HUST

ĐẠI HỌC BÁCH KHOA HÀ NỘI

HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

ONE LOVE. ONE FUTURE.

C PROGRAMMING BASIC

ĐẠI HỌC
BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY
OF SCIENCE AND TECHNOLOGY

C PROGRAMMING BASIC

TREE – PART 2

ONE LOVE. ONE FUTURE.

OUTLINE

- Binary Tree manipulation and traversal (P.04.10.01)
- Check balanced binary tree and compute the height (P.04.10.02)

BINARY TREE MANIPULATION AND TRAVERSAL (P.04.10.01)

- Each node in a binary tree T has a field "id" (unique node identifier). Perform a series of the following operations on tree T (initially, T is an empty tree):
  - **MakeRoot u:** Create a root node with id u.
  - **AddLeft u v:** Create a node with id u and insert it as the left child of the node with id v in T (do not perform the insertion if the node with id u already exists, or the node with id v does not exist, or the node with id v already has a left child).
  - **AddRight u v:** Create a node with id u and insert it as the right child of the node with id v in T (do not perform the insertion if the node with id u already exists, or the node with id v does not exist, or the node with id v already has a right child).
  - **PreOrder:** Output on a new line the sequence of ids of nodes in the pre-order traversal of tree T (elements separated by exactly 1 SPACE character).
  - **InOrder:** Output on a new line the sequence of ids of nodes in the in-order traversal of tree T (elements separated by exactly 1 SPACE character).
  - **PostOrder:** Output on a new line the sequence of ids of nodes in the post-order traversal of tree T (elements separated by exactly 1 SPACE character).

BINARY TREE MANIPULATION AND TRAVERSAL

- **Input:** Each line represents one of the described operations with the format as described above. The end of the input data is marked by a line containing only the character "*".
- **Output:** Write on a single line the result of one of the three operations: InOrder, PreOrder, PostOrder, as described above.

stdin	stdout
MakeRoot 1 AddLeft 2 1 AddRight 3 1 AddLeft 4 3 AddRight 5 2 PreOrder AddLeft 6 5 AddRight 7 5 InOrder *	1 2 5 3 4 2 6 5 7 1 4 3

BINARY TREE MANIPULATION AND TRAVERSAL

- Data structure:

```
typedef struct Node{
    int id;
    struct Node* leftChild;
    struct Node* rightChild;

}Node;
```

- Create a new node with the id =u:

```
Node* makeNode(int u){
    Node* p = (Node*)malloc(sizeof(Node));
    p->leftChild = NULL;
    p->rightChild = NULL;
    p->id = u;
    return p;
}
```

BINARY TREE MANIPULATION AND TRAVERSAL - PSEUDOCODE

- Insert a new node with the identifier (id) u as the left or right child of the node with the identifier (id) v in the tree.

```
void addLeft(int u, int v, Node* r){
     p = find(v, r)
    if p is NULL then
        return
    end if
    if p.leftChild is not NULL then
        return
    end if
    q = find(u, r)
    if q is not NULL then
        return
    end if
    p.leftChild := makeNode(u)
}
```

```
void addRight(int u, int v, Node*
r){
    p = find(v, r)
    if p is NULL then
        return
    end if
    if p.rightChild is not NULL then
        return
    end if
    q = find(u, r)
    if q is not NULL then
        return
   end if
    p.rightChild = makeNode(u)
}
```

```
Node* find(int u, Node* r){
    if r is NULL then
        return NULL
    end if

    if r.id equals u then
        return r
    end if
    p = find(u, r.leftChild)
    if p is not NULL then
        return p
    end if
    return find(u, r.rightChild)
}
```

BINARY TREE MANIPULATION AND TRAVERSAL - PSEUDOCODE

- Perform tree traversal in pre-order, in-order, and post-order.

```
void preOrder(Node* r){
     if r is NULL then
        return
    end if

    print(r.id)
    preOrder(r.leftChild)
    preOrder(r.rightChild)
}
```

```
void inOrder(Node* r){
     if r is NULL then
        return
    end if

    inOrder(r.leftChild)
    print(r.id)
    inOrder(r.rightChild)
}
```

```
void postOrder(Node* r){
    if r is NULL then
        return
    end if

    postOrder(r.leftChild)
    postOrder(r.rightChild)
    print(r.id)
}
```

