数据结构与算法-期末复习

By 2023级 yzx

一、单项选择题（30 分=15 题*2 分/题）

1. Algorithm Analysis

	AList	LList
insert	O(n)	O(1)
remove	O(n)	O(1)
append	O(1)	O(1)
next	O(1)	O(1)
prev	O(1)	O(n)
currPos	O(1)	O(n)
moveToPos	O(1)	O(n)
moveToEnd	O(1)	O(1)
moveToStart	O(1)	O(1)
getValue	O(1)	O(1)
lengh	O(1)	O(1)

	BST-Insert & Delete (recursive)	BST中序遍历/print	Heap-siftdown & remove
best	O(logn)【CBT】		O(logn)
worst	O(n)【单边树】		O(n)
average	O(d)	O(n)	O(d)

图的DFS O(|V|+|N|)

B树/B+查找：O(logn)

B树的insert和delete：O(logn)

prim算法MST：O(n+mlogm)

dijkstra算法：O[(n+m)logm]

floyd算法:O(n^3)

DFS-RootCs/DFS-CsRoot：O(n+m)

- Binary search O(logn)

空间开销

邻接矩阵 O(|V|^2)

邻接链表(O(|V|+|N|)) = O(|V|^2)

2. chapter4 List

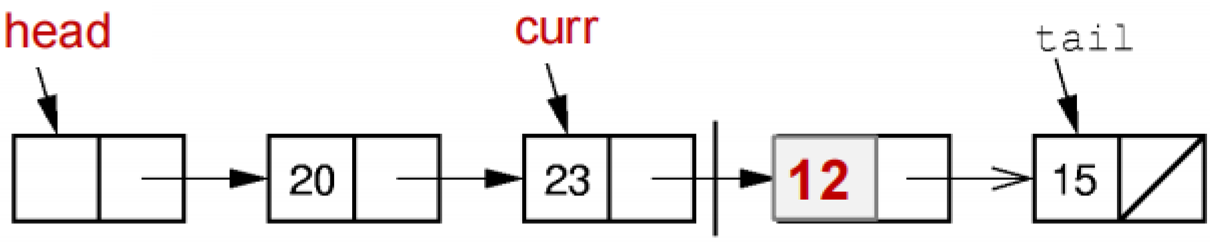
- List ADT会调用各种公共函数

代码块

```
List<int> l1;

l1.insert(16);
l1.remove();
l1.append(20);
x=l1.getValue();
l1.moveToPos(6);
l1.moveToStart();
l1.next();
l1.moveToEnd();
l1.prev();
i=l1.currPos();
l=l1.length();
```

- **Alist**
    - 表满/表空问题，各种操作的功能，*算法及复杂度*
    - private: listArray, maxSize, listSize, curr

- **Llist**
  - 当前指针指向的是当前结点的前一个结点(简化insert remove)，头结点不放数据



  - Private: head, tail, curr, cnt
  - 表空问题
  - *各种操作的功能，算法及复杂度*
  - Free Link：<u>能加快new/delete操作</u>，占用空间多（在Link结点类中添加private: freelist）
  - DoubleLlist：能简化prev操作(On -> O1)，但增加了空间需求

  空间成本

AList	**SC=D*E+3*4**
LList	**SC = (E+P)*N+3P**
E: data space D: maxSize of AList P: data of pointer n: actual size of list	
N = DE/(P+E) **n < N LList** **n > N AList**	

