**CÂY NHỊ PHÂN TÌM KIẾM**

**Định nghĩa:**

Cây nhị phân tìm kiếm là cây nhị phân thỏa mãn các điều kiện sau:

1. Khóa của các đỉnh thuộc cây con trái nhỏ hơn khóa gốc.

2. Khóa của gốc nhỏ hơn khóa các đỉnh thuộc cây con phải.

3. Cây con trái và cây con phải của gốc cũng là cây nhị phân tìm kiếm.

**Đặc điểm:**

1. Có thứ tự
2. Không có phần tử trùng
3. Dễ dàng tạo dữ liệu sắp xếp, và tìm kiếm

**Bài tập:**

Xây dựng những giải thuật trên cây nhị phân thực hiện những thao tác sau đây

* Thêm phần tử (khóa)
* Tìm kiếm phần tử (khóa)
* Sắp xếp
* Duyệt cây
* Xóa phần tử (khóa)
* Quay cây

**Hướng dẫn:**

struct Node{

int key;

Node\* left;

Node\* right;

};

**Hàm tạo node:**

Node \*newNode(int data)

{

Node \*temp = new Node;

temp->key = data;

temp->left = NULL;

temp->right = NULL;

return temp;

}

**Hàm thêm 1 node vào cây**

void addNode(Node\*& root, int data) {

if (root == nullptr) {

root = createNode(data);

return;

}

if (data < root->key) {

addNode(root->left, data); // Recurse on the left subtree

} else if (data > root->key) {

addNode(root->right, data); // Recurse on the right subtree

}

// Note: If data is equal to root->key, you can decide how to handle duplicates.

}

void Visit(Node\* p)

{

cout<<p->key;

}

**Hàm xóa node có x = data**

Node\* deleteNode(Node\* root, int data) {

if (root == nullptr) return root; // Base case: empty tree

if (data < root->key) {

root->left = deleteNode(root->left, data); // Recurse on the left subtree

} else if (data > root->key) {

root->right = deleteNode(root->right, data); // Recurse on the right subtree

} else {

// Node with the key to be deleted found

if (root->left == nullptr) {

Node\* temp = root->right;

delete root;

return temp;

} else if (root->right == nullptr) {

Node\* temp = root->left;

delete root;

return temp;

}

// Node with two children: Get the inorder successor (smallest in the right subtree)

Node\* temp = findMin(root->right);

root->key = temp->key;

root->right = deleteNode(root->right, temp->key);

}

return root;

}

**Hàm duyệt giữa**

void inOrder(Node\* A)

{

if(A!=NULL)

{

inOrder(A->left);

Visit(A);

inOrder(A->right);

}

}

**Hàm duyệt trước**

void preOrder(Node\* A)

{

if(A!=NULL)

{

Visit(A);

preOrder(A->left);

preOrder(A->right);

}

}

**Hàm duyệt sau**

void postOrder(Node\* A)

{

if(A!=NULL)

{

postOrder(A->left);

postOrder(A->right);

Visit(A);

}

}

**Cả bài**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Xây dựng những giải thuật trên cây nhị phân thực hiện những thao tác sau đây

• Thêm phần tử (khóa)

• Tìm kiếm phần tử (khóa)

• Sắp xếp

• Duyệt cây

• Xóa phần tử (khóa)

• Quay cây

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <iostream>

using namespace std;

struct Node{

int key;

Node \*left;

Node \*right;

};

Node \*newNode(int data);//hàm tạo 1 node mới

void addNode(Node \*&root,int data);

void preOrder(Node \*root);

void inOrder(Node \*root);

void postOrder(Node \*root);

void visit(Node \*root);

int main()

{

Node \*root;

addNode(root,8);

addNode(root,1);

addNode(root,9);

addNode(root,7);

addNode(root,5);

addNode(root,6);

addNode(root,4);

addNode(root,3);

addNode(root,2);

addNode(root,0);

cout<<"\npreOder";

preOrder(root); //kết quả = 8-1-0-7-5-4-3-2-6-9

cout<<"\ninOrder";

inOrder(root); //kết quả = 0-1-2-3-4-5-6-7-8-9

cout<<"\npostOrder";

postOrder(root); //kết quả = 0-2-3-4-6-5-7-1-9-8

return 0;

}

Node \*newNode(int data)

{

Node \*temp = new Node;

temp->key = data;

temp->left = NULL;

temp->right = NULL;

return temp;

}

void addNode(Node \*&root,int data)

{

if(root==NULL)

root = newNode(data);

else

{

if(root->key > data)

addNode(root->left,data);

if(root->key < data)

addNode(root->right,data);

}

}

void preOrder(Node \*root)

{

if(root!=NULL)

{

visit(root);

preOrder(root->left);

preOrder(root->right);

}

}

void inOrder(Node \*root)

{

if(root!=NULL)

{

inOrder(root->left);

visit(root);

inOrder(root->right);

}

}

void postOrder(Node \*root)

{

if(root!=NULL)

{

postOrder(root->left);

postOrder(root->right);

visit(root);

}

}

void visit(Node \*root)

{

cout<<" "<<root->key;

}

// Function to find the minimum value node in a BST

Node\* findMin(Node\* root) {

while (root->left != nullptr) {

root = root->left;

}

return root;

}

// Function to delete a node with the given key from the BST

Node\* deleteNode(Node\* root, int data) {

if (root == nullptr) return root; // Base case: empty tree

if (data < root->key) {

root->left = deleteNode(root->left, data); // Recurse on the left subtree

} else if (data > root->key) {

root->right = deleteNode(root->right, data); // Recurse on the right subtree

} else {

// Node with the key to be deleted found

if (root->left == nullptr) {

Node\* temp = root->right;

delete root;

return temp;

} else if (root->right == nullptr) {

Node\* temp = root->left;

delete root;

return temp;

}

// Node with two children: Get the inorder successor (smallest in the right subtree)

Node\* temp = findMin(root->right);

root->key = temp->key;

root->right = deleteNode(root->right, temp->key);

}

return root;

}