数据结构基础练习题

简介

- `Author` : *GoogTech*
- `Website` : *https://algorithm.show*
- `Email` : *GoogTech@qq.com*

- `Date` : *2021,6,19 ~ 2021,6,23*

目录

点击链接即可完成快速跳转哈.

选择题

概念

1、计算机资源管理器中, 对文件(夹)的组织用的什么数据结构?
A、 线性表结构
B、 树结构
C、 图结构
D、 集合

> 答案: B

2、在存储数据时, 通常不仅要存储各数据元素的值, 而且还要存储_。
A、 数据的存储方法
B、 数据元素的类型
C、 数据的处理方法
D、 数据元素之间的关系

> 答案: D

3、通常要求同一逻辑结构中的所有数据元素具有相同的特性, 这意味着_。
A、 数据元素具有同一特点
B、 数据元素所包含的数据项的个数要相等
C、 每个数据元素都一样
D、 不仅数据元素所包含的数据项的个数要相同, 而且对应的数据项的类型要一致

> 答案: D

4、以下说法正确的是_。
A、 数据元素是数据的最小单位
B、 数据结构是带结构的各数据项的集合
C、 数据项是数据的基本单位
D、 一些表面上很不相同的数据可以有相同的逻辑结构

> 答案: D
>
> A : 数据元素是数据的基本单位
>
> B : 数据结构是相互之间存在一种或多种特定关系的数据元素的集合
>
> C : 数据项是数据元素不可分割的最小单位, 一个数据元素可由若干个数据项(`Data Item`)组成

6、_表示`结点`之间的`层次`关系
A、 线性表结构
B、 树结构
C、 图结构
D、 散结构

> 答案: B, 同第 1 题哈~

7、下面关于数据的`逻辑结构`与`存储结构`说法正确的是_
A、 逻辑结构要体现出存储结构
B、 存储结构要体现出逻辑结构
C、 二者含义是一样的
D、 二者毫无关系

> 答案: B
>
> 解析 : 存储结构是逻辑结构的`存储映像`, 逻辑结构指的是数据间的关系, 它又分为`线性结构`和`非线性结构`, 这两者并不冲突.

8、下列说法不正确的是_
A、 算法的正确性通常需要人工证明
B、 只有对所有合法输入数据都正确的算法才是正确的算法
C、 经反复调试、长期运行, 也不一定能够发现算法中的错误
D、 只有经长期运行, 从不出错的算法, 才是正确的算法

> 答案: D
>
> 解析 : 算法的五大特性为如下 :
>
> - 输入 : 算法具有 0 个或多个输入.
> - 输出 : 算法至少有 1 个或多个输出.
> - 有穷性 : 算法在有限的步骤之后会自动结束而不会无限循环.
> - 确定性 : 算法中的每一步都有确定的含义, 不会出现二义性.
> - 可行性 : 算法的每一步都是可行的.

240、(重复题目) 下面关于数据的逻辑结构与存储结构说法正确的是_
A、 逻辑结构要体现出存储结构
B、 存储结构要体现出逻辑结构
C、 二者含义是一样的
D、 二者毫无关系

> 答案: B

245、在数据结构中,从逻辑上可以把数据结构分成_。
A、 内部结构和外部结构
B、 动态结构和静态结构
C、 紧凑结构和非紧凑结构
D、 线性结构和非线性结构

> 答案: D

246、数据结构在计算机内存中的表示是指_。
A、 数据的逻辑结构
B、 数据元素之间的关系
C、 数据关系
D、 数据的存储结构

> 答案: D

247、在数据结构中, 与所使用的计算机无关的是数据的_结构。
A、 逻辑
B、 存储
C、 物理
D、 逻辑和存储

> 答案: A
>
> 解析 : 数据结构概念一般包括 3 个方面的内容, 即数据的逻辑结构、存储结构及数据上的运算集合. 数据的逻辑结构只抽象地反映数据元素之间的逻辑关系, 而不管它在计算机中的存储表示形式.
>
> 算法的设计取决于数据的逻辑结构, 而算法的实现依赖于指定的存储结构.