BINARY TREE MANIPULATION AND TRAVERSAL - PSEUDOCODE

- Insert a new node with the identifier (id) u as the left or right child of the node with the identifier (id) v in the tree.

```
void addLeft(int u, int v, Node* r){
    Node* p = find(v,r);
    if(p == NULL) return;

    if(p->leftChild != NULL) return;
    Node* q = find(u,r);
    if(q != NULL) return;// node having
id = u exists -> do not insert more
    p->leftChild = makeNode(u);
}
```

```
void addRight(int u, int v, Node*
r){
    Node* p = find(v,r);
    if(p == NULL) return;
if(p->rightChild != NULL) return;
    Node* q = find(u,r);
    if(q != NULL) return;// node
having id = u exists -> do not
insert more
    p->rightChild = makeNode(u);
}
```

```
Node* find(int u, Node* r){
    if(r == NULL) return NULL;
    if(r->id == u) return r;
    Node* p = find(u,r->leftChild);
    if(p != NULL) return p;
    return find(u,r->rightChild);
}
```

BINARY TREE MANIPULATION AND TRAVERSAL - PSEUDOCODE

- Perform tree traversal in pre-order, in-order, and post-order.

```
void preOrder(Node* r){
    if(r == NULL) return;
    printf("%d ",r->id);
    preOrder(r->leftChild);
    preOrder(r->rightChild);
}
```

```
void inOrder(Node* r){
    if(r == NULL) return;
    inOrder(r->leftChild);
    printf("%d ",r->id);
    inOrder(r->rightChild);
}
```

```
void postOrder(Node* r){
    if(r == NULL) return;
    postOrder(r->leftChild);
    postOrder(r->rightChild);
    printf("%d ",r->id);
}
```

BINARY TREE MANIPULATION AND TRAVERSAL - PSEUDOCODE

- Main function:

```
int main(){
    char cmd[100];
    Node* root = NULL;
    while(1){
        scanf("%s",cmd);
        if(strcmp(cmd,"*")==0) break;
        else if(strcmp(cmd,"MakeRoot") == 0){
            int u;
            scanf("%d",&u);
            root = makeNode(u);
        }else if(strcmp(cmd,"AddLeft")==0){
            int u,v;
            scanf("%d%d",&u,&v);
            addLeft(u,v,root);
        }
        else if(strcmp(cmd,"AddRight")==0){
            int u,v;
            scanf("%d%d",&u,&v);
            addRight(u,v,root);
        }else if(strcmp(cmd,"InOrder")==0){
            inOrder(root);
            printf("\n");
        }else if(strcmp(cmd,"PreOrder")==0){
            preOrder(root);
            printf("\n");
        }else if(strcmp(cmd,"PostOrder")==0){
            postOrder(root);
            printf("\n");
        }
    }
}
```

CHECK BALANCED BINARY TREE AND COMPUTE THE HEIGHT (B21X09)

- Each node of a binary tree has a field **id** which is the identifier of the node. Build a binary tree and check if the tree is a balanced tree, compute the height of the given tree (the number of nodes of the tree can be upto 50000)
- **Input**
  - Line 1 contains MakeRoot u: make the root of the tree having id = u
  - Each subsequent line contains: AddLeft or AddRightcommands with the format
  - AddLeft u v: create a node having id = u, add this node as a left-child of the node with id = v (if not exists)
  - AddRight u v: create a node having id = u, add this node as a right-child of the node with id = v (if not exists)
  - The last line contains * which marks the end of the input
- **Output**
- Write two integers z and h (separated by a SPACE character) in which h is the height (the number of nodes of the longest path from the root to a leaf) and z = 1 if the tree is balanced and z = 0, otherwise

CHECK BALANCED BINARY TREE AND COMPUTE THE HEIGHT

- Example: input and output

stdin	stdout
MakeRoot 1 AddLeft 2 1 AddRight 3 1 AddLeft 9 2 AddRight 4 2 AddLeft 6 3 AddRight 5 3 AddLeft 7 4 AddRight 8 4 *	1 4

CHECK BALANCED BINARY TREE AND COMPUTE THE HEIGHT

- Data structure:

```
#define N 1000001
typedef struct TNode{
    int id;
    struct TNode* left;
    struct TNode*right;
}Node;
Node* root;
//An array containing nodes of the tree where the indices
//of the array correspond to the ids of the nodes.
Node* nodes[N];
```

```
// Structure holding information about the
//tree
typedef struct TINFO{
    int balanced;
    int hl; // Depth of the left child
    int hr; // Depth of the right child
    int h;  // Depth of the tree
}INFO;
```

