Лекція 20
Реалізація ітератора, стек, черга, бінарне пошукове дерево, геш-таблиці

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

План на сьогодні

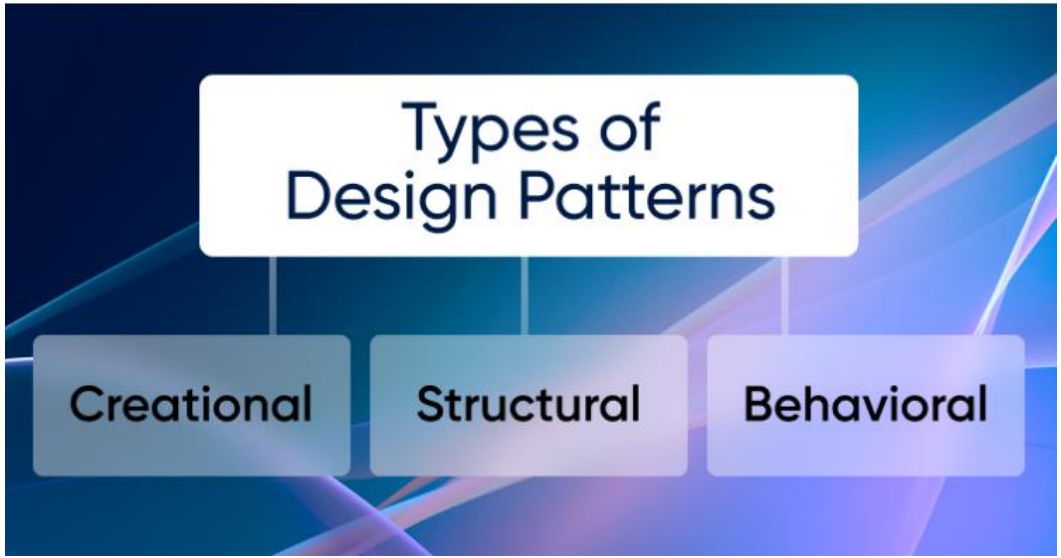
1. Реалізація ітератора
2. Стек
3. Черга
4. Бінарне пошукове дерево
5. Геш-таблиці

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Патерни проектування

Патерни (або шаблони) проектування описують типові способи вирішення поширених проблем при проектуванні програм (https://refactoring.guru/uk/design-patterns).

- ❏ Шаблони утворення об'єктів (Singleton, Builder, Abstract Factory, Factory Method, **Prototype**)
- ❏ Структурні шаблони (Adapter, Bridge, Composite, Decorator, Facade, Proxy...)
- ❏ Шаблони поведінки (Command, Interpreter, **Iterator**, Observer, State, **Strategy**, Visitor...)



Реалізація ітератора

Ітератори

- Ітератор —патерн поведінки об'єктів.

Призначення

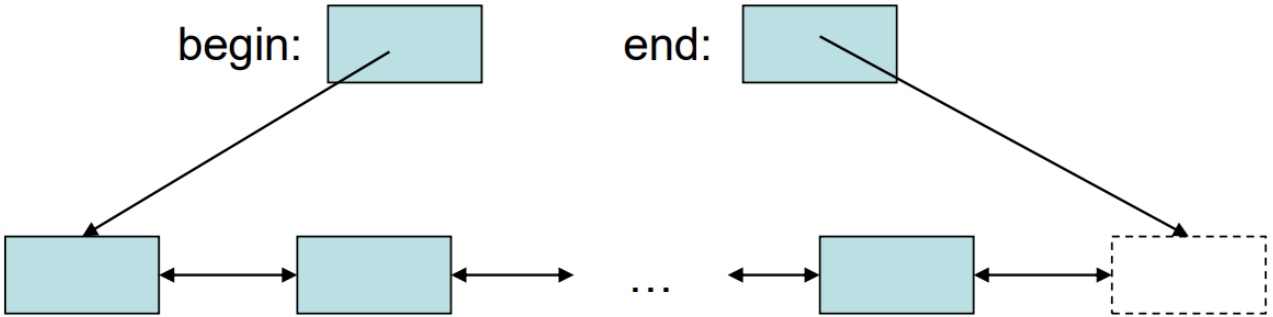
Надає спосіб послідовного доступу до всіх елементів складного об'єкта, напр., списку, не розкриваючи його внутрішньої будови

Мотивація

- за доступ до елементів і обхід відповідає не складний об'єкт, а окремий об'єкт-ітератор
- фіксований набір операцій
- можливість різних обходів
- довільна кількість об'єктів-ітераторів

Ітератори (begin, end)

- Пара ітераторів визначає послідовність (наприклад список)
- **begin (початок)** - вказівник на перший елемент
- **end (кінець)** - (вказівник на елемент, що слідує за останнім елементом послідовності)



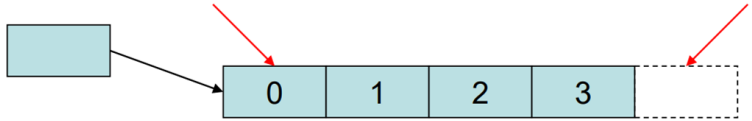
Операції, що підтримує Iterator

- Ітератор - це тип, що підтримує такі операції:
  - ++ перехід на наступний елемент
  - * отримати значення (розіменування)
  - = = порівняння (чи даний ітератор вказує на той самий елемент, що й інший ітератор?)
- Деякі ітератори мають й інші операції (--, +, [ ])

Ітератори

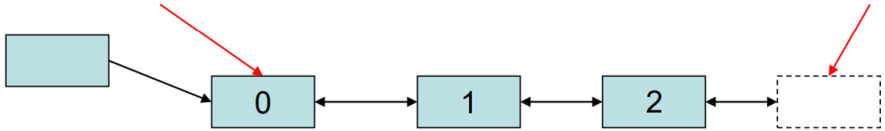
(обхід послідовностей різними способами)

- **vector**



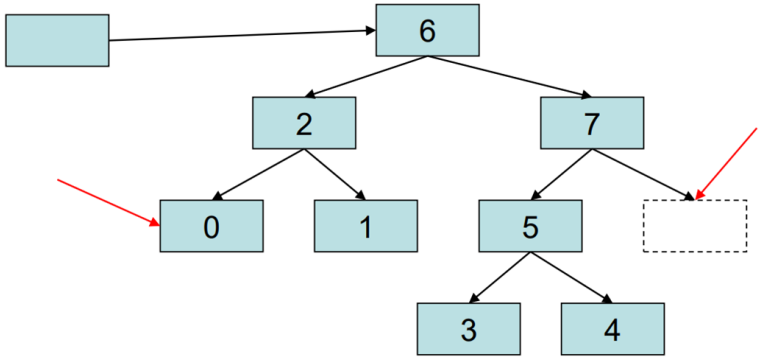
- **list**

(двозв`язний)



- **set**

(дерево)



Визначення типу iterator

- Слід мати на увазі
  - різним типам контейнерів притаманні різні способи обходу
  - **контейнер** та **ітератор** – окремі класи
  - реалізуємо різні види обходів одного і того ж контейнера як підкласи одного класу iterator

Приклад реалізації для класу List

```
template <typename T>
class List {
private:
    struct Node {
        T data;
        Node* next;
        Node* prev;
        Node(const T& data);
    };
public:
    class Iterator {
    private:
        Node* current;
    public:
        Iterator(Node* current);
        T& operator*() const;
        Iterator& operator++();
        Iterator operator++(int);
        Iterator& operator--();
        Iterator operator--(int);
        bool operator!=(const Iterator& other) const;
        bool operator==(const Iterator& other) const;
        Node* getPointer() const;
        friend class List;
    };
private:
    Node* head;
    Node* tail;
    size_t size;
public:
    List();
    ~List();
    void clear();
    size_t get_size() const;
    void push_front(const T& data);
    void push_back(const T& data);
    Iterator begin();
    Iterator end();
    Iterator rbegin();
    Iterator rend();
    Iterator find(const T& data);
    void erase(Iterator pos);
};
```

Реалізація методів ітератора

```cpp
/* ========== Реалізація класу Iterator ========== */
template <typename T>
List<T>::Iterator::Iterator(Node* current) : current(current) {}

template <typename T>
T& List<T>::Iterator::operator*() const { return current->data; }

template <typename T>
typename List<T>::Iterator& List<T>::Iterator::operator++() {
    if (current) current = current->next;
    return *this;
}
template <typename T>
typename List<T>::Iterator List<T>::Iterator::operator++(int) {
    Iterator temp = *this;
    ++(*this);
    return temp;
}
template <typename T>
typename List<T>::Iterator& List<T>::Iterator::operator--() {
    if (current) current = current->prev;
    return *this;
}
template <typename T>
typename List<T>::Iterator List<T>::Iterator::operator--(int) {
    Iterator temp = *this;
    --(*this);
    return temp;
}
```

Перевантажені оператори порівняння

```cpp
template <typename T>
bool List<T>::Iterator::operator!=(const Iterator& other) const { return current != other.current; }

template <typename T>
bool List<T>::Iterator::operator==(const Iterator& other) const { return current == other.current; }

template <typename T>
typename List<T>::Node* List<T>::Iterator::getPointer() const { return current; }
```

