Лабораторна робота № 4

Теоретичні відомості. Поняття про динамічну пам'ять та вказівники

У мові *C/C++* глобальні змінні або змінні, оголошені у мажах блоку є *статичними*. Це означає, що пам'ять для їх збереження виділяється на етапі компіляції програми і не змінюється протягом її виконання. Проте не завжди наперед можна визначити обсяг пам'яті, потрібної для збереження певних даних. В таких випадках використовуються *динамічні змінні*, пам'ять для яких виділяється (а також вивільняється) в процесі виконання програми.

Глобальні змінні зберігаються у ділянці пам'яті, яка називається *сегментом даних*. Локальні змінні, що використовуються у функціях та існують тільки протягом їх роботи зберігаються у *сегменті стеку*. Решта базової пам'яті є *динамічною* і використовується для збереження динамічних змінних. Динамічну пам'ять ще часто називають *купою* (англ. *heap* – купа).

Доступ до значення динамічної змінної здійснюється за її *адресою* у пам'яті. Для збереження адрес динамічних змінних використовуються спеціальні статичні змінні *посилального* типу – *вказівники* (англ. *pointer*). Значенням вказівника є адреса в області динамічної пам'яті, де зберігається певний елемент даних.

Синтаксис оголошення вказівників у мові *C/C++* є наступний:

```
<тип>* <ім'я вказівника>;
```

Кожний вказівник перед використанням потрібно ініціалізувати адресою змінної або значенням іншого вказівника. Для отримання адреси змінної унарна операція взяття адреси „&":

```
<ідентифікатор вказівника> = &<ідентифікатор змінної>;
<ідентифікатор вказівника 1> = <ідентифікатор вказівника 2>;
```

Для отримання значення, на яке посилається вказівник, використовують операцію *розіменування*:

```
<ідентифікатор змінної> = *<ідентифікатор вказівника>;
*<ідентифікатор вказівника> = <вираз>;
```

Часом виникають ситуації, коли вказівник не повинен посилатися на жоден елемент даних. В такому випадку йому надають значення NULL (0) (порожня адреса). Розіменування вказівника зі значенням NULL може призвести до непередбачуваних результатів.

У мові *C/C++* для вказівників допускаються такі арифметичні операції: інкремент/декремент, додавання/віднімання цілого числа, різниця двох вказівників. Ці операції використовуються переважно для роботи з масивами. Особливість додавання/віднімання до вказівника цілого числа (в т.ч. одиниці) полягає в тому, що його значення змінюється на величину цього числа, помноженого на розмір типу вказівника. Різницею двох вказівників є кількість елементів між комірками пам’яті, на які вони посилаються.

Вказівники можна також порівнювати одне з одним та зі значенням NULL.

Слід зауважити, що при всіх вищезгаданих операціях потрібно дотримуватися сумісності типів вказівників.

У мові *C/C++* для виділення та звільнення ділянки динамічної пам’яті використовуються операції new та delete:

```
<ідентифікатор вказівника> = new <тип>;
delete <ідентифікатор вказівника>;
```

Динамічні структури даних. Лінійні списки: загальні положення

До динамічних структур даних належать такі структури, розміри яких визначаються і можуть змінюватися протягом виконання програми. Основними видами динамічних структур є лінійні списки, нелінійні списки (дерева) та графи.

Зв'язний лінійний список – це набір однотипних компонентів, які зв'язані між собою за допомогою вказівників. Зв'язані лінійні списки бувають таких типів:

- однозв'язний лінійний список;
- двозв'язний лінійний список;
- стек;
- черга:
- дек.

Однозв'язний лінійний список – це список, в якому попередній компонент посилається на наступний. У випадку. коли в однозв'язному лінійному списку останній елемент посилається на перший, утворюється *однозв'язний циклічний список*.

Двозв'язний лінійний список – це список, в якому попередній компонент посилається на наступний, а наступний – на попередній. У випадку. коли в двозв'язному лінійному списку останній елемент посилається на перший, а перший – на останній, утворюється *двозв'язний циклічний список*.

Стек – це однозв'язний лінійний список, в якому компоненти додаються та видаляються лише з його вершини, тобто з початку списку.

Черга – це однозв'язний лінійний список, в якому компоненти додаються в кінець списку, а видаляються з вершини, тобто з початку списку.

Дек (англ. *Double ended queue* – двобічна черга) – це черга, в якій елементи можуть додаватися і видалятися з обох кінців.