CHECK BALANCED BINARY TREE AND COMPUTE THE HEIGHT

- Initialize a node with key (or id).

```
Node* makeNode(int id){
    Node* p = (Node*)malloc(sizeof(Node));
    p->id = id;
    p->left = NULL;
    p->right = NULL;
    return p;
}
```

CHECK BALANCED BINARY TREE AND COMPUTE THE HEIGHT

- Insert a new node with the key (id) u as the left or right child of the node with the key (id) v in the tree.

```
int addLeft(int u, int v){
     if nodes[u] is not NULL or nodes[v] is NULL then
         return 0
    end if

    if nodes[v].left is not NULL then
         return 0
    end if

    p = makeNode(u)
    nodes[v].left = p
    nodes[u] = p
    return 1
}
```

```
int addRight(int u, int v){
     if nodes[u] is not NULL or nodes[v] is NULL
then
         return 0
    end if

    if nodes[v].right is not NULL then
         return 0
    end if

    p = makeNode(u)
    nodes[v].right := p
    nodes[u] = p
    return 1
}
```

CHECK BALANCED BINARY TREE AND COMPUTE THE HEIGHT

- The visit function is designed to traverse a binary tree and return a structure INFO containing two pieces of information: the height of the tree rooted at the current node and whether that subtree is balanced or not (balanced is represented by balanced = 1, and 0 otherwise).

```
INFO visit(Node * r){
if r is NULL then
        INFO i
        i.balanced := 1
        i.h := 0
        return i
    end if
    i1 := visit(r.left)
    i2 := visit(r.right)
    INFO i
    i.h := max(i1.h, i2.h) + 1
    if i1.balanced = 0 then
        i.balanced := 0
        return i
    end if
    if i2.balanced = 0 then
        i.balanced := 0
        return i
    end if

    if abs(i1.h - i2.h) >= 2 then
        i.balanced := 0
    else
        i.balanced := 1
    end if

    return i;
}
```

CHECK BALANCED BINARY TREE AND COMPUTE THE HEIGHT

- Insert a new node with the key (id) u as the left or right child of the node with the key (id) v in the tree.

```
int addLeft(int u, int v){// add a new node id = u as a
left child of the node id = v (if not exists)


    if(nodes[u] != NULL || nodes[v] == NULL) return 0;
    if(nodes[v]->left != NULL) return 0;
    Node* p = makeNode(u);
    nodes[v]->left = p;
    nodes[u] = p;
    return 1;
}
```

```
int addRight(int u, int v){


    if(nodes[u] != NULL || nodes[v] == NULL)
           return 0;
    if(nodes[v]->right != NULL) return 0;
    Node* p = makeNode(u);
    nodes[v]->right = p;
    nodes[u] = p;
    return 1;
}
```

CHECK BALANCED BINARY TREE AND COMPUTE THE HEIGHT

- The visit function is designed to traverse a binary tree and return a structure INFO containing two pieces of information: the height of the tree rooted at the current node and whether that subtree is balanced or not (balanced is represented by balanced = 1, and 0 otherwise).

```
INFO visit(Node * r){
    if(r == NULL){
        INFO i;
        i.balanced = 1;
        i.h = 0;
        return i;
    }
    INFO i1 = visit(r->left);
    INFO i2 = visit(r->right);
    INFO i;
    i.h = (i1.h > i2.h ? i1.h : i2.h) + 1;

    if(i1.balanced == 0){
        i.balanced = 0; return i;
    }
    if(i2.balanced == 0){
        i.balanced = 0; return i;
    }
    if(abs(i1.h - i2.h) >= 2){
        i.balanced = 0;
    }else
        i.balanced = 1;

    return i;
}
```

CHECK BALANCED BINARY TREE AND COMPUTE THE HEIGHT

- Some functions:

```
void solve(){
    for(int i = 0; i < N; i++) nodes[i] = NULL;
    while(1){
        char s[20];
        scanf("%s",s);
        if(strcmp(s,"*")==0) break;
        if(strcmp(s,"MakeRoot")==0){
            int u; scanf("%d",&u);
            root = makeNode(u);
            nodes[u] = root;
        }
```

```
    else if(strcmp(s,"AddLeft")==0){
            int u,v;
            scanf("%d%d",&u,&v);
            addLeft(u,v);
        }else if(strcmp(s,"AddRight")==0){
            int u,v;
            scanf("%d%d",&u,&v);
            addRight(u,v);
        }
    }
    INFO i = visit(root);
    printf("%d %d",i.balanced,i.h);
}
```

HUST

hust.edu.vn fb.com/dhbkhn

THANK YOU !