**综合设计报告**

**课程名称：**  数据结构

**题 目：**  树的建立和遍历

**指导教师：**  蔡英

**设计起始日期：** 2013.xx.xx-2013.xx.xx

**学 院：**  计算机学院

**系 别：** 计算机科学与技术系

**学生姓名：**  刘鸿喆

**班级/学号：**  计科1201/2012011107

**成 绩：**

**一、需求分析**

树的生成与构造

实验内容描述：

生成一棵以二叉链表存储的二叉树bt（不少于15个结点）。

分别用递归和非递归方法前序遍历bt，并以缩格形式打印bt上各结点的信息。

编写算法，交换bt上所有结点的左、右子树，并以缩格形式打印出交换前后的bt结点信息。

**二、概要设计**

**1、数据结构**

二叉链表。

**2、使用算法**

树的递归/非递归，前、中、后序遍历，左右子数交换，缩格打印。

**三、详细设计**

**1、数据结构详细设计**

栈：

非递归节点栈：std::stack<Node \*> nodeStack;

后序非递归标识栈：std::stack<typename Position::Tag> tagStack;

节点数组：std::vector<Node \*> nodeVector;

**2、算法**

递归：

前序：

1. 如果无根，返回上一层（或方法外）
2. 访问节点
3. 对于左子树，重复1~4
4. 对于右子树，重复1~4

中序：

1. 如果无根，返回上一层（或方法外）
2. 对于左子树，重复1~4
3. 访问节点
4. 对于右子树，重复1~4

后序：

1. 如果无根，返回上一层（或方法外）
2. 对于左子树，重复1~4
3. 对于右子树，重复1~4
4. 访问节点

非递归：

前序：

1. 栈推入空指针
2. 栈推入根节点
3. 取栈顶，若栈顶为空，结束，若不为空，重复下列操作
   1. 记录并访问栈顶，弹出栈顶
   2. 若有右子树，栈推入右子树树根
   3. 若有左子树，栈推入左子树树根

中序：

1. 栈推入空指针
2. 栈推入根节点
3. 取栈顶，若栈顶为空，结束，若不为空，重复下列操作
   1. 当节点有左子树，重复下列操作
      1. 转到左子树树根
      2. 栈推入左子树树根
   2. 取节点为栈顶，若不为空，重复下列操作
      1. 记录并访问节点
      2. 弹出栈顶
      3. 如果节点无右子树，跳出
   3. 如果当前节点不为空，将当前节点右子树推入栈中

后序：

1. 节点栈推入空指针
2. 节点栈推入根节点
3. 标识栈推入左标识
4. 节点取栈顶，若栈顶为空，结束，若不为空，重复
   1. 检查并弹出节点栈栈顶，若为
      1. 左
         1. 节点栈推入右标识
         2. 若有左子树
            1. 节点栈推入左子树树根
            2. 标识栈推入左标识
      2. 右
         1. 节点栈推入访问标识
         2. 若有右子树
            1. 节点栈推入右子树树根
            2. 标识栈推入左标识
      3. 访问
         1. 节点栈弹出栈顶
         2. 标识栈弹出栈顶
         3. 访问节点

左右子树交换：

1. 如果无根，返回上一层（或方法外）
2. 对于左子树，重复1~4
3. 对于右子树，重复1~4
4. 交换左右子树树根

**四、调试分析**

**1、调试过程中遇到的问题**

无。

**2、算法的时空分析**（n=树节点数）

时间：O(n)

空间：O(n)

**五、使用说明和测试结果**

**1、使用说明**

无。

**2、测试结果**

输出

A

B C

D E F G

H I J K L M N O

P Q R S T U V W X Y Z [ \ ] ^ \_

` a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~ 

Recursive Pre Order: ABDHP`aQbcIRdeSfgEJThiUjkKVlmWnoCFLXpqYrsMZtu[vwGN\xy]z{O^|}\_~

Recursive In Order: `PaHbQcDdReIfSgBhTiJjUkElVmKnWoApXqLrYsFtZuMv[wCx\yNz]{G|^}O~\_

Recursive Post Order: `aPbcQHdeRfgSIDhiTjkUJlmVnoWKEBpqXrsYLtuZvw[MFxy\z{]N|}^~\_OGCA

Non-Recursive Pre Order: ABDHP`aQbcIRdeSfgEJThiUjkKVlmWnoCFLXpqYrsMZtu[vwGN\xy]z{O^|}\_~

Non-Recursive In Order: `PaHbQcDdReIfSgBhTiJjUkElVmKnWoApXqLrYsFtZuMv[wCx\yNz]{G|^}O~\_

Non-Recursive Post Order: `aPbcQHdeRfgSIDhiTjkUJlmVnoWKEBpqXrsYLtuZvw[MFxy\z{]N|}^~\_OGCA

