目录

数组（顺序表）

时间复杂度

查1删n插n

插入：$\frac{n}{2}$

删除：$\frac{n-1}{2}$

算法

二路有序归并

每次都比较二顺序表内的值，并在归并结束后再单独遍历。

逆置顺序表

时间$O(n)$空间$O(1)$

遍历前半部分元素，对于$L.data[i]$，将其与$L.data[L.length-i-1]$交换。

前半后半分类

根据一定规则：使前半部分满足规则A，后半部分满足规则B

维护两个指针，一个从前向后，一个从后向前。搜索到不满足各自半区要求的数值时停下遍历，等待另一个指针也搜到。然后当都搜到了后，对两个位置进行交换。

时间O(n)空间O(1)。

将大于等于0的放在后半部分，小于0的放在前半部分

从L的两端查找，前端找大于等于0，后端找小于0，找到后交换这两个位置的元素

高效筛选顺序表

术语表

- 设原顺序表为L，希望得到的顺序表为L_{new}，该特定规则为$M(x)$，且$M(x)=1 \leftrightarrow x \in L_{new}$。
- 设置一个k，要求对于$L[0...k]$，这部分元素都应该有$M(x)=1$，换句话即$0...k$的元素实际上就是L_{new}。

算法流程

1. 对L进行遍历，取出i号元素，值为L_i。
2. 计算$M(L_i)$。
3. 若$M(L_i)=1$，则将$L_k=L_i$，$k=k+1$。
4. 若$M(L_i)=0$，则不做任何操作。
5. $i=i+1$

时间复杂度

将$M(x)$的时间复杂度记为$O(m)$，该算法的时间复杂度为$O(nm)$。

- 但是大部分时间$M(x)$其实是$O(1)$的，所以基本上是$O(n)$的复杂度，很优秀。

空间复杂度

由于全程在复用原来的数组，即空间复杂度为$O(1)$。

例：删除所有值为x的元素

时间$O(n)$空间$O(1)$

采用新建表的思路，如果i位置不是x，那么$l.data[k]=l.data[i],\ k++$

如果i位置是x，那直接i++。最后把表长设为k。

集合的交并差补集

二路归并

单个数组内两组数据

- 单双交错
- 正反两头出发

问题与解答

- 什么情况下使用顺序表比链表好?

当线性表没有或很少进行插入和删除操作，或者插入和删除操作总是尾部进行时，使用顺序表比较好。

- 简述顺序表与链表的主要优缺点

类别	优势	劣势
顺序表	随机存取；不需要额外空间表示逻辑关系；储存密度高；结构简单。	需要占用一整片连续的空间；插入与删除元素需要移动大量元素；表容量不便于扩充；初始空间大小难以确定。
链表	便于结点的插入与删除；表容量扩充方便。	不具有随机读取的特性，只能顺序访问；每个结点增加了指针域，降低了存储密度。

顺序表

优：随机存取；不需要额外空间表示逻辑关系；储存密度高；结构简单。

劣：需要占用一整片连续的空间；插入与删除元素需要移动大量元素；表容量不便于扩充；初始空间大小难以确定。

链表

优：便于结点的插入与删除；表容量扩充方便。

劣：不具有随机读取的特性，只能顺序访问；每个结点增加了指针域，降低了存储密度。

链表

概念

指针域 数据域

时间复杂度

查n删1插1

单向链表

```
template<typename E>
struct Node{
  E data;
  Node *next;
};

template<typename E>
class Link {
private:
  Node *curr;
  Node *head, *tail;
  int cnt;
public:
  Link();
  E remove();
  Node* insertBefore(E value);
  Node* insertAfter(E value);
  Node* append(E value);
  void moveToStart();
  void moveToEnd();
  void prev();
  void next();
  void set(E value);
  E get(int i);

  void insertFromTail();
}
```

头插法

> 一句话：每次都将结点插在头结点与首节点之间，也即是头结点之后，首结点之前。

```
void insertFromHead() {
  head = (Node*) malloc(sizeof(Node)); //为头结点分配空间
  curr = head;                         //将curr结点指向
  Node *p;
  while(true) {
    p = (Node*) malloc(sizeof(Node));
    scanf("%d", &p->data);             //读入
    if(p->data==0) break;
    p->next = head->next;              //其实就是head与head->next之间插入了一个点
    head->next = p;
  }
}
```

尾插法

> 一句话：每次都在tail的后面插点。

直接把结点插到表尾。需要维护一个尾指针。

每进行一次插入操作，tail的next都设为新结点p，然后再把tail设为p。

tail的next都设为新结点p：这里的tail其实还是插入p之前的那个尾结点，通过这样的设置，可以让p有前驱结点。

```
void insertFromTail() {
  head = (Node*) malloc(sizeof(Node)); //为头结点分配空间
  tail = head;                         //将尾结点指向
  Node *p;
  while(true) {
    p = (Node*) malloc(sizeof(Node));
    scanf("%d", &p->data);             //读入
    if(p->data==0) break;
    tail->next = p;                    //尾结点的next设为p，这相当于插入了p这个结点在next后
    tail = p;                          //尾结点设为p，        这相当于将p设为了新的尾结点
  }
}
```

在curr前插入结点

本质上其实是在curr后面加了一个结点，然后把后面的结点设成了curr的样子，再把curr改成想要的样子。

如此，p就替代了curr，然后curr变成了新的结点。

```
Node* insertBefore(E value) {
  Node *p = (Node*) malloc(sizeof(Node));       //为p分配空间
  p→data = curr→data;                           //设置p的值为curr现在的值
  p→next = curr→next;                           //将p的next设为curr的next
  curr→next = p;                                //把curr→next设为p，这里开始相当于p取代了原来的curr
  curr→data = value;                            //再把现在的curr设成新值，就相当于插入了一个新结点
};
```

在curr后插入结点

```
Node* insertAfter(E value) {
  Node *p = (Node*) malloc(sizeof(Node)); //为p分配空间
  p→data = value;                         //设置p的值
  p→next = curr→next;                     //将p的next设为curr的next→相当于在curr和curr→next中间插了一脚
  curr→next = p;                          //把curr→next设为p
};
```

删除结点

本质上其实是把curr改成了curr→next,然后删掉了curr→next

```
E remove() {
  Node *p = curr→next; //使用p指向curr→next
  curr→data = p→data; //将curr的值设为curr→next的值
  curr→next = p→next; //将curr的next设为curr→next的next
  free(p);            //释放curr→next
};
```

> 单独考虑p为尾结点的情况！尾结点的话还是要遍历找pre的！

逆置链表

采用头插法来逆置链表。

用p点持续指向原链表。然后将链表清空，并头插法插入q遍历下去的点。

正向遍历，然后头插，就会得到一个逆置的链表。

所以原来序列里是12，这里就会先0→null，然后0→1→null，然后0→2→1→null。每次都是在head和head→next之间插点。

空间复杂度 O(1)，本质上是修改链接，所以没有新的空间的支出。

```
void reverse() {
  Node *q, *p;
  q = head→next;              //取出首结点
  head→next = NULL;           //将首节点置空，以便后续进行reverse
  while(q≠NULL) {
    p           = q→next;     //取出q→next
    q    → next = head→next; //这两行其实是在head和head→next之间插入了q这个结点
    head→ next = q;
    q           = p;          //最后把q设为“q”的next，也就是原来序列中的下一个
  }
}
```

归并算法

> 后面补一下

析构

思路需要简单记一下，pre指向head，p永远指向pre的next，两个指针同步后移，每次删一下pre。

```
~Link {
  Node *pre = head, *p = head→next;
  while(p) {   //p与pre同步后移
    free(pre); //只释放pre
    pre = p;
    p = p→next;
  }
}
```

根据一定规则：拆表

本质上就是根据一定规则，分别尾插法。

找出中间结点

快慢指针。由于q的速度是p的一倍，所以如果q到达了终点，那么p应该正好在中间。

```
static Node* findMiddle(Node *head) {
  Node *p = head, *q = head;
  while(q && q→next) {
    p = p→next;
    q = q→next→next;
  }
  return p;
}
```

2-3-33

【破坏链表|不破坏链表】

两个链表，以一定顺序合并成一个新的链表，这里就出现了抉择。可以破坏原链表，也可以不破坏。

破坏 新链表直接用原链表里的指针连。

不破坏 新链表先生成新节点，然后新节点们去连。

双向链表

需要记的

1. 各种操作，见P77
2. 计算修改指针域的个数（其实就是操作）
3. 节点顺序交换 -P80

```
template<E>
struct biNode {
  E value;
  node *prev;
}
```

在curr前插入结点

```
p = (Node*) malloc(sizeof(Node));
p→data = data;
p→prev = curr→prev;
curr→prev→next = p;
p→next = curr;
curr→prev = p;
```

在curr后插入结点

```
p = (Node*) malloc(sizeof(Node));
p→data = data;
p→next = curr→next;
curr→next→prev = p;
p→prev = curr;
curr→next = p;
```

删除curr后的结点

```
//删除p
p = curr→next;
p→next→prev = curr;
curr→next = p→next;
free(q);
```

删除curr

```
//删除p
curr→prev→next = curr→next;
curr→next→prev = curr→prev;
Node *tmp = curr→next;
free(curr);
curr = tmp;
```

循环链表

tail的next设为head的prev。

应用

1. 龟兔赛跑判断是否存在环
2. 全部反向

本章问题与解答

- **以下问题的简单总结：**

2、4、5本质上是同一个问题，思路是一致的，都是建立节点以模拟节点而非直接删除/添加。

- **1. 对单链表设置头结点的作用是什么?**

1. 简化插入和删除操作。首结点的删除不需要特殊处理。
2. 统一了空表与非空表的操作。

- **2. O(1)算法删除p结点**

困难 如何修改p的前驱结点的next?

解法 不修改。删除p后结点(q)，把p的值设为q，即可。

```
node *q = p→next;
p→data = q→data;
p→next = q→next;
delete q;
```

- **3. 将长度为n的单链接在长度为m的单链后，时间复杂度为?**

O(m)，因为需要找到m的尾结点

- **4. 在p前插入一个结点，O(1)**

困难 怎么知道p前面是谁?

解法 在p后插入，然后交换两个点的值。

```
p前插入s
s→next=p→next→next;
p→next=s;
t=p→data;
p→data=s→data;
s→data=t;
```

- 5. 单链表中O(1)修改p结点前驱结点的next

思路 不修改前驱结点，二是把p结点的值进行修改。

- 如何在P前插入一个结点?
- 如何删除P?

栈

顺序栈

进出栈序列

1. $p_1 = n$、 $p_n = 1$整个栈依次进入，全部进入后再弹出
2. $p_3 = 1, p_2$__2: 一定不等于
3. $p_3 = 3, p_1$__2: 可能等于
4. $p_1 = 3, p_2 =$ __: 2

根据进出栈序列求容量

看每次进出栈对应时，栈内变量个数

> 进(a,b,c,d,e)，出(c,e,d,b,a)

可能的出栈序列个数

$\frac{1}{n+1}C_{2n}^{n}$

顺序栈的定义

1. 顺序：代表其储存结构，并不是说变量按顺序储存
2. 对顺序栈进出栈不涉及变量的移动

根据初始储存方式写操作

1. data[1..n]，初始栈顶top为n+1 ⟹ data[--top]=x;
2. data[1..n]，初始栈顶top为n ⟹ data[top--]=x;

> 核心在于一开始data[top]是否存在，如果不存在那就先减

中缀转后缀

先手动加括号，然后把右括号换成对应的运算符，然后删掉左括号

应用

- 进制转换
- 平衡符号
- 波兰表达式
  - INFIX与RPN的转化

    - 四则运算
- 迷宫与BFS
- 汉诺塔
- 简单背包问题

队列

顺序队列

一点简单的性质，就不赘述了，FIFO这种

循环队列

循环队列的优点

解决了假溢出的问题，提高了空间利用率

各种操作与判定

1. 队空：$front == read$
2. 队满：$front = (rear + 1) \mod maxSize$
3. 队内元素个数：$(read - front + maxSize) \mod maxSize$

为什么f指向首前，r指向尾，浪费一个元素?

因为对于一个data[0..m-1]的数组而言，其状态有：$\{$空$, 1, 2, \ldots, m-1,$满$\}$共计$m-1+2=m+1$种状态，由于f与r必须取[0,m-1]中的数，故$|f-r|$只有m种取值，无法表示完全m-1个状态。
因而需要浪费一个元素。

数组与矩阵

对称矩阵压缩

$k = \frac{大 \cdot (大+1)}{2} + 小$

稀疏矩阵

存放于三元组<i,j,aij>

树

概念部分

普通树

1. 度:
2. 层:
3. 兄弟

定理

二叉树

定理

1. 第i层的结点数小于等于2^(i-1)
2. 高为k的二叉树最多有2^k-1个结点
3. n0=n2+1 (叶子结点是枝干结点+1)
4. k=floor(log 2 n)+1

满二叉树

完全二叉树

二叉树的遍历

1. 前序遍历:
2. 中序遍历:
3. 后序遍历:

应用

1. 搜索二叉树
2. Huffman树
3. 森林转换为二叉树后的对应遍历序列

算法

最小生成树

最大联通分量

图

概念

储存

最小生成树

Prim

思想

贪心

流程

1. 将点划分为U与TU两部分，一开始TU内只有S。
2. 从U到TU之间的所有边中，选择一个最近的点，将对应点加入TU。
3. 再更新U与TU之间的点距离，继续重复2。直到所有点都加入TU。

代码实现

1. 使用 lowCost[j] 数组表示 点j与TU的距离 。
2. lowCost[j] 若为0，则代表 该点在TU内 。
3. 每次取出最小的 lowCost[j] ，对应的j点记为 k 。
4. 如果 edge[k][j] 小于 lowCost[j] ，将后者更新。其实就是因为k加入了TU，所以要看k向外的所有边的长度，如果这些边的长度短于对应其他边的lowCost，则将他们的lowCost更新。

特殊注意

1. 检测重边
2. lowCost的初始化

代码

```
void Prim(int v) {
  int minn, i,j,k;
  for(i=1;i≤n;i++) lowcost[i] = edge[v][i];
  for(i=1;i<n;i++) { //进行n-1次循环，加入n-1个点
    minn = INF;
    for(j=1;j≤n;j++) {
      if(lowcost[j] && lowcost[j]<minn) {
        minn = lowcost[j];
        k=j;
      }
    }
    lowcost[k] = 0;
    ans1+=minn;
    for(j=1;j≤n;j++) {
      if(edge[k][j]<lowcost[j]) {
        lowcost[j] = edge[k][j];
      }
    }
```

```
  }
}
int main() {
  int m;
  cin>>n>>m;
  int x,y,z;
  for(int i=1;i≤n;i++) {
    for(int j=1;j≤n;j++) {
      edge[i][j] = INF;
      if(i==j) edge[i][j] = 0;
    }
  }
  for(int i=0;i<m;i++) {
    cin>>x>>y>>z;
    if(z<edge[x][y]) edge[x][y] = edge[y][x] = z; //重边检测！很重要！
  }
  Prim(n>>1);
  int ans = 0;
  for(int i=1;i≤n;i++) {
    if(lowcost[i]≠0) {
      cout<<"orz";
      return 0;
    }
  }
  cout<<ans1;
  return 0;
}
```

Kruskal

思想

贪心

流程

1. 选取边权值最小的一条，加入最小生成树。
2. 重复1，除非加入这条边会产生环。
3. 直到所有点都被加入最小生成树，循环结束。

代码实现

1. 使用struct生成edge，设计comp自定义比较函数。初始化，将vset[i]=i；
2. sort一下。按由小到大取边出来。
3. 对这个边，比较其from和end的vset值。若不同，k++，修改所有from的vset为to的vset。
4. 直到k=n，结束循环。

代码

```
struct edge {
  int fr,to,v;
} edges[MAXM];

bool comp(edge a, edge b) {
  return a.v<b.v;
}
int edge1[MAXN][MAXN];
const int INF = 0×3f3f3f;
int main() {
  int m;
  cin>>n>>m;
  int x,y,z;
  for(int i=1;i⩽n;i++) {
    for(int j=1;j⩽n;j++) {
      edge1[i][j] = INF;
    }
  }
  for(int i=1;i⩽m;i++) {
    cin>>x>>y>>z;
    if(edge1[x][y]≠INF) {
      int tmp = edge1[x][y];
      if(edges[tmp].v>z) {
        if(x>y) edges[tmp] = (edge){y,x,z};
        else edges[tmp] = (edge){x,y,z};
      }
    } else {
      edge1[x][y] = edge1[y][x] = cnt;
      if(x>y) edges[cnt++] = (edge){y,x,z};
      else edges[cnt++] = (edge){x,y,z};
    }
  }
  sort(edges, edges+cnt, comp);
  edge tmp;
  int vset[MAXN];
  for(int i=1;i⩽n;i++) vset[i] = i;
  int k = 1, j=0;
  int ans = 0;
  while(k<n) {
    if(j==cnt) {
      cout<<"orz";
      return 0;
    }
    int fr1 = edges[j].fr, to1 = edges[j].to;
    int set1 = vset[fr1], set2 = vset[to1];
    if(set1 ≠ set2) {
      ans += edges[j].v;
      k++;
      for(int i=1;i⩽n;i++)
        if(vset[i] == set2)
          vset[i] = set1;
    }
```

```
53      j++;
54    }
55    cout<<ans;
56  }
```

最短路径

Djikstra

思想

1. 也蛮贪心的，本质上与Prim很像，核心在于松弛方法不同。可以直接见后文。

流程

程序实现

1. 链式前向星+优先队列优化。

代码

*与Prim的异同

区别

Prim:

```
for(j=1;j≤n;j++) {
  if(edge[k][j]<lowcost[j]) {
    lowcost[j] = edge[k][j];
  }
}
```

Djikstra:

```
for(j=1;j≤n;j++) {
  if(dis[k]+edge[k][j]<dis[j]) {
    dis[j] = dis[k]+edge[k][j];
  }
}
```

核心在于，Prim只关注与TU之间的距离，而Djikstra还要关注其与源点的距离，因而会有比较上的不同。

搜索

查找

顺序查找

成功	$\frac{n+1}{2}$
失败	n

二分查找

平均查找长度 $log_2^{(n+1)}-1$

- 求成功与不成功查找次数
  - n较小时：
    - 画出判定树
    - $ASL=(1*n_1+2*n_2+\cdots+m*n_m)/n$ 每层的层数乘以该层点数
  - n较大时：
    - 成功：$\log_2^{(n+1)}-1$
    - 失败/最大：$\lceil \log_2^{(n+1)} \rceil$

Binary Search Tree

思想

构造 左子树上的值都小于根，右子树上的值都大于根

优点 可以以log_n的时间复杂度进行搜索

核心结构

node 一个struct，用于存储data和l、r。

BST 实现如下功能：

1. insert
2. find
3. remove

- 删除
  - 把 右子树 的最小值提上来

remove 四种情况！

```
void removeHelp(node* root, int key) {
  node* tmpNodeForDelete = root;
  if(root→l == NULL && root→r == NULL) {
    //如果是叶子结点，直接删除此结点
    delete root;
    return;
  }
  if(root→l == NULL) {
    //若左子树为空，直接将右子树作为此root值
    root = root→r;
    delete tmpNodeForDelete;
  } else if(root→r == NULL) {
    //若右子树为空，直接将左子树作为此root值
    root = root→l;
    delete tmpNodeForDelete;
  } else {
    //第四种情况。
    //要找到左子树最大的叶子，提上来使用
    //用其值替换root的值，然后删除之
    node* leftMaxLeaf = getLeftMaxLeaf(root);
    int data = leftMaxLeaf→data;
    delete leftMaxLeaf;
    root→data = data;
  }
}
```

AVL

思想

本质 特殊的二分搜索树，它是为了防止某一侧特别长而诞生的

不平衡 左右的高度差大于等于2

修正 通过旋转树

核心结构

node 一个struct，用于存储data和l、r。

判断不平衡

1. 比较两侧高度
2. 比较值与根的左（右）儿子值大小

> 在左枝加点，可能出现LL或LR。通过key的值判断究竟是LL还是LR，如果key小于root→l的data，它会加在左边，即LL；反之就是LR。

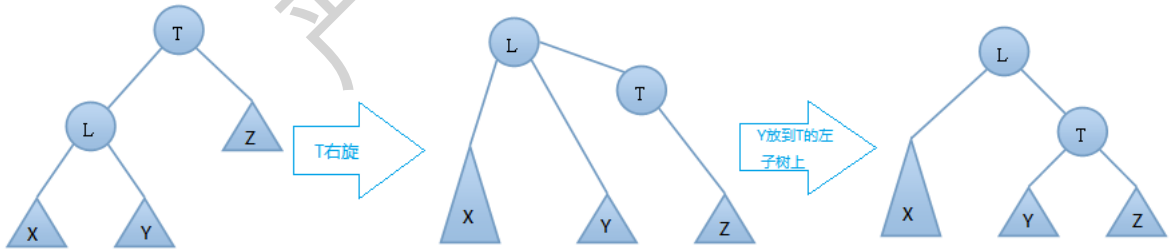
```
if(key < data) {
  root→l = insertHelp(root→l, key);
```

```
  //在左枝加点，可能出现LL或LR
  if(getHeight(root->l) - getHeight(root->r) == 2) {
    //通过key的值判断究竟是LL还是LR
    //如果key小于root->l的data，它会加在左边，即LL
    //反之就是LR
    if(key < root->l->data) { //LL
      root = rotateLL(root);
    } else {
      root = rotateLR(root);
    }
  }
}
else if(key > data) {
  root->r = insertHelp(root->r, key);
  //在右枝加点，可能出现RR或RL
  if(getHeight(root->r) - getHeight(root->l) == 2) {
    //通过key的值判断究竟是RR还是RL
    //如果key小于root->l的data，它会加在左边，即RL
    //反之就是RR
    if(key < root->r->data) { //LL
      root = rotateRL(root);
    } else {
      root = rotateRR(root);
    }
  }
}
```

旋转

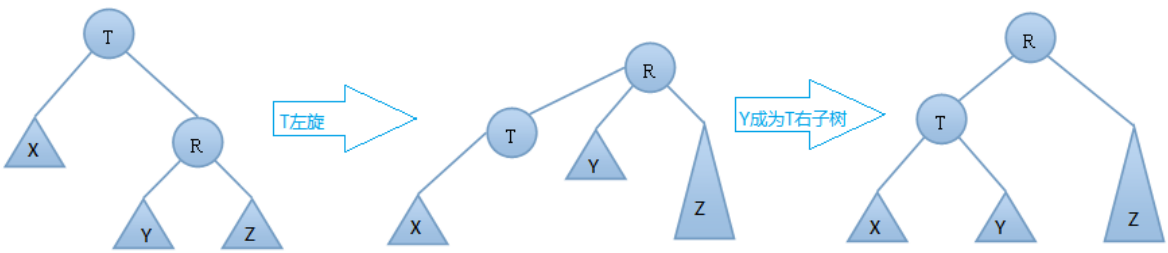
分为四个类型：

LL



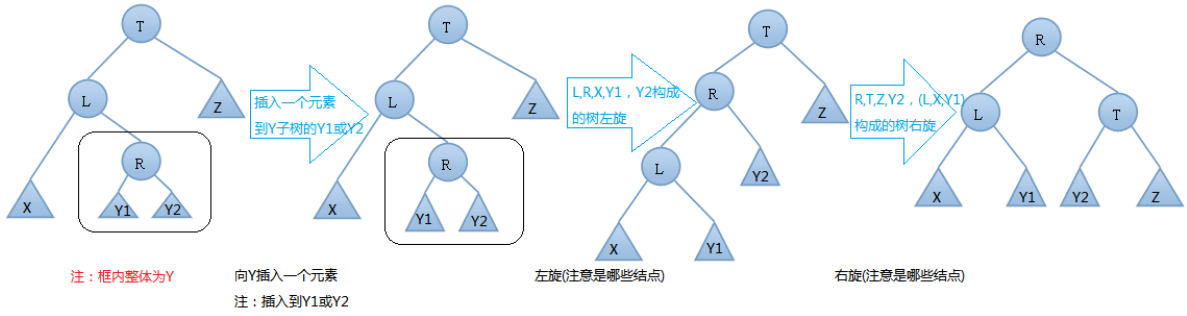
左左情况的右旋结果

RR

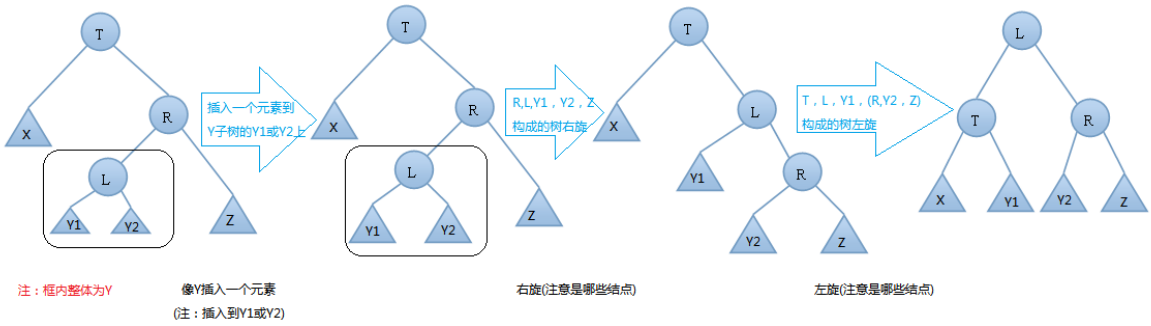


右右情况的左旋结果

LR



RL




```
node* rotateRR(node *root) {
  //a→b→c a←b→c
  node *newRoot = root→r;
  root    →r = newRoot→l;
  newRoot→l = root;
  return newRoot;
}
node* rotateLL(node *root) {
  //a←b←c a←b→c
  node *newRoot = root→l;
  root    →l = newRoot→r;
  newRoot→r = root;
  return newRoot;
}
node* rotateLR(node *root) {
  root→l = rotateRR(root→l);
  return rotateLL(root);
}
node* rotateRL(node *root) {
  root→r = rotateLL(root→r);
  return rotateRR(root);
}
```

Hash

概念

哈希表 一个表，用于储存

哈希函数 $H(key1) = H(key2)$

哈希冲突 $key1 \neq key2, H(key1) = H(key2)$

构造哈希表

直接定址法

取对应变量的某个线性函数值为哈希地址$H(key)=a\times key+b,\quad a,b$是常数

平方取中法

可以保留更多原信息

除留余数法

$H(key)=key\mod p,p\leq m$

非常简单。但是如果p选的不好，可能会出现很多同义词。

处理冲突的方法

- 开散列方法
  - 拉链法
- 闭散列方法
  - 桶式散列法
    - 对每个值保留n个桶，如果每个值有多个数据，那就在这个桶里放
  - 线性探查法
    - 1
    - 常数
    - 伪随机
    - 二次
  -

线性探查法

从发生冲突的地址(d)开始，依次探查d的下一个地址（当到达m-1的表尾，从0处重新开始），直到找到一个空闲单元为止。

$H_i(key)=(H(key)+i)\mod m\ (1\leq i\leq m-1)$

但是其实这样也容易产生堆积问题。

平方探查法

$d=(d+1^2,d-1^2,d+2^2,d-2^2,\ldots)$
$H_i(key)=(H(key)+d_i)\mod m\ (1\leq i\leq m-1)$

拉链法

排序

基础概念

稳定 排序后有相同关键字的元素之间的相对次序是否改变

基本思想

将一个待排序的元素按其关键字值的大小插入到已有序的子表中的适当位置，直到全部插入完成。

直接插入排序

直接把元素插入一个已经有序的子表里。假设元素存放在R[0..n-1]中，R[0..i-1]是已经排序好的子表，取出R[i]将之插入到R[0..i-1]的适当位置（通过遍历来查找位置）。

平均	最坏	最坏条件	最好	最好条件	比较次数	移动次数	排序趟数	空间复杂度	稳定性	有序区
$O(n^2)$	$O(n^2)$	【初始逆序】	$O(n)$	【初始正序】	n^2	n^2	n	$O(1)$	稳定	全局有序

折半插入排序

直接把元素插入一个已经有序的子表里。假设元素存放在R[0..n-1]中，R[0..i-1]是已经排序好的子表，取出R[i]将之插入到R[0..i-1]的适当位置（通过二分搜索来查找位置）。

平均	最坏	最坏条件	最好	最好条件	比较次数	移动次数	排序趟数	空间复杂度	稳定性	有序区
$O(n^2)$	$O(n^2)$	【初始逆序】	$O(n)$	【初始正序】	$n \cdot \log_2^n$	n^2	n	$O(1)$	稳定	全局有序

希尔排序

选择排序

基本思想

每步从待排序的元素中选出关键字最小的元素，放到已排序的序列的最后。

简单选择排序

从$R[i..n-1]$(无序区)遍历找出最小值，然后将其与$R[i]$交换。

如此，可以保证$R[0..i]$是全局有序的。

平均	最坏	最坏条件	最好	最好条件	比较次数	移动次数	排序趟数	空间复杂度	稳定性	有序区
$O(n^2)$	$O(n^2)$	无	$O(n^2)$	无	n^2	n	n	$O(1)$	不稳定	全局有序

特点

1．简单选择排序的效率与初始数据的次序无关。
2．由于每次都在选，所以不稳定。（？？？

堆排序

用维护堆的方式，取代了之前的比较方法。堆产生一个最小（大）值只需要log_2^n的时间复杂度，所以会更优秀。

平均	最坏	最坏条件	最好	最好条件	比较次数	移动次数	排序趟数	空间复杂度	稳定性	有序区
$O(n\cdot\log_2^n)$	$O(n\cdot\log_2^n)$	无	$O(n\cdot\log_2^n)$	无	$n\cdot\log_2^n$	n	n	$O(1)$	不稳定	全局有序

特点

1．简单选择排序的效率与初始数据的次序无关。

交换排序

基本思想

两两比较待排序元素的关键字，交换不满足次序要求的对，直到满足要求为止。

冒泡排序

通过无序区元素的相互比较和位置的交换，使最小的元素一直向上漂浮。

从最下面的元素开始，对每两个相邻元素的关键字进行比较，且使关键字较小的元素换至关键字较大的元素前，经过一趟冒泡排序之后最小值抵达最上端。

平均	最坏	最坏条件	最好	最好条件	比较次数	移动次数	排序趟数	空间复杂度	稳定性	有序区
$O(n^2)$	$O(n^2)$	初始反序	$O(n)$	初始正序	n^2	n^2	n	$O(1)$	稳定	全局有序

快速排序

由冒泡排序改进而来。基本思想是从待排序的元素中任取一个（一般第一个）作为基准，把基准放入最终位置后？整个区间被其分割成两个部分，然后把所有关键字比基准小的放在前子区间内，所有比基准大的放在后子区间内，并把基准排在中间，这称为一趟或划分。然后对两个子区间进行同样的步骤，直至每个子区间内只有一个元素或为空为止。

平均	最坏	最坏条件	最好	最好条件	比较次数	移动次数	排序趟数	空间复杂度	稳定性	有序区
$O(n\cdot\log_2^n)$	$O(n^2)$	初始正序	$O(n\cdot\log_2^n)$	随机	?	?	$\log_2^n$?	$O(\log_2^n)$	不稳定	无

归并排序

将R[0..n-1]看成n个长度为1的有序表，将相邻的有序表成对归并（并在这个过程中排序），得到n/2个长度为2的有序表，再将他们成对归并下去，直到最终得到一个长度为n的有序表。上述称为二路归并排序。

平均	最坏	最坏条件	最好	最好条件	比较次数	移动次数	排序趟数	空间复杂度	稳定性	有序区
$O(n\cdot\log_2^n)$	$O(n\cdot\log_2^n)$	无	$O(n\cdot\log_2^n)$	无	?	?	$\log_2^n$	$O(n)$	稳定	不全局有序

时间效率与待排序数据无关

基数排序

平均	最坏	最坏条件	最好	最好条件	比较次数	移动次数	排序趟数	空间复杂度	稳定性	有序区
$O(d(n+r))$	$O(d(n+r))$	无	$O(d(n+r))$	无	?	?	d	$O(n+r)$	稳定	不一定

总结

类别	名称	平均	最坏	最坏	最好	最好	比较次数	移动次数	排序趟数	空间复杂度	稳定性	有序区
插入	直接插入排序	$O(n^2)$	$O(n^2)$	逆序	$O(n)$	正序	n^2	n^2	$n-1$	$O(1)$	稳定	全局有序
	希尔排序	$O(n^{1.5})$	$O(n^{1.5})$	逆序		正序				$O(1)$	不稳定	无
选择	简单选择排序	$O(n^2)$	$O(n^2)$	无	$O(n^2)$	无	n^2	n	$n-1$	$O(1)$	不稳定	全局有序
选择	堆排序	$O(n\cdot\log_2^n)$	$O(n\cdot\log_2^n)$	无	$O(n\cdot\log_2^n)$	无	$n\cdot\log_2^n$	n	$n-1$	$O(1)$	不稳定	全局有序
交换	冒泡排序	$O(n^2)$	$O(n^2)$	逆序	$O(n)$	正序	n^2	n^2	不一定	$O(1)$	稳定	全局有序
交换	快速排序	$O(n\cdot\log_2^n)$	$O(n^2)$	正序	$O(n\cdot\log_2^n)$	随机	?	?	（最好）$\lceil\log_2^n\rceil$	$O(\log_2^n)$	不稳定	无
归并	归并排序	$O(n\cdot\log_2^n)$	$O(n\cdot\log_2^n)$	无	$O(n\cdot\log_2^n)$	无	?	?	$\lceil\log_2^n\rceil$	$O(n)$	稳定	不全局有序
基数	基数排序	$O(d(n+r))$	$O(d(n+r))$	无	$O(d(n+r))$	无	?	?	d	$O(n+r)$	稳定	不一定

应试优化

1. 归位：选择+交换
2. 稳定：简单插入+冒泡+2

3. 空间：快速(logn) 归并(n) 基数(n+r) 其余(1)
4. 与初序无关：选择+2