного

університету. Для того, щоб отримати доступ до поля SubLevel вибраного університету необхідно знайти вказівник на цей університет. Такий пошук можна здійснити за допомогою функції пошуку findUniv, яка отримає Start списку університетів і ключ пошуку - назва університету, яку було задано з консолі. Текст функції пошуку університету приведений на малюнку 4.10.

```
//--------------------------------------------------------
bool findUniv(TUniv * Start, string keyUniv, TUniv * &pp, TUniv * &wp)
{
        // Инициализация указателей
    pp=NULL;
    wp=Start;
        // Цикл прохода по списку
    while (wp != NULL)
    {
        if (wp->nameUniv <= keyUniv)
            { break;} // если нашли элемент или место ваставки
        else
            {
              pp=wp;
              wp=wp->next;
            }
    }
       // проверяем, найден ли элемент

    if (wp != NULL && wp->nameUniv == keyUniv)
        { return true;}
    else {return false; }
        // если элемент не найден, то праметры pp и wp
        // будут содержать адреса элементов списка, между
        //  которыми необхожимо вставить элемент с ключом keeyUniv
}
```

Рисунок 4.10 – Текст функції пошуку университету

Функція виводу списку факультетів на консоль показана на рис. 4.11

Рисунок 4.11 – Текст функції виводу списку факультетів на консоль

4.2.7 Реалізація функції видалення елемента

При видаленні елементу верхнього рівня перед тим, як звільнити пам'ять від елементу, що видаляється, необхідно викликати процедуру видалення підсписку факультетів. У іншому, видалення нічим не відрізняється від

```
void showFacult(TFacult * Start)
{
  TFacult * wp=Start; //инициализируем рабочий указатель
                    //значением Start.
                    //С помощью wp последовательно получаем доступ
                    // к каждому элементу списка
  while(wp != NULL)  // Пока wp не получил значение null
    {
     cout<<wp->nameFacult<<endl; //wp содержит адрес текущего элемента
                              // через него получаем доступ к полю nameFacult
     cout<<endl;
     wp=wp->next;             // wp получает адрес следующего элемента
    }
}
```

видалення елементу в лабораторній № 2.

Послідовність дій при видаленні факультету:

- введення назви університету;
- перевірка - чи введений університет, чи є такий університет в списку (виклик функції пошуку університету по його назві (по ключу));
- введення назви факультету, що видаляється;
- перевірка - чи введена назва факультету, чи є такий факультет в підсписку для заданого університету (виклик функції пошуку вибраного факультету);
- виклик функції видалення факультету;
- виведення списку факультетів на екран наново.

4.2.8 Реалізація функції виводу вмісту мультиспиську на екран

Функція виводу мультисписка на екран приведена на рисунку 4.12.

```
void showMultiList(TUniv * Start)
{
   TUniv * wpU = Start; // указатель на текущий университет
   TFacult * wpF;  // указатель на текущий факультет

   while (wpU != NULL) // проход по всему списку университетов
   {
       wpF = wpU->sublevel; // для текущего университета берем старт подсписка
       cout<<"УНИВЕРСИТЕТ:  " <<wpU->nameUniv<<endl;
       cout<<"ФАКУЛЬТЕТЫ:  "<<endl;
       while (wpF != NULL)
       {
        cout.width(18);
        cout<<wpF->nameFacult<<" ; ДЕКАН: "<<wpF->decan<<"  ;КОЛИЧ.СПЕЦ. "<<wpF->countSpec<<endl;
        wpF = wpF->next;
       }
       wpU = wpU->next;
   }
}
```

Рисунок 4.12 – Функція виводу мультисписка на екран

Результат роботи функції приведений на рисунку 4.13.

```
Л бор торн я №2

3.Добавление факультета

4.Вывод мультиспиcка

--------------------------------------------------

Выберите опер цию для списк
4
СОДЕРЖИМОЕ МУЛЬТИСПИСКА:
УНИВЕРСИТЕТЫ:  КПИ
ФАКУЛЬТЕТЫ:
    Меоборудование ; ДЕКАН: Причепа   ;КОЛИЧ.СПЕЦ. 7
УНИВЕРСИТЕТЫ:  МГУ
ФАКУЛЬТЕТЫ:
       Кибернетика ; ДЕКАН: Ткаченко   ;КОЛИЧ.СПЕЦ. 8
          Механика ; ДЕКАН: Петров   ;КОЛИЧ.СПЕЦ. 6
            Оптика ; ДЕКАН: Петров   ;КОЛИЧ.СПЕЦ. 3
УНИВЕРСИТЕТЫ:  ЧДТУ
ФАКУЛЬТЕТЫ:
       Информатика ; ДЕКАН: Толкачев   ;КОЛИЧ.СПЕЦ. 2
                ПИ ; ДЕКАН: Терехов   ;КОЛИЧ.СПЕЦ. 5
              ФЕИТ ; ДЕКАН: Иванец   ;КОЛИЧ.СПЕЦ. 4

Для продолжения н жмите *_
```

Рисунок 4.13 – зображення консолі з виведеним мультиспиcком

4.2.9 Реалізація функції видалення мультиспиcку

Для видалення усього мультиспиcку необхідно організувати цикл за списком верхнього рівня. Для кожного елементу верхнього рівня викликати функцію видалення його підсписку, а потім витягнути зі списку сам цей елемент і звільнити від нього пам'ять.

Ця процедура має бути реалізована самостійно.

4.3 Завдання для виконання проекту

1. Реалізуйте функції ініціалізації, додавання і перегляду для дворівневого мультиспиcку як було показано в п.5.2 згідно варіанту завдання;
2. Реалізуйте функцію видалення усього мультиспиcку;
3. Реалізуйте функції видалення елементів на верхньому і нижньому рівнях мультиспиcку;
4. Виконайте пошук інформації в мультиспиcку, згідно варіанту, завдання і результат пошуку виведіть на консоль.

Таблиця 4.1 – Завдання для виконання проекту

Варіант	Вміст мультиспиcку	Реалізувати пошук в мультиспиcку
0	Верхній рівень – магазини (назва, профіль) Нижній рівень – співробітники (прізвище, посада,	Вивести інформацію про магазини, в яких є співробітники із зарплатою менше 800 грн

	зарплата)	
1	Верхній рівень - аудиторії (номер аудиторії, кафедра) Нижній рівень - інвентар (назва і інвентарний номер)	Вивести інформацію про аудиторії, в яких є вішалка для одягу
2	Верхній рівень - країна (назва і валюта) Нижній рівень - спортсмен (прізвище, розряд, вид спорту)	Вивести інформацію про спортсменів, які займаються плаванням
3	Верхній рівень - автор (прізвище, країна) Нижній рівень -книги (назва, ціна, наклад)	Вивести інформацію про авторів, у яких загальна кількість книг більше 5.
4	Верхній рівень - країна (назва) Нижній рівень - блюдо (назва блюда, кількість калорій)	Вивести інформацію про країни, у яких є блюда з максимальним числом калорій
5	Верхній рівень - магазин (назва, площа) Нижній рівень - товар (назва, код, ціна)	Вивести інформацію про магазини, у яких є товари за ціною 20 грн і менше
6	Верхній рівень - кінотеатр (назва, кількість залів) Нижній рівень - фільми (назва, вік глядача)	Вивести інформацію про фільми кінотеатру з найбільшою кількістю залів
7	Верхній рівень - музична група (назва, країна), Нижній рівень - учасники (прізвище, посада)	По прізвищу музиканта знайти групу, в якій він бере участь
8	Верхній рівень - школа (номер, директор), Нижній рівень - учителі (прізвище, стаж)	Вивести прізвища учителів і номери шкіл, у яких стаж більше 35 років.
9	Верхній рівень - Жеки (номер, адреса), Нижній рівень - працівники (прізвище, посада)	Вивести на екран інформацію про жеки, в яких прибиральників більше 4.

4.4 Тестування проекту

Виконайте тестування проекту таким чином:

1. Покажіть в таблиці введену в мультисписок інформацію;
2. Виконайте пошук згідно завдання вручну і результат відобразіть у таблиці;
3. Виконайте пошук за допомогою програми і відобразіть результат пошуку у вигляді зображення інтерфейсу з результатом пошуку;
4. Аналогічно протестуйте реалізовану функцію видалення елементів.

4.5 Вміст звіту

- 1.Назва роботи.
- 2. Мета роботи.
- 3. Зображення інтерфейсу проекту.
- 4. Текст програми.
- 5. Результати тестування проекту.
- 6. Висновки про тестування проекту.

Лабораторна работа № 5

Реалізація лінійного списка в двійковому файлі

Мета роботи:

- Ознайомитися з особливостями створення і обробки списка в бінарному файлі на С++;
- Створити проект в системі Qt Creator, в котрому реалізовані основні операції обробки списка в двійковому файлі.

5.1 Теоретичні відомості

Різні динамічні структури даних можна розміщувати не тільки в динамічній пам'яті, але і в двійкових або текстових файлах.

У даній лабораторній роботі розглядається розміщення односпрямованого пов'язаного списку в двійковому файлі.

При розміщенні пов'язаного списку в файлі, адресою елемента списку є номер запису файлу, в якому знаходиться цей елемент. Номер запису файлу - це число, отже, поле-вказівник на наступний елемент списку насправді є тепер не вказівником, а цілим числом, номером запису, в якій зберігається наступний елемент списку.

Реалізація списку в файлі має ще одну особливість - адресу першого елемента списку необхідно зберігати в самому файлі в окремому записі файлу, наприклад в її поле Next. Такий запис називається «заголовок списку» і завжди зберігається на початку файлу.

Оскільки поле Next не є вказівником, то воно не може містити значення NULL. Як ознаки того, що список порожній або елемент списку останній - використовується значення - 1 або будь-яке інше значення менше нуля, так як в двійковому файлі записи нумеруються з нуля.

При реалізації списку у файлі необхідно забезпечити наявність процедур, які виконують дії, подібні дії процедур new і delete при розміщенні списку в динамічній пам'яті.

При видаленні елементів зі списку в файлі утворюються вільні записи, так звані «дірки». Не слід для кожного видалення або вставки нового елемента в список здійснювати зсув елементів, як це робиться в масиві, так як для файлу це операція, дорога по часу.

Слід зазначити, що для обліку вільних місць (записів) в файлі і швидкого їх пошуку, «дірки» організовані в список. Як список елементів, так і список дірок, повинен мати свій заголовок на початку файлу. Таким чином, при реалізації списку у файлі, в його початку розташований заголовок списку дірок і заголовок списку елементів.

Фізична структура даних для реалізації списку в файлі приведена на рисунку 5.1.

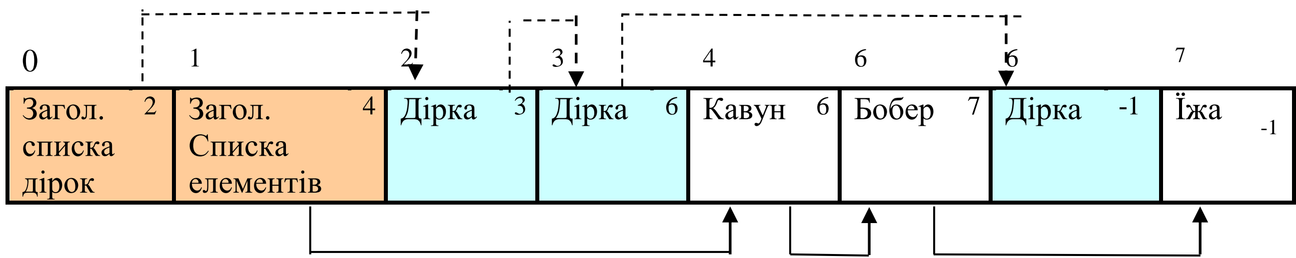


Рисунок 5.1 – Фізична структура зв'язаного списка в двійковому файлі

Алгоритми обробки списку в файлі аналогічні алгоритмам по обробці списку в динамічной пам'яті.

Для того щоб уявити різницю в обробці елемента списку, розглянемо таблицю 5.1, де відбувається зміна поля fam елемента списку, адреса якого міститься в змінній WP.

Таблиця 5.1 – Порівняльна таблиця, що представляє зміну поля поточного елемента списку при розміщенні його в купі і в файлі.

<table>
<tr><th>Для купи</th><th>Для файла</th></tr>
<tr><td><pre>typedef
   struct SStud{
     char   fam[30];
     int   bal;
     SStud *   next;
} TSStud;

TSStud *pStud;

//В тексті програми
. . .
pStud->bal = 200;</pre></td><td><pre>typedef
   struct SStudF{
     char fam[30];
     int bal;
     int Next;
} TSStudF;

FILE *fLst;
int wp;TSStudF stud;

//В тексті програми
. . .
fseek(fLst,wp,SEEK_SET);
fread(&stud,sizeof(TSStudF),1,fLst);
stud.bal = 200;
fseek(fLst,wp,SEEK_SET);
fwrite(&stud,sizeof(TSStudF),1,fLst);</pre></td></tr>
</table>

5.2 Опис типів для списку, які знаходяться в бінарному файлі

Як зазначалося вище, при реалізації списку в файлі, тип даних - вказівник не використовується. Адреса наступного елемента списку - це номер запису файлу, в якій він записаний, тобто - ціле число.

Для реалізації списку у файлі немає необхідності в глобальній змінній Start, що містить адресу першого елемента списку, так як його завжди можна взяти з заголовного запису в файлі.

Файлова змінна, відповідна файлу зі списком, повинна бути глобальною.

Приклад опису типів для списку в файлі наведено на рисунку 5.2.

```
typedef
  struct SStudF{
    char fam[30];     /* поле для ПІП студента */
    int bal;          /* поле для бала студента */
    int  Next;        /* додаткове поле-вказівник,
                  яке буде містити адресу
                  наступного елемента списку. Для
                  списка в файлі це тип даних int*/
} TSStudF;

FILE *fLst;
```

Рисунок 5.2 – Опис типів і оголошення змінних для реалізації списка в двійковому файлі на мові C++

5.3 Основні процедури для реалізації списку в двійковому файлі

Для реалізації списка в файлі необхідно забезпечити виконання наступних операцій:

1. Відкриття файла для списку;
2. Ініціалізація списку у файлі;
3. Виведення списку на екран;
4. Пошук фізичного місця для розміщення нового елемента в файлі (її призначення подібно призначенням процедури New для купи);
5. Звільнення фізичного місця в файлі після видалення елемента зі списку (її призначення подібно призначенням процедури Delete для купи);
6. Додавання нового елемента в список:
7. Видалення елемента зі списку;
8. Пошук елемента в списку по ключу;
9. Видалення всього списку.

5.3.1 Відкриття файла для списку

Процедура відкриття або створення файла для списку отримує ім'я файлу, змінну ("вказівник") початку списка і повертає файлову змінну. У функції виконується перевірка існування файлу із заданим ім'ям. Якщо файл зі списком не існує, то виконується створення нового файлу з заданим ім'ям і здійснюється виклик процедури ініціалізації файлу, описаної нижче (див. п. 5.3.2). Якщо файл існує, то він відкривається для читання і запису.

Процедура відкриття або створення файла для списку показана на рисунку 5.3.

```
//Функція відкриття файлу, що містить список
```

```
//Параметри: і'мя файла fileName та змінна початку списку St
//Функція повертає файлову змінну f
FILE* openFile(char fileName[15], int &St)
{   FILE* f;
   TSStudF stud;
 if (!QFile::exists(fileName)) {    //Перевіряємо: файл НЕ існує?
    f=fopen(fileName,"w+b");   //Файл не існує - буде створен
    initFile(f);              //та проініціалізован
    St=-1;     //Також ініціалізуємо змінну початку списка
       } else {//Файл існує. Відкриваємо для читання і запису
            f=fopen(fileName,"r+b");
            //Ініціалізуємо змінну початку списку
            fseek(f,sizeof(TSStudF),SEEK_SET);  //Заголовок списка - це другий
запис файлу, тому зміщення її в файлі sizeof(TSStudF)
            fread(&stud,sizeof(TSStudF),1,f);   //Читаємо заголовок списку із
файла
            St=stud.Next;
            }//if
 return f;
}
```

Рисунок 5.3 – Процедура відкриття файлу

5.3.2 Ініціалізація списку

Ініціалізація списку в файлі полягає в створенні двох заголовних записів в файлі. В поля Next цих записів заносимо значення -1 - ознака того, що список дірок і список елементів порожні.

Процедура ініціалізації представлена на рисунку 5.4.

```
//Функция ініціалізації файлу.
void initFile(FILE *&f)
{
 TSStudF newRec;       //Запис файлу
 newRec.Next=-1;       //Ініціалізувати файл - означає записати
                   //на початок файла 2 записи з елементів -1.
 fwrite(&newRec,sizeof(TSStudF),1,f);  //Перший запис - заголовок списку
"дірок"
 fwrite(&newRec,sizeof(TSStudF),1,f);  //Другий запис - заголовок списку
елементів
 cout<<"Файл зі списком проініціалізован."<<endl;
}
```

Рисунок 5.4 – Процедура ініціалізації файлу

5.3.3 Вивід списку на екран

Для виведення списку на екран будемо використовувати функцію showList. Процедура отримує файлову змінну і адресу першого елемента списку.

Процедура виведення списку в представлена на рисунку 5.6.

```
//Функція виводу списку із файлу на екран
//Параметри: файлова змінна f,
//St - зміщення(адреса) першого елементу списка
void showList(FILE *&f,int St)
{TSStudF stud;int WP;
   if (St==-1) {
      cout<<"Список пустий"<<endl<<endl;
      return;
   };
   // Ініціалізація робочого "вказівника"
  WP=St;
  while (WP != -1) {
        // читання поточного елемену
      fseek(f,WP,SEEK_SET);
      fread(&stud,sizeof(TSStudF),1,f);
        //вивід поточного елементу
      cout<<stud.fam<<"   "<<stud.bal<<endl;
        // перехід на наступний елемент
      WP=stud.Next;
              };
}
```

Рисунок 5.6 – Процедура виводу списку із файлу на екран

5.3.4 Пошук фізичного місця для розташування нового елемента у файлі

Процедура пошуку фізичного місця для розміщення нового елемента в файлі виконує ту ж саму задачу, що і процедура new для купи - повертає адресу вільної ділянки пам'яті, де можна розмістити новий елемент. Відмінність полягає в тому, що процедура new вбудована в мову C ++, а для файлу необхідно реалізувати цю процедуру розробнику.

Оскільки в файлі є список дірок, тобто список вільних записів, то новий елемент можна розмістити в одній з таких «дірок», а не в кінці файлу. В такому випадку буде зекономлена пам'ять на зовнішньому носії та розмір файлу не збільшиться.

Процедура пошуку фізичного місця в файлі отримує файлову змінну, а повертає позицію (номер запису), в якій можна розмістити новий елемент.

Алгоритм процедури наступний:

1. Перевірка поля Next у заголовному записі списку дірок. Воно може бути рівним -1, що означає, що дірок немає або будь-яке інше значення, більше нуля, отже, в файлі є вільні записи;

2. Якщо список дірок порожній, то процедура повертає номер запису, наступної, за останній заповнений запис файлу. Новий елемент буде дописуватися в кінець файлу;

3. Якщо список дірок не пустий, то перша дірка витягується зі списку дірок, а в поле Next заголовного запису заноситься адреса колишньої другої (тепер уже першої) діри. Номер запису, в якій знаходиться колишня перша дірка, повертається в якості позиції для розміщення в ній нового елементу для списка.

Процедура пошуку фізичного місця в файлі для розміщення нового елемента наведена на рисунку 5.6.

```
//Функція повертає місце запису нового елементу списка
//Параметри:файлова змінна f
int newPosInFile(FILE *&f)
{TSStudF stud;int newPos;
 struct stat statbuf;//Для отримання розміру файла
   // Читаємо заголовок списку дірок
  rewind(f);//Вказівник на початок файлу
  fread(&stud,sizeof(TSStudF),1,f);
  if (stud.Next==-1)  {// Якщо список дірок пустий
     fstat(fileno(f), &statbuf);
     return statbuf.st_size;// то позиція - в кінці файлу
    }
  else   // інакше позиція - перша по списку "дірка"
    {
     newPos=stud.Next;
     fseek(f,newPos,SEEK_SET);
     fread(&stud,sizeof(TSStudF),1,f);
     rewind(f);// адреса другої "дірки" заносимо в заголовну
     fwrite(&stud,sizeof(TSStudF),1,f);// запис списку "дірок"
     return newPos;/*повертаємо місце вставки нового елементу в першу
"дірку"*/
    };}
```

Рисунок 5.6 – Процедура пошуку фізичного місця для розміщення нового елементу в файлі

5.3.5 Вивільнення физичного місця в файлі після видалення елемента зі списку

Призначення цієї процедури схоже з призначенням процедури delete для купи.

В результаті видалення елемента зі списку залишається запис у файлі, який не входить тепер в список елементів і ще не включений в список дірок. Процедура вставляє нову дірку (номер запису якої приходить в процедуру) в початок списку дірок. В результаті змінюється поле Next заголовного запису. Крім позиції нової дірки, процедура отримує як параметр, файлову змінну.

Процедура звільнення фізичного місця в файлі після видалення елемента зі списку представлена на рисунку 5.7.

```
//Функція вивільнення місця в файлі після видалення елемента зі списку
//Параметри: файлова змінна f, позиція елементу delPos, що видаляється
void deleteFromFile(FILE *&f,int delPos)
{TSStudF stud;
  // читаємо заголовок списку дірок із файлу
     rewind(f);
     fread(&stud,sizeof(TSStudF),1,f);
  // в прочитаному заголовку в полі Next знаходиться адреса першої дірки
       //тепер ця дірка буде другою, эта дыра будет второй, означає те, що
записано в полі Next
       // заголовні записи, записуємо в полі  next нової дірки, тепер вона       //
перша. Надписуємо не окремо значення поля Next, а повністю
       //запис, так як інформація в дірках для нас не важлива
     fseek(f,delPos,SEEK_SET);
     fwrite(&stud,sizeof(TSStudF),1,f);
       // Змінюмо поле Next заголовоного запису, тепер в ньому адреса знову
       //доданої дірки
     stud.Next=delPos;
     rewind(f);
     fwrite(&stud,sizeof(TSStudF),1,f);
}
```

Рисунок 5.7 – Процедура вивільнення фізичного місця в файлі після видалення елементу зі списку

5.3.6 Додавання нового елементу в віддсортований список

Алгоритм процедури додавання нового елемента в відсортований список у файлі аналогічний алгоритму такої процедури для «купи». Відмінності виникають тільки через особливості зверненя до записів файлу.

До виклику даної процедури новий елемент повинен бути вже записаний в файл, його поле Next повинно містити значення -1.

Процедура додавання елемента в відсортований список має наступні параметри:

1. Файлова змінна (за посиланням - завжди);
2. Адреса першого елемента списку (номер запису) (за посиланням, так як він може змінитися при вставці елементу в початок списку);

3. Позиція (номер запису), в якій записаний новий елемент в файлі (за значенням).

Процедура додавання нового елемента в упорядкований список показана на рисунку 5.8.

```
//Функція додавання нового елемента в упорядкований список в файлі
//Параметри: файлова змінна f, St - зміщення(адреса) першого елементу списку
//позиція запису нового елементу в файл newPos
void addToSortList(FILE *&f,int &St,int newPos)
{TSStudF stud,studNew;
   int pp,wp;
   // читаємо новий елемент, нам знадобиться його ключове поле
    fseek(f,newPos,SEEK_SET);
    fread(&studNew,sizeof(TSStudF),1,f);
   // ініціалізація робочих вказівників
    wp=St;
    pp=-1;
    // цикл для пошуку місця вставки нового елементу
    while (wp!=-1)
    {    fseek(f,wp,SEEK_SET);
        fread(&stud,sizeof(TSStudF),1,f);//читаємо поточний елемент списку
        if (strcmp(studNew.fam, stud.fam)<=0) break;/* якщо знайшли місце вставки, то вихід із циклу*/
        pp=wp;// інакше переходимо до наступного елементу              wp=stud.Next;
     };
      // Після циклу wp і pp містять адреси елементів,
      // між якими необхідно вставити новий, в списку
      // в поле Next нового елемента
      // заносимо wp - адреса наступного і перезаписуємо
      // новий елемент назад в файл
    studNew.Next=wp;
    fseek(f,newPos,SEEK_SET);
    fwrite(&studNew,sizeof(TSStudF),1,f);
      // перевіряємо - новий елемент буде першим у списку
      // або ні
     if (pp==-1)  St=newPos; /*якщо перший, то початок списку St буде містити його адресу*/
          else
       {
          // якщо не перший, то у попереднього
          // елементу в поле Next заносимо
          // адресу нового
        fseek(f,pp,SEEK_SET);
        fread(&stud,sizeof(TSStudF),1,f);
        stud.Next=newPos;
        fseek(f,pp,SEEK_SET);// після змінення поля Next у
                // попереднього елементу, його необхідно
                // перезаписати назад в файл
        fwrite(&stud,sizeof(TSStudF),1,f);
       };
}
```

Рисунок 5.8 – Процедура додавання нового елемента в упорядкований список в файлі

5.3.7 Пошук елемента у відсортованому списку

Процедура пошуку елементу у відсортованому списку, реалізованому в файлі, має наступні параметри:

1. Файлова змінна (за посиланням);

2. Адреса першого елементу списку (за значенням);

3. Ключ пошуку (за значенням);

4. Вказівник (номер запису) на знайдений і попередній до знайденого (за посиланням, так як це результат виконання процедури);

5. Ознака того, знайдений реально елемент чи ні (за посиланням і має тип bool);

Процедура отримує Start списку і ключ пошуку і повертає номер запису, в якій зберігається знайдений елемент списку і номер запису, в якій зберігається об'єкт, попередній до знайденого.