A

C B

G F E D

O N M L K J I H

\_ ^ ] \ [ Z Y X W V U T S R Q P

 ~ } | { z y x w v u t s r q p o n m l k j i h g f e d c b a `

Recursive Pre Order: ACGO\_~^}|N]{z\yxFM[wvZutLYsrXqpBEKWonVmlJUkjTihDISgfRedHQcbPa`

Recursive In Order: \_~O}^|G{]zNy\xCw[vMuZtFsYrLqXpAoWnKmVlEkUjJiThBgSfIeRdDcQbHaP`

Recursive Post Order: ~\_}|^O{z]yx\NGwv[utZMsrYqpXLFConWmlVKkjUihTJEgfSedRIcbQa`PHDBA

Non-Recursive Pre Order: ACGO\_~^}|N]{z\yxFM[wvZutLYsrXqpBEKWonVmlJUkjTihDISgfRedHQcbPa`

Non-Recursive In Order: \_~O}^|G{]zNy\xCw[vMuZtFsYrLqXpAoWnKmVlEkUjJiThBgSfIeRdDcQbHaP`

Non-Recursive Post Order: ~\_}|^O{z]yx\NGwv[utZMsrYqpXLFConWmlVKkjUihTJEgfSedRIcbQa`PHDBA

**六、心得体会**

在深度不大时非递归会显得既麻烦，又费时间空间，深度大的时候，非递归才会显现出比较大的价值。

**七、附录**

**1、程序**

//

// main.cpp

// ds-ex3-tree

//

// Created by Darren Liu on 13-11-27.

// Copyright (c) 2013年 Darren Liu. All rights reserved.

//

#include <iostream>

#include "include/binary-tree.h"

void print(char &data);

int main(int argc, const char \*argv[])

{

BinaryTree<char> tree;

BinaryTree<char>::Position root = BinaryTree<char>::Position::Root();

for (int i = 0; i < 63; i++, root.goNext()) {

tree.addData(root, 'A' + i);

}

// tree.addData(root.left().right().right().right().right().right(), '\*');

// tree.addData(root, 'A');

// tree.addData(root.left(), 'B');

// tree.addData(root.left().right(), 'C');

// tree.addData(root.left().right().right(), 'D');

// tree.addData(root.left().right().right().right(), 'E');

// tree.addData(root.left().right().right().right().right(), 'F');

// tree.addData(root.left().right().right().right().right().left(), 'G');

std::cout << tree;

std::cout << " Recursive Pre Order: ";

tree.traverse(BinaryTree<char>::PreOrder, BinaryTree<char>::Recursive, print);

std::cout << ENDLINE;

std::cout << " Recursive In Order: ";

tree.traverse(BinaryTree<char>::InOrder, BinaryTree<char>::Recursive, print);

std::cout << ENDLINE;

std::cout << " Recursive Post Order: ";

tree.traverse(BinaryTree<char>::PostOrder, BinaryTree<char>::Recursive, print);

std::cout << ENDLINE;

std::cout << "Non-Recursive Pre Order: ";

tree.traverse(BinaryTree<char>::PreOrder, BinaryTree<char>::NonRecursive, print);

std::cout << ENDLINE;

std::cout << "Non-Recursive In Order: ";

tree.traverse(BinaryTree<char>::InOrder, BinaryTree<char>::NonRecursive, print);

std::cout << ENDLINE;

std::cout << "Non-Recursive Post Order: ";

tree.traverse(BinaryTree<char>::PostOrder, BinaryTree<char>::NonRecursive, print);

std::cout << ENDLINE;

tree.swapPosition();

std::cout << tree;

std::cout << " Recursive Pre Order: ";

tree.traverse(BinaryTree<char>::PreOrder, BinaryTree<char>::Recursive, print);

std::cout << ENDLINE;

std::cout << " Recursive In Order: ";

tree.traverse(BinaryTree<char>::InOrder, BinaryTree<char>::Recursive, print);

std::cout << ENDLINE;

std::cout << " Recursive Post Order: ";

tree.traverse(BinaryTree<char>::PostOrder, BinaryTree<char>::Recursive, print);

std::cout << ENDLINE;

std::cout << "Non-Recursive Pre Order: ";

tree.traverse(BinaryTree<char>::PreOrder, BinaryTree<char>::NonRecursive, print);

std::cout << ENDLINE;

std::cout << "Non-Recursive In Order: ";

tree.traverse(BinaryTree<char>::InOrder, BinaryTree<char>::NonRecursive, print);

std::cout << ENDLINE;

std::cout << "Non-Recursive Post Order: ";

tree.traverse(BinaryTree<char>::PostOrder, BinaryTree<char>::NonRecursive, print);

std::cout << ENDLINE;

return 0;

}

void print(char &data) {

std::cout << data;

}

//

// binary\_tree.h

// ds-ex3-tree

//

// Created by Darren Liu on 13-11-27.