Реалізація методів класу List за допомогою Iterator

```
template <typename T>
typename List<T>::Iterator List<T>::begin() { return Iterator(head); }

template <typename T>
typename List<T>::Iterator List<T>::end() { return Iterator(nullptr); }

template <typename T>
typename List<T>::Iterator List<T>::rbegin() { return Iterator(tail); }

template <typename T>
typename List<T>::Iterator List<T>::rend() { return Iterator(nullptr); }

template <typename T>
typename List<T>::Iterator List<T>::find(const T& data) {
    for (Iterator it = begin(); it != end(); ++it) {
        if (*it == data) return it;
    }
    return end();
}
```

Реалізація методів класу List за допомогою Iterator

```
template <typename T>
void List<T>::erase(Iterator pos) {
    if (pos == end()) return;

    Node* node = pos.getPointer();
    if (!node) return;

    if (node == head) {
        head = head->next;
        if (head) head->prev = nullptr;
        else tail = nullptr;
    }
    else if (node == tail) {
        tail = tail->prev;
        if (tail) tail->next = nullptr;
        else head = nullptr;
    }
    else {
        node->prev->next = node->next;
        node->next->prev = node->prev;
    }
    delete node;
    size--;
}
```

Використання Iterator для обходу списку

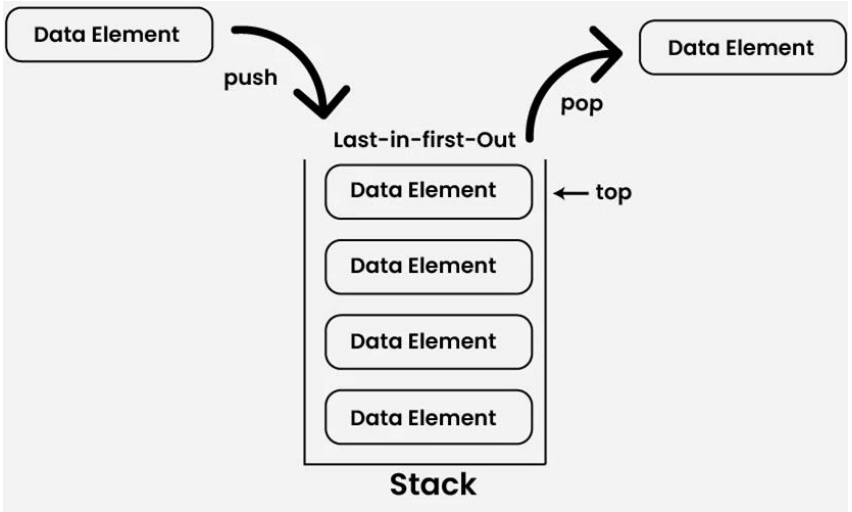
```cpp
int main() {
    List<int> list;
    list.push_front(3);
    list.push_front(2);
    list.push_front(1);
    list.push_back(4);
    list.push_back(5);

    std::cout << "Elements in forward order: ";
    for (auto it = list.begin(); it != list.end(); ++it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    std::cout << "Elements in reverse order: ";
    for (auto it = list.rbegin(); it != list.rend(); --it) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    auto it = list.find(3);
    if (it != list.end()) {
        std::cout << "Found element 3, erasing it" << std::endl;
        list.erase(it);
    }

    // range-based for loop
    std::cout << "List after erasing 3: ";
    for (const auto& el : list) {
        std::cout << el << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Стек

FACULTY OF APPLIED
MATHEMATICS AND
INFORMATICS
LVIV UNIVERSITY

Stack

- **Стек** — це лінійна структура даних, яка працює за принципом **LIFO (Last In, First Out – останнім прийшов, першим вийшов)**, тобто останній доданий елемент видаляється першим. Це означає, що операції додавання (**push**) і видалення (**pop**) виконуються тільки з одного кінця.



Типи стеків

Стек із фіксованим розміром (**Fixed Size Stack**)

- ❏ Має фіксований розмір і не може змінюватися динамічно.
- ❏ Якщо стек заповнений і спробувати додати елемент, виникає **помилка переповнення (overflow error)**.
- ❏ Якщо стек порожній і спробувати видалити елемент, виникає **помилка недостатності (underflow error)**.

Стек із динамічним розміром (**Dynamic Size Stack**)

- ❏ Може змінювати розмір динамічно (збільшуватися або зменшуватися).
- ❏ Якщо стек заповнений, його розмір автоматично збільшується для розміщення нового елемента.
- ❏ Якщо стек порожній, його розмір може зменшуватися.
- ❏ Такий стек зазвичай реалізується за допомогою **зв’язаного списку**, що дозволяє легко змінювати його розмір.

Базові операції в Stack

Щоб виконувати маніпуляції зі стеком, нам доступні певні операції:

- **push()** – додає елемент у стек.
- **pop()** – видаляє елемент зі стеку.
- **top()** – повертає верхній елемент стеку.
- **isEmpty()** – повертає `true`, якщо стек порожній, і `false` – якщо ні.
- **isFull()** – повертає `true`, якщо стек заповнений, і `false` – якщо ні.

Для реалізації стеку необхідно підтримувати посилання на його **верхній** елемент.

Додавання елемента в стек (Push)

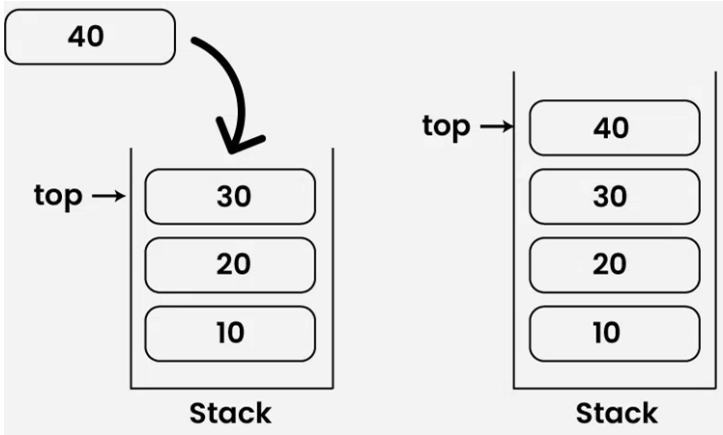
Перед додаванням елемента перевіряємо, чи стек не заповнений.

Якщо стек заповнений (**top == capacity - 1**), виникає **переповнення стеку (Stack Overflow)**, і вставка елемента неможлива.

В іншому випадку:

- Збільшуємо значення `top` на 1 (**top = top + 1**).
- Вставляємо новий елемент у позицію `top`.

Елементи можуть додаватися в стек, поки його розмір не досягне **граничної місткості (capacity)**.

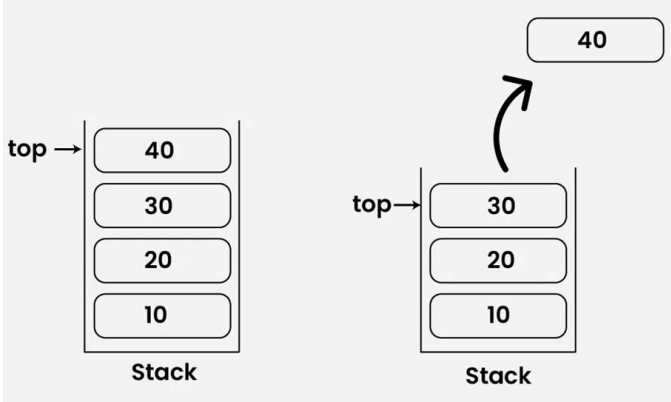


Видалення елемента зі стеку (Pop)

Видаляє елемент зі стеку. Елементи видаляються у зворотному порядку відносно їх додавання (**LIFO — Last In, First Out**). Якщо стек порожній, виникає **перепустка (Underflow condition)**.

Алгоритм для операції Pop:

1. Перед видаленням елемента перевіряємо, чи стек не порожній.
2. Якщо стек порожній (**top == -1**), виникає **перепустка стеку (Stack Underflow)**, і видалення неможливе.
3. В іншому випадку:
    - Зберігаємо значення верхнього елемента (`top`).
    - Зменшуємо значення `top` на 1 (**top = top - 1**).
    - Повертаємо збережене значення верхнього елемента.

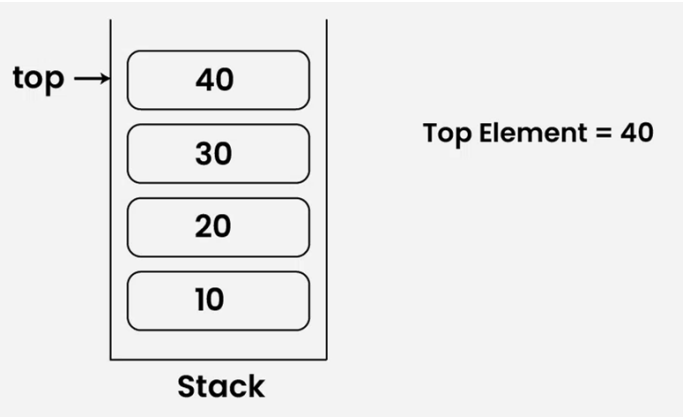


Операція перегляду верхнього елемента стеку (Top або Peek)

Повертає верхній елемент стеку без його видалення.

Алгоритм для операції `Top`:

1. Перед поверненням верхнього елемента перевіряємо, чи стек не порожній.
2. Якщо стек порожній (**top == -1**), виводимо повідомлення: **"Стек порожній"**.
3. В іншому випадку повертаємо значення, яке зберігається за індексом `top`.



Приклад реалізації