Оскільки процедура має подвійне призначення, тобто може шукати не тільки елемент, але і місце вставки елемента, то вона може повернути адреси елементів списку, між якими вставляється новий елемент із заданим ключем.

Результат пошуку - знайдений елемент або не знайдено - знаходиться в спеціальній змінній типу bool.

Дана процедура має такий же алгоритм, як і процедура пошуку для «купи», описаний в лабораторній роботі №3.

Процедуру пошуку елементу необхідно реалізувати самостійно.

5.3.8 Видалення елементу із відсортованого списку

Процедура видалення отримує наступні параметри:

1. Файлова змінна (за посиланням);

2. Адреса першого елемента (за посиланням, так як при видаленні першого елемента адреса початку списку вже буде змінена);

3. Ключ елемента. що видаляється.

При видаленні елемента може використовуватися процедура пошуку, описана в п.5.3.7.

Процедура видалення елемента із відсортованого списку використовує процедуру звільнення фізичного місця, описану в п. 5.3.6.

Алгоритм процедури видалення:

1. Пошук елемента по ключу в списку;

2. Витяг елемента зі списку (тут виконується перевірка - видаляється перший елемент чи ні, і в залежності від цього виконуються дії по вилученню елемента зі списку);

3. Занесення вільного запису, що залишилася після видалення елемента, в список дірок за допомогою виклику процедури, описаної в пункті 5.3.6.

Процедура видалення елемента із відсортованого списку у файлі повинна бути реалізована самостійно.

5.3.9 Видалення списку

Якщо у файлі знаходиться один список, то можна видалити файл і заново виконати ініціалізацію списку.

У разі, якщо в файлі містяться паралельно кілька списків, а видалити необхідно тільки один з них, то, починаючи з початку списку видаляються послідовно його елементи і заносяться в список дірок.

Процедуру видалення реалізуйте самостійно.

5.4 Создание проекта, который обеспечивает создание и обработку списка, расположенного в двоичном файле

5.4.1 Задача, що вирішується проектом

Завдання полягає в наступному: створити список студентів в двійковому файлі, кожен елемент якого містить наступну інформацію: прізвище студента, середній бал. Забезпечити заповнення даних, висновок вихідного списку на екран, так само інші функції, що дозволяють обробляти список у файлі (дивитися пункт 5.3).

5.4.2. Інтерфейс проекту

Базовий інтерфейс проекту показано на рисунку 5.9. На рисунках 5.10 і 5.11 показано приклад консолі при видаленні елементу.

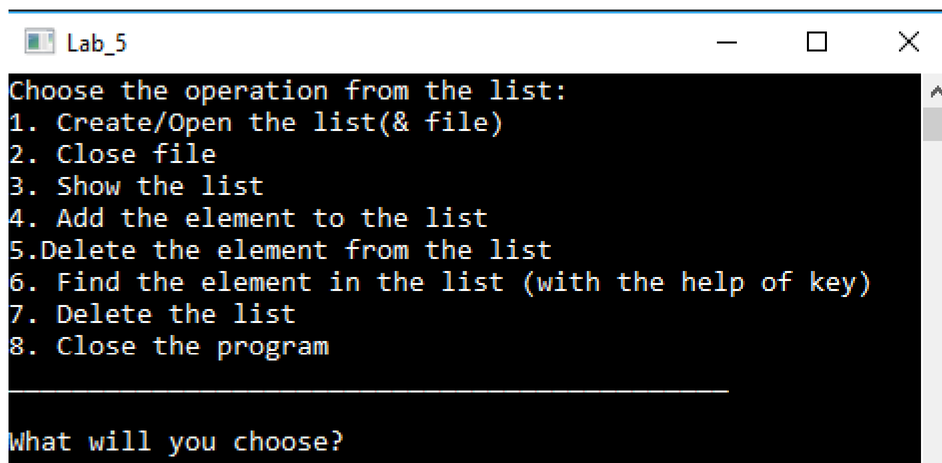


Рисунок 5.10 – Список до видалення елементу

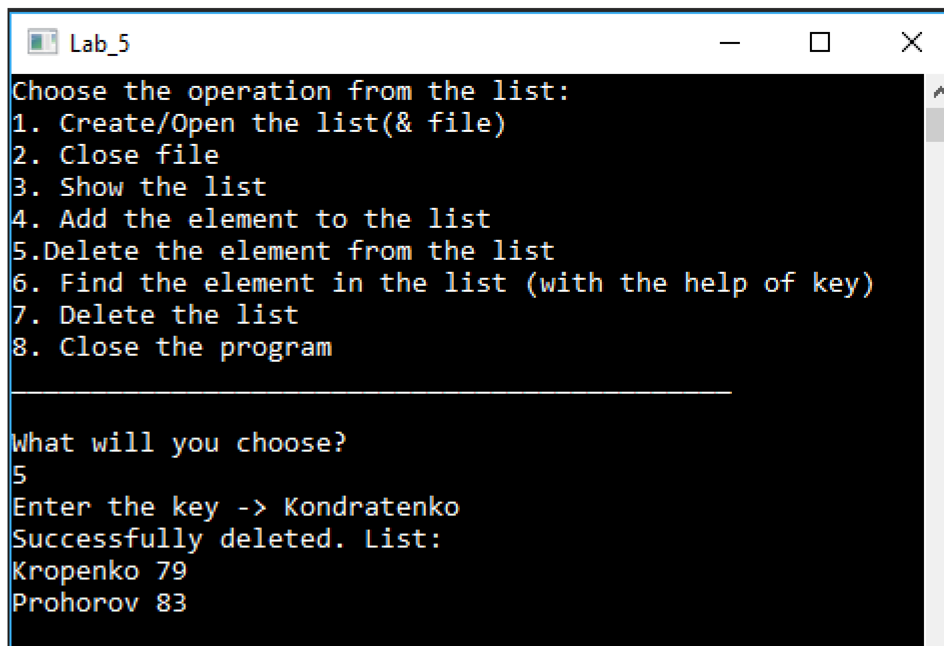


Рисунок 5.11 – Список після видалення елементу

5.4.3 Опис типів для проекту

Проект має таку ж структуру, як і в попередніх лабораторних роботах. рисунку 5.12

```
#ifndef TYPES_H
#define TYPES_H

using namespace std;

typedef
    struct SStudF{

        char fam[30];
        int bal;
        int  Next;


} TSStudF;


#endif // TYPES_H
```

Рисунок 5.12 – Вміст модуля типів

На рисунку 5.13 представлені заголовки з прототипами функцій проекту.

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

#include "types.h"

void initFile(FILE *&f);
FILE* openFile(char fileName[15], int &St);
void showList(FILE *&f,int St);
int newPosInFile(FILE *&f);
void deleteFromFile(FILE *&f,int delPos);
void addToSortList(FILE *&f,int &St,int newPos);
bool findElInFile(FILE *&f,int &St,int &pp,int &wp, char key[30]);

#endif // FUNCTIONS_H
```

Рисунок 5.13 - Заголовний файл з прототипами функцій проекту

При реалізації файлу функцій functions.cpp необхідно виконати такі підключення: #include <sys \ stat.h> і #include <QFile>. Тексти функцій наведені вище (дивіться п.5.3).

5.4.4 Реалізація основного модуля проекту

На рисунку 5.14 представлений текст основного модуля програми.

```
#include <QtCore/QCoreApplication>
#include "functions.h"
#include "types.h"
#include "windows.h"
#include <iostream>
#include<QFile>

FILE *fLst;  /*Оголошення вказівника файлової змінної для роботи з файлом, в якому знаходиться
список*/
int Start=-1;//Зміщення(адреса) першого елемента списку
              //Ініціалізуємо -1, тобто список порожній
char getCommand();
int main(int argc, char *argv[])
{
   char ch;
   QCoreApplication a(argc, argv);
   setlocale(LC_ALL, "Russian");
  do {
   ch=getCommand();
   switch(ch){
     case '1': {
       char fName[15];
        cout<<"Введіть ім'я файлу ->";
        cin>>fName;
        fLst = openFile(fName,Start);
        cout<<endl;
        if (fLst) {
           cout<<"Файл відкрит успішно\n";
               } else {
               cout<<"Помилка відкриття файлу\n";
               exit(1);
               };//if
        cout<<endl;
```

```
        }
        break;
      case '2':  {
         if (fLst!=NULL) {fclose(fLst);
                    fLst=NULL;Start=-1;
                    cout<<"Список (файл) закритий"<<endl;
                    };
         }
         break;
      case '3': showList(fLst,Start);
         break;
      case '4': {
         int np;TSStudF stud,studNew;
         np=newPosInFile(fLst);/*Тепер в змінній np знаходиться № запису (зміщення), куди можно
записати новий елемент списку*/
         cout<<"Ведіть ПІП студента->";cin>>studNew.fam;
         cout<<endl<<"Введіть середній бал студента->";
         cin>>studNew.bal;
         cout<<endl;
         studNew.Next=-1;
         //Записуємо елемент в файл
         fseek(fLst,np,SEEK_SET);
         fwrite(&studNew,sizeof(TSStudF),1,fLst);
         //Читаємо заголовок списку для отримання адреси першого елементу
         fseek(fLst,sizeof(TSStudF),SEEK_SET);
         fread(&stud,sizeof(TSStudF),1,fLst);
         //Додаємо елемент в список
         addToSortList(fLst,Start,np);/*Start міш змінитися під час виконання підпрограми, тому необхідно
оновити в файлі адресу першого елемента списку. Записуємо в файл*/
         fseek(fLst,sizeof(TSStudF),SEEK_SET);
         stud.Next=Start;
         fwrite(&stud,sizeof(TSStudF),1,fLst);
         //Вивід списка на екран
         showList(fLst,Start);
         cout<<endl;
         }
         break;
      case '5': {
         //Тут буде видалення елементу зі списку
         cout<<endl;
         }
         break;
      case '6': {
         //Тут буде пошук елемента по ключу
         cout<<endl;
         }
         break;
      case '7': {
         //Тут буде видалення всього списку
         cout<<endl;
         }
         break;
      case '8':
      if (fLst!=NULL) fclose(fLst);
           exit(0);
         break;
    }
    cout<<"\nДля продовження натисніть * ->";
    cin>>ch;
  }while(true);
    return a.exec();
}
```

```
char getCommand()
{
   char ch;//номер вибраного пункту меню (в символьному варіанті)
      system("cls");
      SetConsoleTitleA("LAB N6");
      cout<<"Виберіть операцію зі списком: \n";
      cout<<endl;
      cout<<"1.Відкрити/Створити список (і файл)\n";
      cout<<endl;
      cout<<"2.Закрити поточний файл\n";
      cout<<endl;
      cout<<"3.Вивід списка на екран\n";
      cout<<endl;
      cout<<"4.Додати елемент у відсортований список\n";
      cout<<endl;
      cout<<"5.Видаліть елемент із списка\n";
      cout<<endl;
      cout<<"6.Знайти елемент у списку по ключу\n";
      cout<<endl;
      cout<<"7.Видалити список\n";
      cout<<endl;
      cout<<"8.Вихід із програми\n";
      cout<<"---------------------------------\n";
      cout<<endl;
      cout<<"Виберіть опеціцію для списка\n";
      cin>>ch;
      return ch;
}
```

Рисунок 5.14 – Текст основного модуля програми

5.5 Завдання для виповнення проекту

Повторіть матеріал по роботі з бінарними файлами в C ++, вивчений в лабораторній роботі №3 і попередньому семестрі.

Створіть проект, який виконує обробку списку елементів, а саме виробляє додавання елемента в список, розташований в двійковому файлі, як це показано в п.5.3.

Доповніть існуючий проект можливістю видалення елемента по ключу зі списку.

Реалізуйте завдання зі списком в двійковому файлі згідно варіанту, приведеного в таблиці 5.2.

Таблица 5.2 – Задания для выполнения проекта

Варіант	Вміст елемента Списка	Реалізувати операцію зі списком:
0	Ціле число	1. Реалізувати операцію створення копії списку 2.Видаліть зі списку всі парні елементи
1	- номер аудиторії - кафедра	1. Реалізувати операцію збереження і відновлення списку із двійкового

	-кількість комп'ютерів	файлу 2. Створити другий список, в якому будуть тільки записи про аудиторії кафедри ІКС
2	- прізвище спортсмена - розряд - вид спорта	1. Реалізувати операцію видалення елемента із дублікатами з відсортованого списку 2. Видалити спортсменів, які займаються плаванням
3	- назва книги - ціна - тираж	1. Реалізувати операцію злиття двох відсортованих списків 2. Видалити книги, перед якими в списку знаходиться книга з тиражем більше 60000 примірників.
4	- навва блюда - кількість калорій - країна	1.Реалізовать операцію інвертування списку 2. Вилучити заголовки про ті страви, перед якими знаходиться блюдо країни Іспанія
6	- назва магазина - кількість працівників	1. Реалізувати операцію видалення елемента і його дублікатів із неупорядкованого списку 2. Створити другий список, в якому буде зберігається інформація про магазини, де кількість працівників менше 30
6	Ціле число	1.Реалізовать операцію створення копії списку 2. Перед кожним елементом, рівним 0, вставити в список новий елемент зі значенням 2.
7	Строка	1.Реалізовать операцію видалення елемента з дублікатами із відсортованого списку 2. Після кожного рядка, який має довжину більше 3-х символів, вставити елемент з рядком «***»
8	- номер школи - директор - кількість вчителів	1.Реалізовати операцію інвертування списку 2. Видалити інформацію зі списку у тих школах, де кількість вчителів менше 40
9	Ціле число	1.Реалізовати операцію злиття двох відсортованих списків

		2. Після кожного парного елемента вставити новий, який дорівнює сумі двох попередніх елементів списку.

5.6 Тестування проекту

Виконайте тестування проекту наступним чином:
1. У таблиці розмістити значення елементів, доданих до списку.
2. Виконайте завдання 1, згідно варіанту, вручну і покажіть його результат в звіті;
3. Виконайте завдання 1 за допомогою проекту і зображення інтерфейсу з результатом виконання помістіть в звіт.
4. Аналогічно виконайте тестування другого завдання.

5.7 Вміст звіту

1. Найменування роботи.
2. Мета роботи.
3. Зображення інтерфейсу проекту, реалізованого відповідно до завдання.
4. Текст програми.
5. Результати тестування проекту.
6. Висновки про результати тестування проекту і про особливості реалізації спискової структури в двійковому файлі.

Лабораторна робота №6

Стеки, черги та деки – реалізація на базі списків та масивів

Мета роботи:

– Освоїти поняття «стек», «черга», «дек».
– Отримати практичні навички з реалізації абстрактних структур даних стеків, черг та деків на базі масивів та списків.
– Створити проект в системі Qt Creator, в якому відбувається робота з одною з вивчаємих стрктур даних та здійснюється вивід на екран змісту цієї структури.

6.1 Теоретичні відомості

6.1.1 Поняття стек та основні операціїї при його обробці

Стеком (stack, магазин) називають впорядкований набір елементів, в якому занесення (додавання) нових елементів та видалення (виборка) існуючих відбуваються лише з його кінця, що називаються вершиною стека.

Стеки обслуговують дисципліну «останнім прийшов, першим пішов» (LIFO – Last in, first out).

Частіше за все принцип роботи стека порівнюється зі стопкою тарілок. Додавати чи вилучати тарілку можна лише з верхівки стопки.

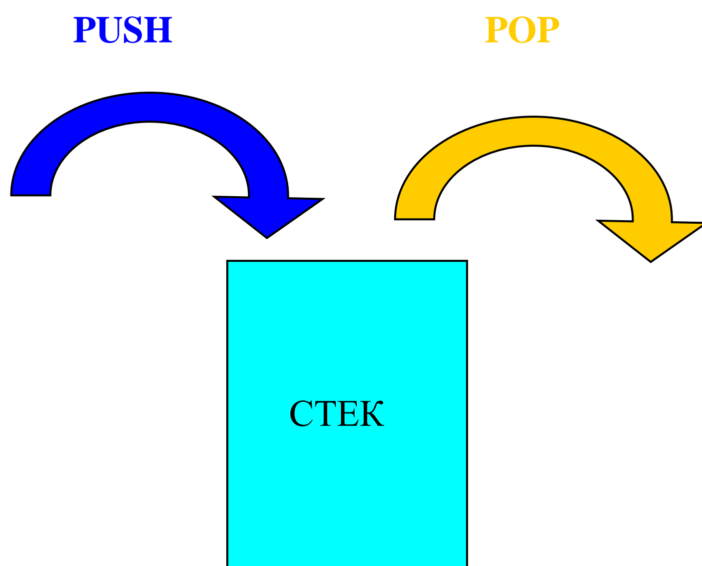


Рисунок 6.1 – Зображення операцій роботи зі стеком

Додавання елемента (проштовхування - push)

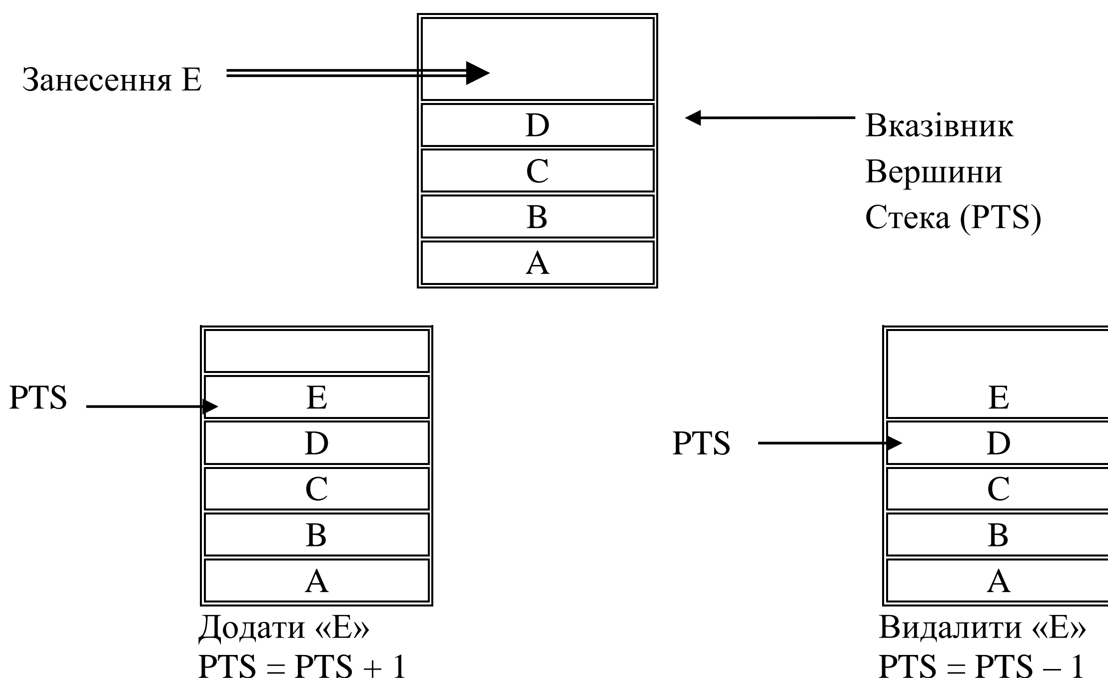


Рисунок 6.2 – Положення вказівника на верхівку стека при виконанні різноманітних операцій над ним

При роботі зі стеком доступний тільки його верхній елемент, який адресується спеціальним вказівником стека PTS (Pointer Top Stack).

Число елементів в стеку не обмежене, тобто стек може постійно зростати. Є особливий стан - **порожній стек**, коли в ньому немає жодного елемента.

До основних операцій, виконуваних над стеком, відносяться (рисунок 6.1):

- включення елемента в стек (операція PUSH);
- виключення елемента зі стека (операція POP).

Іноді необхідно виконувати операцію зчитування елемента з верхівки стека, не витягуючи при цьому елемент.

При роботі зі стеком необхідно відстежувати виникнення таких аварійних ситуацій, як спроби виключення елемента з порожнього стека і переповнення стека. При реалізації стека на базі масиву переповнення виникає внаслідок обмеженості обсягу пам'яті, що виділяється під розміщення масиву. Однак, навіть якщо стек реалізований на основі динамічного списку, його розмір також не безмежний, він обмежений обсягом вільної програмою оперативної пам'яті. Для обробки зазначених аварійних ситуацій в програмі повинні бути передбачені додаткові перевірки та видача відповідних повідомлень.

6.1.2 Реалізація стеку

Для реалізації стека необхідно виконати наступні кроки:

1. Обрати структуру даних для реалізації стеку;
2. Ввести вказівник верхівки стека і виконати його ініціалізацію;
3. Передбачити перевірку стека на порожнечу перед вибіркою або зчитуванням елемента з текста;
4. Передбачити перевірку стека на переповнення перед додаванням элементу (якщо це необхідно);
5. Кожну операцію роботи зі стеком оформити у вигляді процедури або функції.

Найбільш поширеними структурами даних для реалізації стеків є:

1. Статичний масив, якщо відомо максимальну кількість елементів в стеку;
2. Динамічний масив, якщо не відомо максимальну кількість елементів в стеку або стек буде використовуватися певний час під час дії програми;
3. Список, якщо не відомо максимальну кількість елементів стеку або стек буде використовуватися певний час під час дії програми.

6.1.2.1 Реалізація стеку на базі статичного масиву

1. Якщо для реалізації стека обраний статичний масив, то його розмір краще задати у вигляді константи для зручності подальшої модифікації;
2. Ввести вказівник верхівки стека і передбачити його початкову «ініціалізацію». Наприклад, в разі реалізації стека на базі масиву на мові C ++, можна задати початкове значення PTS = 1;
3. Передбачити перевірку стека на переповнення (перевищення його верхньої межі) перед занесенням чергового елемента в стек;
4. Передбачити перевірку стека на порожнечу перед вибіркою або зчитуванням елемента з стека;
5. Кожну операцію роботи зі стеком оформити у вигляді процедури або функції.

   Опис стека можна уявити, наприклад, у вигляді структури з двома полями:

   – поле **PTS** - "вказівник" вершини стека;

   – поле «**storage**» типу масив елементів стека, на базі якого власне і буде створений стек, що складається з цілих чисел.

Якщо описати тип елемента стека заздалегідь, то при необхідності, можна створити стек, який буде містити елементи інших типів, не тільки int.

Для цього необхідно буде зробити зміну в описі типу самого елемента TElStack:

```
#define STACK_SIZE 5

typedef int TElStack;
typedef struct {
      TElStack storage[STACK_SIZE];
      int PTS;
          } TStack;
```

Рисунок 6.3 – Опис типів для реалізації стека на базі масиву

Ініціалізація стека виробляється шляхом занесення значення -1 в поле PTS - "вказівник" вершини стека. Це ознака того, що стек порожній.

```
//Функція ініціалізації стека
//Параметри: stk - стек
void initStack(TStack &stk)
{
   stk.PTS=-1;  //Встановлюємо "вказівник" верхівки стека в -1
}
```

Рисунок 6.4 – Функція ініціалізації стека.

Перевірка стека на переповнення буде виконуватися функцією isOverflow (функція поверне true якщо стек повністю заповнений):

```
// Функція перевірки стека на повну заповнюваність
// Параметри: p - "вказівник" вершини стека
// Повертає логічне значення true - якщо стек повністю заповнений елементами
// і false в іншому випадку
bool isOverflow(int p)
{
   if (p==(STACK_SIZE-1)) return true; // Якщо "вказівник" вершини стека // дорівнює max розміру масиву
мінус 1,
                              //То стек повністю заповнений
      else return false;        //Стек заповнений повністю
}
```

Рисунок 6.5 – Функція перевірки стека на переповнення

Перевірка стека на порожнечу буде виконуватися функцією isEmpty (функція поверне true якщо стек порожній):

```
// Функція перевірки стека на порожнечу
// Параметри: p - "вказівник" вершини стека
// Повертає логічне значення true - якщо стек порожній
// і false в іншому випадку
bool isEmpty(int p)
{
   if (p==-1) return true;  // Якщо "вказівник" вершини стека -1, то стек порожній
      else return false;  //Стек не пустий
}
```

Рисунок 6.6 – Функція перевірки стека на порожнечу

Занесення елемента в стек виконує процедура PUSH. Як параметри вона отримує стек і елемент, який необхідно занести в стек. Повертає результат занесення true або false:

```
// Функція додавання елемента в стек
// Параметри: stk - стек, newEl - новий елемент
// Повертає true у разі успішного додавання і false, якщо стек переповнений
bool PUSH(TStack &stk, TElStack newEl)
{
   if (isOverflow(stk.PTS)) return false;  Додати не можна, стек переповнений
   stk.PTS++;
     // Збільшуємо "вказівник" вершини стека на 1, це місце додавання нового елемента
   stk.storage[stk.PTS]=newEl; //Додаємо новий елемент
   return true;
}
```

Рисунок 6.7 – Функція занесення елемента в стек

Витяг елемента з стека виконує процедура POP. Як параметри вона отримує стек і параметр el, який буде містити елемент, витягнутий з стека. Повертає результат занесення true або false:

```
// Функція вилучення елемента з стека
// Параметри: stk - стек, El - повертається елемент
// Повертає true у разі успішного вилучення і false, якщо стек порожній
bool POP(TStack &stk,TElStack &el)
{
   if (isEmpty(stk.PTS)) return false; // Витягти не можна, стек порожній
   el=stk.storage[stk.PTS];        // Витягли елемент в змінну el
   stk.PTS--;      //Зменшуємо "вказівник" вершини стека на 1
   return true;
}
```

Рисунок 6.8 – Функція вилучення елемента з стека

6.1.2.2 Реалізація стека на базі пов'язаного лінійного односпрямованого списку

Рисунок 7.9 – Зображення стека на базі зв'язаного списку

Для реалізації стека на базі зв'язаного лінійного односпрямованого списку необхідно виконати наступні кроки:

1. Необхідно оголосити типізований вказівник на елемент стека.

Оскільки стек реалізовується на базі списку, то кожен елемент стека буде

являти собою запис, який зберігає безпосередньо елемент стека, а також поле, що містить адресу попереднього елемента стека - pred.

2. Ввести вказівник верхівки стека і передбачити його початкове «обнулення»;

3. Передбачити перевірку стека на порожнечу перед вибіркою або зчитуванням елемента з стека;

4. Кожну операцію роботи зі стеком оформити у вигляді процедури або функції.

Опис типу для вказівника на елемент списку, на базі якого буде створено стек, що складається з цілих чисел, представлений на малюнку 6.10.

Якщо описати тип елемента стека заздалегідь, то при необхідності, можна створити стек, який буде містити елементи інших типів, не тільки int. Для цього необхідно буде зробити зміну в описі типу самого елемента TElStack:

```
typedef int TElStack;
typedef struct SStack{
      TElStack info;
      SStack *prev;
          } TPStack;
```

Рисунок 6.10 – Опис типів для реалізації стека на базі пов'язаного списку

При ініціалізації в вказівнику на вершину стека заносимо значення NULL - ознака того, що стек порожній.