// Copyright (c) 2013 Darren Liu. All rights reserved.

//

#ifndef ds\_ex3\_tree\_binary\_tree\_h

#define ds\_ex3\_tree\_binary\_tree\_h

#include <typeinfo>

#include <iostream>

#include <string>

#include <stack>

#include <cmath>

#include <vector>

#include <cstdlib>

#include <cstdio>

#if defined(\_\_MINGW32\_\_)

# define ENDLINE '\n';

#else

# define ENDLINE std::endl;

#endif

template <typename \_Ty>

class BinaryTree {

public:

BinaryTree() : \_root(NULL) {}

class Position {

public:

Position() : \_pos(1) {}

Position(unsigned long \_pos) : \_pos(\_pos) {}

static Position Root() {

return Position();

}

Position goLeft() {

\_pos = \_pos << 1;

return \*this;

}

Position goRight() {

\_pos = \_pos << 1;

\_pos += 1;

return \*this;

}

Position goParent() {

if (\_pos == 1) {

return \*this;

}

\_pos = \_pos >> 1;

return \*this;

}

Position goPrevious() {

if (\_pos == 1) {

return \*this;

}

\_pos--;

return \*this;

}

Position goNext() {

\_pos++;

return \*this;

}

Position returnRoot() {

\_pos = 1;

return \*this;

}

Position left() {

return Position(\_pos << 1);

}

Position right() {

return Position((\_pos << 1) + 1);

}

Position parent() {

if (\_pos == 1) {

return Root();

}

return Position(\_pos >> 1);

}

Position next() {

return Position(\_pos + 1);

}

Position previous() {

if (\_pos == 1) {

return Position(\_pos);

}

return Position(\_pos - 1);

}

enum Tag {

Self = 0,

Left = 1,

Right = 2

};

std::stack<Tag> getStack() const {

std::stack<Tag> tagStack;

for (unsigned long t = \_pos; t != 1; t /= 2) {

Tag tag = (t % 2) ? Right : Left;

tagStack.push(tag);

}

return tagStack;

}

private:

unsigned long \_pos;

};

bool addData(const Position &pos, const \_Ty &data) {

std::stack<typename Position::Tag> tagStack = pos.getStack();

if (!\_root) {

if (tagStack.empty()) {

\_root = new Node(data);

return true;

}

return false;

} else if (tagStack.empty()) {

return false;

}

Node \*p = \_root;

for (; tagStack.size() > 1; tagStack.pop()) {

if (!p) {

return false;

}

switch (tagStack.top()) {

case Position::Left:

p = p -> left;

break;

case Position::Right:

p = p -> right;

break;

default:

break;

}

}

if (!p) {

return false;

}

switch (tagStack.top()) {

case Position::Left:

if (p -> left) {

return false;

}

p -> left = new Node(data);

break;

case Position::Right:

if (p -> right) {

return false;

}

p -> right = new Node(data);

break;

default:

return false;

}

return true;

}

public:

class Node {

public:

Node() : left(NULL), right(NULL) {}

Node(const \_Ty &data) : left(NULL), right(NULL), data(data) {}

Node \*left;

Node \*right;

\_Ty data;

};

static unsigned long \_getDepth(const Node \*root) {

if (!root) {

return 0;

}

unsigned long leftDepth = \_getDepth(root -> left);

unsigned long rightDepth = \_getDepth(root -> right);

return ((leftDepth > rightDepth) ? leftDepth : rightDepth) + 1;

}

public:

enum TraversalSequence {

PreOrder,

InOrder,

PostOrder

};

enum TraversalType {

Recursive,

NonRecursive

};

void traverse(TraversalSequence sequence, TraversalType type, void visit(\_Ty &)) {

switch (type) {

case Recursive:

switch (sequence) {

case PreOrder:

preOrderRecursiveTraverse(visit, \_root);

break;

case InOrder:

inOrderRecursiveTraverse(visit, \_root);

break;

case PostOrder:

postOrderRecursiveTraverse(visit, \_root);

break;

default:

break;

}

break;

case NonRecursive:

switch (sequence) {

case PreOrder:

preOrderNonRecursiveTraverse(visit, \_root);

break;

case InOrder:

inOrderNonRecursiveTraverse(visit, \_root);

break;

case PostOrder:

postOrderNonRecursiveTraverse(visit, \_root);

break;

default:

break;

}

default:

break;

}

}

public:

void swapPosition() {

recursiveSwapPosition(\_root);

}

private:

static void preOrderRecursiveTraverse(void visit(\_Ty &), Node \*root = NULL) {

if (!root) {

return;

}

visit(root -> data);

preOrderRecursiveTraverse(visit, root -> left);

preOrderRecursiveTraverse(visit, root -> right);

