数据结构大题重点

队列的结构体

```
typedef struct LinkNode{
    ElemType data;
    struct LinkNode *next;
}LinkNode;
typedef struct{
    LinkNode *front,*rear;
}
```

图的结构体

邻接矩阵

```
#define MaxVertexNum 100
typedef struct{
    char Vex[MaxVertexNum];
    int Edge[MaxVertexNum][MaxVertexNum];
    int vexnum,arcnum;
}MGraph;
```

邻接表

```
//用邻接表保存的图
typedef struct{
    AdjList vertices;
    int vernum,arcnum;
}ALGraph;
//顶点
typedef struct VNode{
    VertexType data;
    ArcNode *first;
}VNode,AdjList[MaxVertexNum];
//边/弧
typedef struct ArcNode{
    int adjvex;
    struct ArcNode *next;
}ArcNode;
```

二叉树的遍历

先序

根左右(NLR)

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

第一次路过结点时访问结点

中序

左根右(LNR)

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

第二次路过结点时访问结点

后序

左右根(LRN)

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

第三次路过结点时访问结点

求树的深度

```
int treeDepth(BiTree T){
    if(T==NULL){
        return 0;
    }else{
        int l=treeDepth(T->lchild);
        int r=treeDepth(T->rchild);
        return l>r? l+1:r+1;
    }
}
```

层序遍历

```
void LeveOrder(BiTree T){
    LinkQueue Q;
    InitQueue(Q);
    BiTNode *p;
    EnQueue(Q,T);
    while(!IsEmpty(Q)){
        DeQueue(Q,p);
        visit(p);
        if(p->lchild!=NULL)
            EnQueue(Q,p->lchild);
        if(p->rchild!=NULL)
            EnQueue(Q,p->rchild);
    }
}
```

图的遍历

广度优先遍历（BFS）

```
bool visited[MAX_VERTEX_NUM];   //访问标记数组
void BFSTraverse(Graph G){  //对图G进行广度优先遍历
    for(i=0;i<G.vexnum;i++){
        visited[i]=false;
    }
    InitQueue(Q);
    for(i=0;i<G.vexnum;i++){
        if(!visited[i])
            BFS(G,i);
    }
}
void BFS(Graph G,int v){    //从顶点v出发，广度优先遍历图G
    visit(v);
    visit[v]=true;
    Enqueue(Q,v);
    while(!isEmpty(Q)){
        Dequeue(Q,v);
        for(w=FirstNeighbor(G,v);w=>0;w=NextNeighbor(G,v,w)){
            if(!visit[w]){
                visit(w);
                visit[w]=true;
                Enqueue(Q,w);
            }
        }
```

```
    }
}
```

深度优先遍历（DFS）

```
bool visited[MAX_VERTEX_NUM];
void DFSTraverse(Graph G){
    for(i=0;i<G.vexnum;i++){
        visited[i]=false;
    }
    for(i=0;i<G.vexnum;i++){
        if(!visited[i])
            DFS(G,i);
    }
}
void DFS(Graph G,int v){
    visit(v);
    visited[v]=true;
    for(w=FirstNeighbor(G,v);w=>0;w=NextNeighbor(G,v,w)){
        if(!visited[w])
            DFS(G,w);
    }
}
```

单源最短路径问题

BFS算法(无权图)

```
void BFS_MIN_Distance(Graph G,int u){
    for(i=0;i<G.vertexnum;i++){
        d[i]=∞;
        path[i]=-1;
    }
    d[u]=0;
    //visit(u);
    visited[u]==true;
    Enqueue(Q,u);
    while(!isEmpty(Q)){
        Dequeue(Q,u);
        for(w=FirstNeighbor(G,u);w>=0;w=NextNeighbor(G,u,w))
        if(!visited[w]){
            //visit(w);
            d[w]=d[u]+1;
            path[w]=u;
            visited[w]=true;
            Enqueue(G,w);
        }
    }
}
```

折半查找

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

int Binary_Search(SSTable L,ElemType key){
    int low=0,high=L.Tablelen-1,min;
    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;
}
```

快速排序

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

归并排序

```
int *B=(int *)malloc(n*sizeof(int));

void Merge(int A[],int low,int mid,int high){
    int i,j,k;
    for(k=low;k<=high;k++){
        B[k]=A[k];
    }
    for(i=low,j=mid+1,k=i;i<=mid&&j<=high;k++){
        if(B[i]<=B[j])  A[k]=B[i++];
        else    A[k]=B[j++];
    }
```

