知识点总结

2018年1月8日　21:39

一、程序性能，算法和渐进记法

1. 时间复杂度：运行程序需要的时间，分析方法
   a. 找出一个或多个关键操作，确定他们的执行时间；
   b. 确定程序总的步数

例 2-7[最大元素] 程序 1-37 的返回值是数组 a[0:n−1] 中最大元素的位置。我们可以根据数组元素之间的比较次数来估算时间复杂度。当 n ≤ 0 时，异常抛出，比较次数为 0。当 n=1 时，没有进入 for 循环体，比较次数还是 0。当 n>1 时，for 循环的每一次迭代都执行一次比较，比较次数为 n−1。因此总的比较次数是 max{n−1,0}。函数 indexOfMax 还执行了其他

2. 空间复杂度：程序运行所需内存的大小，需要的空间主要由指令空间，数据空间，环境栈空间三部分组成，实际分析中主要看程序的实例特征S_p需要的空间

例 2-5[阶乘] 考虑程序 1-29 的阶乘函数。它的空间复杂度是 n 的函数而不是输入（只有一个）或输出（也只有一个）个数的函数。递归深度是 max{n,1}。每次调用函数 factorial，递归栈都需要保留返回地址（4 字节）和 n 的值（4 字节）。此外没有其他依赖于 n 的空间，因此 $S_{\text{factorial}}(n)=8*\max\{n,1\}$。 ■

3. 统计一段程序中某语句执行的次数

例 2-9[名次计算（ranking）] 一个元素在一个序列中的**名次**（rank）是所有比它小的元素个数加上在它左边出现的与它相同的元素个数。例如，数组 a=[4,3,9,3,7] 是一个序列，各元素的名次为 r=[2,0,4,1,3]。程序 2-5 的函数 rank 计算数组 a 的各元素的名次。rank 的时间复杂度根据 a 的元素比较次数来估算。比较操作是由 if 语句来完成的。对于每一个 i 的值，比较次数为 i，因此总的比较次数为 1+2+3+⋯+n−1=（n−1）n/2（见公式（1-3））。

程序 2-5　名次计算

```
template<class T>
void rank(T a[], int n, int r[])
{// 给数组 a[0:n-1] 的 n 个元素排名次
 // 结果在 r[0:n-1] 中返回
   for (int i = 0; i < n; i++)
      r[i] = 0;                    // 初始化

   // 比较所有元素对
   for (int i = 1; i < n; i++)
      for (int j = 0; j < i; j++)
         if (a[j] <= a[i]) r[i]++;
         else r[j]++;
}
```

注意，在估算时间复杂度时，没有考虑 for 循环的经常性用时、数组 r 初始化的用时以及每次 a 的两个元素比较时 r 增值操作的用时。 ■

4. 根据一个函数大O记法算复杂度
5. 设文件F={13, 3, 38, 1, 30, 28, 7, 24, 10}，画出利用冒泡法对其进行排序的过程。
6. 写出对数列13,7,9,15,8,16,12,11,20,10进行堆排序的过程。

二、链表结构

1. 链表顺序存储和链接存储的优缺点
    a. 存储空间（空间复杂度）（指针和创建）
        i. 顺序存储不需要为表示结点间的逻辑关系而增加额外的存储空间，而链接存储需要为指针增加存储空间
        ii. 顺序存储空间需要事先申请，空闲空间不能共享，且难以扩充，而链接存储没有空闲空间，所有的空间都被有效利用，并可以随时扩充，不需要的空间都被及时销毁
    b. 操作（时间复杂度）（访问，插入删除，搜索）
        i. 顺序存储可以方便的随机存取表中的任一结点。而链接存储每次存取都需要一次到所需节点的遍历（在访问元素操作上，顺序存储效率比链接存储效率更高）
        ii. 顺序存储插入、删除元素需要移动表内数据，而链接存储通过改变指针的指向就可以（在插入删除操作中，链接存储比顺序存储效率更高）
        iii. 搜索元素时，顺序存储可以通过折半搜索提高效率，链接存储不行

    a. 基于空间的考虑。当要求存储的线性表长度变化不大，易于事先确定其大小时，为了节约存储空间，宜采用顺序表；反之，当线性表长度变化大，难以估计其存储规模时，采用动态链表作为存储结构为好。
    b. 基于时间的考虑。若线性表的操作主要是进行查找，很少做插入和删除操作时，采用顺序表做存储结构为宜；反之， 若需要对线性表进行频繁地插入或删除等的操作时，宜采用链表做存储结构。并且，若链表的插入和删除主要发生在表的首尾两端，则采用尾指针表示的单循环链表为宜。

来自 <http://student.zjzk.cn/course_ware/data_structure/web/practice/answer/answer2.2.htm>

2. 顺序存储下的插入删除操作
3. 线性表的不同描述方式的复杂度对比
4. 单链表中插入/删除一个节点值为x的算法
5. 画出循环单链表的结构示意图并写出在其末尾插入一结点的操作步骤。

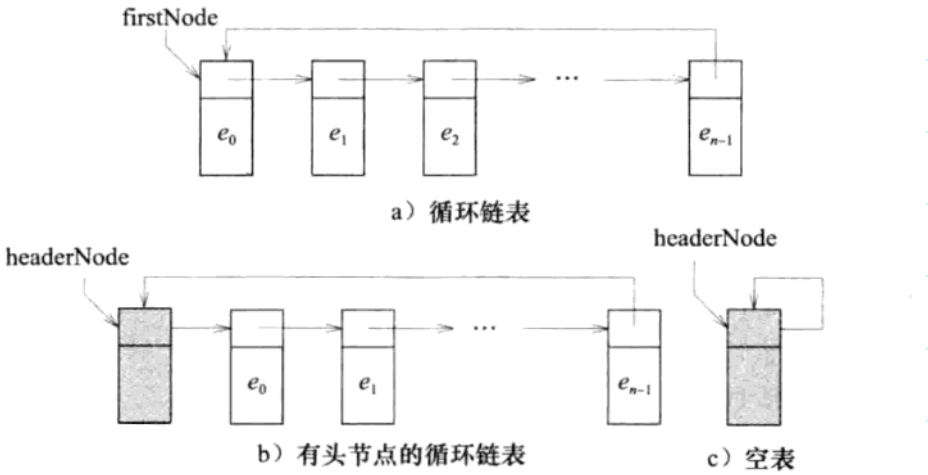


图 6-6　循环链表

根据是线性表（a）和链表（b）的不同在末尾插入的方式不同，线性表的插入操作和普通操作一样，在获取和查找的时候可能会有所变化。

链表的插入（设插入theElement）操作：

```
{
    first->data = theElement;
    Chain* current = first->link;
    while(current->link->data!=theElement)
    {
        current = current->link;
```

```
        }
        Chain* node = new Chain();
        node->data = theElement;
        node->link = first;
        current->link = node;
    }
```

6. 写出将一个单链表中值为x的结点与其后继结点交换位置的算法

```
    {
        Chain * current = first;
        if(first -> data = x)
        {
            first = current->link;
            current->link = first->link;
            first->link = current;
            return;
        }
        while(current & current->link & current->link-data != x)
        {
            curent = current->link;
        }
        Chain* next = current-link;
        if(!next)return;
        current->link = next->link;
        next->link = next->link->link;
        current->link->link = next;
    }
```

7. 画出双链表结构示意图并写出在双链表中P指向的结点之前插入一个结点（值为x）的操作步骤。

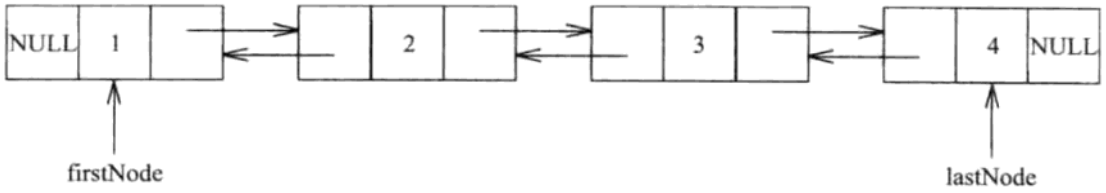


图 6-7　双向链表

```
    {
        Chain*pre = p->left;
        Chain* node = new Chain();
        node->right = p;
        node->left = pre;
        node->data = x;
        p->left = node;
        if(pre)
        {
            firstNode = node;
        }else
        {
            pre->right = node;
        }

    }
```

三、跳表，散列

1. 请说出散列方法组织动态表的主要思想及其要解决的两个基本问题。设有一组关键字{72,35,124,153,84,57}需插入到长度为12的哈希表中，请设计一个适当的哈希函数，并构造出对应的哈希表。

四、堆栈，队列

1. 简述堆栈与队列的结构特点、空与满的条件。

   队列是先进先出，堆栈是先进后出；栈和队列中无任何元素为空，元素个数到达预先设置的最大个数为满。

判别循环队列的"空"或"满"不能以头尾指针是否相等来确定，一般是通过以下几种方法：少用一个元素的空间,每次入队前测试入队后头尾指针是否会重合，如果会重合就认为队列已满。

2. 写出利用单循环链表实现一个队列的入队出队算法。

```
Push(const T&x)
{
        first->data = x;
        Chain* current = first->link;
        while(current->link->data = x)
        {
                current = current->link;
        }
        Chain* node = new Chain();
        node->data = x;
        node->link = first;
        current->link = node;
}
Pop(T& x)
{
        if(first->link = first)
        { throw "Empty";}
        x = first->link->data;
        first->link = first->link->link;
}
```

四、树

1. 写出二叉树的前、中序扫描的递归的算法；按层次遍历的算法。

前序中序：preOrder/inOrder（）

层次遍历：

```
Queue* queue = new Queue();
queue->push(node);
while(!queue-<>Empty())
{
        TreeNode* now = queue->Top();
        if(!now->left)
        {
                queue->Push(now->left);
        }
        if(!now->right)
        {
                queue->Push(now->right);
        }
        visit(now);
        queue->Pop();
}
```

2. 会写出前序，中序，后序的遍历结果，根据任意两个序列推出二叉树结构

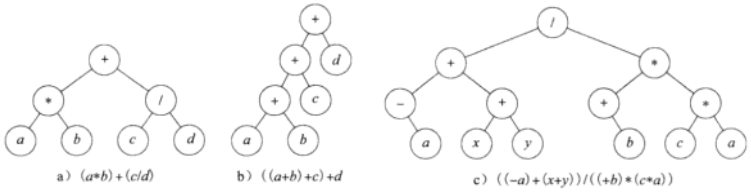


图 11-5　表达式树

	a）	b）	c）
前序	+*ab/cd	+++abcd	/+-a+xy*+b*ca
中序	a*b+c/d	a+b+c+d	-a+x+y/+b*c*a
后序	ab*cd/+	ab+c+d+	a-xy++b+ca**/

图 11-11　二叉树按前序、中序、后序遍历所列出的元素

3. 会将二叉树转为森林，将森林，树转为二叉树（全部为左兄弟，右孩子）

4. 什么叫Huffman树？由给定权的集合{15,4,12,8,5,19,23}构造对应的Huffman树，并计算其赋权外通路长度。

   霍夫曼树：对于给定的频率具有**最小加权外部路径长度**的二叉树

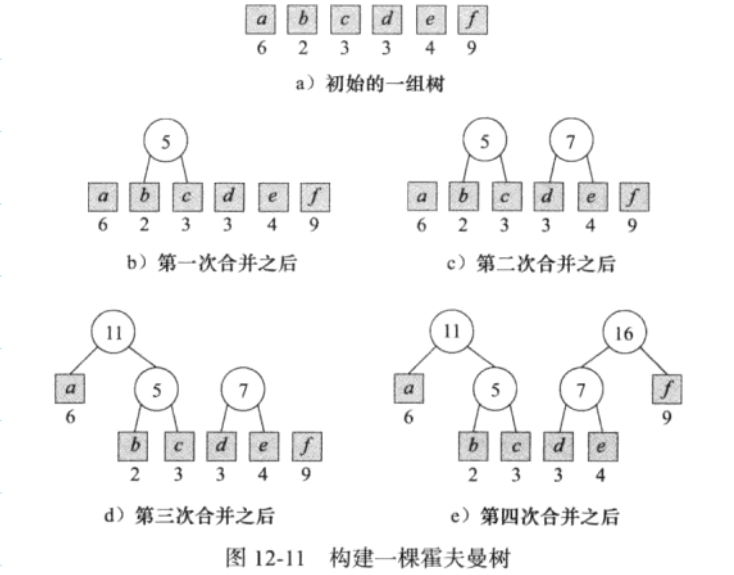


图 12-11　构建一棵霍夫曼树

5. 什么是最大小堆？请画出将序列{11,9,3,6,7,4,5,10,8,1,2}存到一个完全二叉树中的情形，画出将其调整成最小堆阵的过程。

   最大(小)堆：每个节点的值都大于（小于）或等于其子节点（如果有）的值

   ①交换1,7

   ②交换1,9；2,9

   ③交换11,1；2,11；11,7

   ④得到{1,2,3,6,7,4,5,10,8,11,9}

6. 写出复制一棵二叉树的算法；求二叉树叶结点个数的算法。

```
Queue* theQueue = new Queue();
Queue* queue = new Queue();
TreeNode* node = new TreeNode();
node->data = theNode->data;
theQueue->Push(theNode);
queue->Push(node);
while(!theQueue->Empty())
{
        TreeNode* theNow = theQueue->Top();
        TreeNode* now = queue->Top();
        if(!theNow->left)
        {
                theQueue->Push(theNow->left);
                now->InsertInLeft(now->left->data);
                queue->Push(now->left);
        }
        if(!theNow->right)
        {
                theQueue->Push(theNow->right);
                now->InsertInRight(now->right->data);
                queue->Push(now->right);
        }
        theQueue->Pop();
        queue->Pop();
}
```

   求节点个数同理

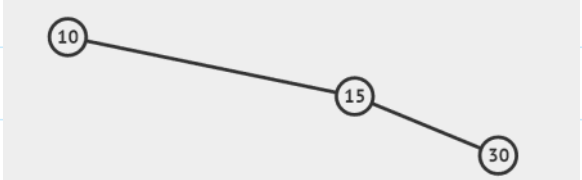
7. 请画出由关键码{10,15,30,32,20,50,5,3,25,28,22,18}建立的二叉搜索树的形状。

8. 什么是AVL树？请画出由关键码{10,15,30,32,20,50, 5,3,25,28,22,18}建立AVL树的过程。

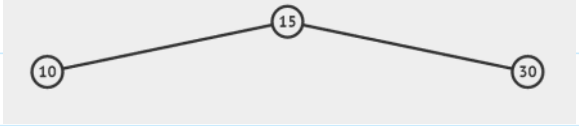
   AVl树是一种平衡二叉树，它要求任一节点的左子树深度和右子树深度相差不超过1

注意是以从下往上的第一个失衡节点为中心开始旋转

a.

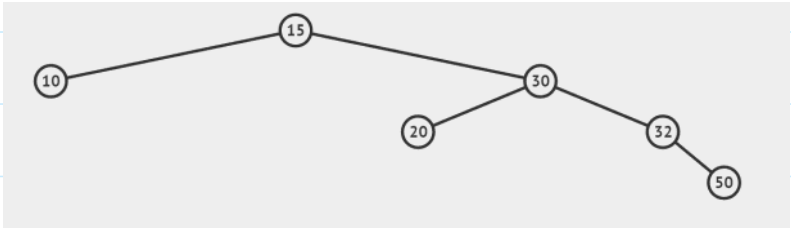


b.

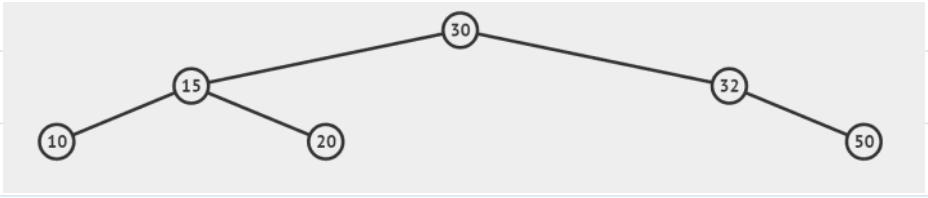


第一次旋转

c.



d.

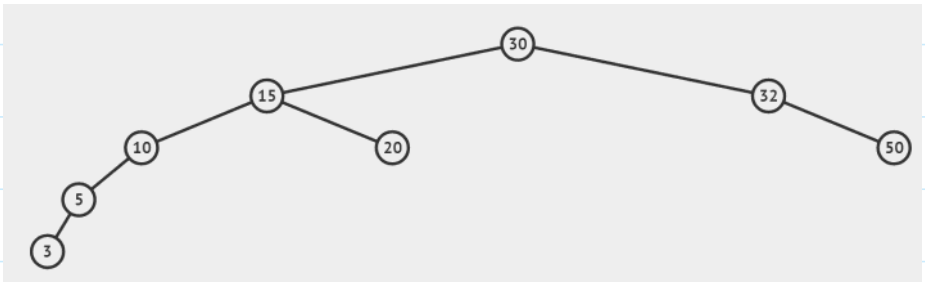


第二次旋转

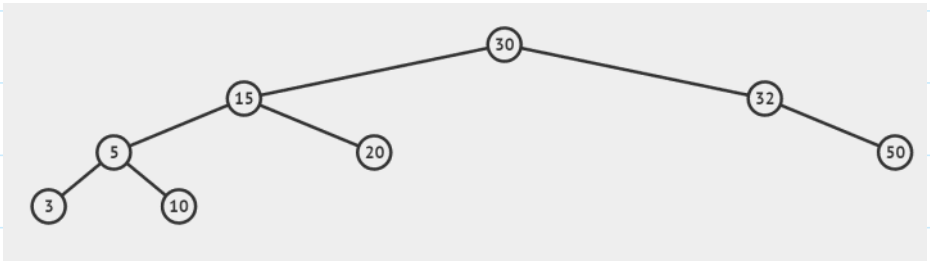
i. 注意，单旋的时候，如果有多余节点，一般是**降阶的节点连接到升阶的多余节点**上，如c中20多余，15降阶，那么15连接20（**同侧相连**的规律）

ii. 这个规律不光在单个节点的时候适用，多个节点一起旋转的时候也适用

e.

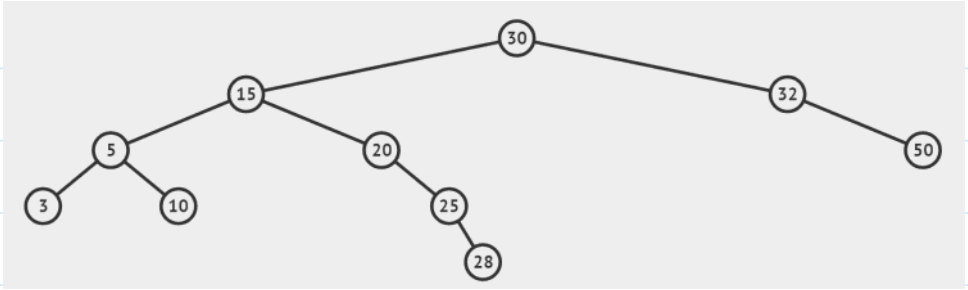


f.

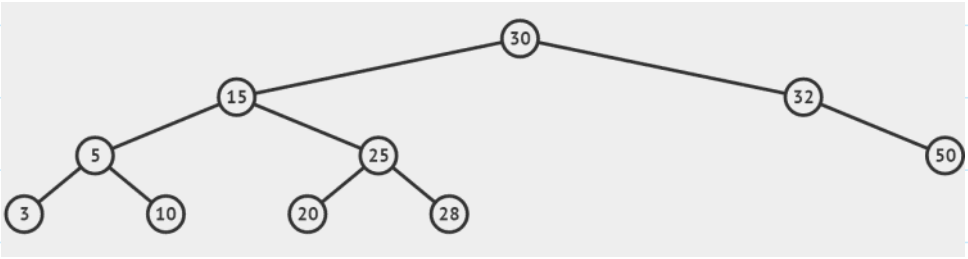


第三次旋转

g.

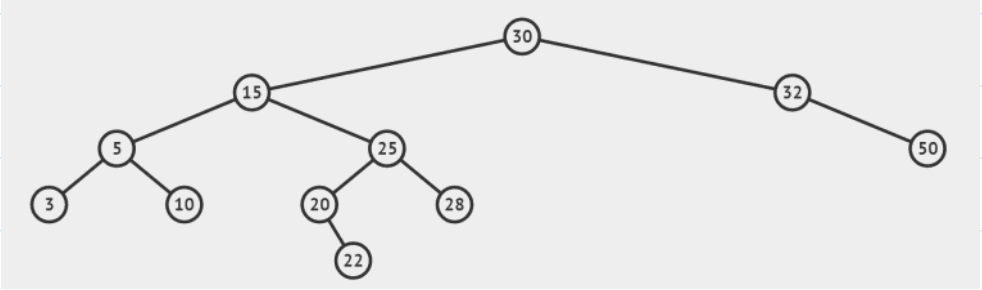


h.

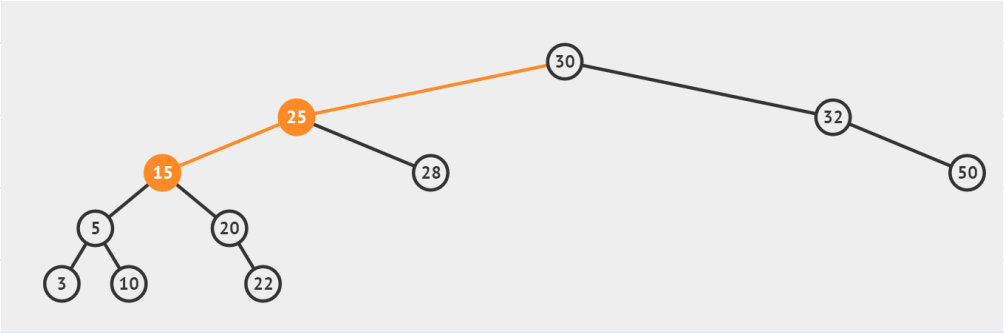


第四次旋转

i.



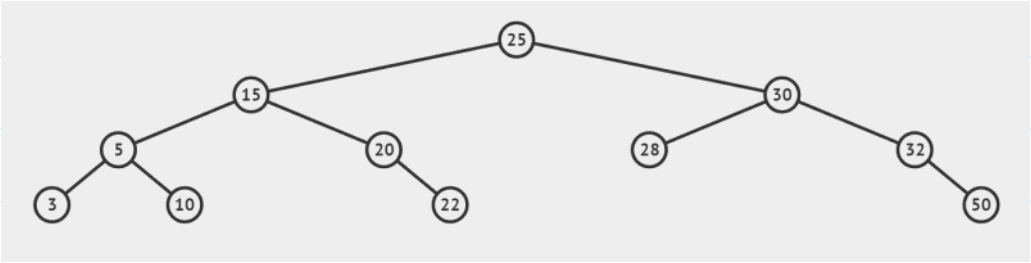
j.



第五次旋转（双旋）

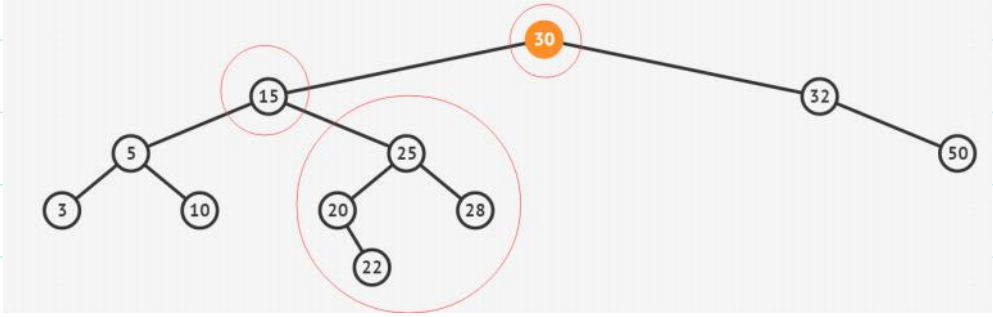
i. 先左旋

k.



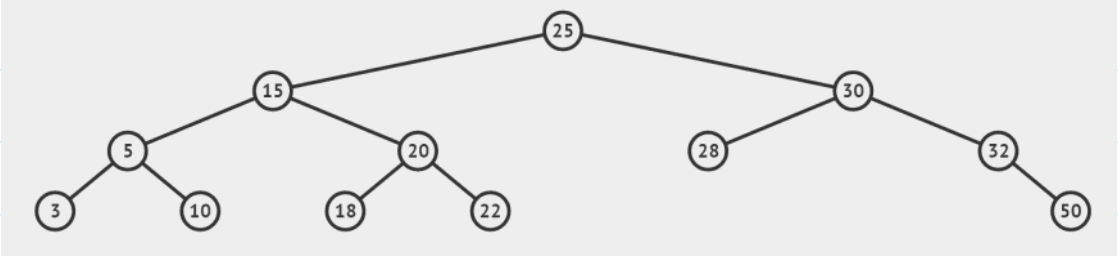
i. 再右旋

ii.



iii. 应该把25所在的树看成一个整体，符合"<",">"形状的要进行两次旋转，注意单次旋转，都满足**同侧相连**的规律

l.



9. 写出在一个二叉搜索树中查找、插入、删除一个结点的算法。

查找，插入都比较好些，删除：

```
template<class K, class E>
void binarySearchTree<K,E>::erase(const K& theKey)
{// 删除其关键字等于 theKey 的数对

   // 查找关键字为 theKey 的节点
   binaryTreeNode<pair<const K, E> > *p = root,
                                      *pp = NULL;


  while (p != NULL && p->element.first != theKey)
  {// p 移到它的一个孩子节点
     pp = p;
     if (theKey < p->element.first)
        p = p->leftChild;
     else
        p = p->rightChild;
  }
  if (p == NULL)
     return; // 不存在与关键字 theKey 匹配的数对

  // 重新组织树结构
  // 当 p 有两个孩子时的处理
  if (p->leftChild != NULL && p->rightChild != NULL)
  {// 两个孩子
     // 转化为空或只有一个孩子
     // 在 p 的左子树中寻找最大元素
     binaryTreeNode<pair<const K, E> > *s = p->leftChild,
                                       *ps = p;        // s 的双亲
     while (s->rightChild != NULL)
     {// 移到最大的元素
        ps = s;
        s = s->rightChild;
     }
      // 将最大元素 s 移到 p, 但不是简单的移动
      //p->element = s->element , 因为 key 是常量
      binaryTreeNode<pair<const K, E> > *q =
         new binaryTreeNode<pair<const K, E> >
             (s->element, p->leftChild, p->rightChild);
      if (pp == NULL)
         root = q;
      else if (p == pp->leftChild)
              pp->leftChild = q;
           else
              pp->rightChild = q;
      if (ps == p) pp = q;
      else pp = ps;
      delete p;
      p = s;
  }
```

```
        //p 最多有一个孩子
        //把孩子指针存放在 c
        binaryTreeNode<pair<const K, E> > *c;
        if (p->leftChild != NULL)
           c = p->leftChild;
        else
           c = p->rightChild;

        //删除 p
        if (p == root)
           root = c;
        else
        {//p 是 pp 的左孩子还是右孩子？
           if (p == pp->leftChild)
              pp->leftChild = c;

            else pp->rightChild = c;
         }
         treeSize--;
         delete p;
      }
```

10. 请解释生成树和最小生成树的概念。

    如果连通图G的一个子图是一棵包含G的所有顶点的树，则该子图称为G的生成树

    生成树的权重达到最小的树是该图的最小生成树

11. 描述一种最小生成树算法的思想。画出下列网络的最小生成树。

    迪克斯特拉算法，每次找到权重最小的并且不会和已经连接的边形成回路的边，将其连接直到找不到这样的边，则此时生成的就是该图的最小生成树

12. 从某节点开始进行BFS/DFS得到的序列
13. 画出某个图的邻接矩阵和邻接链表的存储结构、
14. 描述B_树的结构特点
15. 在下列5阶B_树中删除关键字70，画出调整后的B树。

特别注意

2018年1月14日 11:12

1. 链表和顺序存储的比较以及选择方法
2. 拓扑排序的思想
3. 各种排序算法的复杂度
4. B树的定义，根据给定m序B树，元素个数，高度中的两个求第三个

头指针:是在链表中引入的附加节点。利用该节点通常可以使程序设计更简洁，因为这样可以避免把空表作为一种特殊情况来对待。使用头指针时，每个链表（包括空表）都至少包含一个节点（即头指针）。

双向链表:双向链表由从左至右按序排列的节点构成。right 指针用于把节点从左至右链接在一起，最右边节点的right指针为0。lef t指针用于把节点从右至左链接在一起，最左边节点的left指针为0。

双向循环链表:双向循环链表与双向链表的唯一区别在于，最左边节点的l e f t指针指向最右边的节点，而最右边节点的r i g h t指针指向最左边的节点。

在图 7-1 中，从第一行开始，依次对每一行的索引从左至右连续编号，得到图 7-2a 所示的映射结果。它把二维数组的索引映射为 [0,n−1] 中的数，这种映射方式称为**行主映射**（row major mapping）。索引对应的数称为**行主次序**（row-major order）。图 7-2b 是另一种映射模式，称为**列主映射**（column major mapping）。在列主映射中，对索引的编号从最左列开始，依次对每一列的索引从上到下连续编号。

[0][0]	[0][1]	[0][2]	[0][3]	[0][4]	[0][5]
[1][0]	[1][1]	[1][2]	[1][3]	[1][4]	[1][5]
[2][0]	[2][1]	[2][2]	[2][3]	[2][4]	[2][5]

图 7-1 整型数组 score[3][6] 的索引排列表

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17

a）行主映射

0	3	6	9	12	15
1	4	7	10	13	16
2	5	8	11	14	17

b）列主映射

图 7-2 映射一个二维数组

设无向图G中有n个顶点e条边，则其对应的邻接表中的表头结点和表结点的个数分别为（D ）。

(A) n，e (B) e，n (C) 2n，e (D) n，2e

设某有向图的邻接表中有n个表头结点和m个表结点，则该图中有（C ）条有向边。

(A) n (B) n-1 (C) m (D) m-1

若有18个元素的有序表存放在一维数组A[19]中，第一个元素放A[1]中，现进行二分查找，则查找A［3］的比较序列的下标依次为(C)

A. 1，2，3　　B. 9，5，2，3

C. 9，5，3　　D. 9，4，2，3

1. 二叉树的四种遍历（带非递归）
2. 统计叶子节点的个数
3. 堆的插入删除初始化
4. 二叉搜索树的插入删除
5. 图，找一个节点的子节点
6. bfs、dfs
7. 找i到j是否存在一条路径
8. 迪克斯特拉
9. 克鲁斯科尔
10. 普林

链式栈无栈满问题，空间可扩充

插入与删除仅在栈顶处执行

链式栈的栈顶在链头

适合于多栈操作

1. 跳表比散列表更灵活.
2. 跳表能在线性时间内按升序输出所有的元素.
3. 采用链表散列时，需要(D+n) 时间去收集n 个元素，并且需要O(nlogn)时间进行排序，之后才能输出 。
4. 查找或删除最大或最小元素，散列可能要花费更多的时间（仅考虑平均复杂性）。