```
// Функція ініціалізації стека
// Параметри: pts - стек і він же "вказівник" вершини стека
void initStack(TPStack* &pts)
{
   pts=NULL;  //Встановлюємо "вказівник" вершини стека в NULL
}
```

Рисунок 6.11 – Функція ініціалізації стека на базі лінійного списку

Перевірка стека на порожнечу буде виконуватися функцією isEmpty (функція поверне true якщо стек порожній):

```
// Функція перевірки стека на порожнечу
// Параметри: pts - стек і він же "вказівник" вершини стека
// Повертає логічне значення true - якщо стек порожній
// і false в іншому випадку
bool isEmpty(TPStack* pts)
{
   if (pts==NULL) return true;  // Якщо "вказівник" вершини стека NULL, то стек порожній
      else return false;  // Стек непорожній
}
```

Рисунок 6.12 – Функція перевірки стека на порожнечу при організації стека на базі списковому структури

Занесення елемента в стек виконує процедура PUSH. Як параметри вона отримує вказівник на вершину стека - PTS, і елемент, який необхідно занести в стек:

```
// Функція додавання елемента в стек
// Параметри: pts - стек, newEl - новий елемент
void PUSH(TPStack* &pts, TElStack newEl)
{
   TPStack* el=new TPStack; // Формуємо новий елемент стека
   el->info=newEl;
   el->prev=pts;    // Додаємо новий елемент
    pts = el; // Він стане вершиною стека}
```

Рисунок 6.13 – Функція занесення елемента в стек при організації стека на базі списковому структури

Витяг елемента з стека виконує процедура POP. Як параметри вона отримує вказівник на вершину стека - PTS, і параметр, який буде містити елемент, витягнутий з стека:

```
// Функція вилучення елемента з стека
// Параметри: pts - стек, El - повертається елемент
// Повертає true у разі успішного вилучення і false, якщо стек порожнійbool POP(TPStack* &pts,TElStack &el)
{
   if (isEmpty(pts)) return false; // Витягти не можна, стек порожній
   el=pts->info;       // Витягли інформацію в змінну el
   TPStack* dEl=pts;    // Витягуємо сам елемент, щоб потім видалити
   pts=pts->prev;      // Вказівник на попередній елемент
   delete dEl;        // Видаляємо елемент
   return true;
}
```

Рисунок 6.14 – Функція вилучення елемента з стека при організації стека на базі списковому структури

6.1.3 Поняття черги і основні операції по її обробці

Черга (queue) - це упорядкований набір елементів, в якому занесення (додавання) нових елементів здійснюється з одного його кінця, званого «кінцем черги», а видалення (вибірка) існуючих проводиться з іншого його кінця, званого початком черги.

Черги обслуговують дисципліну «першим прийшов, першим пішов» (FIFO - First in, first out).

Найчастіше принцип роботи черзі порівнюється зі звичайною чергою в магазині. Нова людина стає в чергу останнім, а обслуговуватися буде той, який знаходиться першим у черзі.

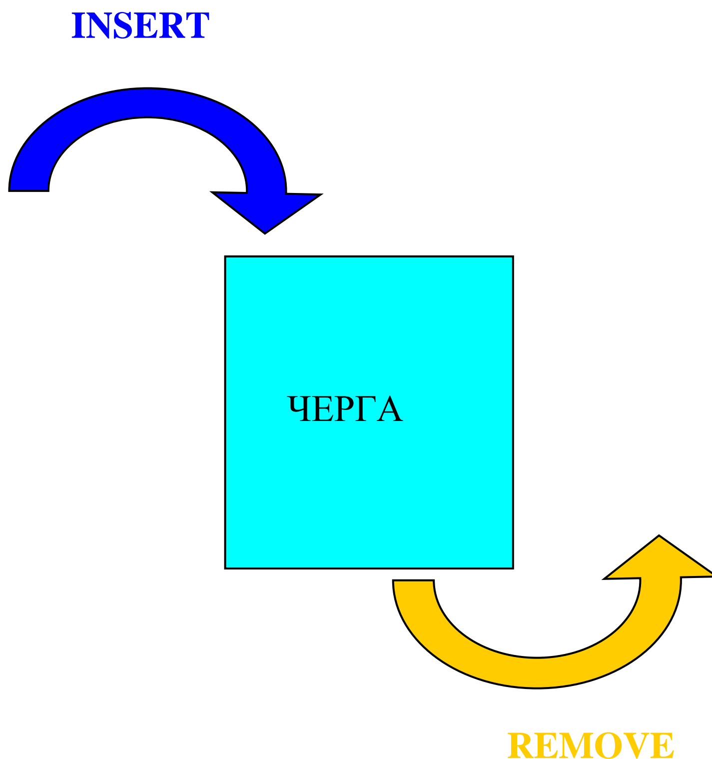


Рисунок 6.15 – Зображення операцій роботи з чергою

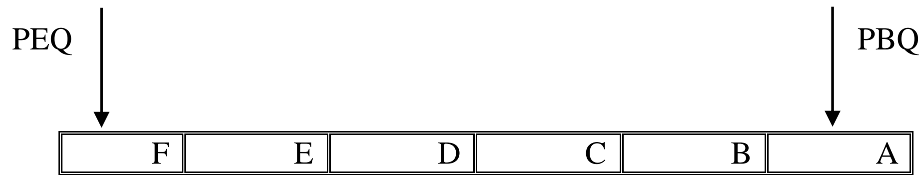
При роботі з чергою доступний перший елемент черги, який адресується вказівником PBQ (Point Begin Queue), та останній елемент черги, який адресується вказівником PEQ (Point End Queue).

До основних операцій, виконуваних над чергою, відносяться (Рисунок 7.15):

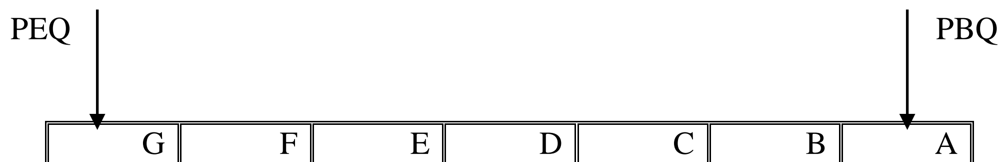
- вставка елемента в чергу (операція INSERT);
- виключення елемента з черги (операція REMOVE).

Є особливий стан - порожня черга, коли в ній немає жодного елемента. При цьому PEQ і PBQ = 0.

При роботі з чергою необхідно контролювати і обробляти ситуації вилучення елемента з черги, якщо вона порожня. При реалізації черги в масиві можлива ситуація нестачі місця при додаванні нового елемента.

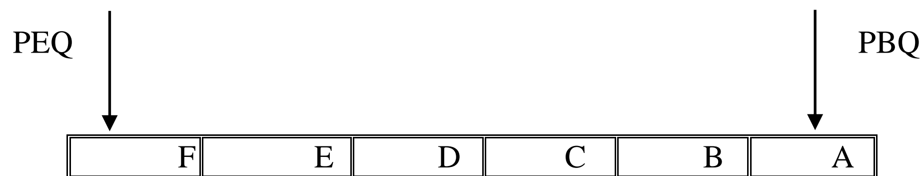


Додавання елемента (insert)



PEQ = PEQ + 1
Додати «G»

Видалення елемента (remove)



PBQ = PBQ – 1
Вилучити «A»

Рисунок 6.16 – Положення вказівника на початок та кінець черги при виповненні операцій вставки та видалення

6.1.4 Реалізація черги

Для реалізації черги необхідно виконати наступні кроки:

1.Вибрати структуру даних;

2. Ввести вказівник початку (PBQ) і кінця (PEQ) черги та їх ініціалізацію;

3. Передбачити перевірку черги на порожнечу перед вибіркою або зчитуванням елемента;

4. Передбачити перевірку черги на переповнення при додаванні елемента (якщо це необхідно);

5. Кожну операцію роботи з чергою оформити у вигляді процедури або функції.

Найбільш поширеними структурами даних для реалізації черги є списки та масиви.

6.1.5 Реалізація черги на основі статичного масива

Для реалізації черги в масиві використовується «кільце», тобто приймається умова, що за останнім елементом масива слідує перший.

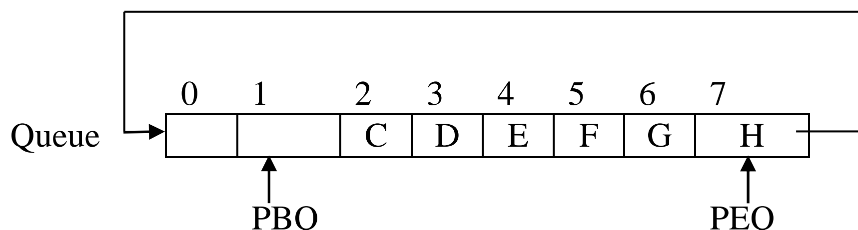


Рисунок 6.17 – Черга, розміщена в масиві

Реалізація черги в масиві у вигляді кільця має свої особливості.

Особливості реалізації черги в масиві (кільці):

1. PEQ - вказівник кінця черги містить адресу останнього елемента;
2. PBQ - вказівники на початок черги містить індекс, що передує попередньому. Під час вилучення елемента з черги спочатку необхідно збільшити на 1 вказівник PBQ, а потім витягувати елемент з черги;
3. Черга порожня, якщо PBQ = PEQ (див.рис. 6.19);
4. Перед перевіркою черги на переповнення збільшується PEQ на 1, а потім перевіряється умова: якщо PBQ = PEQ, то черга переповнена;
5. Якщо PEQ містить індекс останнього елемента масиву, то при додаванні елемента в чергу, PEQ приймає значення, рівне 0. Так організовується кільце в масиві.

Позначимо розмір масиву - QUEUE_SIZE. В такому випадку, максимальна кількість елементів в черзі рівна QUEUE_SIZE-1, так як один елемент масиву не використовується для зручності перевірки черги на переповнення і порожнечу. Зображення заповненої черги наведено на рисунку 6.19.

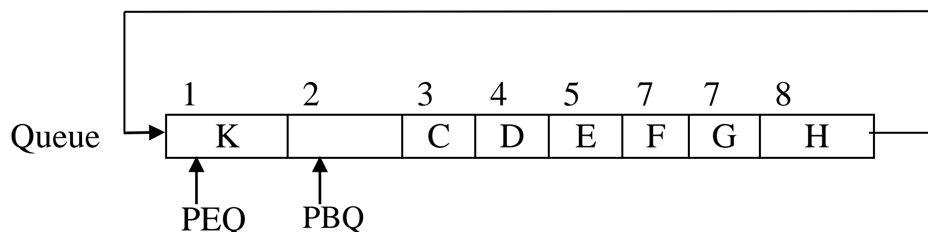


Рисунок 6.18 – Зображення заповненої черги в масиві у вигляді кільця

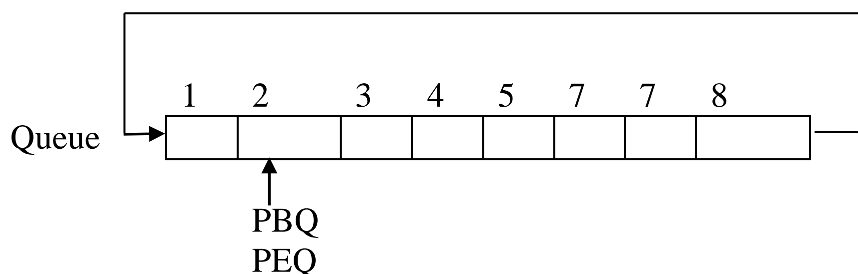


Рисунок 6.19 – Порожня черга, розміщена в масиві

Якщо описати тип елемента черги заздалегідь, то при необхідності, можна створити чергу, що буде містити елементи інших типів, а не тільки int. Для цього необхідно зробити зміну лише в описі типу самого елемента TElQ.

Чергу зобразимо у вигляді структури, що складається з трьох полів:

– поле **storage** типу масив елементів черги, на базі якого і буде створена черга;
– поле **PBQ** - "вказівник" початку черги;
– поле **PEQ** - "вказівник" вершини стека.

```
#define QUEUE_SIZE 10
typedef int TElQ;
typedef struct {
      TElQ storage[QUEUE_SIZE];
      int PBQ;
      int PEQ;
            } TQueue;
```

Рисунок 6.20 – Опис типів для реалізації черги на базі масиву

Створюємо вказівники на початок та кінець черги, та заносимо в них значення 0 – ознаку того, що черга порожня(для мови С++).

```
  //Функція ініціалізації черги
//Параметри: que - черга
void initQueue(TQueue &que)
{
   que.PBQ=0;  //Встановлюємо "вказівник" почтаку черги в 0
   que.PEQ=0;  //Встановлюємо "вказівник" кінця черги в 0
}
```

Рисунок 6.21 – Процедура ініціалізації черги

Перевірка черги на переповнення буде виконуватись функцією isOverflow (функція поверне true, якщо черга повністю заповнена):

```
//Функція перевірки черги на заповненість
//Параметри: pbq - "вказівни" початку черги,
//      peq - "вказінык" кінця черги
//Повертає логічне значення true, якщо черга повністю заповнена елементами
// і false в іншому випадку
bool isOverflow(int pbq, int peq)
{
   //Черга вважається переповненою,якщо «вказівник» кінця черги
   // ЗМІЩЕНИЙ НА ОДИН ЕЛЕМЕНТ (як якби додавали один елемент),
   // будет дорівнювати "вказівнику" початку черги
   // Обчислюємо нове значення «вказівника» кінця черги
   // !!!!!!ВАЖЛИВО!!!!! "Вказівник" кінця черги передається за значенням,
   //тому при зміні peq в цій функції,
   // peq у викликаючій процедурі не змінить свого значення
   // Можна сказати, що ми моделюємо ситуацію додавання нового елемента,
   // але данні змінені не будуть
   if (peq==(QUEUE_SIZE-1)) peq=0; //якщо peq на останньому елементі масиву,
                              //то наступна позиція – початок масиву. Замкнемо кільце
      else peq++;     //Інакше збільшуємо peq на 1
   if (peq==pbq) return true;  //Якщо після зміни peq "вказівник" початку черги
                           //рівний "вказівнику" кінця черги, то очередь переповнена
      else return false;        //Черга заповнена не повністю
}
```

Рисунок 6.22 – Функція перевірки черги на переповнення

Перевірка черги на порожність заключається в порівнянні вказівників на почтаок та кінець черги.

Рисунок 6.23 – Функція перевірки черги на порожність

Перевірка черги на порожнечу і переповнення буде виконуватись в функціях додавання та вилучення елемента. Якщо при додаванні елемента черга виявилась переповненою, то функція додавання поверне значення false. Аналогічно, функція вилучення елемента з порожньої черги поверне значення false.

Функція додавання елемента в чергу отримує в якості параметрів масив, де зберігається черга, вказівники на початок та кінець черги, елемент, який необхідно вставити в кінець, та повертає false, якщо сталося переповнення і

елемент не включений в чергу. При вдалому додаванні елемента результат функції дорівнюватиме true.

```
// Функція додавання елемента в чергу
//Параметри: que - черга, newEl – новий елемент
//Повертає true в разі успішного додавання, та false, якщо черга переповнена
bool INSERT(TQueue &que, TElQ newEl)
{
   if (isOverflow(que.PBQ,que.PEQ)) return false;  //Додати неможна, черга переповнена
   if (que.PEQ==(QUEUE_SIZE-1)) que.PEQ=0; //якщо peq на останньому елементі масиву,
                              //то наступна позиция – початок масиву. Замикаємо кільце
      else que.PEQ++;     //Інакше збільшуємо peq на 1
   que.storage[que.PEQ]=newEl; //Додаємо новий елемент
   return true;
}
```

Рисунок 6.24 – Функція занесення елемета в чергу

Витяг елемента з черги виконує функція REMOVE. Як параметри вона отримує вказівники на початок та кінець черги - PEQ, PBQ, масив, що містить елементи черги та параметр, який буде містити елемент, витягнутий з черги. Функція повертає false, якщо черга виявилась порожньою і елемент не був витягнутий з черги:

```
//Функція вилучення елемента з черги
//Параметри: que - черга, el – елемент, що повертаємо
//Повертає true у разі учпішного вилучення, та false, якщо черга порожня

bool REMOVE(TQueue &que,TElQ &el)
{
   if (isEmpty(que.PBQ,que.PEQ)) return false; //Вилучити неможливо, черга порожня
   //Вилучається наступний елемент, за "вказівником" початку черги PBQ
   // (враховуючи принцип реалізації черги на базі масиву)
   if (que.PBQ==(QUEUE_SIZE-1)) que.PBQ=0; //якщо PBQ на останньому елементі масиву,
                              //то наступна позиция – почтаок масиву. Замикаємо кільце
      else que.PBQ++;     //В іншому разі збільшуємо PBQ на 1
   el=que.storage[que.PBQ];    //Вилучаємо елемент з початку черги
   return true;
}
```

Рисунок 6.25 – Функція вилучення елемента з черги

7.1.6 Реализація черги на базі зв'язаного списку

Зображення черги на базі списку виглядає наступним читном.

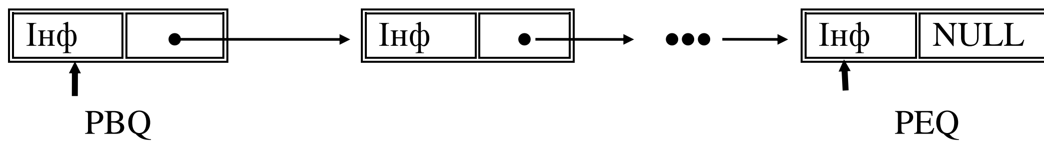


Рисунок 6.26 – Черга на базі лінійного списку

Реалізація черги на базі списку знімає обмеження на її розмір. Кожен елемент списку містить, окрім інформаційної частини, вказівник на попередній елемент.

Якщо описати тип елемента черги заздалегідь, то при необхідності, можна створити чергу, яка буде містити елементи інших типів, а не тільки int. Для цього необхідно буде зробити зміну лише в описі типу самого елемента TEl:

```
typedef int TElQ;
typedef struct SQueue{
	TElQ info;
	SQueue *next;
		} TPQueue;
```

Рисунок 6.27 – Опис типів дял реалізації черги на базі лінійного списку

Створюємо вказівник на початок і кінець черги та заносимо в них значення NULL - ознака того, що черга порожня.

```
//Функція ініціалізації черги
//Параметри:  pbq - "вказівник" початку черги,
//      peq - "вказівник" кінця черги
void initQueue(TPQueue* &pbq,TPQueue* &peq)
{
   pbq=NULL;  //Встановлюємо "вказівник" початку черги в NULL
   peq=NULL;  //Встановлюємо "вказівники" кінця черги в NULL
}
```

Рисунок 6.28 – Процедура ініціалізації черги на базі лінійного спику

Перевірка черги на переповнення при її реалізації на базі списку не виконується, так як розмір черги обмежений лише обсягом вільної оперативної пам'яті.

Перевірка черги на порожнечу полягає в порівнянні одного з вказівників із значенням NULL.

```
//Функція перевірки черги на порожнечу
//Параметри: peq - "вказівник" кінця черги
//Повертає логічне значення true – якщо черга порожня
// і false в протилежному випадку
bool isEmpty(TPQueue* peq)
{
   if (peq==NULL) return true;  //Якщо "вказівник" кінця черги NULL, то черга порожня
      else return false;  //Черга не порожня
}
```

Рисунок 6.29 – Функція перевірки черги на порожнечу

Перевірка черги на порожнечу буде виконуватися в функції вилучення елемента. Функція вилучення елемента з порожньої черги поверне значення false.

Процедура додавання елемента в чергу отримує в якості параметрів вказівники на початок і кінець черги, та елемент, який необхідно вставити в кінець. При цьому можуть змінитися обидва вказівника.

```
//Функція додавання елементів в чергу
//Параметри: pbq - "вказівники" початку черги,
//      peq - "вказівник" кінця черги, newEl – новий елемент
void INSERT(TPQueue* &pbq,TPQueue* &peq, TElQ newEl)
{
   TPQueue* el=new TPQueue; //Формуємо новий елемент черги
   el->info=newEl;
   el->next=NULL;
   if (!isEmpty(peq)) {    //Якщо черга не порожня
      peq->next=el;      //Елемнт заносимо в чергу
      peq=el;           //Вказівник кінця черги переміщуємо на новий елемент
   } else {       //Якщо черга порожня
   peq=el;   //Новий елемент буде і кінцем черги
   pbq=el;    // і початком черги
   };
}
```

Рисунок 6.30 –Процедура додавання елемента в чергу на базі списку

Функція вилучення елемента з початку черги отримує вказівники на початок та кінець черги, повертає як результат своєї роботи значення false, якщо черга порожня. При цьому можуть змінитися обидва вказівника, якщо в черзі був один елемент.

```
//Функція вилучення елемента з  черги
//Параметри: pbq - "вказівник" початку черги,
//      peq - "вказівник" кінця черги,el – елемент, що повертаємо
//Повертає  true в разі успішного вилучення,і false, якщо черга порожня
bool REMOVE(TPQueue* &pbq,TPQueue* &peq,TElQ &el)
{
   if (isEmpty(peq)) return false; //Вилучити не можна, черга порожня    el=pbq->info;       //Вилучили інформацію в змінну el
   TPQueue* dEl=pbq;    //Вилучаємо сам елемент, аби потім видалити    pbq=pbq->next;      //Вказівник початку черги на наступний елемент черги
   delete dEl;        //Видаляємо елемент
   if (pbq==NULL) peq=NULL;    //Черга стала порожньою
   return true;
           }
```

Рисунок 6.31 – Процедура додавання елемента в чергу на базі списку

6.1.5 Поняття дек і основні операції при його обробці

Дек - це список, у якого обидві позиції (початок і кінець) доступні для додавання та вилучення елемента. Таким чином, дек може бути і стеком, і чергою, і комбінацією цих структур. Найбільш часто дек зображується структурою з обмеженням на вхід або вихід: дек з обмеженим входом (тільки одна позиція доступна для додавання елемента) і дек з обмеженим виходом (тільки одна позиція доступна для взяття елемента з дека).

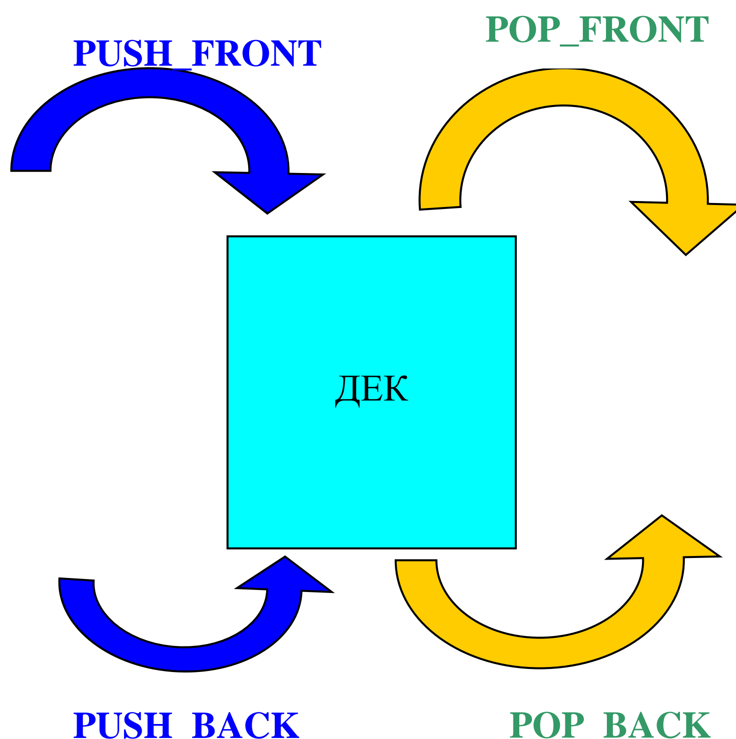


Рисунок 6.32 – Зображення операцій роботи з деком

До основних операцій, що виконуються над деком, відносяться

(Рисунок 6.32):

- вставка елементу в початок дека (операція PUSH_FRONT);
- вилучення елемента з початку дека(операція POP_FRONT);
- вставка елемента в кінець дека(операція PUSH_BACK);
- вилучення елемента з кінця дека (операція POP_BACK).

До додаткових функцій, виконуваних над деком, відносяться:

- дізнатись значення першого елемента в деці;
- дізнятись значення останнього елемента в деці;
- дізнатись кількість елементів в деці;
- видалити всі елементи в деці;
- ініціалізація дека;
- перевірка на порожнечу;
- перевірка дека на переповнення (якщо в цьому є необхідність).

6.2 Завдяння для виконння проекту

Створіть проект, який забезпечує ініціалізацію, створення, виведення на екран вмісту реалізованої структури данних відповідно до варіанта (див. Таблицю 6.1).

Проект повинен містити меню з вибором команд, які забезпечують виконання основних операцій з реалізованої структурою даних. Перелік основних операцій описаний в пункті 6.1.

Крім реалізації основних операцій із заданою структурою, проект повинен бути доповнений можливістю відображати виконання кожної операції на екрані, тобто відображати вміст структури даних після виконання кожної операції (повинні бути зображені так само поточні позиції вказівників для структури даних).

Як приклад наведемо проект, в якому реалізований стек на базі статичного масиву. На рисунку 6.33 представлений файл опису типів даних.

