**linux程序实验报告**

1. 实验题目：

数据结构（红黑树、堆、栈、二叉搜索树、队列、链表、图）的c代码实现。

1. 实验环境：

linux系统、dev C++。

1. 实验要求：

要求实现至少四项数据结构，并熟悉其实现机制。

1. 实验内容：
2. 首先，我做了一个菜单选项，用于实现选择机制的main函数，即想要执行那种数据结构就选择对应的选项，这部分代码很简单，用了一个while函数实现执行一次某一个选项后还能在返回菜单界面；至于选择的实现通过 switch（）即可实现：

int main()

{

int x;

while (1)

{

printf("-------------操作选项-----------\n");

printf("1红黑树\t\t2、堆\n");

printf("3、栈\t\t4、搜索树\n");

printf("5、队列\t\t6、链表\n");

printf("7、图 \n");

printf("8:退出程序 \n");

printf("--------------------------------\n");

printf("请按数字键选择要执行的操作: ");

scanf("%d",&x);

printf("\n");

//输入8跳出循环，退出程序

if(x==8)

break;

switch(x)

{

case 1:rbtree();

break; //输入1，跳出switch语句，进入下一次循环

case 2:heap();

break;

case 3:stack1();

break;

case 4:searchtree();

break;

case 5:quene();

break;

case 6:linklist();

break;

case 7:graph();

break;

default: //数字输入错误，跳出siwtch语句，进入下一次循环

printf("输入的数字不正确\n");

break;

}

}

return 0;

}

2、红黑树（rbtree）：

1. B Tree，全称是Red-Black Tree，又称为“红黑树”，它一种特殊的二叉查找树。红黑树的每个节点上都有存储位表示节点的颜色，可以是红(Red)或黑(Black)。

**红黑树的特性**:  
**（1）每个节点或者是黑色，或者是红色。**  
**（2）根节点是黑色。**  
**（3）每个叶子节点（NIL）是黑色。** **[注意：这里叶子节点，是指为空(NIL 或NULL)的叶子节点！]**  
**（4）如果一个节点是红色的，则它的子节点必须是黑色的。**  
**（5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。注意**：  
(01) 特性(3)中的叶子节点，是只为空(NIL或null)的节点。  
(02) 特性(5)，确保没有一条路径会比其他路径长出俩倍。因而，红黑树是相对是接近平衡的二叉树。

红黑树的应用比较广泛，主要是用它来存储有序的数据，它的时间复杂度是O(lgn)，效率非常之高。

红黑树特有的操作是左旋和右旋，目的是将节点多的一支出让节点给另一个节点少的一支，旋转操作在插入和删除操作中经常会用到。



3、堆排序

堆是一个完全二叉树，完全二叉树即是：若设二叉树的深度为h，除第 h 层外，其它各层 (1～h-1) 的结点数都达到最大个数，第 h 层所有的结点都连续集中在最左边，这就是完全二叉树。堆分为最小堆和最大堆。最大堆就是每个父节点的数值要大于孩子节点，最小堆就是每个父节点的数值要小于孩子节点。排序要求从小到大的话，我们需要建立最大堆，反之建立最小堆。

堆排序后的顺序为从小到大，因此需要建立最大堆把数组建成为最大堆

第一个非叶子节点的位置序号为len/2-1，堆顶元素和最后一个元素交换位置，这样最后的一个位置保存的是最大的数，每次循环依次将次大的数值在放进其前面一个位置，这样得到的顺序就是从小到大，将arr[0...i-1]重新调整为最大堆。

堆排序在排序元素较少时有点大才小用，待排序列元素较多时，堆排序还是很有效的。另外，堆排序在最坏情况下，时间复杂度也为O（n\*logn）。相对于快速排序（平均时间复杂度为O（n\*logn），最坏情况下为O（n\*n）），这是堆排序的最大优点。

4、栈

栈作为一种数据结构，是一种只能在一端进行插入和删除操作的特殊线性表。它按照后进先出的原则存储数据，先进入的数据被压入栈底，最后的数据在栈顶，需要读数据的时候从栈顶开始弹出数据（最后一个数据被第一个读出来）。栈具有记忆作用，对栈的插入与删除操作中，不需要改变栈底指针。

栈是允许在同一端进行插入和删除操作的特殊线性表。允许进行插入和删除操作的一端称为栈顶(top)，另一端为栈底(bottom)；栈底固定，而栈顶浮动；栈中元素个数为零时称为空栈。插入一般称为进栈（PUSH），删除则称为退栈（POP）。栈也称为后进先出表。

（1）输入的形式：表达式，例如2\*(3+4)

包含的运算符只能有'+' 、'-' 、'\*' 、'/' 、'('、 ')'；

输出的形式：运算结果，例如2\*(3+4)=14；

程序所能达到的功能：对表达式求值并输出

（2）栈的抽象数据类型定义：

ADT Stack{

数据对象：D={ai|ai∈ElemSet,i=1,2,…,n,n≥0}

数据关系：R1={<ai-1,ai>|ai-1,ai∈D,i=2,…,n}

      约定an端为栈顶，ai端为栈底

基本操作：

Push(&S,e)

初始条件：栈S已存在

操作结果：插入元素e为新的栈顶元素

Pop(&S,&e)

初始条件：栈S已存在且非空

操作结果：删除S的栈顶元素，并用e返回其值

}ADT Stack

（3）、各个模块的主要功能：

\*Push(SC \*s,char c)：把字符压栈

\*Push(SF \*s,float f)：把数值压栈

\*Pop(SC \*s)：把字符退栈

\*Pop(SF \*s)：把数值退栈

Operate(a,theta,b)：根据theta对a和b进行'+' 、'-' 、'\*' 、'/' 、'^'操作

In(Test,\*TestOp)：若Test为运算符则返回true，否则返回false

ReturnOpOrd(op,\*TestOp)：若Test为运算符，则返回此运算符在数组中的下标precede(Aop,Bop)：根据运算符优先级表返回Aop与Bop之间的优先级

EvaluateExpression(\*MyExpression)：用算符优先法对算术表达式求值

5、二叉查找树

二叉查找树是一种动态查找表，具有这些性质：

（1）若它的左子树不为空，则左子树上的所有节点的值都小于它的根节点的值；

（2）若它的右子树不为空，则右子树上所有节点的值都大于它的根节点的值；

（3）其他的左右子树也分别为二叉查找树；

（4）二叉查找树是动态查找表，在查找的过程中可见添加和删除相应的元素，在这些操作中需要保持二叉查找树的以上性质。

（5）、一般来说，定义树是用递归的方式，树是一些节点的集合。集合可以是空的，也可以是非空。如果不是非空，那么就是由一个根节点和0个或者多个的非空节点的字树组成，子树都由一一条来自根节点的有向的边所连结。一颗树是N个节点和N-1条边的集合。没有子的节点被称为树叶。N1到Ni的路径定义为路径。根节点到Ni的唯一路径称为深度。Ni的高是指从它到一片树叶的最长的路径。

二叉查找树是特殊的二叉树，假设x为二叉树中的任意一个结点，x结点包含关键字key，节点x的key值为key[x]。如果y是x的左子树中的一个节点，则key[y]<=key[x];如果y是x的右子树的一个节点，则key[y]>=key[x]。

6、队列

队列是一种操作受限的线性表，其限制条件为允许在表的一端进行插入，而在表的另一端进行删除。插入的一端叫做队尾，删除的一端叫做队头。向队列中插入新元素的行为称为进队，从队列中删除元素的行为称为出队。

队列的特点是先进先出。

EnQueue()为进队列函数，从队尾（tail）进入；

DeQueue()为出队列函数，从队头（head）出去；这样就实现了先进先出的原则。

7、链表

如果链表为空，则创建一个链表，指针域指向自己，否则寻找尾节点，

将尾节点的指针域指向这个新节点，新节点的指针域指向头结点。如果输入链表是空。则创建一个新的节点，使其next指针指向自己(\*head)->next=\*head

输入的链表不是空的，寻找链表的尾节点，使尾节点的next=新节点。新节点的next指向头节点，尾节点指向新节点。

（1）、向链表中插入结点

链表中插入头结点，根据插入位置的不同，分为3种：

* 插入到链表的首部，也就是头结点和首元结点中间；
* 插入到链表中间的某个位置；
* 插入到链表最末端；

（2）、从链表中删除节点

当需要从链表中删除某个结点时，需要进行两步操作：

* 将结点从链表中摘下来;
* 手动释放掉结点，回收被结点占用的内存空间;

8、图

图（graph），它表明了物件与物件之间的“多对多”的一种复杂关系。图包含了两个基本元素：顶点（vertex, 简称V）和边（edge,简称E）。

（1）、有向图与无向图

　　如果给图的每条边规定一个方向，那么得到的图称为有向图。在有向图中，从一个顶点出发的边数称为该点的出度，而指向一个顶点的边数称为该点的入度。相反，边没有方向的图称为无向图。

（2）、有权图与无权图

　　如果图中的边有各自的权重，得到的图是有权图。比如地铁路线图，连接两站的边的权重可以是距离，也可以是价格，或者其他。反之，如果图的边没有权重，或者权重都一样（即没有区分），称为无权图。

（3）、连通图

　　如果图中任意两点都是连通的，那么图被称作连通图。图的连通性是图的基本性质。无向图中的一个极大连通子图称为其的一个连通分量。有向图中，如果对任意两个顶点 V\_i 与 V\_j 都存在 i 到 j 以及 j 到 i 的路径，则称其为强连通图，对应有强连通分量的概念。

（4）、邻接矩阵

采用一个大小为 V\times V 的矩阵 G ，对于有权图， G\_{ij} 可以表示V\_i 到 V\_j的边的权重，如果是无权图，则可设为1表示存在边，0表示不存在边。因此邻接矩阵的表示相当的直观，而且对于查找某一条边是否存在、权重多少非常快。但其比较浪费空间，对稠密图（ E>>V ）来说，会比较适合。

一般用邻接矩阵或邻接表实现图结构，图的邻接矩阵存储方式是用两个数组来表示图。一个一维数组存储图中顶点信息，一个二维数组（邻接矩阵）存储图中的边或弧的信息。