```
    while(i<=mid)   A[k++]=A[i++];
    while(j<=high)  A[k++]=A[j++];
}
void MergeSort(int A[],int low,int high){
    if(low<high){
        int mid=(low+high)/2
        MergeSort(A,low,mid);
        MergeSort(A,mid+1,high);
        Merge(A,low,mid,high);
    }
}
```

并查集

并

```
void Union(int S[],int root1,int root2){
    if(root1==root2)    return ;
    S[root1]=root2;
    root2+=root1;
}
void Union(int S[],int root1,int root2){
    if(root1==root2)
        retunr ;
    if(S[root1]>S[root2])
    {
        S[root1]=root2;
        root1+=root2;
    }
    else
    {
        S[root2]=root1;
        root2+=root1;
    }
}
```

查

```
int Find(int S[],int x){
    int root=x;
    while(S[root]>=0)   root=S[root];
    while(x!=root){
        int temp=S[x];
        S[x]=root;
        x=temp;
    }
    return root;
}
```

求图的连通分量个数

```
int ComponentCount(int G[5][5]){
    int S[5];
    for(int i=0;i<5;i++)
        S[i]=-1;
    for(int i=0;i<0;i++)
        for(int j=0;j<0;j++)
            if(G[i][j]>0){
                int root1=Find(S,i);
                int root2=Find(S,j);
                if(root1!=root2)
                    Union(S,root1,root2);
            }
    int count=0;
    for(int i=0;i<5;i++)
        if(S[i]==-1)
            count++;
    return count;
}
```

表格对比

顺序表vs链表

	顺序表	链表
存储结构	优点：支持随机存取、存储密度高 优点：离散的小空间分配方便，改变容量翻遍	缺点：大片连续空间分配不方便，改变容量不方便 缺点：不可随机存取，存储密度低
创建	需要预分配大片连续空间。若分配空间过小，则之后不方便拓展容量；若分配空间过大，则浪费内存空间	只需分配一个头结点，之后方便拓展
销毁	修改Length=0，静态分配—自动回收，动态分配—需要手动free	一次删除各个结点
增加删除	插入/删除元素要将后续元素都后移/前移 时间复杂度为O(n),主要时间来自元素的移动 若元素很大，则移动时间带价很高	插入/删除只需要修改指针即可 时间复杂度为O(n),主要时间开销来自查找目标元素 查找元素的时间代价更低
查找	按位查找：O(1) 按值查找：O(n)	按位查找：O(n) 按值查找：O(n)

树、森林、二叉树的遍历

树	森林	二叉树
先根遍历	先序遍历	先序遍历
后根遍历	中序遍历	中序遍历

线索二叉树

	中序线索二叉树	先序线索二叉树	后序线索二叉树
找前驱	✓	×	✓
找后继	✓	✓	×

单元最短路径问题

	BFS 算法	Dijkstra 算法	Floyd 算法
无权图	✓	✓	✓
带权图	✗	✓	✓
带负权值的图	✗	✗	✓
带负权回路的图	✗	✗	✗
时间复杂度	$O(\|V\|^2)$或$O(\|V\|+\|E\|)$	$O(\|V\|^2)$	$O(\|V\|^3)$
通常用于	求无权图的单源最短路径	求带权图的单源最短路径	求带权图中各顶点间的最短路径

B树 vs B+树

	m阶B树	m阶B+树
类比	二叉查找树的进化——>m叉查找树	分块查找的进化——>多级分块查找
关键字与分叉	n个关键字对应n+1个分叉(子树)	n个关键字对应n个分叉
节点包含的信息	所有节点都包含记录的信息	只有最下层叶子结点才包含记录的信息(可使树更矮)
查找方式	不支持顺序查找。查找成功时，可能停在任何一层结点，查找速度“不稳定”	支持顺序查找。查找成功或失败都会到达最下一层结点，查找速度“稳定”
相同点	除根节点外，最少「m/21个分叉(确保结点不要太“空”") 任何一个结点的子树都要一样高（确保“绝对平衡”）	

内部排序

算法种类	时间复杂度			空间复杂度	是否稳定	特点
	最好情况	平均情况	最坏情况			
直接插入排序	O(n)	$O(n^2)$	$O(n^2)$	O(1)	是	1.当序列基本有序时，最适合 因为比较次数是最少的
冒泡排序	O(n)	$O(n^2)$	$O(n^2)$	O(1)	是	1.每趟确定一个元素的最终位置 2.每趟的比较次数是确定的， 但是趟数会根据序列不同而改变
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	否	1.每趟确定一个元素的最终位置 2.比较次数不会根据元素排列顺序的不同而改变
希尔排序				O(1)	否	1.因为使用了数组的随机存取特性 故使用链表会增加排序的时间复杂度
快速排序	O(nlogn)	O(nlogn)	$O(n^2)$	O(nlogn)	否	1.每趟确定一个元素的最终位置
堆排序	O(nlogn)	O(nlogn)	O(nlogn)	O(1)	否	1.因为使用了数组的随机存取特性 故使用链表会增加排序的时间复杂度 2.适用于大量数据寻找最大或最小的小部分数据
2路归并排序	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	是	
基数排序	O(d(n+r))	O(d(n+r))	O(d(n+r))	O(r)	是	1.元素的移动次数与初始的排列次序无关

易遗忘点

1. 中缀表达式转换方法
2. KMP算法next数组的计算
3. 二叉搜索的树的插入删除
4. 二叉平衡树的平衡

5. 迪杰斯特拉算法的流程
6. 弗洛伊德算法的流程
7. B树的高度、插入和删除操作
8. B+树的概念
9. 堆的建立、插入和删除
10. 排序算法比较的表格
11. 外部排序
12. 红黑树的插入
13. 并查集代码实现
14. 各个算法的时间复杂度和空间复杂度

错题

1. 6. 【2013 统考真题】已知两个长度分别为 m 和 n 的升序链表，若将它们合并为长度为 $m+n$ 的一个降序链表，则最坏情况下的时间复杂度是（ ）

   A. $O(n)$　　B. $O(mn)$　　C. $O(\min(m,n))$　　D. $O(\max(m,n))$

2. 10. 程序段如下：

```
for(i=n-1;i>1;i--)
    for(j=1;j<i;j++)
        if(A[j]>A[j+1])
            A[j]与A[j+1]对换;
```

   其中 n 为正整数，则最后一行语句的频度在最坏情况下是（ ）。

   A. $O(n)$　　B. $O(n\log n)$　　C. $O(n^3)$　　D. $O(n^2)$

   $(n-1)+(n-2)+\ldots\ldots+1=\frac{(n-1+1)*(n-1)}{2}=\frac{n^2+n}{2}$,所以时间复杂度为O($n^2$)

3. 10. 若长度为 n 的非空线性表采用顺序存储结构，在表的第 i 个位置插入一个数据元素，则 i 的合法值应该是（ ）。

   A. $1\le i\le n$　　B. $1\le i\le n+1$　　C. $0\le i\le n-1$　　D. $0\le i\le n$

   线性表元素的序号是从1开始的

   8. 将长度为 n 的单链表链接在长度为 m 的单链表后面，其算法的时间复杂度采用大 O 形式表示应该是（ ）。

   A. $O(1)$　　B. $O(n)$　　C. $O(m)$　　D. $O(n+m)$

4. 19. 一个链表最常用的操作是在末尾插入结点和删除结点，则选用（ ）最节省时间。

   A. 带头结点的双循环链表　　B. 单循环链表

   C. 带尾指针的单循环链表　　D. 单链表

5. 22. 静态链表中指针表示的是（ ）。

   A. 下一元素的地址　　B. 内存储器地址

   C. 下一个元素在数组中的位置　　D. 左链或右链指向的元素的地址

6. 5. 循环队列存储在数组 A[0…n] 中，入队时的操作为（ ）

   A. rear=rear+1　　B. rear=(rear+1) mod (n-1)

   C. rear=(rear+1) mod n　　D. rear=(rear+1) mod (n+1)

7. 14. 用链式存储方式的队列进行删除操作时需要（ ）。

   A. 仅修改头指针　　B. 仅修改尾指针

   C. 头尾指针都要修改　　D. 头尾指针可能都要修改

   队列用链式存储时，删除元素从表头删除，通常仅需修改头指针，但若队列中仅有一个元素，则尾指针也需要被修改，当仅有一个元素时，删除后队列为空，需修改尾指针为rear=front。

8.

3. 下面（ ）用到了队列。

A. 括号匹配　　B. 迷宫求解　　C. 页面替换算法　　D. 递归

9. 11.【2020 统考真题】将一个 10×10 对称矩阵 M 的上三角部分的元素 $m_{i,j}$（$1 \le i \le j \le 10$）按列优先存入 C 语言的一维数组 N 中，元素 $m_{7,2}$ 在 N 中的下标是（ ）。

A. 15　　B. 16　　C. 22　　D. 23

10. 3. 设主串的长度为 n，子串的长度为 m，则简单的模式匹配算法的时间复杂度为（ ），KMP 算法的时间复杂度为（ ）。

A. $O(m)$　　B. $O(n)$　　C. $O(mn)$　　D. $O(m+n)$

简单的模式匹配算法时间复杂度为O(mn),KMP算法的时间复杂度为O(m+n)

11. 3. 树的路径长度是从树根到每个结点的路径长度的（ ）。

A. 总和　　B. 最小值　　C. 最大值　　D. 平均值

12. 5. 在二叉树中有两个结点 m 和 n，若 m 是 n 的祖先，则使用（ ）可以找到从 m 到 n 的路径。

A. 先序遍历　　B. 中序遍历　　C. 后序遍历　　D. 层次遍历

13. 23 线索二叉树是一种（ ）结构。

A. 逻辑　　B. 逻辑和存储　　C. 物理　　D. 线性

14. 34.【2015 统考真题】先序序列为 a, b, c, d 的不同二叉树的个数是（ ）。

A. 13　　B. 14　　C. 15　　D. 16

遍历的时候，先序遍历是先访问节点，随后节点入栈，中序遍历是，出栈的时候，访问这个节点。

所以无论怎么出栈入栈，先序遍历就是取栈顺序，中序遍历就是出现顺序

15. 13 含有 20 个结点的平衡二叉树的最大深度为（ ）。

A. 4　　B. 5　　C. 6　　D. 7

平衡二叉树深度为h的最少结点为：$2^{h-2}-1+2^{h-3}/2+1$ 而此时，所有非叶子结点平衡因子均为1

估算$2^4-1=15$ $2^3/2=4$ 1 所以最大深度为4+2=6

16. 17【2012 统考真题】若平衡二叉树的高度为 6，且所有非叶子结点的平衡因子均为 1，则该平衡二叉树的结点总数为（ ）。

A. 12　　B. 20　　C. 32　　D. 33

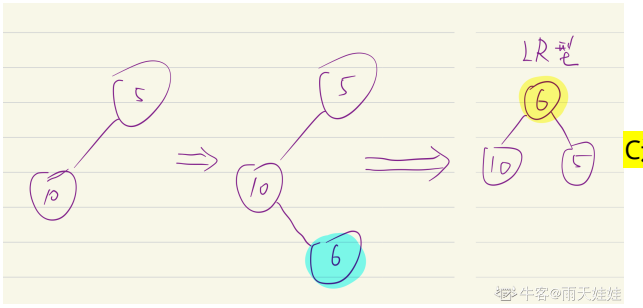
17. 18.【2013 统考真题】若将关键字 1, 2, 3, 4, 5, 6, 7 依次插入初始为空的平衡二叉树 T，则 T 中平衡因子为 0 的分支结点的个数是（ ）。

A. 0　　B. 1　　C. 2　　D. 3

18. 28.【2015 统考真题】现有一棵无重复关键字的平衡二叉树（AVL），对其进行中序遍历可得到一个降序序列。下列关于该平衡二叉树的叙述中，正确的是（ ）。

A. 根结点的度一定为 2　　B. 树中最小元素一定是叶结点

C. 最后插入的元素一定是叶结点　　D. 树中最大元素一定是无左子树



C选项错误

19. 27. 若度为 m 的哈夫曼树中，叶子结点个数为 n，则非叶子结点的个数为（ ）。

A. $n-1$　　B. $\lfloor n/m \rfloor - 1$

C. $\lceil (n-1)/(m-1) \rceil$　　D. $\lceil n/(m-1) \rceil - 1$

20. 33. 【2019 统考真题】在任意一棵非空平衡二叉树（AVL 树）T_1 中，删除某结点 v 之后形成平衡二叉树 T_2，再将 v 插入 T_2 形成平衡二叉树 T_3。下列关于 T_1 与 T_3 的叙述中，正确的是（ ）。

I. 若 v 是 T_1 的叶结点，则 T_1 与 T_3 可能不相同

II. 若 v 不是 T_1 的叶结点，则 T_1 与 T_3 一定不相同

III. 若 v 不是 T_1 的叶结点，则 T_1 与 T_3 一定相同

A. 仅 I　　B. 仅 II　　C. 仅 I、II　　D. 仅 I、III

II错误

21. 16. 设有无向图 $G=(V,E)$ 和 $G'=(V',E')$，若 G' 是 G 的生成树，则下列不正确的是（ ）。

I. G' 为 G 的连通分量

II. G' 为 G 的无环子图

III. G' 为 G 的极小连通子图且 $V'=V$

A. I、II　　B. 只有 III　　C. II、III　　D. 只有 I

22. 12. 判断有向图中是否存在回路，除可以利用拓扑排序外，还可以利用（ ）。

A. 求关键路径的方法　　B. 求最短路径的 Dijkstra 算法

C. 深度优先遍历算法　　D. 广度优先遍历算法

23. 2. 由 n 个数据元素组成的两个表：一个递增有序，一个无序。采用顺序查找算法，对有序表从头开始查找，发现当前元素已不小于待查元素时，停止查找，确定查找不成功，已知查找任一元素的概率是相同的，则在两种表中成功查找（ ）。

A. 平均时间后者小　　B. 平均时间两者相同

C. 平均时间前者小　　D. 无法确定

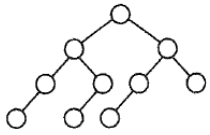
24. 13. 已知一个长度为 16 的顺序表，其元素按关键字有序排列，若采用折半查找算法查找一个不存在的元素，则比较的次数至少是（ ），至多是（ ）。

A. 4　　B. 5　　C. 6　　D. 7

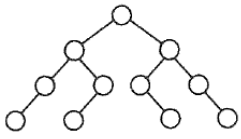
考察当元素个数为偶数时折半查找中间结点左边元素数量比右边元素数量少一个

25. 15. 【2017 统考真题】下列二叉树中，可能成为折半查找判定树（不含外部结点）的是（ ）。

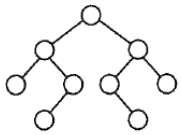
A.



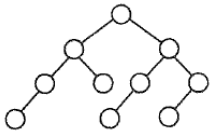
B.



C.



D.



折半查找判定树实际上是一棵二叉排序树，它的中序序列是一个有序序列。可以在树结点上依次填上相应的元素，符合折半查找规则的树即是所求

26. 7. 具有 n 个关键字的 m 阶 B 树，应有（ ）个叶结点。

A. $n+1$　　B. $n-1$　　C. mn　　D. $nm/2$

27. 5. 下列关于散列冲突处理方法的说法中，正确的有（ ）。

I. 采用再散列法处理冲突时不易产生聚集

II. 采用线性探测法处理冲突时，所有同义词在散列表中一定相邻

III. 采用链地址法处理冲突时，若限定在链首插入，则插入任一个元素的时间是相同的

IV. 采用链地址法处理冲突易引起聚集现象

A. I 和 III　　B. I、II 和 III　　C. III 和 IV　　D. I 和 IV

28. 9. 采用开放定址法解决冲突的散列查找中，发生聚集的原因主要是（ ）。

A. 数据元素过多　　B. 负载因子过大

C. 散列函数选择不当　　D. 解决冲突的方法选择不当

29. 9. 一般情况下，以下查找效率最低的数据结构是（ ）。

A. 有序顺序表　　B. 二叉排序树　　C. 堆　　D. 平衡二叉树

堆只能用来排序，查找时是无序的

30. 4. 对任意 7 个关键字进行基于比较的排序，至少要进行（ ）次关键字之间的两两比较。

A. 13　　B. 14　　C. 15　　D. 6

$log_2 7!$向上取整 $= 13$

31. 10.【2015 统考真题】下列排序算法中，元素的移动次数与关键字的初始排列次序无关的是（ ）。

A. 直接插入排序　　B. 起泡排序　　C. 基数排序　　D. 快速排序

基数排序每趟每一个元素一定会都移动

32. 11.【2017 统考真题】下列排序方法中，若将顺序存储更换为链式存储，则算法的时间效率

会降低的是（ ）。

I. 插入排序　II. 选择排序　III. 起泡排序　IV. 希尔排序　V. 堆排序

A. 仅 I、II　　B. 仅 II、III　　C. 仅 III、IV　　D. 仅 IV、V

33. 14.【2020 统考真题】对大部分元素已有序的数组排序时，直接插入排序比简单选择排序效率更高，其原因是（ ）。

I. 直接插入排序过程中元素之间的比较次数更少

II. 直接插入排序过程中所需的辅助空间更少

III. 直接插入排序过程中元素的移动次数更少

A. 仅 I　　B. 仅 III　　C. 仅 I、II　　D. I、II 和 III

34.

3. 下列选项中，正确的是（A）

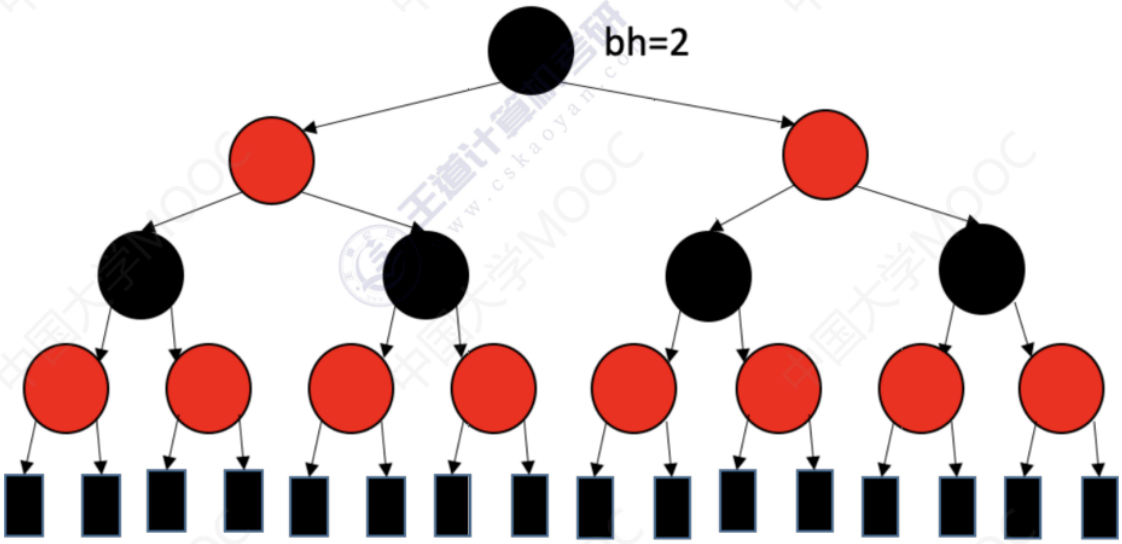
① 若红黑树根节点黑高度为h，则内部结点数（关键字）最少有 2^h-1 个

② 若红黑树根节点黑高度为h，则内部结点数（关键字）最多有 2^{2h}-1 个

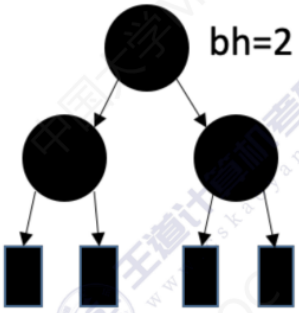
③ 包含n个关键字的红黑树，高度不超过 $2\log_2(n+1)$

④ 红黑树中，红结点的数量不会超过内部结点总数的一半

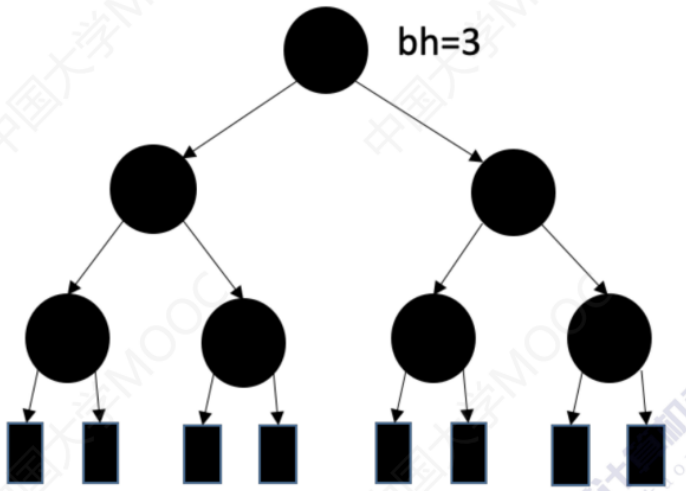
A.③④　B.①③④　C.④　D.①②③



根节点黑高=2，内部结点数最多的情况



根节点黑高=2，内部结点数最少的情况



根节点黑高=3，内部结点数最少的情况