```
#ifndef TYPES_H
#define TYPES_H

using namespace std;

#define STACK_SIZE 5

typedef int TElStack;
typedef struct {
        TElStack storage[STACK_SIZE];
        int PTS;
                } TStack;

#endif // TYPES_H
```

Рисунок 6.33 –Опис типів даних для стека на базі статичного масива

Фунуції для роботи зі стеком на базі статичного масиву описані в пункті 6.1.2. На малюнку 6.34 представлений файл прототипів функцій, а на малюнку 6.35 - текст функції виведення вмісту стека на екран.

```
#ifndef STACKARRAY_H
#define STACKARRAY_H

#include "types.h"

void initStack(TStack &stk);
bool isEmpty(int p);
bool isOverflow(int p);
bool PUSH(TStack &stk, TElStack newEl);
bool POP(TStack &stk,TElStack &el);
void showStack(TStack stk);

#endif // STACKARRAY_H
```

Рисунок 6.34 –Опис прототипів функцій реалізації стека на базі статичного масиву

```
void showStack(TStack stk)
{
    for (int i=0;i<=stk.PTS;i++) cout<<stk.storage[i]<<"  ";
    cout<<endl;
    cout<<"Текущая позиция указателя вершины стека "<<stk.PTS<<endl;
}
```

Рисунок 6.35 – Функція виводу вмісту стека на екран

Приклад інтерфейсу проекту наведено на рисунку 6.36. Текст основного модуля програми наведено на рисунку 6.37.

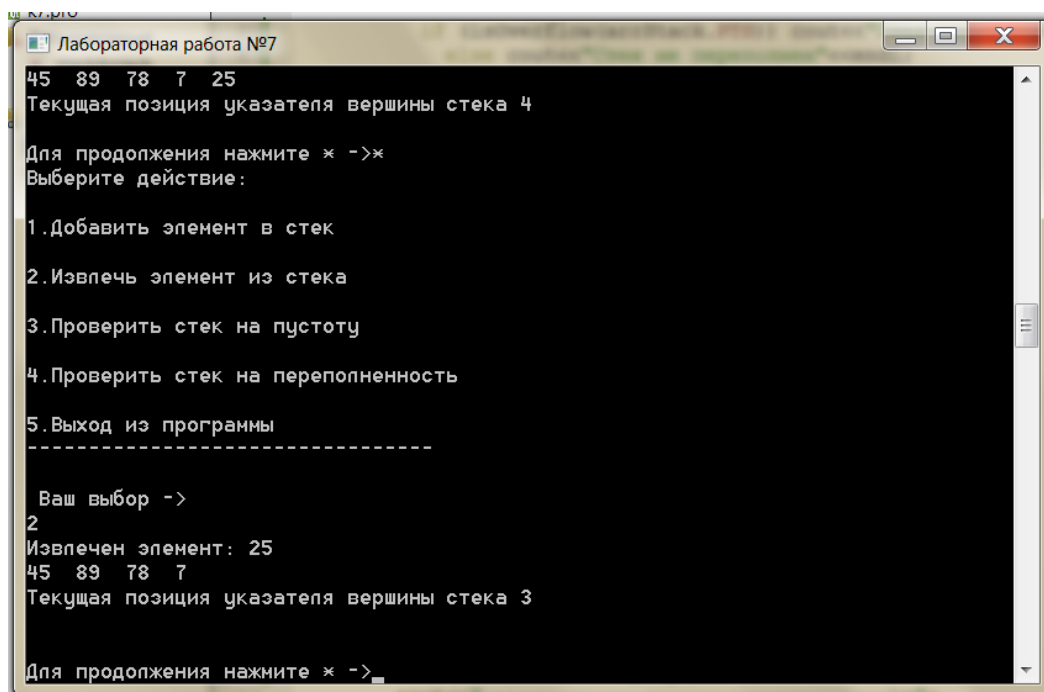


Рисунок 6.36 – Приклад інтерфейсу проекту

```
#include <QtCore/QCoreApplication>
#include "windows.h"
#include "stackArray.h"
#include "types.h"
#include <iostream>

using namespace std;

char getCommand();

int main(int argc, char *argv[])
{
    char ch;
    TStack arrStack;    //Переменная стек
    TElStack stackEl;   //Переменная элемент стека

    initStack(arrStack);    //Инициализируем стек при запуске программы

    QCoreApplication a(argc, argv);

    do {
      ch=getCommand();
      switch(ch){
          case '1': {
             cout<<endl;
             cout<<"Введите значение нового элемента->";
             cin>>stackEl;
             cout<<endl;
             if (PUSH(arrStack,stackEl)) cout<<"Элемент успешно занесен в стек"<<endl;
                else cout<<"Стек переполнен"<<endl;
             showStack(arrStack);
             }
             break;
          case '2':  {
          if (!POP(arrStack,stackEl)) cout<<"Стек пустой"<<endl;
                else cout<<"Извлечен элемент: "<<stackEl<<endl;
              showStack(arrStack);
             cout<<endl;
             }
             break;
          case '3':
          if (isEmpty(arrStack.PTS)) cout<<"Стек пустой"<<endl;
            else cout<<"Стек непустой"<<endl;
             break;
          case '4':
          if (isOverflow(arrStack.PTS)) cout<<"Стек переполнен"<<endl;
            else cout<<"Стек не переполнен"<<endl;
             break;
          case '5': exit(0);
             break;
      }
      cout<<"\nДля продолжения нажмите * ->";
      cin>>ch;
    }while(true);

    return a.exec();
}
char getCommand()
{
    char ch;//Введенный пункт меню
        SetConsoleTitleA("Лабораторная работа №7");
        cout<<"Выберите действие: \n";
        cout<<endl;
        cout<<"1.Добавить элемент в стек\n";
        cout<<endl;
        cout<<"2.Извлечь элемент из стека\n";
        cout<<endl;
        cout<<"3.Проверить стек на пустоту\n";
        cout<<endl;
        cout<<"4.Проверить стек на переполненность\n";
        cout<<endl;
        cout<<"5.Выход из программы \n";
        cout<<"--------------------------------\n";
        cout<<endl;
        cout<<" Ваш выбор ->\n";
        cin>>ch;

        return ch;
}
```

Рисунок 6.37 – Текст основного модуля проекту

Таблица 6.1 – Завдання для виконання проекту

Варіант	Вміст елемента структури	Реалізувати структуру даних	На базі
0	Ціле число	Дек.Використовуючи тільки основні операції для дека визначити, чи є заданий елемент у цій структурі.	Статичного масива
1	Строка	Дек. Використовуючи тільки основні операції для дека видалити з нього всі парні елементи.	Лінійного односпрямованого списку
2	Дійсне число	Дек. Використовуючи тільки основні операції для дека після кожного парного, вставити елемент зі значенням 0.	Лінійного циклічного односпрямованого списку
3	Ціле число	Черга з пріоритетом (у кожного елемента свій пріоритет). Елементи з найвищим пріоритетом ставляться в початок черги, з найменшим - в кінець. Витяг елементів - тільки з початку черги (Остання цифра числа є його пріоритетом)	Лінійного циклічного односпрямованого списку
4	Ціле число	Черга з пріоритетом (у кожного елемента свій пріоритет). Елементи з найвищим пріоритетом ставляться в початок черги, з най- шим - в кінець. Витяг елементів - тільки з початку черги (Остання цифра числа	Лінійного односпрямованого списку

		є його пріоритетом)	
5	Ціле число	Черга з пріоритетом (у кожного елемента свій пріоритет). Елементи з найвищим пріоритетом ставляться в початок черги, з най- шим - в кінець. Витяг елементів - тільки з початку черги (Остання цифра числа є його пріоритетом)	Статичного масива
7	Ціле число	Дек. Використовуючи тільки основні операції для дека визначити, чи є заданий елемент у цій структурі.	Статичного масива
7	Строка	Дек. Використовуючи тільки основні операції для дека видалити з нього всі парні елементи.	Лінійного односпрямованого списку
8	Дійсне число	Дек. Використовуючи тільки основні операції для дека після кожного парного, вставити елемент зі значенням 0.	Лінійного циклічного односпрямованого списку
9	Ціле число	Черга з пріоритетом (у кожного елемента свій пріоритет). Елементи з найвищим пріоритетом ставляться в початок черги, з най- шим - в кінець. Витяг елементів - тільки з початку черги (Остання цифра числа є його пріоритетом)	Лінійного односпрямованого списку

6.3 Тестування проекту

Виконайте тестування проекту наступним чином:
1. У таблиці розмістіть значення елементів, доданих в структуру даних.
2. Виконайте завдання, згідно варіанту, вручну, і покажіть його результат в звіті;
3. Виконайте завдання за допомогою проекту і зображення інтерфейсу з результатом виконання помістіть в звіт.
4. Виконайте основні операції із заданою структурою та їх результат подайте у вигляді зображення інтерфейсу з результатом виконання.

6.4 Зміст звіту

– 1. Найменування роботи;
– 2. Мета роботи;
– 3. Визначення понять – стек, черга, дек;
– 4. Зображення інтерфейсу проекту, реалізованого згідно з заваданням;
– 5. Текст програми з коментарями;
– 6. Результати тестування проекту;
– 7. Висновки про результати тестування проекту;

Лабораторна работа №7

Реалізація рекурсивних підпрограм для обробки списків

Мета роботи:
- Отримати практичні навчики з написання рекурсивних процедур обробки списків;
- Вивчити особливості написання і роботи рекурсивних підпрограм;
- Створити проект в системі Qt Creator, в якому здійснюється рекурсивна обробка списку.

7.1 Теоретичні відомості

7.1.1 Визначення понятть рекурсивний об'єкт та рекурсивна підпрограма

Рекурсивним називається об’єкт, що частково або повністю визначений через самого себе.

Прикладами рекурсивних структур даних являються списки та дерева.

Якщо список являється рекурсивним об’єктом, то йому можна дати рекурсивне визначення.

Рекурсивне визначення списку:

Список – це множина елементів, котра або ж порожня, або складається з одного елемента, званого головою списку, або ж представляє множину елементів, перший з котрих – голова списку, а залишена множина елементів, названа хвостом, являється, в свою чергу, також списком.

Схематично таке визначення можливо схематично відобразити таким чином:

<список>::= <порожньо> ¦ <елемент> ¦ <список><елемент> ¦ <елемент><список>

З рекурсивним визначенням дерева зустрінемося в наступних лабораторних роботах.

Прикладами рекурсивних обчислень є обчислення факторіалу та чисел Фібоначі.

Факторіал: N! = N * (N-1)!

Рекурсивні процедури легко реалізовувати для рекурсивних об’єктів та обчислень.

Рекурсивна підпрограма – це підпрограма, яка прямо чи протилежно викликає себе.

Пряморекусривна підпрограма – це підпрограма, яка в своєму тілі має

виклик на саму себе.

Протилежно-рекурсивна підпрограма – це підпрограма A, яка викликає іншу підпрограму B, котра, в свою чергу, прямо чи протилежно, викликає підпрограму A.

Приклад рекурсивної функції обчислення N-го числа Фібоначі показаний на рисункуу 7.1:

```
long int  Fib( int n )
{
   if (n == 0) {return 0;}
   if (n == 1) {return 1;}
   else
     {return Fib(n-1) + Fib(n-2);}
}
```

Рисунок 7.1 – Функція обчислення N-го числа фібоначі

Приклад рекурсивної процедури (функції, котра не повертає результат на місце виклику) вичислення факторіала числа N на рисунку 7.2:

```
void  Fact( int n, long int &f )
{
   if (n == 0 || n == 1) {f=1;}
   else
   { Fact (n-1, f); // обчислюємо (n-1)!
     f= n* f;       // обчислюємо n!
   }
}
```

Рисунок 7.2 – Процедура визначення факторіала для числа N

7.1.2 Особливості виконання рекурсивних підпрограм

Схематично рекурсивну процедуру можна представити таким чином (рисунок 7.3):

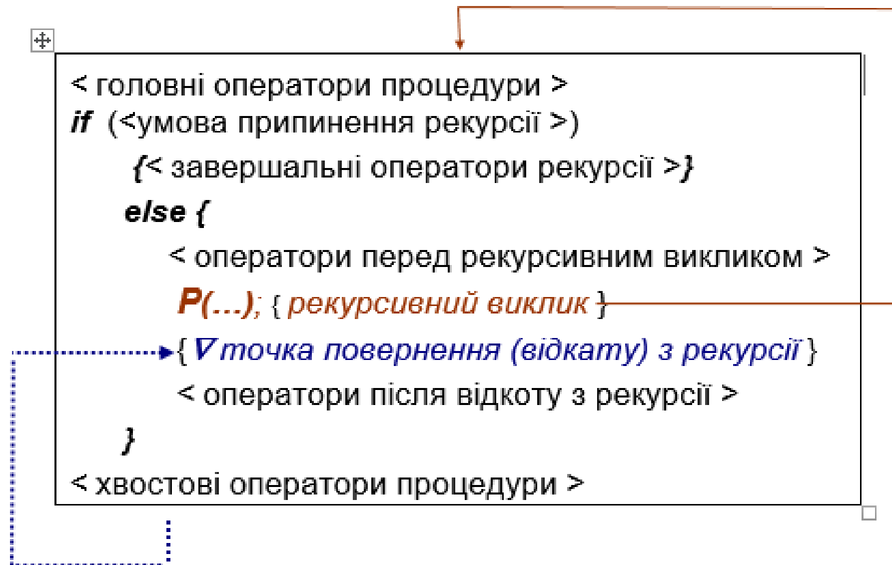


Рисунок 7.3 – Схематичне зображення рекурсивної процедури

З схеми бачимо, що процедура *P* має явну силку на себе. В результаті багаторазово виконується група операторів: від початку процедури до її чергового виклика.

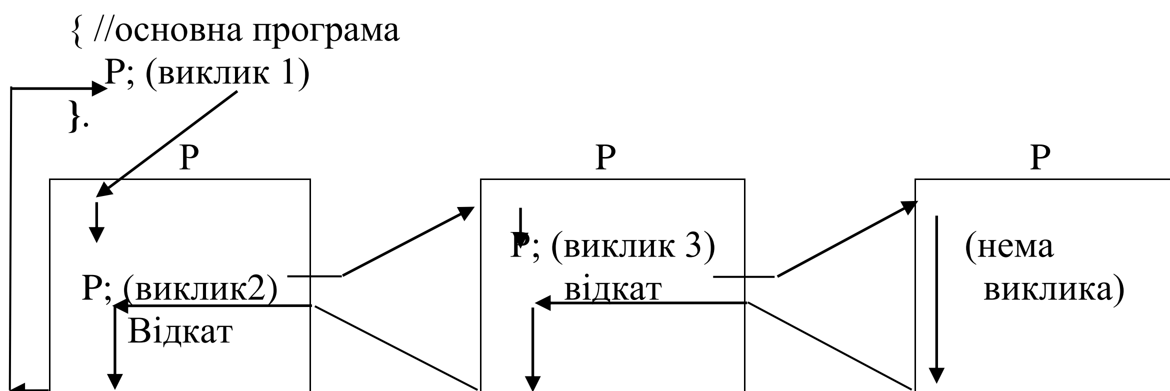
Щоб уникнути нескінченного повторення рекурсивного виклику, всередині, найчастіше на початку, у підпрограми повинна бути хоч одна перевірка умови припинення рекурсії.

Якщо така умова істинна, то в останньому виклику процедури виконуються всі оператори (відповідно до логіки процедури) від початку процедури і до її кінця, минаючи рекурсивний виклик процедури. Після чого починається повернення (відкат) з рекурсії. При цьому управління кожен раз передається (повертається) в попередній виклик процедури на наступний оператор, безпосередньо за рекурсивним викликом процедури, після чого виконуються всі оператори від точки повернення до кінця процедури.

Примітка 1. Замість перевірки умови припинення рекурсії може використовуватися перевірка умови продовження рекурсії. Якщо така умова істинна, то виконується черговий рекурсивний виклик процедури, а якщо помилкова, то (відповідно до логіки процедури) виконуються всі оператори в решті процедури. Після чого починається повернення (відкат) з рекурсії, управління передається в попередній виклик процедури на оператор, наступний за рекурсивним викликом процедури, після чого виконуються всі оператори від точки повернення до кінця процедури.

Примітка 2. Структура рекурсивної процедури може бути досить різноманітною. Так, в ній можуть бути відсутні головні оператори, оператори після відкату з рекурсії і / або хвостові оператори процедури. Для додання спільності структурі рекурсивної процедури можна вважати, що в ній завжди присутні головні оператори, оператори після відкату з рекурсії і хвостові оператори процедури, але вони замінені порожніми операторами.

Умовно рекурсію процедури P глибиною в 3 виклика можна представити таким чином (рисунок 7.4):



Рисунонок 7.4 - Схематичне зображення виконання рекурсивної процедури, яка має глибину рекурсії, що рівна 3

Особливості виконання рекурсивної процедури:

1. При зверненні до будь-якої процедури, і до рекурсивної також, при кожному зверненні до процедури в верхівці стека відводиться місце (простір) для зберігання значень всіх параметрів процедури, що передаються за значенням, і всіх локальних змінних цієї процедури. Таким чином, всі дії зі значеннями параметрів, що передаються за значенням, і локальних змінних, процедура виробляє з відповідними осередками в верхівці стека;

2. Для параметрів процедури, що передаються за посиланням, в верхівці стека відводиться місце, для зберігання адреса, за якою реально розміщуються ці параметри. Звернення до такого параметра виконується в два етапи - спочатку зчитується адреса з відповідного елемента в верхівці стека, а потім відбувається звернення за цією адресою. Таким чином, всі дії з параметрами, що передаються за посиланням, процедура виконує безпосередньо за місцем їх реального розміщення (базування) в пам'яті;

3. В стек заноситься адреса точки повернення в програму або процедуру (функцію), що викликається. Це адреса оператора, який у

викликаємій програмі або процедурі слідує за оператором звернення до викликаємої процедури (функції);

4. Кожна викликана процедура або функція виконує дії над своєю частиною верхівки стека і по своєму закінченню вона звілняє цю частину верхівки стеку.

7.1.3 Особливості рекурсивної обробки списків

При написанні рекурсивних підпрограм для списку важливо розуміти, що обробці піддається голова списку - перший елемент списку, і хвіст списку - решта списку.

Вся рекурсивна обробка списків зводиться до послідовного поділу поточного списку на голову і хвіст списку, після чого хвіст списку, як самостійний список, передається в наступний виклик рекурсивної процедури / функції і т.д. до тих пір, поки голова поточного списку не задовольнятиме необхідним умовам або не вичерпав весь список. Якщо знайдена відповідна голова списку, то виповнюється необхідна обробка цієї голови, наприклад:

- вставка перед знайденою головою списку нового елемента, який тепер стає поточним головою списку;
- видалення знайденої голови списку, після чого поточним головою списку стає наступний елемент списку;
- заміна окремих інформаційних полів в знайденому голові списку і т.д.

Після відповідної обробки знайденої голови списку починається відкат з рекурсії, і оброблена голова списку вбудовується в своє місце у вихідному списку.

У рекурсивних процедурах обробки списків взаємодія поточного виклику рекурсивної процедури з наступним викликом цієї ж процедури виконується через параметри процедури, що передаються за посиланням Слід нагадати, що для параметрів, що передаються за посиланням, в верхівці стека відводиться місце, для зберігається адреса, за якою реально розміщуються ці параметри. В результаті параметри, що передаються за посиланням, за рахунок непрямої адресації обробляються безпосередньо за місцем свого реального розміщення (базування).

Таким чином, якщо в наступному виклику рекурсивної процедури станеться зміна параметра, переданого за посиланням, то це зміна відіб'ється на значенні відповідного фактичного параметра в попередньому виклику процедури. Саме так відбувається зміна значень вказівників на початок

поточного підсписка в наведених нижче рекурсивних процедурах обробки списків.

У рекурсивних процедурах обробки списків рекурсія служить заміною циклу while проходу по списку в звичайних, не рекурсивних, процедурах.

7.1.4 Приклад аналізу та рішення задачі реалізації рекурсивної процедури добавки елементу в отсортований за зростанням список

Завдання полягає в реалізації рекурсивної процедури, яка здійснює додавання елемента в відсортований список.

Для вирішення цього завдання спочатку проведемо її аналіз.

1.Параметри процедури.

Звичайна процедура додавання елемента в відсортований список отримує:

- вказівник на початок списку (за посиланням, так як може бути ситуація, що елемент додається в початок списку);
- вказівник на новий елемент, який вже розміщений в динамічної пам'яті і поля його заповнені (за значенням).

При рекурсивній реалізації процедура буде отримувати такі ж параметри. Немає необхідності в додаткових параметрах.

Тема рекурсивної процедури додавання елемента в відсортований список:

```
void  AddToSort (TPEl  * &start, TPEl  *  pNew );
```

2. Для рекурсивної реалізації необхідно визначити умову закінчення рекурсії, тобто умову (вона може бути і не одна), при якій не буде повторного виклику процедури, умова, при якій можемо відразу виконати дії по вставці елементу в відсортований список, пам'ятаючи при цьому, що відомий лише тільки start і pNew;

Відразу можемо виконати дії по вставці елементу в список в двох випадках:

- список порожній (тоді новий елемент стає першим і єдиним елементом списку);
- ключове поле нового елемента менше ключового поля першого елемента списку, отже, новий елемент повинен стати перед першим.

Розглянуті випадки і будуть умовами закінчення рекурсії.

Слід зазначити, що при виконанні будь-якого з цих умов повинні бути виконані одні і ті ж дії. Тому об'єднаємо обидва умови.

Виконання інструкцій для цих умов в рекурсивної процедури матимуть вигляд, показаний рисунку 7.5

```
void addToSortReccurs(TStud * &Start, TStud * pNew)
{
      // умова завершення рекурсії
   if (Start == NULL  || pNew->fam <= Start->fam)
     {
       pNew->next = Start;
       Start = pNew;
     }
   else

      // продовження рекурсії
}
```

Рисунок 7.5 - Виконання інструкцій по вставці нового елементу при виконанні умови закінчення рекурсії

3. Якщо умова завершення рекурсії не виконана, то необхідно тепер наступний елемент списку вважати першим і проводити подальше порівняння ключового поля нового елемента з черговим елементом списку.

Для забезпечення обробки наступного елемента списку замість циклу використовується рекурсія.

Рекурсивно викликаємо процедуру, куди в якості вказівника на перший елемент передаємо адресу наступного.

addToSortReccurs (Start->next , pNew);

Знову викликана процедура отримає параметр Start, який тепер буде містити адресу наступного елемента списку.

Слід зазначити, що параметр Start передається за посиланням, отже, якщо виконається умова закінчення рекурсії, то при виконанні операції

Start = pNew

Start є посиланням на поле next попереднього елемента списку, а значить поле next попереднього елемента списку отримає значення PNew. В результаті при вставці елементу в список утворилася зв'язок нового елемента з попереднім.

Повна рекурсивна процедура вставки нового елемента в упорядкований список має вигляд, показаний на рисунку 7.6:

