**信息科学与工程学院**

**2020－2021学年第一学期**

实 验 报 告

课程名称： 数据结构与数据库技术

实验名称： 实验三

专 业 班 级 通信工程 二班

学 生 学 号 201800121050

学 生 姓 名 孟麟芝

实 验 时 间 2020年10月8日

实验报告

## 【实验目的】

1.掌握指针变量、动态变量的含义。

2.掌握二叉树的结构特征，以及各种存储结构的特点及适用范围。

3.掌握指针类型描述、访问和处理二叉树的运算。

## 【实验要求】

实现二叉树表的层次遍历算法，并给出应用。

## 【实验代码】

1.主程序main.c代码

#include <stdio.h>

#include "fatal.h"

#include "tree.h"

#include "queue.h"

int main()

{

    SearchTree root, nodes[5];

    root = (SearchTree)malloc(sizeof(struct TreeNode));

    for (int i = 0; i < 5; i++)

    {

        nodes[i] = (SearchTree)malloc(sizeof(struct TreeNode));

    }

    //下面建立的树满足二叉查找树的规则

    root->Element = 6;

    nodes[0]->Element = 2;

    nodes[1]->Element = 8;

    nodes[2]->Element = 1;

    nodes[3]->Element = 4;

    nodes[4]->Element = 3;

    //下面将结点连接起来

    root->Left = nodes[0];

    root->Right = nodes[1];

    nodes[0]->Left = nodes[2];

    nodes[0]->Right = nodes[3];

    nodes[3]->Left = nodes[4];

    nodes[1]->Left = nodes[1]->Right = NULL;

    nodes[2]->Left = nodes[2]->Right = NULL;

    nodes[3]->Right = NULL;

    nodes[4]->Left = nodes[4]->Right = NULL;

    //下面对树进行层次遍历并输出显示

traverseTheTree(root, 6);

system("pause");

}

2.库文件queue.c代码

#include "queue.h"

#include "fatal.h"

#include <stdlib.h>

#include "tree.h"

#define MinQueueSize (5)

struct QueueRecord

{

    int Capacity;

    int Front;

    int Rear;

    int Size;

    ElementType \*Array;

};

/\* START: fig3\_58.txt \*/

int IsEmpty(Queue Q)

{

    return Q->Size == 0;

}

/\* END \*/

int IsFull(Queue Q)

{

    return Q->Size == Q->Capacity;

}

Queue CreateQueue(int MaxElements)

{

    Queue Q;

    /\* 1\*/ if (MaxElements < MinQueueSize)

        /\* 2\*/ Error("Queue size is too small");

    /\* 3\*/ Q = malloc(sizeof(struct QueueRecord));

    /\* 4\*/ if (Q == NULL)

        /\* 5\*/ FatalError("Out of space!!!");

    /\* 6\*/ Q->Array = malloc(sizeof(ElementType) \* MaxElements);

    /\* 7\*/ if (Q->Array == NULL)

        /\* 8\*/ FatalError("Out of space!!!");

    /\* 9\*/ Q->Capacity = MaxElements;

    /\*10\*/ MakeEmptyQueue(Q);

    /\*11\*/ return Q;

}

/\* START: fig3\_59.txt \*/

void MakeEmptyQueue(Queue Q)

{

    Q->Size = 0;

    Q->Front = 1;

    Q->Rear = 0;

}

/\* END \*/

void DisposeQueue(Queue Q)

{

    if (Q != NULL)

    {

        free(Q->Array);

        free(Q);

    }

}

/\* START: fig3\_60.txt \*/

static int Succ(int Value, Queue Q)

{

    if (++Value == Q->Capacity)

        Value = 0;

    return Value;

}

void Enqueue(ElementType X, Queue Q)

{

    if (IsFull(Q))

        Error("Full queue");

    else

    {

        Q->Size++;

        Q->Rear = Succ(Q->Rear, Q);

        Q->Array[Q->Rear] = X;

    }

}

/\* END \*/

ElementType Front(Queue Q)

{

    if (!IsEmpty(Q))

        return Q->Array[Q->Front];

    Error("Empty queue");

    return 0; /\* Return value used to avoid warning \*/

}

void Dequeue(Queue Q)

{

    if (IsEmpty(Q))

        Error("Empty queue");

    else

    {

        Q->Size--;

        Q->Front = Succ(Q->Front, Q);

    }

}

ElementType FrontAndDequeue(Queue Q)

{

    ElementType X = 0;

    if (IsEmpty(Q))

        Error("Empty queue");

    else

    {

        Q->Size--;

        X = Q->Array[Q->Front];

        Q->Front = Succ(Q->Front, Q);

    }

    return X;

}

3.库文件tree.c代码

#include "tree.h"

#include <stdlib.h>

#include "fatal.h"

struct TreeNode

{

    ElementType Element;

    SearchTree Left;

    SearchTree Right;

};

/\* START: fig4\_17.txt \*/

SearchTree MakeEmptyTree(SearchTree T)

{

    if (T != NULL)

    {

        MakeEmptyTree(T->Left);

        MakeEmptyTree(T->Right);

        free(T);

    }

    return NULL;

}

/\* END \*/

/\* START: fig4\_18.txt \*/

Position Find(ElementType X, SearchTree T)

{

    if (T == NULL)

        return NULL;

    if (X < T->Element)

        return Find(X, T->Left);

    else if (X > T->Element)

        return Find(X, T->Right);

    else

        return T;

}

/\* END \*/

/\* START: fig4\_19.txt \*/

Position FindMin(SearchTree T)

{

    if (T == NULL)

        return NULL;

    else if (T->Left == NULL)

        return T;

    else

        return FindMin(T->Left);

}

/\* END \*/

/\* START: fig4\_20.txt \*/

Position FindMax(SearchTree T)

{

    if (T != NULL)

        while (T->Right != NULL)

            T = T->Right;

    return T;

}

/\* END \*/

/\* START: fig4\_22.txt \*/

SearchTree Insert(ElementType X, SearchTree T)

{

    /\* 1\*/ if (T == NULL)

    {

        /\* Create and return a one-node tree \*/

        /\* 2\*/ T = malloc(sizeof(struct TreeNode));

        /\* 3\*/ if (T == NULL)

            /\* 4\*/ FatalError("Out of space!!!");

        else

        {

            /\* 5\*/ T->Element = X;

            /\* 6\*/ T->Left = T->Right = NULL;

        }

    }

    else

        /\* 7\*/ if (X < T->Element)

        /\* 8\*/ T->Left = Insert(X, T->Left);

    else

        /\* 9\*/ if (X > T->Element)

        /\*10\*/ T->Right = Insert(X, T->Right);

    /\* Else X is in the tree already; we'll do nothing \*/

    /\*11\*/ return T; /\* Do not forget this line!! \*/

}

/\* END \*/

/\* START: fig4\_25.txt \*/

SearchTree Delete(ElementType X, SearchTree T)

{

    Position TmpCell;

    if (T == NULL)

        Error("Element not found");

    else if (X < T->Element) /\* Go left \*/

        T->Left = Delete(X, T->Left);

    else if (X > T->Element) /\* Go right \*/

        T->Right = Delete(X, T->Right);

    else                         /\* Found element to be deleted \*/

        if (T->Left && T->Right) /\* Two children \*/

    {

        /\* Replace with smallest in right subtree \*/

        TmpCell = FindMin(T->Right);

        T->Element = TmpCell->Element;

        T->Right = Delete(T->Element, T->Right);

    }

    else /\* One or zero children \*/

    {

        TmpCell = T;

        if (T->Left == NULL) /\* Also handles 0 children \*/

            T = T->Right;

        else if (T->Right == NULL)

            T = T->Left;

        free(TmpCell);

    }

    return T;

}

/\* END \*/

ElementType Retrieve(Position P)

{

    return P->Element;

}

void traverseTheTree(SearchTree root, int nodeCount)

{

    Queue myQueue;

    myQueue = CreateQueue(nodeCount);

    if (root == NULL)

    {

        //若树为空，直接返回

        return;

    }

    //先将根节点入队

    Enqueue((int)root, myQueue);

    while (!IsEmpty(myQueue))

    {

        SearchTree front;

        //下面的实现在指针型与int之间发生了相互转换，其实是有风险的，但在本实验中并没有体现出影响

        //出队一个节点，查看其子节点

        front = (SearchTree)FrontAndDequeue(myQueue);

        printf("%d ", front->Element);

        //将出队结点的左子节点入队

        if (front->Left)

            Enqueue((int)(front->Left), myQueue);

        //将出队结点的右子节点入队

        if (front->Right)

            Enqueue((int)(front->Right), myQueue);

    }

}

4.头文件queue.h代码

typedef int ElementType;

/\* START: fig3\_57.txt \*/

#ifndef \_Queue\_h

#define \_Queue\_h

struct QueueRecord;

typedef struct QueueRecord \*Queue;

int IsEmpty(Queue Q);

int IsFull(Queue Q);

Queue CreateQueue(int MaxElements);

void DisposeQueue(Queue Q);

void MakeEmptyQueue(Queue Q);

void Enqueue(ElementType X, Queue Q);

ElementType Front(Queue Q);

void Dequeue(Queue Q);

ElementType FrontAndDequeue(Queue Q);

#endif /\* \_Queue\_h \*/

/\* END \*/

5.头文件tree.h代码

typedef int ElementType;

/\* START: fig4\_16.txt \*/

#ifndef \_Tree\_H

#define \_Tree\_H

struct TreeNode;

typedef struct TreeNode \*Position;

typedef struct TreeNode \*SearchTree;