Структура однозв'язного лінійного списку схематично зображена на рис. ??. Кожен компонент цього списку містить інформаційне поле `data` та поле-вказівник `next` на наступний компонент. Очевидно, що для двозв'язного лінійного списку потрібно ввести ще одне поле-вказівник `previous` на попередній компонент. Інформаційне поле може бути

змінною (змінними) будь-якого вбудованого чи складеного типів. Вказівник останнього компонента списку повинен мати значення `NULL` – ознака кінця списку. Окрім того потрібно ввести вказівник `head` на перший елемент у списку. Очевидно, що тип вказівника у лінійному списку повинен бути типом компонента цього списку. У мові *C/C++ структура* може містити компонент, що є вказівником на тип цієї ж структури:

```
struct Item
{
datatype data;
Item* next;
Item* previous; //у випадку двозв'язного списку
};
Item *head, *temp, *front, *rear;
```

Лінійні списки. Стек

Стек (англ. *stack*) – це однозв'язний лінійний список, в якому доступ (вставка/видалення) до його компонент здійснюється через початок (вершину – `head`) списку. Стек працює за принципом *LIFO* (від англ. *Last In First Out* – останній прийшов – перший пішов).

Алгоритм створення стеку.

Для створення порожнього стеку достатньо вказівнику `head` присвоїти значення `NULL`.

Алгоритм додавання елемента у стек (функція `push(…)`*).*

1. Виділити динамічну пам'ять для нового елементу: (`Item *temp = new Item;`).
2. Заповнити інформаційне поле нового елемента: `(temp->data = data;)`.
3. Зв'язати новий елемент з вершиною стеку: `(temp->next = head;)`.
4. Встановити вершину стеку на новостворений елемент: `(head = temp;)`.

Алгоритм видалення елементу зі стеку (функція `pop(…)`*).*

1. Перевірити, чи стек не є порожній. Якщо стек порожній (`head == NULL`) – видати відповідне повідомлення і вийти з функції.
2. Створити копію вказівника на вершину стеку (`Item *temp = head;`).
3. Перемістити вказівник вершини стеку на наступний елемент (`head = temp->next;`).
4. Звільнити пам'ять колишньої вершини стеку (`delete temp`).

Алгоритм відображення вмісту стеку (функція `show(…)`*).*

1. Створити тимчасовий вказівник і присвоїти йому значення вершини стеку (`Item *temp = head;`).
2. Поки не досягнуто кінця стеку (`temp != NULL`) відобразити значення інформаційного поля `data` та перевести тимчасовий вказівник на наступний елемент стеку (`temp = temp->next;`).

Хід роботи:

Частина 1.

1. Створити заголовний файл `List.h` для визначення інтерфейсу функцій для роботи з лінійними списками.
2. У файлі `List.h` за допомогою команди `typedef` зв'язати тип інформаційного поля компоненти списку `datatype` з типом даних, заданим викладачем.
3. У файлі `List.h` оголосити структуру `Item` що задає компоненту лінійного списку.
4. Створити файл `List.cpp`, у якому, згідно описаних вище алгоритмів, реалізувати функції `push(…)`, `pop(…)` та `show(…)` для додавання, видалення елементів стеку та відображення його вмісту. Функція

`push(…)` повинна містити параметр для заповнення інформаційного поля елементу, що додається у стек.

5. Занести прототипи всіх створених функцій у заголовний файл `List.h`.
6. Створити файл з функцією `main()`, яка створює порожній стек та реалізує меню для додавання, видалення елементів стеку та виходу з програми. Після кожного додавання/видалення елементів стеку потрібно відображати його вміст.
7. Відкомпілювати проект та продемонструвати його роботу для набору даних, отриманих від викладача.

Лінійні списки. Черга

Черга (англ. *queue*) – це однозв'язний лінійний список, в якому вставка елементів здійснюється у її кінець (`rear`), а видалення – через початок (`front`). Черга працює за принципом *FIFO* (від англ. *First In First Out* – перший прийшов – перший пішов).

Алгоритм створення черги.

Для створення порожньої черги достатньо вказівникам `front` та `rear` присвоїти значення `NULL`.

Алгоритм додавання елемента у чергу (функція `enqueue(…)`*).*

1. Виділити динамічну пам'ять для нового елементу: (`Item *temp = new Item;`).
2. Заповнити інформаційне поле нового елемента (`temp->data = data;`).
3. Встановити новий елемент останнім у черзі (`temp->next = NULL;`).
4. Якщо черга порожня, то встановити її вершину на новостворений елемент (`front = temp;`).

5. Якщо черга не порожня, то зв'язати її кінець з новоствореним елементом (`rear->next = temp;`).
6. Встановити вказівник кінця черги на новостворений елемент: (`rear = temp;`).