```cpp
template<typename T>
class Stack {
private:
    struct Node {
        T data;
        Node* next;
        Node(const T& value, Node* nextNode = nullptr) :
            data(value), next(nextNode) {}
    };
    Node* head;
    int size;

public:
    Stack();
    ~Stack();
    void push(const T& value);
    void pop();
    const T& top() const;
    bool empty() const;
    int getSize() const;
};
template<typename T>
Stack<T>::Stack() {
    head = nullptr;
    size = 0;
}
template<typename T>
Stack<T>::~Stack() {
    while (head) {
        Node* temp = head;
        head = head->next;
        delete temp;
    }
}
```

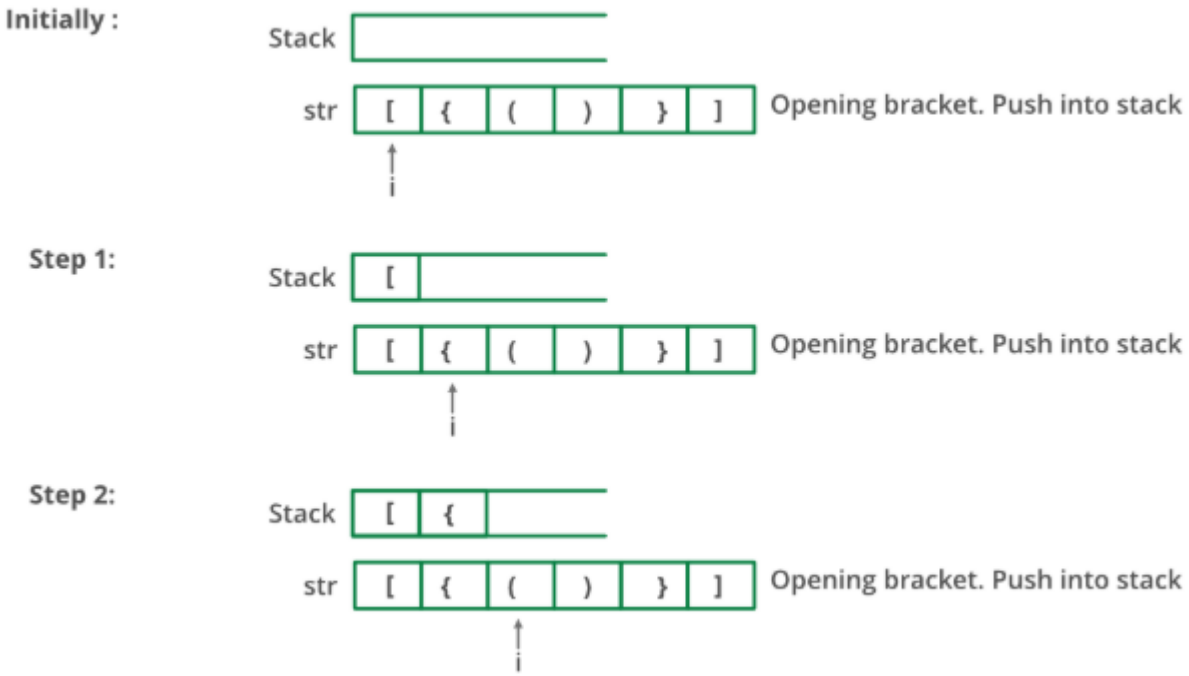
```cpp
template<typename T>
void Stack<T>::push(const T& value) {
    Node* newNode = new Node(value, head);
    head = newNode;
    size++;
}
template<typename T>
void Stack<T>::pop() {
    if (empty()) {
        std::cout << "Stack is empty, cannot pop!" << std::endl;
        return;
    }
    Node* temp = head;
    head = head->next;
    delete temp;
    size--;
}
template<typename T>
const T& Stack<T>::top() const {
    if (empty()) {
        std::cout << "Stack is empty, no top element!" << std::endl;
        return T(); // Повертаємо значення за замовчуванням
    }
    return head->data;
}

template<typename T>
bool Stack<T>::empty() const {
    return size == 0;
}

template<typename T>
int Stack<T>::getSize() const {
    return size;
}
```

Задача визначення збалансованості дужок

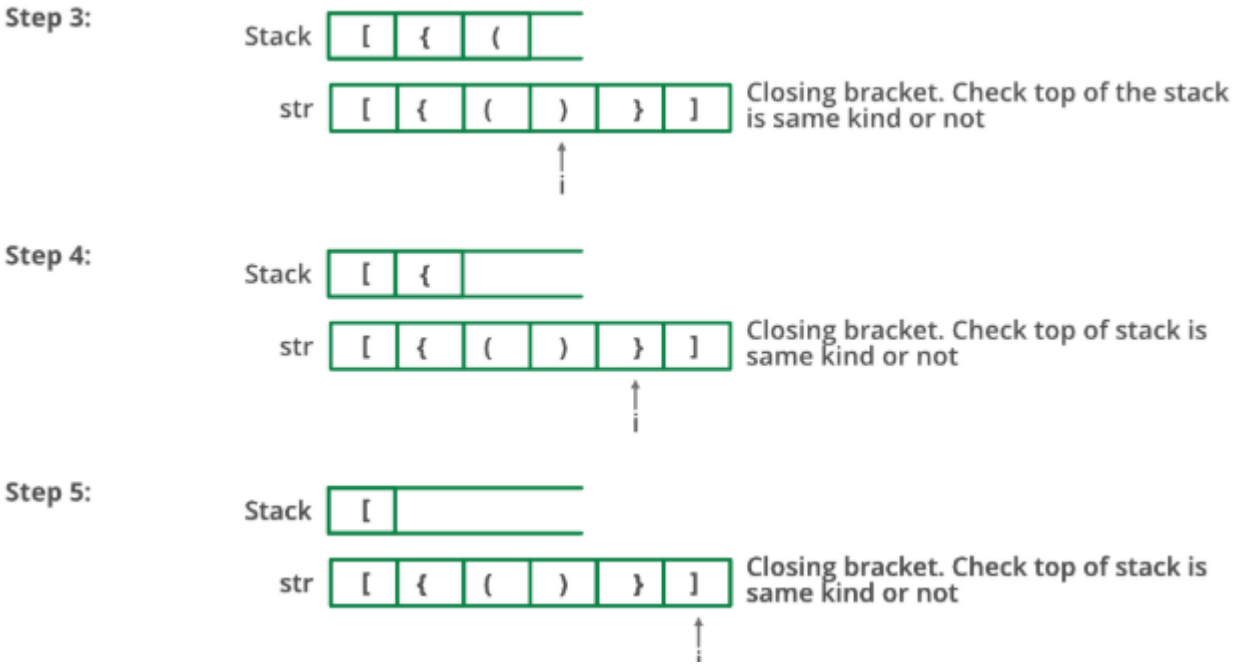
Визначити чи збалансовані дужки

```
Input: exp = “[()]{}{[()()]()}”
Output: Balanced

Input: exp = “[(])”
Output: Not Balanced
```

Застосування стеку до розв’язування задачі



Застосування стеку до розв’язування задачі



Програмна реалізація задачі

```cpp
bool isBalanced(const std::string& exp) {
    Stack<char> stack;

    // Відповідність відкриваючих і закриваючих дужок
    for (char ch : exp) {
        if (ch == '(' || ch == '[' || ch == '{') {
            stack.push(ch);  // Додаємо відкриваючі дужки до стека
        }
        else if (ch == ')' || ch == ']' || ch == '}') {
            if (stack.empty()) {
                return false;  // Якщо стек порожній, то немає відповідної відкриваючої дужки
            }

            char top = stack.top();
            stack.pop();