250、(重复题目) 以下说法正确的是_。
A、 数据元素是数据的最小单位
B、 数据结构是带结构的各数据项的集合
C、 数据项是数据的基本单位
D、 一些表面上很不相同的数据可以有相同的逻辑结构

> 答案: D

254、数据结构是( )。
A、 一种数据类型
B、 数据的存储结构
C、 一组性质相同的数据元素的集合

D、 相互之间存在`一种`或`多种`特定`关系`的`数据元素`的`集合`

> 答案: D

259、下列有关数据结构的说法中，正确的是
A、 数据的逻辑结构独立于其存储结构
B、 数据的存储结构独立于其逻辑结构
C、 数据结构仅由其逻辑结构和存储结构决定
D、 数据的逻辑结构唯一决定其存储结构

> 答案: A
>
> 解析 : 数据的逻辑结构可以独立于存储结构来考虑, 反之不可以.
>
> 数据的逻辑结构是用于描述数据之间的关系, 而数据的存储结构是与相应数据在内存中的物理地址有关的, 它是数据的逻辑结构在计算机上的具体语言实现.
>
> 例如我们熟知的线性表, 可以是顺序表, 也可以是链表. 它们在逻辑结构上都属于“线性表”的范畴, 但在存储结构上顺序表的方式是基于一片连续的物理地址, 而链表并不连续. 对于一种数据存储结构而言, 必定有包括它的逻辑结构, 既某一种逻辑结构在计算机语言中映射其存储结构.

261、以下属于逻辑结构的是
A、 顺序表
B、 哈希表
C、 有序表
D、 单链表

> 答案: C
>
> 解析 : 顺序表、哈希表和单链表表示几种数据结构, 既描述逻辑结构, 也描述存储结构和数据运算, 而有序表是指关键字有序的线性表, 可以链式存储也可以顺序存储, 仅描述了元素之间的逻辑关系, 故它属于逻辑结构.

262、以下与数据的存储结构无关的术语是（）
A、 循环队列
B、 链表
C、 哈希表
D、 栈

> 答案: D
>
> 解析 : 数据的存储结构有 :
>
> - 顺序存储 (A)
> - 链式存储 (B)
> - 索引存储 (C)
> - 散列存储, 又称 hash 存储

266、算法的`有效性`指的是_
A、 `时间复杂性`和`空间复杂性`
B、 最坏情况和平均情况
C、 制作周期与使用时效的关系

D、 制作费用与实用价值的关系

> 答案: A

271、(重复题目) 下列说法不正确的是_
A、 算法的正确性通常需要人工证明
B、 只有对所有合法输入数据都正确的算法才是正确的算法
C、 经反复调试、长期运行,也不一定能够发现算法中的错误
D、 只有经长期运行,从不出错的算法,才是正确的算法

> 答案: D

275、算法分析的目的是。
A、 找出数据结构的合理性
B、 研究算法中的输入和输出的关系
C、 分析算法的易懂性和文档性
D、 分析算法的效率以求改进

> 答案: D

277、计算机算法必须具备输入、输出和等 5 个特性。
A、 确定性、有穷性和稳定性
B、 易读性、稳定性和安全性
C、 可行性、确定性和有穷性
D、 可行性、可移植性和可扩充性

> 答案: C

278、分析以下程序段的时间复杂度

```
x = 9000; y = 10000;
while(y > 0) {
   if(x > 100) {
        x = x - 10;
        y--;
   }
   else {
        x++;
   }
}
```

A、 T(n) =O(1)
B、 T(n) =O(y)

> 答案： A

281、分析以下程序段的时间复杂度

```
i = 1;
  while(i <= n)
     i = i * 3;
```

A、 T(n) =O(n)
B、 T(n) =O(log3n)
C、 T(n) =O(n^3)
D、 T(n) =O(n^2)

> 答案： B

表

1、顺序表中第一个元素的存储地址是100, 每个元素的长度为2, 则第5个元素的地址是( )。
A、 110
B、 108
C、 100
D、 120

> 答案 : B
>
> 解析 : 顺序表中的数据连续存储, 故第 5 个元素的地址为 100 + (2 * 4) = 108

2、在n个结点的顺序表中,算法的时间复杂度是O(1)的操作是( )。
A、 访问第 i 个结点(1 ≤ i ≤ n)和求第i个结点的直接前驱(2 ≤ i ≤ n)
B、 在第i个结点后插入一个新结点(1 ≤ i ≤ n)
C、 删除第i个结点(1 ≤ i ≤ n)
D、 将n个结点从小到大排序

> 答案 : A
>
> 解析 : 在顺序表中插入或删除一个结点的时间复杂度都是`O(n²)`, 排序的时间复杂度为`O(n²)`或`O(nlog2n)`. 顺序表是一种随机存取结构, 访问第 i 个结点和求第 i 个结点的直接前驱都可以直接通过数组的下标直接定位, 时间复杂度是`O(1)`.

