线性表、栈、队列、数组、树、二叉树以及图，排序、查找

一．introduce

data structure: logical structure, storage structure, operation

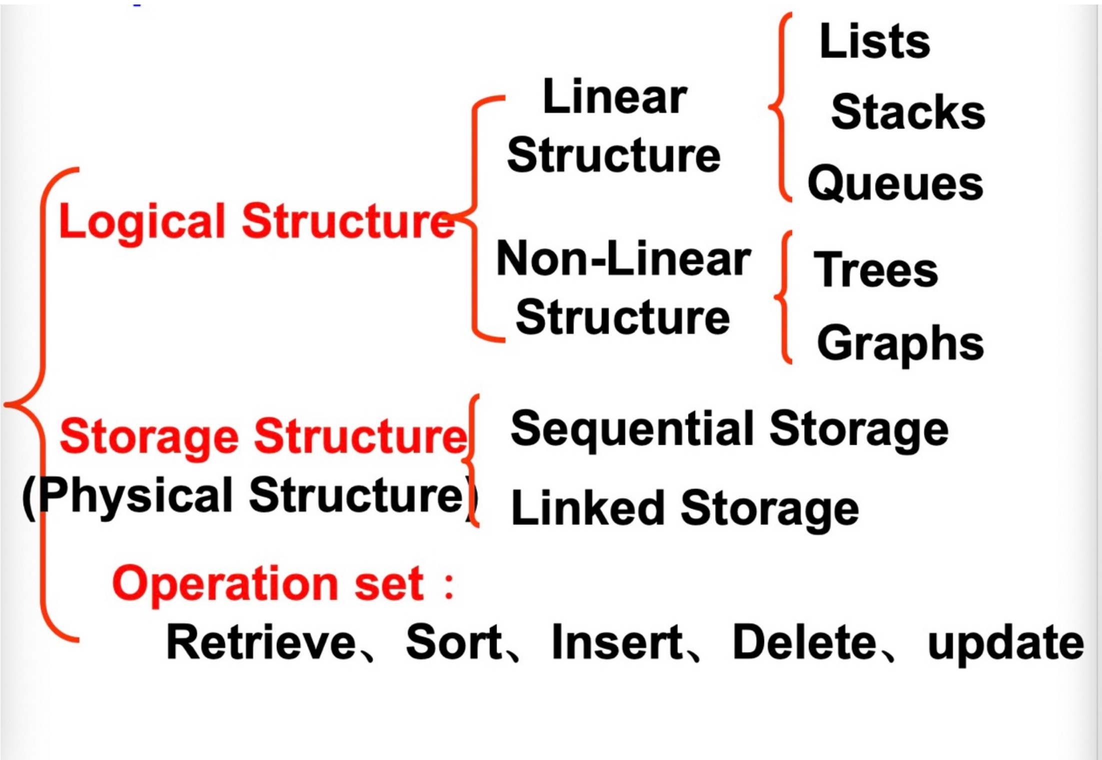
four types of logical data structure:

linear structure

trees

graphs

set



Abstract data type: object and operation

Algorithm: time and space

二．Linear List

1. sequential representation: estimate max size, find time O(1), insert and delete time O(N)
2. linked representation:

    1) singly linked list: create, length O(n), find O(n), search O(n), delete O(n), insert O(n), erase O(n)

    2) circular linked list: 约瑟夫 circle

    3) doubly linked circular list: insert O(n), delete O(n)
3. indirectly addressing representation(间接寻址表示):
4. simulated pointer representation(静态链表)

三．stack and queue

stack：

1) Base = 0, top = 0,top 指向当前顶部元素的下一位

2) Base = -1, top = -1，top 指向当前元素

queue:

1) The condition of queue is full or empty:

    a) Empty: s.front = s.rear

    b) Full: (s.rear+1)%MAXSIZE = s.front

四.Array

array representation:

1) Row-major mapping

2) Colum-major mapping

We can use array to represent Matrix, matrix starts from 1 rather than 0

Compress storage of Matrix:

Low triangular matrix

Upper triangular matrix

Symmetric matrix

Diagonal matrix

Tri-diagonal matrix

Sparse matrix：

1） sequence of triple(array representation)

2） linked representation

Orthogonal matrix(十字链表)