SearchTree MakeEmptyTree(SearchTree T);

Position Find(ElementType X, SearchTree T);

Position FindMin(SearchTree T);

Position FindMax(SearchTree T);

SearchTree Insert(ElementType X, SearchTree T);

SearchTree Delete(ElementType X, SearchTree T);

ElementType Retrieve(Position P);

void traverseTheTree(SearchTree root, int nodeCount);

#endif /\* \_Tree\_H \*/

/\* END \*/

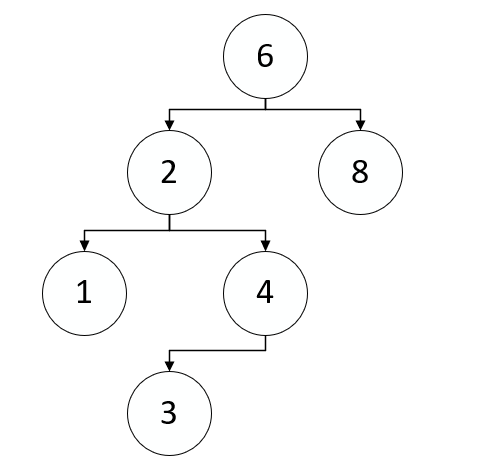
## 【实验记录与说明分析】

本实验中用到了上一次实验中的队列，所以调用了queue.c这一库函数，其中主要用到了CreateQueue()、Enqueue()、FrontAndDequeue()这三个函数。

另外，我在tree.c库函数中添加了traverseTheTree()这一函数用来实现遍历，其他的函数其实并不需要用到，核心的部分在于这个结构体的使用：

1. **struct** TreeNode
2. {
3. ElementType Element;
4. SearchTree Left;
5. SearchTree Right;
6. };

其实创建一棵树是一个很复杂的过程，很难用程序去自动化的处理，所以只能利用这个结构体手动创建一棵树，这棵树是一个二叉搜索树：



下面是创建这棵树的代码。

1. SearchTree root, nodes[5];
2. root = (SearchTree)malloc(**sizeof**(**struct** TreeNode));
3. **for** (**int** i = 0; i < 5; i++)
4. {
5. nodes[i] = (SearchTree)malloc(**sizeof**(**struct** TreeNode));
6. }
7. //下面建立的树满足二叉查找树的规则
8. root->Element = 6;
9. nodes[0]->Element = 2;
10. nodes[1]->Element = 8;
11. nodes[2]->Element = 1;
12. nodes[3]->Element = 4;
13. nodes[4]->Element = 3;
14. //下面将结点连接起来
15. root->Left = nodes[0];
16. root->Right = nodes[1];
17. nodes[0]->Left = nodes[2];
18. nodes[0]->Right = nodes[3];
19. nodes[3]->Left = nodes[4];
20. nodes[1]->Left = nodes[1]->Right = NULL;
21. nodes[2]->Left = nodes[2]->Right = NULL;
22. nodes[3]->Right = NULL;
23. nodes[4]->Left = nodes[4]->Right = NULL;

接下来对遍历函数做一些说明，首先，它接受的参数是二叉树的根节点的指针，还有二叉树节点数目，首先要对树是否为空进行判断，空时不执行下面的程序，不然容易导致错误。

先将一个节点入队，为了能复用前面写的队列的程序，入队时对指针型变量做了强制类型转换，变为int型，而这其实是不太合适的，因为二者长度不同，但本实验中并没有产生问题，因为本实验中的指针型变量长度都比较短。

将队列中的首个元素出队并显示，然后查看它有无左子节点，有的话则要将其指针入队，再查看有无右子节点，有的话也要将其指针入队。

循环往复的操作，即可实现对树的遍历。

该方法对队列长度是有要求的，最坏的情况下，若树共有N层，有个节点，则队列的长度至少为，才能保证能实现功能。

1. **void** traverseTheTree(SearchTree root, **int** nodeCount)
2. {
3. Queue myQueue;
4. myQueue = CreateQueue(nodeCount);
5. **if** (root == NULL)
6. {
7. //若树为空，直接返回
8. **return**;
9. }
10. //先将根节点入队
11. Enqueue((**int**)root, myQueue);
12. **while** (!IsEmpty(myQueue))
13. {
14. SearchTree front;
15. //下面的实现在指针型与int之间发生了相互转换，其实是有风险的，但在本实验中并没有体现出影响
16. //出队一个节点，查看其子节点
17. front = (SearchTree)FrontAndDequeue(myQueue);
18. printf("%d ", front->Element);
19. //将出队结点的左子节点入队
20. **if** (front->Left)
21. Enqueue((**int**)(front->Left), myQueue);
22. //将出队结点的右子节点入队
23. **if** (front->Right)
24. Enqueue((**int**)(front->Right), myQueue);
25. }
26. }

将程序运行，可见实现了相应功能：