```
void addToSortReccurs(TStud * &Start, TStud * pNew)
{
```

```
    // умова завершення рекурсії
  if (Start == NULL  || pNew->fam <= Start->fam)
   {
     pNew->next = Start;
     Start = pNew;
   }
  Else
 addToSortReccurs (Start->next ,pNew);//рекурсійний виклик
}
```

Рисунок 7.6 - Рекурсивна процедура додавання елемента в відсортований список

Візьмемо, наприклад, список, показаний на рисунку 8.7, і покроково виконаємо рекурсивную процедуру включення елемента в список, відобразивши при цьому вміст системного стека.

Список має вигляд:

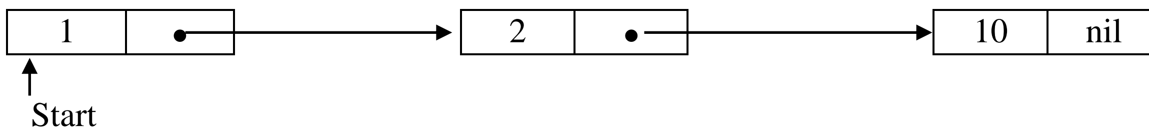


Рисунок 7.7 - Список перед додаванням нового елемента

Тоді буде виконано три виклики процедури addToSortReccurs. Один раз з основної програми і два рази рекурсивно.

Для ілюстрації процесу рекурсії будемо використовувати конструкцію типу Adr (BeginList) як адреса елемента BeginList, що заноситься в стек, конструкцію x.next як вміст поля next елемента списку з ключем x і конструкцію типу Adr (x.next) як адреса елемента x.next, заноситься в стек.

Послідовність викликів процедур і відкатів з них можна схематично представити наступним чином:

I. addToSortReccurs (BeginList, pNew)

II. addToSortReccurs (1->next, pNew)

III. addToSortReccurs (2->next, pNew)

Завершення рекурсії

pNew->next = Start, відповідно pNew->next = 2->next

Start = pNew, відповідно 2->next = pNew

Відкат з III виклика. 2->next посилається на pNew

Відкат з II виклика. 1->next не змінюється

Відкат з I виклика. BeginList не змінюється

Нижче наводиться опис цього процесу з демонстрацією вмісту стека.

Перший виклик процедури addToSortReccurs з основної програми. При цьому:

Start = BeginList
Start != NULL и pNew->inf > 1

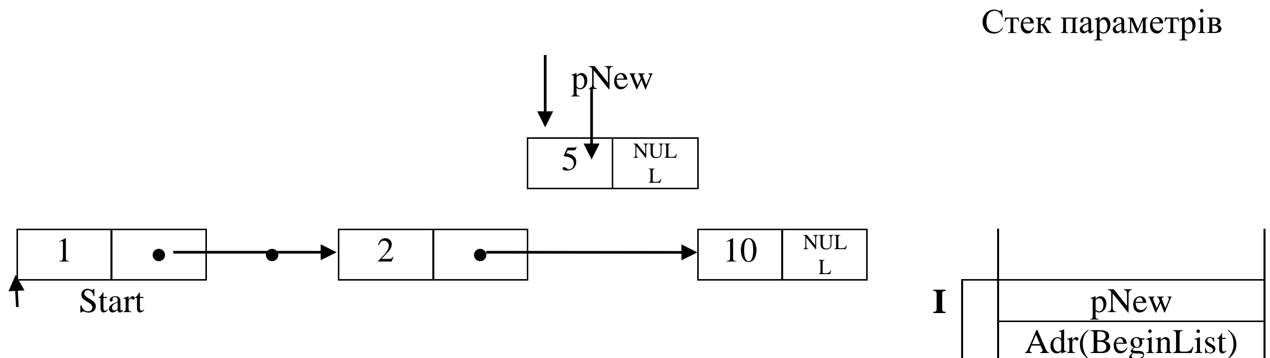


Рисунок 7.8 - Список перед додаванням нового елемента, де Start містить посилання першого елемента

Другий виклик процедури addToSortReccurs. При цьому:

Star = 1->next
1->next != NULL и pNew->inf > 2

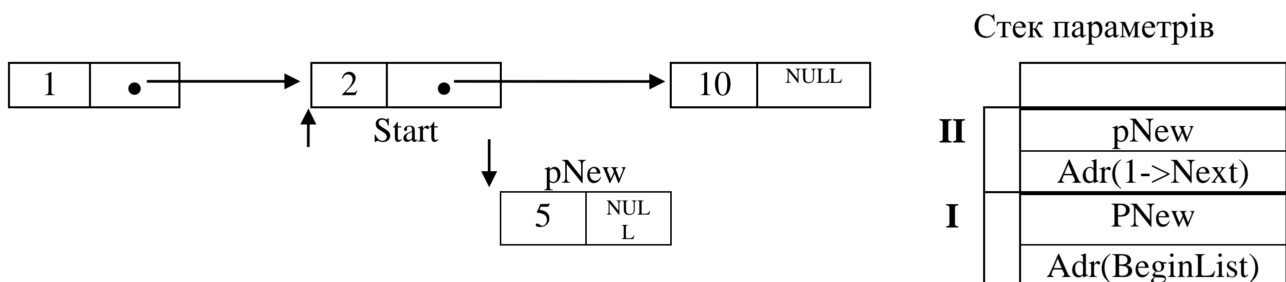
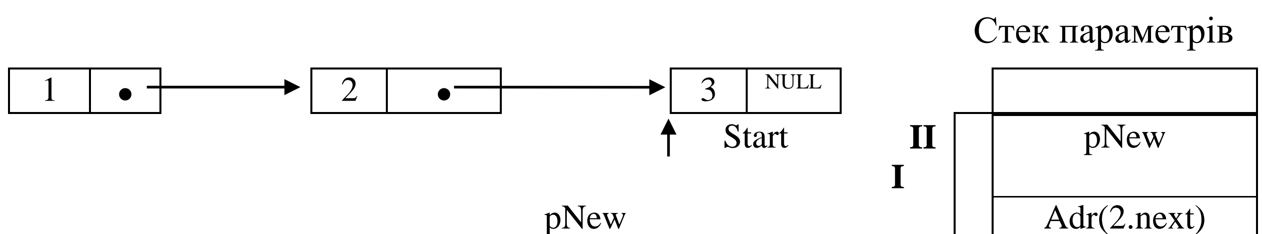


Рисунок 7.9 - Список перед додаванням нового елемента, де Start містить посилання другого елементу

Третій виклик процедури addToSortReccurs. При цьому:
Start = 2.next
2.next != NULL и pNew->inf < 10



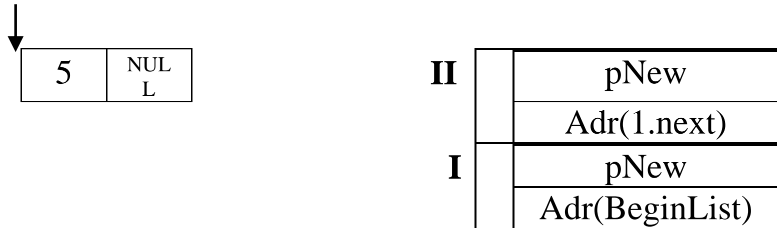


Рисунок 7.10 - Список перед додаванням нового елемента, де Start містить посилання третього елемента

В результаті наступає припинення рекурсії і виконується вставка в список нового елементу pNew з ключем 5. Потім виконується коригування формального параметра процедури:

Start = PNew

що призводить до корегування фактичного параметра цієї процедури, тобто:

2.next = pNew

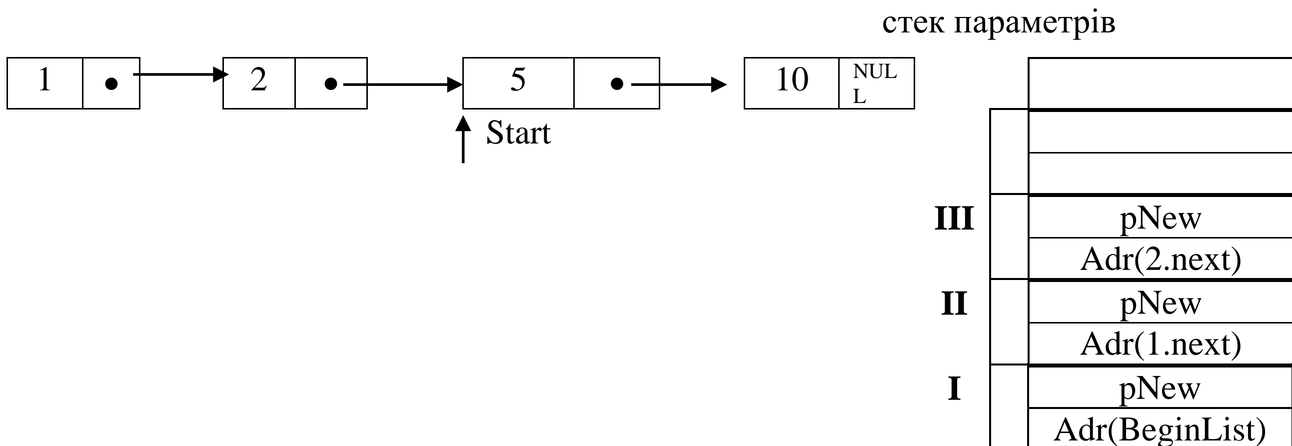


Рисунок 7.11 - Список після вставки нового елемента, де Start містить знову доданого елемента

Після цього починається повернення з рекурсії і звільнення верхівки стека. Яких-небудь серйозних дій при цьому не відбувається.

7.1.5 Заміна рекурсії роботою з власним стеком

Використання рекурсії є потужним інструментом у вирішенні багатьох завдань, особливо в області пошуку і обробки динамічних структур даних. При цьому важливо правильно оцінити дометаність використання рекурсії в тому чи іншому випадку. Якщо глибина рекурсії стає досить великою і суттєво впливає на продуктивність обчислень, дометано повністю відмовитись від використання рекурсії.

При вирішенні обчислювальних задач, рекурсії дометано замінити рекурентними обчисленнями. Наприклад, обчислення N! для досить великих значень N за допомогою рекурсії істотно поступається за швидкістю

обчисленням за рекуррентною формулою, яка реалізується за допомогою простого оператора циклу.

При роботі з динамічними структурами даних можливі різні рішення задачі заміни рекурсії. Наприклад, вище наведені механізми роботи з списковими структурами, які використовують або не використовують рекурсію.

Примітка 1. На практиці використовувати рекурсію для обробки списків великої розмірності не рекомендується, так як рекурсія швидко призведе до переповнення апаратного або системного стека. Однак як засіб освоєння рекурсивних методів вирішення завдань списки є чудовим об'єктом.

Примітка 2. Хорошим додатком рекурсії є бінарні і багатовітчасті дерева.

7.2 Завдання для виконання лабораторної роботи

1. Використовуючи проект лабораторної роботи № 3, створіть список, забезпечивши, як зазвичай, його ініціалізацію і додавання нових елементів.

2. Забезпечте рекурсивне виконання операцій зі списком згідно варіанту, приведеного в таблиці 8.12. При необхідності додайте відсутні візуальні компоненти на форму.

Таблиця 7.12 - Завдання для виконання проекту

Варіант	Зміст елементів списку	Реалізувати операцію зі списками (рекурсивно)
0	ціле число	1. Реалізувати операцію створення копії списку 2. Видалити зі списку всі парні елементи
1	- номер аудиторії - кафедра - кількість комп'ютерів	1. Реалізувати операцію збереження і відновлення списку з файлу 2. Створити другий список, в якому будуть тільки записи про аудторії кафедри ІКС
2	- назва книги - ціна - тираж	1. Реалізувати операцію злиття двох відсортованих списків 2. Видалити книги, перед якими в списку знаходиться книга з тиражем більше 50000 екземплярів.
3	- назва страви - кількість калорій - країна	1. Реалізувати операцію видалення елемента і його дублікатів з невпорядкованого списку 2. Створити другий список, в якому буде зберігається інформація про магазини, де кількість працівників менше 30
4	- назва магазину - кількість рботніков	1. Реалізувати операцію інвертування списку

		2. Видалити записи про ті блюда, перед якими знаходиться страва країни Іспанія
5	- назва магазину - кількість рботніков	1. Реалізувати операцію видалення елемента і його дублікатів з невпорядкованого списку 2. Створити другий список, в якому буде зберігається інформація про магазини, де кількість працівників менше 30
6	ціле число	1. Реалізувати операцію створення копії списку 2. Видалити зі списку всі парні елементи
7	Рядок	1.Реалізувати операцію видалення елемента з дублікатами з відсортованого списку 2. Після кожного рядка, який має довжину більше 3-х символів, вставити елемент з рядком «***»
8	- номер школи - директор - кількість вчителів	1.Реалізувати операцію перетворення списку 2. Видалити інформацію зі списку тих шкіл, де кількість вчителів менше 40
9	ціле число	1.Реалізувати операцію злиття двох відсортованих списків 2. Після кожного парного елемента вставити новий, який дорівнює сумі двох попередніх елементів списку.

7.3 Тестування проекту

Виконайте тестування проекту наступним чином:

- У таблиці розмістіть значення елементів, доданих до списка.
- Виконайте завдання, згідно варіанту, власноруч, і покажіть його результат в звіті;
- Виконайте завдання за допомогою проекту, зображення інтерфейсу з результатом виконання помістіть в звіт.

7.4 Зміст звіту

1) Найменування роботи.
2) Мета роботи.
3) Зображення інтерфейсу проекту, реалізованого відповідно до завдання.
4) Аналіз завдань при створенні рекурсивних процедур.
5) Тексти рекурсивних підпрограм.
6) Результати тестування проекту.

Лабораторна работа №8

Реалізація упорядкованого бінарного дерева пошуку як спискової структури

Мета роботи:

– Отримати практичні навички зі створення і реалізації упорядкованого бінарного дерева, як спискової структури;
– Вивчити поняття строго-бінарне дерево, повне бінарне дерево, майже повне бінарне дерево, впорядковане бінарне дерево;
– Створити проект в системі Qt Creator, в якому здійснюється побудова і обробка упорядкованого бінарного дерева.

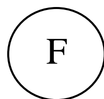
8.1 Теоретичні відомості

8.1.1 Поняття бінарного дерева. Види бінарних дерев

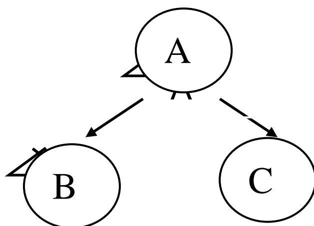
8.1.1.1 Поняття бінарне дерево і термінологія, пов'язана з ним

Бінарне дерево (Binary tree) має наступне рекурсивне визначення - це скінченна множина елементів, яка або порожня, або містить один елемент, який називається коренем дерева, а інші елементи діляться на дві непересічні підмножини, кожна з яких сама є бінарним деревом (БД). Зазвичай їх називають піддеревами.

Приклад бінарних дерев представлений на рисунку 10.1.

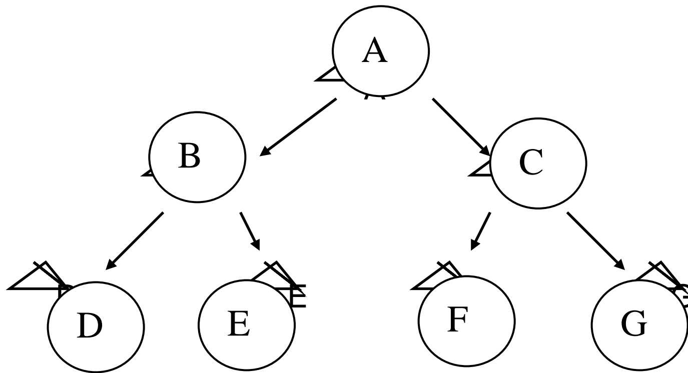


Бінарне дерево з одного елемента.



Бінарне дерево з трьох елементів.

Бінарне дерево з семи елементів.



Це не бінарне дерево (чому?)

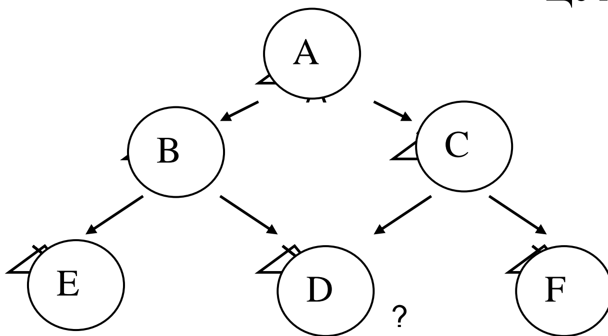


Рисунок 8.1 - Зображення бінарного дерева

Зазвичай елемент дерева називають *вузлом (node)*.

Залежно від місця розташування вузла в дереві розрізняють (на прикладі, наведеному на рисуннку 10.1):

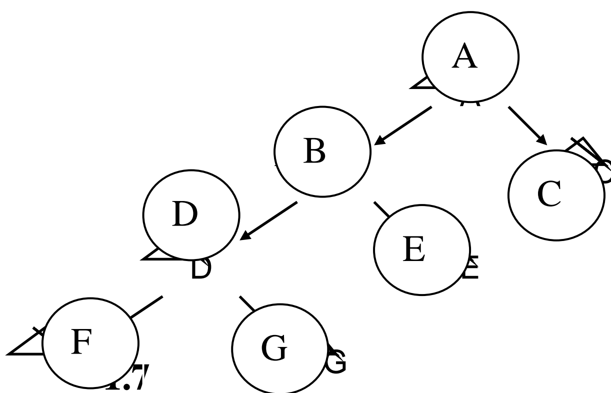
- *корінь (root) дерева* (елемент A)*;*
- *вузол (node) дерева* - будь-який елемент дерева*;*
- *батько (father) вузла*. Наприклад, вузол A є батьком для вузлів B і C, а вузол B - батько для вузлів D і E і т.д.;
- *предок (ancestor) вузла* - будь-який вузол на шляху від батька до кореня, наприклад вузол A, є предком вузлів D і E;
- *власники вузла* - список всіх предків вузла від кореня і до батька, наприклад вузли A і B, є власниками вузла D. Вузли в списку власників мають індекси відповідно 0, 1 і т.д.;

- *нащадок (descendent)* або *спадкоємець (heir) вузла,* наприклад, вузли B і E - нащадки вузла A;
- *син (son),* нащадок (descendent), дитина (child) або *спадкоємець (heir) вузла* або *дочірній (daughter) вузол,* наприклад, вузли B і З - сини вузла A, а вузли D і E - сини вузла B і т.д.;
- *брати (brothers)*, Це вузли, у яких загальний батько, наприклад вузли B і C - брати;
- *лист* (Leaf) дерева - це вузол, який не має синів, наприклад вузли D, E, F і G, є листям.
- *рівень (level) вузла* визначається наступним чином. Якщо корінь має рівень 0, то рівень будь-якого іншого вузла дерева на 1 більше, ніж рівень його батька. У прикладі 5 рівень вузла F дорівнює 3;
- *глибина (depth) дерева* - це максимальний рівень листа дерева плюс 1, Що дорівнює довжині найдовшого шляху від кореня до листа дерева.

8.1.1.2 Строге бінарне дерево

Строго бінарне дерево (СБД) - це дерево, кожен вузол якого, який не є листом, має непорожні праві і ліві піддерева.

Рисунок 8.1 дає приклад дерева такого типу. Але цей же тип має і дерево, зображене на рисунку 8.2:



Строго бінарне дерево, але не майже повне бінарне дерево.

Рисунок 8.2 - Зображення строго-бінарного дерева

8.1.1.3 Повне бінарне дерево

Повне бінарне дерево (ПБД) рівня N - це СБД, все листя якого має рівень N.

Повний бінарне дерево представлено на рисунку 10.3

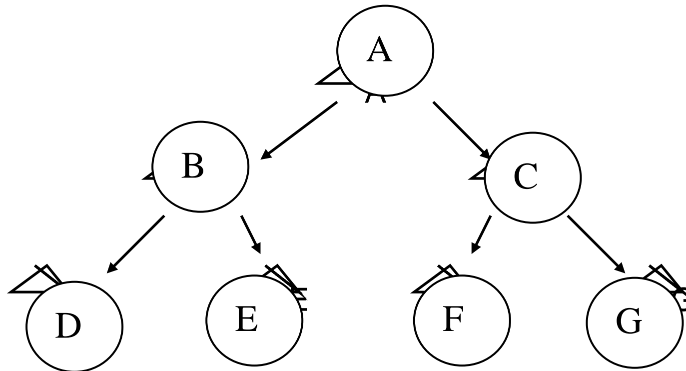


Рисунок 8.3 - Зображення повного бінарного дерева

8.1.1.4 Майже повне бінарне дерево

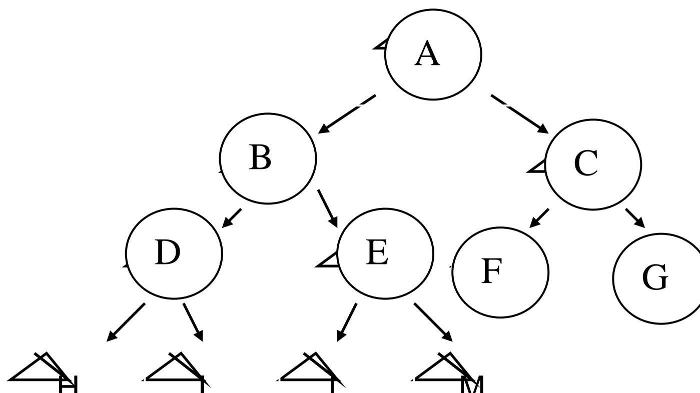
Майже повне бінарне дерево (ППБД) - це дерево, для якого існує невід'ємне ціле число К таке, що:

1. Кожен лист в дереві має рівень К або К + 1.
2. Якщо деякий вузол дерева має правого нащадка, уровеня К + 1, Тоді все ліві нащадки цього вузла, що є листям, також мають рівень К + 1.

Примітка. В такому дереві, якщо у вузла є безпосередній правий нащадок-син, який є листом рівня К + 1, То його лівий нащадок-син також є листом рівня К + 1.

Приклад майже-повних бінарних дерев представлений на рисунках 8.4 і 8.5.

У наведеного нижче дерева умови 1 і 2 виконуються. Число К = 2. це ППБД і СБД. У нього 6 листів і 2 * 6 - 1 = 11 вузлів.



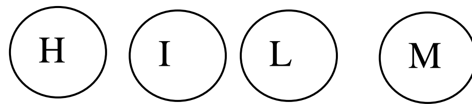


Рисунок 8.4 - Зображення майже повного бінарного дерева

У наведеного нижче дерева умови 1 і 2 виконуються. Число К = 2. Це ППБД і СБД. У нього 6 листів і 2 * 6 - 1 = 12 вузлів.

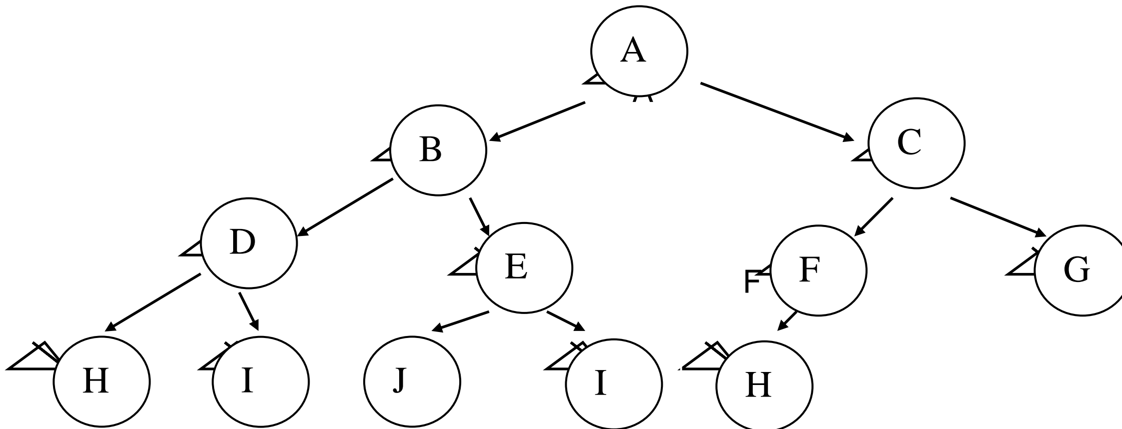


Рисунок 8.5 - Зображення майже повного бінарного дерева

На рисунку 18.6 наведено приклад бінарного дерева, яке не є майже повним.

У наведеного нижче дереві число К = 2, але умова 2 не виконуються. Це не ППБД, але СБД. У нього 6 листів і 2 * 6 - 1 = 12 вузлів. Дійсно, у вузла А є правий нащадок рівня 3 (вузол К) І лівий нащадок рівня 2 (вузол Е).

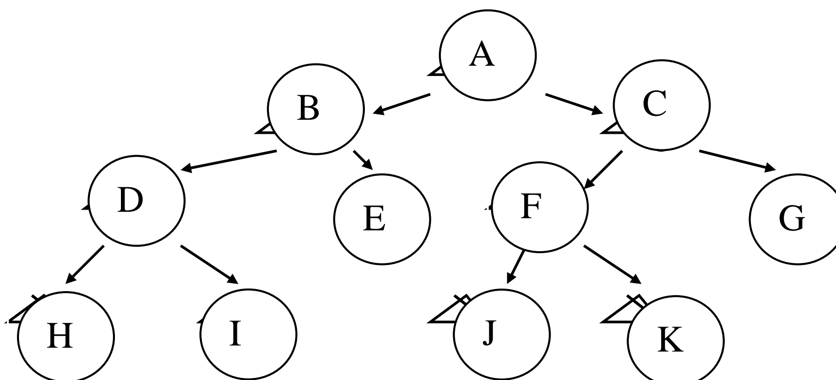


Рисунок 8.6 - Зображення бінарного дерева, яке не є майже повним

Важливою властивістю ППБД є можливість перенумерувати його вузли так, що кореню призначається номер 1, лівому синові подвоєний номер батька, а правому синові подвоєний номер батька плюс 1.

Виконаємо таку нумерацію для дерева, представленого на рисунку 10.7

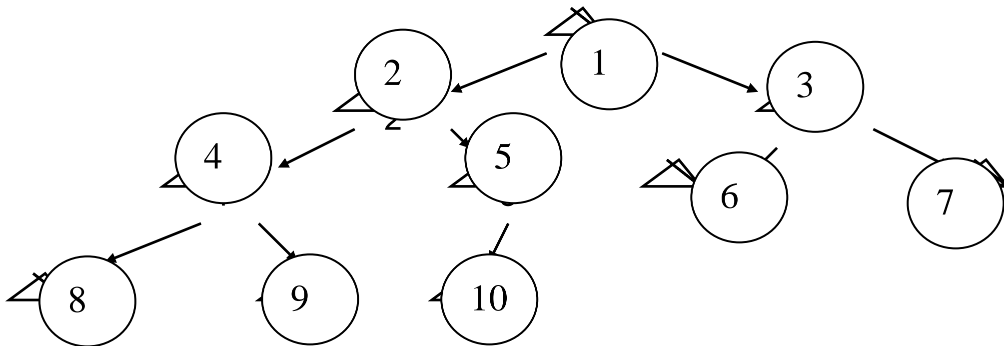


Рисунок 8.7 - Зображення майже повного бінарного дерева з пронумерованими вершинами

При такій схемі нумерації кожного вузла ППБД присвоєно унікальний номер, який однозначно визначає позицію вузла всередині дерева. Дірок в нумерації не допускається.

8.1.2 Способи подання бінарних дерев

8.1.2.1 Вузлове представлення бінарних дерев

Одним з можливих варіантів представлення бінарних дерев є вузлове представлення, коли кожен вузол містить посилання (вказівники) на своїх нащадків. Таким чином, в вузлі вводиться як мінімум три поля:

- поле інформації;
- вказівник на лівого нащадка;
- вказівник на правого нащадка.

Можлива ситуація, коли вузол крім посилань на нащадків, містить вказівник на свого батька.

На рисунку 8.8 зображена структура вузла бінарного дерева, що має вказівники на своїх нащадків:

лівий потік	інформація	правий потік

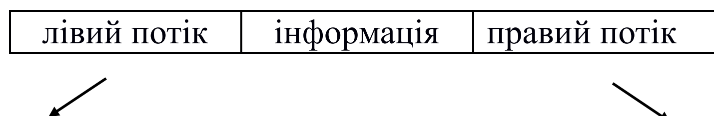


Рисунок 8.8 - Структура вузла бінарного дерева

Нижче наводиться приклад опису вузла бінарного дерева на С ++ (Див. рисунок 8.9).

