Лабораторна работа 1.8(до Л2.5)

Мета роботи: Вивчення принципів організації і роботи з абстрактної структурою даних бінарного дерева.

Завдання:

Реалізувати операції:

1. Створенні бінарного дерева.
2. Додавання елемента до бінарного дерева.
3. Видалення елемента з бінарного дерева.
4. Видалення бінарного дерева.
5. Перевірка бінарного дерева на пустоту.
6. Знаходження та видобування даних.
7. Копіювання бінарного дерева.
8. Обхід бінарного дерева за трьома різними маршрутами.
9. Виведення на консоль всіх елементів дерева.

Вказівки

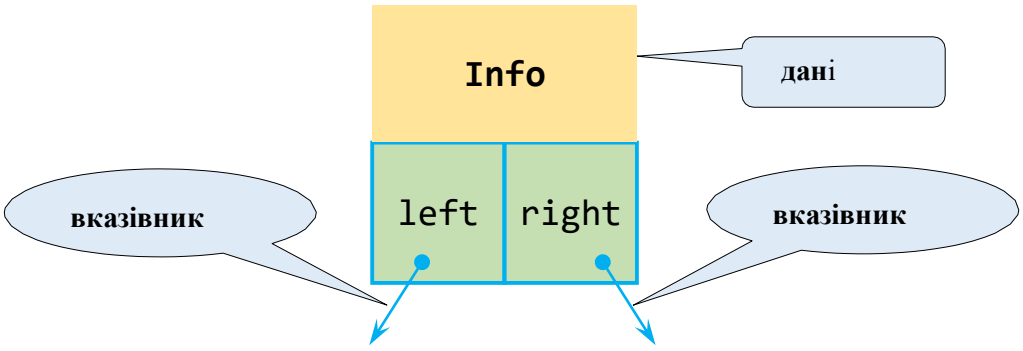
Бинарне дерево — структура даних, що складається з множини вузлів, які об'єднані зв'язками типу **батько->дитина** (**parent->child**). Ці зв'язки задовольняють таким умовам:

1. Існує єдиний елемент (вузол), у якого немає «батька». Цей елемент називається коренем (всього) дерева.
2. Будь-який елемент (вузол), який не є коренем дерева, має рівно одного з батьків.
3. Кожен елемент (вузол) може мати одного, двох або жодного «дитини».

Властивість: Всякий елемент (вузол) є коренем дерева, що складається з його лівого і правого піддерев.

Крім того, елементи дерева певним чином впорядковані. Відношення порядку визначається в залежності від типу елементів, що становлять дерево. Для чисел це відношення більше, менше.

Графічна ілюстрація одного вузла бінарного дерева:



Кожен елемент бінарного дерева містить дані (Info) і два вказівники. Лівий (left) вказівник направлений на «ліву» дитину (child), правий (right) - на «праву». Якщо якогось (або обох) «дітей» немає, то відповідні покажчики рівні NULL.

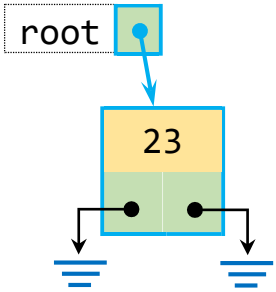
Впорядковування елементів бінарного дерева здійснюється за наступним

1

правилом:

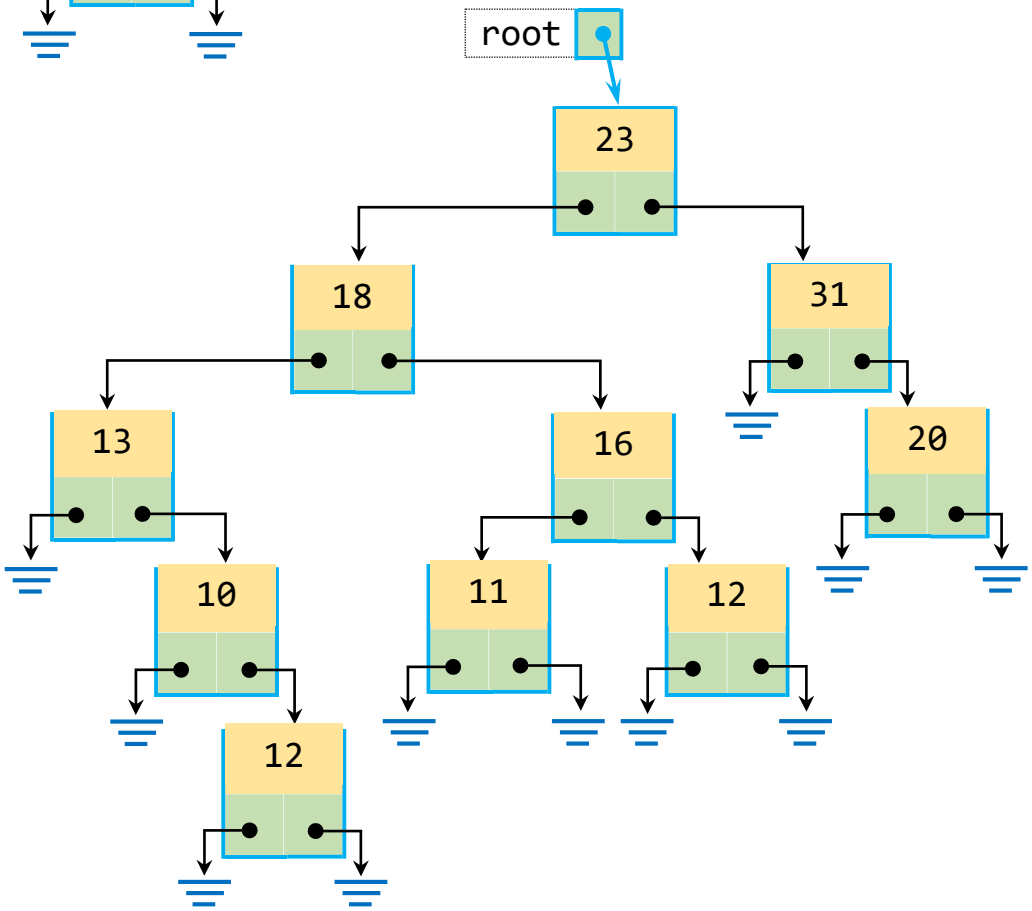
Ліворуч — елемент, **що передує** поточному (в разі чисел - менше), праворуч — **наступний за порядком** (в разі чисел - більше). Відношення порядку встановлюється в залежності від природи самих елементів і (або) логіки додатка.
У лівому піддереві розташовані тільки **попередні** кореневому елементи**, в правому** — **наступні** за порядком.

Графічна ілюстрація бінарних дерев



1. Бінарне дерево, що складається з єдиного елемента - кореня дерева.

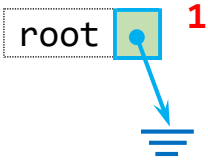
2. Бінарне дерево з лівим і правим піддеревами.



2

СТВОРЕННЯ БІНАРНОГО ДЕРЕВА І ДОДАВАННЯ ЕЛЕМЕНТІВ

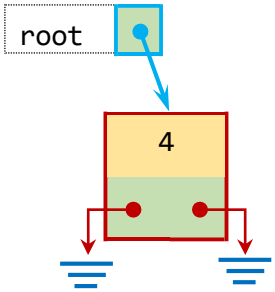
Графічна ілюстрація:



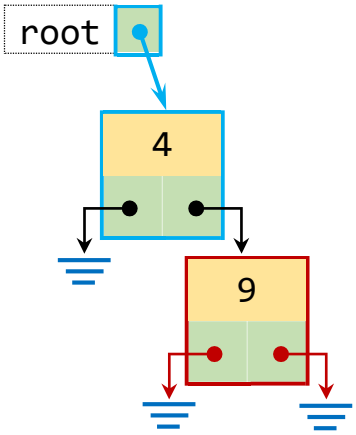
1. Створення бінарного дерева.

