Лекція 19.
Динамічні структури даних

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

План на сьогодні

1. Однозв’язний список
2. Двозв'язний список

Недоліки динамічних масивів

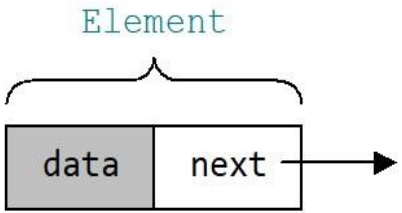
- **Ефективність при додаванні або видаленні елементів (O(n)):**
  Якщо потрібно додавати або видаляти елементи в середині масиву або на початку, це вимагає переміщення всіх наступних елементів, що може бути дуже витратно з точки зору часу.

- **Обмеження за розміром:**
  Масиви мають фіксований розмір, що може бути проблемою, коли потрібно змінювати розмір масиву. У разі, якщо масив переповнюється, потрібно виділяти нову пам'ять і копіювати всі елементи в новий масив (це може бути дорого).

Однозв’язний список

Однозв’язний список

Однозв'язний список — це основна структура даних, яка складається з **вузлів**, де кожен вузол містить **поле даних** і **посилання на наступний вузол у списку**. Посилання на наступний вузол останнього елемента в списку є **null**, що вказує на кінець списку. Однозв'язний список підтримує ефективні операції вставки та видалення.

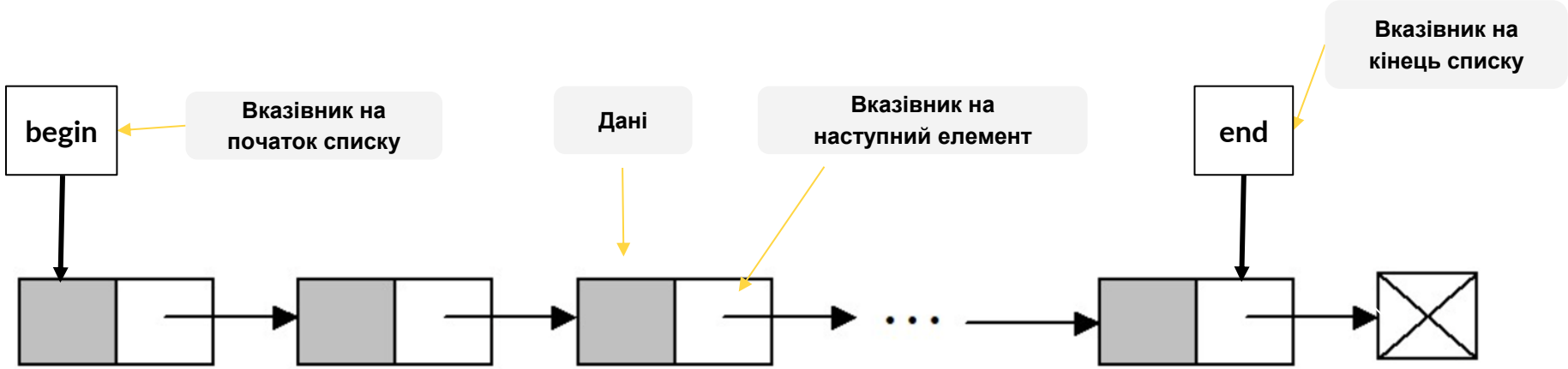



```
template <class T>
struct Element
{
  T data;
  Element* next;
};
```

Однозв’язний список

Кожен елемент (вузол) однозв’язного списку складається з двох частин:

- **Дані** (це може бути змінна будь-якого примітивного типу (int, double, char, …), об’єкт класу чи складна структура. Дані можуть складатися з декількох полів (змінних);
- **Адреса або позиція наступного елементу**

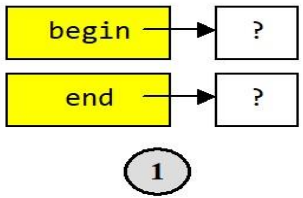


Створення порожнього списку

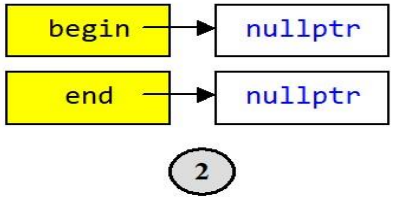
- оголошення вказівників на початок та кінець списку (begin, end);
- встановлення нульових значень вказівника;
- встановлення довжини списку count у нульове значення.

```
template <class T>
class List
{
private:
    Element<T>* begin; // вказівник на початок списку
    Element<T>* end; // вказівник на кінець списку
    size_t count; // к-сть елементів у списку
public:

    List()
    {
        // На початку список порожній
        begin = end = nullptr;
        count = 0;
    }
```

```
Element<T>* begin;
Element<T>* end;
int count;
```




```
begin = nullptr;
end = nullptr;
count = 0;
```



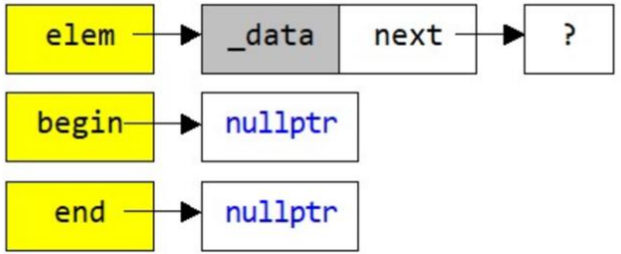
Додавання нового елементу в кінець порожнього

- ❑ **Створити новий елемент. Потрібно виділити пам’ять для нового елементу та заповнити її деякими даними data. Для цього оголошується вказівник (наприклад, elem)**
- ❑ Встановити вказівник next в нульове значення
- ❑ Встановити вказівник begin та end рівними значенню вказівника elem
- ❑ Збільшити кількість елементів у списку на 1

1

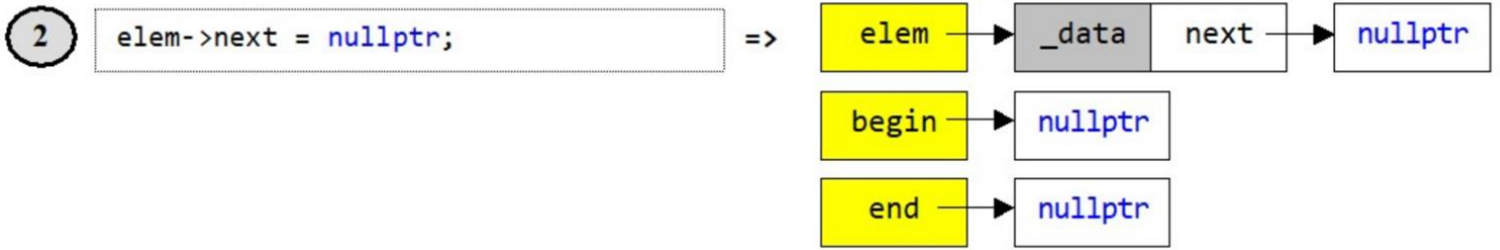
```
Element<T>* elem = new Element<T>;
elem->data = _data;
```

=>



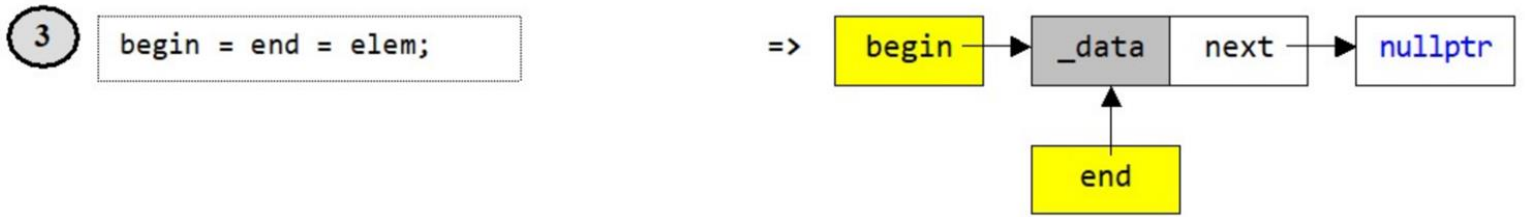
Додавання нового елементу в кінець порожнього списку

- ❏ Створити новий елемент. Потрібно виділити пам’ять для нового елементу та заповнити її деякими даними data. Для цього оголошується вказівник (наприклад, elem)
- ❏ **Встановити вказівник next в нульове значення**
- ❏ Встановити вказівник begin та end рівними значенню вказівника elem
- ❏ Збільшити кількість елементів у списку на 1



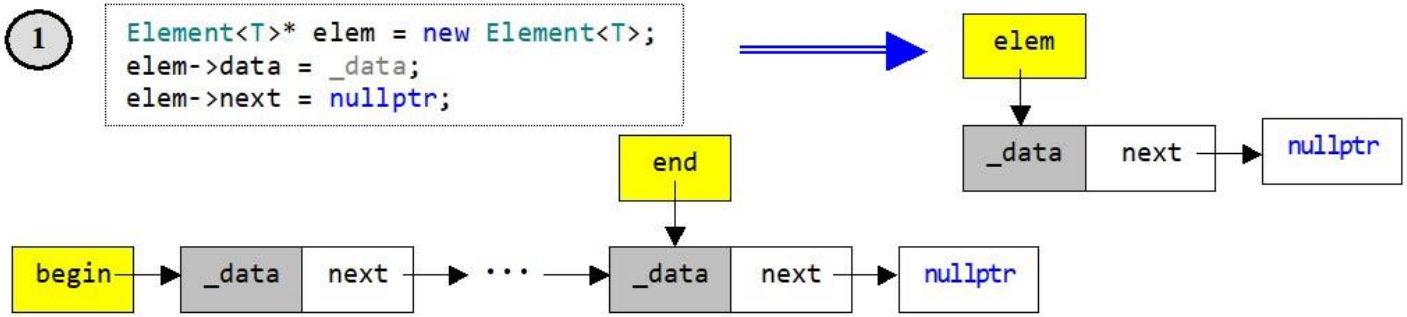
Додавання нового елементу в кінець порожнього списку

- ❏ Створити новий елемент. Потрібно виділити пам’ять для нового елементу та заповнити її деякими даними data. Для цього оголошується вказівник (наприклад, elem)
- ❏ Встановити вказівник next в нульове значення
- ❏ **Встановити вказівник begin та end рівними значенню вказівника elem**
- ❏ Збільшити кількість елементів у списку на 1



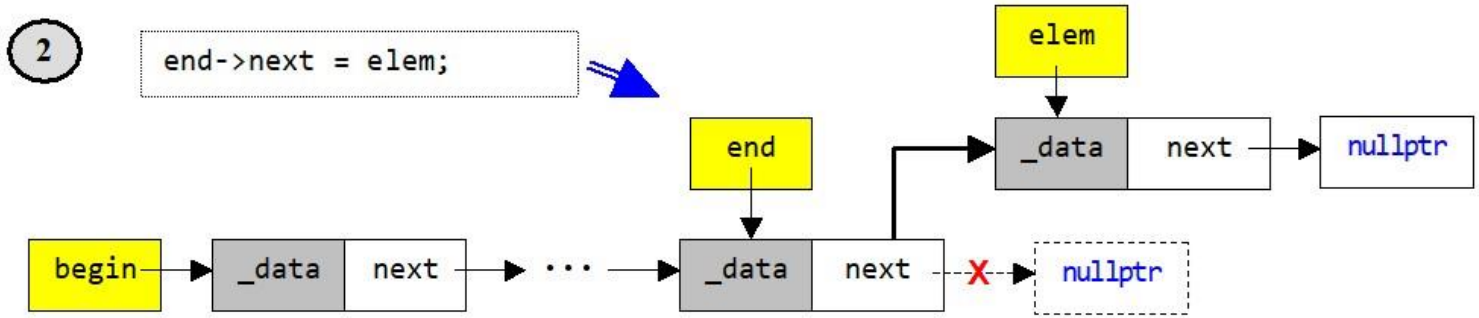
Додавання елементу в кінець існуючого списку

- **Створити новий елемент та заповнити його даними. Заповнити поля data та next. Створення здійснюється таким самим чином як у випадку з порожнім списком**
- Встановити вказівник next елементу, на який вказує вказівник end, в значення адреси елементу elem
- Встановити вказівник end рівним значенню вказівника elem
- Збільшити кількість елементів у списку на 1



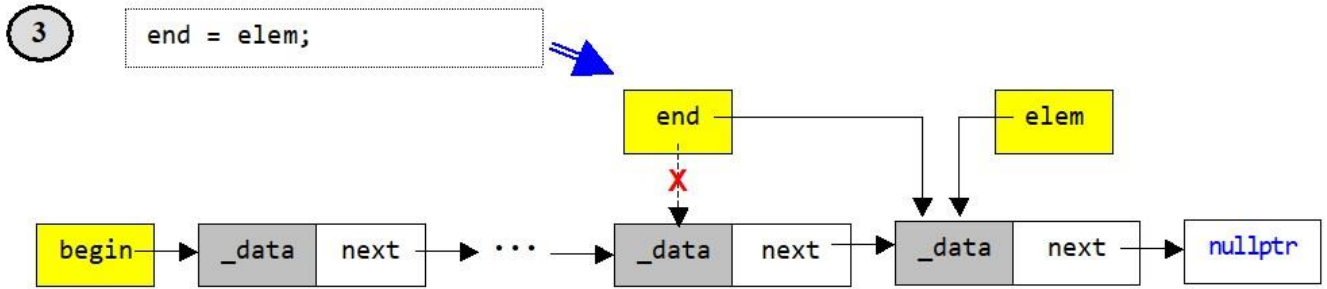
Додавання елементу в кінець існуючого списку

- ❏ Створити новий елемент та заповнити його даними. Заповнити поля data та next. Створення здійснюється таким самим чином як у випадку з порожнім списком
- ❏ **Встановити вказівник next елементу, на який вказує вказівник end, в значення адреси елементу elem**
- ❏ Встановити вказівник end рівним значенню вказівника elem
- ❏ Збільшити кількість елементів у списку на 1