```
typedef
 struct node{
    int inf;
    node* left, * right;
 }TNode;

TNode * root;
```

Рисунок 8.9 - Визначення типів і глобальної змінної для реалізації бінарного дерева

8.1.2.2 Реалізація майже повного бінарного дерева на базі масиву

В цьому випадку, спираючись на запропонований спосіб нумерації вузлів майже повного бінарного дерева, можна уявити ці вузли як елементи одновимірного масиву. Дійсно, немає необхідності явно представляти посилання на лівий і правий нащадки, тому що вони можуть бути визначені за номером вузла К відповідно як 2 * К і 2 * К + 1.

Аналогічно, за номером вузла можна визначити номер його батька.

Якщо К - номер вузла, то К / 2 - номер його батька.

Для представлення майже повного бінарного дерева використовується одновимірний масив записів.

Розглянемо таке уявлення ППБД для представленого на рисунку 8.10.

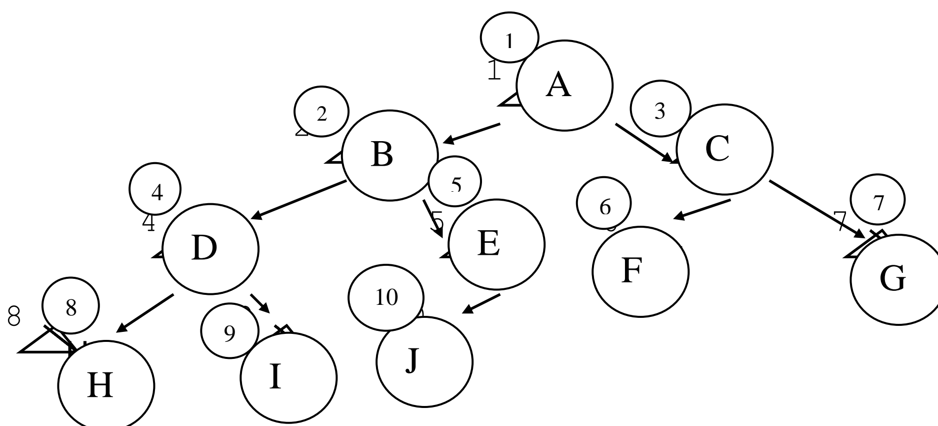


Рисунок 8.10 - Майже повне бінарне дерево, яке буде представлено на базі масиву в пам'яті

Реалізація такого дерева на базі масиву матиме вигляд, представлений на

рисунку 8.11.

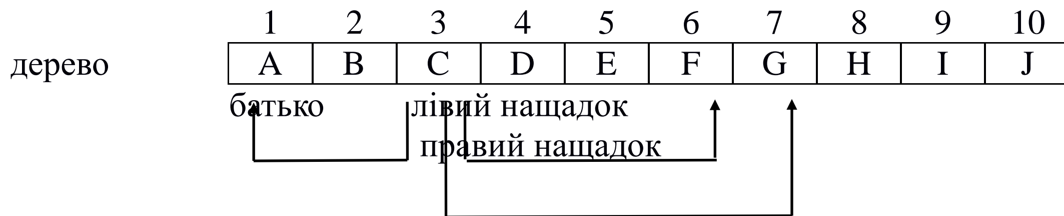


Рисунок 8.11 - Представлення майже повного бінарного дерева в масиві

В принципі таке уявлення ППБД можна використовувати для подання довільних БД. Для цього таке БД має бути приведене (доповнено) фіктивними елементами до відповідного ППБД.

Розглянемо цей підхід на прикладі дерева з рисунка 8.12.

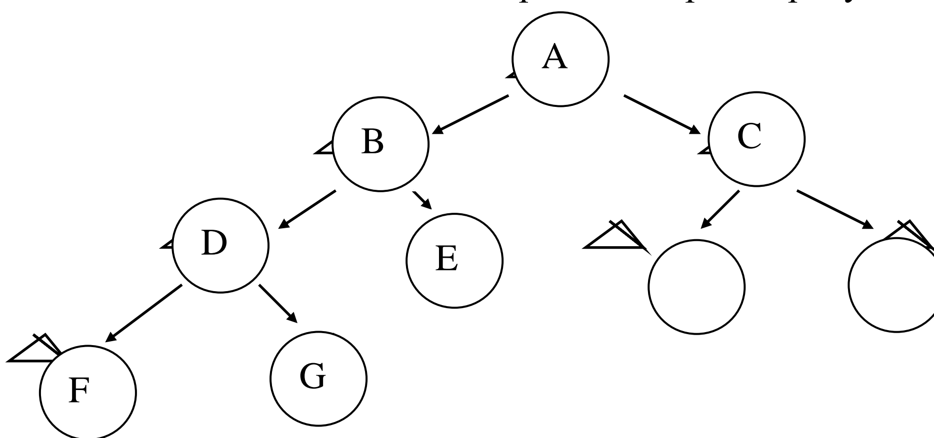


Рисунок 8.12 - Початкове бінарне дерево, яке буде добудовується до майже повного

Для дерева, представленого на рисунку 8.12, його реалізація на основі масиву матиме вигляд, показаний на рисунку 8.13.

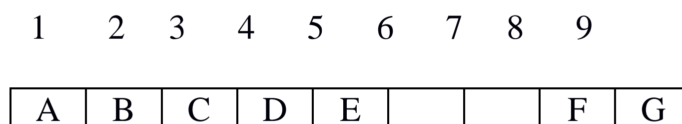


Рисунок 8.13 - Представлення бінарного дерева, яке є майже повним в масиві

Тут виникає питання, як помітити фіктивні вузли. Можливі два підходи:

- ✓ помістити у відповідний елемент масиву, значення якого в дереві не може бути, наприклад код символу "0 " або знаку "**!**"і т.д.;

- ✓ ввести додатковий масив, що містить 1 (true), якщо вузол дерева дійсний, і 0 (false) - якщо вузол фіктивний (див. рисунок 8.14).

A	B	C	D	E			F	G

1	1	1	1	1	0	0	1	1

Рисунок 8.14 - Представлення бінарного дерева, яке є майже повним в масиві з використанням додаткового масиву, що містить інформацію про те фіктивний вузол чи ні

Це особливо зручно в мовах, що дозволяють працювати з бітовим рядком.

Використання динамічних масивів для реалізації дерев дозволяє зняти обмеження на розмір дерева, але на програміста покладаються додаткові завдання по виділенню / звільненню пам'яті під елементи масиву при додаванні / видаленні елементів дерева.

8.1.3 Методи проходження бінарних дерев

Загальний принцип проходження вузлів дерева - повинні бути пройдені всі вузли дерева і кожен вузол повинен бути пройдений тільки один раз.

Найбільш поширені чотири методи проходження БД:

- ✓ *в прямому порядку;*
- ✓ *в симетричному порядку;*
- ✓ *в зворотньому напрямку;*
- ✓ *в горизонтальному порядку.*

У кожному з чотирьох методів не потрібно ніяких дій для проходження порожнього дерева.

Перші три методи визначаються рекурсивно, так що проходження БД вимагає відвідування кореня і проходження його лівого і правого піддерев. Разом з тим ці методи можуть бути реалізовані і не рекурсивно.

Четвертий метод проходження дерев реалізується не рекурсивно і передбачає використання допоміжної структури даних - черги.

Нижче спочатку розглядаються рекурсивні методи проходження бінарних дерев в прямому, симетричному і зворотному порядках, а потім метод проходження такого дерева в горизонтальному порядку.

Найбільш наочними є рекурсивні методи проходження бінарних дерев в прямому, симетричному і зворотному порядках.

Нижче наводяться методи і функції проходження бінарних дерев для випадку їх вузловогопредставлення.

8.1.3.1 Проходження бінарного дерева в прямому порядку

Інакше цей процес називають *перегляд дерева в глибину*. Він складається з 3-х послідовних кроків:

1. Потрапити в корінь (надрукувати вміст);
2. Пройти в прямому порядку ліве піддерево;
3. Пройти в прямому порядку праве піддерево.

Застосуємо цей метод до дерева, представленому на рисунку 8.15.

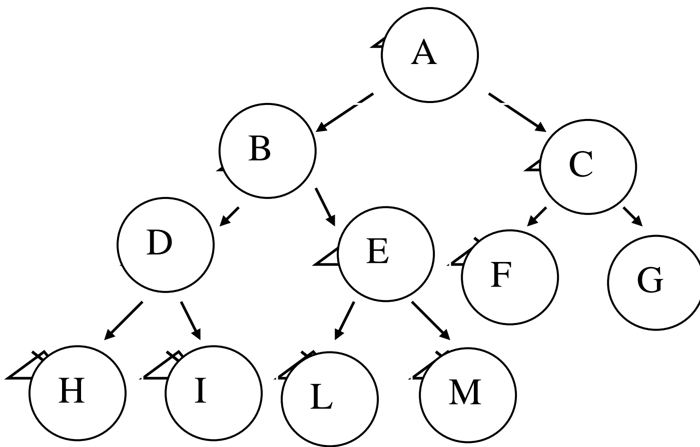


Рисунок 8.15 - Бінарне дерево, яке підлягає обходу

Отримаємо:

A B D H I E L M C F G

Рекурсивна функція обходу бінарного дерева в прямому порядку представлена на рисунку 8.16.

```
void PBDPP (TNode* ptree) {
   if (ptree ! = NULL)
      {cout<< ptree->inf<<endl;
       PBDPP(Ptree->left);
       PBDPP(Ptree->right);
      }
}
```

Рисунок 8.16 - Рекурсивна функція обходу бінарного дерева в прямому порядку

8.1.3.2 Проходження бінарного дерева в симетричному порядку

Процес проходження складається з 3-х послідовних кроків:

1. Пройти в симетричному порядку ліве піддерево;
2. Потрапити в корінь (надрукувати вміст);
3. Пройти в симетричному порядку праве піддерево.

Застосуємо цей метод до заданого дереву. Отримаємо:

H D I B L E M A F C G

Рекурсивна функція обходу бінарного дерева в прямому порядку представлена на рисунку 8.17.

```
void PBDSP (TNode* ptree) {
   if (ptree ! = NULL)
      {
      PBDPP(Ptree->left);
      cout<< ptree->inf<<endl;
      PBDPP(Ptree->right);
      }
}
```

Рисунок 8.17 - Рекурсивна функція обходу бінарного дерева в симетричному порядку

8.1.3.3 Проходження бінарного дерева в зворотному порядку

Процес проходження складається з 3-х послідовних кроків:

1. Пройти в зворотному порядку ліве піддерево;
2. Пройти в зворотному порядку праве піддерево;
3. Потрапити в корінь (надрукувати вміст).

Застосуємо цей метод до заданого дереву. Отримаємо:

H I D L M E B F G C A

Рекурсивна функція обходу бінарного дерева в прямому порядку представлена на рисунку 8.18.

```
void PBDOP (TNode* ptree) {
   if (ptree ! = NULL)
      {
      PBDPP(Ptree->left);
      PBDPP(Ptree->right);
      cout<< ptree->inf<<endl;
      }
}
```

Рисунок 8.18 - Рекурсивна функція обходу бінарного дерева в зворотному порядку

8.1.3.4 Проходження бінарного дерева в горизонтальному порядку

Інакше цей процес називають *перегляд дерева за рівнями*. Він складається з 3-х кроків:

1. Встановити рівень вузла К = 0;
2. Вибрати всі вузли рівня К;
3. Поки не досягнуть максимальний рівень К плюс 1 перейти на пункт 2.

Застосуємо цей метод до заданого дереву. Отримаємо:

A B C D E F G H I L M

Відповідно до наведеного вище алгоритму проходження бінарного дерева в горизонтальному порядку для всіх вузлів рівня К необхідно запам'ятовування всіх їх синів тобто вузлів рівня К + 1. У розробляємій функції на кожному кроці необхідно запам'ятовувати всіх синів всіх вузлів рівня К. Для реалізації цієї дії буде потрібна додаткова структура даних, здатна динамічно зростати і дозволяє запам'ятовувати і вибирати вузли в порядку їх надходження. Для цих цілей добре підходить черга, реалізована за допомогою циклічного списку.

Введемо необхідні додаткові типи даних, змінні і функції роботи з чергою. Елементами черги є вказівники на відповідні вузли дерева, а не їх вміст. Це дозволяє економити пам'ять в разі, якщо обсяг інформації в вузлі дерева вимагає значний обсяг пам'яті.

Нижче наводяться описи типів даних для представлення дерева і черги. Див. рисунок 8.19.

```
wp = Start; // ініціалізація вказівника на поточний елемент
                    // тепер поточним є перший

    // цикл проходу по списку задля обробки всіх його елементів

  while(wp !=NULL)  //поки поточний не отримав значення NULL
       {

            …  // тут знаходяться інструкції по обробці поточного
                  // елементу, до якого отримуємо доступ через вказівник wp

            wp = wp->next;  // в wp заносимо адресу наступного елементу

       }
```

Рисунок 8.19 - Опис типів і оголошення змінних для дерева і циклічної черги.

Функції для подання черги у вигляді циклічного списку представлені на малюнку (див. рисунок 8.20).

```
//функція EmptyQC виконує перевірку циклічної черги на пустоту.
bool EmptyQC (TQueue * EndQue)
{
if (! EndQue)
  {Return true;}
else {return false;}
}
функція InsertQC заносить елемент в циклічну чергу.
void InsertQC (TQueue * & EndQue, TNode * NewElm)
{
   TQueue * wp = new TQueue;
   wp-> info = NewElm;
   wp-> next = NULL;
   if (EndQue == NULL)
     {// випадок першого елемента
EndQue = wp;
EndQue-> next = EndQue; // зациклення черги
  }
else
  {
wp-> next = EndQue-> next; // посилання на перший елемент черги
EndQue-> next = wp;
EndQue = wp;
 }} // InsertQC
```

Функція PBDGPC виконує горизонтальний обхід дерева і повертає вказівник на початок циклічної черги, що містить вказівники на всі вузли дерева в порядку їх горизонтального обходу. (Див. рисунок 8.21)

```
void PBDGPC (TQueue * & EndQ, TNode * PNode)
{
 TQueue * wpB, wpE;
  if (! PNode)
     {Return;} // дерево порожнє
InsertQC (EndQ, PNode); // занесення кореня дерева в чергу
wpB = EndQ-> next; // обхід починається з кореня дерева
do
  {/ * Елементи дописують в чергу до тих пір, поки не будуть перебрані всі вузли дерева. У останнього
елемента черги в поле next варто NULL * /
if (WpB-> info-> left! = NULL)
{
InsertQC (EndQ, WPB-> info-> left); / * Занесення лівого нащадка вузла дерева в чергу * /
}
if  (WpB-> info-> right)! = NULL)
 {
InsertQC (EndQ, WPB-> info-> right); / * Занесення правого нащадка вузла дерева в чергу * /
}
    wpB = wpB-> next;

  while (wpB! = EndQ-> next);
} // PBDGPC
```

Рисунок 8.21 - Функція горизонтального обходу дерева

8.1.4 Впорядковані бінарні дерева (або двійкові дерева пошуку) і основні операції з ними

Впорядковане бінарне дерево - це дерево, в якому для кожного вузла виконується умова - в лівому поддереві розташовані менші елементи, а в правому - великі чи рівні.

Обхід УБС в симетричному порядку дає послідовність елементів, упорядкованих за зростанням.

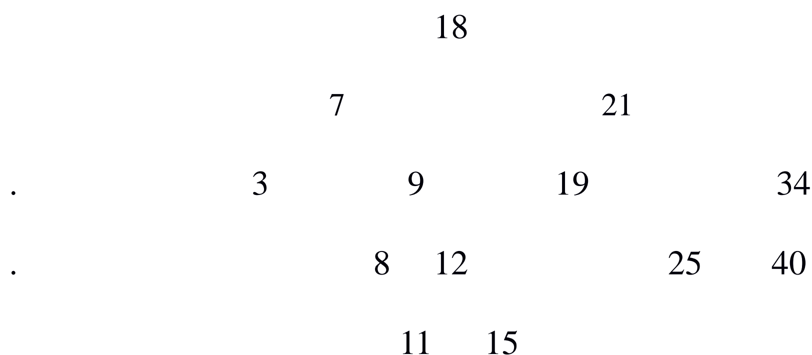


Рисунок 8.22 - Приклад упорядкованого бінарного дерева

Результат симетричного обходу дерева, зображеного на рисунку 10.22, буде наступним:

3, 7, 8, 9, 11, 12, 15, 18, 19, 21, 25, 34, 40

З іншого боку, якщо елемент, зазначений у вузлі упорядкованого бінарного дерева, розглядати як ключ запису, то за допомогою такого дерева можна прискорити пошук запису з необхідним ключем.

Наприклад, для пошуку в дереві запису з ключем 19 необхідно виконати тільки три порівняння. Таке мале число порівнянь обумовлено наведеної організацією дерева.

До основних операцій роботи з бінарними деревами відносяться:

- додавання елемента в дерево;
- обхід дерева;
- пошук елемента в дереві;
- видалення елемента з дерева.

Примітка. У даних методичних вказівках операції будуть виконуватися з впорядкованими бінарними деревами, тобто такими деревами, в яких для

кожного елемента X виконується правило: в лівому поддереве розміщуються ключі, менші X, в правому піддереві - великі чи рівні X.

8.1.5 Додавання елемента в бінарне дерево

При вставці в впорядковане бінарне дерево вузол вставляється або як поддерево вже існуючого вузла або як єдиний вузол дерева (в разі, коли дерево порожнє і новий вузол стає коренем дерева). Тому і лівий, і правий зв'язки нового вузла повинні бути рівні nul.

Коли дерево порожнє, значення, що передається у вигляді параметра вказівника на корінь дерева, так само nul. В цьому випадку потрібно змінити його так, щоб він вказував на новий вузол, який був вставлений як кореневий. При вставці другого елементу переданий з основної програми параметр RootTree вже не буде дорівнює nul, і треба приймати рішення про те, в яке поддерево необхідно вставити новий вузол.

Текст рекурсивної функції додавання нового вузла в впорядковане бінарне дерево наступний (див. Рис. 10.23):

```
void insTree(TNode * & RootTree, TNode * elem)
{
   if (rootTree == NULL)
   {
     rootTree = elem;
   }
   else
     if (Elem->inf < rootTree->inf)
       {
        insTree(RootTree->left, elem);
       }
     else
        {
        insTree(RootTree->right, elem);
        }
} // insTree
```

Рисунок 8.23 - Функція додавання нового елемента в упорядковане бінарне дерево

8.1.6 Пошук елемента в бінарному дереві

Рекурсивна функція в якості параметрів має ключ вузла, вказівник на вузол дерева і вказівник на шуканий вузол, який передається за посиланням. Функція повертає вказівник на шуканий вузол.

Функція пошуку елемента в упорядкованому бінарному дереві приведена на рисунку 8.24.

```
Void findInTree (String key, TNode* pNode, TNode* findNode)
{
   if (pNode == NULL)
    {
     findNode = NULL;
     return;
   }
   if (PNode->inf > key) // спуск по лівому піддереву
      {
       findInTree(Key, pNode->left, findNode);
      }
   else
      if (PNode->inf < key)// спуск по правому піддереву
         {
         findInTree(Key, pNode->right, findNode);
         }
      else
         {
          findNode = pNode;
          return; // вузол знайдений
          }
} // findInTree
```

Рисунок 8.24 - Функція пошуку елемента в упорядкованому бінарному дереві

8.1.7 Видалення елемента з бінарного дерева

Більш цікавою є функція видалення елемента з дерева. Сторінки видаляються дуже просто, шляхом очищення вказівника у батьківського вузла. Вузли з одним нащадком також видаляються нескладно - у батьківського вузла посилання перекидається на цього нащадка. Дещо складніше справа йде з видаленням вузла, у якого два нащадка. Для цього потрібно обміняти його ключ з найлівішим нащадком з правого піддерева (або з найправішим нащадком лівого піддерева), а потім вже видаляти цього нащадка (він або не буде мати нащадків, або буде мати лише одного нащадка, інакше він не може виявитися найправішим, або найлівішим). Для початку проілюструємо це на рисунку 8.25. Нехай ми хочемо видалити елемент з ключем 7 і вибираємо найправішого нащадка з лівого піддерева:

Рисунок 8.25 - Видалення елемента з упорядкованого бінарного дерева

У лівій частині рисунка ми знаходимо найправіший елемент з лівого піддерева і записуємо його значення в «видаляємий» вузол. На середній частині рисунка ми здійснюємо пошук вузла з ключем, що збігається з кореневим ключем, в лівому поддереві. У правій частині рисунка показано видалення цього елемента (у нього обов'язково тільки одне піддерево).

8.2 Виконання проекту

Нижче розглядається приклад створення програми, яка подібна до того, що слід розробити.

Сформувати впорядковане бінарне дерево. Маючи вказівник P на корінь непустого дерева, вивести максимальне і мінімальне зі значень його вузлів.

Передбачуваний інтерфейс програми, представлений на малюнку 8.26.

Рисунок 8.26 - Інтерфейс створюваної програми

Головне меню програми має наступні пункти (рисунок 8.27)

1. Створення дерева
2. Додавання нового елемента в впорядковане дерево
3. Знайти мінімальний і максимальний елемент

Рисунок 8.27 Головне меню програми

Як видно з малюнка 8.26 інтерфейс складається з двох областей: області введення вихідних даних і області виведення результатів.

Вихідними даними є кількість вузлів дерева, яке вводиться користувачем.

В область результатів виводиться впорядковане бінарне дерево і знайдені максимальний і мінімальний елементи дерева.

Розглянемо докладно функції, необхідні для вирішення поставленого завдання.

1. ***Функція додавання елементу*** в бінарне дерево аналогічна функції, описаної на малюнку 8.23 методичних вказівок.

2. ***Функція ініціалізації бінарного дерева*** в якості параметрів отримує вказівник на корінь дерева і кількість елементів дерева. Вказівник на корінь дерева передається в функцію за посиланням, і нове його значення повертається в головну підпрограму. Текст функції наведено на малюнку 8.28.

(Для її виконання необхідно підключити бібліотечний фал <Ctime>)

```
void initTree(TNode * & RootTree, int cntEl)
{
   TNode* pNew; // вказівник на новий вузол дерева

   rootTree = NULL;
   qsrand(time(NULL) | clock());
```

```
  for (int i =0; i <cntEl; i ++)
   {
    pNew = new TNode;
    pNew->inf = qrand() % 100;
    pNew->left = NULL;
    pNew->right = NULL;
    insTree(RootTree, pNew);
   }
}
```

Рисунок 8.28 - Функція ініціалізації бінарного дерева

3. ***Для виведення бінарного дерева*** використовувалася функція прямого обходу. Текст функції наведено на малюнку 8.29. У неї доданаможливість виконання відступів для виведення вмісту вузлів різних рівнів для кращої наочності.