Алгоритм видалення елементу з черги (функція `dequeue(…)`*).*

Алгоритм видалення елементу з черги є аналогічним до видалення елементу зі стеку (вказівник `head` замінюється на `front`).

Для відображення вмісту черги можна використовувати раніше створену функцію `show(…)`.

Хід роботи:

Частина 2.

1. Доповнити файл `List.cpp` реалізаціями функцій для додавання та видалення елементів черги згідно описаних вище алгоритмів (функції `enqueue(…)` та `dequeue(…)`). Функція `enqueue(…)` повинна містити параметр для заповнення інформаційного поля елементу, що додається у чергу.
2. Занести прототипи всіх створених функцій у заголовний файл `List.h`.
3. У файлі, що містить функцію `main()` створити порожню чергу та розширити меню пунктами додавання та видалення елементів черги. Після кожного додавання/видалення елементів черги потрібно відображати її вміст.
4. Відкомпілювати проект та продемонструвати його роботу для набору даних, отриманих від викладача.

Лінійні списки. Двозв'язний список.

Розглянемо роботу з двозв'язними лінійними списками. Окрім описаних вище функцій додавання та видалення елементів з початку

списку (робота зі стеком) та додавання елементу в кінець списку (робота з чергою) для даної структури можливі операції додавання/видалення елементу всередині списку та видалення елементу з кінця списку. Окрім того, для двозв'язного списку при кожній операції з його елементом, потрібно ще ініціалізовувати вказівник на попередній елемент (`previous`). Для вставки елемента всередині списку потрібно вказувати, між якими з уже існуючих елементів його потрібно вставляти, а для видалення – який елемент з існуючих у списку видаляти. Тому додатково потрібно ще створити функцію пошуку, яка б знаходила потрібний елемент у списку і повертала його адресу. Враховуючи вищесказане реалізуємо такі функції для роботи з двозв'язними лінійними списками:

- `add_begin(…)` – додавання елементу в початок списку;
- `add_end(…)` – додавання елементу в кінець списку;
- `del_begin(…)` – видалення елементу з початку списку;
- `del_end(…)` – видалення елементу з кінця списку;
- `search(…)` – пошук елементу у списку за ключем;
- `add_mid(…)` – додавання елементу у список після знайденого елементу;
- `del_mid(…)` – видалення знайденого елементу зі списку.

Вказівники на початок і кінець списку будемо позначати `first` і `last`.

Алгоритм створення списку.

Для створення порожнього списку достатньо вказівникам `first` та `last` присвоїти значення `NULL`.

Алгоритм додавання елемента в початок списку (функція `add_begin(…)`*).*

1. Виділити динамічну пам'ять для нового елементу: (`Item *temp = new Item;`).

2. Заповнити інформаційне поле нового елемента: `(temp->data = data;)`.
3. Зв'язати новий елемент з початком списку як з наступним елементом: `(temp->next = first;)`.
4. Встановити вказівник нового елементу на попередній як NULL (`temp->previous = NULL;`).
5. Якщо список не порожній, то зв'язати перший елемент з новостворени м як з попереднім (`first->previous = temp;`).
6. Якщо список порожній, то встановити вказівник кінця списку на новостворений елемент (`last = temp;`)
7. Встановити вказівник початку списку на новостворений елемент `(first = temp;)`.

Алгоритм додавання елемента в кінець списку (функція `add_end(…)`*).*

1. Виділити динамічну пам'ять для нового елемента: (`Item *temp = new Item;`).
2. Заповнити інформаційне поле нового елемента `(temp->data = data;)`.
3. Встановити новий елемент останнім у списку (`temp->next = NULL;`).
4. Зв'язати новостворений елемент з попереднім (`temp->previous = last;`)
5. Якщо список не порожній, то зв'язати його кінець з новостворени м елементом як з наступним (`last->next = temp;`).
6. Якщо список порожній, то встановити вказівник початку списку на новостворений елемент (`first = temp;`).
7. Встановити вказівник кінця списку на новостворений елемент: (`last = temp;`).

Алгоритм видалення елементу з початку списку (функція `del_begin(…)`*).*

1. Перевірити, чи список не є порожній. Якщо список порожній (`first == NULL`) – видати відповідне повідомлення і вийти з функції.
2. Створити копію вказівника на початок списку (`Item *temp = first;`).
3. Перемістити вказівник початку списку на наступний елемент (`first = temp->next;`).
4. Якщо список не порожній (видаляється не останній елемент), то встановити вказівник початку на попередній як NULL (`first->previous = NULL;`)
5. Якщо список порожній (видаляється останній елемент), то встановити вказівник кінця списку як NULL (`last = NULL;`)
6. Звільнити пам'ять колишнього першого елементу (`delete temp`).

Алгоритм видалення елементу з кінця списку (функція `del_end(…)`*).*

1. Перевірити, чи список не є порожній. Якщо список порожній (`last == NULL`) – видати відповідне повідомлення і вийти з функції.
2. Створити копію вказівника на кінець списку (`Item *temp = last;`).
3. Перемістити вказівник кінця списку на попередній елемент (`last = temp->previous;`).
4. Якщо список не порожній (видаляється не останній елемент), то встановити вказівник кінця списку на наступний як NULL (`last->next = NULL;`)
5. Якщо список порожній (видаляється останній елемент), то встановити вказівник початку списку як NULL (`first = NULL;`)
6. Звільнити пам'ять колишнього останнього елементу (`delete temp`).

Алгоритм пошуку елементу у списку за ключем (функція `search(…)`)

1. Створити тимчасовий вказівник і присвоїти йому значення початку списку (`Item *temp = first;`).
2. Поки не досягнуто кінця списку (`temp != NULL`): якщо інформаційне поле співпало з ключем, то функція повертає вказівник на нього, інакше перевести тимчасовий вказівник на наступний елемент (`temp = temp->next;`).
3. Якщо шуканий елемент не знайдено – видати відповідне повідомлення та завершити роботу.

Алгоритм додавання елементу у список <u>після</u> знайденого елементу (функція `add_mid(…)`)

1. Виконати пошук елементу у списку за заданим ключем.
2. Якщо елемент не знайдено – вийти з функції.
3. Якщо знайдений елемент знаходиться вкінці списку, то виконати функцію додавання в кінець списку (`add_end(…)`).
4. Якщо знайдений елемент знаходиться не вкінці списку, то:
5. Створити вказівник на знайдений за ключем елемент (`Item *pkey = search(…)`).
6. Виділити динамічну пам’ять для нового елементу (`Item *temp = new Item;`).
7. Заповнити інформаційне поле новоствореного елементу (`temp->data = data;`).
8. Зв’язати новостворений елемент з наступним після знайденого (як з наступним) (`temp->next = pkey->next;`)
9. Зв’язати новостворений елемент зі знайденим як з попереднім (`temp->previous = pkey;`).
10. Зв’язати знайдений з новостворением як з наступним (`pkey->next = temp;`)
11. Зв’язати наступний після знайденого з новостворением як з попереднім (`(temp->next)->previous = temp;`).

Алгоритм видалення знайденого елементу зі списку (функція `del_mid(…)`)

1. Виконати пошук елементу у списку за заданим ключем.
2. Якщо елемент не знайдено – завершити роботу.
3. Якщо знайдений елемент знаходиться з початку списку, то виконати функцію видалення з початку списку (`del_begin(…)`).
4. Якщо знайдений елемент знаходиться в кінці списку, то виконати функцію видалення з кінця списку (`del_end(…)`).
5. Інакше:
6. Зв'язати попередній від знайденого з наступним від знайденого (як наступний) (`(pkey->previous)->next = pkey->next;`).
7. Зв'язати наступний від знайденого з попереднім від знайденого (як попередній) (`(pkey->next)->previous = pkey->previous;`)
8. Звільнити пам'ять від знайденого за ключем елемента.

Хід роботи

Частина 3.

1. Доповнити файл `List.cpp`, відповідно до описаних вище алгоритмів, реалізаціями функцій:
   - `add_begin(…)` – додавання елементу в початок списку;
   - `add_end(…)` – додавання елементу в кінець списку;
   - `del_begin(…)` – видалення елементу з початку списку;
   - `del_end(…)` – видалення елементу з кінця списку;
   - `search(…)` – пошук елементу у списку за ключем;
   - `add_mid(…)` – додавання елементу у список після знайденого елементу;
   - `del_mid(…)` – видалення знайденого елементу зі списку.

Функції для додавання даних у список повинні містити параметр для заповнення інформаційного поля елементу, що додається. Функції додавання/видалення з середини списку повинні містити у собі виклик функції пошуку за заданим ключем (який також передається як параметр).

2. Занести прототипи всіх створених функцій у заголовний файл `List.h`. В цьому файлі модифікувати структуру `Item`, додавши до неї поле `previous` для зв'язку з попереднім елементом списку.
3. Створити новий файл з функцією `main()`, яка створює порожній двозв'язний список та реалізує меню для всіх вищеописаних функцій та виходу з програми. Після кожного додавання/видалення елементів списку потрібно відображати його вміст.
4. Відкомпілювати проект та продемонструвати його роботу для набору даних, отриманих від викладача.