

# Linux 操作系统实验报告

学院：信息学院

班级：计算机科学与技术三班

姓名：赵彩霞

学号：161403322

指导教师：纪文迪

日期：2019 年 6 月 5 日

## 目录

Linux 操作系统实验报告.....	1
一、 实验题目.....	3
二、 实验要求.....	3
三、 实验设备.....	3
四、 实验内容.....	3
(一) 数据结构——堆.....	3
1、 实验原理.....	3
(1) 堆的概念.....	3
(2) 举例说明.....	3
(3) 堆排序的思想.....	5
2、 实验步骤.....	6
(1) C 语言代码.....	6
(2) 实验结果.....	7
(二) 数据结构——栈.....	7
1、 实验原理.....	7
(1) 栈的概念.....	7
(2) 栈的存储表示方法.....	7
1)、顺序栈.....	7
2)、链表栈.....	7
2、 实验步骤.....	8
(1) C 语言代码.....	8
(2) 实验结果.....	13
(三) 数据结构——B+树.....	14
1、 实验原理.....	14
(1) 二叉树.....	14
(2) 平衡二叉树.....	14
1)、B-tree.....	14
2)、B+ tree.....	14
2、 实验步骤.....	15
(1) C++语言代码.....	15
1)、BPlusTree.h.....	15
2)、BPlusTree.cpp.....	15
3)、test.cpp.....	25
(2) 实验结果.....	26
(四) 数据结构——红黑树.....	26
1、 实验原理.....	26
(1) avl 树.....	26
(2) 红黑树.....	26
2、 实验步骤.....	27
(1) C++语言代码.....	27
1)、RedBlackTree.h.....	27
2)、rbt.cpp.....	28
3)、test.cpp.....	43
(2) 实验结果.....	45
五、 实验小结.....	45

## 一、实验题目

在 Linux 系统下实现四种数据结构的相关功能

## 二、实验要求

- 1、学习堆、栈、B+树和红黑树等四种数据结构；
- 2、在 Linux 环境中利用上述四种数据结构进行编程，实现相关功能；
- 3、熟练掌握相关 Linux 操作命令；

## 三、实验设备

Linux 操作系统

## 四、实验内容

### （一）数据结构——堆

#### 1、实验原理

##### （1）堆的概念

堆实际上是一棵完全二叉树，其任何一非叶节点满足性质：

$Key[i] \leq key[2i+1] \&\& Key[i] \leq key[2i+2]$  或者  $Key[i] \geq Key[2i+1] \&\& key \geq key[2i+2]$

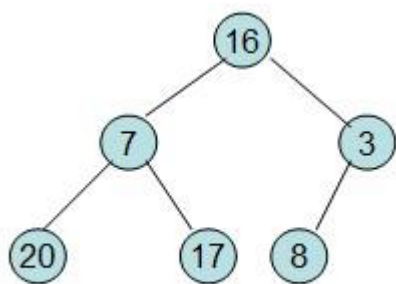
即任何一非叶节点的关键字不大于或者不小于其左右孩子节点的关键字。

堆分为大顶堆和小顶堆，满足  $Key[i] \geq Key[2i+1] \&\& key \geq key[2i+2]$  称为大顶堆，满足  $Key[i] \leq key[2i+1] \&\& Key[i] \leq key[2i+2]$  称为小顶堆。

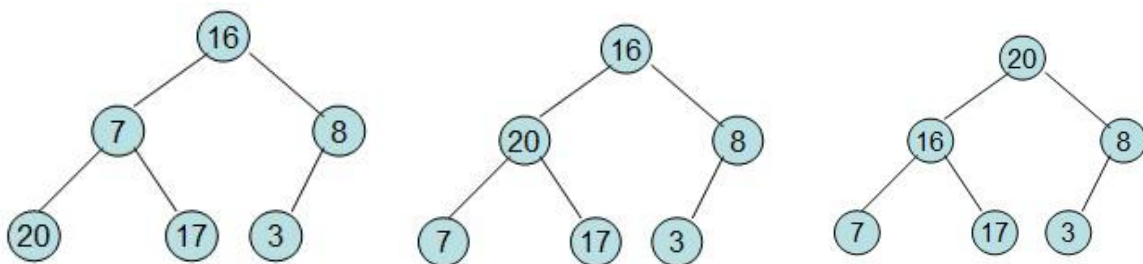
由上述性质可知大顶堆的堆顶的关键字肯定是所有关键字中最大的，小顶堆的堆顶的关键字是所有关键字中最小的。

##### （2）举例说明

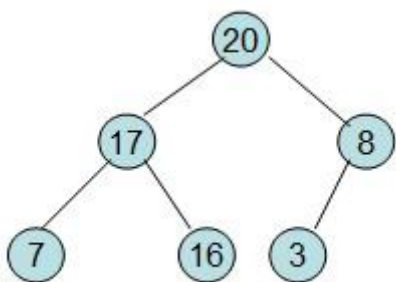
- 1）、首先根据该数组元素构建一个完全二叉树，得到



2)、然后需要构造初始堆，则从最后一个非叶节点开始调整，调整过程如下：

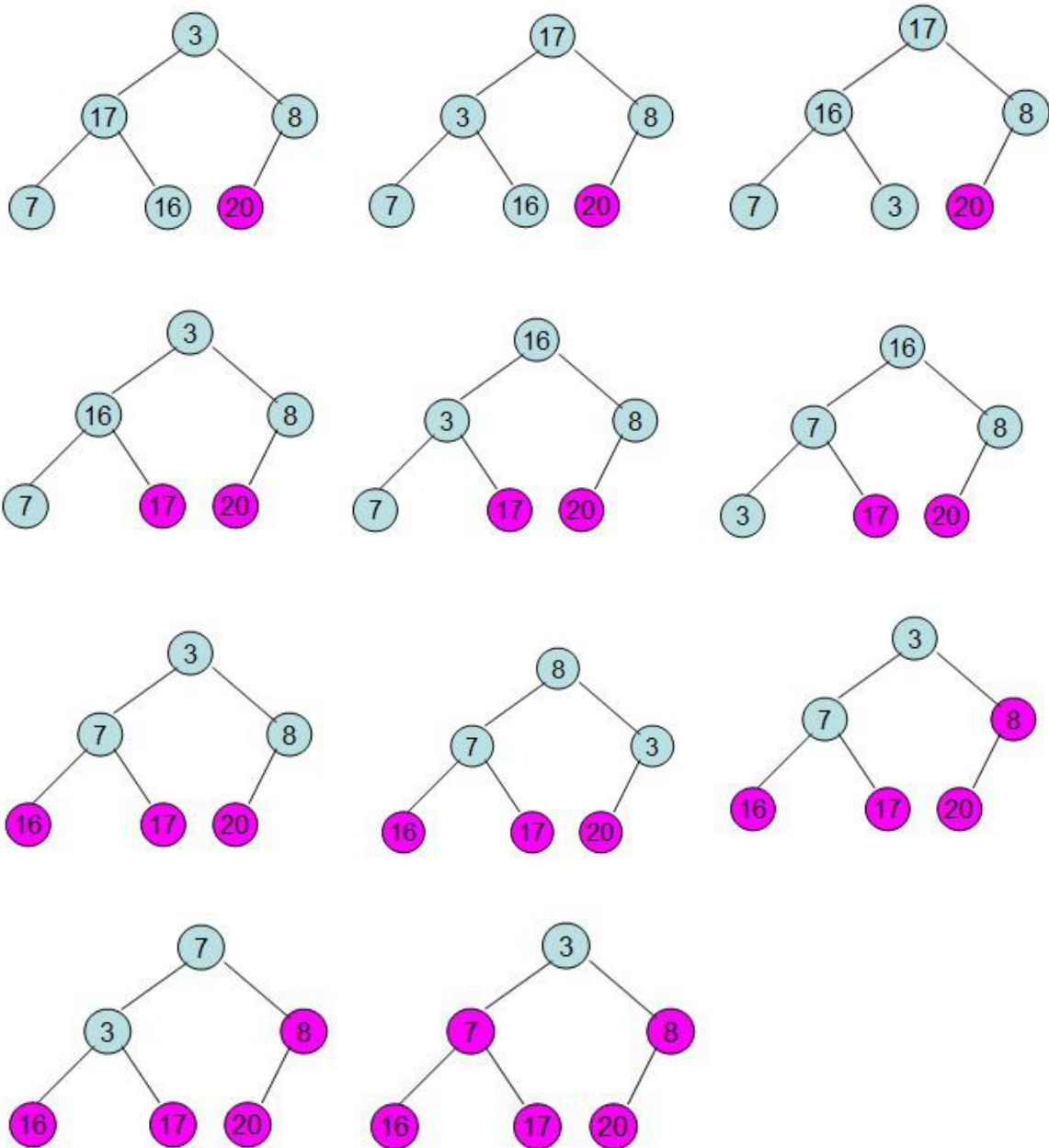


20 和 16 交换后导致 16 不满足堆的性质，因此需重新调整，得到了初始堆。



即每次调整都是从父节点、左孩子节点、右孩子节点三者中选择最大者跟父节点进行交换(交换之后可能造成被交换的孩子节点不满足堆的性质，因此每次交换之后要重新对被交换的孩子节点进行调整)。有了初始堆之后就可以进行排序了。

3)、进行排序



这样整个区间便已经有序了。

### (3) 堆排序的思想

- 1)、将待排元素依次存入数组，该数组即为一颗完全二叉树；
- 2)、从最后一个非叶节点开始递减，逐次将每个相应的二叉树调整成最大堆，最后，整个二叉树即为最大堆；
- 3)、交换堆顶元素与堆尾元素，调整新的堆为最大堆，数组分为两部分：堆与已排序列，直至最后一个堆元素。