```
void showTree(TNode* rootTree)
{
  static int tab =0;// кількість відступів в залежно
                    // від рівня вузла
  if (! RootTree)  // умова закінчення рекурсії
                   // вказівник на корінь дорівнює NULL

  {
    cout<<"NULL"<<endl;
    return;
  }
 cout<< rootTree->inf<<endl; // вивід кореня

  tab ++; // збільшення кількості відступів
  for(int i =0; I <tab; i ++)// виконання відступу на екрані
    { cout<<"      "; }
  cout<<"Лев:";
  showTree(RootTree->left); // рекурсивний спуск
                            // в ліве піддерево
  for(int i =0; I <tab; i ++)// виконання відступу на екрані
    { cout<<"      "; }
  cout<<"Пр:";
  showTree(RootTree->right); // рекурсивний спуск
                             // в праве піддерево

  tab--;// зменшення кількості відступів

}
```

Рисунок 8.29 - Функція виведення бінарного дерева на екран

4. У рекурсивную функцію пошуку мінімального і максимального елемента в дереві як параметр передається вказівник на корінь дерева (RootTree) і дві змінні Max і Min, в яких з функції повертаються значення максимального та мінімального елементів дерева. Ці дві змінні передаються по посиланню. Текст функції наведено на малюнку 10.31. До виклику функції необхідно ініціалізувати параметри max і min відповідно мінімальним значенням і максимальним з безлічі чисел типу int, використовуючи граничні

константи INT_MIN і INT_MAX (для їх використання необхідно підключити модуль<Climits>), В такий спосіб (див. рисунок 8.30):

```
int max =INT_MIN, min =INT_MAX;
getMaxMin(root, max, min);
cout<<"Максимальне значення: "<< max <<endl;
cout<<endl;
cout<<"Мінімальна значення: "<< min <<endl;
```

Рисунок 8.30 - Підготовка виклику і виклик функції знаходження мінімального і максимального елементів на екран

```
void getMaxMin(TNode* rootTree, int & Max, int & Min)
{
   if (RootTree == NULL) // умова закінчення рекурсії
                  // дерево пусте
   {
      return;
   }
   if (RootTree->inf > max)
   {
      max = rootTree->inf;
   }
   if (RootTree->inf < min)
   {
      min = rootTree->inf;
   }
   getMaxMin(RootTree->left, max, min);
   getMaxMin(RootTree->right, max, min);
}
```

Рисунок 8.31 - Функція пошуку мінімального і максимального елемента в упорядкованому бінарному дереві

Для виконання операції «Створення дерева» необхідно створити функцію, яка буде містити наступне (див. рисунок 8.32).

```
int cnt;
cout<<"Введіть кількість елементів дерева: ";
cin>> cnt;
cout<<endl;
initTree(root, cnt);
showTree(root);
```

Рисунок 10.32 - Вміст функції, які забезпечують виконання операції «Створення дерева»

Слід зазначити, що вказівник на корінь дерева повинен бути глобальною змінною. Не забудьте описати типи для реалізації бінарного дерева, як це було зроблено на малюнку 10.9.

8.3 Завдання для виконання лабораторної роботи

1. Використовуючи функції, представлені в попередньому пункті, створіть впорядковане бінарне дерево, ключами якого є цілі числа, забезпечивши його ініціалізацію і додавання нових елементів.

2. Забезпечте рішення задач для упорядкованого бінарного дерева згідно варіанту, наведеного в таблиці 10.1. При необхідності додайте відсутні пункти меню для вибору операцій з деревом.

3. Якщо після виконання завдання дерево змінюється, то його вміст слід виводити на екран в початковому стані і після виконання деяких дій (двічі).

Таблиця 8.1 - Завдання для виконання проекту

Варіант	Використовуваний спосіб обходу дерева для виконання завдання 1	Завдання
0	прямий	1. Знайти суму всіх елементів дерева 2. Вивести на екран всі листи дерева
1	зворотній	1. Обчислити середнє арифметичне елементів дерева 2. Вивести на екран ті листи дерева, які мають парні значення
2	симетричний	1. Знайти кількість парних елементів дерева 2. Вивести на екран значення вузлів, у яких тільки один нащадок (лівий або правий)
3	прямий	1. Обчислити кількість елементів дерева, кратних 10 2. Видалити ті листи в дереві, у яких немає братів
4	зворотній	1.Збільшити всі елементи дерева в два рази 2. Визначити, чи є дерево майже повним
5	симетричний	1. Всі негативні елементи дерева замінити на нулі 2. Вивести на екран ті вузли, сума значень синів яких парна
6	прямий	1. Кожен елемент дерева замінити на зворотній. Тобто якщо є елемент 345, то замінюємо його на 543 2. Визначити в якому поддереві (лівому або правому) кількість парних елементів більша. Вивести ці елементи на екран для кожного піддерева окремо

7	зворотній	1. Всі елементи дерева, які закінчуються на цифру 5 замінити на елемент зі значенням 0. 2. Вивести на екран ті вузли, у яких кількість непарних елементів в лівому поддереві дорівнює кількості парних елементів в правому поддереве і для кожного з цих вузлів вивести ці елементи (непарні з лівого піддерева і парні з правого піддерева)
8	симетричний	1. Визначити кількість елементів дерева, що закінчуються на цифру 9 2. Вивести перелік тих вузлів, у яких кількість парних елементів в правому і лівому піддереві рівні.
9	прямий	1.Ті елементи дерева, які складаються з однакових цифр, замінити значенням 0 2.Вивесті на екран ті вузли, у яких різниця сум елементів в правому і лівому поддереві більша 20. Вивести ці суми для кожного вузла

8.4 Завдання з виконання обходу бінарних дерев

Для дерев, зображених на рисунках 8.33 і 8.34, перерахуйте вузли дерева при прямому, зворотному, симетричному і горизонтальному обході.

Рисунок 8.33 - Бінарне дерево для виконання завдання п.8.4

Рисунок 8.34 - Бінарне дерево для виконання завдання п.8.4

8.5 Зміст звіту

1. Найменування роботи.
2. Мета роботи.

3. Визначення бінарного, строго-бінарного, повного, майже повного і упорядкованого дерев.
4. Зображення інтерфейсу проекту, реалізованого відповідно до завдання.
5. Текст завдання і зображення інтерфейсу з вихідним деревом і результатом виконання завдання.
6. Схема алгоритму розв'язання задачі пункту 2.
7. Текст програми.
8. Результат виконання завдання п.10.4. (Рисунок дерева і для кожного назви обходу - послідовність вершин).
9. Висновки про результати тестування проекту і про особливості реалізації функцій обробки дерева за завданням відповідно до варіанту

Лабораторна робота №9
Знайомство з бібліотекою шаблонів C++ (STL)

Мета работи:
- Ознайомитися з бібліотекою шаблонів C ++ STL і її основними термінами і поняттями;
- Отримати практичні навички використання контейнерних класів і алгоритмів STL;
- Реалізувати проект за завданням, згідно варіанту, з використанням зазначеного контейнерного класу STL.

9.1.1 Короткі теоретичні відомості

9.1.2 Загальний опис STL

Стандартна бібліотека шаблонів **(STL)** (англ. Standard Template Library) - набір узгоджених узагальнених алгоритмів, контейнерів, засобів доступу до їх вмісту і різних допоміжних функцій в C ++.

Стандартна бібліотека шаблонів до її включення в C ++ була сторонньою розробкою, спочатку фірми HP, а потім SGI. Стандарт мови не називає її «STL», так як ця бібліотека стала невід'ємною частиною мови, проте багато людей до сих пір використовують цю назву, щоб відрізняти її від іншої частини стандартної бібліотеки (потоки введення / виводу (iostream), підрозділ Сі та ін.).

Проект під назвою STLPort, заснований на SGI STL, здійснює постійне оновлення STL, iostream і строкових класів. Деякі інші проекти також займаються розробкою приватних застосувань стандартної бібліотеки для різних конструкторських завдань. Кожен виробник компіляторів C ++ обов'язково поставляє якусь реалізацію цієї бібліотеки, так як вона є дуже важливою частиною стандарту і широко використовується.

У бібліотеці виділяють п'ять основних компонентів:
- Контейнер (англ. Container) - зберігання набору об'єктів в пам'яті.
- Итератор (англ. Iterator) - забезпечення засобів доступу до вмісту контейнера.
- Алгоритм (англ. Algorithm) - визначення обчислювальної процедури.
- Адаптер (англ. Adaptor) - адаптація компонентів для забезпечення різного інтерфейсу.
- Функціональний об'єкт (англ. Functor) - приховування функції в об'єкті для використання іншими компонентами.

Такий поділ дозволяє зменшити кількість компонентів. Наприклад,

замість написання окремої функції пошуку елемента для кожного типу контейнера забезпечується єдина версія, яка рабо-тане з кожним з них, поки дотримуються основні вимоги.

9.1.3 Контейнери STL

Контейнери

Контейнери бібліотеки STL можна розділити на чотири категорії: послідовні, асоціативні, контейнери-адаптери і псевдоконтейнери.

Таблиця 9.1 – Послідовні контейнери

vector	С-подібний динамічний масив довільного доступу з автоматичною зміною розміру при додаванні / видаленні елемента. Доступ за індексом за O (1). Додавання / видалення елемента в кінець vector займає амортизоване час O (1), та ж операція на початку або середині vector -O (n). Стандартна швидке сортування за O (n * logn). Пошук елемента перебором займає O (n). Існує спеціалізація шаблону vector для типу bool, яка вимагає менше пам'яті за рахунок зберігання елементів у вигляді бітів, однак вона не підтримує всіх можливостей роботи з ітераторами.
list	Двозвя'зний список, елементи якого зберігаються в довільних шматках пам'яті, на відміну від контейнера vector, де елементи зберігаються в безперервній області пам'яті. Пошук перебором повільніший, ніж у vector через більшого часу доступу до елементу. Доступ за індексом за O (n). У будь-якому місці контейнера вставка і видалення здійснюються дуже швидко - за O (n).
deque	Дек. Контейнер схожий на vector, але з можливістю швидкої вставки і видалення елементів на обох кінцях за O (1). Реалізований у вигляді двозвязанного списку лінійних масивів. З іншого боку, на відміну від vector, дек не гарантує розташування всіх своїх елементів в безперервній ділянці пам'яті, що робить неможливим безпечне використання арифметики вказівників для доступу до елементів контейнера.

Таблиця 9.2– Асоціативні контейнери

Set	Впорядкована множина унікальних елементів. При вставці / видаленні елементів множини ітератори, що вказують на елементи цієї множини, не стають недійсними. Забезпечує стандартні операції над множинами типу об'єднання, перетину, віднімання. Тип елементів безлічі повинен реалізовувати оператор порівняння «operator <» або потрібно надати функцію-компаратор. Реалізовано на основі самобалансірующіхся дерева

	двійкового пошуку.
Multiset	Те ж що і set, але дозволяє зберігати повторювані елементи.
Map	Упорядкований асоціативний масив пар елементів, що складаються з ключів та відповідних їм значень. Ключі повинні бути унікальні. Порядок проходження елементів визначається ключами. При цьому тип ключа повинен реалізовувати оператор порівняння "operator <," або потрібно надати функцію-компаратор.
multimap	Те ж що і map, але дозволяє зберігати кілька однакових ключів.

Таблиця 9.3 – Контейнери-адаптери

stack	Стек - контейнер, в якому додавання і видалення елементів здійснюється з одного кінця.
queue	queue - контейнер, з одного кінця якого можна додавати елементи, а з іншого - витягувати.
priority_queue	priority_queue - організований так, що найбільший елемент завжди стоїть на першому місці.

Таблиця 9.4 – Псевдоконтейнери

bitset	Служить для зберігання бітових масок. Схожий на vector <bool> фіксованого розміру. Розмір фіксується тоді, коли оголошується об'єкт bitset. Ітераторів в bitset немає. Оптимізовано за розміром пам'яті.
basic_string	Контейнер, призначений для зберігання і обробки рядків. Зберігає в пам'яті елементи поспіль єдиним блоком, що дозволяє організувати швидкий доступ до всієї послідовності. Елементи повинні бути простих (фундаментальних) типів даних. Визначено конкатенація за допомогою +.
valarray	Шаблон служить для зберігання числових масивів і оптимізований для досягнення підвищеної обчислювальної продуктивності. В деякій мірі схожий на vector, але в ньому відсутня більшість стандартних контейнерих операцій. Визначено операції над двома valarray і над valarray і скаляром (поелементні). Ці операції можливо ефективно реалізувати як на векторних процесорах, так і на скалярних процесорах з блоками SIMD.

У контейнерах для зберігання елементів використовується семантика передачі об'єктів за значенням. Іншими словами, при додаванні, контейнер отримує копію елемента. Якщо створення копії небажано, то використовують контейнер вказівників на елементи. Присвоєння елементів реалізується за допомогою оператора присвоювання, а їх руйнування відбувається з використанням деструктора. У таблиці наведені основні вимоги до елементів в контейнерах:

Таблиця 9.5 – Вимоги до елементів контейнера

Метод	Опис	Примітка
Конструктор копії	Створює новий елемент, ідентичний старому	Використовується при кожній вставці елементу в контейнер
Оператор присвоювання	Замінює вміст елемента копією вихідного елемента	Використовується при кожній модифікації елемента
Деструктор	Руйнує елемент	Використовується при кожному видаленні елемента
Конструктор за замовчуванням	Створює елемент без аргументів	Застосовується тільки для певних операцій
Operator ==	Порівнює два елементи	Використовується при виконанні operator == для двох контейнерів
Operator <	Визначає, чи менше один елемент від іншого	Використовується при виконанні operator <для двох контейнерів

Всі «повноцінні» стандартні контейнери задовольняють певним набором вимог. У наведеній нижче таблиці вважається, що С - клас контейнера, що містить об'єкти типу Т.

Таблиця 9.6 – Вимоги до елементів контейнера

Вираз	Тип, що повертається	Складність	Примітка
C::value_type	T	Час компіляції	
C::reference	T	Час компіляції	
C::const_reference		Час компіляції	
C::pointer	Тип вказівника, що вказує на C::reference	Час компіляції	Вказівник на T
C::iterator	Тип вказівника, що вказує на C::reference	Час компіляції	Ітератор будь-якого типу, крім ітератора виведення
C::const_iterator	Тип вказівника, що вказує на	Час компіляції	Ітератор будь-якого типу, крім ітератора

	C::const_reference		виведення
C::size_type	Беззнаковий цілочисельний тип	Час компіляції	
C obj;		Постійна	Після: obj.size() == 0
C obj1; obj1 = obj2;		Лінійна	Після: obj1 == obj2
C obj; (&obj)->~C();	Результат не використовується	Лінійна	Після: a.size() == 0.
obj.begin()		Постійна	
obj.end()		Постійна	
obj1 == obj2	Зворотній в bool	Лінійна	
obj1 != obj2	Зворотній в bool	Лінійна	
obj.size()	Size_type	Залежить від типу	Не рекомендується застосовувати для перевірки, чи порожній контейнер
obj.empty()	Зворотній в bool	Постійна	
obj1 < obj2	Зворотній в bool	Лінійна	
obj1 > obj2	Зворотній в bool	Лінійна	
obj1 <= obj2	Зворотній в bool	Лінійна	
obj1 >= obj2	Зворотній в bool	Лінійна	
obj.swap(obj2)	Void	Постійна	

9.1.4 Ітератори STL

У Бібліотеці STL для доступу до елементів в якості посередника вікорістовується узагальнена Абстракція, іменована ітератором.

Ітератори - це узагальнення вказівників, які дозволяють програмісту працювати з різнімі структурами даних (контейнерами) однаково способом.

Кожен контейнер підтримує «свій» вид ітератора, який являє собою «модернізований» інтелектуальний вказівник, «знає» як отримати доступ до елементів конкретного контейнера. Стандарт С ++ визначає п'ять категорій ітераторів.

Таблиця 9.7 – Категорії ітераторів

Категорія	**– Підтримувані операції**	**Примітка**
Вхідні	– operator ++,	Забезпечують доступ для читання

	– operator *, – operator->, – конструктор копії, operator =, – operator ==, – operator! =	в одному напрямку. Дозволяють виконати присвоювання або копіювання за допомогою оператора присвоювання і конструктора копії
Вихідні	– operator ++, – operator *, – конструктор копії	Забезпечують доступ для запису в одному напрямку. Їх не можна порівнювати.
Однонаправлені	– perator ++, – operator *, – operator->, – конструктор копії, конструктор за замовчуванням, – operator =, – operator ==, – operator! =	Забезпечують доступ для читання і запису в одному напрямку. Дозволяють виконати присвоювання або копіювання за допомогою оператора присвоювання і конструктора копії. Їх можна порівнювати.
Двонаправлені	– operator ++, – operator--, – operator *, – operator->, – конструктор копії, конструктор за замовчуванням, – operator =, – operator ==, – operator! =	Підтримують всі функції, описані для односпрямованих ітераторів (див. Вище). Крім того, вони дозволяють переходити до попереднього елемента.
Довільного доступу	– operator ++, – operator--, – operator *, – operator->, – конструктор копії, конструктор за замовчуванням, – operator =, – operator ==, – operator! =, – Operator +, – operator-,	Еквівалентні звичайним вказівникам: підтримують арифметику вказівників, синтаксис індексації масивів і всі форми порівняння.

	– operator + =, – operator- =, – operator <, – operator>	

9.1.5 Шаблони в С++

Шаблони використовуються, щоб полегшити процес написання повноцінних програм, де під поняттям "написання" мається на увазі не тільки процедура початкового написання коду програми, але і слідуючий за цим довгий процес налагодження, модифікації і супроводу, створеного вами програмного продукту. Чим же шаблони можуть спростити процес написання програм?

Раніше ми дублювали частини програм, використовуючи простий, але ефективний засіб - текстовий редактор. Сьогодні С ++ пропонує нам більш досконалий спосіб дублювання, і назва йому - "шаблони".

Найбільш поширеним приводом для дублювання фрагментів програм є необхідність реалізувати якийсь новий обсяг коду, аналогічний тим самим написаному, але змінивши типи даних (наприклад, цілі на цілі довгі). Найчастіше для подібної операції за допомогою програми-редактора повторювали текст програми ще раз і міняли типи даних.

Програмуючи на С ++, ви могли б скористатися засобами перевизначення (перевантаження) і дати обом функцій одне і теж ім'я. Перевизначення робить текст програми більш наочним, але не позбавляє нас від необхідності повторювати один і той же алгоритм в декількох місцях.

Механізм шаблонів пропонує досконале рішення, що дозволяє відокремити загальний алгоритм від його реалізації стосовно до конкретних типів даних. Ви можете скласти текст підпрограми зараз, а використовувані типи уточнювати пізніше. Це можливо, тому що використовуваний тип даних є в цьому випадку параметром. Шаблони поєднують в собі переваги одноразової підготовки фрагментів програми (аналогічно макрокомандам) і контроль типів, властивий переобумовленої функцій.

У мові С ++ є два типи шаблонів - шаблони функцій і шаблони класів.

9.1.4.1 Шаблони функцій

Оголошення шаблону функції починається з заголовка, що складається з ключового слова template, за яким слідує список параметрів шаблону.

```
template <class X>
X min (X a, X b)
{
   return a<b ? a : b;
```

```
}
```

Рисуноу 9.1. - Опис шаблона функції

Ключове слово class в описі шаблону означає тип, ідентифікатор в списку параметрів шаблону X означає ім'я будь-якого типу.
В описі заголовка функції цей же ідентифікатор означає тип що повертається функцією, значення і типи параметрів функції.

```
int m = min (1, 2);
//Екземпляр шаблону функції породжується (генерується) компілятором

int min (int a, int b)

{
    return a<b ? a : b;
}
```

Рисунок 9.2. - Використання шаблону функції

У списку параметрів шаблону слово class може також відноситься до звичайного типу даних. Таким чином, список параметрів шаблону <class T> просто означає, що T являє собою тип, який буде поставлено пізніше. Так як T є параметром, що позначає тип, шаблони іноді називають параметризрвані типами.

Наведемо опис шаблону функції:

```
template <class T>

T toPower (T base, int exponent) { T result = base;

if (exponent==0) return (T)1; if (exponent<0) return (T)0;
while (--exponent) result *= base;
return result;

}
```

Рисунок 9.3. - Опис шаблону функції

Змінна result має тип T, так що, коли передане в програму значення є 1 або 0, то воно спочатку приводиться до типу T, щоб відповідати оголошенню шаблону функції.

Типовий аргумент шаблону функції визначається згідно типам даних, використовуваних у виклику цієї функції:

```
int i = toPower (10, 3);
long l = toPower (1000L, 4);
```

```
double d = toPower (1e5, 5);
```

Рисунок 9.4 - Виклик цієї функції.

У першому прикладі T стає типом int, в другому - long. Нарешті, в третьому прикладі T стає типом double. Наступний приклад приведе до помилки компіляції, так як в ньому бере змінна і повернені значення мають різні типи:

```
int i = toPower (1000L, 4);
```

Рисунок 9.5 - Виклик функції.

9.1.4.2 Вимоги до фактичних параметрів шаблону

Шаблон функції toPower () може бути використаний майже для будь-якого типу даних. Застереження "майже" виникає з характеру операцій, що виконуються над параметром base і змінної result в тілі функції toPower ().

Який би тип ми не використали в функції toPower (), ці операції для неї повинні бути визначені. В іншому випадку компілятор не знатиме, що йому робити. Ось список дій, які виконуються в функції toPower () зі змінними base і result:T result = base;

1. return (T)1;
2. return (T)0;
3. result *= base;
4. return result;

Всі ці дії визначені для вбудованих типів. Однак якщо ви створите функцію toPower () для будь-якого класового типу, то в цьому випадку такий клас повинен буде включати загальнодоступні належать функції, які забезпечують такі можливості:

- дія 1 ініціалізує об'єкт типу T таким чином, що клас T повинен містити конструктор копіювання,

- дії 2 і 3 перетворять значення типу int в об'єкт типу T, тому клас T повинен містити конструктор з параметром типу int, оскільки саме таким способом в класах реалізується перетворення до класових типів,

- дія 4 використовує операцію * = над типом T, тому клас повинен містити власну функцію-operator * = ().

- дія 5 передбачає, що в типі T передбачена можливість побудови безпечної копії об'єкта, що повертається (див. Конструктор копіювання).

Схема такого класу виглядає наступним чином:

```
class T
{  public:
     T (const T &base); // конструктор копіювання
     T (int i); //приведення int до T operator *= (T base);
}
```

Рисунок 9.6 – Схема класу

Використовуючи класи в шаблонах функцій, переконайтеся в тому, що ви знаєте, які дії з ними виконуються в шаблоні функції, і чи визначено для класу ці дії. Якщо ви не забезпечили клас необхідними функціями, виникнуть різні незрозумілі повідомлення про помилки.

9.1.4.3 Ототожнення типів аргументів

Так як компілятор генерує екземпляри шаблонів функцій згідно типам, заданим при їх виклики, то критичним моментом є передача коректних типів, особливо якщо шаблон функції має два або більше параметрів. Хорошим прикладом є класична функція max ():

```
template <class T> T max (T a, T b)
{ return a > b ? a : b;
}
```

Рисунок 9.7 – Функція max ().

Функція max () буде працювати правильно, якщо обидва її аргументу мають один і той же тип:

```
int i = max (1, 2);
double d = max (1.2, 3.4);
```

Однак, якщо аргументи різних типів, то виклик max () призведе до помилки, так як компілятор не зможе зрозуміти, що йому робити.

Один з можливих способів для вирішення неоднозначності полягає в використанні приведення типів, щоб прояснити наші наміри:

```
int i = max ((int)'a', 100);
```

Друга можливість - це явно оголосити версію примірника шаблону функції перед її викликом:

```
int max (int, int);
int j = max ('a', 100);
```

Третій спосіб вирішити проблему полягає в створенні шаблону функцій, який має параметри різних типів.

```
template <class T1, class T2> T1 max (T1 a, T2 b)
{  return a > (T1)b ? a : (T1)b;
}
```

Використання цієї нової версії max () не призведе до неоднозначності в разі

використання двох разів-особистих типів. Наприклад, якщо написати «max ('a', 100);» то компілятор буде використовувати два заданих (за допомогою аргументів типу) і побудує версію функції max () з заголовком

char max (char, int);

Далі компілятор перед виконанням порівняння призведе тип другого аргументу до типу першого аргументу. Такий спосіб допустимо, однак використання двох типових параметрів в шаблоні функції, яка повинна була б працювати тільки з одним типом, часто лише ускладнює життя. Досить важко пам'ятати, що max ('a', 100) дає значення типу char, в той час як max (100, 'a') передає в зухвалу програму int.

9.1.4.5 Шаблони класів

Ви можете створювати шаблони і для класів, що дозволяє працювати з будь-якими типами даних. Класичним прикладом є контейнерні класи (тобто класи, що містять типи), наприклад безлічі. Використовуючи шаблон, можна створювати родової клас для множин, після чого породжувати конкретні безлічі - квітів, рядків і т.д.

Спочатку розглянемо приклад. Розглянутий далі приклад - клас, який зберігає пару значень. Функції-елементи цього класу повертають мінімальне і максимальне значення, а також дозволяють визначити, чи є два значення однаковими. Оскільки перед нами шаблон класу, то тип значення може бути майже будь-яким.

