浙大数据结构

基本概念

数据结构

解决问题方法的效率和下面三个因素有关

- 数据的存储方式
- 空间的利用率
- 算法的巧妙程度

数据结构就是数据对象在计算机中的组织方式

- 逻辑结构
- 物理结构

抽象数据类型ADT（Abstract Data Type）

算法Algorithm及复杂度分析

复杂度有

- 空间复杂度S(n)
- 时间复杂度T(n)

在分析一般算法效率时，我们主要关注

- 最坏情况复杂度$T_{worst}(n)$
- 平均复杂度$T_{avg}(n)$

大O表示法

🌈

应用实例：最大子列和问题

n个整数，求出最大连续子列和

算法一：暴力遍历所有子列$O(N^3)$

```
int Solution1(int A[], int N) {
 int thisSum = 0, maxSum = 0;
 for (int i = 0; i < N; ++i) {
     for (int j = i; j < N; ++j) {
         thisSum = 0;
         for (int k = i; k < j; ++k) {
             thisSum += A[k];
         }
         if (thisSum > maxSum) {
             maxSum = thisSum;
         }
     }
 }
 return maxSum;
}
```

算法二：每一个子列的求和不需要重新计算，只需要在前一个的基础上加上当前元素$O(N^2)$

```
int Solution2(int A[], int N) {
 int thisSum = 0, maxSum = 0;
 for (int i = 0; i < N; ++i) {
     thisSum=0;
     for (int j = i; j < N; ++j) {
         thisSum += A[j];
         if (thisSum > maxSum) {
             maxSum = thisSum;
         }
     }
 }
 return maxSum;
}
```

算法三：分治法$O(nlog_n)$

```cpp
int calc(int l, int r, vector<int> &A) {
 if (l == r) {
     return A[l];
 }
 int mid = (l + r) >> 1;
 int leftSum = 0, rightSum = 0, leftMax = 0, rightMax = 0;
 for (int i = mid; i >= l; --i) {
     leftSum += A[i];
     if (leftMax < leftSum) {
         leftMax = leftSum;
     }
 }
 for (int i = mid + 1; i <= r; ++i) {
     rightSum += A[i];
     if (rightMax < rightSum) {
         rightMax = rightSum;
     }
 }
 return max(max(calc(l, mid, A), calc(mid + 1, r, A)), rightMax + leftMax);
}

int Solution3(vector<int> &A) {
 int N = A.size();
 return calc(0, N - 1, A);
}
```

算法四：在线处理$O(n)$

```cpp
int Solution4(vector<int> &A) {
 int N = A.size();
 int thisSum = 0, maxSum = 0;
 for (int i = 0; i < N; ++i) {
     thisSum += A[i];
     if (thisSum > maxSum) {
         maxSum = thisSum;
     } else if (thisSum < 0) {
         thisSum = 0;
     }
 }
 return maxSum;
}
```

线性结构

线性表list

- 利用数组的顺序存储实现
- 利用链表的链式存储实现
    - 广义表：元素自身是一个链表，在C语言中使用union实现
    - 多重链表：链表中的节点可能同时隶属于多个链

堆栈&队列

堆栈可以用于表达式求值

> 🌈
>
> 应用：表达式求值（中缀表达式转后缀表达式）
>
> 1. 遇到数字需要直接输出，但是有时数字可能不只是一个个位数，因此需要遍历表达式，获取该值。
> 2. 如果运算符栈为空，如果遇到运算符，直接入栈。
> 3. 如果遇到"("，直接入栈。
> 4. 如果遇到")"，连续出栈，一直到栈顶元素是"("，然后出栈"("。

5. 如果遇到运算符且运算符栈不为空，此时需要比较当前运算符和栈顶运算符的优先级。分两种情况：
    1. 当前运算符优先级大于栈顶运算符优先级，直接入栈。
    2. 当前运算符优先小于栈顶运算符优先级，需要一直出栈，直到栈顶运算符优先小于当前运算符，将当前运算符入栈。

堆栈其他用途

- 函数调用和递归实现
- 深度优先搜索
- 回溯算法

树

分层次组织在管理上具有更高的效率

引子--查找

- 静态查找：集合中记录是固定的
    1. 顺序查找
    2. 二分查找（条件是必须有序）

```cpp
bool BinarySearch(vector<int> &v, int k) {
    int n = v.size();
    int left = 0, right = n - 1, mid;
    while (left <= right) {
        mid = (left + right) >> 1;
        if (v[mid] < k)
            left = mid + 1;
        else if (v[mid] > k)
            right = mid - 1;
        else
            return true;
    }
    return false;
```

```
14 }
```

- 动态查找：集合中记录可能有插入和删除

树

树的特点

- 子树不相交
- 除了根节点，每个节点有且仅有一个父节点
- 一颗N结点的树有N-1条边

树的表示方法

- 父亲-兄弟表示
- 儿子-兄弟表示，节省空间

二叉树

子树有左右之分的度为2的树

- 斜二叉树
- 完美二叉树/满二叉树
- 完全二叉树

性质

- 二叉树的第i层的最大结点数为：2^{i-1}
- 深度为k的二叉树最大结点总数为：2^k-1
- 对于任何非空二叉树T，若n_i表示度为i的结点个数，则有：$n_0=n_2+1$

存储结构

- 顺序存储（也就是使用数组，适合存储完全二叉树，存储一般二叉树会浪费空间）
  - 非根节点（序号为i）的父节点序号是$\lfloor i/2 \rfloor$
  - 结点（序号为i）的左孩子序号是2i，右孩子是2i+1

- 链表存储

二叉树遍历(二维结构的线性化)

1. 先序/中序/后序遍历：使用递归
2. 中序遍历非递归实现

```
// 每个结点都会被push到栈中，然后后面又会被pop出来，当被pop出来之后就访问了，并且下一步会往它的右子树去操作。
void inOrderTraversal(BinaryNode *root) {
    BinaryNode *p = root;
    stack<BinaryNode *> stack1;
    while (p || !stack1.empty()) {
        while (p) {
            stack1.push(p);
            p = p->left;
        }
        if (!stack1.empty()) {
            p = stack1.top();
            stack1.pop();
            cout << p->data << endl;
            p = p->right;
        }
    }
}
```

3. 先序遍历非递归实现

```
// 每个结点都会被push到栈中，然后后面又会被pop出来，当被push到栈中之前就访问。
void preOrderTraversal(BinaryNode *root) {
    BinaryNode *p = root;
    stack<BinaryNode *> stack1;
    while (p || !stack1.empty()) {
        while (p) {
            cout << p->data << endl;// 只改动了这一行
            stack1.push(p);
            p = p->left;
        }
        if (!stack1.empty()) {
```

```
            p = stack1.top();
            stack1.pop();
            p = p->right;
        }
    }
}
```

4. 后序遍历非递归实现：不能通过在先序或中序遍历的基础上移动cout语句实现，因为根结点无法在右结点或右子树后前弹出。可以借助两个堆栈实现，其中一个堆栈用于储存中间结点，一个堆栈用于储存遍历顺序。

```
void lateOrderTraversal(BinaryNode *root) {
    BinaryNode *p = root;
    stack<BinaryNode *> stack1;
    stack<BinaryNode *> stack2;
    while (p || !stack1.empty()) {
        while (p) {
            stack1.push(p);
            stack2.push(p);
            p = p->right;
        }
        if (!stack1.empty()) {
            p = stack1.top();
            stack1.pop();
            p = p->left;
        }
    }
    while (!stack2.empty()) {
        cout << stack2.top()->data << endl;
        stack2.pop();
    }
}
```

5. 层序遍历

```
void levelOrderTraversal(BinaryNode *root) {
    queue<BinaryNode *> queue1;
    queue1.push(root);
    while (!queue1.empty()) {
        auto curr=queue1.front();
```

```
            cout << curr->data << endl;
            queue1.pop();
            if (curr->left != nullptr) {
                queue1.push(curr->left);
                cout << "dddd" << endl;
            }
            if (curr->right != nullptr) {
                queue1.push(curr->right);
            }
        }
    }
```

树的应用

二叉搜索树

对于任何一个结点，其左子树所有结点键值小于该节点键值，右子树所有结点键值大于该节点键值。

几种操作：

- 查找find
  - 递归实现
  - 循环实现
- 查找最大元素和最小元素
- 插入：要记录父节点
  - 递归实现：每一层递归的返回值都是左子树或右子树插入该元素之后的根节点
- 删除
  - 待删除结点是叶结点直接删除
  - 待删除结点只有一个孩子
  - 待删除结点有两个孩子
    - 右子树最小元素代替待删除元素并在右子树删除该最小元素
    - 左子树最大元素代替待删除元素并在左子树删除该最大元素

平衡二叉树（AVL树）

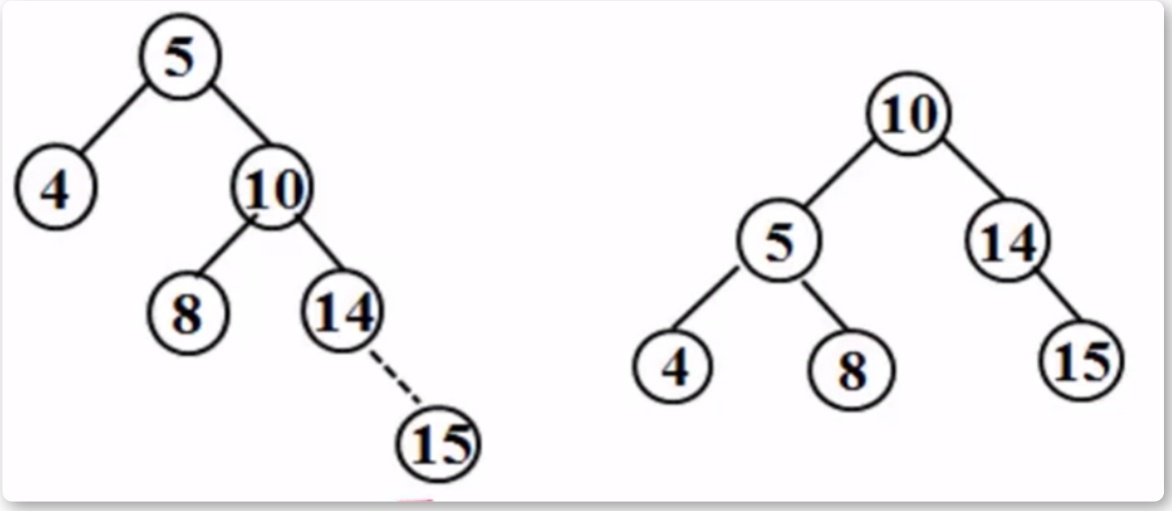
平衡因子被定义为左右子树高度之差，平衡二叉树就是任一子树的平衡因子不超过1.

定义高度为n的二叉树最少有$f(n)$个结点，则有

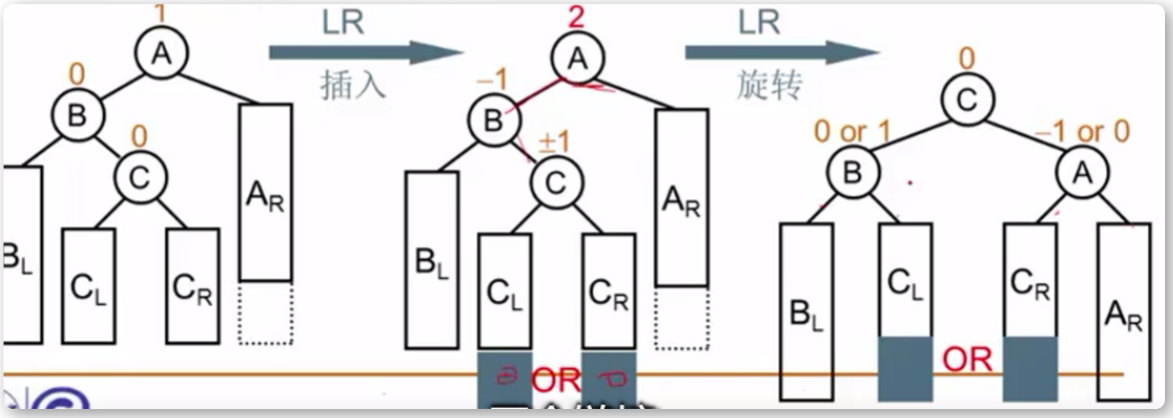
$$f(n) = f(n-1) + f(n-2) + 1$$

平衡二叉树的调整（插入新结点后可能不再平衡）

- RR旋转



- LL旋转
- LR旋转



- RL旋转

堆

优先队列：一种特殊的队列，其取出元素的顺序是依照元素的优先权大小，而不是元素进入队列的先后顺序。

可以使用数组或者链表实现优先队列，但是就会面临的问题是如果插入简单（插入时不排序），查找操作就会相应复杂；如果想要查找时相对简单，那就需要更复杂的插入操作。

如果使用二叉搜索树简单地实现优先队列的话，将会面临的问题是如果每次都删除最大的元素（最右边的元素），树就会不平衡，需要复杂的调整操作。

最优的方法是使用完全二叉树（存储方式可以使用数组，且A[0]不存元素，存哨兵），令根节点的优先级（关键字）大于任意子树结点，这种叫做最大堆，相应的也有最小堆。

堆的操作

- 插入：先插到完全二叉树末尾，然后可能需要向上调整位置。
- 删除：删除的是根节点，此时将末尾元素提到根的位置作为根，然后可能需要向下调整位置（和较大的儿子交换位置）
- 建堆：先将各个元素存入数组，然后进行调整（从倒数第一个有儿子的结点开始向下调整，调整所有非叶子节点）

哈夫曼树和哈夫曼编码

核心思想是根据结点不同的查找频率构造更有效的搜索树--哈夫曼树

带权路径长度WPL

WPL最小的二叉树就叫最优二叉树或哈夫曼树。

构造哈夫曼树的思想：每次把权值最小的两棵二叉树合并（利用堆可以从一堆元素中找到两个最小的元素）

哈夫曼树的特点：

- 没有度为1的结点
- n个叶子结点的哈夫曼树共有2n-1个结点，因为$n_0 = n_2 + 1$

- 同一组权值，存在不同构的哈夫曼树

哈夫曼编码问题：如何对字符进行编码，使得该字符串的编码存储空间最少。

如何避免编码二义性：保证任何字符的编码都不是另一字符编码的前缀码，可以使用二叉树编码，字符只在叶节点上面。

集合

并查集问题：集合并，查某元素属于什么集合

集合存储的实现：使用树表示集合，每个结点代表一个集合元素，可以使用链表实现双亲表示法的树或者使用数组实现树（也是双亲表示法）。

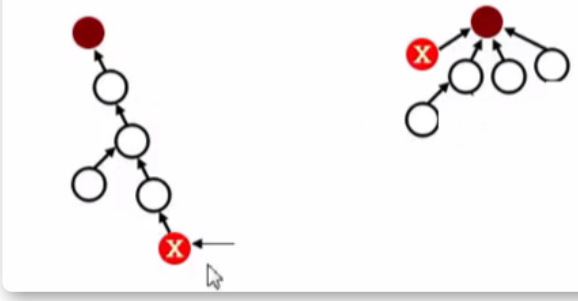
集合的查找运算就很简单，可以直接按顺序查找到该元素所属集合的根结点。

集合的并运算就是将一个集合的根节点的父亲指针设置为另一个集合的根结点，为了改善合并后的查找性能，尽量将小的集合合并到相对大的集合中，可以利用根节点的父亲指针存储集合大小。

> 🌈
>
> file transfer问题：十台独立计算机，在给某些计算机之间插上网线后，查询某两台计算机之间是否可以及进行文件传输。
>
> 其实就是并查集问题。
>
> 插网线其实就是在做集合的并集，查询是否可以传输文件就是查询两个元素是否属于同一集合，也就是查询根节点。
>
> 按秩归并：在并的时候，我们应该让矮树贴到高树上，可以提高查找效率。
>
> 路径压缩：

```
SetName Find( SetType S, ElementType X )
{  if ( S[X] < 0 ) /* 找到集合的根 */
      return X;
   else
      return S[X] = Find( S, S[X] );
}
```

先找到根；
把根变成 X 的父结点；
再返回根。



图

概念和表示方法

- 无向图
- 有向图
- 网络：带权重的图

图的表示方法

- 邻接矩阵：对于无向图，浪费了一半空间，我们可以使用一维数组存储上三角或者下三角。对于网络直接使用矩阵元素值表示权重即可。
  - 方便检查任一对顶点间是否有边
  - 方便查找任一顶点的邻接点
  - 方便计算任一顶点的度
  - 但是对于稀疏图，浪费空间
- 邻接表：为矩阵每行定义一个链表，只存非0元素
  - 方便查找任一顶点的邻接点
  - 需要N个头指针 + 2E个结点，每个结点至少两个域，E为边的个数
  - 对于无向图方便计算任一顶点的度，但有向表只能计算出度

图的遍历

深度优先搜索Depth First Search（DFS）

- 可以递归实现
- 记录每个结点的访问状态
- 对于一个初始结点，先访问它，然后对它的每个未被访问的邻接点做DFS，类似于树的先序遍历
- 邻接表$O(N+E)$
- 邻接矩阵$O(N^2)$

广度优先搜索Breadth First Search(BFS)

- 类似于树的层序遍历
- 使用队列，压入初始结点，进入循环，只要队列不是空，就访问队列头元素并弹出，然后将所有这个结点的邻接未访问结点放到队列尾部。
- 邻接表$O(N+E)$
- 邻接矩阵$O(N^2)$

几个概念：

- 连通、路径、回路、连通图
- 简单路径（路径中没有重复顶点）
- 连通分量：无向图的极大连通子图
- 强连通：有向图中某两个顶点存在双向路径，则称强连通
- 强连通图、强连通分量

> 🌈
>
> 应用实例：拯救007、六度空间

图的应用

最短路径问题

- 单源最短路径问题
- 多源最短路径问题

无权图单源最短路径：使用BFS可以分别找到某个点到所有点的最短路径，并且可以将路径记录下来。

有权图单源最短路径（要保证图中没有负值圈--路径权值总和为负的回路），这里使用dijkstra算法（无法解决有负权值的图）：

- S={源点s + 已经确定了最小路径的顶点v_i}
- 对于任一未收录的顶点v，定义dist[v]为s到v的最短路径长度，但该路径仅经过S中的顶点。
- 路径必须按照非递减顺序生成，这样真正的最短路径才只经过S中的顶点
- 每次从未收录的顶点中选一个dist最小的收录
- 增加一个v进入s，可能会影响另一个点的dist值，要做更新

```
void Dijkstra( Vertex s )
{ while (1) {
    V = 未收录顶点中dist最小者;
    if ( 这样的V不存在 )
      break;
    collected[V] = true;
    for ( V 的每个邻接点 W )
      if ( collected[W] == false )
        if ( dist[V]+E<V,W> < dist[W] ) {
          dist[W] = dist[V] + E<V,W> ;
          path[W] = V;
        }
  }
}
   /* 不能解决有负边的情况 */
```

有权图多源最短路径，使用Floyd算法

```
void Floyd()
{   for ( i = 0; i < N; i++ )
      for( j = 0; j < N; j++ ) {
        D[i][j] = G[i][j];
        path[i][j] = -1;
      }
    for( k = 0; k < N; k++ )
      for( i = 0; i < N; i++ )
        for( j = 0; j < N; j++ )
          if( D[i][k] + D[k][j] < D[i][j] ) {
            D[i][j] = D[i][k] + D[k][j];
            path[i][j] = k;
          }
}
```

最小生成树问题

生成树：包含了全部图中顶点且树的所有边都属于图，生成树中任意加一条边都一定构成回路

最小生成树：权值总和最小的生成树

Prim算法：让一棵小树长大，不断加入到小树最近的点加入小树，适合稠密图

Kruskal算法：将森林合并成树，相当于不停收集最小边，适合稀疏图

拓扑排序问题

有向无环图（Directed Acycline Graph, DAG）是一类特殊的有向图。DAG有着广泛应用，AOE网和AOV网都是DAG的典型应用。

AOV网（Activity On Vertex NetWork）用顶点表示活动，边表示活动（顶点）发生的先后关系。AOV网的边不设权值，若存在边<a,b>则表示活动a必须发生在活动b之前。

拓扑排序就是按照事件发生依赖条件将事件进行排序，使得后面的事件只依赖排在前面的某些事件。

只需要不停找到没有前置结点的结点输出并去除即可。可以将入度变为0的顶点放入一个容器，以提高算法效率。

关键路径问题

AOE网(Activity On Edge Network)是边表示活动的网，AOE网是带权有向无环图。边代表活动，顶点代表所有指向它的边所代表的活动已完成这一事件。由于整个工程只有一个起点和一个终点，网中只有一个入度为0的点（源点）和一个出度为0的点（汇点）。

求整个工期：找到每个活动的最早完成时间

机动时间：不能耽误的活动的最晚完成时间，从总时间向前回推最晚需要完成时间

关键路径就是最早完成时间=最晚完成时间的活动路径

排序

这里我们只讨论内部排序，并在每种算法中考虑其稳定性（任意两个相等的元素在排序前后相对位置不发生改变），没有一种排序是任何情况下都表现最好的。

简单排序

冒泡排序bubbleSort

冒泡排序跑第i趟：通过两两比较并移动相邻的前n-i个元素，将当前最大的元素移动到有序部分的前面。

一种优化方式是当某一趟并没有发生元素移动时，说明已经完成排序，直接跳出循环。

逆序下是最坏情况：$O(N^2)$

是稳定的

插入排序insertionSort

插入排序跑第i趟：选中第i个元素将其向左移动到符合自己大小的位置上。

比冒泡排序好在不需要频繁交换元素位置，只需要记住当前元素即可。

逆序下是最坏情况：$O(N^2)$

时间复杂度下界

对于下标i<j，如果有A[i]>A[j]，称(i, j)是一对逆序对。

对于冒泡和插入排序，他们都是交换相邻元素，每交换一次正好消去一个逆序对，所以T(N,I)=O(N+I)，其中I为逆序对个数。

任意序列平均逆序对是$O(N^2)$，所以任何仅交换相邻元素排序的算法，其平均时间复杂度为$\Omega(N^2)$

想要提高算法效率

- 每次消去不止一个逆序对
- 每吃交换相隔较远的2个元素

希尔排序

定义希尔增量序列，比如5，3，1

先做5间隔插入排序，再做3间隔插入排序，再做1间隔插入排序就会简单很多。

做完n间隔插入排序后，再做小于n间隔的排序仍然能保证n间隔有序。

选择排序

和冒泡、插入一样属于简单排序

选择排序的第i趟：从第i个元素到最后一个元素中选出最小元素，并将其换到有序部分的最后。

无论如何时间复杂度都是N^2

堆排序

属于选择排序的优化

一种实现方式是将数组当作堆，将其调整为最小堆，然后不断选出最小元素并存储即可完成排序，但是需要额外O(N)的空间。

另一种更好的实现方式是将当前数组调整为最大堆，然后将最大元素和最后一个位置交换，然后重新调整堆（不考虑已有序部分），$O(Nlog_N)$。

归并排序

核心：有序子序列合并，每个元素扫描一次并存入一次，O(N)

归并排序的第一种实现方式是递归实现，使用到了分治的思想：对于一个数组，将其分为等长的两部分，将这两部分归并排序的结果进行有序子序列合并即可，在合并过程中需要使用相应的临时数组去保存合并的结果，时间复杂度是严格的$O(Nlog_N)$，需要提前申请空间N的一个数组，这样就不需要频繁申请释放。

另一种实现是非递归的实现，首先归并长度为1的子序列，然后对于所有序列反复做相邻的两两归并即可，同样需要需要提前申请空间N的一个数组存放归并完成的数组。

是稳定的，一般用于外排序

快速排序

和归并排序一样，也是分而治之的策略。

其步骤是对当前序列选出主元，根据其他元素和主元的大小情况将其分为两组（采用双指针法，一个从左找大的，一个从右找小的，然后交换位置即可），然后再分别对这两组元素继续做类似递归的的划分，将划分好的两组和主元放到一起就是结果。

快排的问题，因为使用了递归，所以对于小规模的数据可能还不如插入排序快，所以当递归的数据规模充分小时侯停止递归，直接使用简单排序，如插入排序。

物理排序

基数排序

桶排序：桶排序(Bucket sort)或所谓的箱排序，是一个排序算法，工作的原理是将数组分到有限数量的桶子里。每个桶子再个别排序（有可能再使用别的排序算法或是以递归方式继续使用桶排序进行排序）。 桶排序是鸽巢排序的一种归纳结果。

多关键字排序：对某一个关键字建桶，在桶内排序。

排序算法比较

排序方法	平均时间复杂度	最坏情况时间复杂度	额外空间复杂度	稳定性
简单选择排序	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
直接插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
希尔排序	$O(n^d)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(NlogN)$	$O(NlogN)$	$O(1)$	不稳定
快速排序	$O(NlogN)$	$O(n^2)$	$O(logN)$	不稳定
归并排序	$O(NlogN)$	$O(NlogN)$	$O(n)$	稳定
基数排序	$O(P(N+B))$	$O(P(N+B))$	$O(N+B)$	稳定

散列

已知的查找方法：

- 顺序查找
- 二分查找（静态查找）
- 二叉搜索树
- 平衡二叉搜索树

散列是一种新的查找方法，就是根据关键词计算出存储位置，但是要进行冲突解决。所以需要

- 散列函数

- 冲突解决策略

散列函数

- 关键字为数字
  - 直接定址法：取关键字的某个线性函数值为散列地址
  - 除留余数法：取余
  - 数值分析法：取手机号码后四位
  - 折叠法：分割叠加
  - 平方取中法：取平方值的中间三位
- 关键字为字符
  - ASCII码加和法：ASCII码相加取余
  - 前三个字符移位法
  - 移位法

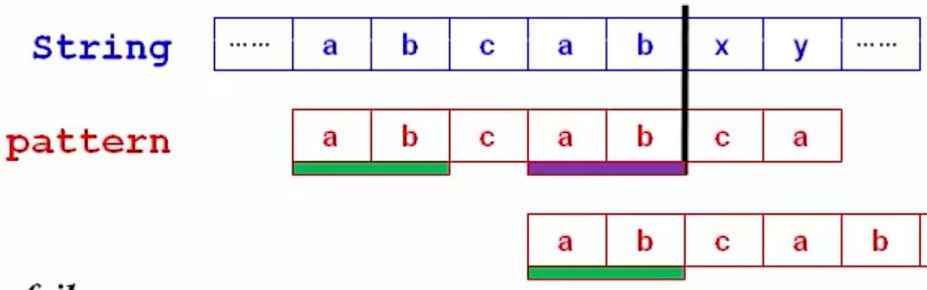
冲突解决策略

- 开放地址法：利用规则去找下一个位置，发生第i次冲突，就去找该位置后面的第d_i个位置
  - 线性探测：$d_i = i$
  - 平方探测：$d_i = i^2$
  - 双散列
- 分离链接法：将相应位置上冲突的所有关键字存储在同一个单链表中

串的模式匹配（KMP算法）

目标是查找pattern在string中第一次出现位置。

KMP算法O(m+n)



failure

$$match(j)=\begin{cases}\text{满足 } p_0\cdots p_i=p_{j-i}\cdots p_j \text{ 的最大} i\,(<j)\\ -1 \quad \text{如果这样的} i \text{ 不存在}\end{cases}$$

pattern	a	b	c	a	b	c	a	c	a	b
j	0	1	2	3	4	5	6	7	8	9
match	-1	-1	-1	0	1	2	3	-1	0	1

```
Position KMP( char *string, char *pattern )
{   int n = strlen(string);      /* O(n) */
    int m = strlen(pattern);     /* O(m) */
    int s, p, *match;
    if ( n < m ) return NotFound;
    match = (int *)malloc(sizeof(int) * m);
    BuildMatch(pattern, match); /* Tm = O(?) */
    s = p = 0;
    while (s<n && p<m) { /* O(n) */
        if (string[s]==pattern[p]) { s++; p++; }
        else if (p>0) p = match[p-1]+1;
        else s++;
    }
    return (p == m)? (s-m) : NotFound;
}
```

```
void BuildMatch(char *pattern, int *match)
{   int i, j;
    int m = strlen(pattern); /* O(m) */
    match[0] = -1;
    for (j=1; j<m; j++) {  /* O(m) */
        i = match[j-1];
        while ((i>=0) && (pattern[i+1]!=pattern[j]))
            i = match[i];
        if (pattern[i+1]==pattern[j])
            match[j] = i+1;
        else match[j] = -1;
    }
}
```

i 回退的总次数不会超过 i 增加的总次数

$$T_m = O(\mathrm{m})$$