            // Перевірка відповідності пар дужок
            if ((ch == ')' && top != '(') ||
                (ch == ']' && top != '[') ||
                (ch == '}' && top != '{')) {
                return false;
            }
        }
    }
    return stack.empty();  // Якщо стек порожній в кінці, то дужки збалансовані
}
```

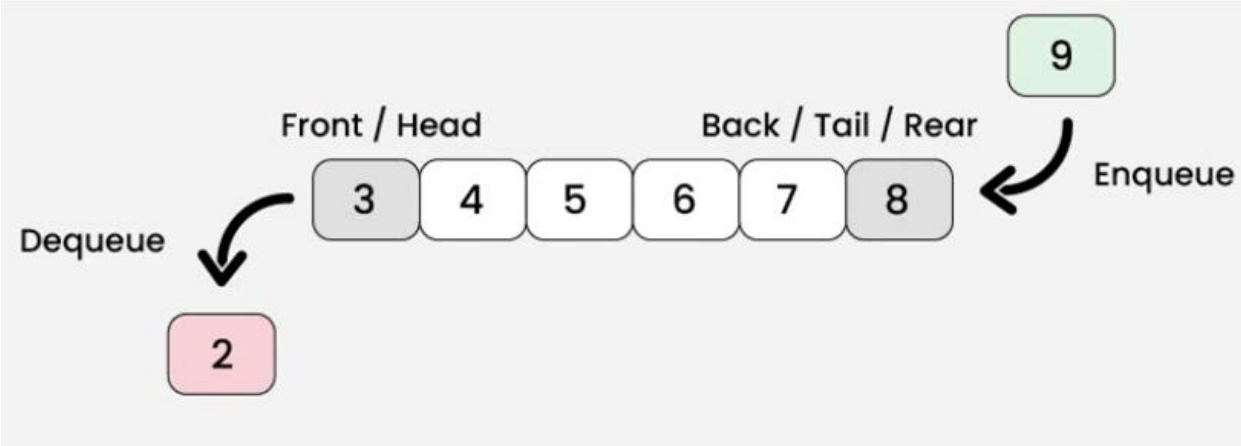
Черга

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Структура даних Черга (Queue)

Черга — це фундаментальна структура даних у комп'ютерних науках, яка використовується для зберігання та управління даними в певному порядку.

- **Принцип FIFO (First In, First Out)** – перший елемент, який додається до черги, буде видалений першим.
- Черги широко використовуються в алгоритмах і додатках завдяки **простоті та ефективності** в управлінні потоком даних.



Базові операції в Queue

Щоб виконувати маніпуляції з чергою, нам доступні такі операції:

- **enqueue() (or push())** – додає елемент у кінець черги.
- **dequeue() (or pop())** – видаляє елемент з початку черги.
- **front()** – повертає перший (початковий) елемент черги.
- **back()** – повертає останній (кінцевий) елемент черги.
- **isEmpty()** – повертає `true`, якщо черга порожня, і `false` – якщо ні.
- **isFull()** – повертає `true`, якщо черга заповнена, і `false` – якщо ні.

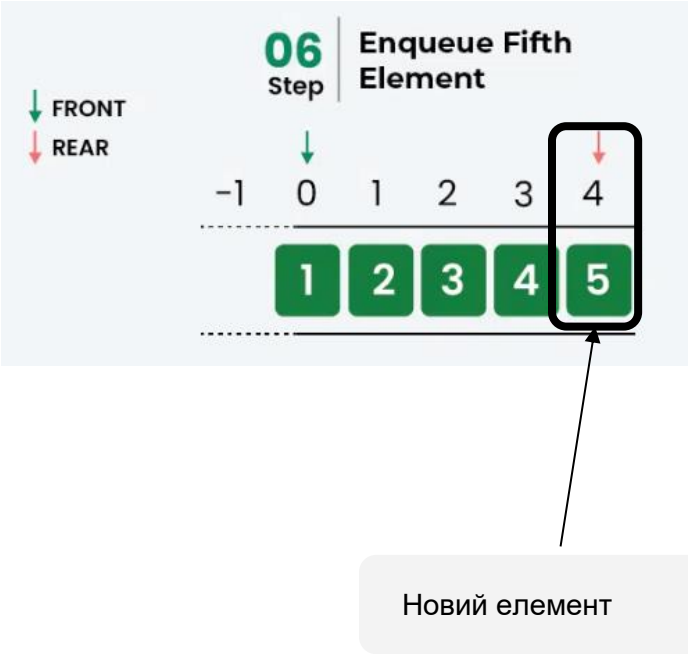
Для реалізації черги необхідно підтримувати **посилання на початок (front) і кінець (back) черги**.

Операція Enqueue

Операція **enqueue** додає (або зберігає) елемент у **кінець черги**.

Кроки:

1. Перевірити, чи черга **заповнена**.
   - Якщо так → повернути **помилку переповнення (overflow error)** та завершити операцію.
2. Якщо черга **не заповнена**, **збільшити back** (вказівник на останній елемент) на наступну доступну позицію.
3. Вставити новий елемент у позицію **back**.

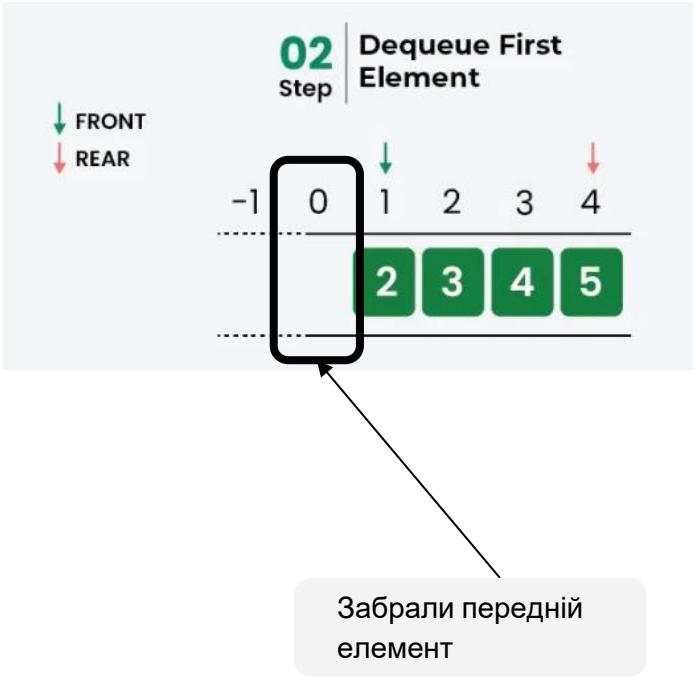


Операція Dequeue

Операція **dequeue** видаляє **перший (фронтальний) елемент** із черги.

Кроки:

1. Перевірити, чи черга **порожня**.
   - Якщо так → повернути **помилку недостатності (underflow error)**.
2. Видалити елемент у позиції **front**.
3. Збільшити **front** (вказівник на перший елемент списку) на наступний елемент.



Приклад реалізації черги на основі списку

```cpp
template<class T>
class Node {
public:
    T data;
    Node<T>* next;
    Node(T data);
};

template<class T>
Node<T>::Node(T data) {
    this->data = data;
    next = nullptr;
}

template<class T>
class LinkedList {
public:
    Node<T>* head;
    Node<T>* tail;
    size_t size;

    LinkedList();
    ~LinkedList();
    void append(T data);
    bool pop(T& data);
    bool front(T& data) const;
    bool back(T& data) const;
    size_t getSize() const;
};
```

```cpp
template<class T>
class Queue {
private:
    LinkedList<T>* list;
public:
    Queue();
    ~Queue();
    void enqueue(T data);
    bool dequeue(T& data);
    bool front(T& data);
    bool back(T& data);
    size_t size() const;
};