2. Додавання елементів (**Insert**)

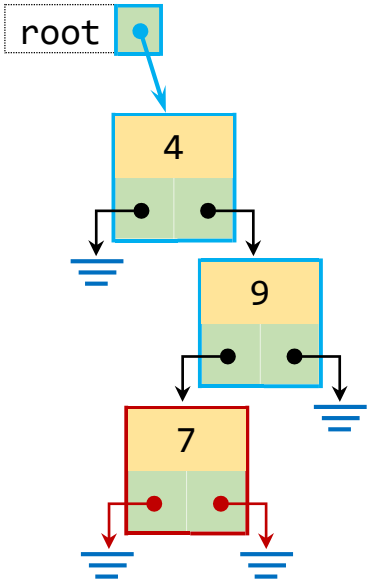
Insert 4:



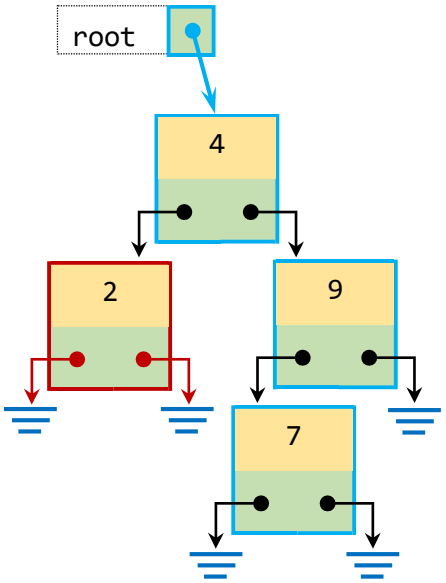
Insert 9(9>4):



Insert 7(7>4, 7<9):

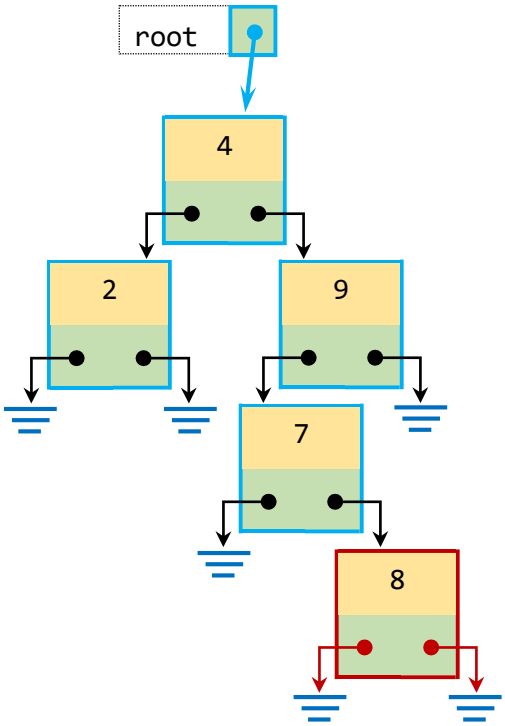


Insert 2(2<4):

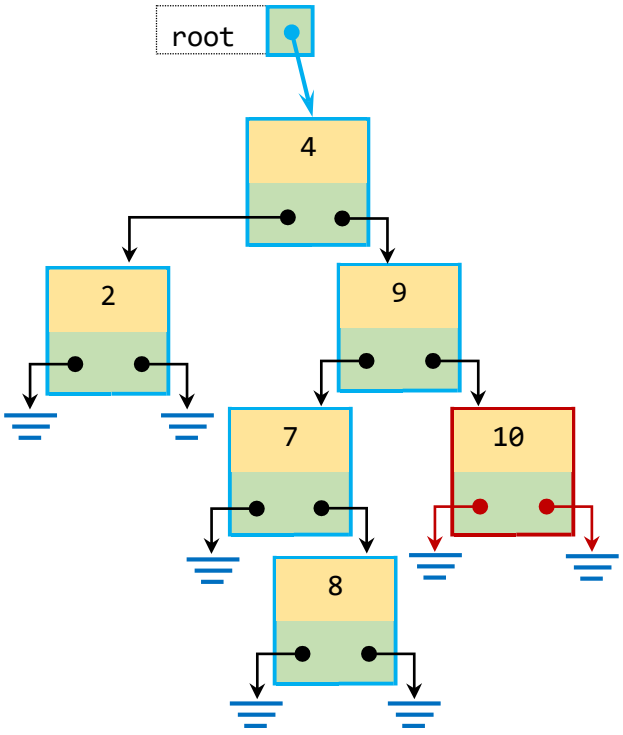


3

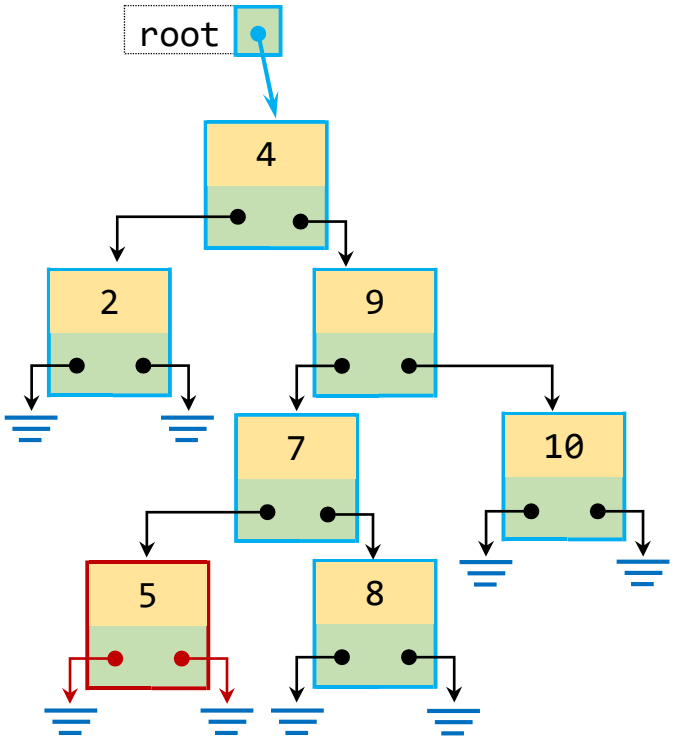
Insert 8(8>4, 8<9, 8>7):



Insert 10(10>4, 10>9):



Insert 5(5>4, 5<9, 5<7):



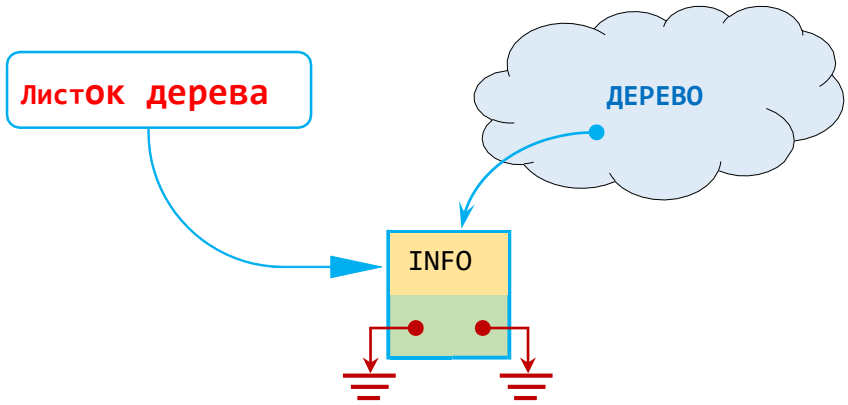
4

Правило додавання елементів в бінарне дерево:

- *Елемент додається на відповідну йому в бінарному дереві позицію* **в якості ЛИСТКА**.
- *Новий елемент* ***заміщає*** *будь-якої вказівник* **NULL** *в існуючому (або порожньому) дереві.*

Листком називається елемент дерева, у якого немає «дітей». Іншими словами, вказівники **left** і **right** такого елемента дорівнюють **NULL**.

Загальний вигляд елемента типу «ЛИСТОК»:



Для додавання елементів в бінарне дерево може бути застосована рекурсія, оскільки щоразу необхідно виконати одні й ті ж дії, аж до основного випадку (base case), коли потрібне місце для додавання елемента знайдено.

Кожен раз потрібно:

1. «Порівняти» елемент, що додається, з поточним.
2. Якщо елемент, що додається, **передує** поточному (*менший*), то перейти до лівої (*вказівник* **`left`**) «дитини» поточного елемента.
3. Якщо елемент, що додається, **наступний за** поточним (*більший*), то перейти до правої (*вказівник* **`right`**) «дитини» поточного елемента.

Основний випадок (*base case*):

Якщо «дитина» *(лева або права)* ***дорівнює*** **NULL**, ***то місце для елемента, що додається, знайдено.***

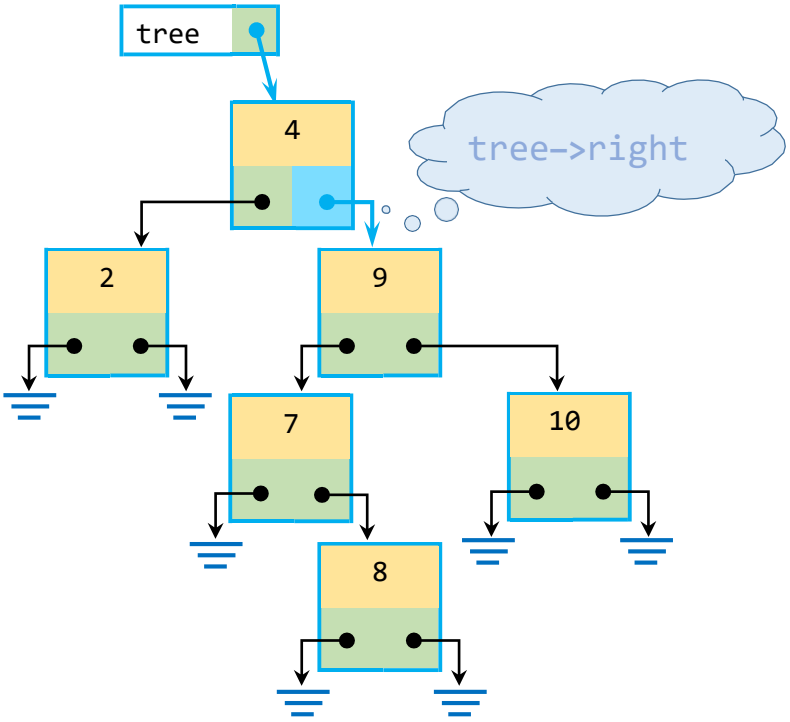
Теперь

1. Створюється новий елемент.
2. Записується інформаційна частина.
3. Вказівники **`left`** і **`right`** встановлюються на **NULL**.

5

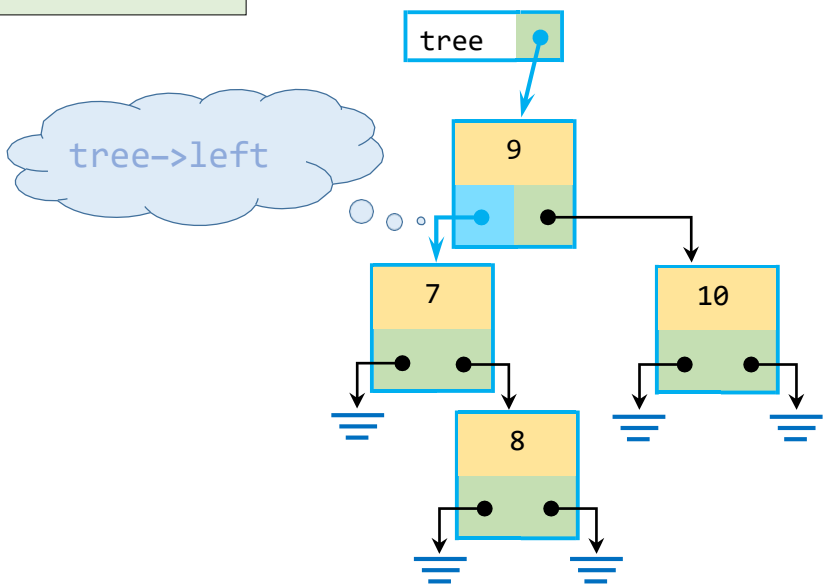
Графічна ілюстрація:

Для визначеності розглянемо випадок `Insert 5(5>4, 5<9, 5<7)`.
Початкове бінарне дерево:



1. Вказівникь `tree` на поточний елемент, значення `tree->info == 4`.
2. Оскільки `5>tree->info` (=4), переходимо для порівняння до кореня правого піддерева (`info=9`):

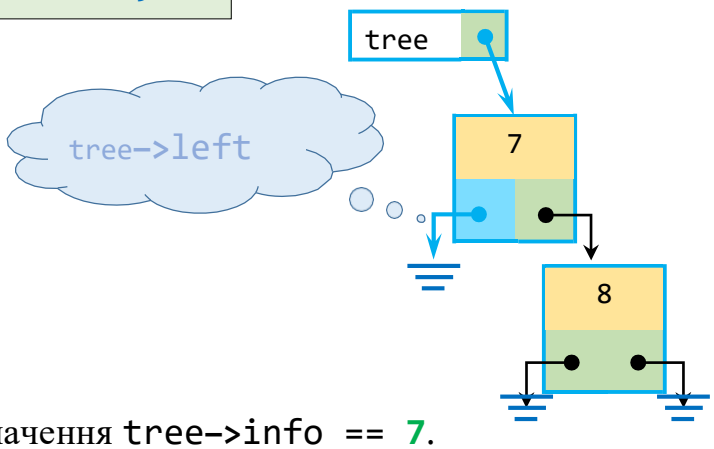
```
tree=tree->left;
```



6

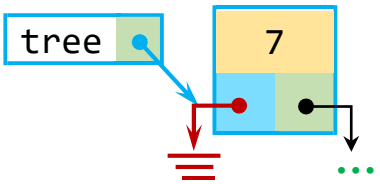
3. Поточне значення `tree->info == 9`.
4. Оскільки `5<tree->info (=9)`, переходимо для порівняння до кореня лівого піддерева (`info=7`):

```
tree=tree ->left;
```



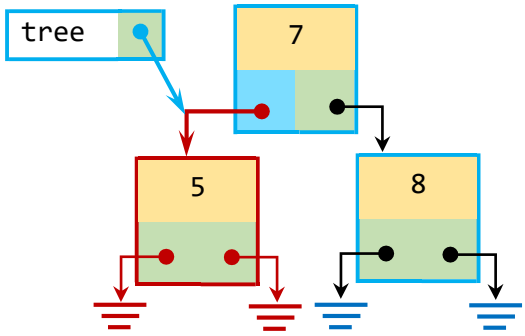
5. Поточне значення `tree->info == 7`.
6. Оскільки `5<tree->info (=7)`, переходимо для порівняння до кореня лівого піддерева, алевказівник `tree->left == NULL`:

```
tree=tree ->left(=NULL);
```



7. Тому настає **основний випадок** (*base case*) – створення і додавання до дерева нового елемента:

```
tree = new Node(5, nullptr, nullptr);
```



7

Приклад **private**-функції, що реалізує рекурсивний алгоритм додавання елементів до бінарного дерева:

```
template<class Type>
void BinaryTree<Type>::insertNode(
         const Type& data, Node<Type>*& tree)
{
if(tree == nullptr )// Основной случай
   tree = new Node<Type>{item, nullptr, nullptr}; else if(data <
tree->info )
   insert(data, tree->left ); else
   insert(data, tree->right );
}
```

Приклад **public**- функції, елемента классу **BinaryTree**:

```
template<class Type>
  void BinaryTree<Type>::insert(const Type& data)
  {
     insert(data,root);
  }
```

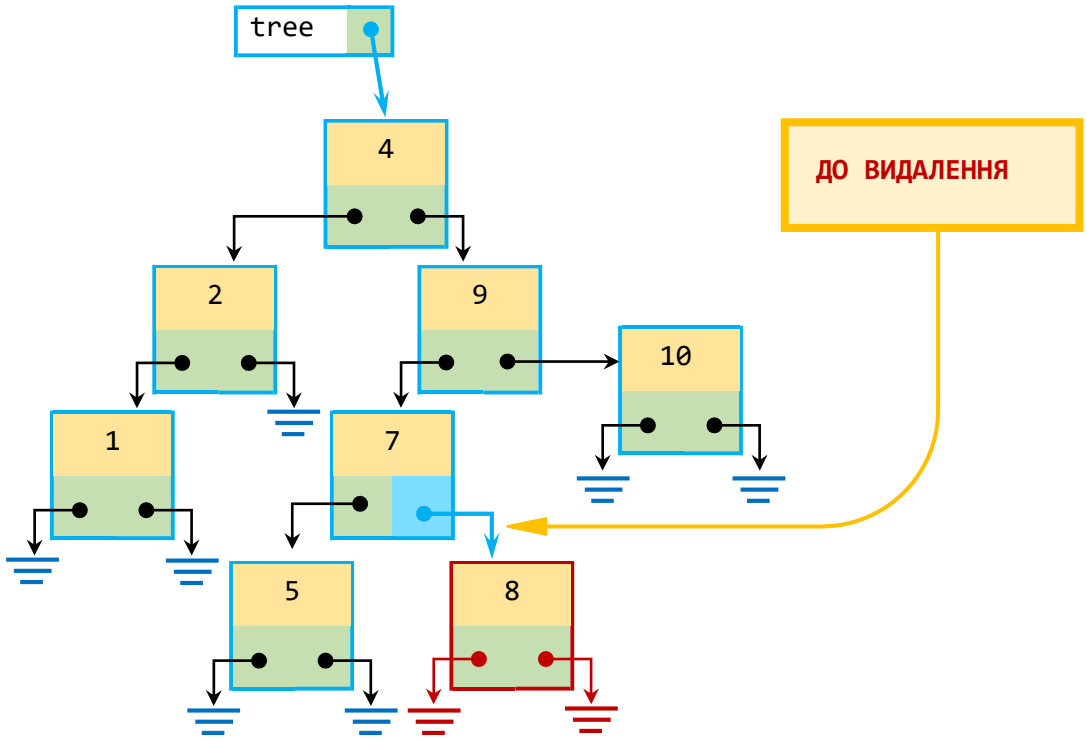
Важливо! Для правильної работи алгоритму, значення **info** елемента, що додається, повинно бути **унікальним**!

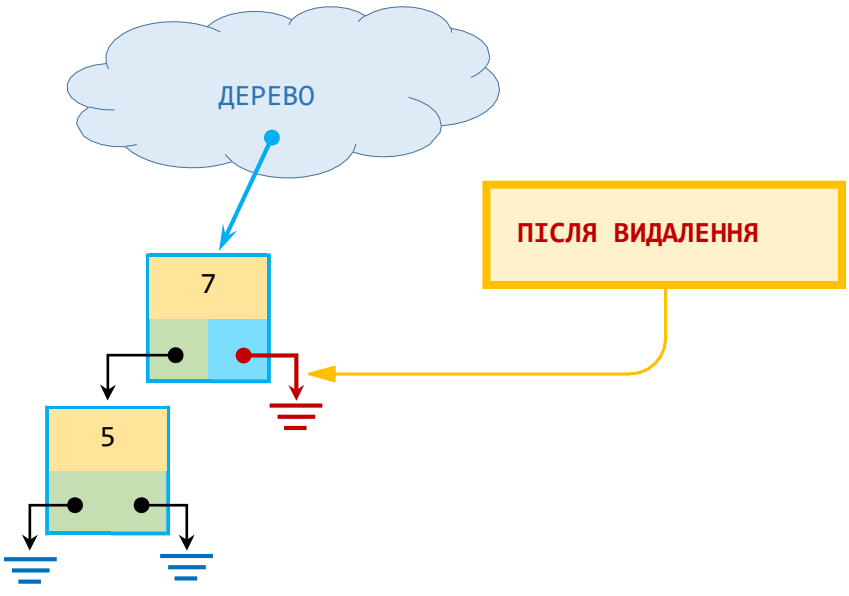
8

ВИДАЛЕННЯ ЕЛЕМЕНТІВ ІЗ БІНАРНОГО ДЕРЕВА

При видаленні елементів з бінарного дерева розглядається 3 (*три*) різних випадки:

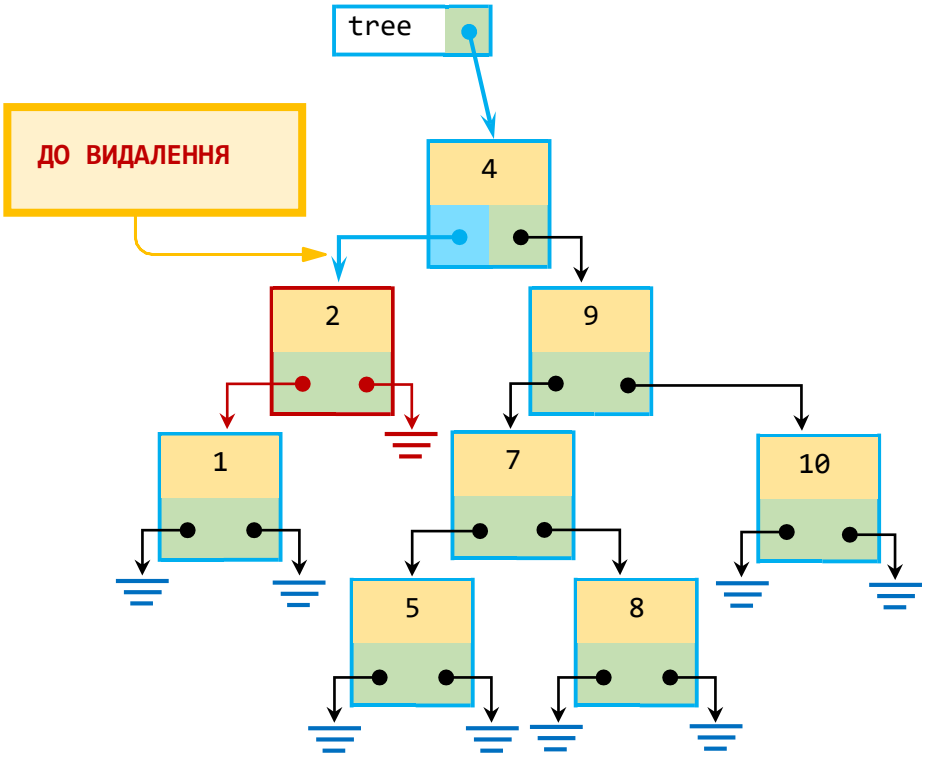
1. **Видаляється «ЛИСТОК». Обидва вказівники `left` і `right` дорівнюють `NULL`.**

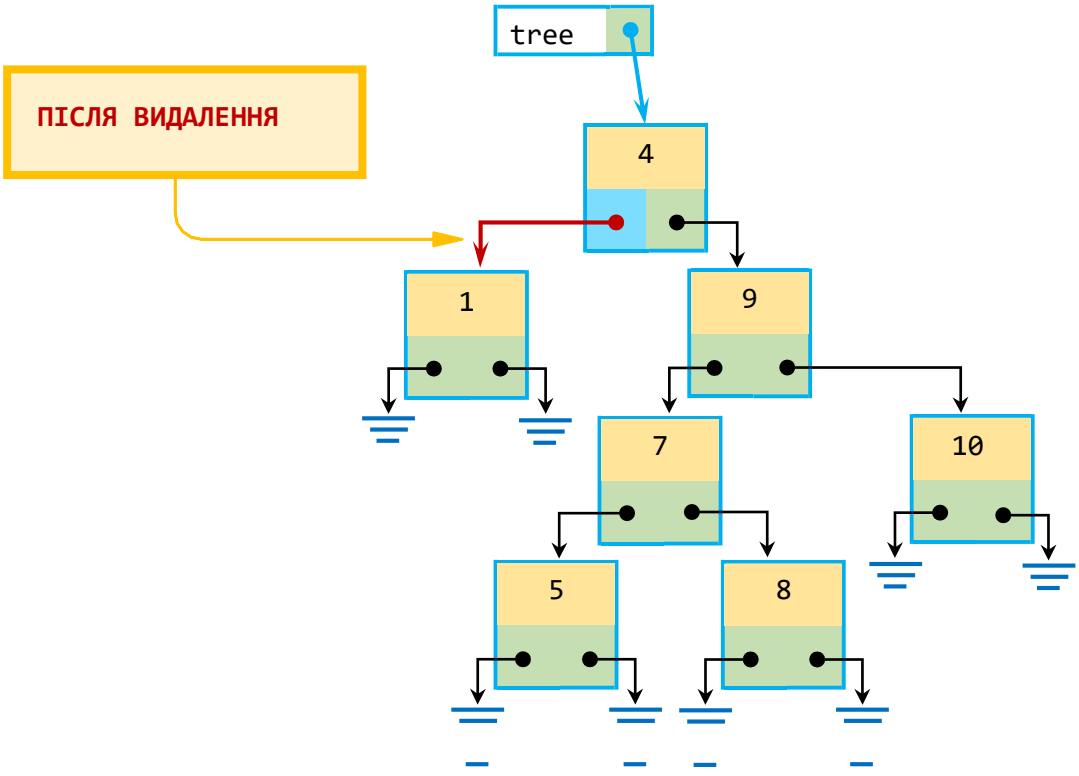




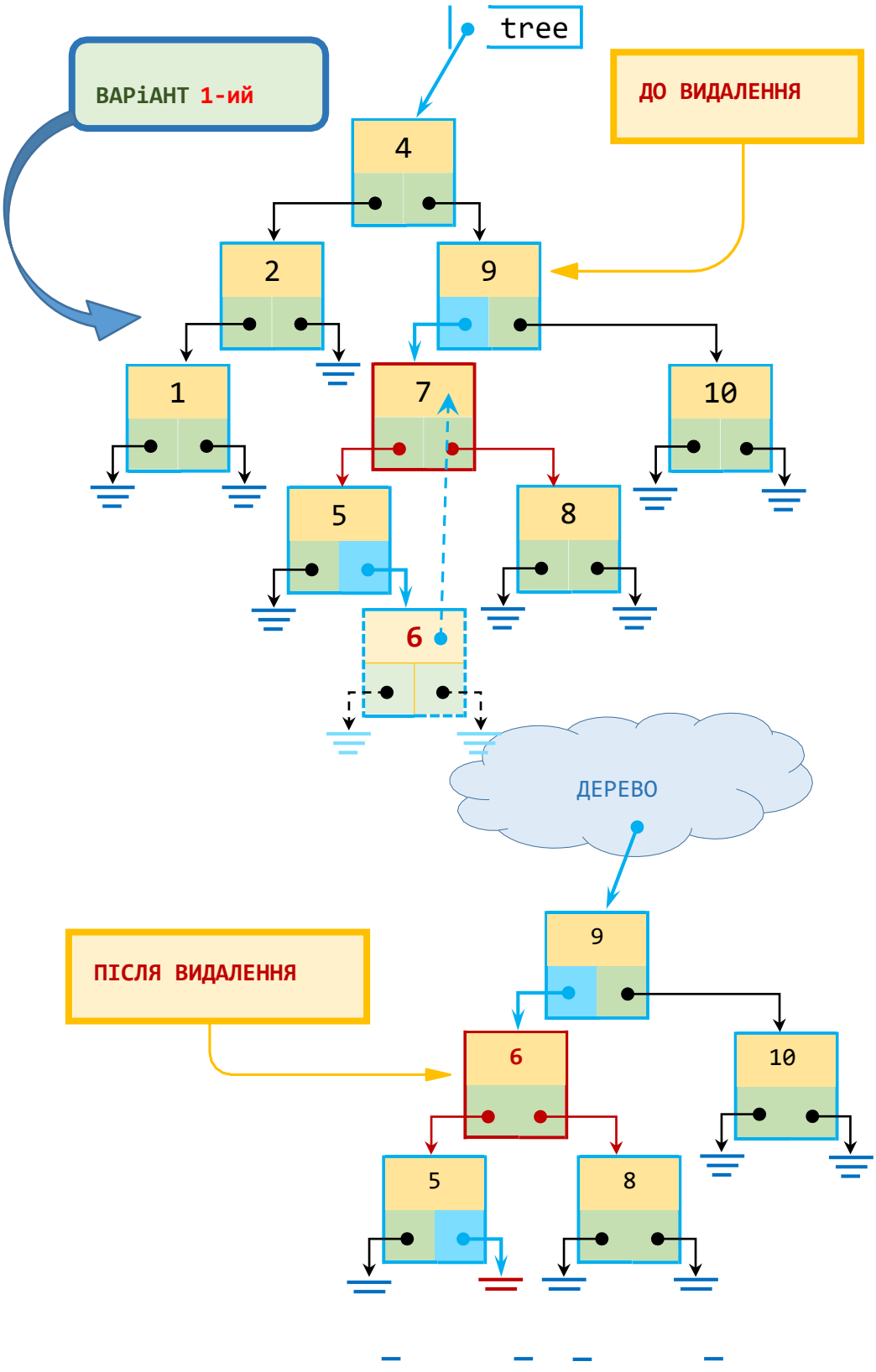
9

2. **Видаляється елемент з ОДНІЄЮ «дитиною». Один із вказівників `left` або `right` елемента дорівнює `NULL`.**

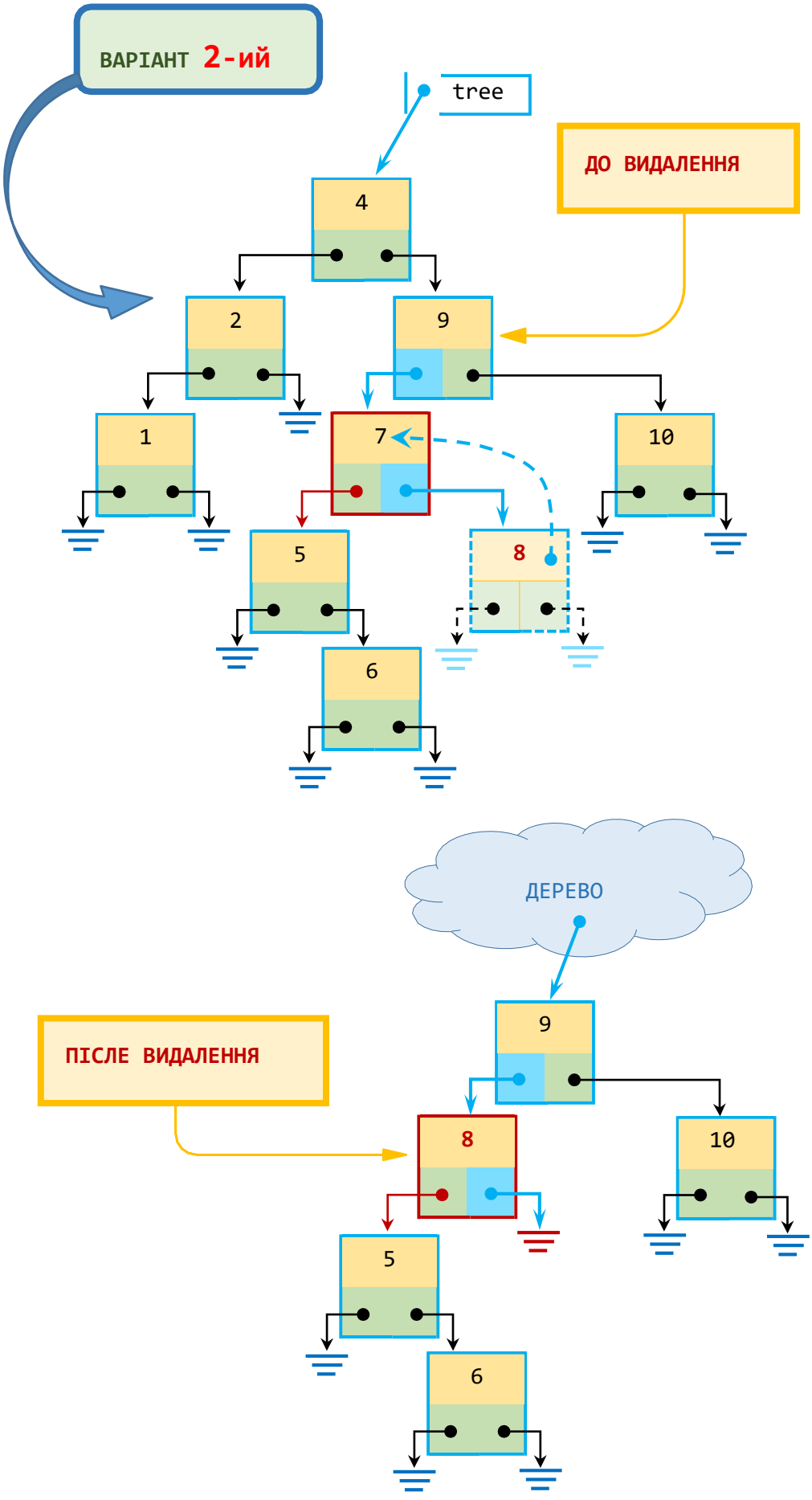




3. **Видаляється елемент з ДВОМА «дітьми». Обидва вказівники `left` і `right` не дорівнюють `NULL`. Можливі 2 варіанти видалення.**



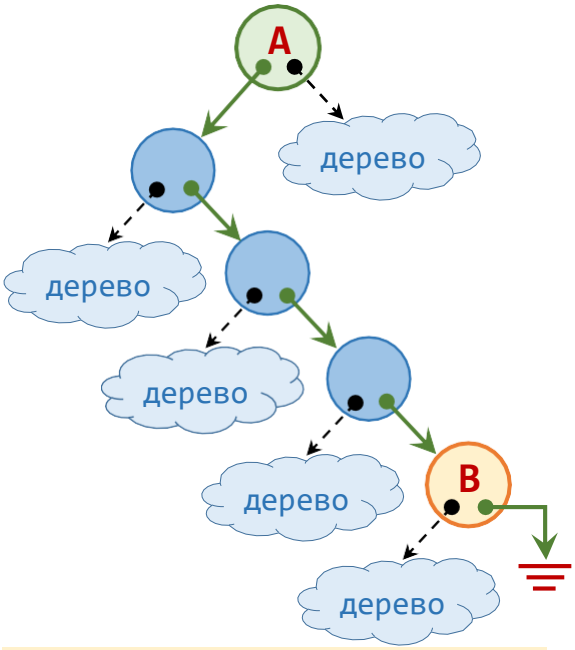
11


ВАРІАНТ 2-ий
tree
ДО ВИДАЛЕННЯ
4
2
9
1
7
10
5
8
6
ДЕРЕВО
ПІСЛЯ ВИДАЛЕННЯ
9
8
10
5
6

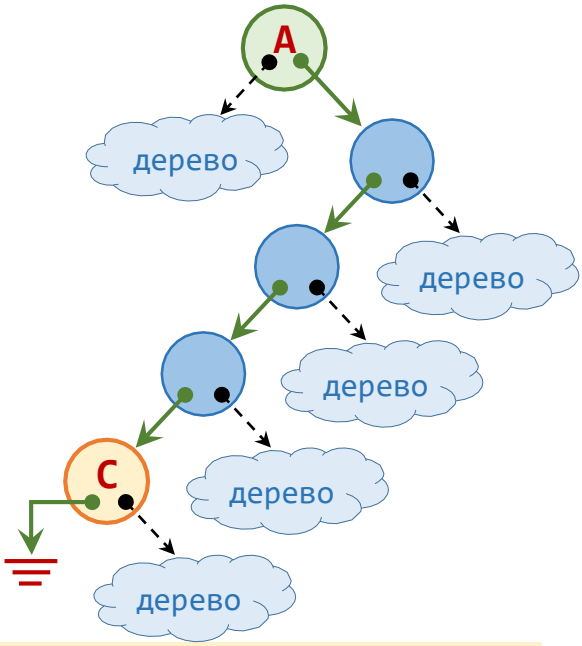
Важливо! При видаленні з дерева елемента, у якого **обидва вказівники** `left` **і** `right` **не дорівнюють** `NULL`, фактичена відбувається **перезапис інформаційної частини** цього елемента, а не його видалення зі структури даних. Видаляється при цьому **інший** елемент. Для цього

1. здійснюється пошук елемента, що **передує або є наступним** відносно елемента, що видаляється, позначимо його ***A***;
2. інформаційна частина елемента ***A*** записується **замість** інформаційної частини елемента, що видаляється;
3. після цього елемент ***A*** видаляється зі структури дерева.

В силу самої структури дерева, елемент, що **передує або є наступним** відносно елемента **ЗАВЖДИ** буде або «листком», або вузлом з однією «дитиною». Тому видалення такого вузла зводиться до випадку **1** або **2**.



Елемент B – **«найбільший»** серед «менших», тобто є таким, що **передує**.



Елемент C – **«найменший»** серед «більших», тобто є таким, що **слідує**.

Правила видалення елементів з дерева у випадках 1 або 2

- ***при видаленні «листка» відповідний вказівник батьківського елемента перенаправляється на будь-який із вказівників елемента, що видаляється*** (`NULL`);

- *при видаленні* ***вузла с одним дочірнім елементом, відповідний вказівник батьківського елемента перенаправляється на той із вказівників вузла, що видаляється, який не дорівнює*** **NULL**.

Фактично, при видаленні вузла зі структури дарева, перенаправляється вказівник батьківського елемента.

Для реалізації зазначеного алгоритму можна застосувати 2 функції:

1. **перша** приймає в якості аргумента вказівник на елемент дерева і «видаляє» його в залежності від кількості дочірніх елементів («*листок», один дочірній елемент, два дочірніх елемента*) відповідним чином;
2. **друга** приймає в якості аргумента дані (*значення поля* **info**), які потрібно видалити, і, якщо знаходить їх в дереві, передає вказівник на відповідний вузол дерева першої функції.

Варіант функції 1, що реалізує видалення даного вузла дерева в залежності від кількості дочірніх елементів.

```
template<class Type>
void deleteNode(Node<Type>*& tree)
{
Node<Type>* temp;

temp = tree;
if(tree->left == nullptr )
     // Один или ни одного дочернего узла
{
    tree = tree->right;
    delete temp;
}
elseif( tree->right == nullptr )
     // Один или ни одного дочернего узла
{
    tree = tree->left;
    delete temp;
}
else // Ровно два дочерних узла
{
           Node<Type>* pred= GetPredecessor(tree);
```

```
        // Находит «предыдущий» элемент
    tree->info = pred->info;
        // Записывает данные в «удаляемый» узел
    deleteNode(pred); //Удаляет «предыдущий» элемент
}
```

Варіант функції 2 (*рекурсія*), яка шукає вузол з даними, які потрібно видалити, і передає вказівник на цей вузол функції 1.

```
template<class Type>
void delete(Node<Type>*& tree, const Type& data)
{
if(data < tree->info ) // Если данные «меньше», чем у текущего
  узла ...
    delete(tree->left, data); // Идём налево
elseif( data > tree->info ) // Если данные «больше», чем у текущего
  узла ...
    delete(tree->right, data); // Идём направо
else // Узел найден, вызываем ф-цию №1
    deleteNode(tree);
}
```

Функція GetPredecessor () повертає вказівник на «попередній» елемент.

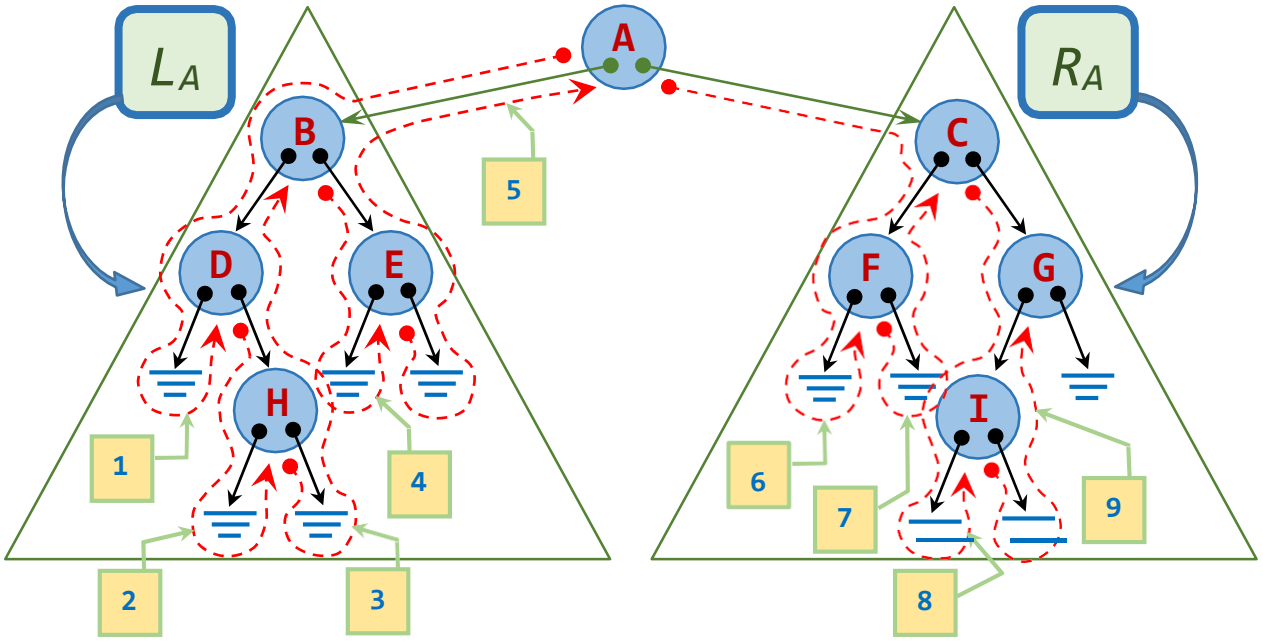
```
template<class Type>
Node<Type>* GetPredecessor(Node<Type>* tree)
{
if(tree == nullptr ) {...} // Проверка...
Node<Type>* curr = tree->left;
        // Идём направо, пока есть узлы...
while(curr->right != nullptr ) curr =
    curr->right;
        return curr; //Получаем указатель
}
```

ОБХІД БІНАРНОГО ДЕРЕВА (*tree traversal*)

Існує три поширені способи обходу або, іншими словами, переміщення між вузлами бінарного дерева.

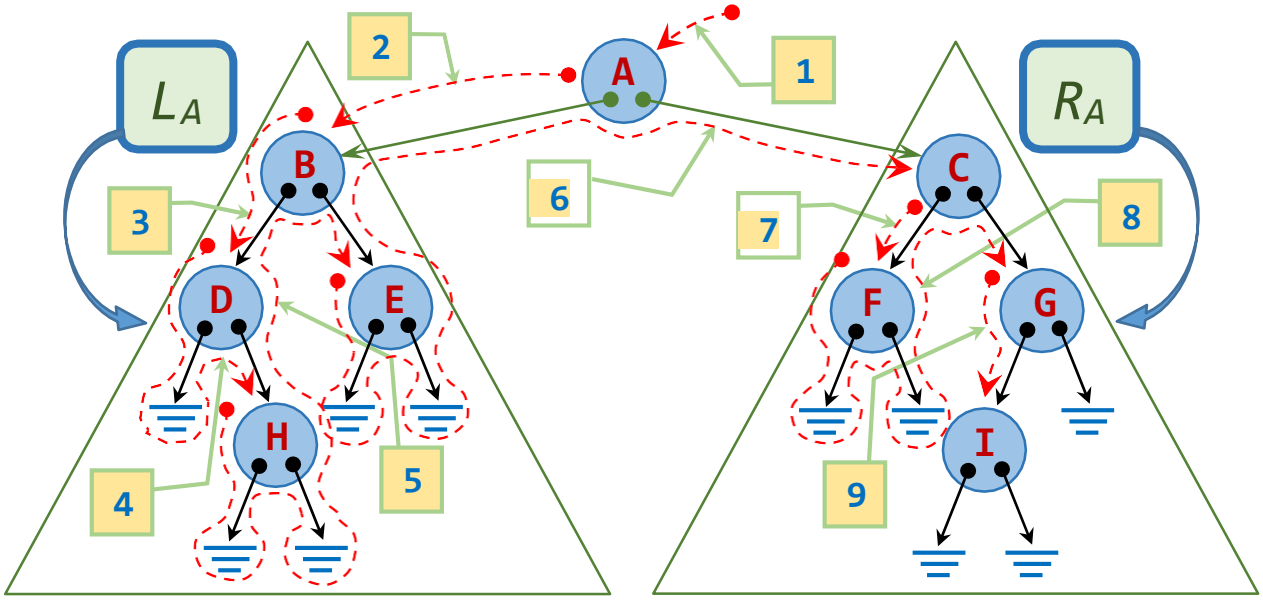
1. **Симетричний обхід** (*inorder traversal*).
   1.1. Обхід лівого піддерева даного вузла.
   1.2. Відвідування даного вузла.
   1.3. Обхід правого піддерева даного вузла.
2. **Обхід у прямому порядку або обхід в ширину** (*preorder traversal*).
   2.1. Відвідування даного вузла.
   2.2. Обхід лівого піддерева даного вузла.
   2.3. Обхід правого піддерева даного вузла.
3. **Обхід у зворотньому порядку або обхід в глубину** (*postorder traversal*).
   3.1. Обхід лівого піддерева даного вузла.
   3.2. Обхід правого піддерева даного вузла.
   3.3. Відвідування даного вузла.

СИМЕТРИЧНИЙ ОБХІД(*inorder traversal*)



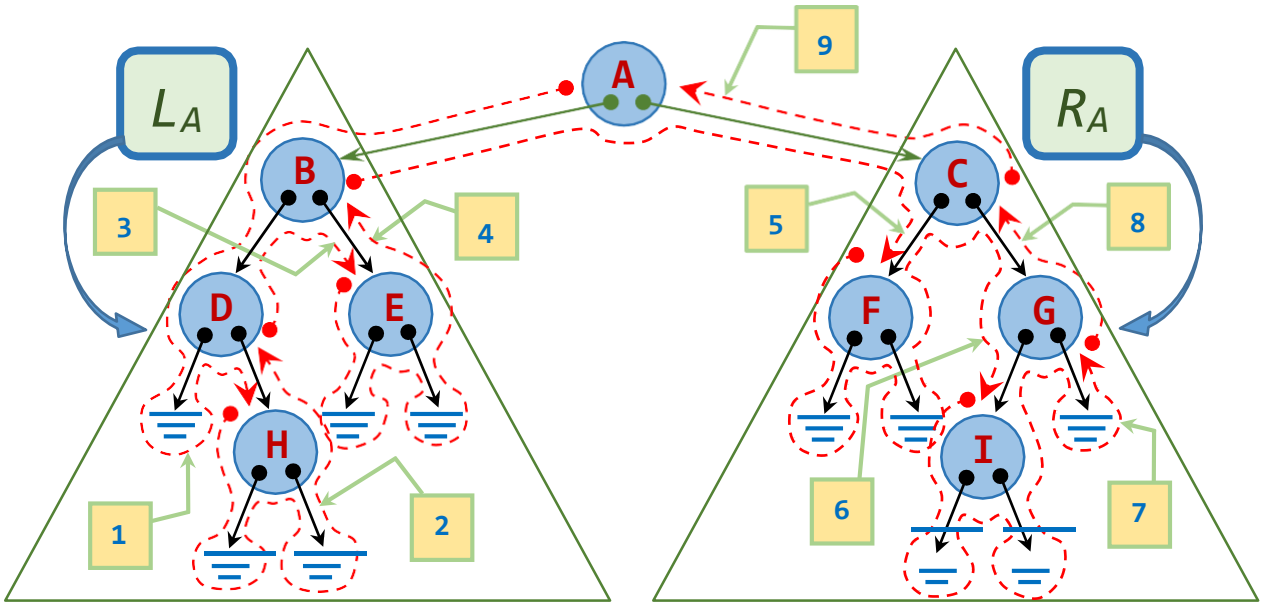
Маршрут симетричного обходу: *DHBEAFCIG*.

ОБХІД В ШИРИНУ(*preorder traversal*)



Маршрут обходу в ширину: *ABDHECFGI*.

ОБХІД В ГЛУБИНУ(*postorder traversal*)



Маршрут обходу в глубину: *HDEBFIGCA*.

При виборі того чи іншого маршруту обходу дерева, інформаційні частини вузлів можуть, наприклад, бути записані в відповідну чергу або стек, які потім і будуть використовуватися для виведення даних.

Рекурсивний алгоритм обходу бінарного дерева

Приклад **private**-функції, що реалізує рекурсивний алгоритм обходу бінарного дерева в **симетричному** (*inorder*) порядку:

```
template<class Type>
void BinaryTree<Type>::inorder(Node<Type>*
                                                    tree) const
{
if(tree != nullptr )//Только если дерево не пусто
   {
     inorder(tree->left );//Рекурсивный вызов
     cout << tree->info << " ";
               //Или добавление в очередь или стэк...
     inorder(tree->right ); //Рекурсивный вызов
   };
}
```

Приклад **public**- функції, що належить класу **BinaryTree**:

```
template<class Type>
void BinaryTree<Type>::inorderTree() const
{
   inorder(root);
}
```

Аналогічно реалізуються інші маршрути обходу бінарного дерева.

Приклад інтерфейсу класів BinaryTree и Node

```
template<class T>
class Node
{
 public:
     // Конструктор по умолчанию.
  Node() { left = right = nullptr;}
     // Конструктор с параметрами.
  Node(const T& d, Node<T>* l = nullptr,
                                   Node<T>* r = nullptr)
  {
    info = d;
    left = l;
    right = r;
  }
  T info;// Информационная часть.
  Node<T> *left, *right;// Указатели на дочерние узлы.
};

template<class T>
class BinaryTree
```

```
{
 public:
  BinaryTree(); // Конструктор по умолчанию. Создаёт Пустое
  дерево.
  ~BinaryTree(); // Деструктор
  const BinaryTree<T>& operator=(const
                              BinaryTree<T>& otherTree);
                  // Перегрузка оператора присваивания.
  BinaryTree(const BinaryTree<T>& otherTree);
  // Копирующий конструктор
  bool  isEmpty()  const;//  Отвечает  на  вопрос  содержит  дерево
  элементы или нет. Возвращает true, если дерево не пусто и false в
  противном случае.
  void print() const;// Выводит данные издерева.
  int nodeCount() const;// Возвращает количество
     узлов в дереве.
  void destroy() {
    clear(root);
  }; // Удаляет все узлы дерева. В результате дерево пусто и
          root == NULL;
  void inorderTree()const{
    inorder(root);
  };// Симметричный обход дерева.
  void preorderTree()const{
    preorder(root);
  }; // Обход дерева в ширину.
  void postorderTree()const{
    postorder(root);
  }; // Обход дерева в глубину.
  T* retriveItem(const T& item) const{ return
    search(root, item);
  };// Возвращает указатель на item, если найдено
  void deleteItem(const T& item){ return
    delete(root, item);
  };
  // Удаляет узел с данными item из дерева.
  void insertItem(const T& item){ return
    insert(root, item);
  }; // Добавляет узел с данными item в дерево.
 private:
  Node<T>*
        copyTree(BinaryTree<T>* otherTree) const;
  // Функция, создающая копию дерева otherTree. Node<T>*
  root;// Указатель на корень дерева. T*
  search(Node<T>*, const T&) const;
  // Ищет и возвращает данные.
  void inorder(Node<T>*) const; // Обход дерева void
```

```
preorder(Node<T>*) const; // Обход дерева void
postorder(Node<T>*) const; // Обход дерева void
clear(Node<T>*&); // Удаляет все узлы дерева
   и устанавливает указатель root в NULL. void
delete(Node<T>*&, const T&);
```

```
    void deleteNode(Node<T>*&);
    void insert(Node<T>*&, const T&);
};
```

Контрольні питання

1. Що таке абстрактна структура даних?
2. Що таке бінарне пошукове дерево?
3. Які основні властивості і способи побудови структури даних бінарне пошукове дерево?
4. Напишіть алгоритм основних операцій для структури даних бінарне пошукове дерево?
5. В якому порядку відвідуються вузли бінарного дерева у випадку:
   - ✓ симетричного обходу – *inorder traversal*;
   - ✓ обходу в прямому порядку (*в ширину*)– *preorder traversal*;
   - ✓ обходу в зворотньому порядку (*в глибину*)– *postorder traversal*.