KMP算法

如果是next，将所有posP替换成posP+1，fail[]替换为fail[]-1即可，但fail[0]仍是-1

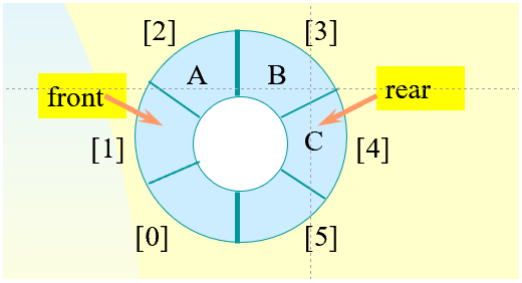
```
int fastFind(string str,string pattern)
{
    int posS = 0, posP = 0;
    while (posS ≠ str.length()&&posP≠pattern.length())
    {   // 匹配，继续该过程
        if (str[posS]) == pattern[posP])
        {
            posS++;
            posP++;
        }
        else
        {   // 不匹配，设定为-1
            if (posP == 0)
                posS++;
            else    // 不匹配，回退到上一个可能位置继续尝试匹配
                posP = fail[posP - 1] + 1;
        }
        if (posP<p.length()||p.length()==0)
            return -1;
        else
            return pos;
    }
}
```

```
void failure(string pattern)
{
    fail[0] = -1;
    int to_match = 1, matched = 0;
    while (to_match≠pattern.length())
    {
        if (pattern[to_match] == pattern[matched])
        {
            fail[to_match] = matched;
            to_match++;
            matched++;
        }
        else
        {
            if (matched == 0)
            {
                fail[to_match] = -1;
                to_match++;
            }
            else
            {
                matched = fail[matched - 1] + 1;
            }
```

```
        }
    }
}
```

循环队列

- 当队列为空时，front==rear
- **无法判断空和满**，有多种实现方法
  1. 使用一个bool值记录上一次操作是put还是remove。若上一次是remove则空，否则满。
  2. 保留一个元素的空间，如将front指向第一个元素。若(rear+1)%maxSize==front，则满，空判断方法不变。
  3. 用一个整型size来记录大小，put时size++，remove时size-。若size==0则空，size==maxSize时满。



链表

获取倒数第n个元素可以用双指针，代码省略。**逆转**代码如下

```
void reverse(Chain chain)
{ // make (a1,.., an) becomes (an,.., a1).
    ChainNode *cur = chain.first, *pre = 0; //cur为将要逆转的元素，pre为其父亲
    while (current) {
        ChainNode *r = pre; // 暂存current的前一个元素（在后续步骤中要用到pre）
        pre = cur;          // 更新previous（若不在此处保存，更新cur后后无法获得cur)
        cur = cur→next;     // 更新current（若不在此处保存，在逆转后无法获取cur->next)
        pre→next = r;       // 将链表逆转
    }
    chain.first = previous; // 更新链表的头结点
}
```

线段树

与正常的二叉树类似，但若没有左孩子或右孩子，则将**左孩子**设为中序遍历的**前驱**，**右孩子**设为**后继**，同时用两个标志位**分别**存储是否有左孩子和右孩子。

建立方式：建树后对二叉树进行一次中序遍历。更新方式如下：

```
void ThreadedTree<T>::InsertRight(ThreadedNode<T> *s, ThreadedNode<T> *r)
{ // Insert r as the right child of s
     r→rightChild = s→rightChild;                    // 自己的右节点设为父亲的右节
点
     r→rightThread = s→rightThread;                  // 自己的右标志位设为与父亲相
同
     r→leftChild = s;                                // 自己的左节点设为父亲
     r→leftThread = true;                            // 设定自己不存在左孩子
     s→rightChild = r;                               // 父亲的右孩子设为自己
     s→rightThread = false;                          // 设定父亲存在右孩子
     if (! r→rightThread) {                          // 如果父亲原来有右孩子
         ThreadedNode<T>* temp = InorderSucc(r);     // 获取中序遍历的下一个节点
         temp→leftChild = r;                         // 下一个节点的前序设为自己
     }
}
```

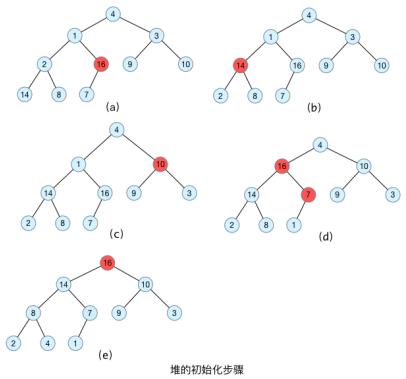
堆

一般用**数组**存储，i节点的父亲是i/2，左子节点是i*2，右子节点是i*2+1

建立：自深到浅，自右向左地比较各个子树，将较大/小的节点上浮

插入：按照插入节点到根节点的路径依次上浮直到不需要上浮

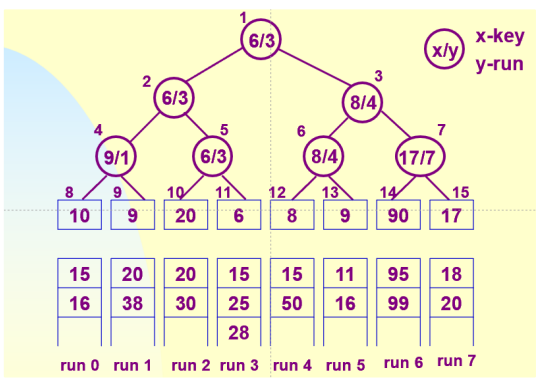
删除堆顶：将最后的节点提到根节点，删除最后的节点并重新建立步骤

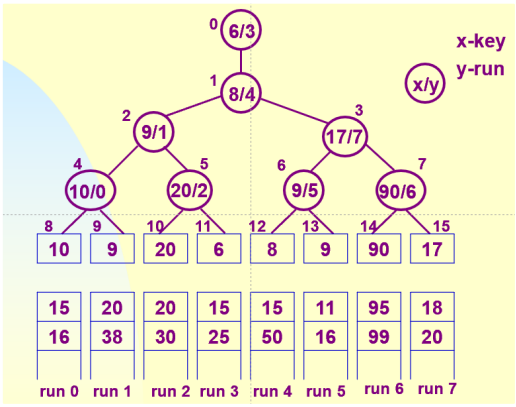


堆的初始化步骤

胜者树/败者树

一般用于外部排序。胜者树每个非叶子节点存储子节点中胜出的（较小）和其对应的叶子节点序号，败者树中存储子节点中失败的，但其根节点还有一个节点存储全局胜者。叶子节点还包含一个数组，叶子节点的值为该数组的第一个值。





排序

插入排序

A．直接插入排序

采用顺序查找法查找插入位置：复制插入元素，记录后移，查找插入位置 插入到正确位置

原始数据越接近有序，排序速度越快

时间 最好情况O(n)，平均和最坏情况O(n^2)

B．二分插入排序（又称折半插入排序）

采用折半查找法（二分法）查找插入位置

与直接插入排序相比，减少了比较次数，但没有减少移动次数

排序速度与原始数据有序性无关

时间 O(n^2)

交换排序

基本思想：两两比较，如果发生逆序则交换，直到所有记录都排好序为止

A．冒泡排序

n个元素，冒n-1趟，第m趟比较n-m次

某一趟没发生交换，就可以提前结束

时间最好O(n) 平均和最坏O(n^2)

B．快速排序

递归，平均需要O(log_2n)栈 空间，最坏需要O(n)栈空间

平均时间O($nlog_2n$) ，最坏 O(n^2)

不适于对原本有序或基本有序的记录序列进行排序（退化为冒泡排序）

选择排序

基本思想：在待排序的数据中选出最大（小）的元素放在其最终的位置

A．简单选择排序

第i趟从n-i个元素中挑一个最值的

时间最好平均最坏最坏 O(n^2)

B．堆排序

利用完全二叉树中父结点和孩子结点之间的内在关系来排序的

时间初始堆化O(n)，一次重新堆化O(logn)，排序n-1次循环O(nlogn)，最好最坏情况总体O(nlogn)

归并排序

基本思想：将两个或两个以上的有序子序列“归并‘为一个有序序列
2-路归并排序
整个归并排序仅需ceil(log_2n)
时间O($nlog_2n$)，空间O(n)

基数排序或桶排序

基本思想：分配，收集

排序方法对比

类别	排序方法	最好复杂度	最坏复杂度	平均复杂度	辅助存储	稳定性
插入	直接插入	O(n)	O(n^2)	O(n^2)	O(1)	稳定
插入	希尔	O(n)	O(n^2)	~O($n^{1.3}$)	O(1)	不稳定
交换	冒泡	O(n)	O(n^2)	O(n^2)	O(1)	稳定
交换	快速	O($nlogn$)	O(n^2)	O($nlogn$)	O($nlogn$)	不稳定
选择	直接选择	O(n^2)	O(n^2)	O(n^2)	O(1)	不稳定
选择	堆排序	O($nlogn$)	O($nlogn$)	O($nlogn$)	O(n^2)	不稳定
归并	归并排序	O($nlogn$)	O($nlogn$)	O($nlogn$)	O(n)	稳定
基数	基数排序	O($n+m$)	O($k*(n+m)$)	O($k*(n+m)$)	O($n+m$)	稳定

注：基数排序中k位待排元素的位数位数，m为基数的个数

- 直接插入和冒泡在关键字有序时是最好情况，但快速排序此时为最坏情况

哈希

哈希碰撞的解决方法：**开放寻址法**（下面讲）和**链地址法**（将所有哈希地址相同的记录都链接在同一链表中，考得少）

- **线性探测法**：不断对求得的哈希值+1后取模，直到找到空闲位置(如最大空间为6，计算得4，尝试4,5,0,1,2,3)
- **平方探测法**：不+1而是+1^2,-1^2,+2^2,-2^2…，直到n^2>哈希表的最大空间(同上例，尝试4,5,3,2,0)