template<class T>
Queue<T>::Queue() {
    list = new LinkedList<T>();
}
template<class T>
Queue<T>::~Queue() {
    delete list;
}
template<class T>
void Queue<T>::enqueue(T data) {
    list->append(data);
}
template<class T>
bool Queue<T>::dequeue(T& data) {
    return list->pop(data);
}
template<class T>
bool Queue<T>::front(T& data) {
    return list->front(data);
}
template<class T>
bool Queue<T>::back(T& data) {
    return list->back(data);
}
template<class T>
size_t Queue<T>::size() const {
    return list->getSize();
}
```

Бінарне пошукове дерево

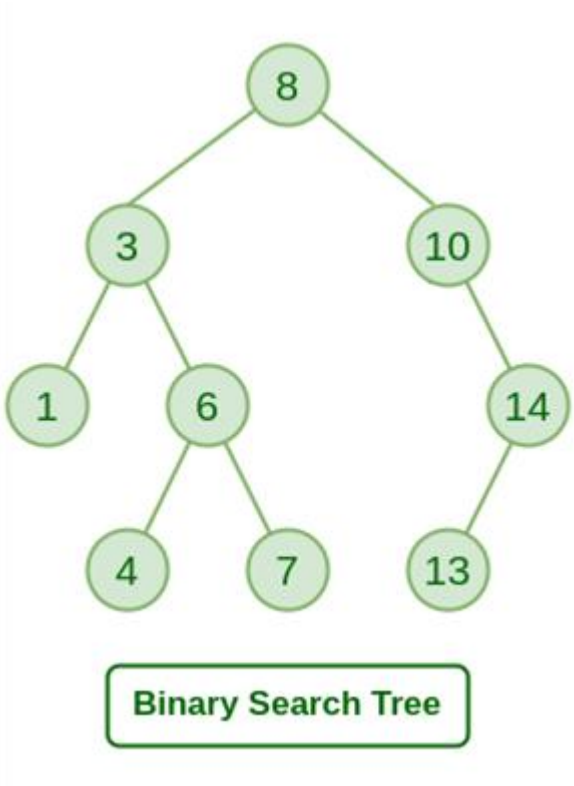
FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Бінарне дерево пошуку (Binary Search Tree, BST)

Бінарне дерево пошуку (BST) — це структура даних, яка використовується в комп’ютерних науках для впорядкованого зберігання та організації даних.

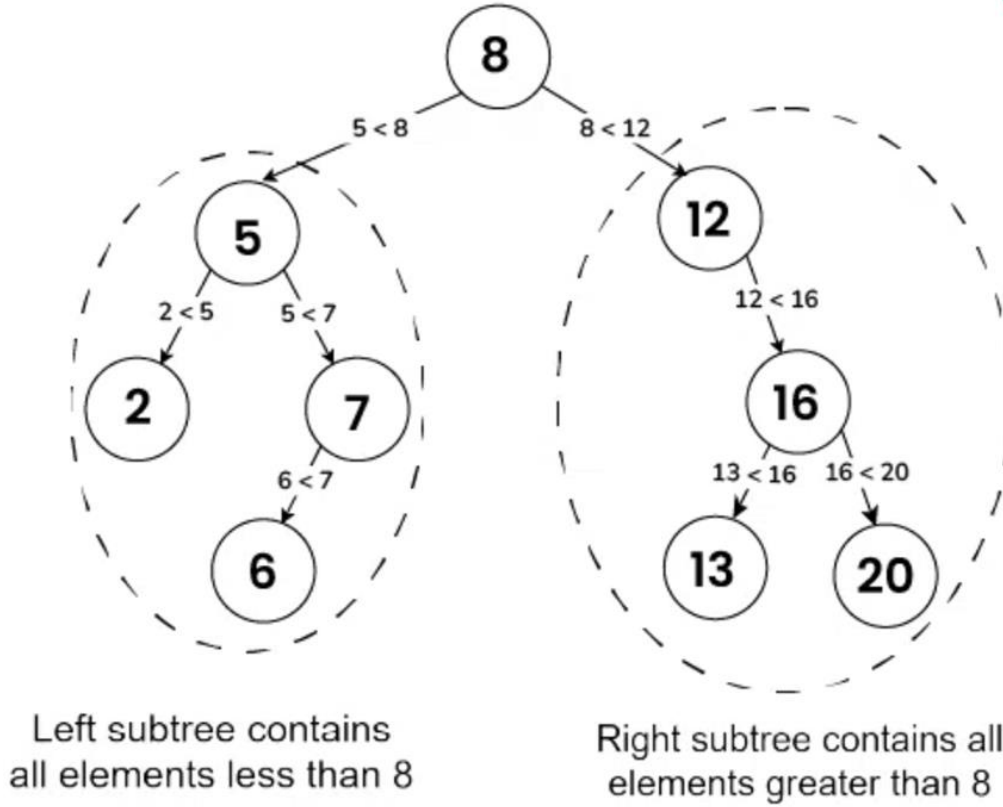
Основні характеристики BST:

- Кожен вузол має **не більше двох дочірніх вузлів**: **лівий** та **правий**.
- **Лівий дочірній вузол** містить значення **менше** за значення батьківського вузла.
- **Правий дочірній вузол** містить значення **більше** за значення батьківського вузла.
- Завдяки цій ієрархічній структурі, BST забезпечує **ефективний пошук, вставку та видалення** даних.



Властивості бінарного дерева пошуку (BST)

- **Ліве піддерево** вузла містить тільки ті вузли, **ключі яких менші** за ключ батьківського вузла.
- **Праве піддерево** вузла містить тільки ті вузли, **ключі яких більші** за ключ батьківського вузла.
- **Ліве і праве піддерева** кожного вузла також повинні бути **бінарними деревами пошуку**.
- **Вузли не повинні містити дублікатів** (проте існують різні підходи до обробки дублікатів у BST).



Корисність бінарного дерева пошуку (BST)

Підтримує впорядкований потік даних та дозволяє ефективно виконувати основні операції:

- **Пошук (`search`)**
- **Вставка (`insert`)**
- **Видалення (`delete`)**
- **Знаходження наступного більшого (`ceiling`)**
- **Максимальне (`max`) та мінімальне (`min`) значення**

Часова складність цих операцій становить **O(h)**, де `h` — висота дерева.

Обхід дерева завжди дозволяє отримати елементи у **відсортованому порядку**

Переваги та недоліки BST

Переваги

- **Ефективний пошук** – Часова складність пошуку в **самобалансованому BST** становить **O(log n)**.
- **Впорядкована структура** – Елементи зберігаються у **відсортованому порядку**, що дозволяє легко знаходити **наступний або попередній елемент**.
- **Динамічна вставка та видалення** – Додавання та видалення елементів відбувається **ефективно**.
- **Збалансована структура**
- **Подвійна черга з пріоритетами**

Недоліки

- **Не самобалансоване (звичайне BST)** – Може **вироджуватися** у **звичайний список**, що призведе до погіршення продуктивності.
- **Найгірша часова складність** – У **найгіршому випадку** (коли дерево вироджується) операції пошуку та вставки можуть займати **O(n)** часу.
- **Витрати пам’яті** – Потрібно зберігати **додаткові вказівники** на лівого та правого нащадків кожного вузла.
- **Не підходить для дуже великих наборів даних** – Для **великих обсягів даних** інші структури (наприклад, **B-дерева або хеш-таблиці**) можуть бути ефективнішими.
- **Обмежений функціонал**

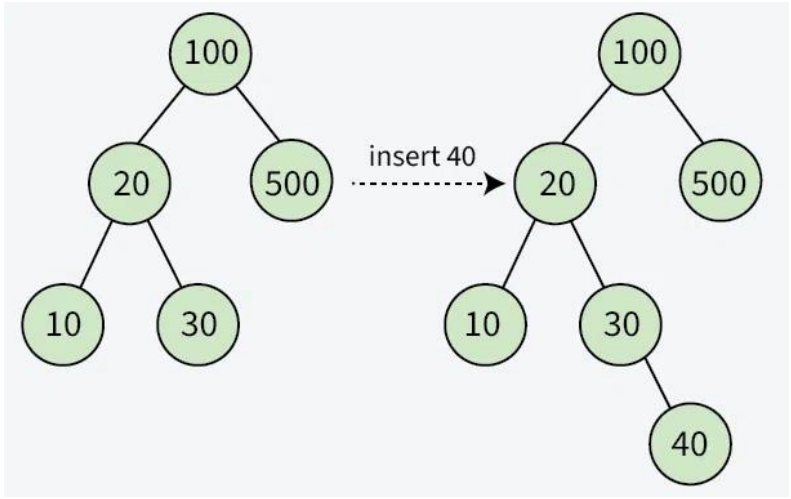
Структура Node

```
// Визначення вузла шаблонного бінарного дерева пошуку
template <typename T>
class BinaryTree {
private:
    // Визначення структури вузла дерева
    struct Node {
        T data;  // Дані, які зберігаються в цьому вузлі
        Node* left;  // Вказівник на ліве піддерево
        Node* right;  // Вказівник на праве піддерево

        // Конструктор для ініціалізації вузла
        Node(T value) {
            data = value;  // Присвоєння значення даним вузла
            left = right = nullptr;  // Ініціалізація вказівників на піддерева значенням nullptr
        }
    };
    Node* root;  // Вказівник на корінь дерева
public:
    BinaryTree();  // Конструктор
    ~BinaryTree();  // Деструктор
    void insert(T value);  // Метод для вставки елемента в дерево
    void inorder() const;  // Метод для обходу дерева у порядку зростання
    void deleteNode(T key);  // Метод для видалення елемента з дерева
private:
    Node* insert(Node* node, T value);  // Допоміжний метод для вставки в дерево
    void inorder(Node* node) const;  // Допоміжний метод для обходу дерева
    Node* findMin(Node* node) const;  // Метод для пошуку мінімального елемента
    Node* deleteNode(Node* node, T key);  // Допоміжний метод для видалення вузла
    void deleteTree(Node* node);  // Допоміжний метод для очищення дерева
};
```

insert

Як вставити значення у бінарне дерево пошуку (BST)?

Новий ключ **завжди вставляється у листовий вузол**, зберігаючи властивості BST. Ми починаємо **пошук місця вставки від кореня**, поки не знайдемо **листовий вузол**. Після цього новий вузол додається як **дочірній елемент**.

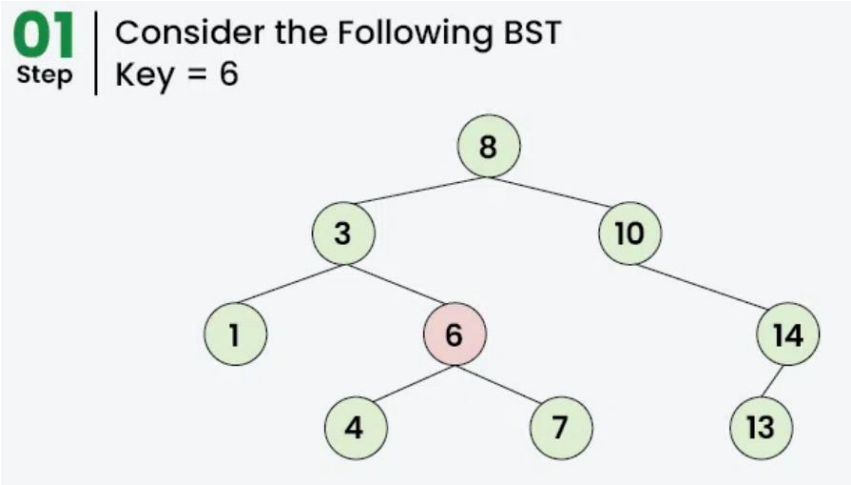


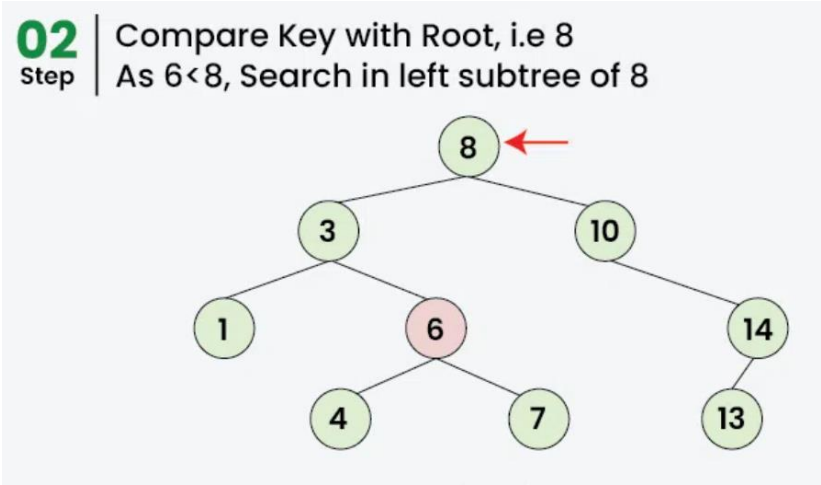

```cpp
// Метод вставки у шаблонне BST
template <typename T>
typename BinaryTree<T>::Node* BinaryTree<T>::insert(Node* node, T value) {
    if (node == nullptr)
        return new Node(value);  // Якщо вузол порожній, створюємо новий вузол
    if (value < node->data)
        node->left = insert(node->left, value);  // Якщо значення менше, йдемо в ліве піддерево
    else if (value > node->data)
        node->right = insert(node->right, value);  // Якщо значення більше, йдемо в праве піддерево
    return node;  // Повертаємо поточний вузол після вставки
}
```

search

Як знайти значення у бінарному дереві пошуку (BST)?

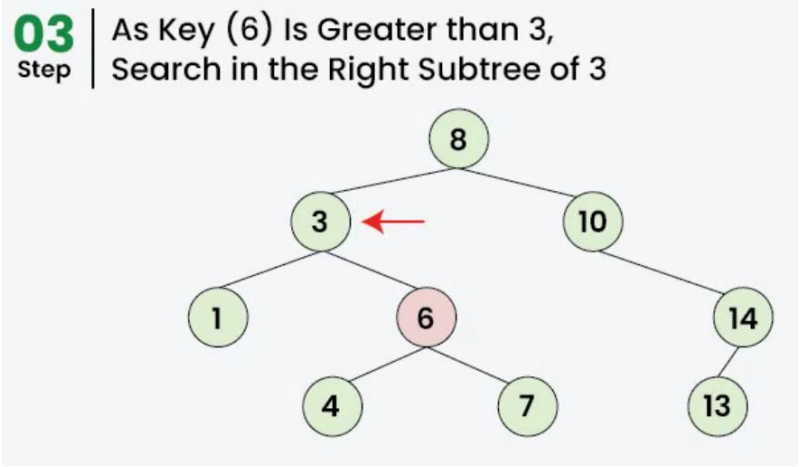
Пошук у BST використовує **рекурсивний** або **ітеративний підхід**. Оскільки BST впорядковане, ми можемо ефективно знайти ключ, **рухаючись вліво або вправо** залежно від значення.

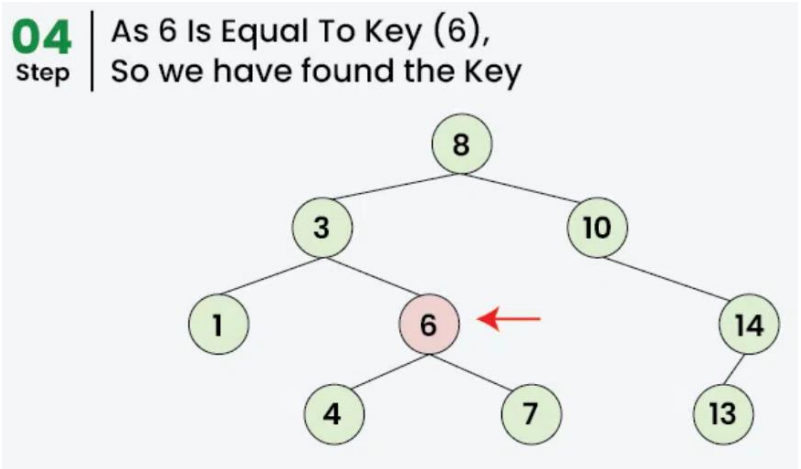




search

```cpp
// Пошук вузла у дереві
template <typename T>
bool BinaryTree<T>::search(Node* node, T key) const {
    if (node == nullptr) return false;
    if (node->data == key) return true;
    if (key < node->data)
        return search(node->left, key);
    else
        return search(node->right, key);
}

template <typename T>
bool BinaryTree<T>::search(T key) const {
    return search(root, key);
}
```





deleteNode

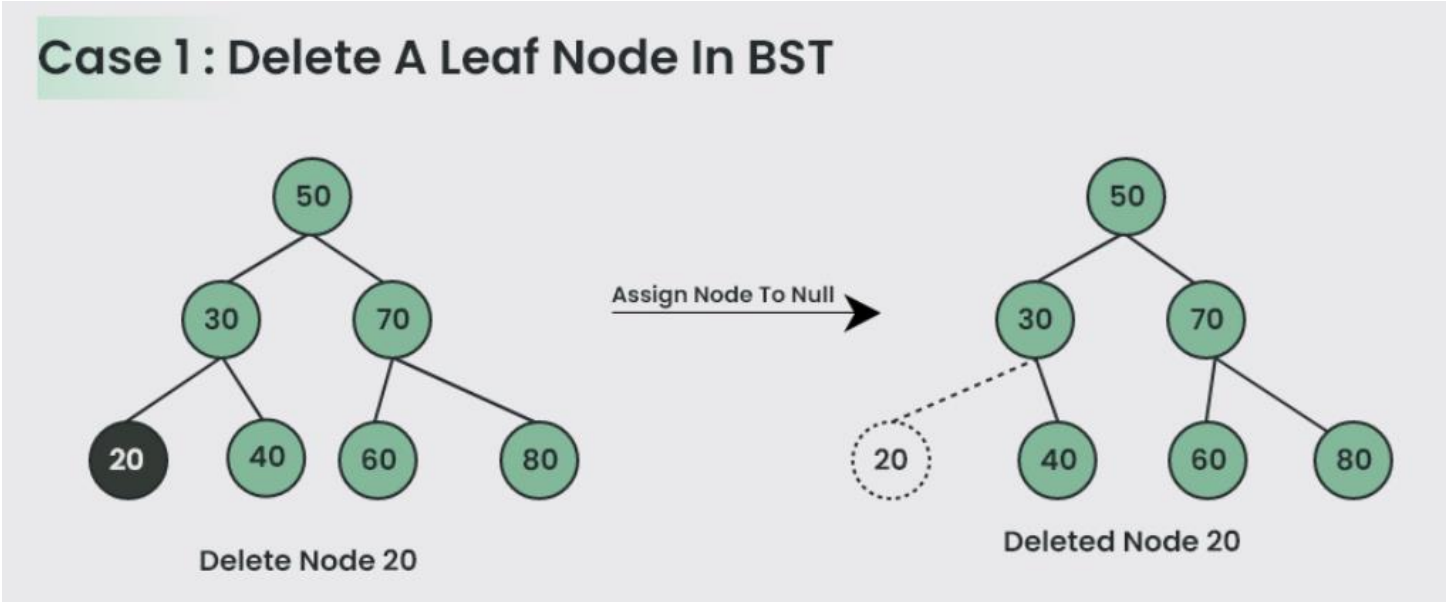
Як видалити вузол у бінарному дереві пошуку (BST)?

Операція **видалення** в BST складніша за вставку та пошук, оскільки потрібно зберегти структуру дерева.

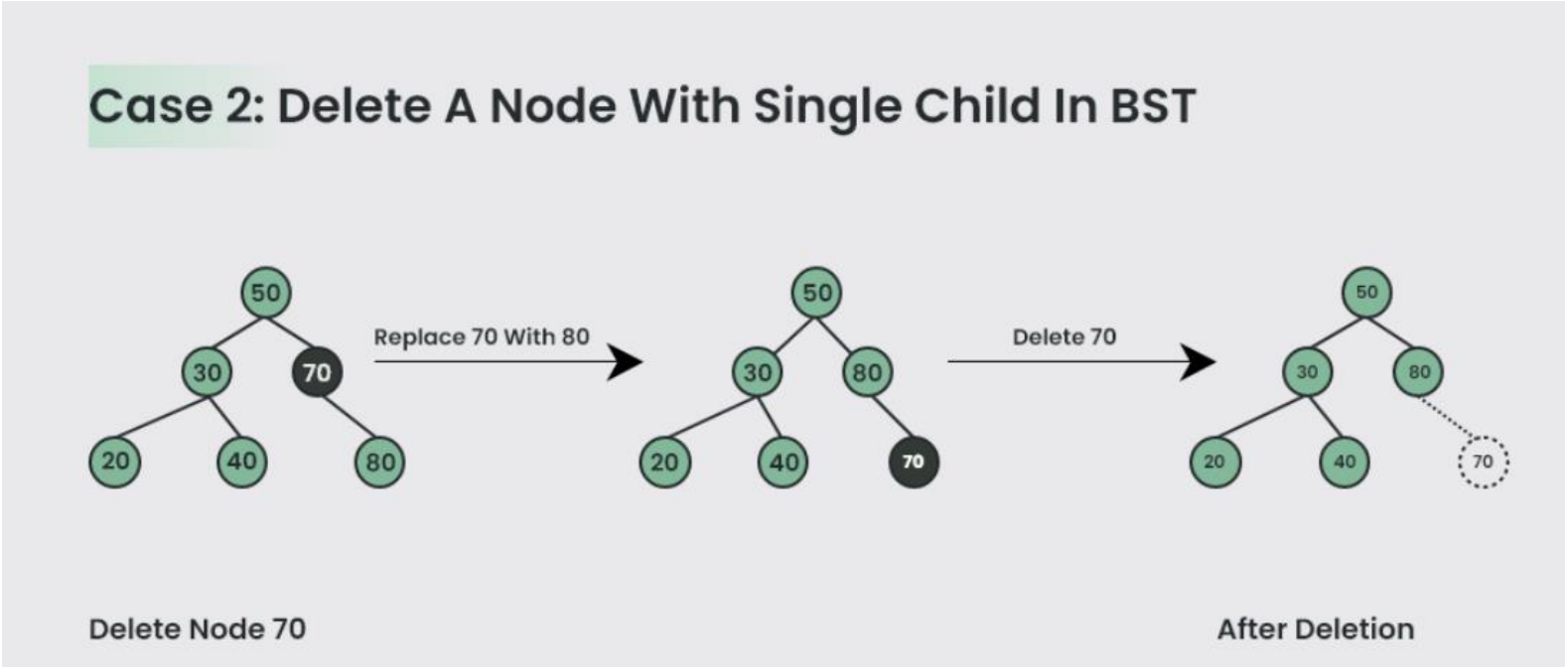
Можливі випадки видалення вузла:

1. **Вузол – листок (не має дітей)** → Просто видаляємо його.
2. **Вузол має одного нащадка** → Замінюємо вузол його дочірнім елементом.
3. **Вузол має двох нащадків** →
    - Знаходимо **мінімальний елемент** у правому піддереві (його **inorder successor**).
    - Замінюємо значення вузла цим мінімальним значенням.
    - Видаляємо цей мінімальний вузол у правому піддереві.

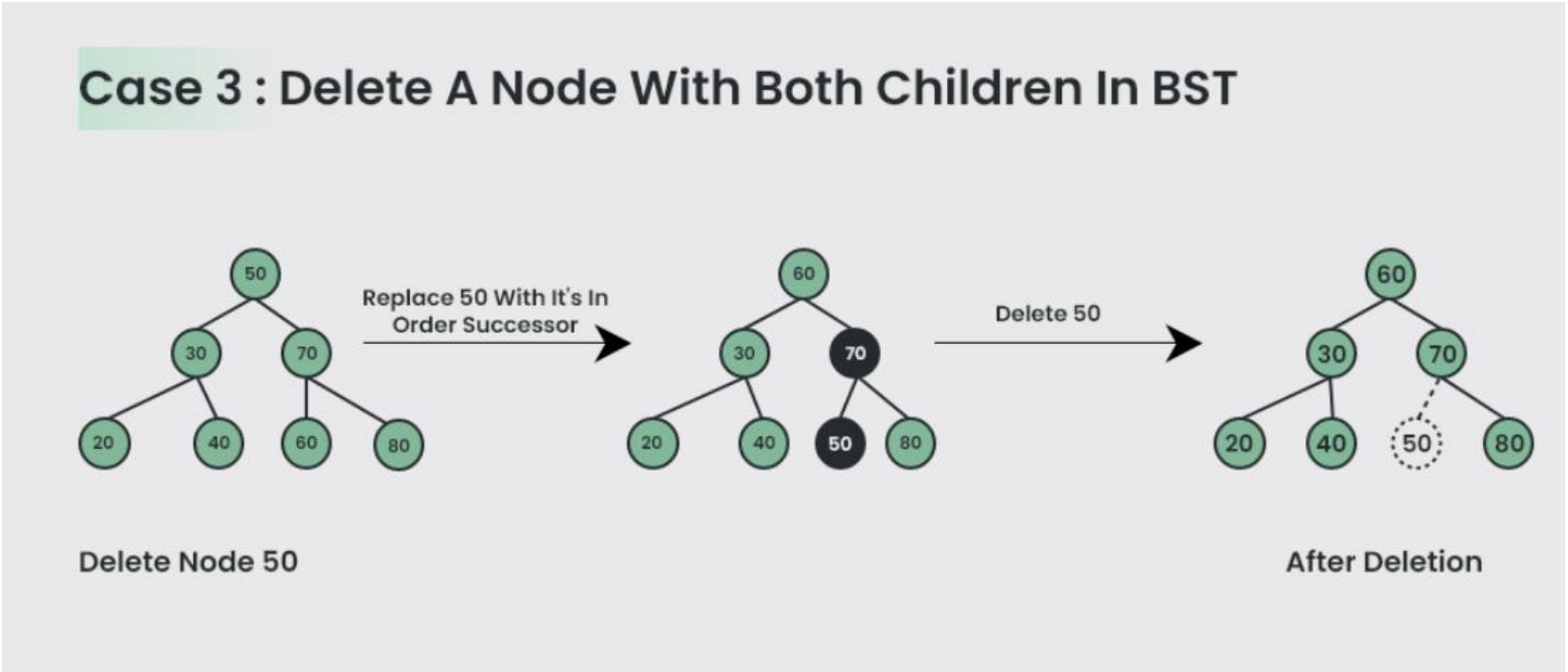
deleteNode


Case 1 : Delete A Leaf Node In BST
50
30
70
20
40
60
80
Delete Node 20
Assign Node To Null
50
30
70
20
40
60
80
Deleted Node 20

deleteNode


Case 2: Delete A Node With Single Child In BST
50
30
70
20
40
80
Replace 70 With 80
50
30
80
20
40
70
Delete 70
50
30
80
20
40
70
Delete Node 70
After Deletion

deleteNode


Case 3 : Delete A Node With Both Children In BST
50
30
70
20
40
60
80
Replace 50 With It's In Order Successor
60
30
70
20
40
50
80
Delete 50
60
30
70
20
40
50
80
Delete Node 50
After Deletion

Приклад

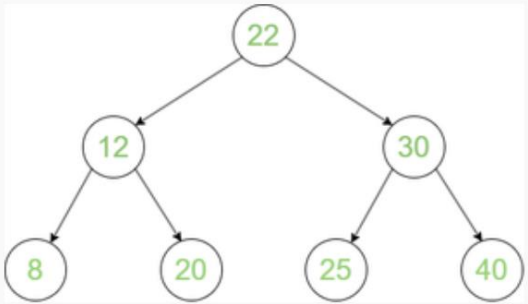
```cpp
// Метод пошуку мінімального елемента у дереві (inorder successor)
template <typename T>
typename BinaryTree<T>::Node* BinaryTree<T>::findMin(Node* node) const {
    while (node && node->left != nullptr)
        node = node->left;  // Продовжуємо рухатися ліворуч, поки не знайдемо найменший елемент
    return node;  // Повертаємо мінімальний елемент
}
```