Додавання елементу в кінець існуючого списку

- ❏ Створити новий елемент та заповнити його даними. Заповнити поля data та next. Створення здійснюється таким самим чином як у випадку з порожнім списком
- ❏ Встановити вказівник next елементу, на який вказує вказівник end, в значення адреси елементу elem
- ❏ **Встановити вказівник end рівним значенню вказівника elem**
- ❏ Збільшити кількість елементів у списку на 1



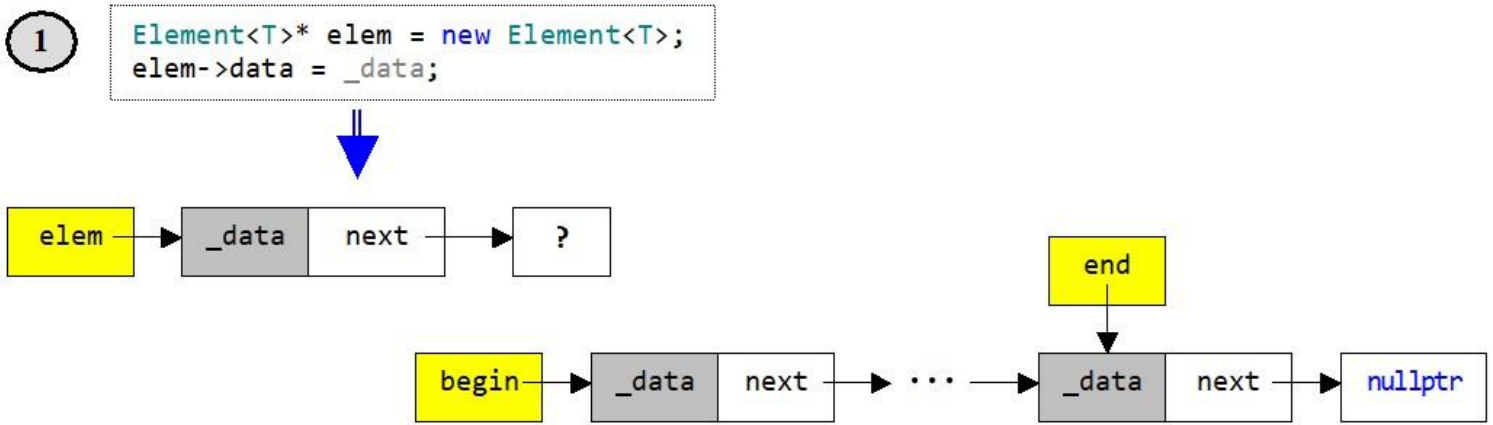
Додавання елементу в кінець списку

```
// Базовий вузол - елемент
template <class T>
struct Element
{
    T data; // дані
    Element* next; // наступний елемент

    Element(T _data = T(), Element* _next = nullptr)
        :data(_data), next(_next) {}
};
```

```
// Додати новий елемент в кінець списку
void addToEnd(T _data)
{

    // 1. Створити новий елемент та заповнити його даними
    Element<T>* elem = new Element<T>(_data); // останній елемент у списку
    // 2. Додати елемент
    // Визначити, чи елемент перший
    if (begin == nullptr)
    {
        // якщо елемент перший у списку
        begin = end = elem;
    }
    else
    {
        // якщо елемент не перший у списку,
        // то додати його в кінець списку.
        end->next = elem;
        end = elem;
    }
    // 3. Збільшити к-сть елементів у списку
    ++size;

}
```

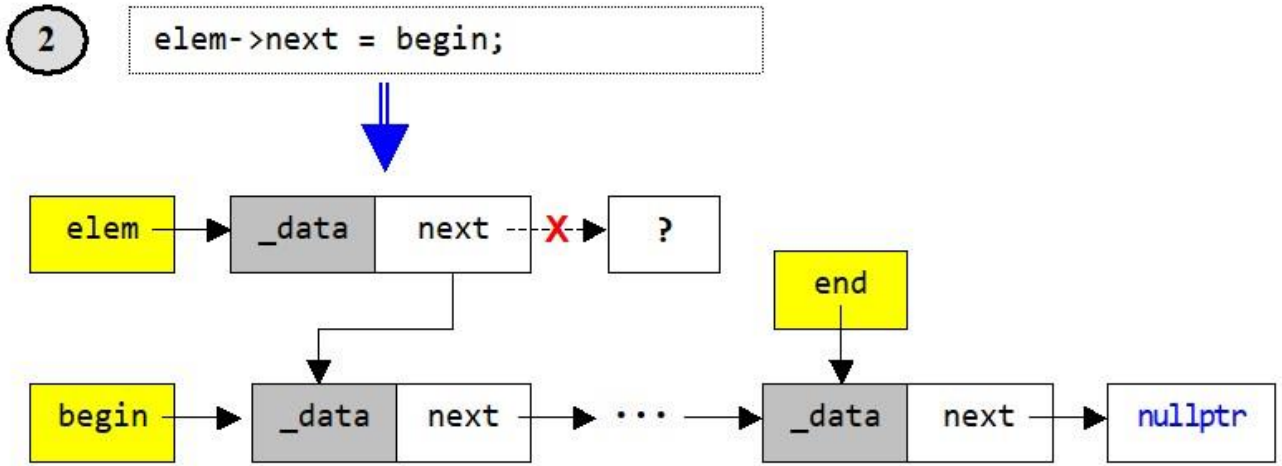
Додавання елементу на початок списку

- **Створити новий елемент. Виділити пам’ять під новий елемент. Заповнити елемент даними**
- Встановити вказівник next елементу на початок списку
- Встановити початок списку на нову комірку



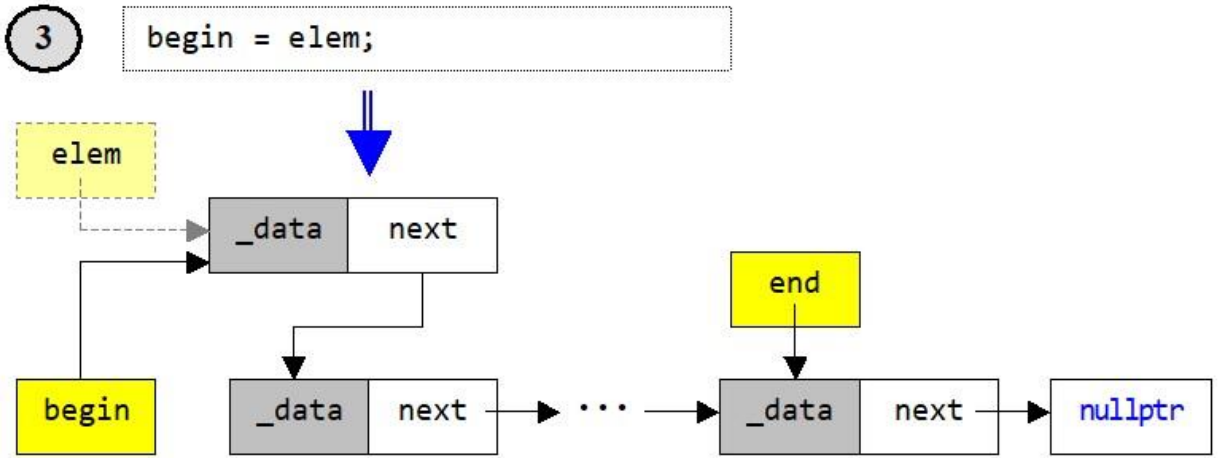
Додавання елементу на початок списку

- ❏ Створити новий елемент. Виділити пам’ять під новий елемент. Заповнити елемент даними
- ❏ **Встановити вказівник next елементу на початок списку**
- ❏ Встановити початок списку на нову комірку



Додавання елементу на початок списку

- ❏ Створити новий елемент. Виділити пам’ять під новий елемент. Заповнити елемент даними
- ❏ Встановити вказівник next елементу на початок списку
- ❏ **Встановити початок списку на нову комірку**



Додавання елементу на початок списку

```
void addToBegin(int _data)
{
    // 1. Створити новий елемент та заповнити його даними
    Element<T>* elem = new Element<T>(_data, begin); //перший елемент у списку

    if (begin == nullptr)
    {
        // якщо елемент перший у списку
        begin = end = elem;
    }
    else
    {
        // якщо елемент не перший, то додати на початок
        elem->next = begin;
        begin = elem;
    }
    size++;
}
```

Вставка елементу в середину списку

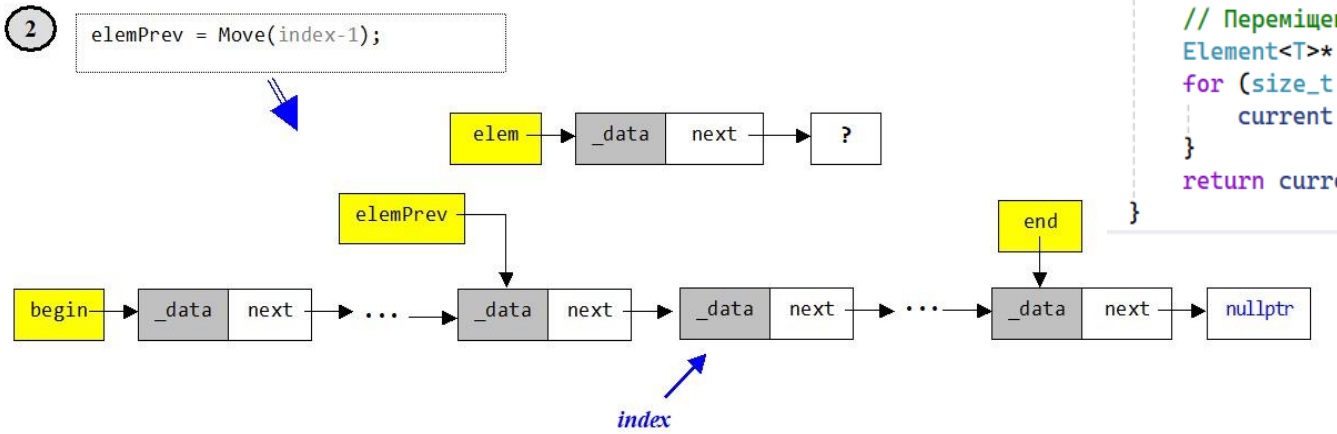
- ❏ **Створити елемент. Заповнити поле даних елементу**
- ❏ Визначити позицію елементу, що слідує перед позицією вставки елементу. Отримати вказівник на цю позицію.
- ❏ Встановити вказівник next елементу elem, який вставляється
- ❏ Встановити вказівник next попереднього елементу elemPrev так, щоб він вказував на елемент elem, що вставляється.

1

```
Element<T>* elem = new Element<T>;
elem->data = _data;
```

Вставка елементу в середину списку

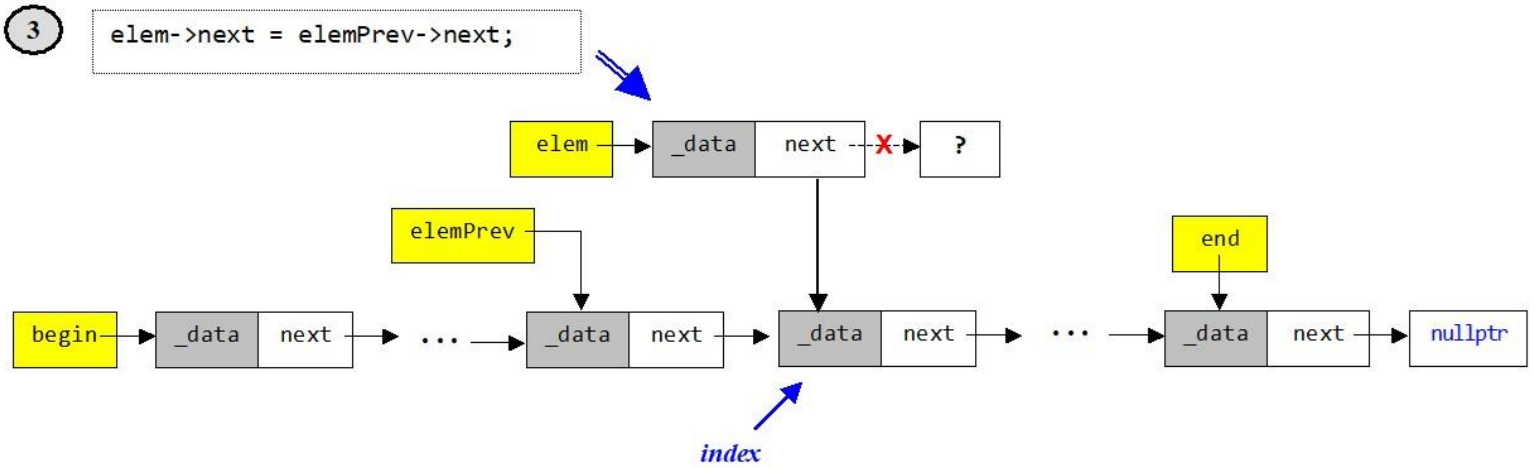
- ❑ Створити елемент. Заповнити поле даних елементу
- ❑ **Визначити позицію елементу, що слідує перед позицією вставки елементу. Отримати вказівник на цю позицію**
- ❑ Встановити вказівник next елементу elem, який вставляється
- ❑ Встановити вказівник next попереднього елементу elemPrev так, щоб він вказував на елемент elem, що вставляється.




```
Element<T>* moveTo(size_t index) {
    if (index >= size) return nullptr;
    // Переміщення вказівника на позицію index
    Element<T>* current = begin;
    for (size_t i = 0; i < index; ++i) {
        current = current->next;
    }
    return current;
}
```