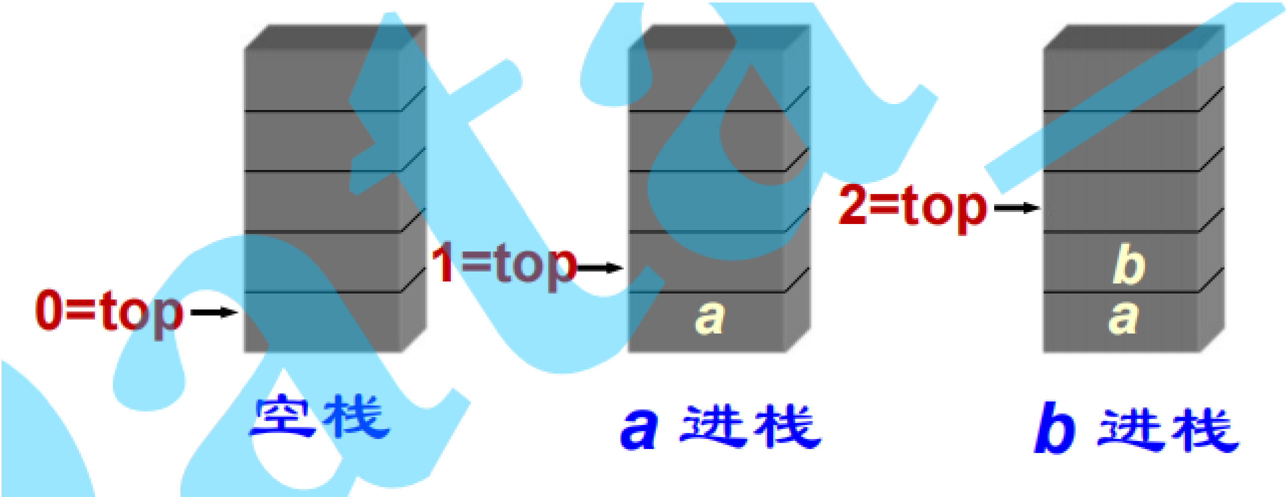
- **Stack**

代码块

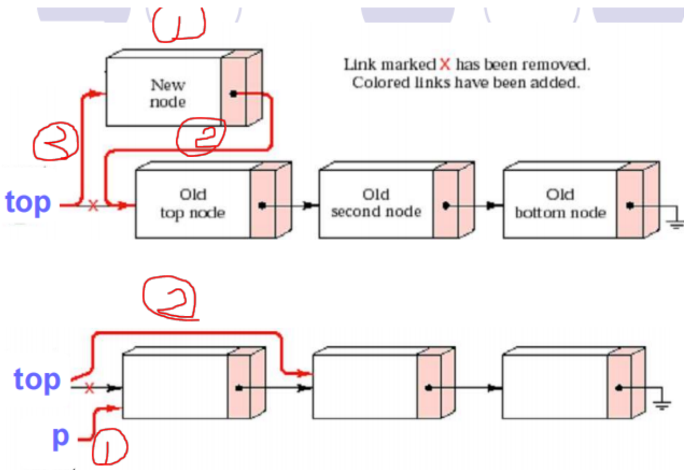
```
AStack<int> s1(100);
LStack<int> s2;

s1.push(a);
b = s1.pop();
x = s1.topValue();
l = s1.lengh();
```

  - LIFO

- AStack各种操作(push，pop)的实现步骤
  - private: listArray, maxSize, top(=0)



  - push步骤：
    - listArray[top]=a
    - top++;
  - Pop
    - top--;
    - return listArrau[top];
- Lstack各种操作(push，pop)的实现步骤



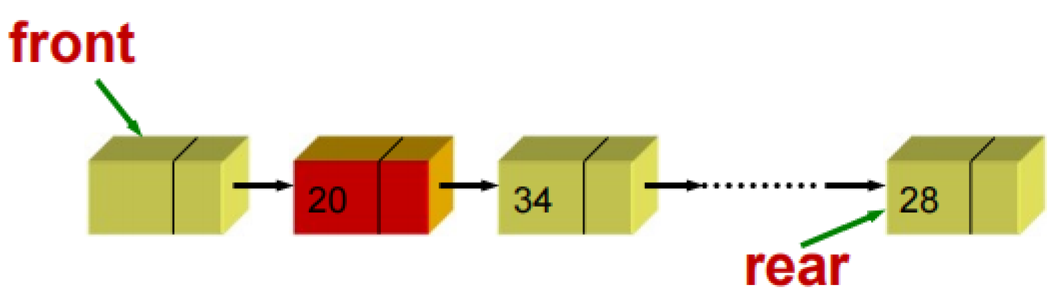
- **Queue**

代码块

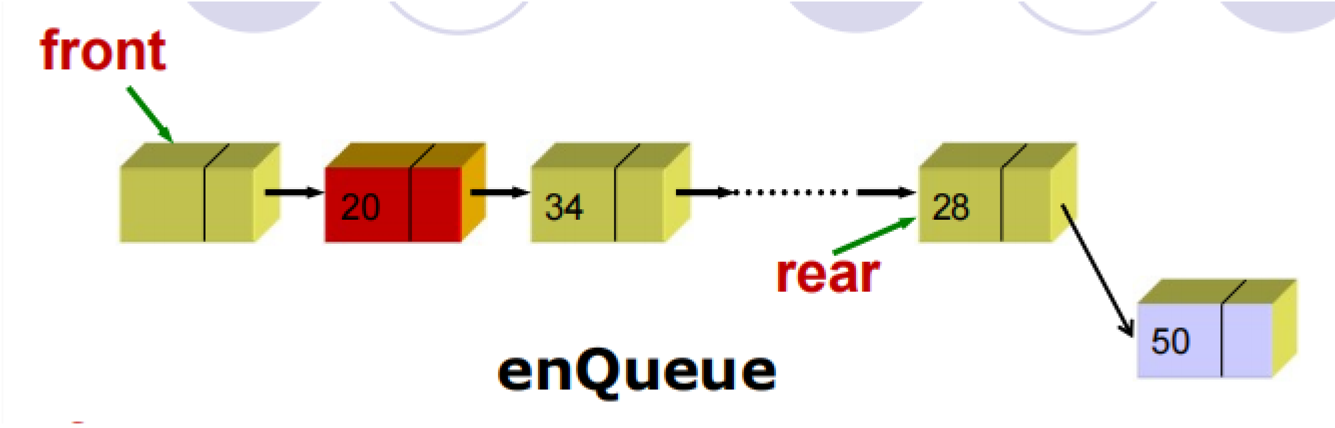
```
CAQueue<int> caq1;
Lqueue<int> lq2;

q1.enQueue(a);
b = q1.deQueue();
```

```
f = q1.frontValue();
l = q1.length();
```

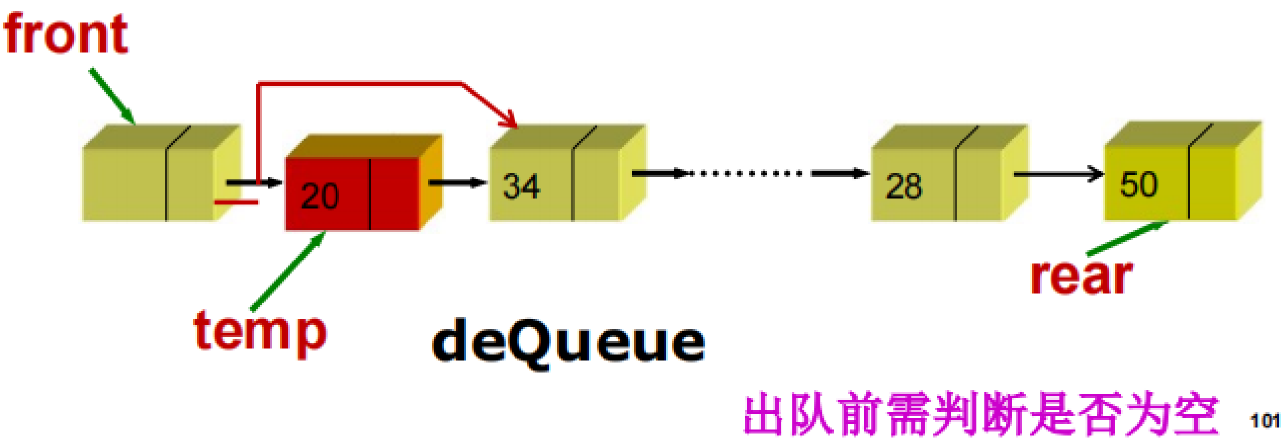
- FIFO
- CAQueue各种操作(EnQueue，DeQueue)的实现步骤
  - Init
    - front = rear = count = 0;或front = rear = 0;
    - 队空：count = 0;或front= rear
    - 队满：count = maxSize 或 (rear+1)%maxSize == front
  - enQueue
    - listArray[rear] = a
    - rear = (rear+1)%maxSize
  - deQueue
    - it = listArray[front]
    - front = (front + 1)% maxSize
    - return it
- Lqueue各种操作(EnQueue，DeQueue)的实现步骤
  - 无队满，有队空
  - private：front，rear，size



  - enQueue
    - rear->next = new Link<E>(it,NULL)
    - rear = rear->next;
    - size++



- E deQueue
  - Link<E> *temp = front->next
  - front->next = temp->next
  - if(rear=temp) rear=front;
  - size--
  - return *temp



3. chapter5 BT

- 术语:Internalnode, leaf, degree,height,depth,BT, FT,CBT
- **BT的遍历**：深度优先DFT(前序，中序，后序)遍历，广度优先遍历BFT(S走向)
- BT相关（没提）

空间利用率	S	Overhead fraction
BSTNode	nE+2nP	(2P)/(2P+E)≈2/3
VarBinNode	n0E+(n-n0)(E+2P)≈ nE+nP	P/(E+P)≈1/2

- 二叉树的性质：
  - 高度为d，至多有2^d-1个结点
  - n0=n2+1
  - n1+2n0=n+1
  - n个结点的CBT d=[log(2)n]+1 (向下取整)
- LList-based for NotCBT
  - 结点类：BSTNode & VarBinNode；
  - 链式二叉树类：BST & HuffmanTree
  - 一个根指针
- AList-based for CBT
  - 左孩子Lc=2n+1；双亲P=[(n-1)/2]
  - 一个数组，两个整型变量(maxSize, size)

5.1 BST的定义，存储结构，构造，插入，删除

- 定义：左子树<root<右子树。中序遍历为排列
- 存储结构：BSTNode<E> root, int nodecount
- 操作：search(e), insert(e),remove(e) size() print()
- 构造：类似于插入
- 插入：找到合适的结点，作为叶子插入
- 删除：
  - 删除叶子
  - 删除只有一个子树的：跨过去
  - 删除两个子树的内部节点：右子树里找最小的替换待删

5.2 Heap(priority queue)的定义，存储结构，构造，插入，删除

左孩子 i = 2n+1；双亲 i = (n-1)/2

存储结构：maxHeap（Array based）

私有的Siftdown

buildHeap()：先按数组写成CBT，从倒二开始siftdown

removeFirst()：移除root放到最后（BT外）

Insert(t)

remove(p)

5.3 Huffman Trees and Huffman coding：定义，构建，编码

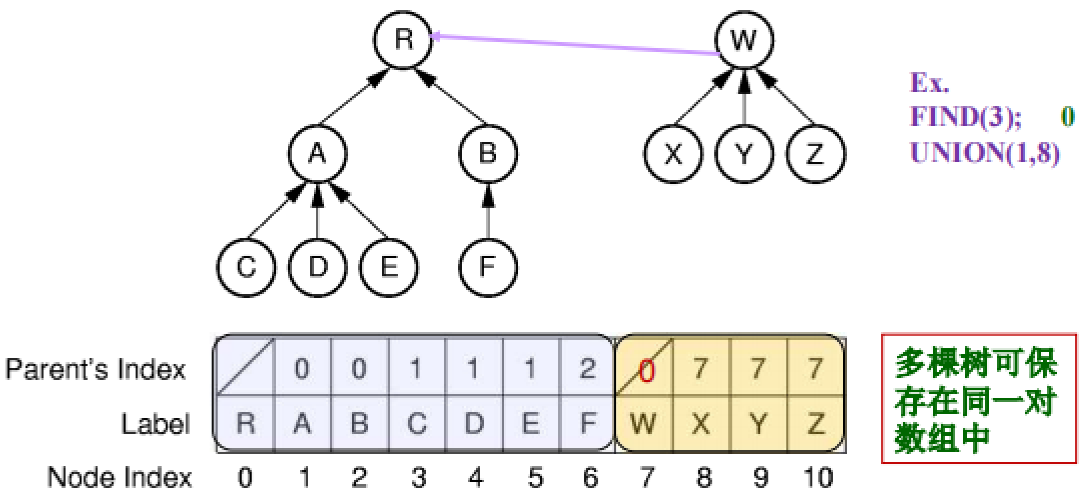
定义：n0哥带权值的叶子BT中必存在一棵**加权路径长度最小的最优树**——哈夫曼树

特点：

- 权值越小离得远
- FBT

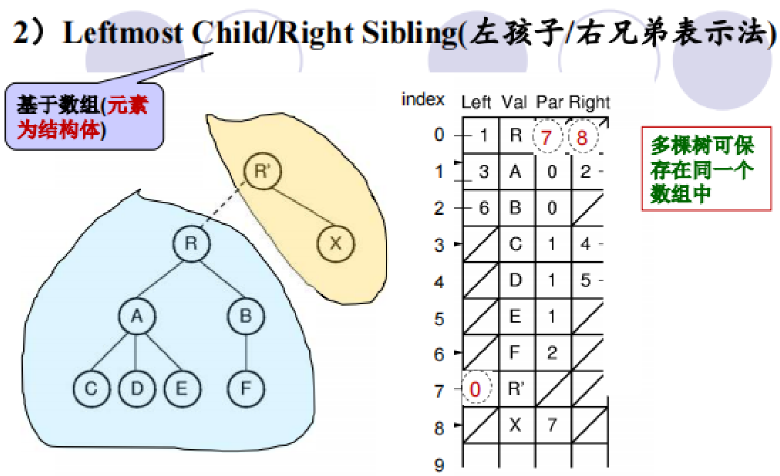
chapter6 NBT

- 通用树的遍历：前根RCs，后根遍历CsR，与对应BT的遍历的关系
- **通用树的表示：**
  - **双亲指针表示——能看懂FIND，UNION操作的实现思路**



UNION(a,b)把b所在树接到a所在树的根上

- **左孩子/右兄弟链接表示法**



- 通用树->二叉树的转换（左孩子右兄弟），森林->二叉树的转换（全部变BT，再右链接）
- 在使用**加权并查集**（weighted union rule）合并不相交集合（disjoint union）时，大小为n的树中任何节点的最大深度是O(logn)

chapter7 内排

	average	best	worst	Stable ?	Space ?	静态?	关键步骤	备注
Bubble Sort	O(n^2)	KCN=O(n^2) RSN=0	KCN=O(n^2) RSN=O(n^2)	Y	N	静态		选最小的冒上来，一次冒多个
Selection Sort	O(n^2)	KCN=O(n^2) RSN=0	KCN=O(n^2)【与初始无关】 RSN=O(n)	N	N	静态		选一个最小的上来
Insertion Sort	O(n^2)	KCN=O(n) RSN=0	KCN=O(n^2) RSN=O(n^2)	Y	N	静态		逐个插到前队里去
Shell Sort	n↑ KCN=RSN=n^1.25			N	N	静态	插入排序	序列分成若干子序列，子序列插入排序 d=[n/2] 或 d=[n/3]+1 向下取整
MergeSort	O(nlogn)			Y	Y (临时数组)		两路归并	(先insertion) 两路归并，直到只有一路
HeapSort	**O(nlogn)** (build O(n), Remove nO(nlogn))	**O(nlogn)**	**O(nlogn)**	N	N		removeFirst	先建堆，重复removeFirst至堆空 （到大建大堆）
Qsort	O(nlogn)		O(n^2)	*N*	N		一次划分 O(n)	

		O(nlogn) 【n很大】	【n很小】				1. 头尾指针lh，p为枢轴，t为枢轴索引。 2. l向右找比p大的，h向左找比p小的，找到后交换 3. 直到lh碰到，交换lh和t的位置的值	1. 对原始数列做一次划分 2. 对子序列递归调用快排（一直分下去直到子序列长度为1） 改进：在序列小的时候插入；选好枢轴（三选一）
BinSort	O(kn)【按位排序】	O(n)【range小】	O(nlog(r)n) 【range0~n】	Y	Y（LList需要n个data+(n+r)个pointer；AList要(n+r)data			

chapter8 外排和文件处理

- track, sector,Cluster, Locality of Reference
- Golden Rule of File Processing, logical/physical file
- Buffer and Buffer pools
- **replacement selection sort**用于**生成初始归并段initial sorted merge files**
- run是A sorted sub-section for a list of records（记录列表的已排序子部分）
- 外排

  - 外部排序的一般步骤:获得初始RUNS+归并排序
  - Simple External Mergesort:
    - 初始runs长度为1，2路归并
    - Replacement Selection的目的，思路:获得尽可能长的初始runs
      - runs平均长度 = 2BM
      - 初始run个数 = N/2BM
    - 最优思路：RS+多路归并
      - 趟数r = Log(B){N/(2BM)}， I/O 次数:(r+1)*2*N/M

      N: 记录总数

      M: 一个block存放的文件数

      B: 给heap可用的buffer/block数量，B路归并

*chapter9 search

- Binary search
- Sequential search
  - Self-organizing lists
    - use a heuristic(启发) for deciding how to reorder the records.
    - Make Records Ordered by search Frequeney
- Hashing:

  在关键字与记录在表中的存储位置之间建立个确定的关系---hashing
  - 构造原则：不能溢出，最好均匀分布hash表里
  - hash functionh(K)
  - Collisions
- Collision Resolution
  - Open Hashing: Separate chaining (linked list)
  - Closed Hashing:增量序列d
    - Hi = (H0 + P(K,i))%m
      - Linear Probing线性探测
      - Quqdratic Probing平方探测
      - Pseudo-eandom Probing
  - Double Hashing: di*h(K)

        - Hi = (H0 + di*h2(K))%m
        - 再插入是等概率
- HT Insert
- SearchinHT
    - 对于给定值K，根据哈希函数及探测函数计算哈希地址
    - 从i=0开始，比较K4HTH，若等于，查找成功，停止
- SL, 平均查找长度ASL

Example : **依次查找 { 19, 01, 23, 14, 55, 68, 11, 82, 36 }**

m=11, H(key) = key % m

$$H_i = (H_0 + d_i)\ \%\ m, \quad i=1, 2, \ldots, m-1, \quad d_i = i$$

	0	1	2	3	4	5	6	7	8	9	10
	55	01	23	14	68	11	82	36	19		
SL	1	1	2	1	3	6	2	5	1		

平均查找长度ASL：(4*1+2*2+3+5+6)/9=2.44

通常ASL ≠ 0，这是因为有冲突存在

*chapter 10 B+

- 2-3Tree的定义，search,insert
- B-tree的定义(2-3Tree推广)
    - 所有结点都存 key/pointerpairs
- B+tree的定义，特点(与B-tree区别
    - 需要两个参数:
        - m(order)决定内部节点的子树个数上限，
        - n决定叶节点中可存放key/record个数上限
    - leaf存 key/pointer,内部结点仅存key
    - 结点半满(除root)
- B+tree的search,insert, delete

	B树	B+树
数据存储位置	所有节点（包括内部节点和叶子节点）都可以存储关键字和数据。	只有叶子节点存储数据，内部节点仅存储关键字，用于索引。
叶子节点连接	没有直接链接	通过指针连接，形成链表，便于范围查询。
搜索效率	访问较少节点（可能在内部节点找到目标数据）	访问更多节点（必须到达叶子节点搜索）
范围查询	不适合范围查询，需要逐层扫描树结构。	叶子节点链表使范围查询效率更高。
插入和删除复杂度	需要调整内部节点的数据和指针，较复杂	插入和删除更简单。只需调整叶子节点的数据和指针，内部节点改动少。
空间利用率	内部节点存储数据，空间利用率更高。	内部节点只存储关键字，可能需要更多的存储空间。
树的高度	可能更矮，因为每个节点能存储的数据更多。	比B树高，因为数据都集中在叶子节点，需要更多节点。

chapter11 graph

- 图的表示：Adjacency Matrix, Adjacency List
- 图的遍历Graph Traversals: DFS, BFS
- Topological SortAlgorithm of DAG- BFS based, DFS-based
- 最短路径Shortest Paths problem
  - Single-Source Shortest Paths, Dijkstra's Algorithm
- Minimal Cost SpanningTrees(MST) problem:
  - Prim's algorithm
  - Kruskal's algorithm

二、应用题（50 分=5 题*10 分/题）

1. 给定代码分析时间复杂度

logn n nlogn n2 n3 2^n n!

2. BT的深度优先遍历；能由两种遍历结果重构出BT;

1. 中序&前序
2. 中序&后序

3. heap的构造/插入/删除; BST的构建/插入/删除;

	Insert	Delete/remove
BST	选择合适的点，插为叶子	• 删叶子 • 删n1：跨越 • 删n2内部节点：找右子树中最大的替换，删叶
Heap(CBT)	插入为叶子，往上爬	和最末叶子替换，删叶，对root siftDown

4. Huffman树的构造/编码

构造：S1:森林 S2:选其中最小的两个构建树，root为权值和，放回森林，重复

编码：01写下去。读表—— Letter Freq Code Bits

平均编码长度=ΣFreq*Bits/ΣFreq

压缩比=不等长的平均编码长度/等长的平均编码长度

5. 内部排序过程和关键操作

给定一序列，能给出某种排序方法的整个排序过程(即每趟后的中间结果)，或者其关键操作(如快排的一次划分)的详细过程;

> 概率最高Qsort

6. *给定哈希函数及冲突处理方法，//双哈希

1. **能将一组数放入HT**
2. **会分析各空巢被下一个数填充的概率**
3. **给定HT后，会计算一个(组)待查询数的SL(ASL)**

7. B+树的构造，插入，删除；2-3树的构造，插入

S1:计算基本信息

Provided (m,n)

	children	key/pointer
root	[2,m]	[1,m-1]
internalNode	[[m/2]↑,m]	[[m/2]↑-1,m-1]
leaf	/	[[n/2]↑,n]

构造&插入：从叶子开建，爆了就分裂升级（叶子序列的第一个）

删除：

- 删后不够了，找兄弟借，调整双亲
- 删后兄弟也不够，合并兄弟，删双亲（双亲还剩）
- 删后兄弟也不够，合并兄弟，删双亲，双亲不剩，再借双亲兄弟的孩子，调整双亲

8. 能画出邻接矩阵 & 邻接List;

相邻的矩阵填权值，LList按顺序指【点编号，权值，指针/N】

9. 若是DAG，能给出基于BFS的Topsort过程及结果;

选没有被指的

10. 能用要求算法获得MST;

Prim：任意选起点，在可选范围内选最短的

Krusal：选最短的边，除非无意义

11. 会用Diikstra's Algorithm求出单源最短路径的距离

要画出Dist[k], Path[k], Mark[k]

Dist[k]

1. 初始是用w[k]填。直接连的是值，间接连/不连=∞
2. i=1，选Dist[k]最小的点v，然后处理和它相连的&没有处理过的点q
    a. 判断Dist[v]+w(v,q) >? Dist[q]
    b. 是的话更新Dist[q] = Dist[v]+w(v,q)
    c. 不是就跳过
3. 直到处理所有点

Path[k]

1. ∞ = -1
2. 更新过的点填更新它的点的索引

Mark[k]

本轮处理的点更新为1

三、编程或算法题（20 分=2 题*10 分/题）

> 给一段描述，不限制自由发挥；
>
> 或者是给一段代码，可以完成一定功能，要么挖空，要么修改代码完成特定功能。

1. 调用相关线性结构(如List，Stack,Queue)的公共函数(基本操作)编写规定的功能函数。

代码块

```
List<int> l1;

l1.insert(16);
l1.remove();
l1.append(20);
x=l1.getValue();
l1.moveToPos(6);
l1.moveToStart();
l1.next();
l1.moveToEnd();
l1.prev();
i=l1.currPos();
l=l1.length();
```

代码块

```
AStack<int> s1(100);
LStack<int> s2;

s1.push(a);
b = s1.pop();
x = s1.topValue();
l = s1.lengh();
```

代码块

```
CAQueue<int> caq1;
Lqueue<int> lq2;

q1.enQueue(a);
b = q1.deQueue();
f = q1.frontValue();
l = q1.length();
```

2. 编写合适的(递归)函数(函数中可调用BSTNode结点类的公用函数)实现一些具体功能

代码块

```
template<class E>
class BSTNode{
public:
    BSTNode()
    BSTNode(E e, BSTNode* l = NULL, BSTNode* r = NULL)

    E& element()  //返回节点值
    void setElement(E& e)  //设置结点值

    BSTNode* left()  //返回左孩子
    void setLeft(BSTNode *b)//设置左孩子
    BSTNode* right()
    void setRight(BSTNode *b)

    bool isLeaf() //是否叶子
}
```

代码块

```
template<class E>
class BST{
public:
    BST()
    BSTNode<E>* find(E& e){}
    void insert(E& e){}
    bool remove(E& e){}
    int size(){}
    void print(){}
}
```

代码块

```
template<class E>
class maxHeap{
public:
    void buildHeap()
    void insert(E& e)
    E removeFirst()
    E remove(int)
}
//案例函数
void main(){
    int i; int a[100],temp,b=5;
    for(int i = 0; i<7; i++) cin>> a[i]

    maxHeap<int> h1(a,7,100);
    h1.print();

    h1.insert(b);
    h1.remove(b);

    while(h1.heapSize()){
         temp = h1.removeFirst()
         cout << temp;
    }
}
```

2.1 寻找BST中的第K大(小)结点值

代码块

```
template<class E>
E findKthSmallest(BSTNode<E>* root, int& k) {
    if (!root) return E(); // 空树返回默认值

    // 递归遍历左子树
    E left = findKthSmallest(root->left(), k);
    if (k == 0) return left; // 找到目标，直接返回

    // 当前结点
    k--; // 访问当前结点
    if (k == 0) return root->element(); // 如果是第K小，返回当前结点值

    // 递归遍历右子树
    return findKthSmallest(root->right(), k);
}
```

```

template<class E>
E findKthLargest(BSTNode<E>* root, int& k) {
    if (!root) return E(); // 空树返回默认值

    // 递归遍历右子树
    E right = findKthLargest(root->right(), k);
    if (k == 0) return right; // 找到目标，直接返回

    // 当前结点
    k--; // 访问当前结点
    if (k == 0) return root->element(); // 如果是第K大，返回当前结点值

    // 递归遍历左子树
    return findKthLargest(root->left(), k);
}

```

2.2 判断两棵BT树(的形状)是否一样

代码块

```
template<class E>
bool isSameShape(BSTNode<E>* root1, BSTNode<E>* root2) {
    // 如果两棵树都为空，则形状相同
    if (!root1 && !root2) return true;

    // 如果只有一棵树为空，则形状不同
    if (!root1 || !root2) return false;

    // 递归比较左子树和右子树的形状
    return isSameShape(root1->left(), root2->left()) &&
           isSameShape(root1->right(), root2->right());
}


void main(){
    bool result = isSameShape(root1, root2);
    if (result) {
        cout << "两棵树形状相同";
    } else {
        cout << "两棵树形状不同";
    }
}


```

2.3 判断BT树中值为K的结点的深度等

代码块

```
template<class E>
int findDepth(BSTNode<E>* root, E& k, int depth = 0) {
    if (!root) return -1; // 空树返回 -1，表示未找到

    if (root->element() == k) return depth; // 找到目标结点，返回深度

    // 递归查找左子树
    int leftDepth = findDepth(root->left(), k, depth + 1);
    if (leftDepth != -1) return leftDepth; // 如果在左子树找到，直接返回

    // 递归查找右子树
    return findDepth(root->right(), k, depth + 1);
}

void main(){
    int depth = findDepth(root, k);
    if (depth != -1) {
        cout << "值为 " << k << " 的结点深度为: " << depth << endl;
    } else {
        cout << "未找到值为 " << k << " 的结点" << endl;
    }
}
```

3. Adjacent matrix / Llist存储结构下图的基本操作;

代码块

```
class Graph{  //ADT
public:

    int n();
    int e();
    int first(int i);  //第一个和它相连的点的索引
    int next(i,j);
    int weight(i,j);
    void setEdge(i,j,w);
    void delEdge(i,j);

    int getMark(int v);
    void setMark(int v, int a);
}
```

Matrix

2D数组保存矩阵，numV, numE

代码块

```
class GraphM : public Graph {
public:
    GraphM(int n){}
    int n();
    int e();
    int first(int i);  //第一个和它相连的点的索引
    int next(i,j);
    void setEdge(i,j,w);
    void delEdge(i,j);
    int weight(i,j);
    int getMark(int v);
    void setMark(int v, int a);
}

int main() {
    GraphM graph(4);
    //GraphL graph(4);
    graph.setEdge(0, 1, 5);
    graph.setEdge(0, 2, 3);
    graph.setEdge(1, 2, 1);
    graph.setEdge(2, 3, 8);

    int w1 = graph.weight(0, 1);
    int e1 = graph.e();
    int n1 = graph.n();

    graph.delEdge(0, 1);

    cout << "1st neighbor: " << graph.first(0) << endl;
    cout << "Next neighbor: " << graph.next(0, 2) << endl;

    return 0;
}
```

4. 拓扑排序算法

代码块

```
void topsort(Graph* G, Queue<int>* Q) {
```

```
    int v, w, *inDegree; // inDegree 用来存放每个顶点的入度
    inDegree = new int[G->n()]; // 动态分配入度数组
    for (v = 0; v < G->n(); v++) inDegree[v] = 0; // 初始化所有顶点的入度为 0

    // 根据边关系设置入度
    for (v = 0; v < G->n(); v++)
        for (w = G->first(v); w < G->n(); w = G->next(v, w))
            inDegree[w]++; // 增加 w 的入度

    // 初始化队列 Q: 将入度为 0 的顶点入队
    for (v = 0; v < G->n(); v++)
        if (inDegree[v] == 0) Q->enqueue(v);

    // 主循环
    while (Q->length() != 0) {
        v = Q->dequeue(); // 从队列中取出顶点
        cout << v << endl; // 处理顶点 v，例如打印其编号

        // 遍历顶点 v 的所有邻接点
        for (w = G->first(v); w < G->n(); w = G->next(v, w)) {
            inDegree[w]--; // 减少 w 的入度 (prereqs 减1)
            if (inDegree[w] == 0) Q->enqueue(w); // 如果 w 的入度为 0，将其入队
        }
    }
}

```

算法描述：

1. 初始化一个入度数组 inDegree[]，长度为图中顶点数，初始值为 0。
2. 遍历图中的每条边 (v -> w)，对每个终点 w 增加其入度 inDegree[w]。
3. 初始化一个空队列 Q，将所有入度为 0 的顶点加入队列。
4. 重复以下步骤直到队列为空：

   a. 从队列中取出一个顶点 v。

   b. 输出或处理顶点 v。

   c. 遍历顶点 v 的所有邻接点 w：

    i. 将 w 的入度减 1。

    ii. 如果 w 的入度变为 0，将 w 加入队列。
5. 如果所有顶点被处理，则输出拓扑排序序列；否则，报告图中有环。

时间复杂度为 `O(V + E)`

使用了额外的入度数组 `inDegree[]` 和一个队列 `Q`，空间复杂度为 `O(V)`。

仅适用于 **有向无环图（DAG）**，即若存在环路，则无法完成拓扑排序。