3、向一个有127个元素的顺序表中插入一个新元素并保持原来顺序不变,平均要移动 的元素个数为( )。
A、 8
B、 63.5
C、 63
D、 7

> 答案 : B
> 解析 : 顺序表中插入元素平均要移动的元素个数为 : `n / 2`

4、链接存储的存储结构所占存储空间( )。
A、 分两部分, 一部分存放`结点值`, 另一部分存放表示结点间关系的`指针`
B、 只有一部分, 存放结点值
C、 只有一部分, 存储表示结点间关系的指针
D、 分两部分, 一部分存放结点值, 另一部分存放结点所占单元数

> 答案 : A

5、线性表若采用`链式`存储结构时, 要求内存中可用存储单元的地址( )。
A、 必须是连续的
B、 部分地址必须是连续的
C、 一定是不连续的
D、 连续或不连续都可以

> 答案 : D
>
> 扩 : 线性表若采用`顺序`存储结构时, 则要求内存中可用存储单元的地址必须是连续的.

6、线性表L在( )情况下适用于使用链式结构实现。
A、 需经常修改L中的结点值
B、 需不断对L进行删除插入
C、 L中含有大量的结点
D、 L中结点结构复杂

> 答案 : B
> 解析 : 链表最大的优点在于插入和删除时不需要移动数据, 直接修改指针即可, 其时间复杂度为`O(1)`. 而顺序表插入和删除节点后都需要移动元素, 故时间复杂度都是`O(n²)`.

8、将两个各有n个元素的有序表归并成一个有序表, 其最少的比较次数是( )。
A、 n
B、 2n-1
C、 2n
D、 n-1

> 答案 : A
>
> 解析 : 当第一个有序表中所有的元素都小于或大于第二个表中的元素时, 只需要用第二个表中的第一个元素依次与第一个表的元素比较, 即最少的比较次数为`N`次. 例如 `[1,2,3]` 与 `[4,5,6]` 合并仅比较了 3 次.
>
> 最坏的情况下比较次数为`2N -1`, 即当两个有序表的数据刚好是插空顺序的时候, 例如 `[1,3,5]` 与 `[2,4,6]` 进行合并需要比较 5 次.

10、线性表L=(a1, a2, ..., an), 下列说法正确的是( )。
A、 每个元素都有一个直接前驱和一个直接后继
B、 线性表中至少有一个元素
C、 表中诸元素的排列必须是由小到大或由大到小
D、 除第一个(`头节点`)和最后一个元素(`尾节点`)外, 其余每个元素都有一个且`仅有一个`直接`前驱`和直接`后继`

> 答案 : D

12、以下说法错误的是( )。
A、 求表长、定位这两种运算在采用顺序存储结构时实现的效率不比采用链式存储结构时实现的效率低
B、 顺序存储的线性表可以随机存取
C、 由于顺序存储要求连续的存储区域, 所以在存储管理上不够灵活
D、 线性表的链式存储结构优于顺序存储结构

> 答案 : D
>
> 解析 : 顺序表的特点 :
>
> - 可按元素序号随机访问
> - 不用为表示节点间的逻辑关系而增加额外的存储开
>
> 顺序表的缺点 :
>
> - 进行插入或删除操作时, 平均移动表中的一半元素
> - 需要预先分配足够大的存储空间, 过大则导致大量闲置, 过小则会造成溢出
>
> 链表的最大特点为插入和删除运算方便, 其缺点为 :
>
> - 其为随机存储结构, 故不能随机存取元素
> - 需占用额外的存储空间来存储元素之间的关系, 故存储密度较低

13、在单链表中,要将 s 所指结点插入到 p 所指结点之后, 其语句应为( )。
A、 s->next=p+1; p->next=s;
B、 (*p).next=s; (s).next=(*p).next;
C、 s->next=p->next; p->next=s->next;
D、 s->next=p->next; p->next=s;

> 答案 : D
>
> 解析 :
>
> 1. `s -> next = p -> next;` : 将 s 节点的指针指向 p 节点的后一个节点
> 2. `p -> next = s;` : 使 p 节点的指针指向 s 节点

14、在`双向`链表存储结构中, 删除 p 所指的结点时须修改指针( )。
A、 p->next->prior=p->prior; p->prior->next=p->next;
B、 p->next=p->next->next; p->next->prior=p;
C、 p->prior->next=p; p->prior=p->prior->prior;
D、 p->prior=p->next->next; p->next=p->prior->prior;