从上面可以看出，堆排序大致可以分为三个步骤：

- 1)、存入数据（完全二叉树）
- 2)、将前  $n$  个元素调整为最大堆
- 3) 交换堆顶和堆尾元素， $n=n-1$ ，转 2

注：本文元素从数组下标 1 开始储存，所以 parent 访问二叉树左右儿子分别为  $2*parent$  和  $2*parent+1$ ；

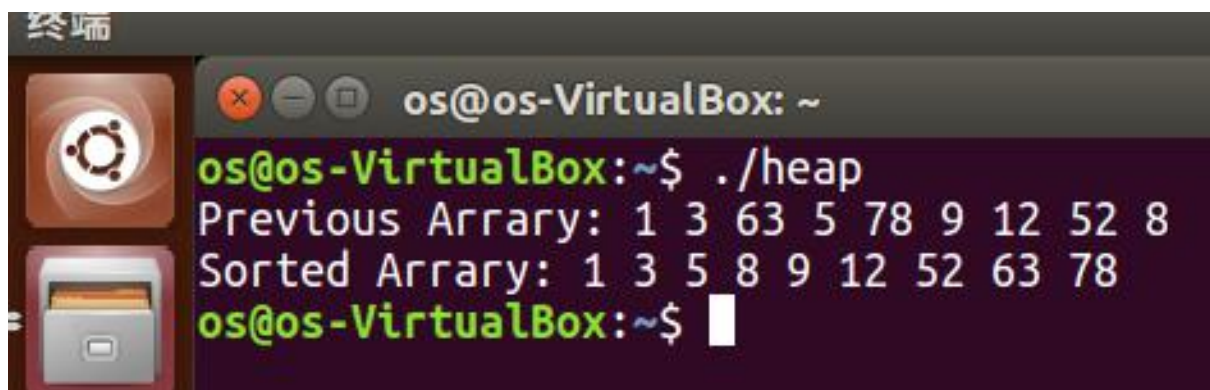
## 2、实验步骤

### (1) C 语言代码

A screenshot of a C code editor window. The window has a menu bar with '文件(F)', '编辑(E)', '查看(V)', '搜索(S)', '工具(T)', '文档(D)', and '帮助(H)'. Below the menu bar is a toolbar with '打开(O)' and a file icon. The left sidebar shows a vertical stack of application icons: Ubuntu, a folder, a document, Firefox, a presentation, a spreadsheet, a shopping bag, and a stack of papers. The main area contains C code for a heap sort algorithm. The code includes headers, defines NA as -1, and implements swap, HeapAdjust, HeapSort, and main functions. The main function initializes an array 'a' with values {NA, 1, 3, 63, 5, 78, 9, 12, 52, 8} and prints the array before and after sorting.

```
#include <stdio.h>
#define NA -1
void swap(int *a,int *b)//该函数用于交换两个变量的值
{
    int temp=*a;
    *a=*b;
    *b=temp;
}
void HeapAdjust(int H[],int start,int end)//堆调整，将start和end之间的元素调整为最大堆
{
    int temp=H[start];
    int parent=start,child;
    while(2*parent<=end)
    {
        child=2*parent;
        if(child!=end&&H[child]<H[child+1])++child;
        if(temp>H[child])break;
        else H[parent]=H[child];
        parent=child;
    } H[parent]=temp;
}
void HeapSort(int H[],int L,int R)
{
    for(int i=(R-L+1)/2;i>=L;--i)//调整整个二叉树为最大堆
        HeapAdjust(H,i,R);
    for(int i=R;i>=L;--i)
    {
        swap(&H[L],&H[i]);
        HeapAdjust(H,L,i-1);
    }
}
int main()
{
    int a[]={NA,1,3,63,5,78,9,12,52,8};//从1开始存入数据
    printf("Previous Array:");
    for(int i=1;i<=9;++i)
        printf(" %d",a[i]);
    HeapSort(a,1,9);
    printf("\nSorted Array:");
    for(int i=1;i<=9;++i)
        printf(" %d",a[i]);
    printf("\n");
    return 0;
}
```

## (2) 实验结果

A terminal window titled '终端' (Terminal) with a Ubuntu logo icon. The window shows a shell prompt 'os@os-VirtualBox: ~'. The user enters './heap', and the output shows 'Previous Array: 1 3 63 5 78 9 12 52 8' and 'Sorted Array: 1 3 5 8 9 12 52 63 78'. The prompt returns to 'os@os-VirtualBox: ~\$' with a cursor.

```
os@os-VirtualBox: ~$ ./heap
Previous Array: 1 3 63 5 78 9 12 52 8
Sorted Array: 1 3 5 8 9 12 52 63 78
os@os-VirtualBox: ~$
```

## (二) 数据结构——栈

### 1、实验原理

(1) 1、2、栈的存储表示方法:

#### (1) 栈的概念

栈 (stack) 是限定仅在表尾进行插入或删除操作的线性表。因此, 对栈来说, 表尾端有其特殊含义, 称为栈顶 (top), 相应的, 表头端称为栈底 (bottom)。不含元素的空表称为空栈。它按照先进后出 (last in first out) 的原则进行, 是一种 LIFO 结构的线性表。

#### (2) 栈的存储表示方法

##### 1)、顺序栈

即栈的顺序存储结构是利用一组地址连续的存储单元一次存放自栈底到栈顶的数据元素, 同时附设指针 `top` 只是栈顶元素在顺序栈中的位置。习惯上用 `top=0` 表示空栈, 由于栈在使用过程中所需最大空间的大小很难估计, 因此, 一般来说, 在初始化设空栈时不应限定栈的最大容量。所以可以先为栈分配一个基本容量, 然后在应用过程中, 当栈的空间不够用时再逐段扩大。为此, 设定两个常量: `STACK_INIT_SIZE` (存储空间初始分配量) 和 `STACKINCHREMENT` (存储空间分配增量)。

##### 2)、链表栈

栈的链式表示, 与单链表和双向链表十分相似, 不在详细描述。

## 2、实验步骤

### (1) C 语言代码

```
//利用栈实现简易计算器,进行包含+,-,*,/,(),间的计算
#include<stdio.h>
#include<string.h>
#define MaxSize 100
typedef struct CharStack    //字符栈
{
    char data[MaxSize];
    int top;
}cStack;
typedef struct DoubleStack //数据栈
{
    double data[MaxSize];
    int top;
}dStack;
int Isop(char );    //当前扫描元素优先级
int Inop(char );    //栈顶元素优先级
void Initc(cStack *);    //初始化字符栈
int Pushc(cStack *,char);    //字符栈压栈
char Gettopc(cStack *);    //返回栈顶元素
char Popc(cStack *);    //出栈
void Initd(dStack *);    //初始化数据栈
int Pushd(dStack *,double); //数据压栈
double Popd(dStack *);    //出栈
void Trans(char*s1,char*s2);    //转化为后缀表达式
double Calculate(char *s2);    //后缀表达式求值
int main()
{
    char s1[MaxSize];    //用于存储前缀表达式
    char s2[MaxSize];    //用于存储转换后的表达式
    printf("请输入表达式:");
    scanf("%s",s1);
    Trans(s1,s2);    //处理字符串,并转化为后缀表达式,存放在 s2 中
    printf("计算结果为: %f",Calculate(s2));    //后缀表达式求值
    return 0;
}
//初始化
void Initc(cStack *s1)
{
    s1->top=-1;
}
```



```

//字符栈压栈
int Pushc(cStack *c1,char op)
{
    if(c1->top<MaxSize)
    {
        c1->data[++c1->top]=op;
        return 1;
    }
    else return 0;
}
//GET 栈顶元素
char Gettopc(cStack *c1)
{
    return c1->data[c1->top];
}
//字符栈出栈
char Popc(cStack *c1)
{
    return c1->data[c1->top--];
}
//初始化数据栈
void Initd(dStack *d1)
{
    d1->top=-1;
}
//数据栈压栈
int Pushd(dStack *d1,double data)
{
    if(d1->top<MaxSize)
    {
        d1->data[++d1->top]=data;
        return 1;
    }
    else return 0;
}
//数据栈出栈
double Popd(dStack *d1)
{
    return d1->data[d1->top--];
}
int Isop(char op)//当前扫描运算符优先级
{
    switch(op)
    {
        case '(': return 6;
        case '+': case '-': return 2;
    }
}

```

```

        case '*': case '/': return 4;
    }
}
int Inop(char op)    // 当前扫描运算符优先级
{
    switch(op)
    {
        case '(': return 1;
        case '+': case '-': return 3;
        case '*': case '/': return 5;
    }
}
void Trans(char *s1,char *s2)
{
    int i=0;
    int j=0;
    int flag1=-1;//flag1 为 0 表示上次输出为数字，flag1 为 1 表示上次输出为字符
    int flag2=-1;    //flag2 为 0 表示上次扫描为数字，flag 为 1 表示上次扫描为运算符，用于
    区分数字后加空格
    cStack st1; //暂放运算符
    Initc(&st1);
    while(s1[i]!='\0') //处理负数
    {
        if(s1[0]=='-')    //第一位数字为负数时
        {
            j=strlen(s1);
            while(j>0)
            {
                s1[j+5]=s1[j];
                j--;
            }
            s1[j++]='(';
            s1[j++]='0';
            s1[j++]='-';
            s1[j++]='1';
            s1[j++]=')';
            s1[j]='*';
        }
        if(s1[j]=='('&& s1[i+1]=='-')    //非第一位负数时
        {
            j=strlen(s1);
            while(j>i+1)
            {
                s1[j+5]=s1[j];
                j--;
            }

```

```

    }
    s1[j++]='(';
    s1[j++]='0';
    s1[j++]='-';
    s1[j++]='1';
    s1[j++]=')';
    s1[j]='*';
    i=i+5;
}
i++;
}
i=0;
j=0;
while(s1[i]!='\0')
{
    if(flag1==0&&flag2==1) //若上次的输出为数字，上次循环扫描为字符，则表示该数字
串结束，则在数字后加空格区分
    {
        s2[j++]=' ';
        flag1=1;
    }
    if(s1[i]>='0'&&s1[i]<='9' || s1[i]=='.')
    {
        s2[j++]=s1[i];
        flag2=0;
        flag1=0;
    }
    else if(s1[i]=='+' || s1[i]=='-' || s1[i]=='*' || s1[i]=='/' || s1[i]=='(')
    {
        flag2=1;
        if(st1.top<0 || Isop(s1[i])>Inop(Gettopc(&st1)))
        {
            Pushc(&st1,s1[i]);
        }
        else
        {
            while(st1.top>=0&&Isop(s1[i])<Inop(Gettopc(&st1)))
//当前扫描字符优先级不断与栈顶字符优先级比较，当前字符小于栈顶字符时退栈并输出
            {
                s2[j++]=Popc(&st1);
                flag1=1;
            }
            if(st1.top<0 || Isop(s1[i])>Inop(Gettopc(&st1)))
//当前字符优先级大于栈顶优先级或栈空时当前字符压入字符栈内
            {
                Pushc(&st1,s1[i]);

```

```

        }

    }
}
else if(s1[i]=='')
{
    flag2=1;
    if(Gettopc(&st1)!='(')    //若括号仅包含数字则没有输出运算符
    {
        flag1=1;
    }
    while(Gettopc(&st1)!='(')
    {
        s2[j++]=Popc(&st1);
    }
    Popc(&st1);    //将 '(' 出栈
}
i++;
}
while(st1.top>=0)    //将栈内剩余的运算符依次退栈输出
{
    s2[j++]=Popc(&st1);
}
s2[j]='\0';
}
//表达式求值
double Calculate(char *s1)
{
    int i=0;
    int flag;    //char 类型转换为 double 类型数据标记
    double data1,data2;
    double sum;
    dStack ds1;
    Initd(&ds1);
    while(s1[i]!='\0')
    {
        if(s1[i]=='+' | | s1[i]=='-' | | s1[i]=='*' | | s1[i]=='/')    //若为运算符获取栈顶两个
元素进行计算
        {
            data1=Popd(&ds1);
            data2=Popd(&ds1);
            if(s1[i]=='+') Pushd(&ds1,data2+data1);
            else if(s1[i]=='-') Pushd(&ds1,data2-data1);
            else if(s1[i]=='*') Pushd(&ds1,data2*data1);
            else if(s1[i]=='/') Pushd(&ds1,data2/data1);
        }
    }
}

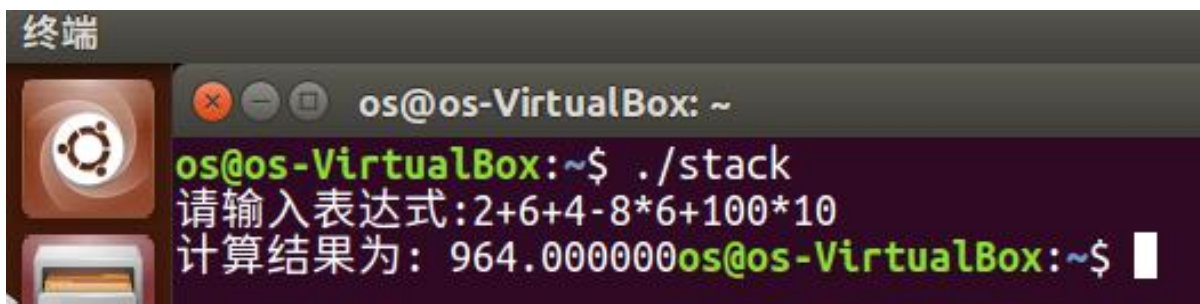
```

```

else                                     //为数据时转化为 double 类型压栈
{
    flag=0;                             //初始化为 0 为整数部分标记，1 为小数部分标记
    sum=0;
    double divider=1;
    while(s1[i]!='&&s1[i]!='+'&&s1[i]!='-'&&s1[i]!='*&&s1[i]!='/' )
    {
        if(s1[i]=='.')                  //若有小数点，进入小数转化模式
        {
            flag=1;
            i++;
            continue;
        }
        if(flag==0)
        {
            sum=sum*10+(double)(s1[i]-'0');
        }
        else
        {
            divider=divider*10;
            sum=sum+((double)(s1[i]-'0'))/divider;
        }
        i++;
    }
    if(s1[i]=='+' || s1[i]=='-' || s1[i]=='*' || s1[i]=='/') i--; //转化成功一个数据，若下个
    字符为运算符，则 i--，回到当前运算的数据位置
    Pushd(&ds1,sum);
}
i++;    //i++准备下一个字符的转换
}
return Popd(&ds1);
}

```

## (2) 实验结果



## （三）数据结构——B+树

### 1、实验原理

#### （1）二叉树

二叉树 **binary tree** 是指每个节点最多含有两个子树的树结构。

特点：

- a. 所有节点最多拥有两个子节点，即度不大于 2
- b. 左子树的键值小于根的键值，右子树的键值大于根的键值。

因为二叉树只是定义了简单的结构，所以存在多种深度可能，导致二叉树的效率低，所以引入了平衡二叉树。

#### （2）平衡二叉树

##### 1）、B-tree

B 通常理解成是 **Balance** 的意思，**B-tree** 就是 B 树，简称平衡树。

B 树是平衡多路查找树（有多个查找路径，不止 2 个），是一种平衡的多叉树。因为 B 树是平衡树，每个节点到叶子节点的高度都是相同的，这样可以保证 B 树的查询是稳定的。

使用 B tree 可以显著减少定位记录时所经历的中间过程，从而快速定位，加快存取速度。

与二叉树相比，**B-tree** 利用多个分支（二叉树只有 2 个分支）节点，减少了获取记录时所经历的节点数，从而达到节省存取时间的目的。

特点：

- a. 每个节点的关键字增多了，特别是 B 树应用到数据库中的时候。

数据库充分利用了磁盘块的原理（磁盘数据的存储采用的是块的形式进行存储，每个块的大小一般为 4k,每次去取数据的时候，就是取出这个 4k 的大小，而不是只取出你想要的大小。就是说每次 IO 的时候，同一磁盘块的数据都是一次性提取出来）。把树的节点关键字增多后，树的层级比原来二叉树的层级少了，这样就可以减少数据查找的次数，降低复杂度了。

- b、所有的页节点都在同一层上

##### 2）、B+ tree

B+tree 是在 B-tree 基础上的优化，使其更适应存储索引结构

B-tree 的结构中，每个节点不仅包括数据的 **key** 值，也包括 **data** 值。而每一页的存储空间都是有限的，如果 **data** 数据较大的时候，会导致，每一页中存储的 **key** 比较少，当存储的数据量比较大时，同样会导致 B-tree 的查询深度很大，增加磁盘 IO 次数，进而影响查询效率

B+ tree 中，非叶子节点上只存储 **key** 的信息，这样可以加大每一页中存储 **key** 的数量，降低 B+tree 的高度。

特点（与 B-tree 相比）：

- a. 非叶子节点只存储 **key** 信息

- b. 所有叶子节点之间有一个链指针
- c. B+的非叶子节点只进行数据的索引，不会存实际的关键字记录的指针，所有数据地址必须要到叶子节点才能获取到，所以每次数据查询的次数都一样。
- d. B+树的应用场景主要是数据库索引结构，数据库的查询有时候可能一次多条，如果分布在不同的层（树的层级），那么在取出数据后，还需要做排序。而在一个层级上，且有指针连接各个叶子节点也使得查询效率更高。

## 2、实验步骤

### （1）C++语言代码

#### 1)、BPlusTree.h

```
#ifndef BPlusTree_h
#define BPlusTree_h
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define M (4)
#define LIMIT_M_2 (M % 2 ? (M + 1)/2 : M/2)
typedef struct BPlusNode *BPlusTree,*Position;
typedef int KeyType;
struct BPlusNode{
    int KeyNum;
    KeyType Key[M + 1];
    BPlusTree Children[M + 1];
    BPlusTree Next;
};
extern BPlusTree Initialize();/* 初始化 */
extern BPlusTree Insert(BPlusTree T,KeyType Key);/* 插入 */
extern BPlusTree Remove(BPlusTree T,KeyType Key);/* 删除 */
extern BPlusTree Destroy(BPlusTree T);/* 销毁 */
extern void Travel(BPlusTree T);/* 遍历节点 */
extern void TravelData(BPlusTree T);/* 遍历树叶节点的数据 */
#endif /* BPlusTree_h */
```

#### 2)、BPlusTree.cpp

```
#include "BPlusTree.h"
static KeyType Unavailable = INT_MIN;
/* 生成节点并初始化 */
static BPlusTree MallocNewNode(){
    BPlusTree NewNode;
    int i;
    NewNode = malloc(sizeof(struct BPlusNode));
    if (NewNode == NULL)
```

```

        exit(EXIT_FAILURE);
    i = 0;
    while (i < M + 1){
        NewNode->Key[i] = Unavailable;
        NewNode->Children[i] = NULL;
        i++;
    }
    NewNode->Next = NULL;
    NewNode->KeyNum = 0;

    return NewNode;
}
/* 初始化 */
extern BPlusTree Initialize(){

    BPlusTree T;
    if (M < (3)){
        printf("M 最小等于 3! ");
        exit(EXIT_FAILURE);
    }
    /* 根结点 */
    T = MallocNewNode();

    return T;
}

static Position FindMostLeft(Position P){
    Position Tmp;

    Tmp = P;

    while (Tmp != NULL && Tmp->Children[0] != NULL) {
        Tmp = Tmp->Children[0];
    }
    return Tmp;
}

static Position FindMostRight(Position P){
    Position Tmp;

    Tmp = P;

    while (Tmp != NULL && Tmp->Children[Tmp->KeyNum - 1] != NULL) {
        Tmp = Tmp->Children[Tmp->KeyNum - 1];
    }
}

```



```

    return Tmp;
}
/* 寻找一个兄弟节点，其存储的关键字未满，否则返回 NULL */
static Position FindSibling(Position Parent,int i){
    Position Sibling;
    int Limit;
    Limit = M;
    Sibling = NULL;
    if (i == 0){
        if (Parent->Children[1]->KeyNum < Limit)
            Sibling = Parent->Children[1];
    }
    else if (Parent->Children[i - 1]->KeyNum < Limit)
        Sibling = Parent->Children[i - 1];
    else if (i + 1 < Parent->KeyNum && Parent->Children[i + 1]->KeyNum < Limit){
        Sibling = Parent->Children[i + 1];
    }
    return Sibling;
}
/* 查找兄弟节点，其关键字数大于 M/2 ;没有返回 NULL*/
static Position FindSiblingKeyNum_M_2(Position Parent,int i,int *j){
    int Limit;
    Position Sibling;
    Sibling = NULL;
    Limit = LIMIT_M_2;
    if (i == 0){
        if (Parent->Children[1]->KeyNum > Limit){
            Sibling = Parent->Children[1];
            *j = 1;
        }
    }
    else{
        if (Parent->Children[i - 1]->KeyNum > Limit){
            Sibling = Parent->Children[i - 1];
            *j = i - 1;
        }
        else if (i + 1 < Parent->KeyNum && Parent->Children[i + 1]->KeyNum > Limit){
            Sibling = Parent->Children[i + 1];
            *j = i + 1;
        }
    }
    return Sibling;
}
/* 当要对 X 插入 Key 的时候，i 是 X 在 Parent 的位置，j 是 Key 要插入的位置
   当要对 Parent 插入 X 节点的时候，i 是要插入的位置，Key 和 j 的值没有用
   */

```

```

static Position InsertElement(int isKey, Position Parent, Position X, KeyType Key, int i, int j){
    int k;
    if (isKey){
        /* 插入 key */
        k = X->KeyNum - 1;
        while (k >= j){
            X->Key[k + 1] = X->Key[k]; k--;
        }
        X->Key[j] = Key;
        if (Parent != NULL)
            Parent->Key[i] = X->Key[0];
        X->KeyNum++;
    }else{
        /* 插入节点 */
        /* 对树叶节点进行连接 */
        if (X->Children[0] == NULL){
            if (i > 0)
                Parent->Children[i - 1]->Next = X;
            X->Next = Parent->Children[i];
        }
        k = Parent->KeyNum - 1;
        while (k >= i){
            Parent->Children[k + 1] = Parent->Children[k];
            Parent->Key[k + 1] = Parent->Key[k];
            k--;
        }
        Parent->Key[i] = X->Key[0];
        Parent->Children[i] = X;
        Parent->KeyNum++;
    }
    return X;
}

```

```

static Position RemoveElement(int isKey, Position Parent, Position X, int i, int j){
    int k, Limit;
    if (isKey){
        Limit = X->KeyNum;
        /* 删除 key */
        k = j + 1;
        while (k < Limit){
            X->Key[k - 1] = X->Key[k]; k++;
        }
        X->Key[X->KeyNum - 1] = Unavailable;

        Parent->Key[i] = X->Key[0];
        X->KeyNum--;
    }
}

```

```

}else{
    /* 删除节点 */
    /* 修改树叶节点的链接 */
    if (X->Children[0] == NULL && i > 0){
        Parent->Children[i - 1]->Next = Parent->Children[i + 1];
    }
    Limit = Parent->KeyNum;
    k = i + 1;
    while (k < Limit){
        Parent->Children[k - 1] = Parent->Children[k];
        Parent->Key[k - 1] = Parent->Key[k];
        k++;
    }
    Parent->Children[Parent->KeyNum - 1] = NULL;
    Parent->Key[Parent->KeyNum - 1] = Unavailable;
    Parent->KeyNum--;
}
return X;
}
/* Src 和 Dst 是两个相邻的节点, i 是 Src 在 Parent 中的位置;
   将 Src 的元素移动到 Dst 中 ,n 是移动元素的个数*/
static Position MoveElement(Position Src,Position Dst,Position Parent,int i,int n){
    KeyType TmpKey;
    Position Child;
    int j,SrcInFront;
    SrcInFront = 0;
    if (Src->Key[0] < Dst->Key[0])
        SrcInFront = 1;
    j = 0;
    /* 节点 Src 在 Dst 前面 */
    if (SrcInFront){
        if (Src->Children[0] != NULL){
            while (j < n) {
                Child = Src->Children[Src->KeyNum - 1];
                RemoveElement(0, Src, Child, Src->KeyNum - 1, Unavailable);
                InsertElement(0, Dst, Child, Unavailable, 0, Unavailable);
                j++;
            }
        }
    }else{
        while (j < n) {
            TmpKey = Src->Key[Src->KeyNum - 1];
            RemoveElement(1, Parent, Src, i, Src->KeyNum - 1);
            InsertElement(1, Parent, Dst, TmpKey, i + 1, 0);
            j++;
        }
    }
}

```

```

    Parent->Key[i + 1] = Dst->Key[0];
    /* 将树叶节点重新连接 */
    if (Src->KeyNum > 0)
        FindMostRight(Src)->Next = FindMostLeft(Dst);
} else {
    if (Src->Children[0] != NULL) {
        while (j < n) {
            Child = Src->Children[0];
            RemoveElement(0, Src, Child, 0, Unavailable);
            InsertElement(0, Dst, Child, Unavailable, Dst->KeyNum, Unavailable);
            j++;
        }
    } else {
        while (j < n) {
            TmpKey = Src->Key[0];
            RemoveElement(1, Parent, Src, i, 0);
            InsertElement(1, Parent, Dst, TmpKey, i - 1, Dst->KeyNum);
            j++;
        }
    }
    Parent->Key[i] = Src->Key[0];
    if (Src->KeyNum > 0)
        FindMostRight(Dst)->Next = FindMostLeft(Src);
}
return Parent;
}

```

```

static BPlusTree SplitNode(Position Parent, Position X, int i) {
    int j, k, Limit;
    Position NewNode;
    NewNode = MallocNewNode();
    k = 0;
    j = X->KeyNum / 2;
    Limit = X->KeyNum;
    while (j < Limit) {
        if (X->Children[0] != NULL) {
            NewNode->Children[k] = X->Children[j];
            X->Children[j] = NULL;
        }
        NewNode->Key[k] = X->Key[j];
        X->Key[j] = Unavailable;
        NewNode->KeyNum++; X->KeyNum--;
        j++; k++;
    }
    if (Parent != NULL)
        InsertElement(0, Parent, NewNode, Unavailable, i + 1, Unavailable);
}

```

```

else{
    /* 如果是 X 是根，那么创建新的根并返回 */
    Parent = MallocNewNode();
    InsertElement(0, Parent, X, Unavailable, 0, Unavailable);
    InsertElement(0, Parent, NewNode, Unavailable, 1, Unavailable);
    return Parent;
}
return X;
}
/* 合并节点,X 少于 M/2 关键字，S 有大于或等于 M/2 个关键字*/
static Position MergeNode(Position Parent, Position X, Position S, int i){
    int Limit;
    /* S 的关键字数目大于 M/2 */
    if (S->KeyNum > LIMIT_M_2){
        /* 从 S 中移动一个元素到 X 中 */
        MoveElement(S, X, Parent, i, 1);
    }else{
        /* 将 X 全部元素移动到 S 中，并把 X 删除 */
        Limit = X->KeyNum;
        MoveElement(X, S, Parent, i, Limit);
        RemoveElement(0, Parent, X, i, Unavailable);
        free(X);
        X = NULL;
    }
    return Parent;
}
static BPlusTree RecursiveInsert(BPlusTree T, KeyType Key, int i, BPlusTree Parent){
    int j, Limit;
    Position Sibling;
    /* 查找分支 */
    j = 0;
    while (j < T->KeyNum && Key >= T->Key[j]){
        /* 重复值不插入 */
        if (Key == T->Key[j])
            return T;
        j++;
    }
    if (j != 0 && T->Children[0] != NULL) j--;
    /* 树叶 */
    if (T->Children[0] == NULL)
        T = InsertElement(1, Parent, T, Key, i, j);
    /* 内部节点 */
    else
        T->Children[j] = RecursiveInsert(T->Children[j], Key, j, T);

    /* 调整节点 */

```

```

Limit = M;
if (T->KeyNum > Limit){
    /* 根 */
    if (Parent == NULL){
        /* 分裂节点 */
        T = SplitNode(Parent, T, i);
    }
    else{
        Sibling = FindSibling(Parent, i);
        if (Sibling != NULL){
            /* 将 T 的一个元素 (Key 或者 Child) 移动的 Sibling 中 */
            MoveElement(T, Sibling, Parent, i, 1);
        }else{
            /* 分裂节点 */
            T = SplitNode(Parent, T, i);
        }
    }
}
if (Parent != NULL)
    Parent->Key[i] = T->Key[0];
return T;
}
/* 插入 */
extern BPlusTree Insert(BPlusTree T,KeyType Key){
    return RecursiveInsert(T, Key, 0, NULL);
}

static BPlusTree RecursiveRemove(BPlusTree T,KeyType Key,int i,BPlusTree Parent){
    int j,NeedAdjust;
    Position Sibling,Tmp;
    Sibling = NULL;
    /* 查找分支 */
    j = 0;
    while (j < T->KeyNum && Key >= T->Key[j]){
        if (Key == T->Key[j])
            break;
        j++;
    }
    if (T->Children[0] == NULL){
        /* 没找到 */
        if (Key != T->Key[j] || j == T->KeyNum)
            return T;
    }else
        if (j == T->KeyNum || Key < T->Key[j]) j--;
    /* 树叶 */
    if (T->Children[0] == NULL){

```

```

        T = RemoveElement(1, Parent, T, i, j);
    }else{
        T->Children[j] = RecursiveRemove(T->Children[j], Key, j, T);
    }
    NeedAdjust = 0;
    /* 树的根或者是一片树叶，或者其儿子数在 2 到 M 之间 */
    if (Parent == NULL && T->Children[0] != NULL && T->KeyNum < 2)
        NeedAdjust = 1;
    /* 除根外，所有非树叶节点的儿子数在[M/2]到 M 之间。(符号[]表示向上取整) */
    else if (Parent != NULL && T->Children[0] != NULL && T->KeyNum < LIMIT_M_2)
        NeedAdjust = 1;
    /* (非根) 树叶中关键字的个数也在[M/2]和 M 之间 */
    else if (Parent != NULL && T->Children[0] == NULL && T->KeyNum < LIMIT_M_2)
        NeedAdjust = 1;
    /* 调整节点 */
    if (NeedAdjust){
        /* 根 */
        if (Parent == NULL){
            if(T->Children[0] != NULL && T->KeyNum < 2){
                Tmp = T;
                T = T->Children[0];
                free(Tmp);
                return T;
            }
        }else{
            /* 查找兄弟节点，其关键字数目大于 M/2 */
            Sibling = FindSiblingKeyNum_M_2(Parent, i,&j);
            if (Sibling != NULL){
                MoveElement(Sibling, T, Parent, j, 1);
            }else{
                if (i == 0)
                    Sibling = Parent->Children[1];
                else
                    Sibling = Parent->Children[i - 1];

                Parent = MergeNode(Parent, T, Sibling, i);
                T = Parent->Children[i];
            }
        }
    }
    return T;
}
/* 删除 */
extern BPlusTree Remove(BPlusTree T,KeyType Key){
    return RecursiveRemove(T, Key, 0, NULL);
}

```

```

/* 销毁 */
extern BPlusTree Destroy(BPlusTree T){
    int i,j;
    if (T != NULL){
        i = 0;
        while (i < T->KeyNum + 1){
            Destroy(T->Children[i]);i++;
        }
        printf("Destroy:");
        j = 0;
        while (j < T->KeyNum)/* T->Key[i] != Unavailable*/
            printf("%d:",T->Key[j++]);
        printf(" \n");
        free(T);
        T = NULL;
    }
    return T;
}

```

```

static void RecursiveTravel(BPlusTree T,int Level){
    int i;
    if (T != NULL){
        printf(" ");
        printf("[Level:%d]-->",Level);
        printf("(");
        i = 0;
        while (i < T->KeyNum)/* T->Key[i] != Unavailable*/
            printf("%d:",T->Key[i++]);
        printf(")\n");
        Level++;
        i = 0;
        while (i <= T->KeyNum) {
            RecursiveTravel(T->Children[i], Level);
            i++;
        }
    }
}

```

```

/* 遍历 */
extern void Travel(BPlusTree T){
    RecursiveTravel(T, 0);
    printf("\n");
}

/* 遍历树叶节点的数据 */
extern void TravelData(BPlusTree T){
    Position Tmp;
    int i;

```



```

    if (T == NULL)
        return ;
    printf("All Data:");
    Tmp = T;
    while (Tmp->Children[0] != NULL)
        Tmp = Tmp->Children[0];
    /* 第一片树叶 */
    while (Tmp != NULL){
        i = 0;
        while (i < Tmp->KeyNum)
            printf(" %d",Tmp->Key[i++]);
        Tmp = Tmp->Next;
    }
}

```

### 3) 、 test.cpp

```

#include <stdio.h>
#include <time.h>
#include "BPlusTree.h"
int main(int argc, const char * argv[]) {
    int i;
    BPlusTree T;
    T = Initialize();
    clock_t c1 = clock();
    i = 10000000;
    while (i > 0)
        T = Insert(T, i--);
    i = 5000001;
    while (i < 10000000)
        T = Insert(T, i++);
    i = 10000000;
    while (i > 100)
        T = Remove(T, i--);
    Travel(T);
    Destroy(T);
    clock_t c2 = clock();
    printf("\n 用时:  %lu 秒\n",(c2 - c1)/CLOCKS_PER_SEC);
}

```

## (2) 实验结果



```
os@os-VirtualBox:~$ ./bt
[Level:0]-->(1:05:)
[Level:1]-->(1:17:33:49:)
[Level:2]-->(1:5:9:13:)
[Level:3]-->(1:2:3:4:)
[Level:3]-->(5:6:7:8:)
[Level:3]-->(9:10:11:12:)
[Level:3]-->(13:14:15:16:)
[Level:2]-->(17:21:25:29:)
[Level:3]-->(17:18:19:20:)
[Level:3]-->(21:22:23:24:)
[Level:3]-->(25:26:27:28:)
[Level:3]-->(29:30:31:32:)
[Level:2]-->(33:37:41:45:)
[Level:3]-->(33:34:35:36:)
[Level:3]-->(37:38:39:40:)
[Level:3]-->(41:42:43:44:)
[Level:3]-->(45:46:47:48:)
[Level:2]-->(49:53:57:61:)
[Level:3]-->(49:50:51:52:)
[Level:3]-->(53:54:55:56:)
[Level:3]-->(57:58:59:60:)
[Level:3]-->(61:62:63:64:)
[Level:1]-->(65:81:97:)
[Level:2]-->(65:69:73:77:)
[Level:3]-->(65:66:67:68:)
[Level:3]-->(69:70:71:72:)
[Level:3]-->(73:74:75:76:)
[Level:3]-->(77:78:79:80:)
[Level:2]-->(81:85:89:93:)
[Level:3]-->(81:82:83:84:)
[Level:3]-->(85:86:87:88:)
[Level:3]-->(89:90:91:92:)
[Level:3]-->(93:94:95:96:)
[Level:2]-->(97:99:)
[Level:3]-->(97:98:)
[Level:3]-->(99:100:)
Destroy:(1:2:3:4:)
Destroy:(5:6:7:8:)
Destroy:(9:10:11:12:)
Destroy:(13:14:15:16:)
Destroy:(1:5:9:13:)
Destroy:(17:18:19:20:)
Destroy:(21:22:23:24:)
Destroy:(25:26:27:28:)
Destroy:(29:30:31:32:)
Destroy:(17:21:25:29:)
Destroy:(33:34:35:36:)
Destroy:(37:38:39:40:)
Destroy:(41:42:43:44:)
Destroy:(45:46:47:48:)
Destroy:(33:37:41:45:)
Destroy:(49:50:51:52:)
Destroy:(53:54:55:56:)
Destroy:(57:58:59:60:)
Destroy:(61:62:63:64:)
Destroy:(49:53:57:61:)
Destroy:(1:17:33:49:)
Destroy:(65:66:67:68:)
Destroy:(69:70:71:72:)
Destroy:(73:74:75:76:)
Destroy:(77:78:79:80:)
Destroy:(65:69:73:77:)
Destroy:(81:82:83:84:)
Destroy:(85:86:87:88:)
Destroy:(89:90:91:92:)
Destroy:(93:94:95:96:)
Destroy:(81:85:89:93:)
Destroy:(97:98:)
Destroy:(99:100:)
Destroy:(65:81:97:)
用时: 0秒
os@os-VirtualBox:~$
```

## (四) 数据结构——红黑树

### 1、实验原理

#### (1) avl 树

平衡二叉树，基于 avl 算法，即是 avl 树 (avl tree)  
特点:

- a、符合二叉树的条件下
- b、任何节点的两个子树的高度最大差为 1

如果在 avl 树，中进行插入和删除节点操作，可能导致 avl 树失去平衡，那么可以通过旋转重新达到平衡。因此我们说的二叉树也称自平衡二叉树。

#### (2) 红黑树

红黑树和 avl 树类似，都是在进行插入和删除操作时通过特定的操作保持二叉树的平衡，从而获

得较高的查找性能。

在 java 中 TreeSet, TreeMap 的底层就是用的这个方法。

特点:

- a、节点是红色或黑色
- b、根节点是黑色
- c、叶子节点 (nil,空节点) 是黑色
- d、每个红色节点的两个子节点都是黑色

## 2、实验步骤

### (1) C++语言代码

#### 1)、RedBlackTree.h

```
#ifndef _RED_BLACK_TREE_H_
#define _RED_BLACK_TREE_H_
#define RED      0    // 红色节点
#define BLACK    1    // 黑色节点
typedef int Type;
// 红黑树的节点
typedef struct RBTreeNode{
    unsigned char color;    // 颜色(RED 或 BLACK)
    Type    key;            // 关键字(键值)
    struct RBTreeNode *left; // 左孩子
    struct RBTreeNode *right; // 右孩子
    struct RBTreeNode *parent; // 父结点
}Node, *RBTree;
// 红黑树的根
typedef struct rb_root{
    Node *node;
}RBRoot;
// 创建红黑树, 返回"红黑树的根"!
RBRoot* create_rbtree();
// 销毁红黑树
void destroy_rbtree(RBRoot *root);
// 将结点插入到红黑树中。插入成功, 返回 0; 失败返回-1。
int insert_rbtree(RBRoot *root, Type key);
// 删除结点(key 为节点的值)
void delete_rbtree(RBRoot *root, Type key);
// 前序遍历"红黑树"
void preorder_rbtree(RBRoot *root);
// 中序遍历"红黑树"
void inorder_rbtree(RBRoot *root);
// 后序遍历"红黑树"
void postorder_rbtree(RBRoot *root);
```

```

// (递归实现)查找"红黑树"中键值为 key 的节点。找到的话，返回 0；否则，返回-1。
int rbtree_search(RBRoot *root, Type key);
// (非递归实现)查找"红黑树"中键值为 key 的节点。找到的话，返回 0；否则，返回-1。
int iterative_rbtree_search(RBRoot *root, Type key);
// 返回最小结点的值(将值保存到 val 中)。找到的话，返回 0；否则返回-1。
int rbtree_minimum(RBRoot *root, int *val);
// 返回最大结点的值(将值保存到 val 中)。找到的话，返回 0；否则返回-1。
int rbtree_maximum(RBRoot *root, int *val);
// 打印红黑树
void print_rbtree(RBRoot *root);
#endif

```

## 2) 、rbt.cpp

```

#include <stdio.h>
#include <stdlib.h>
#include "RedBlackTree.h"
#define rb_parent(r) ((r)->parent)
#define rb_color(r) ((r)->color)
#define rb_is_red(r) ((r)->color==RED)
#define rb_is_black(r) ((r)->color==BLACK)
#define rb_set_black(r) do { (r)->color = BLACK; } while (0)
#define rb_set_red(r) do { (r)->color = RED; } while (0)
#define rb_set_parent(r,p) do { (r)->parent = (p); } while (0)
#define rb_set_color(r,c) do { (r)->color = (c); } while (0)
/*
 * 创建红黑树，返回"红黑树的根"!
 */
RBRoot* create_rbtree()
{
    RBRoot *root = (RBRoot *)malloc(sizeof(RBRoot));
    root->node = NULL;
    return root;
}
/*
 * 前序遍历"红黑树"
 */
static void preorder(RBTree tree)
{
    if(tree != NULL)
    {
        printf("%d ", tree->key);
        preorder(tree->left);
        preorder(tree->right);
    }
}
void preorder_rbtree(RBRoot *root)

```

```

{
    if (root)
        preorder(root->node);
}
/*
 * 中序遍历"红黑树"
 */
static void inorder(RBTree tree)
{
    if(tree != NULL)
    {
        inorder(tree->left);
        printf("%d ", tree->key);
        inorder(tree->right);
    }
}
void inorder_rbtree(RBRoot *root)
{
    if (root)
        inorder(root->node);
}
/*
 * 后序遍历"红黑树"
 */
static void postorder(RBTree tree)
{
    if(tree != NULL)
    {
        postorder(tree->left);
        postorder(tree->right);
        printf("%d ", tree->key);
    }
}

void postorder_rbtree(RBRoot *root)
{
    if (root)
        postorder(root->node);
}
/*
 * (递归实现)查找"红黑树 x"中键值为 key 的节点
 */
static Node* search(RBTree x, Type key)
{
    if (x==NULL || x->key==key)
        return x;
}

```

```

        if (key < x->key)
            return search(x->left, key);
        else
            return search(x->right, key);
    }
int rbtree_search(RBRoot *root, Type key)
{
    if (root)
        return search(root->node, key)? 0 : -1;
}
/*
 * (非递归实现)查找"红黑树 x"中键值为 key 的节点
 */
static Node* iterative_search(RBTree x, Type key)
{
    while ((x!=NULL) && (x->key!=key))
    {
        if (key < x->key)
            x = x->left;
        else
            x = x->right;
    }

    return x;
}
int iterative_rbtree_search(RBRoot *root, Type key)
{
    if (root)
        return iterative_search(root->node, key) ? 0 : -1;
}
/*
 * 查找最小结点: 返回 tree 为根结点的红黑树的最小结点。
 */
static Node* minimum(RBTree tree)
{
    if (tree == NULL)
        return NULL;
    while(tree->left != NULL)
        tree = tree->left;
    return tree;
}

int rbtree_minimum(RBRoot *root, int *val)
{
    Node *node;

```

```

    if (root)
        node = minimum(root->node);
    if (node == NULL)
        return -1;
    *val = node->key;
    return 0;
}
/*
 * 查找最大结点：返回 tree 为根结点的红黑树的最大结点。
 */
static Node* maximum(RBTree tree)
{
    if (tree == NULL)
        return NULL;
    while(tree->right != NULL)
        tree = tree->right;
    return tree;
}

int rbtree_maximum(RBRoot *root, int *val)
{
    Node *node;
    if (root)
        node = maximum(root->node);
    if (node == NULL)
        return -1;
    *val = node->key;
    return 0;
}
/*
 * 找结点(x)的后继结点。即，查找"红黑树中数据值大于该结点"的"最小结点"。
 */
static Node* rbtree_successor(RBTree x)
{
    // 如果 x 存在右孩子，则"x 的后继结点"为 "以其右孩子为根的子树的最小结点"。
    if (x->right != NULL)
        return minimum(x->right);
    // 如果 x 没有右孩子。则 x 有以下两种可能：
    // (01) x 是"一个左孩子"，则"x 的后继结点"为 "它的父结点"。
    // (02) x 是"一个右孩子"，则查找"x 的最低的父结点，并且该父结点要具有左孩子"，找到的
    这个"最低的父结点"就是"x 的后继结点"。
    Node* y = x->parent;
    while ((y!=NULL) && (x==y->right))
    {
        x = y;
        y = y->parent;
    }
}

```

```

    }
    return y;
}
/*
 * 找结点(x)的前驱结点。即，查找"红黑树中数据值小于该结点"的"最大结点"。
 */
static Node* rbtree_predecessor(RBTree x)
{
    // 如果 x 存在左孩子，则"x 的前驱结点"为 "以其左孩子为根的子树的最大结点"。
    if (x->left != NULL)
        return maximum(x->left);
    // 如果 x 没有左孩子。则 x 有以下两种可能：
    // (01) x 是"一个右孩子"，则"x 的前驱结点"为 "它的父结点"。
    // (01) x 是"一个左孩子"，则查找"x 的最低的父结点，并且该父结点要具有右孩子"，找到的
    这个"最低的父结点"就是"x 的前驱结点"。
    Node* y = x->parent;
    while ((y!=NULL) && (x==y->left))
    {
        x = y;
        y = y->parent;
    }
    return y;
}
/*
 * 对红黑树的节点(x)进行左旋转
 *
 * 左旋示意图(对节点 x 进行左旋):
 *
 *      px                                px
 *      /                                /
 *     x                                y
 *    / \    --(左旋)-->      / \    #
 *   lx  y                    x  ry
 *  /  \                    /  \
 * ly   ry                  lx  ly
 *
 *
 */
static void rbtree_left_rotate(RBRoot *root, Node *x)
{
    // 设置 x 的右孩子为 y
    Node *y = x->right;
    // 将 "y 的左孩子" 设为 "x 的右孩子";
    // 如果 y 的左孩子非空，将 "x" 设为 "y 的左孩子的父亲"
    x->right = y->left;
    if (y->left != NULL)
        y->left->parent = x;

```



```

// 将“x的父亲”设为“y的父亲”
y->parent = x->parent;
if (x->parent == NULL)
{
    //tree = y;           // 如果“x的父亲”是空节点，则将y设为根节点
    root->node = y;       // 如果“x的父亲”是空节点，则将y设为根节点
}
else
{
    if (x->parent->left == x)
        x->parent->left = y;    // 如果 x 是它父节点的左孩子，则将 y 设为“x 的父节点的
左孩子”
    else
        x->parent->right = y;    // 如果 x 是它父节点的左孩子，则将 y 设为“x 的父节点
的左孩子”
}
// 将“x”设为“y的左孩子”
y->left = x;
// 将“x的父节点”设为“y”
x->parent = y;
}
/*
* 对红黑树的节点(y)进行右旋转
*
* 右旋示意图(对节点 y 进行左旋):
*
*      py                      py
*      /                        /
*      y                        x
*     / \                    /  \
*    x  ry                  lx  y
*   / \                    /  \
*  lx  rx                  rx  ry
*
*      --(右旋)-->
*
*      #
*      #
*/
static void rbtree_right_rotate(RBRoot *root, Node *y)
{
    // 设置 x 是当前节点的左孩子。
    Node *x = y->left;
    // 将“x的右孩子”设为“y的左孩子”;
    // 如果“x的右孩子”不为空的话，将“y”设为“x的右孩子的父亲”
    y->left = x->right;
    if (x->right != NULL)
        x->right->parent = y;
    // 将“y的父亲”设为“x的父亲”
    x->parent = y->parent;
    if (y->parent == NULL)

```

```

{
    //tree = x;          // 如果 “y 的父亲” 是空节点，则将 x 设为根节点
    root->node = x;      // 如果 “y 的父亲” 是空节点，则将 x 设为根节点
}
else
{
    if (y == y->parent->right)
        y->parent->right = x;    // 如果 y 是它父节点的右孩子，则将 x 设为“y 的父节点
的右孩子”
    else
        y->parent->left = x;    // (y 是它父节点的左孩子) 将 x 设为“x 的父节点的左孩子”
    }
    // 将 “y” 设为 “x 的右孩子”
    x->right = y;
    // 将 “y 的父节点” 设为 “x”
    y->parent = x;
}
}
/*
* 红黑树插入修正函数
*
* 在向红黑树中插入节点之后(失去平衡)，再调用该函数；
* 目的是将它重新塑造成一颗红黑树。
*
* 参数说明：
*     root 红黑树的根
*     node 插入的结点          // 对应《算法导论》中的 z
*/
static void rbtree_insert_fixup(RBRoot *root, Node *node)
{
    Node *parent, *gparent;
    // 若“父节点存在，并且父节点的颜色是红色”
    while ((parent = rb_parent(node)) && rb_is_red(parent))
    {
        gparent = rb_parent(parent);
        // 若“父节点”是“祖父节点的左孩子”
        if (parent == gparent->left)
        {
            // Case 1 条件：叔叔节点是红色
            {
                Node *uncle = gparent->right;
                if (uncle && rb_is_red(uncle))
                {
                    rb_set_black(uncle);
                    rb_set_black(parent);
                    rb_set_red(gparent);
                    node = gparent;
                }
            }
        }
    }
}

```

```

        continue;
    }
}
// Case 2 条件：叔叔是黑色，且当前节点是右孩子
if (parent->right == node)
{
    Node *tmp;
    rbtree_left_rotate(root, parent);
    tmp = parent;
    parent = node;
    node = tmp;
}
// Case 3 条件：叔叔是黑色，且当前节点是左孩子。
rb_set_black(parent);
rb_set_red(gparent);
rbtree_right_rotate(root, gparent);
}
else // 若“z 的父节点”是“z 的祖父节点的右孩子”
{
    // Case 1 条件：叔叔节点是红色
    {
        Node *uncle = gparent->left;
        if (uncle && rb_is_red(uncle))
        {
            rb_set_black(uncle);
            rb_set_black(parent);
            rb_set_red(gparent);
            node = gparent;
            continue;
        }
    }
    // Case 2 条件：叔叔是黑色，且当前节点是左孩子
    if (parent->left == node)
    {
        Node *tmp;
        rbtree_right_rotate(root, parent);
        tmp = parent;
        parent = node;
        node = tmp;
    }
    // Case 3 条件：叔叔是黑色，且当前节点是右孩子。
    rb_set_black(parent);
    rb_set_red(gparent);
    rbtree_left_rotate(root, gparent);
}
}

```

```

    // 将根节点设为黑色
    rb_set_black(root->node);
}
/*
 * 添加节点：将节点(node)插入到红黑树中
 *
 * 参数说明：
 *     root 红黑树的根
 *     node 插入的结点          // 对应《算法导论》中的 z
 */
static void rbtree_insert(RBRoot *root, Node *node)
{
    Node *y = NULL;
    Node *x = root->node;

    // 1. 将红黑树当作一颗二叉查找树，将节点添加到二叉查找树中。
    while (x != NULL)
    {
        y = x;
        if (node->key < x->key)
            x = x->left;
        else
            x = x->right;
    }
    rb_parent(node) = y;

    if (y != NULL)
    {
        if (node->key < y->key)
            y->left = node;          // 情况 2: 若“node 所包含的值” < “y 所包含的值”，
    则将 node 设为“y 的左孩子”
        else
            y->right = node;         // 情况 3: (“node 所包含的值” >= “y 所包含的值”)
    将 node 设为“y 的右孩子”
    }
    else
    {
        root->node = node;          // 情况 1: 若 y 是空节点，则将 node 设为根
    }
    // 2. 设置节点的颜色为红色
    node->color = RED;
    // 3. 将它重新修正为一颗二叉查找树
    rbtree_insert_fixup(root, node);
}
/*
 * 创建结点

```

```

*
* 参数说明:
*      key 是键值。
*      parent 是父结点。
*      left 是左孩子。
*      right 是右孩子。
*/
static Node* create_rbtree_node(Type key, Node *parent, Node *left, Node* right)
{
    Node* p;
    if ((p = (Node *)malloc(sizeof(Node))) == NULL)
        return NULL;
    p->key = key;
    p->left = left;
    p->right = right;
    p->parent = parent;
    p->color = BLACK; // 默认为黑色
    return p;
}
/*
* 新建结点(节点键值为 key), 并将其插入到红黑树中
*
* 参数说明:
*      root 红黑树的根
*      key 插入结点的键值
* 返回值:
*      0, 插入成功
*      -1, 插入失败
*/
int insert_rbtree(RBRoot *root, Type key)
{
    Node *node;    // 新建结点
    // 不允许插入相同键值的节点。
    // (若想允许插入相同键值的节点, 注释掉下面两句话即可!)
    if (search(root->n, key) != NULL)
        return -1;
    // 如果新建结点失败, 则返回。
    if ((node=create_rbtree_node(key, NULL, NULL, NULL)) == NULL)
        return -1;
    rbtree_insert(root, node);
    return 0;
}
/*
* 红黑树删除修正函数
*
* 在从红黑树中删除插入节点之后(红黑树失去平衡), 再调用该函数;

```

```

* 目的是将它重新塑造成一颗红黑树。
*
* 参数说明:
*     root 红黑树的根
*     node 待修正的节点
*/
static void rbtree_delete_fixup(RBRoot *root, Node *node, Node *parent)
{
    Node *other;
    while ((!node || rb_is_black(node)) && node != root->node)
    {
        if (parent->left == node)
        {
            other = parent->right;
            if (rb_is_red(other))
            {
                // Case 1: x 的兄弟 w 是红色的
                rb_set_black(other);
                rb_set_red(parent);
                rbtree_left_rotate(root, parent);
                other = parent->right;
            }
            if ((!other->left || rb_is_black(other->left)) &&
                (!other->right || rb_is_black(other->right)))
            {
                // Case 2: x 的兄弟 w 是黑色，且 w 的两个孩子也都是黑色的
                rb_set_red(other);
                node = parent;
                parent = rb_parent(node);
            }
            else
            {
                if (!other->right || rb_is_black(other->right))
                {
                    // Case 3: x 的兄弟 w 是黑色的，并且 w 的左孩子是红色，右孩子为黑色。
                    rb_set_black(other->left);
                    rb_set_red(other);
                    rbtree_right_rotate(root, other);
                    other = parent->right;
                }
                // Case 4: x 的兄弟 w 是黑色的；并且 w 的右孩子是红色的，左孩子任意颜色。
                rb_set_color(other, rb_color(parent));
                rb_set_black(parent);
                rb_set_black(other->right);
                rbtree_left_rotate(root, parent);
                node = root->node;
            }
        }
    }
}

```

```

        break;
    }
}
else
{
    other = parent->left;
    if (rb_is_red(other))
    {
        // Case 1: x 的兄弟 w 是红色的
        rb_set_black(other);
        rb_set_red(parent);
        rbtree_right_rotate(root, parent);
        other = parent->left;
    }
    if ((!other->left || rb_is_black(other->left)) &&
        (!other->right || rb_is_black(other->right)))
    {
        // Case 2: x 的兄弟 w 是黑色，且 w 的两个孩子也都是黑色的
        rb_set_red(other);
        node = parent;
        parent = rb_parent(node);
    }
    else
    {
        if (!other->left || rb_is_black(other->left))
        {
            // Case 3: x 的兄弟 w 是黑色的，并且 w 的左孩子是红色，右孩子为黑色。
            rb_set_black(other->right);
            rb_set_red(other);
            rbtree_left_rotate(root, other);
            other = parent->left;
        }
        // Case 4: x 的兄弟 w 是黑色的；并且 w 的右孩子是红色的，左孩子任意颜色。
        rb_set_color(other, rb_color(parent));
        rb_set_black(parent);
        rb_set_black(other->left);
        rbtree_right_rotate(root, parent);
        node = root->node;
        break;
    }
}
}
if (node)
    rb_set_black(node);
}
/*

```

```

* 删除结点
*
* 参数说明:
*     tree 红黑树的根结点
*     node 删除的结点
*/
void rbtree_delete(RBRoot *root, Node *node)
{
    Node *child, *parent;
    int color;
    // 被删除节点的"左右孩子都不为空"的情况。
    if ( (node->left!=NULL) && (node->right!=NULL) )
    {
        // 被删节点的后继节点。(称为"取代节点")
        // 用它来取代"被删节点"的位置, 然后再将"被删节点"去掉。
        Node *replace = node;
        // 获取后继节点
        replace = replace->right;
        while (replace->left != NULL)
            replace = replace->left;
        // "node 节点"不是根节点(只有根节点不存在父节点)
        if (rb_parent(node))
        {
            if (rb_parent(node)->left == node)
                rb_parent(node)->left = replace;
            else
                rb_parent(node)->right = replace;
        }
        else
            // "node 节点"是根节点, 更新根节点。
            root->node = replace;
        // child 是"取代节点"的右孩子, 也是需要"调整的节点"。
        // "取代节点"肯定不存在左孩子! 因为它是一个后继节点。
        child = replace->right;
        parent = rb_parent(replace);
        // 保存"取代节点"的颜色
        color = rb_color(replace);
        // "被删除节点"是"它的后继节点的父节点"
        if (parent == node)
        {
            parent = replace;
        }
        else
        {
            // child 不为空
            if (child)