```cpp
// Метод для видалення вузла з дерева
template <typename T>
typename BinaryTree<T>::Node* BinaryTree<T>::deleteNode(Node* node, T key) {
    if (node == nullptr) return node;  // Якщо вузол порожній, повертаємо порожній вузол

    // Рекурсивно шукаємо вузол для видалення
    if (key < node->data)
        node->left = deleteNode(node->left, key);  // Якщо значення менше, йдемо в ліве піддерево
    else if (key > node->data)
        node->right = deleteNode(node->right, key);  // Якщо значення більше, йдемо в праве піддерево
    else {
        // Випадок 1: Вузол - листок або має лише одного нащадка
        if (node->left == nullptr) {
            Node* temp = node->right;
            delete node;  // Видаляємо поточний вузол
            return temp;  // Повертаємо праве піддерево (якщо воно є)
        }
        else if (node->right == nullptr) {
            Node* temp = node->left;
            delete node;  // Видаляємо поточний вузол
            return temp;  // Повертаємо ліве піддерево (якщо воно є)
        }

        // Випадок 2: Вузол має двох нащадків
        Node* temp = findMin(node->right);  // Знаходимо наступника (найменший елемент у правому піддереві)
        node->data = temp->data;  // Замінюємо значення поточного вузла на значення наступника
        node->right = deleteNode(node->right, temp->data);  // Видаляємо наступника з правого піддерева
    }
    return node;  // Повертаємо модифікований вузол
}
```

Методи обходу дерева

- **Inorder traversal (ЛКП – Лівий, Корінь, Правий)** – це спосіб обходу дерева, який **виводить вузли у відсортованому порядку**.
- Preorder traversal (КЛП - Корінь, Лівий, Правий)
- Postorder traversal (ЛПК - Лівий, Правий, Корінь)

Input:



Binary Search Tree

Output:

Inorder Traversal: 8 12 20 22 25 30 40
Preorder Traversal: 22 12 8 20 30 25 40
Postorder Traversal: 8 20 12 25 40 30 22

Inorder traversal

Inorder traversal (ЛКП – Лівий, Корінь, Правий) – це спосіб обходу дерева, який **виводить вузли у відсортованому порядку**.

```
// Метод обходу дерева у порядку зростання (Inorder Traversal)
template <typename T>
void BinaryTree<T>::inorder(Node* node) const {
    if (node == nullptr) return;  // Якщо вузол порожній, виходимо
    inorder(node->left);   // Обхід лівого піддерева
    cout << node->data << " ";  // Виведення значення поточного вузла
    inorder(node->right);  // Обхід правого піддерева
}

// Метод для виведення елементів дерева у порядку зростання
template <typename T>
void BinaryTree<T>::inorder() const {
    inorder(root);  // Викликаємо допоміжний метод для обходу всього дерева
    cout << endl;  // Перехід на новий рядок після виведення
}
```

Приклад використання

```cpp
int main() {
    // Створення дерева з цілими числами
    BinaryTree<int> tree;
    tree.insert(50);
    tree.insert(30);
    tree.insert(70);
    tree.insert(20);
    tree.insert(40);
    tree.insert(60);
    tree.insert(80);

    // Видалення елементу
    tree.deleteNode(60);  // Видаляємо вузол зі значенням 60

    // Виведення елементів дерева у порядку зростання
    cout << "Inorder Traversal (sorted order): ";
    tree.inorder();  // output: 20 30 40 50 70 80

    // Створення дерева з рядками
    BinaryTree<string> stringTree;
    stringTree.insert("apple");
    stringTree.insert("orange");
    stringTree.insert("banana");