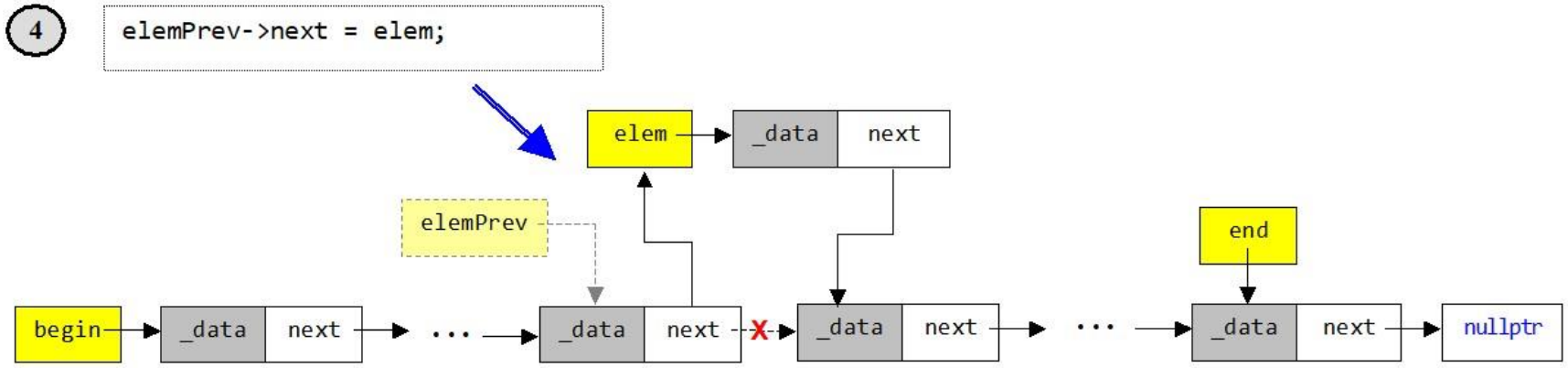
Вставка елементу в середину списку

- ❏ Створити елемент. Заповнити поле даних елементу
- ❏ Визначити позицію елементу, що слідує перед позицією вставки елементу. Отримати вказівник на цю позицію.
- ❏ **Встановити вказівник next елементу elem, який вставляється**
- ❏ Встановити вказівник next попереднього елементу elemPrev так, щоб він вказував на елемент elem, що вставляється.



Вставка елементу в середину списку

- ❏ Створити елемент. Заповнити поле даних елементу
- ❏ Визначити позицію елементу, що слідує перед позицією вставки елементу. Отримати вказівник на цю позицію.
- ❏ Встановити вказівник next елементу elem, який вставляється
- ❏ **Встановити вказівник next попереднього елементу elemPrev так, щоб він вказував на елемент elem, що вставляється.**

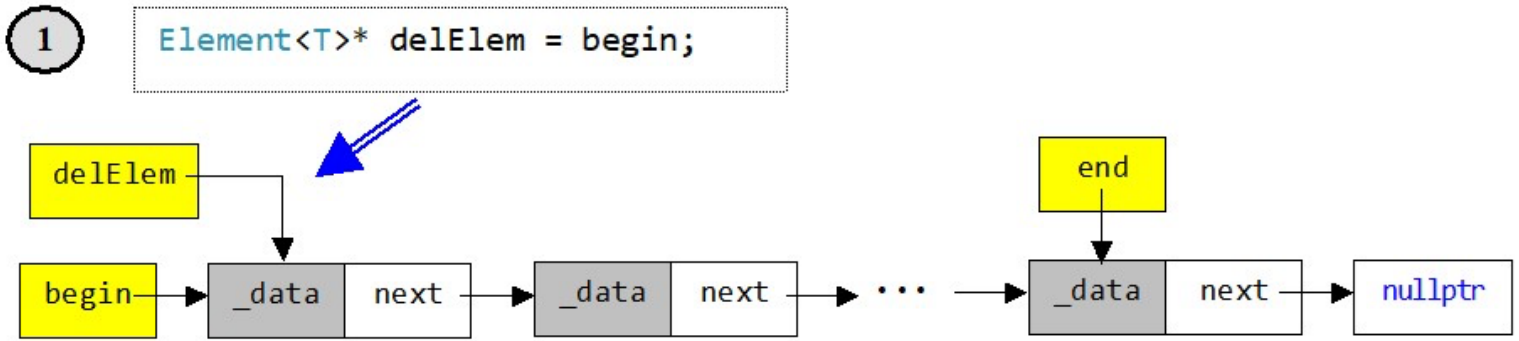


Вставка елементу в середину списку

```
// Вставка елемента на позицію index
void insertAt(size_t index, T _data) {
    if (index > size) {
        std::cerr << "Індекс виходить за межі списку!\n";
        return;
    }

    if (index == 0) {
        addToBegin(_data);
        return;
    }
    if (index == size) {
        addToEnd(_data);
        return;
    }

    Element<T>* prev = moveTo(index - 1);
    if (!prev) return;

    Element<T>* elem = new Element<T>(_data, prev->next);
    prev->next = elem;
    ++size;
}
```

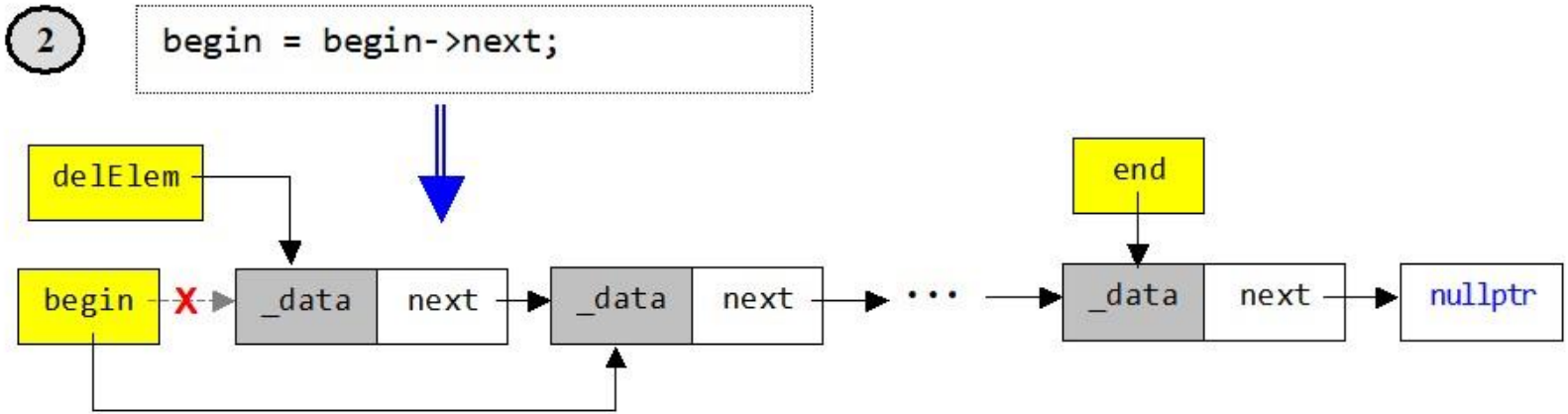
Видалення першого елементу зі списку

- ❏ **Отримати адресу першого елементу**
- ❏ Перемістити вказівник на початок списку begin на наступний елемент
- ❏ Звільнити пам’ять, виділену для елементу delElem



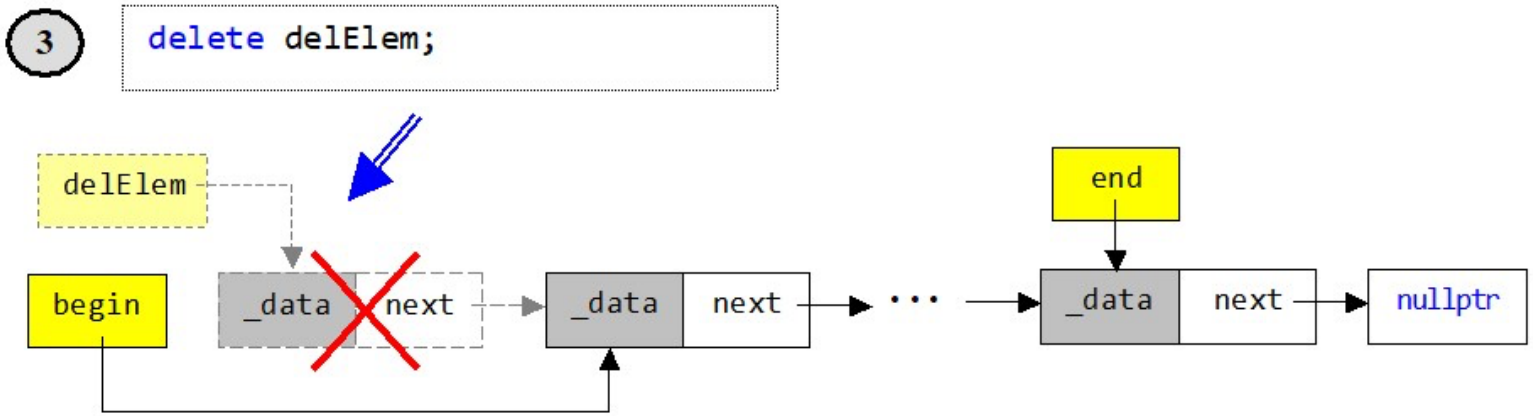
Видалення першого елементу зі списку

- ❏ Отримати адресу першого елементу
- ❏ **Перемістити вказівник на початок списку begin на наступний елемент**
- ❏ Звільнити пам’ять, виділену для елементу delElem



Видалення першого елементу зі списку

- ❏ Отримати адресу першого елементу
- ❏ Перемістити вказівник на початок списку begin на наступний елемент
- ❏ **Звільнити пам'ять, виділену для елементу delElem**

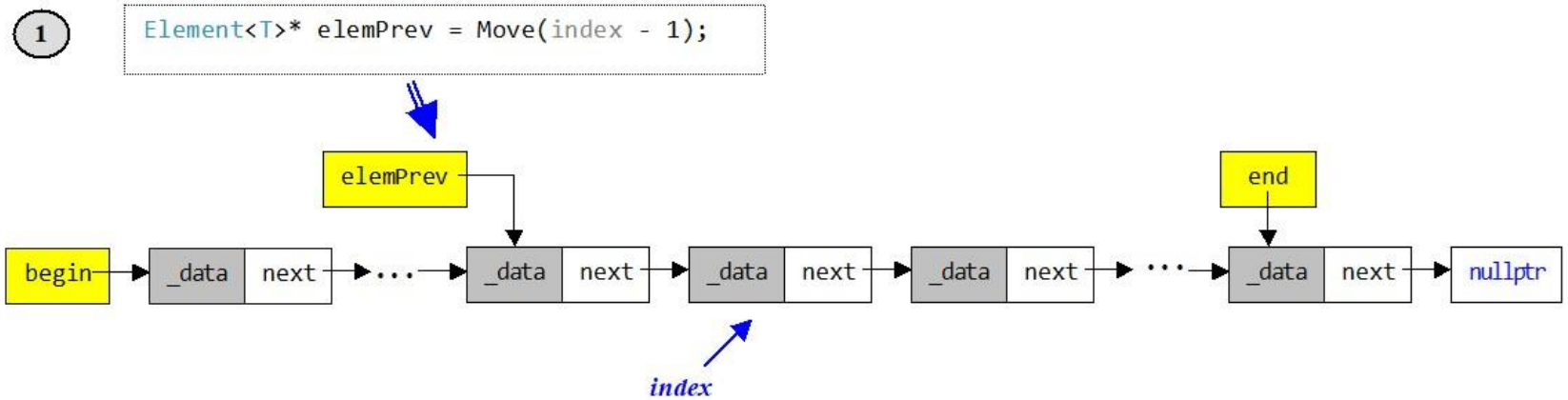


Видалення першого елементу зі списку

```
// Видалення першого елемента
void removeFirst() {
    if (!begin) {
        std::cerr << "Список порожній! Немає що видаляти.\n";
        return;
    }

    Element<T>* delElem = begin;
    begin = begin->next;
    delete delElem;
    --size;

    if (!begin) { // Якщо список став порожнім
        end = nullptr;
    }
}
```

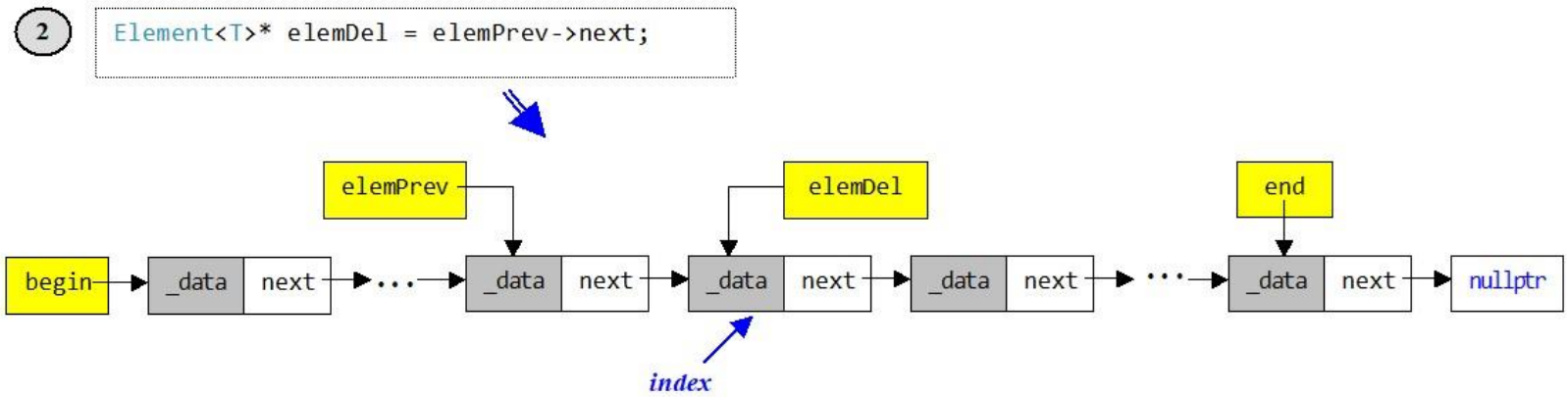
Видалення не першого елементу зі списку

- **Перемістити вказівник на позицію, що слідує перед елементом, який треба видалити. Отримати елемент, що передує елементу що видаляється**
- Запам'ятати елемент який видаляється
- Змістити вказівник next попереднього елементу elemPrev в обхід видаляємого елементу elemDel
- Видалити елемент elemDel (звільнити пам'ять, що була виділена для елементу).



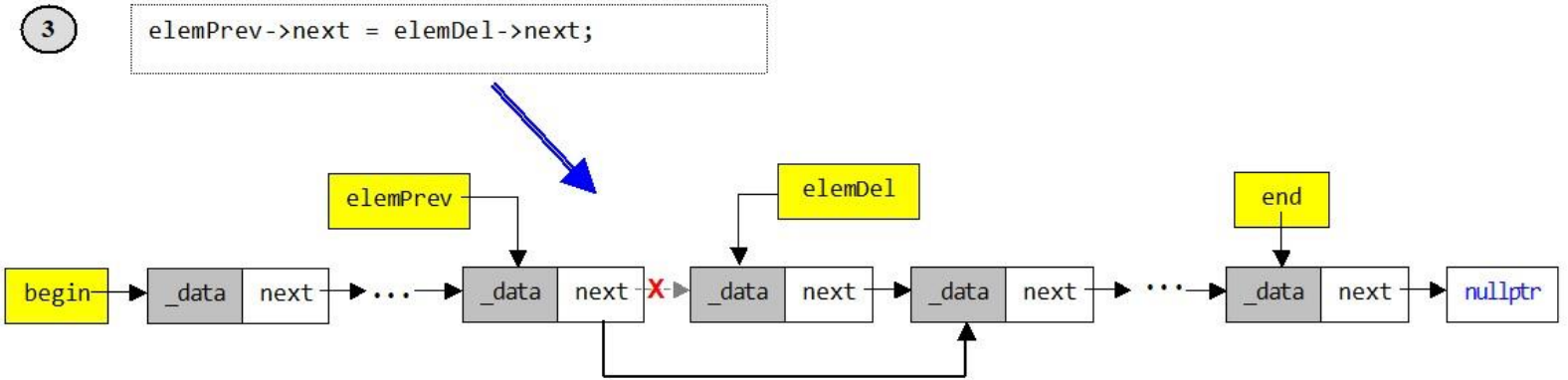
Видалення не першого елементу зі списку

- ❏ Перемістити вказівник на позицію, що слідує перед елементом, який треба видалити. Отримати елемент, що передує елементу що видаляється
- ❏ **Запам'ятати елемент який видаляється**
- ❏ Змістити вказівник next попереднього елементу elemPrev в обхід видаляємого елементу elemDel
- ❏ Видалити елемент elemDel (звільнити пам'ять, що була виділена для елементу).



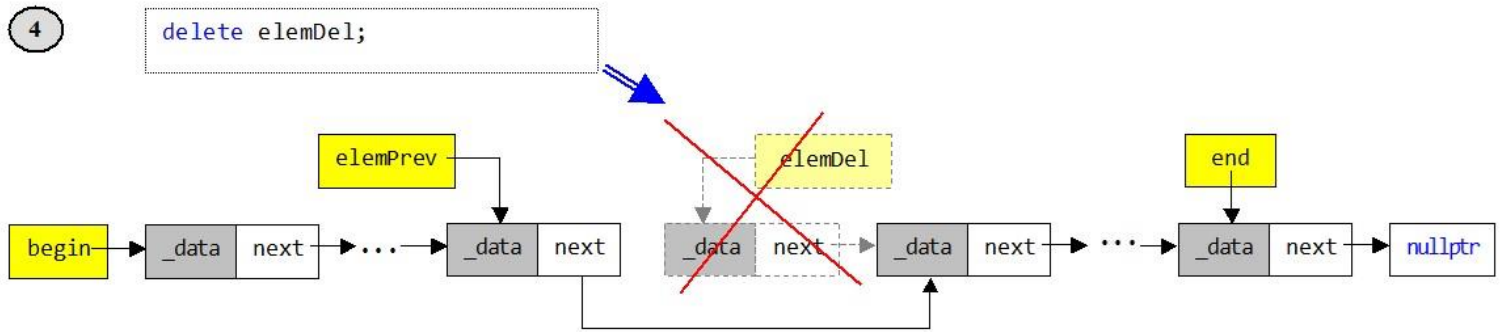
Видалення не першого елементу зі списку

- ❏ Перемістити вказівник на позицію, що слідує перед елементом, який треба видалити. Отримати елемент, що передує елементу що видаляється
- ❏ Запам'ятати елемент який видаляється
- ❏ **Змістити вказівник next попереднього елементу elemPrev в обхід видаляємого елементу elemDel**
- ❏ Видалити елемент elemDel (звільнити пам'ять, що була виділена для елементу).



Видалення не першого елементу зі списку

- ❏ Перемістити вказівник на позицію, що слідує перед елементом, який треба видалити. Отримати елемент, що передує елементу що видаляється
- ❏ Запам'ятати елемент який видаляється
- ❏ Змістити вказівник next попереднього елементу elemPrev в обхід видаляємого елементу elemDel
- ❏ **Видалити елемент elemDel (звільнити пам'ять, що була виділена для елементу).**



Заміна існуючого елементу в позиції (replace)

Метод replace() замінює значення вузла на заданій позиції у однозв'язному списку. Список обходиться до вказаної позиції за допомогою циклу for, переміщуючи вказівник current до потрібного вузла. Якщо позиція виходить за межі списку (current == nullptr або current->next == nullptr), метод завершується. Нарешті, дані вузла (current->data) замінюються на нове значення (value).

```
// Заміна елемента на заданій позиції
void replaceAt(size_t index, T newData) {
    Element<T>* elem = moveTo(index);
    if (!elem) {
        std::cerr << "Індекс виходить за межі списку!\n";
        return;
    }
    elem->data = newData;
}
```

Пошук існуючого елементу (find)

Метод find() повертає індекс першого входження заданого елемента у списку. Вказівник current починає з голови списку (begin), а змінна index відстежує позицію. У циклі while перевіряється кожен вузол: якщо дані вузла (current->data) збігаються із шуканим значенням (value), повертається поточний індекс. Якщо збігу немає, current переміщується до наступного вузла (current->next), а index збільшується. Якщо елемент не знайдено, повертається -1.

```
// Пошук елемента та повернення його індексу
int find(T value) {
    Element<T>* current = begin;
    size_t index = 0;

    while (current) {
        if (current->data == value) {
            return index;  // Повертаємо індекс першого входження
        }
        current = current->next;
        ++index;
    }
    return -1; // Якщо елемент не знайдено
}
```

Конструктор копіювання

- ❏ Викликається при передачі списку у функцію за значенням або при явному копіювання.
- ❏ Не викликається при передачі списку у функцію за посиланням або при використанні конструктора переміщення.

```
// Конструктор копіювання
List(const List& other) : begin(nullptr), end(nullptr), count(0) {
    cout << "List copy constructor\n";
    Element<T>* temp = other.begin;
    while (temp) {
        addToEnd(temp->data);
        temp = temp->next;
    }
}
```

```
// Функція, яка приймає список за значенням (конструктор копіювання)
template <class T>
void copyList(List<T> lst) {
    std::cout << "copyList function is invoked\n";
    std::cout << lst << std::endl;
}
```

```
// Тест конструктора копіювання
List<double> copiedList(lst1); // Явне копіювання
std::cout << "Copied list:\n" << copiedList << std::endl;
```

Конструктор переміщення (move constructor)

- **Мув семантика (переміщення) в C++** дозволяє ефективно передавати ресурси між об'єктами без їх копіювання. Це особливо корисно при роботі з великими об'єктами або контейнерами, де копіювання може бути дорогим.
- **Конструктор переміщення** “**T(T&& other)**” та **оператор присвоєння переміщення** “**T& operator=(T&& other)**” використовуються для "переміщення" ресурсів з одного об'єкта в інший, замість копіювання.
- **&& (rvalue reference)** - це вираз, який не має ідентифікатора і є тимчасовим. Він зазвичай використовується з правого боку операцій, наприклад, при ініціалізації змінних або присвоєнні значень. Дозволяє працювати з тимчасовими об'єктами і ефективно їх переміщати без копіювання.
- **`std::move()`** використовується для перетворення **lvalue** в **rvalue**, дозволяючи передати об'єкти в конструктор переміщення.

```
// Конструктор переміщення
List(List&& other) noexcept : begin(other.begin), end(other.end), count(other.count) {
    cout << "List move constructor\n";
    other.begin = other.end = nullptr;
    other.count = 0;
}

 // Тест конструктора переміщення
 List<int> movedList(std::move(lst2));
 std::cout << "Moved list:\n" << movedList << std::endl;
```

NRVO (Named Return Value Optimization)

RVO — це оптимізація, яка дозволяє компілятору уникнути копіювання або переміщення об'єкта, коли об'єкт створюється безпосередньо в місці повернення з функції.

Як це працює:
Коли ви повертаєте об'єкт з функції, компілятор може створити тимчасовий об'єкт, а потім передати його в місце виклику (це може бути виконано через копіювання або переміщення). Але з допомогою **RVO**, компілятор може створити об'єкт прямо в місці виклику функції, тим самим уникнувши копіювання.

NRVO — це оптимізація, яка подібна до **RVO**, але вона застосовується для об'єктів, які мають ім'я, тобто об'єкти, що створюються в тілі функції і повертаються за допомогою імені (не тимчасові об'єкти).

```
// Функція, яка повертає список (NRVO)
List<int> createList() {
    List<int> lst;
    lst.addToEnd(1);
    lst.addToEnd(2);
    lst.addToEnd(3);
    std::cout << "Function createList is invoked\n";
    return lst;
}

List<int> lst2 = createList(); // Викликає конструктор переміщення або NRVO в залежності від опцій компілятора
```

Двозв'язний (двонаправлений) список.

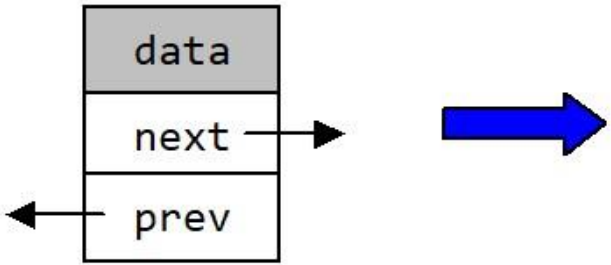
FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Двозв’язний список

Двозв'язний список — це динамічна структура даних, що складається з вузлів (елементів), кожен з яких містить три частини:

- Значення (дані) — інформація, що зберігається у вузлі.
- Посилання на наступний вузол — вказівник на наступний елемент списку.
- Посилання на попередній вузол — вказівник на попередній елемент списку.

Така структура дозволяє ефективно переміщуватися як вперед, так і назад по списку, спрощуючи операції вставки та видалення елементів.

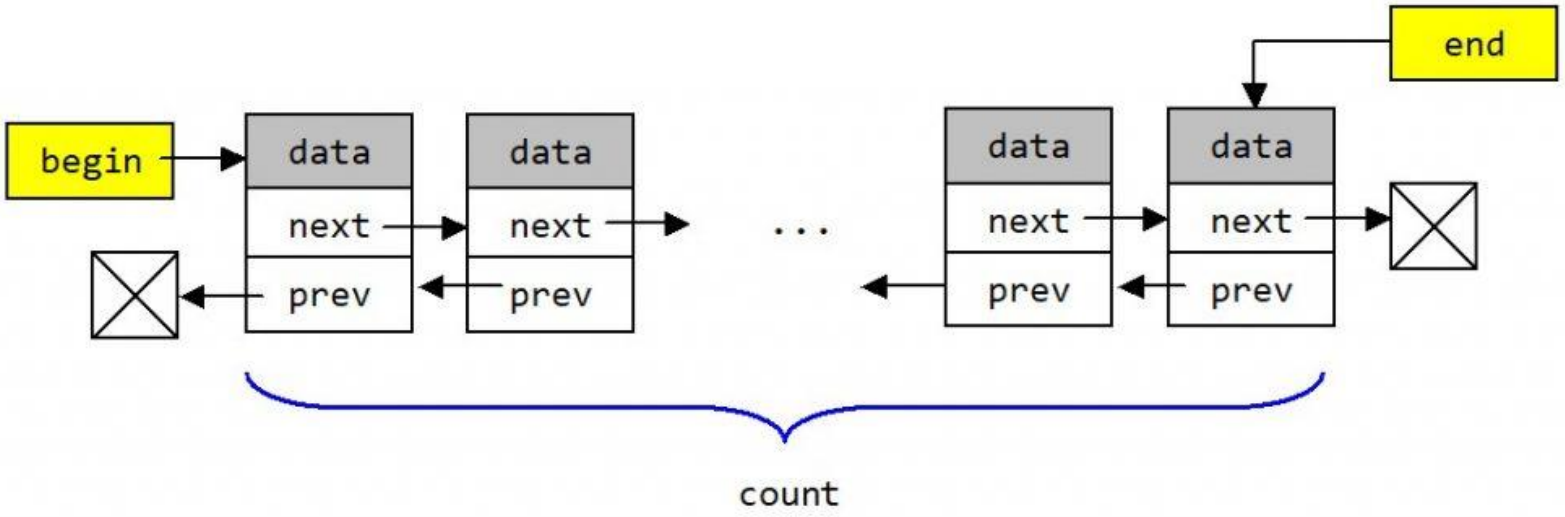



```
template <class T>
struct Element
{
    T data;
    Element<T>* next;
    Element<T>* prev;
};
```

Загальний вигляд двозв'язного списку

```
template <class T>
struct Element
{
    T data;
    Element* next;
    Element* prev;

    Element(T _data = T(), Element* _next = nullptr, Element* _prev = nullptr);
};
```

```
template <class T>
class DoublyLinkedList
{
private:
    Element<T>* begin;
    Element<T>* end;
    size_t count;

public:
    DoublyLinkedList();
    ~DoublyLinkedList();
```



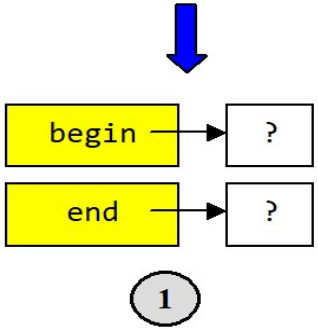
Створення списку

Перед додаванням нових елементів у список першочерговою задачею є створення порожнього списку.
При створенні списку можна виділити такі етапи:

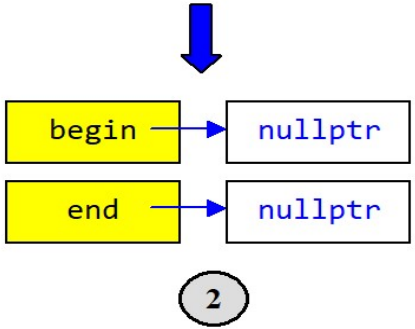
- **оголошення вказівників на початок та кінець списку (begin, end);**
- **встановлення нульових значень вказівника;**
- **встановлення довжини списку count у нульове значення.**

```
template <class T>
DoublyLinkedList<T>::DoublyLinkedList() : begin(nullptr), end(nullptr), count(0) {}
```

```
Element<T>* begin;
Element<T>* end;
int count;
```




```
begin = nullptr;
end = nullptr;
count = 0;
```

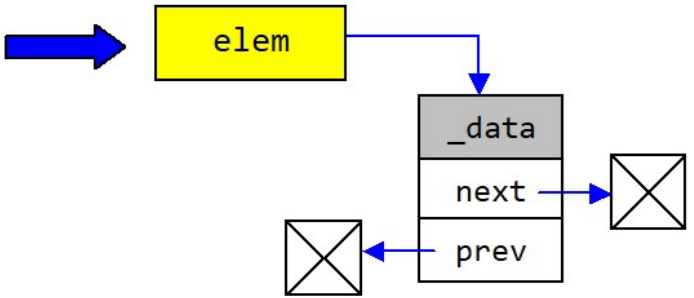


Додавання нового елементу в кінець порожнього

- ❏ **створити новий елемент elem та заповнити його даними**
- ❏ встановити початок та кінець списку (begin, end) так, щоб вони показували на новостворений елемент elem
- ❏ збільшити загальну кількість елементів у списку (поле count).

1

```
Element<T>* elem = new Element<T>;
elem->next = nullptr;
elem->prev = nullptr;
elem->data = _data;
```

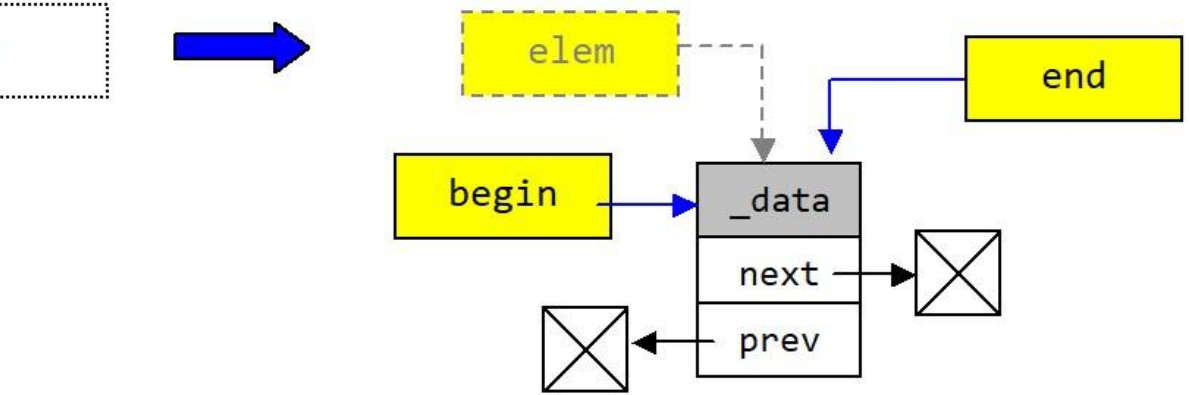


Додавання нового елементу в кінець порожнього

- створити новий елемент elem та заповнити його даними
- **встановити початок та кінець списку (begin, end) так, щоб вони показували на новостворений елемент elem**
- збільшити загальну кількість елементів у списку (поле count).

2

```
begin = end = elem;
```

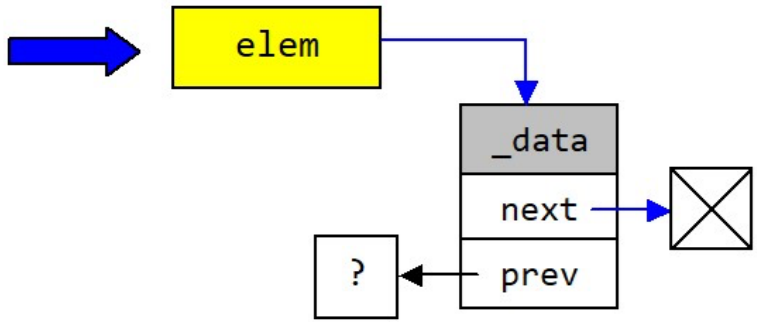


Додавання нового елементу до списку в якому вже є елементи

- **створення нового елементу elem. Заповнення полів data та next елементу**
- налаштування вказівник prev елементу elem таким чином, що він вказує на поточну позицію вказівника кінця списку end
- налаштування вказівника next останнього елементу end таким чином, щоб він вказував на елемент elem
- зміщення вказівника кінця списку elem на одну позицію, яка визначена вказівником elem.

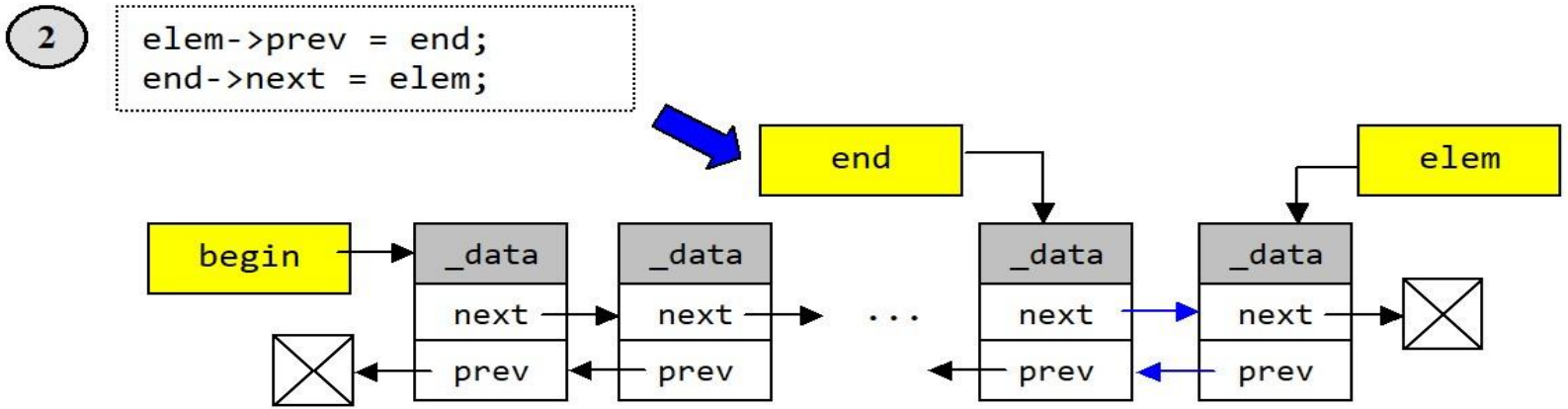
1

```
Element<T>* elem = new Element<T>;
elem->next = nullptr;
elem->data = _data;
```



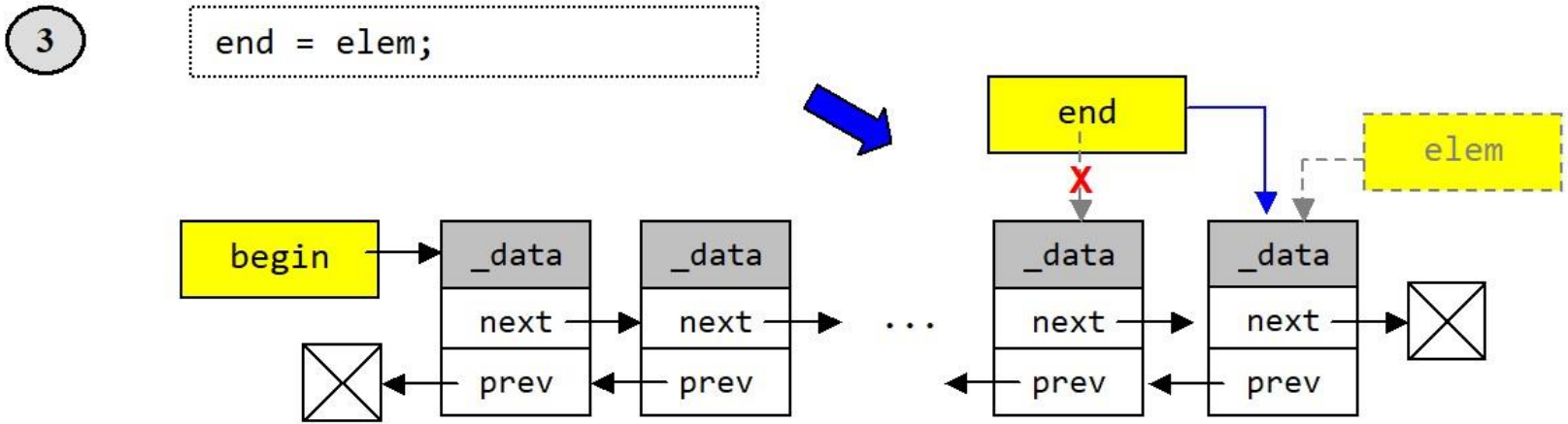
Додавання нового елементу до списку в якому вже є елементи

- створення нового елементу elem. Заповнення полів data та next елементу
- **налаштування вказівник prev елементу elem таким чином, що він вказує на поточну позицію вказівника кінця списку end**
- **налаштування вказівника next останнього елементу end таким чином, щоб він вказував на елемент elem**
- зміщення вказівника кінця списку elem на одну позицію, яка визначена вказівником elem.



Додавання нового елементу до списку в якому вже є елементи

- створення нового елементу elem. Заповнення полів data та next елементу
- налаштування вказівник prev елементу elem таким чином, що він вказує на поточну позицію вказівника кінця списку end
- налаштування вказівника next останнього елементу end таким чином, щоб він вказував на елемент elem
- **зміщення вказівника кінця списку elem на одну позицію, яка визначена вказівником elem.**



Додавання нового елементу в кінець списку

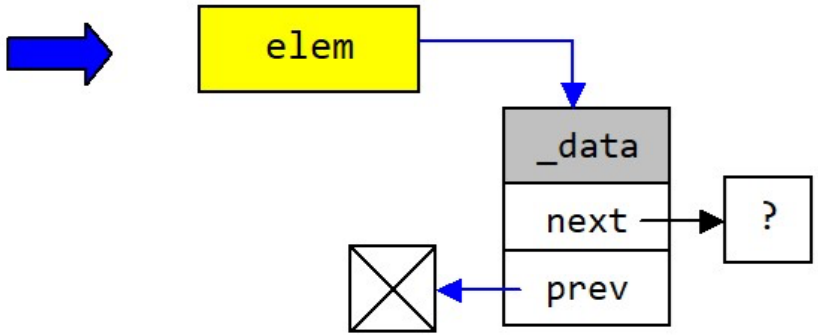
```
template <class T>
void DoublyLinkedList<T>::addToEnd(T _data)
{
    Element<T>* newNode = new Element<T>(_data);
    if (!begin)
    {
        begin = end = newNode;
    }
    else
    {
        end->next = newNode;
        newNode->prev = end;
        end = newNode;
    }
    ++count;
}
```

Вставка елементу в першу позицію списку

- **створити елемент та заповнити його даними. Заповнюються поля data та prev**
- встановити вказівник next елементу elem на перший елемент в списку begin
- встановити вказівник prev першого елементу списку на елемент elem, що вставляється
- перенаправити вказівник begin так, щоб він вказував на новостворюваний елемент elem
- збільшити загальну кількість елементів у списку

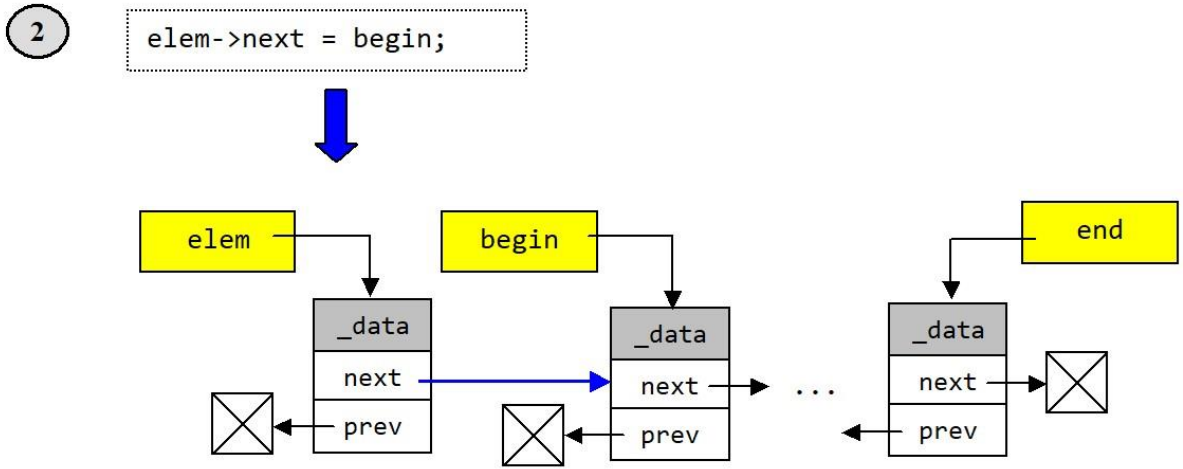
1

```
Element<T>* elem = new Element<T>;
elem->data = _data;
elem->prev = nullptr;
```



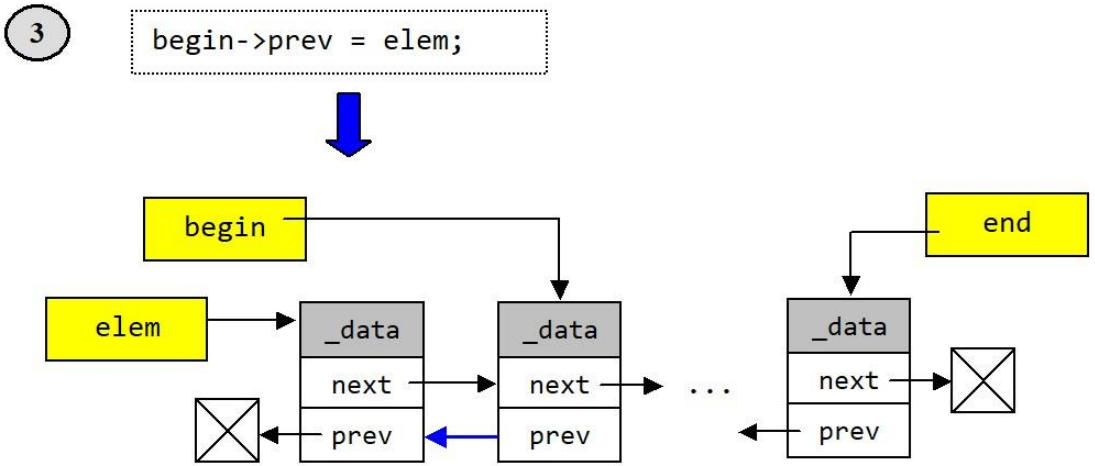
Вставка елементу в першу позицію списку

- створити елемент та заповнити його даними. Заповнюються поля data та prev
- **встановити вказівник next елементу elem на перший елемент в списку begin**
- встановити вказівник prev першого елементу списку на елемент elem, що вставляється
- перенаправити вказівник begin так, щоб він вказував на новостворюваний елемент elem
- збільшити загальну кількість елементів у списку



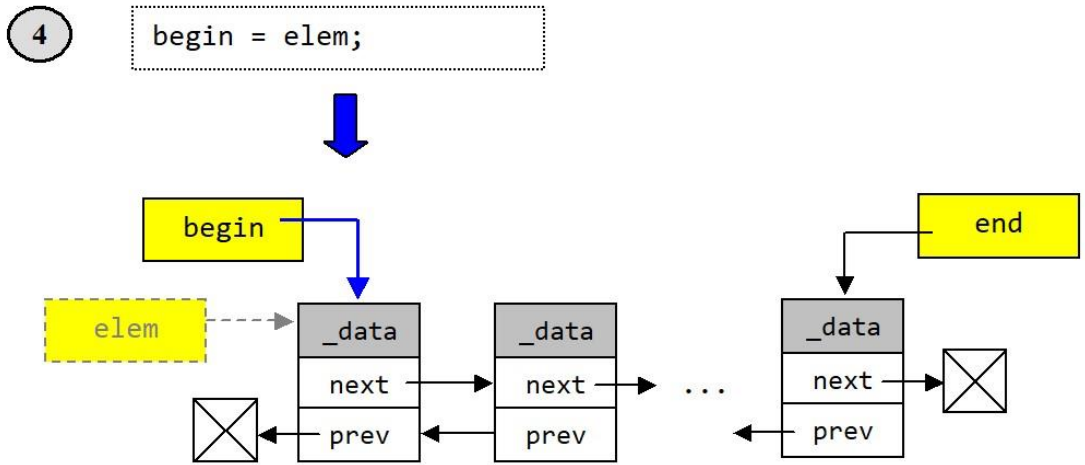
Вставка елементу в першу позицію списку

- створити елемент та заповнити його даними. Заповнюються поля data та prev
- встановити вказівник next елементу elem на перший елемент в списку begin
- **встановити вказівник prev першого елементу списку на елемент elem, що вставляється**
- перенаправити вказівник begin так, щоб він вказував на новостворюваний елемент elem
- збільшити загальну кількість елементів у списку



Вставка елементу в першу позицію списку

- ❏ створити елемент та заповнити його даними. Заповнюються поля data та prev
- ❏ встановити вказівник next елементу elem на перший елемент в списку begin
- ❏ встановити вказівник prev першого елементу списку на елемент elem, що вставляється
- ❏ **перенаправити вказівник begin так, щоб він вказував на новостворюваний елемент elem**
- ❏ збільшити загальну кількість елементів у списку



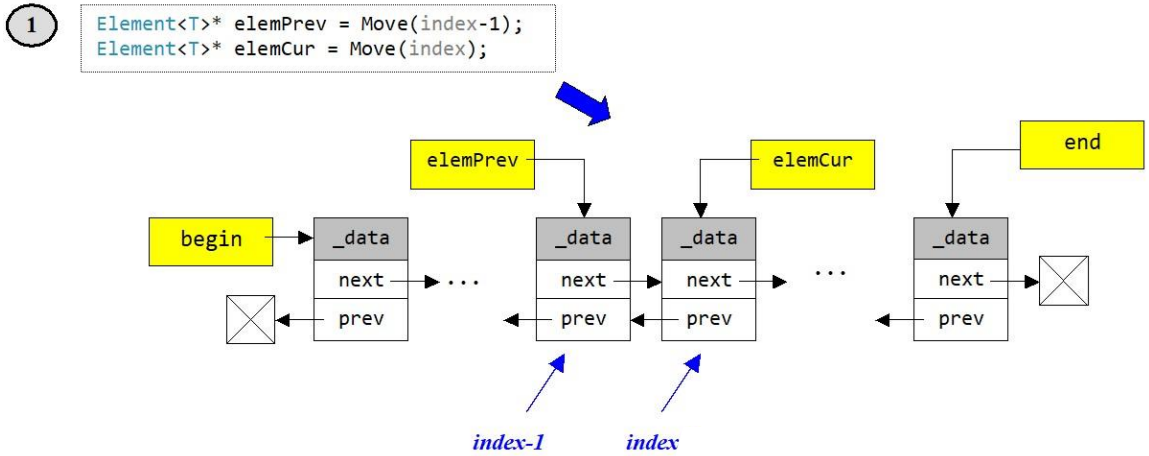
Вставка елементу в першу позицію списку

```
template <class T>
void DoublyLinkedList<T>::addToBegin(T _data)
{
    Element<T>* newNode = new Element<T>(_data, begin);
    if (!begin)
    {
        begin = end = newNode;
    }
    else
    {
        begin->prev = newNode;
        begin = newNode;
    }
    ++count;
}
```

Вставка елементу в середину списку

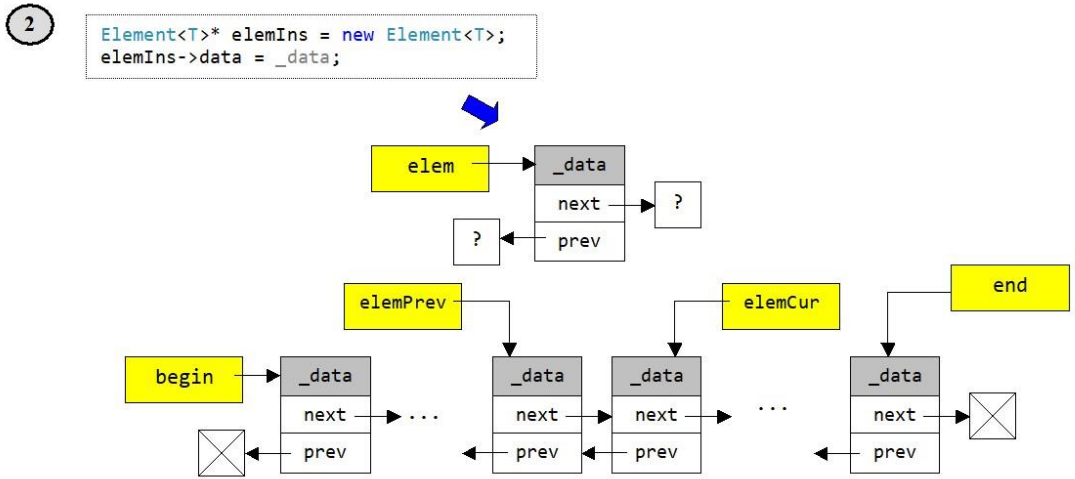
- ❑ **отримати елемент elemPrev який слідує перед позицією вставки**
- ❑ **отримати елемент elemCur, який розміщується в позиції вставки index**
- ❑ створити новий елемент elemIns, який буде вставлятись між елементами elemPrev та elemCur. Заповнити поле data елементу
- ❑ встановити вказівник next елементу elemIns рівним адресі елементу elemCur
- ❑ встановити вказівник prev елементу elemIns рівним адресі елементу elemPrev
- ❑ змістити вказівник next елементу elemPrev на елемент elemIns
- ❑ змістити вказівник prev елементу elemCur на елемент elemIns

```
// Отримати елемент в заданій позиції
template <class T>
Element<T>* List<T>::Move(int index)
{
    // 1. Встановити вказівник на початок списку
    Element<T>* t = begin;
    // 2. Перемотати до позиції index
    for (int i = 0; i < index; i++)
        t = t->next;
    // 3. Повернути вказівник
    return t;
}
```



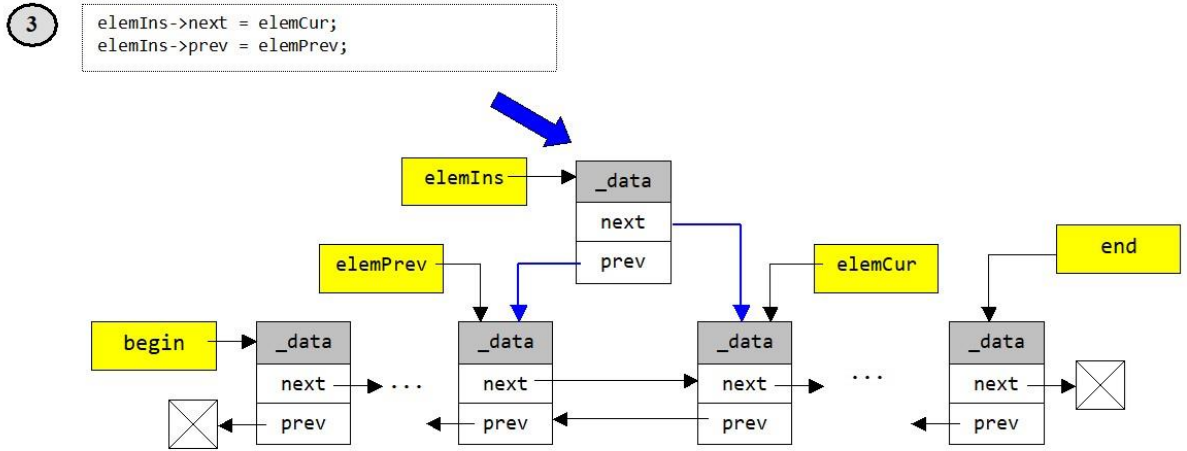
Вставка елементу в середину списку

- ❑ отримати елемент elemPrev який слідує перед позицією вставки
- ❑ отримати елемент elemCur, який розміщується в позиції вставки index
- ❑ **створити новий елемент elemIns, який буде вставлятись між елементами elemPrev та elemCur. Заповнити поле data елементу**
- ❑ встановити вказівник next елементу elemIns рівним адресі елементу elemCur
- ❑ встановити вказівник prev елементу elemIns рівним адресі елементу elemPrev
- ❑ змістити вказівник next елементу elemPrev на елемент elemIns
- ❑ змістити вказівник prev елементу elemCur на елемент elemIns



Вставка елементу в середину списку

- отримати елемент elemPrev який слідує перед позицією вставки
- отримати елемент elemCur, який розміщується в позиції вставки index
- створити новий елемент elemIns, який буде вставлятись між елементами elemPrev та elemCur. Заповнити поле data елементу
- **встановити вказівник next елементу elemIns рівним адресі елементу elemCur**
- **встановити вказівник prev елементу elemIns рівним адресі елементу elemPrev**
- змістити вказівник next елементу elemPrev на елемент elemIns
- змістити вказівник prev елементу elemCur на елемент elemIns

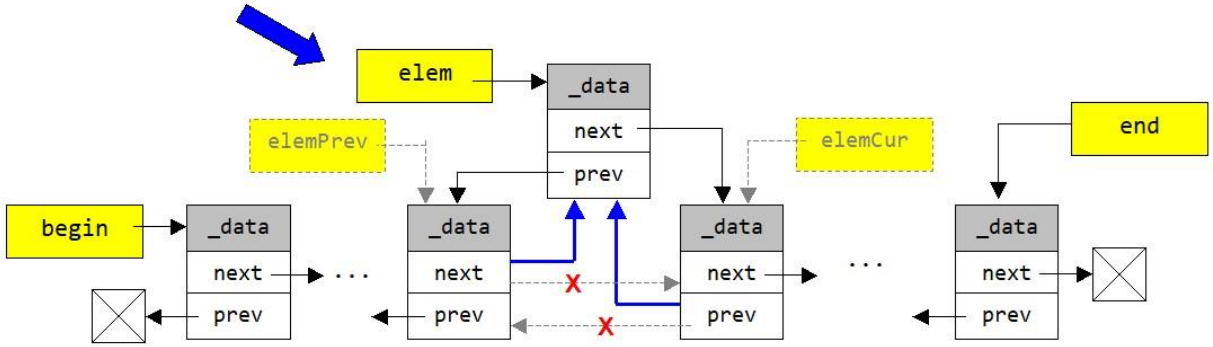


Вставка елементу в середину списку

- ❏ отримати елемент elemPrev який слідує перед позицією вставки
- ❏ отримати елемент elemCur, який розміщується в позиції вставки index
- ❏ створити новий елемент elemIns, який буде вставлятись між елементами elemPrev та elemCur. Заповнити поле data елементу
- ❏ встановити вказівник next елементу elemIns рівним адресі елементу elemCur
- ❏ встановити вказівник prev елементу elemIns рівним адресі елементу elemPrev
- ❏ **змістити вказівник next елементу elemPrev на елемент elemIns**
- ❏ **змістити вказівник prev елементу elemCur на елемент elemIns**

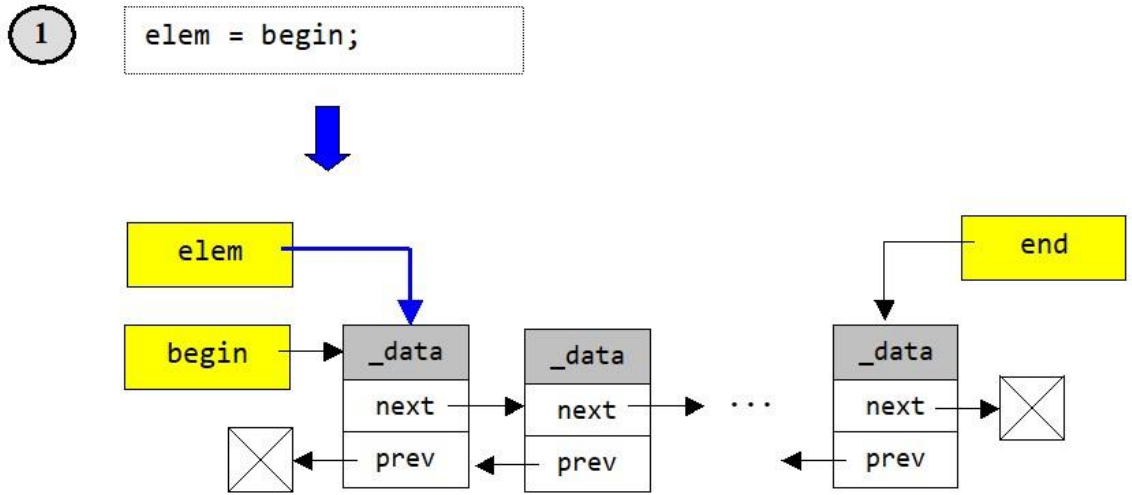
4

```
elemPrev->next = elem;
elemCur->prev = elem;
```



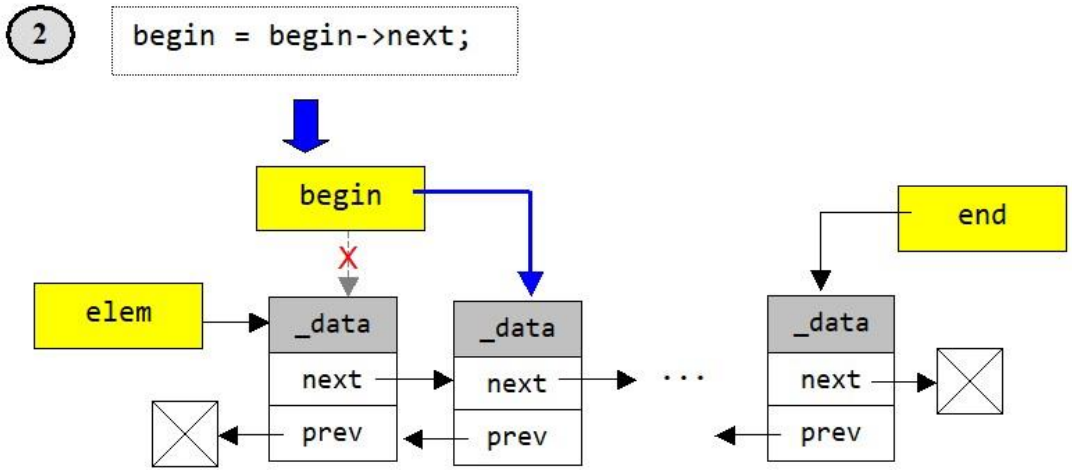
Видалення першого елементу зі списку

- **запам'ятати адресу першого елементу в значення elem**
- змінити значення вказівника begin так, щоб він вказував на наступний елемент. Якщо наступного елементу немає, то вказівник матиме значення nullptr;
- встановити значення вказівника prev у нульове значення
- звільнити пам'ять, виділену для елементу elem;
- зменшити загальну кількість елементів на 1.



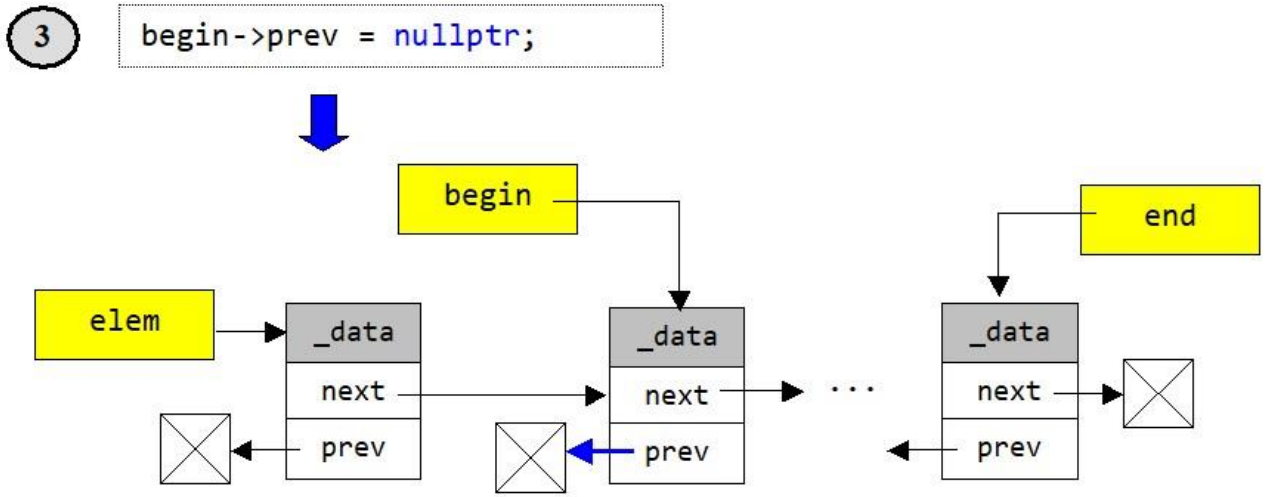
Видалення першого елементу зі списку

- ❏ запам'ятати адресу першого елементу в значення elem
- ❏ **змінити значення вказівника begin так, щоб він вказував на наступний елемент. Якщо наступного елементу немає, то вказівник матиме значення nullptr;**
- ❏ встановити значення вказівника prev у нульове значення
- ❏ звільнити пам'ять, виділену для елементу elem;
- ❏ зменшити загальну кількість елементів на 1.



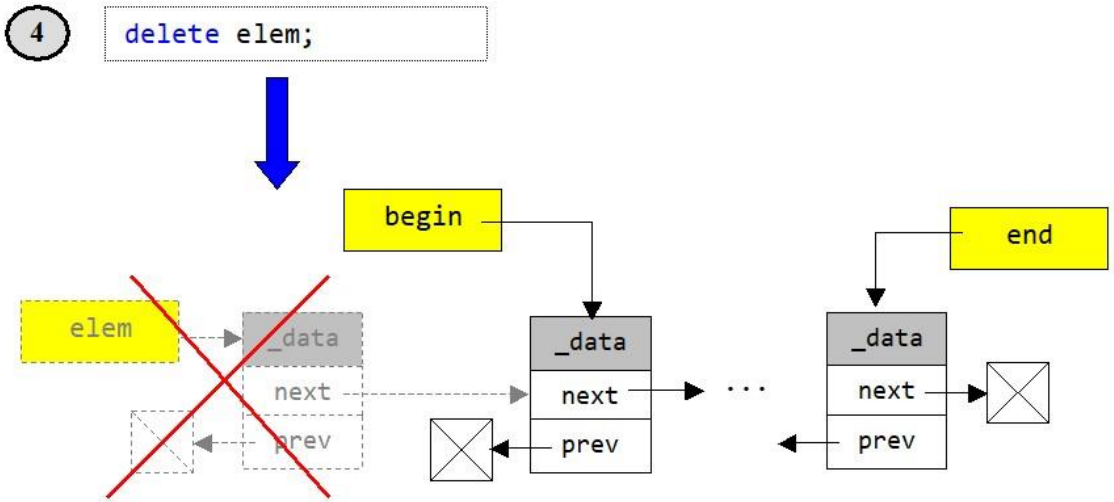
Видалення першого елементу зі списку

- ❏ запам’ятати адресу першого елементу в значення elem
- ❏ змінити значення вказівника begin так, щоб він вказував на наступний елемент. Якщо наступного елементу немає, то вказівник матиме значення nullptr;
- ❏ **встановити значення вказівника prev у нульове значення**
- ❏ звільнити пам’ять, виділену для елементу elem;
- ❏ зменшити загальну кількість елементів на 1.



Видалення першого елементу зі списку

- ❏ запам’ятати адресу першого елементу в значення elem
- ❏ змінити значення вказівника begin так, щоб він вказував на наступний елемент. Якщо наступного елементу немає, то вказівник матиме значення nullptr;
- ❏ встановити значення вказівника prev у нульове значення
- ❏ **звільнити пам’ять, виділену для елементу elem;**
- ❏ зменшити загальну кількість елементів на 1.



Видалення першого елементу зі списку

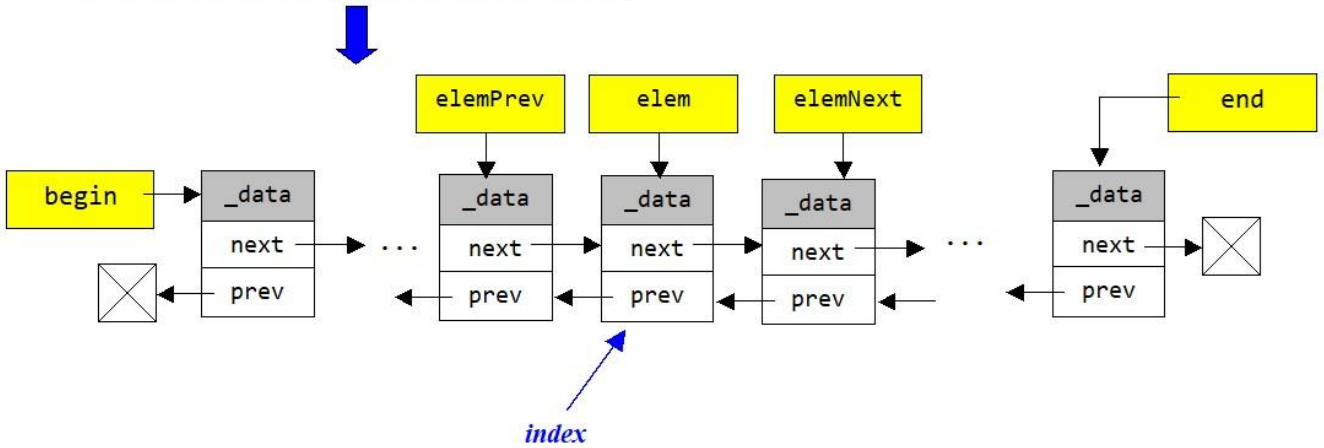
```
template <class T>
void DoublyLinkedList<T>::removeFirst()
{
    if (!begin)
        return;

    Element<T>* temp = begin;
    begin = begin->next;

    if (begin)
        begin->prev = nullptr;
    else
        end = nullptr;

    delete temp;
    --count;
}
```

Видалення з середини списку

- **на основі індексу елементу index отримати вказівник на елемент elem, що видаляється, а також на попередній elemPrev та наступний elemNext елементи**
- встановити вказівник next елементу elemPrev рівним адресі елементу elemNext
- встановити вказівник prev елементу elemNext рівним адресі елементу elemPrev
- знищити (звільнити пам'ять) елемент elem

1

```
Element<T>* elem = Move(index);
Element<T>* elemPrev = elem->prev;
Element<T>* elemNext = elem->next;
```

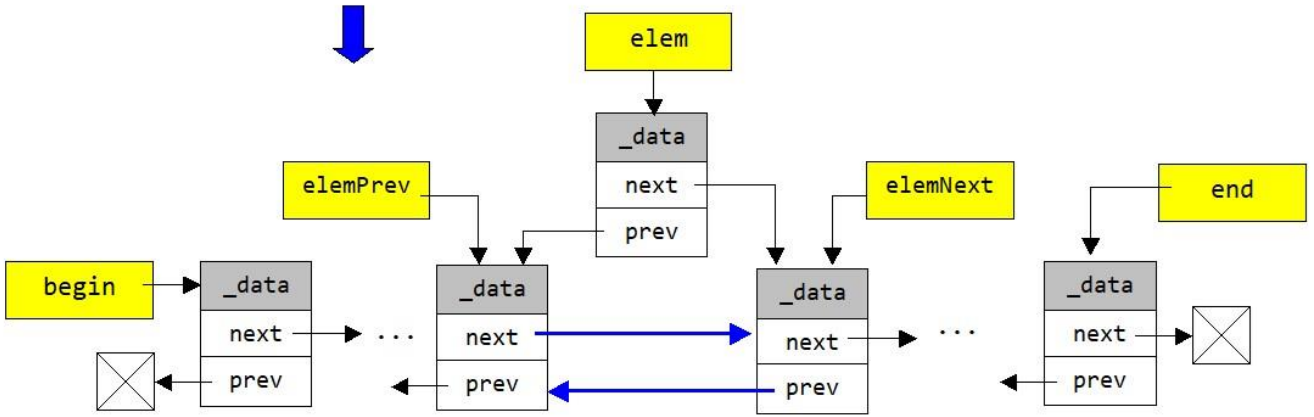


Видалення з середини списку

- ❏ на основі індексу елементу index отримати вказівник на елемент elem, що видаляється, а також на попередній elemPrev та наступний elemNext елементи
- ❏ **встановити вказівник next елементу elemPrev рівним адресі елементу elemNext**
- ❏ **встановити вказівник prev елементу elemNext рівним адресі елементу elemPrev**
- ❏ знищити (звільнити пам'ять) елемент elem

2

```
elemPrev->next = elemNext;
elemNext->prev = elemPrev;
```

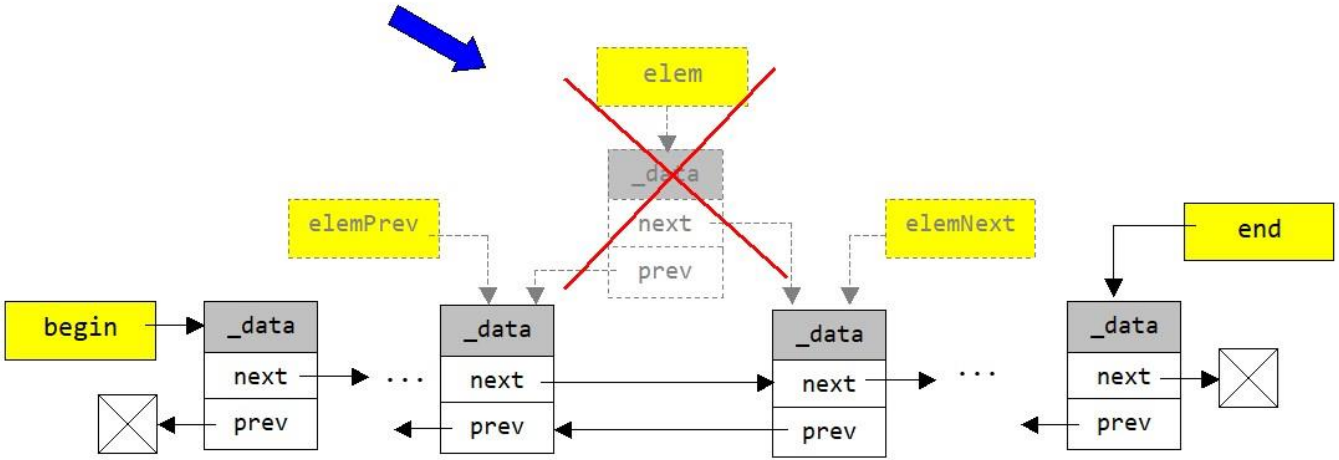


Видалення з середини списку

- ❏ на основі індексу елементу index отримати вказівник на елемент elem, що видаляється, а також на попередній elemPrev та наступний elemNext елементи
- ❏ встановити вказівник next елементу elemPrev рівним адресі елементу elemNext
- ❏ встановити вказівник prev елементу elemNext рівним адресі елементу elemPrev
- ❏ **знищити (звільнити пам'ять) елемент elem**

3

```
delete elem;
```



Видалення з середини списку

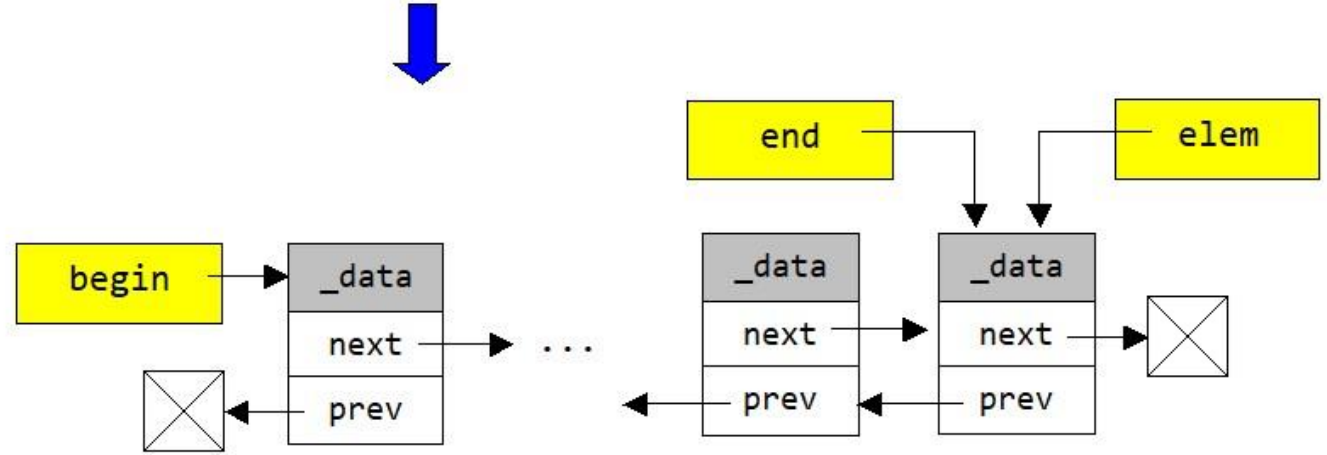
```
template <class T>
void DoublyLinkedList<T>::removeAt(size_t index)
{
    if (index >= count) {
        std::cerr << "Index out of range !\n";
        return;
    }

    if (index == 0)
    {
        removeFirst();
        return;
    }

    if (index == count - 1)
    {
        removeLast();
        return;
    }

    Element<T>* current = begin;
    for (size_t i = 0; i < index; ++i)
    {
        current = current->next;
    }

    current->prev->next = current->next;
    current->next->prev = current->prev;
    delete current;
    --count;
}
```

Видалення останнього елементу зі списку

- **запам'ятати адресу останнього елементу в списку у вказівнику elem**
- перемістити на попередній елемент вказівник end, який вказував на останній елемент у списку
- звільнити пам'ять, що була виділена під останній елемент у списку
- встановити вказівник next елементу end в нульове значення

1

```
Element<T>* elem = end;
```

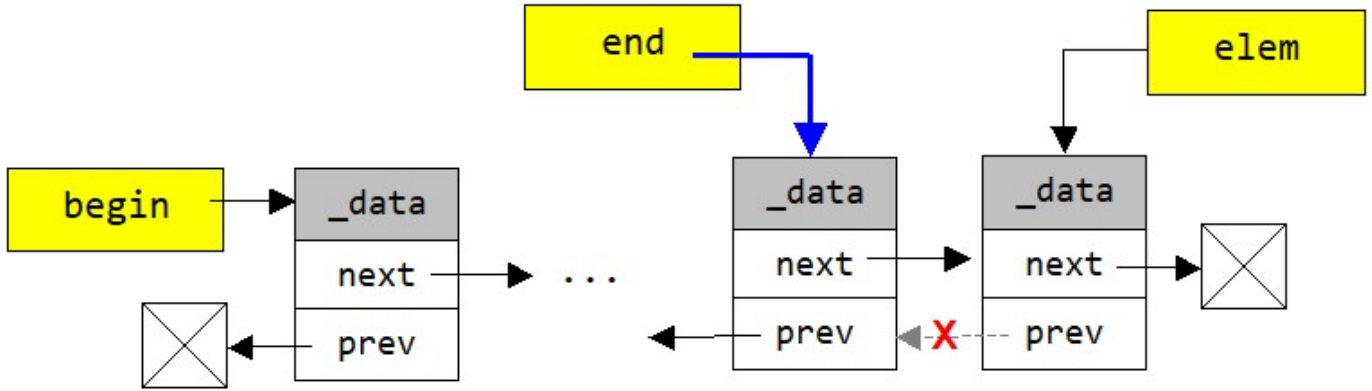


Видалення останнього елементу зі списку

- ❏ запам’ятати адресу останнього елементу в списку у вказівнику elem
- ❏ **перемістити на попередній елемент вказівник end, який вказував на останній елемент у списку**
- ❏ звільнити пам’ять, що була виділена під останній елемент у списку
- ❏ встановити вказівник next елементу end в нульове значення

2

```
end = end->prev;
```

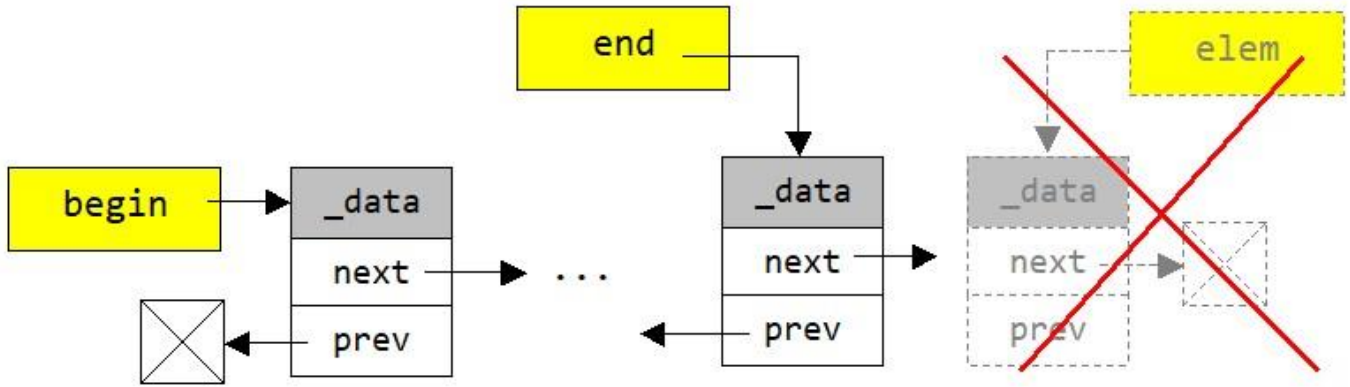


Видалення останнього елементу зі списку

- ❏ запам’ятати адресу останнього елементу в списку у вказівнику elem
- ❏ перемістити на попередній елемент вказівник end, який вказував на останній елемент у списку
- ❏ **звільнити пам’ять, що була виділена під останній елемент у списку**
- ❏ встановити вказівник next елементу end в нульове значення

3

```
delete elem;
```

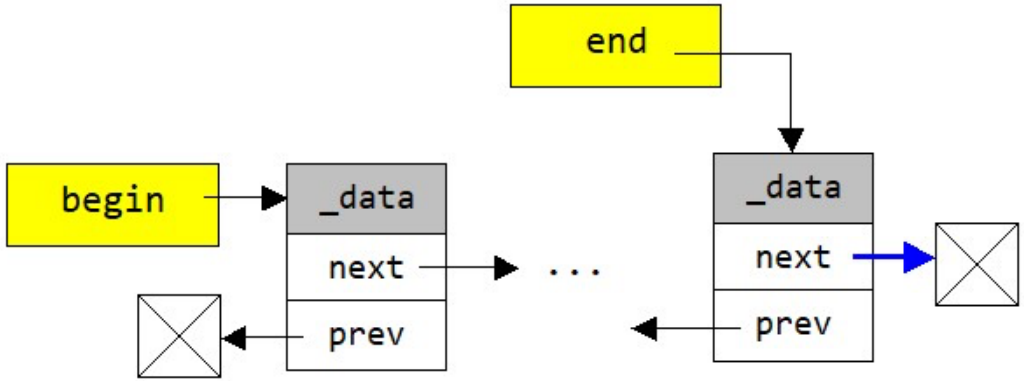


Видалення останнього елементу зі списку

- ❏ запам’ятати адресу останнього елементу в списку у вказівнику elem
- ❏ перемістити на попередній елемент вказівник end, який вказував на останній елемент у списку
- ❏ звільнити пам’ять, що була виділена під останній елемент у списку
- ❏ **встановити вказівник next елементу end в нульове значення**

4

```
elem->next = nullptr;
```



Видалення останнього елементу зі списку

```
template <class T>
void DoublyLinkedList<T>::removeLast()
{
    if (!end)
        return;

    Element<T>* temp = end;
    end = end->prev;

    if (end)
        end->next = nullptr;
    else
        begin = nullptr;

    delete temp;
    --count;
}
```

Сортування елементів (Bubble sort)

```cpp
template <class T>
void DoublyLinkedList<T>::sort()
{
    if (!begin || !begin->next) return; // Empty or single-element list is already sorted

    bool swapped;
    Element<T>* lastSorted = nullptr; // Last sorted element (to reduce comparisons)

    do
    {
        swapped = false;
        Element<T>* current = begin;

        while (current->next != lastSorted)
        {
            if (current->data > current->next->data)
            {
                std::swap(current->data, current->next->data);
                swapped = true;
            }
            current = current->next;
        }

        lastSorted = current; // Last element is now sorted
    } while (swapped);
}
```

Порівняння динамічних масивів та списків

Характеристика	Динамічний масив	Linked List
Доступ до елементів	O(1)	O(n)
Додавання/видалення в кінець	O(1) (середнє), O(n) (в найгіршому)	O(1) (якщо є вказівник на кінець)
Додавання/видалення в початок	O(n)	O(1)
Додавання/видалення в середині	O(n)	O(1) (якщо є вказівник на елемент)
Пам'ять	Одна область пам'яті, фіксований розмір	Розподілена пам'ять, додаткові вказівники
Підтримка змінного розміру	Потрібен перерозподіл пам'яті	Автоматично адаптується
Пам'ять на елемент	Мінімум	Додаткові витрати на зберігання вказівника
Швидкість доступу	Швидкий доступ за індексом (O(1))	Повільний доступ (O(n))
Складність реалізації	Проста	Складніша, потребує керування вказівниками

Дякую

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY