数据结构

1 基础知识点

- 数据元素 数据项 学生记录是一个数据元素，一个学生记录由多个数据项组成
- 数据类型有：原子类型、结构类型、抽象数据类型（数据对象、数据关系、基本操作）
- 数据结构有：逻辑结构、存储结构、数据的运算
- 逻辑结构：线性结构（栈、广义表等）、非线性结构（集合、树形、网状）
- 存储结构：顺序存储（随机存取）、链式存储（顺序存取）、索引存储（随机）、散列存储（随机，有冲突）
- 易错点：循环队列是用顺序表（顺序存储结构的单链表）表示的队列；栈是一种逻辑结构，可以使用顺序存储或者链式存储实现
- $T(n) = T_1(n) + T_2(n) = O(max(f(n), g(n)))$
- $T(n) = T_1(n) * T_2(n) = O(f(n) * g(n))$

2 线性表（逻辑结构）

- 顺序表：线性表的顺序存储
- 单链表：线性表的链式存储（非随机存取）
- 引入头结点的优点：无需对第一个结点进行特殊处理
- 双链表：既有next又有prior
- 循环单链表：单链表尾指针指向头结点
- 循环双链表
- 静态链表：以next=-1作为结束的标志；顺序存储，顺序存取；插入删除和其他链表一样，不移动元素，只修改指针
- 当顺序表无序，时间复杂度O(n)；顺序表有序时，时间复杂度O(log(n))
- 建立一个有序链表的最小时间复杂度是O(nlog(n))

3 栈（逻辑结构）

- 只能一端插入、删除的线性表
- 共享栈 为了有效的利用存储空间，让两个栈共享一个存储空间；对存储效率没有影响
- 易错点：注意开始时候栈顶元素的位置
- 技巧 对于n个元素进栈，出栈序列为 $\frac{1}{n+1}C_{2n}^{n}$
- 易错点：C语言中标识符的规定 第一个字符必须是大小写英文字母或者下划线，不能是数字
- 中缀表达式（对应中序遍历，左中右）
- 栈的应用 进制转换、迷宫求解、递归

4 队列

- 只允许在表的一端插入，另一端删除
- 队列操作 rear 进，front 出（rear 中不存放元素）
  - 初始条件 Q.front = Q.rear = 0
  - 进队操作 SqQueue[Q.rear] = x; Q.rear ++;
  - 出队操作 x = SqQueue[Q.front]; Q.front ++;
- 循环队列操作
  - 初始条件 Q.front = Q.rear = 0
  - 进队操作 SqQueue[Q.rear] = x; Q.rear = (Q.rear+1)%MaxSize
  - 出队操作 x = SqQueue[Q.front]; Q.front = (Q.front+1)%MaxSize
  - 队列长度 (Q.rear+MaxSize-Q.front)%MaxSize
  - 队空条件 Q.front = Q.rear
- 队满条件
  - 牺牲一个单元（rear所在的单元不放元素） (Q.rear+1)%MaxSize = Q.front
  - 增设表示元素个数的数据成员 Q.size = MaxSize
  - 增设 tag 表示是否队满
- 能由输入受限的双端队列得到而不能由输出受限的双端队列得到的是 4,1,3,2
- 能由输出受限的双端队列得到而不能由输入受限的双端队列得到的是 4,2,1,3
- 既不能由输出受限的双端队列得到也不能由输入受限的双端队列得到的是 4,2,3,1
- 易错点：用链表形式存储队列时，一般情况下，删除操作只需要修改头指针即可；但如果队列中只有一个元素，删除后队列为空，需要修改尾指针Q.rear=Q.front
- 队列的应用 缓存区

5 树与二叉树

- 结点的度 一个结点的子结点的个数
- 树的度 一棵树中结点的最大度数
- 结点的层次 从根结点开始为第一层
- 结点的高度 从根结点开始自低向上增加
- 结点的深度 从根结点开始自上向下增加
- 树的高度（深度） 从1开始
- 路径长度 路径上经过的边的个数
- 树的路径长度 所有路径长度的总和
- 易错点 度为2的树至少有三个结点；而二叉树可以为空；二叉树区分左右子树
- 二叉树的性质 $N_0 + N_1 + N_2 = 2N_2 + N_1 + 1 => N_0 = N_2 + 1$
- 具有n个结点的 **完全** 二叉树的高度为 $log_2 N + 1$
- 二叉树中不可能存在奇数个的度为1的结点

- 由二叉树的中序和先序、后序、层次遍历可以唯一的确定一颗二叉树
- 线索二叉树是加上线索的链表结构，他是一种物理结构
- 先序二叉树中查找前驱需要知道结点的双亲；后序二叉树中查找后继也需要知道结点的双亲
- 树的存储结构
    - 双亲表示法 静态链表，有一个单元存储双亲的位置
    - 孩子表示法 每个结点一个链表，链表中有结点的孩子结点
    - 孩子兄弟表示法 每个结点包含结点值，指向结点第一个孩子结点的指针，指向结点下一个兄弟结点的指针
- 易错点 树的后根遍历相当于二叉树的中序遍历（左根右）
- 二叉排序树的删除操作
    - 叶结点 直接删除
    - 只有一颗左子树或者右子树，使用右子树或者左子树替代
    - 有左和右子树，找右子树最左边的或者左子树最右边的替代
- 平衡二叉树的插入
    - 每次调整的对象都是最小不平衡子树
    - N_h 表示深度为h的平衡树中含有的最小结点数，
      $N_h = N_{h-1} + N_{h-2} + 1, N_0 = 0, N_1 = 1, N_2 = 2$
    - 含有n个结点的平衡二叉树的最大深度是 $log_2 n$
    - 平均查找长度 $log_2 n$
- 非前缀编码 任何一个编码都不是其他编码的前缀

6 图

- 表可以是空表，树可以是空树，但图不可以是空图（不能没有一个顶点）
- 简单图 1.不存在重复边 2.不存在到自身的边
- 无向完全图 无向图任意两个顶点之间都有边 n个顶点的无向完全图中有 $\frac{n(n-1)}{2}$
- 有向完全图 任意两个顶点之间都有方向相反的弧 $n(n-1)$
- 子图 子图U的所有顶点和边都在原图V的顶点集和边集中（任意有向图的边集和顶点集的子集有可能构不成原有向图的子集）
- 连通子图 子图是连通的
- 连通分量 极大连通子图
- 极小连通子图 既使图连通，又使边数最小
- 强连通图 有向图中任意两个顶点可以互相到达
- 强连通分量 有向图中的极大强连通子图
- 连通图的生成树 包含图中所有顶点的极小连通子图（不连通没有生成树）
- 简单路径 顶点不重复出现的路径
- 简单回路 只有顶点和终点成环
- 有向树 一个顶点入度为0，其他顶点入度均为1

- 邻接矩阵 有向图和无向图（唯一） 可以压缩，适合稠密图
- 邻接表法 顶点表 边表 顶点表后跟这个顶点相关的顶点 有向图和无向图（不唯一）
- 十字链表
  - 弧结点 尾域tailvex（弧尾在图中的位置） 头域headvex（弧头在图中的位置） hlink（弧头相同的下一个弧结点） tlink（弧尾相同的下一个弧结点） info（弧带的信息）
  - 顶点结点 以该顶点为弧头的一个弧结点link 以该顶点为弧尾的一个弧结点link 数据
- 邻接多重表
  - 边结点 ivex，jvex（边的两个顶点） ilink，jlink（依附于i，j的下一条边）
  - 顶点结点 顶点信息 一条依附于该顶点的边 （不唯一）
- 以邻接矩阵为存储结构，时间复杂度$O(n^2)$；以邻接表为存储结构，时间复杂度$O(V+E)$
- 最小生成树
  - 当权值互不相等时，最小生成树是唯一的
  - Prim算法 从一个点出发，每次找出已确定点集中的顶点的最短路径
  - Kruskal算法 找图中距离最小的点，如果构成环则舍去
- 最短路径
  - Dijkstra 从顶点开始，每一趟遍历各个顶点，记录最短路径，选出最短的点加入到集合S中；之后的每一次遍历都要绕过新加的点，如果新的路径长那么原路径不变
  - Floyd dist[-1] 各点到其余点的路径数组；dist[i] 各点绕i点到其余点的路径，小的保留；直到遍历所有顶点
- 有向无环图 DAG
- AOV网 DAG图中顶点表示活动，活动有先后顺序
- 拓扑排序 深度遍历倒过来就是拓扑排序结果；如果邻接矩阵是三角矩阵，则存在拓扑序列，反之不成立
- AOE网 DAG图中顶点表示事件，边表示活动
- 缩短关键路径可以加快活动；但不能随意缩短，可能会使关键路径不再是关键路径
- 关键路径不唯一，找多条关键路径的共同关键活动缩短才可以缩短事件

7 查找

- 静态查找表 不做插入、删除操作的表
-

查找方法	ASL(S)	ASL(U)	说明	时间复杂度
一般表的线性查找	$\frac{n+1}{2}$	$n+1$	顺序存储和链式存储都可以；链表的查找只能使用这个	O(n)
有序表的顺序查找	$\frac{n+1}{2}$	$\frac{n}{2}+\frac{n}{n+1}$	顺序存储和链式存储都可以；不成功时最后有两个元素查找为n	O(n)

查找方法	ASL(S)	ASL(U)	说明	时间复杂度
折半查找	n 个结点	n+1 个结点	适合线性表的顺序存储，不适合链式存储；有序，随机存取	O($log_2 n$)
分块查找	X	X	块内元素可以无序，块间元素必须有序	O($L_1 + L_2$)

- 易错点 折半查找的判定树平衡二叉树，不一定是完全二叉树
- B 树
  - m阶B树 每个结点最多只能有m棵子树，m-1个s原始；或空树
  - **除根结点以外** 其他结点最少有 $\lceil m/2 \rceil$ 棵子树
  - 根结点至少有两棵子树
  - 所有叶结点都出现在同一层次上，并且不带信息；可以看作是查找失败点，这些点实际不存在，指向这些结点的指针是空指针
  - $log_m(n+1) \leq h \leq log_{\lceil m/2 \rceil}((n+1)/2) + 1$
  - 插入的结点一定在最底层的某个非叶结点
  - 删除 直接删除 -> 兄弟够借 -> 兄弟不够借
- B+ 树
  - m阶B+树 每个结点最多只能有m棵子树 m个元素
  - **除根结点以外** 其他结点至少有两棵子树
  - 所有非叶结点只起到索引所用，只含有对应子树的最大关键字和指向子树的指针，不含有关键字对应记录的存储地址
  - 可以进行两种运算：从根结点的多路查找；从最小关键字开始的顺序查找
  - B+ 适用于操作系统的文件索引和数据库索引
- Hash 表
  - 散列函数 直接定址法 除留余数法 数字分析法 平方取中法 折叠法
  - 处理冲突的方法 开放定址法（既向它的同义词开放，也向非同义词开放）（线性探测法 平方探测法 再散列法 伪随机序列法） 拉链法（链接法）
  - 散列性能取决于散列函数、处理冲突的方法和装填因子（\(\\frac{表中记录数}{散列表长度}\)）
  - 平均查找长度依赖于装填因子，而不直接依赖表中记录数和散列表长度
- 字符串模式匹配 KMP算法
  - 思想：母串只遍历一次 根据子串自身元素重复的特性
  - next 数组的计算方法：编号前面元素的匹配个数，匹配就+1，不匹配-1

8 排序

-

算法种类	Best时间	Avg时间	Bad时间	空间复杂度	是否稳定
直接插入排序	O(n)	O(n^2)	O(n^2)	O(1)	是
冒泡排序	O(n)	O(n^2)	O(n^2)	O(1)	是

算法种类	Best时间	Avg时间	Bad时间	空间复杂度	是否稳定
简单选择排序	O(n^2)	O(n^2)	O(n^2)	O(1)	否
希尔排序				O(1)	否
快速排序	O($nlog_2n$)	O($nlog_2n$)	O(n^2)	O(log_2n)	否
堆排序	O($nlog_2n$)	O($nlog_2n$)	O($nlog_2n$)	O(1)	否
2-路归并	O($nlog_2n$)	O($nlog_2n$)	O($nlog_2n$)	O(n)	是
基数排序	O(d(n+r))	O(d(n+r))	O(d(n+r))	O(r)	是

- 冒泡排序最好情况：一趟比较（n-1），flag没有变，就认为是有序
- 算法是否稳定不是衡量一个算法优劣的条件
- 对于任意序列基于比较的排序，求最少的比较次数，应考虑在最坏的情况下。对于任意n个关键字的比较次数至少为$\lceil log_2(n!) \rceil$
- 快速排序的运行时间和划分是否对称有关
- 堆排序 元素之间比较的次数与序列的初始状态无关，始终是$n(n-1)/2$
- 基数排序不是基于比较的排序算法
- 外部排序
  - 时间代价主要取决于访问磁盘的次数
  - 外排序的总时间=内部排序所需的时间+外存信息读写的时间+内部归并的时间
- 败者树（内部归并） 大的为失败者，小的为胜利者，根结点指向最小数 败者树的高度为log_nm
- 置换-选择排序（生成初始归并段） 关键步骤：取工作区比开始的最小值大的记录中的最小记录；选不出来时输入归并结束的标志
- 最佳归并树（设计m路归并排序的优化方案） 当$(n-1)$，n 结点个数，m 度，那么正好可以构造m叉归并树
- 为实现输入/内部归并/输出的并行处理，需要设置2m个输入缓存区和2个输出缓冲区

9 伪代码（C）

9.1 代码常识

- 指针 存储数据所在位置，地址
- 引用 数据

9.2 线性表

- 线性表中的元素是从1开始的，而数组中的顺序是从0开始的
- 线性表的顺序存储类型表述（数组定义）（静态分配）

```
#define MaxSize 50
typedef struct {
  ElemType data[MaxSize];
  int length;
} SqList;
```

- 线性表的顺序存储类型表述（数组定义）（动态分配）

```
#define InitSize 20
typedef struct {
  // 指针
  ElemType *data;
  int MaxSize, length;
} SeqList;
// 动态分配
L.data = (ElemType) malloc(sizeof(ElemType) *InitSize);
// C++ 动态分配
L.data = new ElemType[InitSize]
```

- 顺序表的插入

```
bool ListInsert(SqList &L, int i, ElemType e) {
  // 将元素e插入到L的第i个位置；i从1开始
  if (i < 1 || i > L.length + 1)
    return false;
  if (L.length >= MaxSize)
    return false;
  // 后移
  for (int j=L.length; j>=i; j--) {
    L.data[j] = L.data[j-1];
  }
  L.data[i-1] = e;
  L.length ++;
  return true;
}
```

- 顺序表的删除

```
bool ListDelete(SqList &L, int i, ElemType &e) {
  // 将L中第i个元素删除，赋值给e
  if (i < 1 || i > L.length)
    return false;
```

```
  e = L.data[i-1];
  // 前移
  for (int j=i; j<L.length; j++)
    L.data[j-1] = L.data[j];
  L.length --;
  return false;
}
```

9.3 单链表

- 结点定义

```
typedef struct LNode {
  ElemType data;
  struct LNode *next;
} LNode, *LinkedList;
```

- 头插法

```
LinkedList createList1(LinkList &L) {
  // 带头结点的单链表
  LNode *s; int x;
  L = (LinkedList)malloc(sizeof(LNode));
  L -> next = null;
  scanf("%d", &x);
  while (x != 9999) {
    s = (LNode*)malloc(sizeof(LNode));
    s -> data = x;
    s -> next = L -> next;
    L -> next = s;
    scanf("%d", &x);
  }
  return L;
}
```

- 尾插法

```
LinkedList createList2(LinkedList &L) {
  int x;
  // malloc 分配出来的是指针类型
  L = (LinkedList)malloc(sizeof(LNode));
  LNode *s, *r = L;
  scanf("%s", x);
```

```
  while(x != 9999) {
    s = (LNode*)malloc(sizeof(LNode));
    s -> data = x;
    r -> next = s;
    r = s;
    scanf("%d", x);
  }
  r -> next = NULL;
  return r;
}
```

- 双链表

```
typedef struct DNode {
  ElemType data;
  Struct DNode *prior, *next;
} DNode, *DLinkList;
```

- 静态链表

```
#define MaxSize 50
typedef struct {
  ElemType data;
  int next;
} SLinkList[MaxSize];
```

9.4 栈

- 顺序栈（顺序存储）

```
#define MaxSize 50
typedef struct {
  ElemType data;
  int top;
} SqStack;
```

9.5 队列

- 顺序队列

```
#define MaxSize 50
typedef struct {
```

```
  ElemType data;
  int front, rear;
} SqQueue;
```

9.6 二叉树

- 先序遍历

```
void PreOrder(BiTree T) {
  if (T != NULL) {
    visit(T);
    PreOrder(T -> lchild);
    PreOrder(T -> rchild);
  }
}
```

- 中序遍历

```
void InOrder(BiTree T) {
  if (T != NULL) {
    inOrder(T -> lchild);
    visit(T);
    inOrder(T -> rchild);
  }
}
```

- 后序遍历

```
void PostOrder(BiTree T) {
  if (T != NULL) {
    PostOrder(T -> lchild);
    PostOrder(T -> rchild);
    visit(T);
  }
}
```

- 中序遍历的非递归

```
void InOrder2(BiTree T) {
  // 需要借助栈
  InitStack(S);
  BiTree p = T;
```

```
  while (p || !IsEmpty(S)) {
    if (p) {
      Push(S, p);
      p = p -> lchild;
    } else {
      Pop(S, p);
      visit(p);
      p = p -> rchild;
    }
  }
}
```

- 先序遍历的非递归

```
void PreOrder2(BiTree T) {
  InitStack(S);
  BiTree p = T;
  while (p || IsEmpty(S)) {
    if (p) {
      visit(p);
      Push(p);
      p = p -> lchild;
    } else {
      Pop(p);
      p = p -> rchild;
    }
  }
}
```

9.7 顺序查找

```
typedef struct {
  ElemType *elem;
  int TableInt;
} SSTable;

int Search_Seq(SSTable ST, ElemType key) {
  // 哨兵存放查找值
  ST.elem[0] = key;
  for (i=ST.TableLen; ST.elem[i]!=key; --i)
    // 从后往前找，找到或者到ST[0] 退出
    return i;
}
```

9.8 折半查找

```
int Binary_Search(SeqList L, ElemType key) {
  int low = 0, high = L.TableLen-1, mid;
  while (low < high) {
    mid = (low + high) / 2;
    if (L.elem[mid] == key)
      return mid;
    else if (L.elem[mid] > key)
      high = mid - 1;
    else
      low = mid + 1;
  }
  return -1;
}
```

9.9 直接插入排序

```
void InsertSort(ElemType A[], int n) {
  int i, j;
  for (i=2; i<=n; i++) {
    if (A[i].key < A[i-1].key) {
      A[0] = A[i];
      for (j=i-1; A[0].key<A[j].key; --j) {
        // 向后挪位
        A[j+1] = A[j];
      }
      A[j+1] = A[0];
    }
  }
}
```

9.10 折半插入排序

```
void InserSort2(ElemType A[], int n) {
  int i, j, low, high, mid;
  for (i=2; i<=n; i++) {
    A[0] = A[i];
    low = 1; high = i-1;
    while (low <= high) {
      mid = (low + high) / 2;
      if (A[mid].key > A[0].key)
        high = mid-1;
```

```
      else if (A[mid].key < A[0].key)
        high = mid + 1;
    }
    for (j=i-1; j>=high+1; --j) {
      A[j+1] = A[j];
    }
    A[high+1] = A[0];
  }
}
```

9.11 希尔排序

```
void ShellSort(ElemType A[], int n) {
  for (dk=n/2; dk>=1; dk=dk/2) {
    for (i=dk+1; i<=n; ++i) {
      if (A[i].key < A[i-dk].key) {
        A[0] = A[i];
        for (j=i-dk; j>0 && A[0].key<A[j].key; j-=dk) {
          A[j+dk] = A[j];
        }
        A[j+dl] = A[0];
      }
    }
  }
}
```

9.12 冒泡排序

```
void BubbleSort(ElemType A[], int n) {
  for (i=0; i<n-1; i++) {
    flag = false;
    for (j=n-1; j>i; i--) {
      if (A[j-1].key > A[j].key) {
      // 可以先记住这个位置，最后找到最小的元素再交换
        swap(A[j-1], A[j]);
        flag = true;
      }
      if (flag == false) {
        return;
      }
    }
  }
}
```

9.13 快速排序

```
void QuickSort(ElemType A[], int low, int high) {
  if (low < high) {
    int pivotpos = Partition(A, low, high);
    QuickSort(A, low, pivotpos-1);
    QuickSort(A, pivotpos+1, high);
  }
}

int Partition(ElemType A[], int low, int high) {
  ElemType pivot = A[low];
  while (low < high) {
    while (low<high && A[high]>=pivot) --high;
    A[low] = A[high];
    while (low<high && A[low]<=pivot) ++low;
    A[high] = A[low];
  }
  A[low] = pivot;
  return low;
}
```

9.14 选择排序

```
void SelectSort(ElemType A[], int n) {
  for (i=0; i<n-1; i++) {
    min = i;
    for (j=i+1; j<n; j++) {
      if (A[j] < A[min]) min=j;
    }
    if (min != i)
      swap(A[i], A[min]);
  }
}
```

9.15 堆排序

```
// 建立大根堆
void BuildMaxHeap(ElemType A[], int len) {
  for (int i=len/2; i>0; i--) {
    AdjustDown(A, i, len);
  }
}
```

```
void AdjustDown(ElemType A[], int k, int len) {
  // 自上向下
  A[0] = A[k];
  for (int i=2*k; i<len; i*=2) {
    if (i<len && A[i]<A[i+1])
      i ++;
    if (A[0] > A[i])
      break;
    else {
      A[k] = A[i];
      k = i;
    }
  }
  A[k] = A[0];
}

void AdjustUp(ElemType A[], int k) {
  // 自下向上
  A[0] = A[k];
  int i = k/2;
  while (i<0 && A[i]<A[0]) {
    A[k] = A[i];
    k = i;
    i = k / 2;
  }
  A[k] = A[0];
}

void HeapSort(ElemType A[], int len) {
  BuildMaxHeap(A, len);
  for (int i=len; i>1; i--) {
    // 取最大值
    Swap(A[i], A[l]);
    // 将 n-1 个元素取最大值
    AdjustDown(A, 1, i-1)
  }
}
```

Date: 2018-11-10

Author: hiro

Created: 2018-12-11 二 17:33