了解广义表的概念，有两种元素：原子和列表，广义表的操作：head and tail，广义表用链表表示，两种表示方法，各有利弊。

五.Tree and binary tree

Tree: n nodes and n-1 edges

Binary tree: no more than 2 children every node

n0=n2+1 n0 是叶子结点的数量，n2 是 degree 是 2 的结点

证明：

n=n0+n1+n2 B+1=n B=n1+2n2

so n1+2n2+1=no+n1+n2 n0=n2+1

full BT:

complete BT:结点 i 的父节点为 1 或者 i/2，左结点为 2i，右结点为 2i+1

有 n 个结点的完全二叉树的高度 h：log2(n+1)<=h<=h

binary tree 的各种性质：

storage structure of BT:

1. sequential representation

   if it not complete, waste memory

2. linked representation

   1) binary linked list

      left child   data   right child

   2) trifurcate linked list

      parent lchild data rchild

3) parent linked list

data parent LRTag

traversal of binary tree

lRr: in-order:

```
void In-order (BiTree T)
 {
     if (T) {
          In-order(T->lchild);
          printf( "%d" ,T->data) ;
          In-order(T->rchild);
     }
 }
```

lrR: post-order

```
void Post-order (BiTree T)
 {
     if (T) {
          Post-order(T->lchild);
          Post-order(T->rchild);
          printf( "%d" ,T->data) ;
     }
```

```
    }
```

Rlr: pre-order

```
void Pre-order (BiTree T)
{
    if (T) {
        printf( “%d” ,T->data) ;
        Pre-order(T->lchild);
        Pre-order(T->rchild);
    }
}
```

above are recursive algorithm, it is easy but not fast.

Non-recursive algorithm:

In-order: 从根节点开始，把左节点都入栈，pop，将节点指向 pop 出的节点的右节点进行遍历。直到节点为 null and 栈为空停止。

Pre-order: 入栈根节点，while 循环，p=pop print，push p 的 rchild， push p 的 lchild。

Post-order:

1. p 如果是叶子节点，直接输出。
2. p 如果有孩子，且孩子没有被访问过，则按照右孩子，左孩子的顺序依次入栈。
3. p 如果有孩子，而且孩子都已经访问过，则访问 p 节点。

我们可以保存最后一个访问的节点 last，如果满足 (p->right==NULL && last ==p->left) || last=p->right，那么显然 p 的孩子都访问过了。

Threaded binary tree:

If left child is null, point to previous node, else left child.

If right child is null, point to successor, else right child

Tree and forest:

Tree representation:

1) parent representation

   data parent, use a array to store nodes, root’ s parent is -1

2) child linked representation

   data firstchild

   or with parent: data parent firstchild

3) first child next sibling

   can use this to transfer a tree to a binary tree

   transfer between forest and BT

Tree traversal and forest traversal

Preorder, post order and level order

1. Obtain the height of a tree
2. Obtain the number of leaf nodes
3. Output all paths from root to leaves.
4. Construct the storage structure of tree

Huffman Tree & Huffman codes：

Optimal tree: minimum WPL(weighted path length)

掌握根据权重构建哈夫曼树的过程：

哈夫曼树的节点只有度为 0 或 2

叶子结点为 n 的树的总结点为 2n-1

六.Graph

representation:

vertex and edge

undirected graph and directed graph

1) adjacency matrix

use 2D matrix to represent

2) adjacency linked lists

adjvex info nextrc

traversal:

depth first search：

```
void DFS(Graph G, int v) //using Adjacent List
{ // starting from v, traverse G using DFS
visited[v] = TRUE;      Visit(v);
     w=G-> AdjList[v].firstarc
    while(w!=0)
```

```
        { if (!visited[w->adjvex])
                    DFS(G, w ->adjvex);
            w= w->nextarc;
        }
} // DFS
```

breadth first search

1) Visit start vertex and put it into a FIFO queue.
2) Repeatedly remove a vertex from the queue, visit its unvisited adjacent vertices, put newly visited vertices into the queue.

Application of traversal:

1) Search a path from I to s

   Use DFS
2) Search the shortest path from I to s

   Use BFS

Minimum cost spanning tree

1. Prim algorithm (O(n^2))

    1） 初始化

```
/***初始化lowcost数组，closest数组(即从起点开始设置lowcost数组，closest数组相应的值，以便后续生成使用)***/
for (i = 0; i < g.n; i++)//赋初值，即将closest数组都赋为第一个节点v，lowcost数组赋为第一个节点v到各节点的权重
{
        closest[i] = v;
        lowcost[i] = g.edges[v][i];//g.edges[v][i]的值指的是节点v到i节点的权重
}
```

    2） 循环找出下一个结点，并更新数组

```
/**********************************开始生成其他的节点*********************************/
for (i = 1; i < g.n; i++)//接下来找剩下的n-1个节点（g.n是图的节点个数）
{

        /*****找到一个节点，该节点到已选节点中的某一个节点的权值是当前最小的*****/
        min = INF;//INF表示正无穷（每查找一个节点，min都会重新更新为INF，以便获取当前最小权重的节点）
        for (j = 0; j < g.n; j++)//遍历所有节点
        {
                if (lowcost[j] != 0 && lowcost[j] < min)//若该节点还未被选且权值小于之前遍历所得到的最小值
                {
                        min = lowcost[j];//更新min的值
                        k = j;//记录当前最小权重的节点的编号
                }
        }

        /****************输出被连接节点与连接节点，以及它们的权值***************/
        printf("边(%d,%d)权为:%d\n", closest[k], k, min);

        /***********更新lowcost数组，closest数组，以便生成下一个节点************/
        lowcost[k] = 0;//表明k节点已被选了(作标记)
        //选中一个节点完成连接之后，作数组相应的调整
        for (j = 0; j < g.n; j++)//遍历所有节点
        {
                /* if语句条件的说明：
                 *  (1) g.edges[k][j] != 0是指k！=j，即跳过自身的节点
                 *  (2) g.edges[k][j]是指刚被选的节点k到节点j的权重，lowcost[j]是指之前遍历的所有节点与j节点的
                 *  (3) 有人会问：为什么只跳过掉自身的节点（即k==j），而不跳过所有的已选节点？当然我们可以在if语句
                 */
                if (g.edges[k][j] != 0 && g.edges[k][j] < lowcost[j])
                {
                        //更新lowcost数组，closest数组
                        lowcost[j] = g.edges[k][j];//更新权重，使其当前最小
                        closest[j] = k;//进入到该if语句里，说明刚选的节点k与当前节点j有更小的权重，则closest[
                }
        }
}
```

2. Kruskal algorithm（O（eloge)）

依次找出最小的边，判断是否形成了环，没有环则加入，否则继续

```
void Kruskal ( Graph G )
{     T = { } ;
      while   ( T contains less than |V| – 1 edges
                              && E is not empty ) {
            choose a least cost edge (v, w) from E ;
            delete (v, w) from E ;
            if   ( (v, w) does not create a cycle in T )
    add (v, w) to T ;
            else
    discard (v, w) ;
      }
      if   ( T contains fewer than |V| – 1 edges )
            Error (   "No spanning tree"   ) ;
}
```

Shortest path algorithm

1）Shortest Path from source vertex to every other vertex

使用 Dijkstra:

Dist[k] = <源点到顶点 k 的弧上的权值>

或者 = <源点到其它顶点的路径长度>

+ <其它顶点到顶点 k 的弧上的权值>

2）All-Pairs Shortest Path Problem

弗洛伊德算法：

```
/* A[ ] contains the adjacency matrix with A[ i ][ i ] = 0 */
/* D[ ] contains the values of the shortest path */
/* N is the number of vertices */
/* A negative cycle exists iff D[ i ][ i ] < 0 */
void AllPairs( TwoDimArray A, TwoDimArray D, int N )
{    int   i, j, k;
     for ( i = 0; i < N; i++ )   /* Initialize D */
          for( j = 0; j < N; j++ )
    D[ i ][ j ] = A[ i ][ j ];
     for( k = 0; k < N; k++ )   /* add one vertex k into the
path */
          for( i = 0; i < N; i++ )
    for( j = 0; j < N; j++ )
       if( D[ i ][ k ] + D[ k ][ j ] < D[ i ][ j ] )
```

```
        /* Update shortest path */
          D[ i ][ j ] = D[ i ][ k ] + D[ k ][ j ];
}
```

Topological Sort

critical path

七．Searching（7 大查找办法）

1. sequential search(顺序查找)

from left to right
from right to left, set a sentinel at 0 position
查找成功时的平均查找长度为： ASL = 1/n(1+2+3+…+n) = (n+1)/2 ;
当查找不成功时，需要 n+1 次比较，时间复杂度为 O(n);

2. binary search（二分查找）

有序查找（二分，用于有序数组）
mid=low+1/2*(high-low)
Decision tree
The height of a decision tree with n nodes is log2n+1
最坏情况下，关键词比较次数为 log2(n+1)，且期望时间复杂度为 O(log2n)；

3. 插值查找

mid=low+(key-a[low])/(a[high]-a[low])*(high-low)
思路同二分查找，但是把系数作为自适应的，减少查找次数
查找成功或者失败的时间复杂度均为 O(log2(log2n))

4. 斐波那契查找

5. 树表查找

1) 二叉树(binary search tree)
   插入和查找的时间复杂度均为 O(logn)，但是在最坏的情况下仍然会有 O(n)的时间复杂度
   思路同二分，插入时，先寻找是否存在，存在则 update value，否则创建新的结点，并放在查找位置
   二叉树的增删查找，删除比较麻烦

2) 平衡树
   子树深度差绝对值不大于 1
   平衡因子为 BF
   BF>1 右旋转，BF<-1 左旋转，最小不平衡子树的 BF 与它的子树的 BF 符号相反时，就需要对结点先进行一次旋转以使得符号相同后，再反向旋转一次才能够完成平衡操作
3) b 树
   B 树可以看作是对 2-3 查找树的一种扩展，即他允许每个节点有 M-1 个子节点。
   - 根节点至少有两个子节点
   - 每个节点有 M-1 个 key，并且以升序排列
   - 位于 M-1 和 M key 的子节点的值位于 M-1 和 M key 对应的 Value 之间
   - 其它节点至少有 M/2 个子节点
4) b+树
   B+树是对 B 树的一种变形树，它与 B 树的差异在于：
   - 有 k 个子结点的结点必然有 k 个关键码；
   - 非叶结点仅具有索引作用，跟记录有关的信息均存放在叶结点中。
   - 树的所有叶结点构成一个有序链表，可以按照关键码排序的次序遍历全部记录。

   * 需要注意 b+树的插入，中间插入与顶点处插入，顶点处破坏平衡，该插入的点要加入协助平衡

6. 分块查找

将 n 个数据元素"按块有序"划分为 m 块（m ≤ n）。每一块中的结点不必有序，但块与块之间必须"按块有序"；即第 1 块中任一元素的关键字都必须小于第 2 块中任一元素的关键字；而第 2 块中任一元素又都必须小于第 3 块中的任一元素，……

7. 哈希查找

处理哈希相同导致的冲突问题:
线性表示：
1）冲突之后++1
2）冲突之后先加 1^2，然后 2^2，然后 3^2

链式：a[i] mod n 的值相同的放在一个链上

八．Sorting

1. Insertion Sort

从前到后，一个一个的排序

(1) If A is sorted: O(n) comparisons.

(2) If A is reverse sorted: O(n2) comparisons.

(3) If A is randomly permuted: O(n2) comparisons.

2. Binary Insertion Sort

在插入前，先使用二分查找找到合适的位置，再移动位置后的数组，最后插入

```
low = 1;   high = i-1;
while (low<=high)
{  m = (low+high)/2;
   if (L.r[0].key < L.r[m].key)
         high = m-1;   // in the left segment
   else  low = m+1; // in the right segment
}
```

查找位置的代码，最后的 high 为比当前数字小的中最大的数字

3. shell sort

每次设置不同的 increment，然后对不同的 increment 中的进行排序

```
void ShellInsert ( SqList &L, int dk )
{ for ( i=dk+1; i<=n; ++i )
     if ( L.r[i].key< L.r[i-dk].key)
     {L.r[0] = L.r[i];           // sentinel
       for (j=i-dk;  j>0&&(L.r[0].key<L.r[j].key);
         j-=dk)  L.r[j+dk] = L.r[j];  // move to the right
       L.r[j+dk] = L.r[0];       // insert
     } // if
} // ShellInsert
```

4. bubble sorting

```
void BubbleSort(Elem R[ ], int n) {
  i = n;
  while (i >1) {
    lastExchangeIndex = 1;
    for (j = 1;  j < i;  j++)
      if (R[j+1].key < R[j].key) {
        Swap(R[j], R[j+1]);
        lastExchangeIndex = j; //记下进行交换的记录位置
      } //if
    i = lastExchangeIndex; // 本趟进行过交换的
  } // while             // 最后一个记录的位置
} // BubbleSort
```

5. quick sort

Time complexity of quick sort is O(nlogn)

```
#快速排序 传入列表、开始位置和结束位置
def quick_sort( li , start , end ):
    # 如果start和end碰头了，说明要我排的这个子数列就剩下一个数了，就不用排序了
    if not start < end :
        return

    mid = li[start] #拿出第一个数当作基准数mid
    low = start   #low来标记左侧从基准数始找比mid大的数的位置
    high = end  #high来标记右侧end向左找比mid小的数的位置

    # 我们要进行循环，只要low和high没有碰头就一直进行,当low和high相等说明碰头了
    while low < high :
        #从high开始向左，找到第一个比mid小或者等于mid的数，标记位置，(如果high的数比mid大，我们就左移high)
        # 并且我们要确定找到之前，如果low和high碰头了，也不找了
        while low < high and li[high] > mid :
            high -= 1
        #跳出while后，high所在的下标就是找到的右侧比mid小的数
        #把找到的数放到左侧的空位 low 标记了这个空位
        li[low] = li[high]
        # 从low开始向右，找到第一个比mid大的数，标记位置，(如果low的数小于等于mid，我们就右移low)
        # 并且我们要确定找到之前，如果low和high碰头了，也不找了
        while low < high and li[low] <= mid :
            low += 1
        #跳出while循环后low所在的下标就是左侧比mid大的数所在位置
        # 我们把找到的数放在右侧空位上，high标记了这个空位
        li[high] = li[low]
        #以上我们完成了一次 从右侧找到一个小数移到左侧，从左侧找到一个大数移动到右侧
    #当这个while跳出来之后相当于low和high碰头了，我们把mid所在位置放在这个空位
    li[low] = mid
    #这个时候mid左侧看的数都比mid小，mid右侧的数都比mid大

    #然后我们对mid左侧所有数进行上述的排序
    quick_sort( li , start, low-1 )
    #我们mid右侧所有数进行上述排序
    quick_sort( li , low +1 , end )

```

6. selection sort

1) simple selection sort:
   每次选出最大或者最小的放在下一个位置
   time complexity is $O(n^2)$
2) tree selection sort:
   由于含 n 个叶子结点的完全二叉树的深度为⌊log2n⌋+1 次，则在树形择排序中，除了最小关键字之外，每选择一个次小关键字仅需进行⌊log2n⌋次比较，因此，树形选择排序的时间复杂度为 O(nlog2n)。

7.heap sort

包含大顶堆和小顶堆

8.merge sort

T (N) = O(N log N)

```
// O(n(logn))
void Merge_Sort(float data[], int left, int right, float sorted_data[])
{
    if(left < right)
    {
        int mid = (left + right) / 2;
        Merge_Sort(data, left, mid, sorted_data);
        Merge_Sort(data, mid+1, right, sorted_data);
        Merge_Array(data, left, mid, right, sorted_data);
    }
}

void Merge_Array(float data[], int left, int mid, int right, float temp[])
{
    int i = left, j = mid + 1;
    int k = 0;

    // 从子数组的头开始比较
    while(i <= mid && j <= right)
    {
        if (data[i] <= data[j])
        {
            temp[k++] = data[i++];
        }
        else
        {
            temp[k++] = data[j++];
        }
    }

    // 判断哪个子数组还有元素，并拷贝到 temp 后面
    while(i <= mid)
    {
        temp[k++] = data[i++];
    }
    while(j <= right)
    {
        temp[k++] = data[j++];
    }

    // 将 temp 中的数据拷贝到原数组对应位置
    for(i = 0; i < k; i++)
    {
        data[left+i] = temp[i];
    }
}
```