```
emplate <class T> class Pair

{          T a, b; public:

Pair (T t1, T t2); T Max();

T Min ();
int isEqual ();
};
```

Рисунок 9.8 – Шаблон класу

Поки все виглядає також витончено, як і для шаблонів функцій. Єдина різниця полягає в тому, що замість опису функції використовується оголошення класу. Шаблони класів стають все більш складними, коли ви описуєте належать функції класу. Ось, наприклад, опис належить функції Min () класу Pair:

```
template <class T> T Pair <T>::Min()
{  return a < b ? a : b;
}
```

Щоб зрозуміти цей запис, давайте повернемося трохи назад. Якби Pair був звичайним класом (а не шаблоном класу) і T був би деяким конкретним

типом, то функція Min класу Pair була б описана таким чином:

```
T Pair::Min()
{  return a < b ? a : b;
}
```

Для випадку шаблонної версії нам необхідно, по-перше, додати заголовок шаблону template <class T>

Потім потрібно дати ім'я класу. Пам'ятайте, що насправді ми описуємо безліч класів - сімейство Pair. Повторюючи синтаксис префікса (заголовка) шаблону, екземпляр класу Pair для цілих типів, можна назвати Pair <int>, екземпляр для типу double - Pair <double>, для типу Vector - Pair <Vector>. Однак в описі принад-лежить класу функції нам необхідно використовувати ім'я класу Pair <T>. Це має сенс, тому що заголовок шаблону каже, що T означає ім'я будь-якого типу.

Наведемо текст інших методів класу Pair. Описи методів поміщаються в заголовки, так як вони повинна бути видимі всюди, де використовується клас Pair.

Конструктор **template <class T>**

```
Pair <T>::Pair (T t1, T t2) : a(t1), b(t2) { }
```

Метод **Max**

```
template <class T> T Pair <T>::Max() {
   return a>b ? a : b;
}
```

Метод **isEqual**

```
template <class T> int Pair <T>::isEqual()
  { if (a==b) return 1;
    return 0;
}
```

Раніше вже зазначалося, що шаблони функцій можуть працювати тільки для тих (вбудованих) типів даних або класів, які підтримують необхідні операції. Те ж саме справедливо і для шаблонів класів. Щоб створити екземпляр класу Pair для деякого класового типу, наприклад для класу X, цей клас повинен містити наступні загальнодоступні функції

X (X &); // конструктор копіювання

int operator == (X)

int operator <(X);

Три зазначені функції необхідні, так як вони реалізують операції, що виконуються над об'єктами типу T в методу X шаблону класу Pair.

Якщо ви збираєтеся використовувати деякий шаблон класу, як дізнатися

які операції потрібні? Якщо шаблон класу забезпечений документацією, то ці вимоги повинні бути в ній зазначені. В іншому випадку доведеться читати первинну документацію - вихідний текст шаблону. При цьому враховуйте, що вбудовані типи даних підтримують всі стандартні операції.

9.1.4.6 Шаблони класів: не тільки для типів

Параметризрвані деякий клас так, щоб він працював для будь-якого типу даних - це тільки половина того, що шаблони забезпечують для класів. Інший аспект полягає в тому, щоб дати можливість завдання числових параметрів. Це дозволяє Вам, наприклад, створювати об'єкти типів "Вектор з 20 цілих", "Вектор з 1000 цілих" або "Вектор з 10 змінних типу double".

Основна ідея проста, хоча використовується синтаксис може здатися складним. Давайте як приклад розглянемо деякий узагальнений клас Vector. Як і клас Pair, клас Vector містить функції Min (), Max (), isEqual (). Однак в ньому може бути будь-яка кількість учасників, а не два. У шаблоні Vector замість цього використовується другий параметр заголовка шаблону:

```
template <class T, int n> class Vector {
public:
Vector();

~Vector() {delete[] coord;}
void newCoord (T x);
T Max ();
T Min();

        int isEqual();
private:

T *coord; int current;
};
```

Значення n, задане в заголовку шаблону не використовується в описі класу, але застосовується в описі його методів. Конструктор Vector, який використовує значення n для завдання розміру масиву, виглядає так:

Конструктор

```
template <class T, int n> Vector <T, n>::Vector():
        { coord = new T[n];
current = 0;
}
```

Метод **Max**

```
template <class T, int n>
T Vector <T, n>::Max():
{  T result (coord[0]);     // *
for (int i=0; i<n; i++)
if (result < coord[i])   // **
```

```
        result = coord[i];   // ***
        }
```

У конструкторі задається список ініціалізацій, що встановлюють початкові значення для двох елементів класу. Елемент coord инициализируется адресою динамічно розміщеного масиву розміром n і що складається з елементів типу T, а елемент current инициализируется значенням 0.

Знову, зауважимо, що в якості Т може виступати майже будь-який тип. Однак і в цьому випадку успішна реалізація можлива лише за певних умов - для об'єктів, чий тип передається в шаблон в якості параметра, повинні бути визначені наступні операції

- конструктор копіювання (*),
- оператор <(**), і> для методу Max (),
- оператор = (***).

Є кілька варіантів використання шаблонів з параметрами-значеннями для динамічного розміщення масивів різних розмірів. Наприклад, можна передати розмір масиву конструктору. Вказівка розмірів об'єкта під час конструювання або шляхом звернення до деякого методу дійсно забезпечує завдання розміру, проте такий спосіб не дозволяє створити окремий тип для кожного окремого розміру. Підхід з використанням шаблону гарантує, що розмір стає частиною типу. Так, Vector з п'ятьма елементами типу double є типом, відмінним від Vector з чотирма елементами типу double.

Чи хочете Ви, щоб різні розміри були різними типами, залежить від ваших потреб. Якщо порівняння двох векторів з чотирма і п'ятьма координатами не має особливого сенсу, то було б непогано зробити їх різними типами. Разом з тим, в разі класів для матриць, Ви, можливо, не захочете мати особливий тип для кожного розміру матриць, так як, наприклад, множення, працює з матрицями різних розмірів. Якщо Ви зіткнетеся з подібною проблемою, то Вам будуть потрібні тільки в розумних межах перевірки часу виконання, а не контроль типів при компіляції.

Хоча числові параметри шаблонів часто використовуються для завдання розмірів різних елементів, як це було показано для класу Vector, цим їх застосування не обмежується. Наприклад, за допомогою числових параметрів можна задавати діапазони значень графічних координат в графічному класі.

9.1.4.7 Наслідування в шаблонах класів

Шаблони класів, як і класи, підтримують механізм успадкування. Всі основні ідеї успадкування при цьому залишаються незмінними, що дозволяє побудувати ієрархічну структуру шаблонів, аналогічну ієрархії класів.

Розглянемо дуже простий приклад, на якому продемонструємо, яким чином можна створити шаблон класу, похідний з нашого шаблону класу Pair. Нехай це буде клас Trio, в якому до пари елементів a і b з Pair, додамо елемент c.

```
template <class T>
class Trio: public Pair <T>

{   T c;

public:

Trio (T t1, T t2, T t3);
...
};

template <class T>

Trio<T>::Trio (T t1, T t2, T t3): Pair <T> (t1, t2), c(t3) {}
```

9.2 Завдання для виконання лабораторної роботи

Для будь-якого варіанта завдання програма повинна забезпечувати вибір користувачем основних операцій з набором даних за варіантом згідно з таблицею 9.5:

- додавання;
- видалення;
- пошук (якщо це можливо);
- висновок на екран вмісту;
- повне видалення даних;
- деякі інші операції, які могли б знадобиться при реальній роботі з такими даними (наприклад, пошук по якогось певного умові або виконання операцій перетину або об'єднання з множинами (якщо реалізація завдання відбувається на базі використання множини) і т.д.)

Таблиця 9.5 – Таблиця з завданнями по варіантам

Варіант	Завдання
1	Використовуйте шаблон vector для масиву даних про магазинах.
2	Використовуйте шаблон list для двусвязного списку даних про маршрутках.
3	Використовуйте шаблон deque для обліку даних про черги авто на заправці.
4	Використовуйте шаблон Set для побудови двох множин цілих чисел і обчислення їх перетину і деяких інших операцій з множинами.
5	Використовуйте шаблон Multiset для підрахунку числа входжень кожного числа в безліч цілих чисел з повторами.
6	Використовуйте шаблон Map для виключення повторів серед безлічі цілих чисел.
7	Використовуйте шаблон multimap для виключення повторів

	комбінацій серед безлічі пар цілих чисел.
8	Використовуйте шаблон queue для черги жителів міста на отримання квартири.
9	Використовуйте шаблон priority_queue для черги замовлень на ремонт квартир, щоб обслуговувати найбільші замовлення в першу чергу.
0	Використовуйте шаблон stack для рядка дійсних чисел і виконання основних операцій з ним ..

Після того, як будуть реалізовані основні операції з даними, ознайомтеся з методами контейнера.

Додайте в програму додаткові операції, які будуть використовувати деякі з вивчених вами методів контейнерного класу, не використані при реалізації основних операцій.

9.2 Зміст звіту

- Назва роботи.
- Мета роботи.
- Теоретичні відомості, в яких міститься: короткий опис STL і її складових, визначення контейнера, визначення ітератора, опису поняття шаблону, короткий переченеь контейнерних класів STL з їх описом
- Текст завдання згідно варіанту.
- Докладний огляд контейнерного класу, з використанням якого вирішується завдання згідно варіанту (С описом його основних методів і структури, на базі якої він побудований).
- Рисунки із зображенням інтерфейсу програми і результати її тестування
- Текст програми з коментарями.
- Висновки про працездатність програми та особливості роботи з контейнерними класами STL.

Лабораторна робота №10

Алгоритми хешування. Розширення колізій.

Мета роботи:

– Вивчити особливості організації хеш-таблиць для різних типів алгоритмів хешування;
– Отримати практичні навички використання функцій рехешування для розширення колізій в хеш-таблицях;
– Створити консольний додаток, в якому реалізовані операції вставки, видалення і пошуку елементів і пошуку елементів в хеш-таблиці.

10.1 Короткі теоретичні відомості

10.1.1 Загальні поняття і визначення

Часто бувають потрібні динамічні множини, що підтримують тільки словникові операції - додавання, пошук і видалення елемента. Основне завдання організації такої множини - знайти структуру даних, що забезпечує мінімальні витрати на пошук.

У цьому випадку застосовують так зване хешування, відповідна структура даних називається «хеш-таблиця».

Отже, задана множина елементів організовується як звичайний масив. Однак місце розташування окремого елемента множини в масиві визначається не місцем розташування (порядку проходження) ключа відносно інших ключів, як у випадку відсортованої множини, а власне *значенням (вмістом) ключа цього елемента*.

Таким чином, занесення нового елемента в масив виконується за допомогою *функції перетворення ключів (розстановки, хешування)*, яка перетворює ключ елемента в індекс масиву.

У цьому випадку говорять *про хешируваних таблицях, таблицях з обчислюваними входами або таблицях з прямим доступом.*

Хешування - це спосіб відомості зберігання одної множини, до більш меншої, шляхом розбиття множини ключів на непересічні підмножини, що володіють певними властивостями. Ця властивість описується хеш-функцією і називається хеш-адресою.

Хеш-функція - функція, яка перетворює ключ в певний індекс в масиві.

Хеш-таблиця - це звичайний масив з адресацією, що задається хеш-функцією.

10.1.2 Приклад організації хеш-таблиці

Припустимо, що деяка фірма випускає деталі і кодує їх семизначними числами. Для застосування прямої індексації з використанням повного семизначного ключа потрібен був би масив з 100 мільйонів елементів. Зрозуміло, що це призвело б до втрати дуже великого простору, оскільки реально фірма випускає не більше 1000 деталей.

Таким чином, для зберігання інформації про деталі досить мати масив для зберігання 1000 елементів з діапазоном індексів від 0 до 999.

Для того, щоб мати можливість прямого доступу до елементу масиву по ключу, необхідний деякий метод перетворення ключа в відповідний індекс масиву всередині діапазону 0 - 999.

В якості індексу записи про виріб в цьому масиві використовуються три останні цифри номера вироба.

В даному випадку хеш-функція для занесення записів про деталі в таблицю може мати наступний вигляд:

```
int Hesh(int key)
{return key%1000}
```

Структура хеш-таблиці для розглянутого прикладу приведена на рисунку 10.1

Індекс	Ключ (код деталі)	Запис
0	4967000	
1	6584001	
2	8421002	
	. . .	
396	4618396	
397	4759397	
398	9572398	
399	1286399	
	. . .	
990	0000990	
991	0000991	
992	1200396	
993	0047993	
	. . .	
998	5682998	
999	0001999	

Рисунок 10.1 – Приклад організації хештаблиці для зберігання записів про деталі

10.1.3 Колізії і їх розширення

Основна проблема організації хеш-таблиць полягає в тому, що хеш-адреси двох різних ключів можуть збігатися. Наприклад, при додаванні в таблицю, наведену на рис. 10.1, записи з ключем 0596998 виникне така ситуація, що дана комірка вже зайнята.

Ситуація, коли два або більше ключа асоціюються з однією і тією ж коміркою таблиці називається **колізією** при хешуванні.

Слід зазначити, що гарною хеш-функцією є така функція, яка мінімізує колізії і розподіляє записи рівномірно по всій таблиці.

Для вирішення проблеми колізій необхідно визначити:

1) Якого виду хеш-функцію потрібно використовувати, щоб зменшити число колізій?

2) Що робити в разі виникнення колізій?

a) Використовувати іншу хеш-функцію або послідовність функцій;

b) Використовувати додаткову структуру даних (найчастіше упорядкований список), в якому розміщуються всі елементи множини за ключами, які претендують на один і той же індекс.

Зазначені два підходи вирішення колізій відображаються в різних типах алгоритмів хешування.

10.1.4 Типи алгоритмів хешування

Рисунок 10.2 – Типи алгоритмів хешування

Існує дві різні форми хешування. Одна з них називається відкритою, або зовнішньою, хешуванням і дозволяє зберігати множини в потенційно нескінченному просторі, знімаючи тим самим обмеження на розмір множин. Інша називається закритим, або внутрішнім, хешуванням і використовує обмежений простір для зберігання даних, обмежуючи таким чином розмір множин.

10.1.5 Основні операції з хеш-таблицею

- Вставка елемента.
- Пошук елемента.
- Видалення елемента
- Розширення хеш-таблиці

10.1.6 Функції хешування

Хеш-функція має вирішальний вплив на продуктивність хеш-таблиці. Гарна хеш-функція повинна якомога рівномірніше розподіляти ключі по всьому діапазону значень індексів. Це дозволяє скоротити довжину ланцюжків розміщення і час, що витрачається на пошук елементів в таблиці.

Найбільш прості і відомі алгоритми хешування ґрунтуються на таких функціях хешування:

- ділення;
- множення;
- зведення в квадрат;
- перетворення символьних ключів.

Функція хешування - ділення:

Отримання хеш-адреси елемента в таблиці засноване на використанні залишку від ділення числового еквівалента ключа **Key** на число **S**, що дорівнює або близьке до числа елементів (сегментів) хеш-таблиці **N**.

$$Hach (Key) = Key \% \ S$$

Отриманий цілий залишок приймається за індекс сегмента в таблиці. Для зменшення колізій необхідно значення **S** вибирати рівним простому числу.

Функція хешування - множення:

Обчислення робиться в два етапи:

1. Обчислення нормалізованої хеш-адреси в інтервалі [0 ... 1] за формулою:

$$H (Key) = (C * Key) \% \ 1$$

де **Key** - числовий еквівалент ключа, **C** - деяка константа з інтервалу [0 ... 1], **H (Key)** є дробовою частиною твору **C * Key.**

2. Обчислення кінцевого хеш-адреси за формулою:

$$Hach (Key) = Trunc (N * H (Key))$$

де N - розмір таблиці, а Trunc - стандартна функція отримання цілої частини числа в C ++.

Часто в якості константи використовують величину, рівну

$((\sqrt{5}) -1) / 2 = 0,6180339887$.

Функція хешування - зведення в квадрат:

Обчислення хеш-адреси відбувається за такою схемою:

Числовий еквівалент ключа зводиться в квадрат, а потім з отриманого числа вибирається певна кількість цифр, розташованих по середині. Кількість обираємих цифр залежить від розміру таблиці. Наприклад, якщо розмір таблиці N = 100, то вибираються дві цифри, що зададуть діапазон [0 ... 99].

Функція хешування - перетворення символьних ключів:

1 спосіб: підсумувати значення кодів відповідних символів ключа. Недолік - для різних рядків з однаковим набором символів буде виданий один і той же індекс.

2 спосіб: вихідний ключ розбивається на групи по 4-ри символу. Неповна група доповнюється зліва пропусками. Кожна група, наприклад група, що складається з 4-х символів abcd, обробляється як число на основі розміру таблиці кодування. Для повної таблиці символів ASCII (або ANSI) розміром в 256 рядків отримаємо:

a * 2563 + b * 2562 + c * 2561 + d * 2560

де символ, наприклад *a*, замінюється його кодом у відповідній таблиці символів.

Після перетворення всіх блоків в цілі числа вони підсумовуються, потім отримана сума зводиться в квадрат і з результату вибираються середні цифри.

10.1.7 Спискові алгоритми відкритого хешування

Рисунок 10.3 – Організація при відкритому хешуванню

На рис. 10.3 показана базова структура даних при відкритому хешуванні. Основна ідея полягає в тому, що потенційна множина (можливо, нескінченна) розбивається на кінцеве число класів. Для B класів, пронумерованих від 0 до B - 1, будується хеш-функція h така, що для будь-якого елемента x вихідної множини функція h (x) приймає цілочисельне значення з інтервалу 0, ..., B - 1, яке відповідає класу, якому належить елемент x. Елемент x часто називають ключем, h (x) - хеш-значенням x, а "класи" - сегментами. Кажуть, що елемент x належить сегменту h (x). Масив, званий таблицею сегментів і проіндексований номерами сегментів 0, 1, ..., B - 1, містить заголовки для B списків. Елемент x i-го списку - це елемент вихідної множини, для якого h (x) = i.

Для вирішення колізій в облікових алгоритмах відкритого хешування використовують списки. При цьому:

1) Множина ключів розбивається на кінцеве число сегментів M;
2) Будується функція хешування Hesh (key), яка по ключу видає значення в діапазоні 0..M-1;
3) Вводиться масив (таблиця сегментів) T з індексами 0..M-1;
4) Кожен i-й елемент масиву містить заголовок списку елементів таких, що Hesh (key) = i;

10.1.7.1 Основні функції спискових алгоритмів відкритого хешування

Рисунок 10.4 – Функції хешування та ініціалізації хеш-таблиці при списковому відкритому хешуванню

Рисунок 10.5 – Функція вставки елемента в хеш-таблицу при списоковому відкритому хешуванню

Рисунок 10.6 – Функція видалення елемента з хеш-таблиці при списковому відкритому хешуванню

Рисунок 10.7 – Функція пошуку елемента в хеш-таблиці при списковому відкритому хешуванню.

10.1.8 Алгоритми закритого хешування

При закритому хешуванні в таблиці сегментів зберігаються безпосередньо елементи словника, а не заголовки списків. Тому в кожному сегменті може зберігатися тільки один елемент словника. При закритому хешуванні застосовується методика повторного хешування (рехешування). Якщо ми спробуємо помістити елемент x в сегмент з номером *h (x),* який вже зайнятий іншим елементом (ситуація колізії), то відповідно до методики повторного хешування вибирається послідовність інших номерів сегментів *h1 (x), h2 (x), ...* , куди можна помістити елемент x. Кожне з цих місць розташування послідовно перевіряється, поки не буде знайдено вільне. Якщо вільних сегментів немає, то, отже, таблиця заповнена і елемент x вставити не можна. Передбачається, що спочатку вся хеш-таблиця порожня, тобто в кожен сегмент поміщено спеціальне значення *empty* (порожній), яке не збігається ні з одним елементом словника.

Для вирішення колізій при закритому хешуванні використовується поширені методи рехешування:

- Лінійне рехешування;
- Квадратичне рехешування;
- Рехешування множенням;
- Псевдовипадкове рехешування.

Рисунок 10.8 – Методи рехешування

10.1.8.1 Приклад реалізації алгоритма закритого хешування з лінійним рехешуванням

1. Перш за все необхідна ініціалізація масиву (вільний / зайнятий).
2. Для запису з ключем ***Key*** обчислити індекс = *Hesh (key).*
3. Якщо елемент *T_h [Hesh (key)]* вільний - помістити новий запис в даний елемент таблиці.
4. Якщо в елементі *T_h [Hesh (key)]* вже є дані, то виконати повторне хешування (рехеширування).

Тобто використовується чергова хеш-функція - вибирається новий осередок в масиві і так до тих пір, поки не буде знайдений вільний елемент з ознакою "вільний" або кількість проб перевищить максимальну кількість елементів в таблиці.

Для лінійного рехешірованія вибирається *Hesh_i (key)* за схемою:

$$Hesh_i\ (key) = (Hesh_0\ (key) + i)\%\ N$$

де *N* - число елементів в таблиці;
i = 0, 1, 2,. . . N-1 - номер проби.

Ланцюжок осередків (сегментів) - починаючи з елемента з *індексом Hesh_0 (key)* і закінчуючи сегментом, куди елемент реально розміщений називається ланцюжок розміщень.

Рисунок 10.9 – Функції хешування та ініціалізації хеш-таблиці при закритому хешуванні з лінійним рехешуванням

Якщо при пошуку шуканий елемент не виявиться в сегменті, куди його адресує функція, то це зовсім не означає, що такого елемента немає в таблиці. Для подальшого пошуку елемента необхідно за певною методикою виконати перебір сегментів таблиці. Виникає питання, коли слід припинити перебір. При цьому розрізняють 2 ситуації: коли елементи в таблиці можуть просто додаватися, і коли елементи в таблиці можуть додаватися і віддалятися.

Сегмент містить ознаку *empty* - якщо елементи в цьому сегменті ще не розміщувалися і ознаку *deleted* - якщо елемент в цьому сегменті вже розміщувався і був видалений.

Елемент з ознакою *deleted*: при пошуку означає продовження ланцюжка розміщень; при додаванні - порожній сегмент.

Рисунок 10.10 – Функція пошуку елемента в хеш-таблиці при закритому

хешуванні з лінійним рехешуванням

Рисунок 10.11 – Функція пошуку в хеш-таблиці місця для вставки елемента з ключом *key* при закритому хешуванні з лінійним рехешуванням

Рисунок 10.12 – Функція вставки елемента в хеш-таблицю при закритому хешуванні з лінійним рехешуванням

Рисунок 10.13 – Функція видалення елемента з хеш-таблиці при закритому хешуванні з лінійним рехешуванням

10.2 Завдання для виконання лабораторної роботи

У перелік операцій, який з'являється на екрані при запуску програми створіть пункти:

- ініціалізація хеш-таблиці;
- додавання елемента в хеш-таблицю;
- видалення елемента з хеш-таблиці;
- пошук елемента в хеш-таблиці;
- висновок вмісту хеш-таблиці на екран.

Продумайте сценарій роботи з хеш-таблицею, тобто послідовність виконуваних дій і виведення відповідних повідомлень про результати їх виконання: успішне запевнення, довжина ланцюжків розміщення і т.д.

Для занесення в таблицю використовуйте двозначні числові елементи.

Після кожної операції з хеш-таблицею, її новий вміст має автоматично виводитися на екран.

Таблиця 10.1 – Завдання для виконання проекту

Варіант	Алгоритм хешування	Функція хешування	Функція рехешування	Розмір хеш-таблиці	Додати елементів
0	Списковий алгоритм відкритого хешування	Ділення	-	10	15
1	Списковий алгоритм відкритого хешування	Множення	-	10	15

2	Списковий алгоритм відкритого хешування	Зведення до квадрату	-	10	15
3	Закрите хешування	Ділення	Лінійне	10	11
4	Закрите хешування	Множення	Лінійне	10	11
5	Закрите хешування	Множення	Лінійне	10	11
6	Закрите хешування	Множення	Квадратичне	10	11
7	Закрите хешування	Зведення до квадрату	Квадратичне	10	11
8	Закрите хешування	Умножение	Множимо	10	11
9	Закрите хешування	Зведення до квадрату	Множимо	10	11

10.3 Тестування проекту

Виконайте тестування проекту наступним чином:

1. У таблиці розмістіть значення елементів, доданих в таблицю.
2. Виконайте завдання, згідно варіанту, вручну і покажіть його результат в звіті;
3. Виконайте завдання за допомогою проекту і зображення інтерфейсу з результатом виконання помістіть в звіт.

10.4 Зміст звіту

- Назва роботи.
- Мета роботи.
- Рисунок із зображенням результату виконання кожної операції з хеш-таблицею.
- Текст програми з поясненнями у вигляді коментарів для всіх його основних елементів.

Результати та висновки тестування проекту.

Перелік використаної літератури

1. Kochan S.G. Programming in C/ Stephen G. Kochan – 3nd ed. – Sams Publishing, 2004. – 505 p.
2. Роберт Седжвик. Фундаментальные алгоритмы на C. Части 1 - 5. Анализ. Структуры данных. Сортировка. Поиск. Алгоритмы на графах. 2003 год. 1136 стр.
3. И. Красиков, И. Красикова. Алгоритмы. Просто как дважды два. 2007 год. 247 стр.
4. Альфред В. Ахо и др. Структуры данных и алгоритмы. 2000 год. 380 стр.
5. Алексеев В.Е., Таланов В.А. Графы. Модели вычислений. Структуры данных. Учебник. ННГУ, 2005 год. 307 стр.
6. Гашков С.Б., Чубариков В.Н., Садовничий В.А. Арифметика. Алгоритмы. Сложность вычислений. 3-е изд. испр. 2005 год.