}

static void inOrderRecursiveTraverse(void visit(\_Ty &), Node \*root = NULL) {

if (!root) {

return;

}

inOrderRecursiveTraverse(visit, root -> left);

visit(root -> data);

inOrderRecursiveTraverse(visit, root -> right);

}

static void postOrderRecursiveTraverse(void visit(\_Ty &), Node \*root = NULL) {

if (!root) {

return;

}

postOrderRecursiveTraverse(visit, root -> left);

postOrderRecursiveTraverse(visit, root -> right);

visit(root -> data);

}

static void preOrderNonRecursiveTraverse(void visit(\_Ty &), Node \*root = NULL) {

std::stack<Node \*> nodeStack;

nodeStack.push(NULL);

nodeStack.push(root);

while (Node \*node = nodeStack.top()) {

nodeStack.pop();

visit(node -> data);

if (node -> right) {

nodeStack.push(node -> right);

}

if (node -> left) {

nodeStack.push(node -> left);

}

}

}

static void inOrderNonRecursiveTraverse(void visit(\_Ty &), Node \*root = NULL) {

std::stack<Node \*> nodeStack;

nodeStack.push(NULL);

nodeStack.push(root);

while (Node \*node = nodeStack.top()) {

while (node -> left) {

node = node -> left;

nodeStack.push(node);

}

while ((node = nodeStack.top())) {

visit(node -> data);

nodeStack.pop();

if (node -> right) {

break;

}

}

if (node) {

nodeStack.push(node -> right);

}

}

}

static void postOrderNonRecursiveTraverse(void visit(\_Ty &), Node \*root = NULL) {

std::stack<Node \*> nodeStack;

std::stack<typename Position::Tag> tagStack;

nodeStack.push(NULL);

nodeStack.push(root);

tagStack.push(Position::Left);

while (Node \*node = nodeStack.top()) {

switch (tagStack.top()) {

case Position::Left:

tagStack.pop();

tagStack.push(Position::Right);

if (node -> left) {

nodeStack.push(node -> left);

tagStack.push(Position::Left);

}

break;

case Position::Right:

tagStack.pop();

tagStack.push(Position::Self);

if (node -> right) {

nodeStack.push(node -> right);

tagStack.push(Position::Left);

}

break;

case Position::Self:

nodeStack.pop();

tagStack.pop();

visit(node -> data);

break;

default:

break;

}

}

}

void recursiveSwapPosition(Node \*root) {

if (!root) {

return;

}

recursiveSwapPosition(root -> left);

recursiveSwapPosition(root -> right);

Node \*t = root -> left;

root -> left = root -> right;

root -> right = t;

}

public:

std::vector<Node \*> getNodeVector() const {

std::vector<Node \*> nodeVector;

unsigned long depth = getDepth();

if (!depth) {

return nodeVector;

}

nodeVector.push\_back(\_root);

for (unsigned long i = 0; i < depth - 1; i++) {

for (unsigned long j = (unsigned long)pow(2, i) - 1; j < (unsigned long)pow(2, i + 1) - 1; j++) {

Node \*p = nodeVector[j];

if (!p) {

nodeVector.push\_back(NULL);

nodeVector.push\_back(NULL);

continue;

}

if (p -> left) {

nodeVector.push\_back(p -> left);

} else {

nodeVector.push\_back(NULL);

}

if (p -> right) {

nodeVector.push\_back(p -> right);

} else {

nodeVector.push\_back(NULL);

}

}

}

return nodeVector;

}

public:

unsigned long getDepth() const {

return \_getDepth(\_root);

}

friend std::ostream &operator<<(std::ostream &cout, const BinaryTree<\_Ty> &tree) {

unsigned long depth = tree.getDepth();

std::vector<Node \*> nodeVector = tree.getNodeVector();

typename std::vector<Node \*>::iterator p;

unsigned long i = 0;

unsigned int length = (unsigned int)pow(2, depth - 1) - 1;

for (i = 0; i < length; i++) {

std::cout << ' ';

}

for (i = 0, p = nodeVector.begin(); p != nodeVector.end(); p++, i++) {

int layer = 0;

bool last = true;

long long t = i;

for (long long j = 0; t > 0; j++, t -= (unsigned long)pow(2, j), layer++);

if (t != 0) {

last = false;

}

if (\*p) {

std::cout << (\*p) -> data;

}

else {

std::cout << ' ';

}

if (last) {

std::cout << ENDLINE;

length = (unsigned int)pow(2, depth - layer - 2);

if (length > 0) {

length--;

}

for (unsigned int j = 0; j < length; j++) {

std::cout << ' ';

}

}

else {

length = (unsigned int)pow(2, depth - layer);

if (length > 0) {

length--;

}

for (unsigned int j = 0; j < length; j++) {

std::cout << ' ';

}

}

}

return cout;

}

private:

Node \*\_root;

};

#endif