Data Structure Final Review

RandomStar in 2020.01

目录

Chapter 1 Introduction and Algorithm Analysis

- 算法的几个要素：输入，输出，明确性 (Definiteness)，有穷性 (Finiteness)，高效性 (Effectiveness)

1.1 时间复杂度和空间复杂度

- 几个基本的运算规则
  - 顺序结构：直接相加
  - 循环中：复杂度 = 一次循环的复杂度 x 循环次数
  - 嵌套循环中：循环规模的乘积 x 一次循环的复杂度
  - if/else 语句：选其中复杂度最高的

1.2 最大子列和问题的分析

- 算法 1：使用 3 层嵌套循环直接计算，复杂度为 $O(N^3)$
- 算法 2：使用两层循环并设置标记位，复杂度为 $O(N^2)$
- 算法 3：使用归并的方法 (这一部分 PPT 上的代码比较复杂，可以好好看看)，复杂度为 $O(N\lg N)$
- 算法 4：一种在线算法，复杂度时线性的，代码如下

```
int MaxSubsequenceSum(int A[],int N) {
    int ThisSum = 0,MaxSum = 0,j;
    for(j = 0; j < N; j++) {
        ThisSum += A[j];
        if(ThisSum > MaxSum)
            MaxSum = ThisSum;
              else if(ThisSum < 0)
```

```
            ThisSum = 0;
    }
    return MaxSum;
}
```

Chapter 2 Linked List, Stacks and Queues

2.1 List 的抽象数据结构 (ADT)

- 包含如下操作：
  - 获得长度
  - 打印列表
  - 清空
  - 查询一个元素
  - 插入删除
  - 找到下一个
  - 找最值

2.1.1 Array List

- 需要估计好数组的最大长度，找第 K 个元素的时间复杂度是**常数级别**的，而插入和删除的时间复杂度是 $O(N)$

2.1.2 Linked List

- 插入和删除消耗**常数时间**，而查询第 K 个的时间复杂度是 $O(N)$
- □ 链表的冒泡排序

```
List BubbleSort(List L)
{
    if(L->Next == NULL || L->Next->Next == NULL)
        return L;
    List p = L;
```

```
        while(p->Next->Next != NULL) {
            if(p->Next->key > p->Next->Next->Key) {
                List q = p->Next;
                p->Next = q->Next;
                q->Next = p->Next->Next;
                p->Next->Next = q;
            }
        }
    }
```

2.2 栈 Stack

- 是一个 LIFO 的列表 (Last-in-First-out)
- 支持这样一些操作
  - Push 将一个元素添加到栈的末尾
  - Pop 弹出栈的末尾元素

2.3 队列 Queue

- 是一种 FIFO 的列表 (First-in-First-out)
- 支持如下操作
  - Enqueue 将元素添加到队列的末尾
  - Dequeue 将位于队列最前面的元素弹出

chapter 3 Trees

3.1 定义

- 树是一系列结点构成的 (也可以为空) 并且包含
  - 一个根节点 r
  - 若干和 r 相连的子树 (subtree)
  - 一个 N 个节点的树一定有 N-1 条边

- 几个树中的基本概念
  - 父节点，子节点，同辈节点，叶节点
  - 节点的度数：节点子树的个数，空节点的度数为 0
  - 树的度数：节点度数的最大值
  - 祖先 ancestors：所有拥有通往这个节点路径的节点
  - 后代 decendants：这个节点子树中的所有节点，不包含自己
  - 深度 depth 从根节点到当前节点的路径长度，根节点的深度为 0
  - 高度 height 从叶节点到当前节点路径的最大长度，叶节点的高度为 0，空节点的高度为-1

3.2 二叉树 Binary Tree

- 每一个节点最多有 2 个子节点的树叫做二叉树
- 二叉树的遍历：
  - 前序遍历：按照上左右的顺序递归地遍历
  - 中序遍历：按照左上右的顺序递归地遍历
  - 后序遍历：按照左右上的顺序递归地遍历

```
void PreOrder(Tree T)
{
        if(T == NULL) return;
        printf("%d ",T->Value);
    PreOrder(T->Left);            先访问根节点
    PreOrder(T->Right);
}

void InOrder(Tree T)
{
    if(T == NULL) return;
```

```
    InOrder(T->Left);
    printf("%d ",T->Value);
    InOrder(T->Right);
}

void PostOrder(Tree T)
{
    if(T == NULL) return;
    PostOrder(T->Left);
    PostOrder(T->Right);
    printf("%d ",T->Value);
}
```

先访问子节点

- 二叉树的性质：
  - 第 i 层上最多可以用 2^{i-1} 个节点
  - 深度为 K 的二叉树最多拥有 $2^k - 1$ 个节点

3.3 二叉搜索树 BST

- 定义：
  - 左子树的键值都不超过根节点
  - 右子树的键值都大于根节点
  - 左右子树都是二叉搜索树
  - 只有一个节点的树和空树是二叉搜索树
- 二叉搜索树的几个基本操作
  - 查找一个键值：递归地进行查询，比要查的小查右边，比要查的大查左边
  - 找到最小/最大的键值：直接找最左边的或者最右边的节点
  - 插入：先查找键值，找到合适的位置在进行插入
  - 以上三个操作的时间复杂度都是 $O(h)$ 而 h 是树的高度，最好情况下 $h = O(\log N)$

Chapter 4: Heaps(Priority Queues)

- 二叉堆
  - 实现方式：一棵用数组表示的完全二叉树
    - 完全二叉树的特点：1-H-1 层的节点是满的，第 H 层的节点从左边开始依次放置没有空的，其中 H 是整棵树的高度
    - 高度为 H 的完全二叉树有 $[2^H, 2^{H+1}-1]$ 个节点
  - 二叉堆的性质：
    - 对于下标为 K 的节点，其父节点的下标是 K/2，其子节点的下标是 2K 和 2K+1
    - 分为最小堆和最大堆两种
      - 最小堆，所有的父节点中的值都要小于子节点
      - 最大堆，所有的父节点中的值要大于子节点
  - 堆的几种操作：
    - 插入：向下调整到合适的位置

```
void Insert(PriorityHeap H, int X) {
    int i;
    if(IsFull(H) == 1) return;
    else {
        H->size++;
        for(i = H->size; H->Elements[i/2] > X; i=i/2)
            H->Elements[i] = H->Elements[i/2];
               H->Element[i]=X;
    }
}
```

    - 删除最小值 (最小堆)：直接删除根节点，向上调整

```
int DeleteMin(PriorityHeap H)
{
    int i, child, Min, Last;
```

```
        if (IsEmpty(H) == 1)
            return H->Element[0];
        else
        {
            Min = H->Element[1];
            Last = H->Element[H->size--];
            for (i = 1; 2 * i <= H->size; i = child)
            {
                child = i * 2;
                if (child != H->size && H->Element[child + 1] < H->Element[child])
                    child++;
                if (Last > H->Element[child])
                    H->Elements[i] = H->Elements[child];
                else
                    break;
            }
            H->Elements[i] = Last;
            return Min;
        }
    }
```

- 一个常见的应用：找数组中第 K 小的元素
  - 将数组插入一个最小堆中，不断 deletemin，K 次之后的删除的值就是数组中第 K 小的元素

Chapter 5: Disjoint Set

- 等价类的定义：一个定义在集合 S 上的关系是一个等价关系当且仅当它具有兑成性，自反性和传递性
- 并查集的操作
  - Union 操作：普通的 union，根据 size/height 进行 union
    - 负数表示这个节点是根节点，并且负数的绝对值表示其元素个数

        * 正数表示当前下标的数据的根节点的编号

  – Find 操作

```
void Union(DisjSet S,int root1,int root 2)
{
    S[root1]=root2;
}

int Find(DisjSet S,int X)
{
    while(S[X] > 0)
        X = S[X];
    return X;
}
```

  – 路径压缩：每次合并直接连接到根上面，避免路径过长

```
int Find(DisjSet S, int X)
{
    if(S[X] <= 0) return X;
    else return S[X] = Find(S,S[X]);
}
//Another Method
int Find(DisjSet S, int  X)
{
    int root,train,lead;
    for(root = X, S[root] > 0;X = S[root]);
    for(trail = X; trail != root;trail = lead)
    {
        lead = S[trail];
        S[trail] = root;
    }
    return root;
}
```

    - 根据大小来合并：将小的合并到大的上面去

```
void Union(DisjSet S,int root1,int root 2)
{
    if(S[root1] <= S[root2]) //root1 is larger
    {
        S[root1] += S[root2];
        S[root2] = root1; //insert root2 to root 1
    } else {
        S[root2] += S[root1];
        S[root1] = root2;
    }
}
```

Chpater 6: Graph

6.1 图的基本概念

- 有向图/无向图：区别在于边是否有方向
- 完全图：图中的所有节点两两相连，对于 N 个点有 N(N+1)/2 条边
- 子图 G：顶点和边都是图 G 的子集
- 路径、路径的长度
    - 路径分为简单路径和环两种
- 连通分量：图 G 的一个最大连通子图
- 强连通有向图：任意两个顶点之间存在有向的路径可以到达
- 顶点 V 的度数
    - 有向图中分为出度和入度
    - 总而言之表示这个顶点所在的边的数量，其中有向图还区分出去的边和进入的边

6.2 拓扑排序

- 定义：

  - 拓扑逻辑顺序是顶点的一种线性排列，如果存在顶点 i 指向顶点 j 的边，那么拓扑排序中 i 一定出现在 j 的前面
  - 只有有向无环图才有拓扑排序，并且可能不唯一
- 实现拓扑排序的算法

```
#define INF 123456789
int TopNum[Max];
void TopSort(Graph G)
{
    int Q[Max], rear, front, counter;
    rear = 0;
    front = 0;
    counter = 0;
    int v, w, i, j;
    //Find the head vertices
    for (v = 0; v < G->Nv; v++)
    {
        if (Indegree[v] == 0)
            Q[rear++] = v;
    }
    while (rear - front != 0)
    {
        v = Q[front++];
        TopNum[v] = ++counter;
        for (w = 0; w < G->nv; w++)
        {
            if (G[v][w] != INF)
            {
                Indegree[w]--;
                if (Indegree[w] == 0)
                    Q[rear++] = w;
            }
        }
```

```
    }
    if (counter != G->Nv)
        return; // The graph has a circle
}
```

6.3 最短路径算法 Dijkstra Algorithm

- 基本的思路：
  - 在未访问的顶点中，寻找一个和目标距离最短的顶点 V
  - 如果没有找到，就停止，如果找到了，将 V 标记位已访问
  - 对所有和 V 相邻的节点 W，更新最多路径距离的值
- 代码实现

```
void Dijkstra(MGraph Graph, int dist[], Vertex S)
{
    //count[MAX] means the number of shortest paths,count[S]=1;
    int visit[MAX] = {0}, i, j;
    int n = Graph->Nv;
    for (i = 0; i < n; i++)
        dist[i] = INF;
    dist[S] = 0;
    for (;;)
    {
        int u = -1, v, min = INF;
        for (i = 0; i < n; i++)
        {
            if (dist[i] < min && visit[i] == 0)
            {
                min = dist[i];
                u = i;
            }
        }
        if (u == -1)
```

```
            break;
        visit[u] = 1;
        for (v = 0; v < n; v++)
        {
            if (Graph->G[u][v] < INF && visit[v] == 0)
            {
                if (dist[v] > dist[u] + Graph->G[u][v])
                    dist[v] = dist[u] + Graph->G[u][v];
                //count[v]=count[u];
                //path[v]=u;
                //if(dist[v]==dist[u]+Graph->G[u][v])
                //count[v]+=count[u];
            }
        }
    }
    for (i = 0; i < n; i++)
        if (dist[i] == INF)
            dist[i] = -1;
}
```

- 算法的时间复杂度是 $O(|V|^2 + |E|)$

6.4 网络流 Network Flow

- 目标：在图 G 中找到从 s 出发到 t 的最大流，步骤如下
  - 在 G 中找到一条从 s 到 t 的路径
  - 将这条路径的最短边长从每一条边中减去，并将这个数值加入结果中
  - 更新图 G 并删除长度为 0 的边
  - 重复上述步骤直到不存在 s 到 t 的路径

```
int minlen=INF;
int maxflow(int s,int e,int n)
{
```

```
    int i,result=0;
    while(1)
    {
        if(search(s,e,n)==0)
            return result;
        for(i=e;i!=s;i=pre[i])
            if(G[pre[i]][i]<minlen)
                minlen=G[pre[i]][i];
        for(i=e;i!=s;i=pre[i])
        {
            G[pre[i]][i]-=minlen;
            G[i][pre[i]]+=minlen;
        }
        result+=minlen;
    }
    return result;
}

int search(int s,int e,int n)
{
    int v,i;
    rear=0;
    front=0;
    memset(visit,0,sizeof(visit));
    q[rear]=s;
    rear++;
    visit[s]=1;
    while(rear-front!=0)
    {
        v=q[front];
        front++;
        for(i=0;i<n;i++)
        {
            if(visit[i]==0&&G[v][i]!=0)
```

```
                {
                    q[rear]=i;
                    rear++;
                    visit[i]=1;
                    pre[i]=v;
                    if(i==e){
                        return 1;
                    }
                }
            }
        }
        return 0;
    }
```

- 算法的时间复杂度是 $O(|E|^2 \log |V|)$

6.5 最小生成树和 DFS

- DFS 的基本模式：从一个顶点 V 开始，遍历所有和 V 相邻并且未访问的顶点，需要递归地进行

```
void DFS(Vertex v)
{
    visit[V]=1;
    int w;
    for(w=0;w<n;w++) {
        if(visit[w] == 0 && G[v][u] < INF)
            DFS(w);
    }
}
```

  - 一个基本的应用：找连通分量

```
void ListComponents(Graph G)
{
    for each v in G
```

```c
        if(visit[v]==0) {
            DFS(v);
            printf("\n");
        }
    }
```

- Prim 算法和 Kruskal 算法都是贪心算法
  - 具体的算法看 PPT 就可以了，一种是 DFS 的算法，一种是 BFS 的算法

6.6 一些历年卷里难以判断的题目

- 图论中一些难以判断的结论（都是对的）
  - If `e` is the only shortest edge in the weighted graph G, then `e` must be in the minimum spanning tree of G.
  - If the BFS sequence of a graph is `1 2 3 4 ...`, and if there is an edge between vertices 1 and 4, then there must be an edge between the vertices 1 and 3.
  - In a directed graph G with at least two vertices, if DFS from any vertex can visit every other vertices, then the topological order must NOT exist.
  - Suppose that a graph is represented by an adjacency matrix. If there exist non-zero entries in the matrix, yet all the entries below the diagonal are zeros, then this graph must be a directed graph.
  - 欧拉回路/欧拉路径：遍历图 G 中的每一条路径
    - 无向图存在欧拉回路，当且仅当该图的所有顶点度数都为偶数且连通
    - 有向图存在欧拉回路，当且仅当所有的出度等于入度且图要连通
  - 哈密顿路径/哈密顿回路：恰好通过图 G 的每个节点一次
- Kruskal's algorithm is to grow the minimum spanning tree by adding one edge, and thus an associated vertex, to the tree in each stage. （**FALSE**）
- 关于拓扑逻辑排序
  - If a graph has a topological sequence, then its adjacency matrix must be triangular.

    * 错的，在无向图中不一定

  - If *Vi* precedes *Vj* in a topological sequence, then there must be a path from *Vi* to *Vj*.

    * 错的，不一定有

  - If the adjacency matrix is triangular, then the corresponding directed graph must have a unique topological sequence.

    * 错的，可以举出反例

  - In a DAG, if for any pair of distinct vertices *Vi* and *Vj*, there is a path either from *Vi* to *V**j* or from *Vj* to *Vi*, then the DAG must have a unique topological sequence.

    * 对的

Chapter 7: Sort

7.1 插入排序

- 最好的情况是 $O(N)$ 最坏的情况是 $O(N^2)$
  - N 个元素中的平均 inversion 个数为 $I = \frac{N(N+1)}{4}$ 并且时间复杂度为 $O(I + N)$

```c
void InsertionSort(int a[], int n)
{
    int j, p, tmp;
    for (p = 1; p < n; p++) {
        tmp = a[p];
        for (j = p; j > 0 && a[j - 1] > tmp; j--) {
            a[j] = a[j - 1];
        }
        a[j] = tmp;
    }
}
```

7.2 希尔排序

- 定义一系列间隔，每次按照间隔进行排序，并且每一轮的间隔不断减小，直到变成 1

```
void ShellSort(int a[], int n)
{
    int i, j, increment, tmp;
    for (increment = n / 2; increment > 0; increment /= 2)
    {
        for (i = increment; i < n; i++)
        {
            tmp = a[i];
            for (j = i; j >= increment; j -= increment)
            {
                if (tmp < a[j - increment])
                {
                    a[j] = a[j - increment];
                }
                else
                    break;
            }
            a[j] = tmp;
        }
    }
}
```

- 最差的时间复杂度依然是平方级别的

7.3 堆排序

- 用建堆 +delete max 操作来进行排序，时间复杂度为 $O(N \lg N)$

```
#define leftchild(i) (2 * (i) + 1)
//different from traditonal heap,a[] start from index 0;
void PercDown(int a[], int i, int n)
{
    int child, tmp;
    for (tmp = a[i]; leftchild(i) < n; i = child)
    {
```

```c
        child = leftchild(i);
        if (child != n - 1 && a[child + 1] > a[child])
            child++;
        if (a[child] > tmp)
            a[i] = a[child];
        else
            break;
    }
    a[i] = tmp;
}

void HeapSort(int a[], int n)
{
    int i;
    for (i = n / 2; i >= 0; i--) //build heap
        PrecDown(a, i, n);
    for (i = n - 1; i > 0; i--) {
        int t = a[0];
        a[0] = a[i];
        a[i] = t;
        PercDown(a, 0, i);
    }
}
```

7.4 归并排序

- 将数组分成两路进行排序然后将两个数组合并成一个，时间复杂度是 $O(N \lg N)$

```c
void MSort(int a[], int tmp[], int left, int right)
{
    int center = (left + right) / 2;
    if (left < right)
    {
        MSort(a, tmp, left, center);
        Msort(a, tmp, center + 1, right);
```

```
        Merge(a, tmp, left, center + 1, right);
    }
}

void MergeSort(int a, int n)
{
    int tmp[Max];
    Msort(a, tmp, 0, n - 1);
    //need O(n) extra space
}

void Merge(int a[], int tmp[], int lpos, int rpos, int rightend)
{
    int i, leftend, num, tmppos;
    leftend = rpos - 1;
    tmppos = lpos;
    num = rightend - lpos + 1;
    while (lpos <= leftend && rpos <= rightend)
    {
        if (a[lpos] <= a[rpos])
            tmp[tmppos++] = a[lpos++];
        else
            tmp[tmppos++] = a[rpos++];
    }
    while (lpos <= leftend)
        tmp[tmppos++] = a[lpos++];
    while (rpos <= rightend)
        tmp[tmppos++] = a[rpos++];
    for (i = 0; i < num; i++, rightend--)
        a[rightend] = tmp[rightend];
}
```

- 一种迭代式的实现方式

```
void merge_pass(ElementType list[], ElementType sorted[], int N, int length);
```

```c
void merge_sort(ElementType list[], int N)
{
    ElementType extra[MAXN]; /* the extra space required */
    int length = 1;          /* current length of sublist being merged */
    while (length < N)
    {
        merge_pass(list, extra, N, length); /* merge list into extra */
        output(extra, N);
        length *= 2;
        merge_pass(extra, list, N, length); /* merge extra back to list */
        output(list, N);
        length *= 2;
    }
}

void merge_pass(ElementType list[], ElementType sorted[], int N, int length)
{
    int i, j;
    for (i = 0; i < N; i += 2 * length)
    {
        int x, y, z;
        x = i;
        y = i + length;
        z = x;
        while (x < i + length && y < i + 2 * length && x < n && y < n)
        {
            if (list[x] < list[y])
                sorted[z++] = list[x++];
            else
                sorted[z++] = list[y++];
        }
        if (x < i + length)
            while (x < i + length)
```

```
                sorted[z++] = list[x++];
        if (y < i + 2 * length)
            while (y < i + 2 * length)
                sorted[z++] = list[y++];
    }
}
```

7.5 快速排序：已知的最快排序方式

- 需要选择一个 pivot，每次将比 pivot 效地放到左边，大的放到右边
  - 最坏的复杂度依然时平方复杂度，但是最优的复杂度是 $O(N \lg N)$，并且平均复杂度是 $O(N \lg N)$
  - 一个结论：基于比较的排序方法的时间复杂度至少是 $O(N \lg N)$ 级别的

```
int median3(int a[], int left, int right)
{
    int center = (left + right) / 2;
    if (a[left] > center)
        swap(a[left], a[center]);
    if (a[left] > a[right])
        swap(a[left], a[right]);
    if (a[center] > a[right])
        swap(a[right], a[center]);
    //these steps makes a[left]<=a[center]<=a[right]
    swap(a[center], a[right - 1]) //hide pivot
        //only need to sort a[left+1]~ a[right-2]
        return a[right - 1];
}

void Qsort(int a[], int left, int right)
{
    int i, j, pivot;
    pivot = median3(a, left, right);
    i = left, j = right - 1;
```

```
    while (1) {
        while (a[++i] < pivot) {
        }
        while (a[--j] > pivot) {
        }
        if (i < j)
            swap(a[i], a[j]);
        else
            break;
    }
    swap(a[i], a[right - 1]);
    Qsort(a, left, i - 1);
    Qsort(a, i + 1, right);
}

void QuickSort(int a[], int n) {
    Qsort(a, 0, n - 1);
}
```

7.6 桶排序，基数排序

- 桶排序：时间换空间的经典方法
- 基数排序：用 Least Significant Digit first(LSD) 的方法进行排序
  - 具体的可以看 PPT 怎么进行操作

7.7 总结：

排序算法的稳定性

- 不稳定的排序：堆排序，快速排序，希尔排序，直接选择排序
- 稳定的排序：基数排序，冒泡排序，插入排序，归并排序

数组排序呈现的特征

- 堆排序：一般没什么特征

- 归并排序：排序中会出现连续若干个数字已经排好了顺序
- 快速排序：有一些数，前面的都比这个数小，后面的都比他大，具体要看 run 了多少次
- 选择排序：每一次都会选出最大或者最小的数排在最后应该出现的位置

Chapter 8: Hash

8.1 定义和基本概念

- 哈希函数 f(x) 的结果表示 x 在哈希表中的位置
- T 表示不同的 x 的个数
- n 表示哈希表中的位置个数
- b 表示哈希表中 bucket 的个数
- s 表示槽的个数
- identifi er density = n / T
- loading density $\lambda = n/sb$
- collision 冲突：两个值被放到了同一个 bucket 中
- overflow 溢出：bucket 超过了承载的上限

8.2 解决 collision

- 需要找到另一个空的地方来放置待放入的元素
- 常用的方法：
  - 线性探查，平方探查，看 PPT 和做题就可以理解两种方法是怎么操作的