    // Виведення елементів дерева з рядками у порядку зростання
    cout << "Inorder Traversal (sorted order for strings): ";
    stringTree.inorder();  // output: apple banana orange
    string searchKey = "banana";
    if (stringTree.search(searchKey))
        cout << searchKey << " Found in a tree.\n";
    else
        cout << searchKey << " Not found in a tree.\n";

    return 0;
}
```

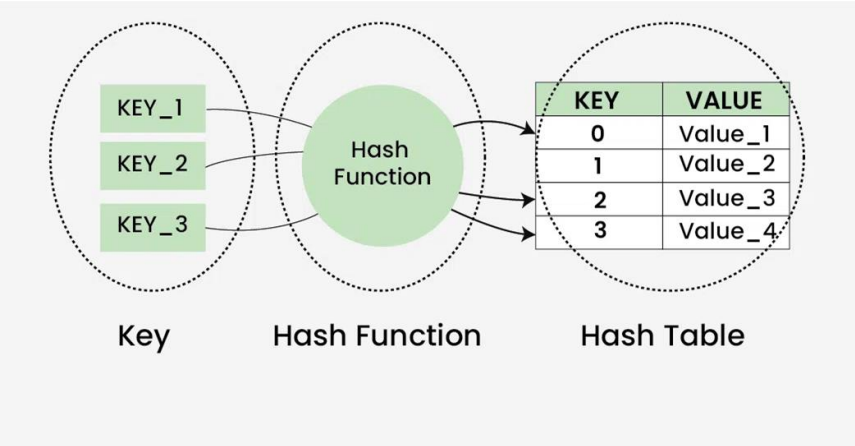
Геш-таблиці

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Геш-таблиця (Hash Table)

Геш-таблиця — це структура даних, яка використовується для **швидкого додавання, пошуку та видалення** пар **ключ-значення**.

- Працює на основі **гешування**, де кожен ключ обробляється **геш-функцією** і перетворюється на **унікальний індекс** у масиві.
- Отриманий індекс використовується як **місце зберігання** відповідного значення.
- **Простими словами:** геш-таблиця **відображає ключі у відповідні значення** для ефективного доступу.



Коефіцієнт заповнення (Load Factor)

Коефіцієнт заповнення **геш-таблиці** визначається як **відношення кількості збережених елементів до розміру таблиці**.

Якщо коефіцієнт заповнення **високий**, таблиця може **переповнюватися**, що призводить до **довшого часу пошуку** та **колізій**.

Для підтримки **оптимального коефіцієнта заповнення** використовують:

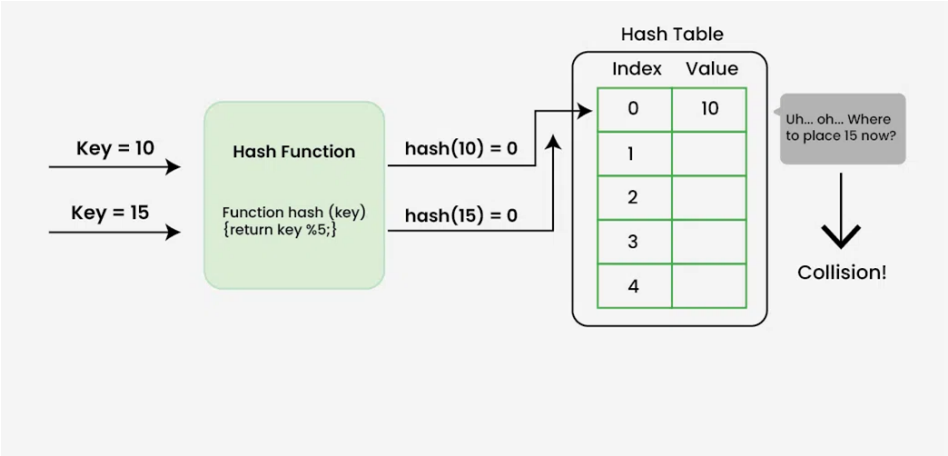
- **ефективну геш-функцію** для рівномірного розподілу ключів,
- **масштабування таблиці** (збільшення її розміру при перевищенні певного рівня заповнення).

Геш-функція

- Функція, яка перетворює ключі в індекси масиву, називається **геш-функцією**. Хороша геш-функція повинна **рівномірно** розподіляти ключі по масиву, щоб зменшити кількість колізій і забезпечити швидкий пошук
- Припускається, що ключі є **цілими** числами в певному діапазоні. Це дозволяє використовувати основні методи гешування, такі як за **модулем** або **множенням**.
- **Гешування за модулем** використовує залишок від ділення ключа на розмір масиву як індекс. Якщо розмір масиву є простим числом і ключі рівномірно розподілені, метод працює добре.
- **Гешування множенням** множить ключ на константу між 0 і 1, після чого береться дробова частина отриманого результату. Потім індекс визначається множенням дробової частини на розмір масиву. Також цей метод ефективний, коли ключі рівномірно розподілені.

Методи розв’язання колізій

- Колізії виникають, коли два або більше ключів відправляються на один і той самий індекс масиву. Деякі методи розв’язання колізій включають **ланцюгове гешування (chaining), відкрите адресування (open addressing)** та **подвійне гешування (double hashing)**.

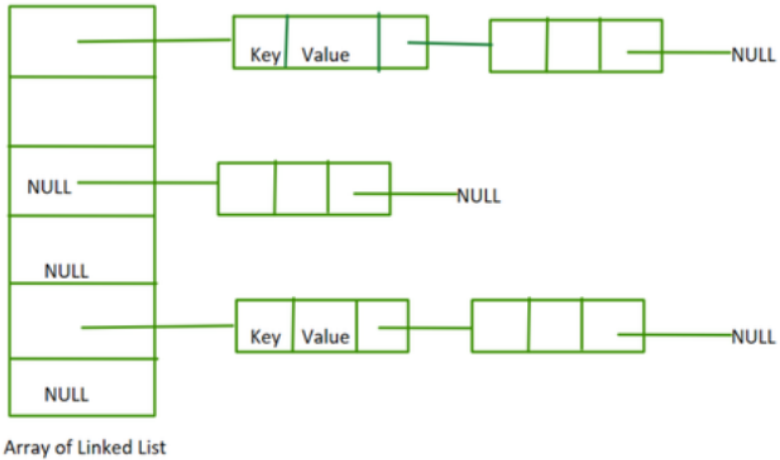


Відкрите адресування (Open Addressing)

- Колізії розв’язуються шляхом пошуку наступного вільного місця в таблиці. Якщо перша комірка вже зайнята, геш-функція застосовується до наступних комірок, поки не знайдеться вільне місце. Цей метод включає **подвійне гешування (double hashing), лінійне випробування (linear probing)** та **квадратичне випробування (quadratic probing)**.

Окреме ланцюгове гешування (Separate Chaining)

- У цьому методі для кожного індексу в геш-таблиці зберігається **зв’язаний список об'єктів**, що мають однаковий хеш. Якщо два ключі хешуються в один і той самий індекс, вони додаються до списку. Цей метод досить простий у реалізації та добре справляється з кількома колізіями.



Hash Map

Гешування Робіна Гуда (Robin Hood Hashing)

- Щоб **зменшити довжину ланцюга**, цей метод вирішує колізії шляхом **переміщення ключів**. Якщо новий ключ гешується у вже зайнятий слот, алгоритм порівнює відстань між ідеальним індексом і фактичним місцем збереження двох ключів. Якщо існуючий ключ **ближчий до свого ідеального слоту**, він залишається; інакше **новий ключ замінює старий**. Це дозволяє **зменшити кількість колізій та середню довжину ланцюга**.

Приклад

```
#include <bits/stdc++.h>
using namespace std;

struct Hash {
    int BUCKET; // No. of buckets

    // Vector of vectors to store the chains
    vector<vector<int>> table;

    // Inserts a key into hash table
    void insertItem(int key) {
        int index = hashFunction(key);
        table[index].push_back(key);
    }

    // Deletes a key from hash table
    void deleteItem(int key);

    // Hash function to map values to key
    int hashFunction(int x) {
        return (x % BUCKET);
    }

    void displayHash();

    // Constructor to initialize bucket count and table
    Hash(int b) {
        this->BUCKET = b;
        table.resize(BUCKET);
    }
};
```

Видалення елементу

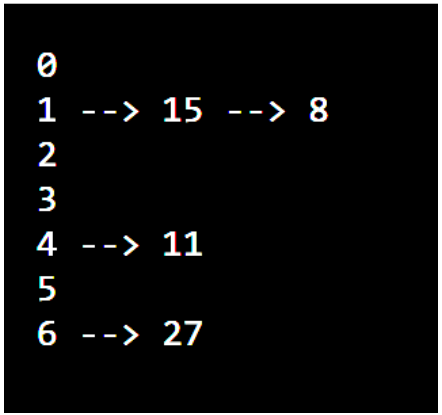
```
// Function to delete a key from the hash table
void Hash::deleteItem(int key) {
    int index = hashFunction(key);

    // Find and remove the key from the table[index] vector
    auto it = find(table[index].begin(), table[index].end(),
key);
    if (it != table[index].end()) {
        table[index].erase(it); // Erase the key if found
    }
 }
```

Відображення хеш таблиці

```
// Function to display the hash table
void Hash::displayHash() {
    for (int i = 0; i < BUCKET; i++) {
        cout << i;
        for (int x : table[i]) {
            cout << " --> " << x;
        }
        cout << endl;
    }
}
```

Приклад

```
// Driver program
int main() {
    // Vector that contains keys to be mapped
    vector<int> a = {15, 11, 27, 8, 12};

    // Insert the keys into the hash table
    Hash h(7); // 7 is the number of buckets
    for (int key : a)
        h.insertItem(key);

    // Delete 12 from the hash table
    h.deleteItem(12);

    // Display the hash table
    h.displayHash();

    return 0;
 }
```



Дякую

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY