# Linux实验

|  |  |  |
| --- | --- | --- |
| 班级 | 姓名 | 学号 |
| 硬件二班 | 张东琪 | 161403204 |

堆排序

       堆排序是利用堆的性质进行的一种选择排序。下面先讨论一下堆。

1.堆

  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]<=key[2i+1]&&Key[i]<=key[2i+2]或者Key[i]>=Key[2i+1]&&key>=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]>=Key[2i+1]&&key>=key[2i+2]称为大顶堆，满足 Key[i]<=key[2i+1]&&Key[i]<=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.堆排序的思想

   利用大顶堆(小顶堆)堆顶记录的是最大关键字(最小关键字)这一特性，使得每次从无序中选择最大记录(最小记录)变得简单。

    其基本思想为(大顶堆)：

    1)将初始待排序关键字序列(R1,R2....Rn)构建成大顶堆，此堆为初始的无须区；

    2)将堆顶元素R[1]与最后一个元素R[n]交换，此时得到新的无序区(R1,R2,......Rn-1)和新的有序区(Rn),且满足R[1,2...n-1]<=R[n];

    3)由于交换后新的堆顶R[1]可能违反堆的性质，因此需要对当前无序区(R1,R2,......Rn-1)调整为新堆，然后再次将R[1]与无序区最后一个元素交换，得到新的无序区(R1,R2....Rn-2)和新的有序区(Rn-1,Rn)。不断重复此过程直到有序区的元素个数为n-1，则整个排序过程完成。

    操作过程如下：

     1)初始化堆：将R[1..n]构造为堆；

     2)将当前无序区的堆顶元素R[1]同该区间的最后一个记录交换，然后将新的无序区调整为新的堆。

    因此对于堆排序，最重要的两个操作就是构造初始堆和调整堆，其实构造初始堆事实上也是调整堆的过程，只不过构造初始堆是对所有的非叶节点都进行调整。

#include<stdio.h>

typedef int ElementType;

int arr1[11]={0,2,87,39,49,34,62,53,6,44,98};

#define LeftChild(i) (2 \* (i) + 1)

void Swap(int\* a,int\* b)

{

int temp=\*a;

\*a=\*b;

\*b=temp;

}

void PercDown(int A[], int i, int N)

{

int child;

ElementType Tmp;

for (Tmp = A[i]; 2\*i+1 < N; i = child){

child = 2\*i+1; //注意数组下标是从0开始的，所以左孩子的求发不是2\*i

if (child != N - 1 && A[child + 1] > A[child])

++child; //找到最大的儿子节点

if (Tmp < A[child])

A[i] = A[child];

else

break;

}

A[i] = Tmp;

}

void HeapSort(int A[], int N)

{

int i;

for (i = N / 2; i >= 0; --i)

PercDown(A, i, N); //构造堆

for(i=N-1;i>0;--i)

{

Swap(&A[0],&A[i]); //将最大元素（根）与数组末尾元素交换，从而删除最大元素，重新构造堆

PercDown(A, 0, i);

}

}

void Print(int A[],int N)

{

int i;

for(i=0;i<N;i++)

{

printf(" %d ",A[i]);

}

}

int main()

{

int arr[10]={2,87,39,49,34,62,53,6,44,98};

Print(arr,10);

printf("\n");

HeapSort(arr,10);

Print(arr,10);

printf("\n");

return 0;

}

查找树

在二叉排序树b中查找x的过程为：

若b是空树，则搜索失败，否则：

若x等于b的根结点的数据域之值，则查找成功；否则：

若x小于b的根结点的数据域之值，则搜索左子树；否则：

查找右子树。

Status SearchBST(BiTree T, KeyType key, BiTree f, BiTree &\*p){

//在根[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88)T所指[二叉排序树](https://baike.baidu.com/item/%E4%BA%8C%E5%8F%89%E6%8E%92%E5%BA%8F%E6%A0%91)中[递归](https://baike.baidu.com/item/%E9%80%92%E5%BD%92)地查找其关键字等于key的[数据元素](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%85%83%E7%B4%A0)，若查找成功，

//则[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88)p指向该[数据元素](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E5%85%83%E7%B4%A0)结点，并返回TRUE，否则指针指向查找[路径](https://baike.baidu.com/item/%E8%B7%AF%E5%BE%84)上访问的最后

//一个结点并返回FALSE，[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88)f指向T的双亲，其初始调用值为NULL

if(!T){ p=f; return FALSE;} //查找不成功

else if EQ(key, T->data.key) {P=T; return TRUE;} //查找成功

else if LT(key,T->data.key)

return SearchBST(T->lchild, key, T, p); //在左子树中继续查找

else return SearchBST(T->rchild, key, T, p); //在右子树中继续查找

#include <stdio.h>

#include <stdlib.h>

#include "biTree.h"

#define N 10

void printNode(BiTreeNode \*node);

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

BiTreeNode \*root;

int i;

root = NULL;

int data[N] = {10, 23, 11, 98, 111, 87, 34, 11, 33, 8};

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

insertNode(&root, data[i]);

}

printf("before delete:\n");

inorderTraversal(&root, printNode);

printf("\n");

deleteNode(&root, 11);

deleteNode(&root, 8);

printf("after delete:\n");

inorderTraversal(&root, printNode);

printf("\n");

exit(0);

}

void printNode(BiTreeNode \*node) {

printf("%d\t", node->data);

}

栈解析算术表达式  
分析用堆栈解析算术表达式的基本方法。给出的示例代码能解析任何包括+，-，\*，/，()和0到9数字组成的算术表达式。   
2 中缀表达式和后缀表达式   
中缀表达式就是通常所说的算术表达式，比如(1+2)\*3-4。   
后缀表达式是指通过解析后，运算符在运算数之后的表达式，比如上式解析成后缀表达式就是12+3\*4-。这种表达式可以直接利用栈来求解。   
3 运算符的优先级   
优先级 运算符   
1 括号()   
2 负号-   
3 乘方\*\*   
4 乘\*，除/，求余%   
5 加+，减-   
6 小于<，小于等于<=，大于>，大于等于>=   
7 等于==，不等于!=   
8 逻辑与&&   
9 逻辑或||   
大致的规律是，一元运算符 > 二元运算符 > 多元运算符。   
4 利用堆栈解析算术表达式的过程   
中缀表达式翻译成后缀表达式的方法如下：   
（1）从右向左依次取得数据ch。   
（2）如果ch是操作数，直接输出。   
（3）如果ch是运算符（含左右括号），则：   
      a：如果ch = '('，放入堆栈。   
      b：如果ch = ')'，依次输出堆栈中的运算符，直到遇到'('为止。   
      c：如果ch不是')'或者'('，那么就和堆栈顶点位置的运算符top做优先级比较。   
          1：如果ch优先级比top高，那么将ch放入堆栈。   
          2：如果ch优先级低于或者等于top，那么输出top，然后将ch放入堆栈。   
（4）如果表达式已经读取完成，而堆栈中还有运算符时，依次由顶端输出。   
如果我们有表达式(A-B)\*C+D-E/F，要翻译成后缀表达式，并且把后缀表达式存储在一个名叫output的字符串中，可以用下面的步骤。   
（1）读取'('，压入堆栈，output为空   
（2）读取A，是运算数，直接输出到output字符串，output = A   
（3）读取'-'，此时栈里面只有一个'('，因此将'-'压入栈，output = A   
（4）读取B，是运算数，直接输出到output字符串，output = AB   
（5）读取')'，这时候依次输出栈里面的运算符'-'，然后就是'('，直接弹出，output = AB-   
（6）读取'\*'，是运算符，由于此时栈为空，因此直接压入栈，output = AB-   
（7）读取C，是运算数，直接输出到output字符串，output = AB-C   
（8）读取'+'，是运算符，它的优先级比'\*'低，那么弹出'\*'，压入'+"，output = AB-C\*   
（9）读取D，是运算数，直接输出到output字符串，output = AB-C\*D   
（10）读取'-'，是运算符，和'+'的优先级一样，因此弹出'+'，然后压入'-'，output = AB-C\*D+   
（11）读取E，是运算数，直接输出到output字符串，output = AB-C\*D+E   
（12）读取'/'，是运算符，比'-'的优先级高，因此压入栈，output = AB-C\*D+E   
（13）读取F，是运算数，直接输出到output字符串，output = AB-C\*D+EF   
（14）原始字符串已经读取完毕，将栈里面剩余的运算符依次弹出，output = AB-C\*D+EF/-   
5 计算算术表达式   
当有了后缀表达式以后，运算表达式的值就非常容易了。可以按照下面的流程来计算。   
（1）从左向右扫描表达式，一个取出一个数据data   
（2）如果data是操作数，就压入堆栈   
（3）如果data是操作符，就从堆栈中弹出此操作符需要用到的数据的个数，进行运算，然后把结果压入堆栈   
（4）如果数据处理完毕，堆栈中最后剩余的数据就是最终结果

#include "stdio.h"

#include<stdlib.h>

#define maxsize 100

#define overflow 1

#define ok 1

#define error 1

#define SElemType int

#define Status int

typedef struct

{

SElemType \*base;

SElemType \*top;

int stacksize;

}SqStack;

Status InitStack(SqStack &S) //顺序栈的初始化

{

//构造一个空栈S

S.base=new SElemType[maxsize]; //为顺序栈动态分配一个最大容量为maxsize的数组空间

if(!S.base) exit(overflow); //存储分配失败

S.top=S.base; //top初始为base，空栈

S.stacksize=maxsize; //stacksize置为栈的最大容量maxsize

return ok;

}

Status Push(SqStack &S,SElemType e) //顺序栈的入栈

{

//插入元素e为新的栈顶元素

if(S.top-S.base==S.stacksize) return error;//栈满

\*S.top++=e; //元素e压入栈顶，栈顶指针加1

return ok;

}

Status Pop(SqStack &S,SElemType &e) //顺序栈的出栈

{

if(S.top==S.base) return error; //栈空

e=\*--S.top; //栈顶指针减1,将栈顶元素赋给e

return ok;

}

SElemType GetTop(SqStack S) //取栈顶元素

{

//返回S的栈顶元素，不修改栈顶指针

if(S.top!=S.base) //栈非空

return \*(S.top-1); //返回栈顶元素的值，栈顶指针不变

}

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

{

return 0;

}