```



```

        rb_set_parent(child, parent);
        parent->left = child;
        replace->right = node->right;
        rb_set_parent(node->right, replace);
    }
    replace->parent = node->parent;
    replace->color = node->color;
    replace->left = node->left;
    node->left->parent = replace;
    if (color == BLACK)
        rbtree_delete_fixup(root, child, parent);
    free(node);
    return ;
}
if (node->left != NULL)
    child = node->left;
else
    child = node->right;
parent = node->parent;
// 保存"取代节点"的颜色
color = node->color;
if (child)
    child->parent = parent;
// "node 节点"不是根节点
if (parent)
{
    if (parent->left == node)
        parent->left = child;
    else
        parent->right = child;
}
else
    root->node = child;
if (color == BLACK)
    rbtree_delete_fixup(root, child, parent);
free(node);
}
/*
 * 删除键值为 key 的结点
 *
 * 参数说明:
 *     tree 红黑树的根结点
 *     key 键值
 */
void delete_rbtree(RBRoot *root, Type key)
{

```

```

    Node *z, *node;
    if ((z = search(root->node, key)) != NULL)
        rbtree_delete(root, z);
}
/*
 * 销毁红黑树
 */
static void rbtree_destroy(RBTree tree)
{
    if (tree==NULL)
        return ;
    if (tree->left != NULL)
        rbtree_destroy(tree->left);
    if (tree->right != NULL)
        rbtree_destroy(tree->right);
    free(tree);
}

void destroy_rbtree(RBRoot *root)
{
    if (root != NULL)
        rbtree_destroy(root->node);
    free(root);
}
/*
 * 打印"红黑树"
 *
 * tree      -- 红黑树的节点
 * key       -- 节点的键值
 * direction  -- 0, 表示该节点是根节点;
 *             -1, 表示该节点是它的父结点的左孩子;
 *             1, 表示该节点是它的父结点的右孩子。
 */
static void rbtree_print(RBTree tree, Type key, int direction)
{
    if(tree != NULL)
    {
        if(direction==0)    // tree 是根节点
            printf("%2d(B) is root\n", tree->key);
        else                // tree 是分支节点
            printf("%2d(%s) is %2d's %6s child\n", tree->key, rb_is_red(tree)?"R":"B", key,
direction==1?"right" : "left");
        rbtree_print(tree->left, tree->key, -1);
        rbtree_print(tree->right, tree->key, 1);
    }
}

```

```

void print_rbtree(RBRoot *root)
{
    if (root!=NULL && root->node!=NULL)
        rbtree_print(root->node, root->node->key, 0);
}

```

### 3) 、 test.cpp

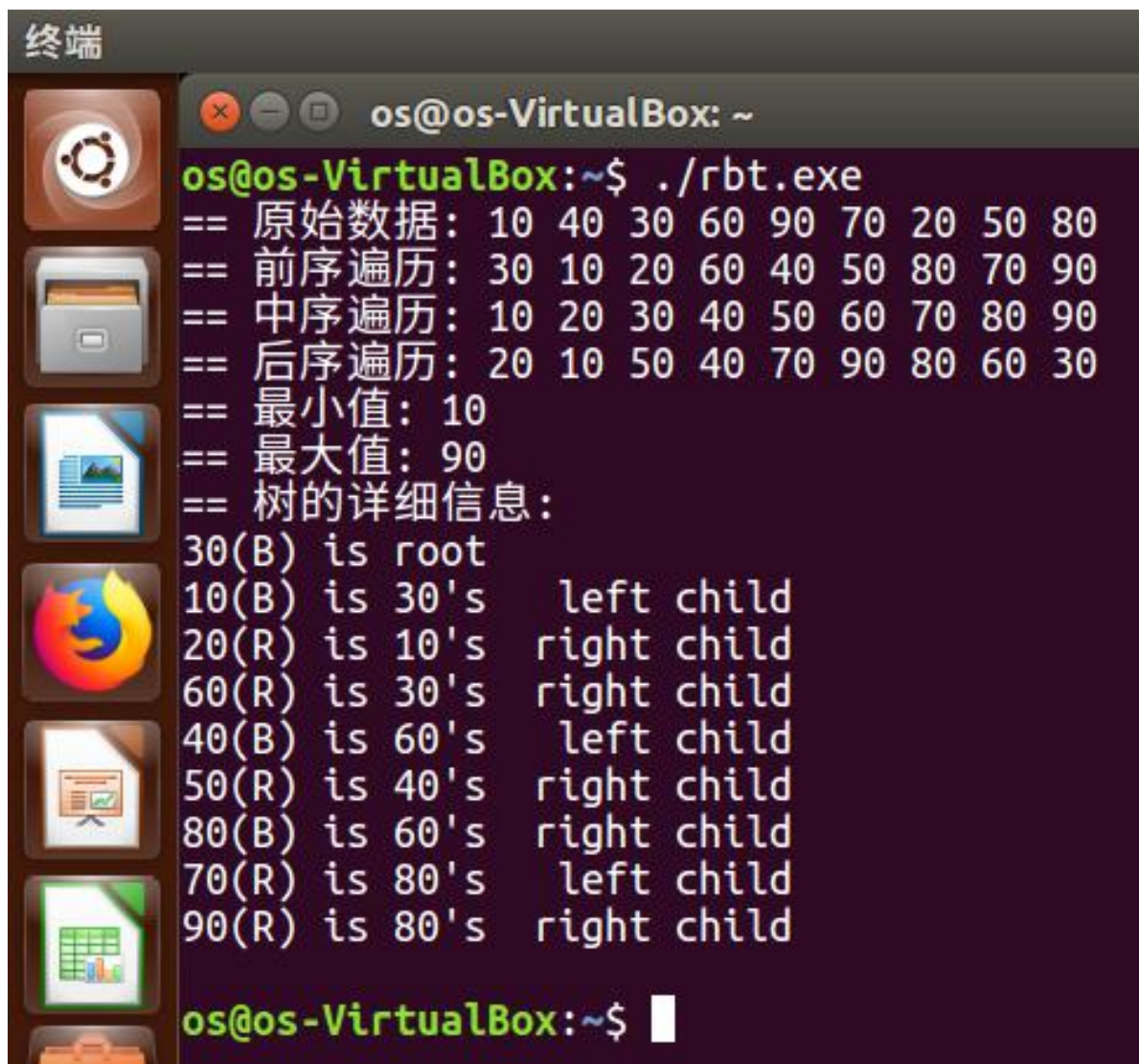
```

#include <stdio.h>
#include "RedBlackTree.h"
#define CHECK_INSERT 0    // "插入"动作的检测开关(0, 关闭; 1, 打开)
#define CHECK_DELETE 0   // "删除"动作的检测开关(0, 关闭; 1, 打开)
#define LENGTH(a) ( (sizeof(a)) / (sizeof(a[0])) )
int main()
{
    int a[] = {10, 40, 30, 60, 90, 70, 20, 50, 80};
    int i, ilen=LENGTH(a);
    RBRoot *root=NULL;
    root = create_rbtree();
    printf("== 原始数据: ");
    for(i=0; i<ilen; i++)
        printf("%d ", a[i]);
    printf("\n");
    for(i=0; i<ilen; i++)
    {
        insert_rbtree(root, a[i]);
    }
    #if CHECK_INSERT
        printf("== 添加节点: %d\n", a[i]);
        printf("== 树的详细信息: \n");
        print_rbtree(root);
        printf("\n");
    #endif
    }
    printf("== 前序遍历: ");
    preorder_rbtree(root);
    printf("\n== 中序遍历: ");
    inorder_rbtree(root);
    printf("\n== 后序遍历: ");
    postorder_rbtree(root);
    printf("\n");
    if (rbtree_minimum(root, &i)==0)
        printf("== 最小值: %d\n", i);
    if (rbtree_maximum(root, &i)==0)
        printf("== 最大值: %d\n", i);
    printf("== 树的详细信息: \n");
    print_rbtree(root);
}

```

```
    printf("\n");
#if CHECK_DELETE
    for(i=0; i<ilen; i++)
    {
        delete_rbtree(root, a[i]);
        printf("== 删除节点: %d\n", a[i]);
        if (root)
        {
            printf("== 树的详细信息: \n");
            print_rbtree(root);
            printf("\n");
        }
    }
#endif
    destroy_rbtree(root);
return 0;
}
```

## (2) 实验结果



```
终端
os@os-VirtualBox: ~
os@os-VirtualBox:~$ ./rbt.exe
== 原始数据: 10 40 30 60 90 70 20 50 80
== 前序遍历: 30 10 20 60 40 50 80 70 90
== 中序遍历: 10 20 30 40 50 60 70 80 90
== 后序遍历: 20 10 50 40 70 90 80 60 30
== 最小值: 10
== 最大值: 90
== 树的详细信息:
30(B) is root
10(B) is 30's left child
20(R) is 10's right child
60(R) is 30's right child
40(B) is 60's left child
50(R) is 40's right child
80(B) is 60's right child
70(R) is 80's left child
90(R) is 80's right child
os@os-VirtualBox:~$
```

## 五、实验小结

本次实验，我不仅仅学会了在 Linux 环境下编写程序，更了解数据结构的四种结构。Linux 和 Windows 在编程上大部分是一样的，但还有些许不同之处，例如 Linux 下 main（）函数必须是 int，而不能是 void 的。

在 github 上提交的方式，更是使我学会一种新的工具。收获很多。