> 答案 : A
>
> 解析 :
>
> 1. `p -> next -> prior = p -> prior;` : 将 p 节点的后继节点的前驱指针, 指向 p 的前驱节点
> 2. `p -> prior -> next = p -> next;` : 使 p 节点的前驱节点的后继指针, 指向 p 的后继节点

15、在`双向循环`链表中, 在 p 指针所指的结点后插入 q 所指向的新结点, 其修改指针的操作是( )。
A、 p->next=q; q->prior=p; p->next->prior=q; q->next=q;
B、 p->next=q; p->next->prior=q; q->prior=p; q->next=p->next;
C、 q->prior=p; q->next=p->next; p->next->prior=q; p->next=q;

D、 q->prior=p; q->next=p->next; p->next=q; p->next->prior=q;

> 答案 : C
>
> 解析 :
>
> 1. `q -> prior = p;` : 将 q 的前驱指针指向 p
> 2. `q -> next = p -> next;` : 将 q 的后继指针指向 p 的后继节点
> 3. `p -> next -> prior = q;` : 将 p 的后继节点的前驱指针, 指向 q
> 4. `p -> next = q` : 将 p 的后继节点指向 q

16、`串`是一种特殊的线性表, 其特殊性体现在( )。
A、 可以顺序存储
B、 数据元素是一个字符
C、 可以链式存储
D、 数据元素可以是多个字符

> 答案 : B
>
> 解析 : 串是由零个或多个字符组成的有限序列.

18、在线性表的下列运算中, 不改变数据元素之间`结构关系`的运算是( )。
A、 插入
B、 删除
C、 排序
D、 查找

> 答案 : D

19、线性表采用链式存储时, 其地址( )。
A、 必须是连续的
B、 一定是不连续的
C、 部分地址必须连续
D、 连续与否均可以

> 答案 : D

20、线性表是( )。
A、 一个有限序列,可以为空
B、 一个有限序列,不可以为空
C、 一个无限序列,可以为空
D、 一个无限序列,不可以为空

> 答案 : A
>
> 解析 : 一个线性表是 n 个具有相同特性的数据元素的有限序列.

27、以下关于线性表的说法不正确的是__。
A、 线性表中的数据元素可以是数字、字符、记录等不同类型。
B、 线性表中包含的数据元素个数不是任意的。
C、 线性表中的每个结点都有且只有一个直接前趋和直接后继。

D、 存在这样的线性表:表中各结点都没有直接前趋和直接后继。

> 答案 : C

28、线性表的顺序存储结构是一种___的存储结构。
A、 随机存取
B、 顺序存取
C、 索引存取
D、 散列存取

> 答案 : A
>
> 解析 : 顺序存储结构的地址在内存中是连续的, 所以可以通过计算地址实现随机存取. 而链式存储结构的存储地址不一定连续, 只能通过第个结点的指针顺序存取.

29、在顺序表中, 只要知道___, 就可在相同时间内求出任一结点的存储地址。
A、 基地址
B、 结点大小
C、 向量大小
D、 基地址和结点大小

> 答案 : D
>
> 解析 : 顺序表是一组地址连续的存储单元, 依次存储表中的元素, 所以可以根据首元素的地址和节点所占空间大小, 直接访问顺序表中任一节点.

31、在__运算中, 使用顺序表比链表好。
A、 插入
B、 删除
C、 根据序号查找
D、 根据元素值查找

> 答案 : C
>
> 解析 : 可按元素序号随机访问是顺序表的特点之一.

58、将长度为 n 的单链表链接在长度为 m 的单链表后面, 其时间复杂度采用大 O 形式表示应该是（）。
A、 O(1)
B、 O(n)
C、 O(m)
D、 O(n+m)

> 答案 : C
>
> 解析 : 需要先遍历长度为 m 的链表, 找到链表尾部, 这个时间复杂度为`O(m)`, 然后再将链表尾部的 next 指针指向长度为 n 的链表的头结点即可.

59、与单链表相比，双链表的优点之一是（ ）
A、 插入、删除操作更简单
B、 可以随机访问
C、 可省略表头指针或表尾指针
D、 访问前后相邻结点更方便

> 答案 : D
>
> 解析 : 双链表比单链表多了`prior`指针, 所以访问前后相邻结点更方便了, 总的来说就是用空间换取时间.

60、设带带头结点的循环单链表L, 当L为空表时应满足（ ）
A、 表头结点指针域next为空即L->next==NULL
B、 L-的值为NULL即L==NULL
C、 表头结点的指针域next与L的值相等即L->next==L
D、 表头结点的指针域next与L的地址相等

> 答案 : C
>
> 解析 : `L -> next == L;` : 即单链表 L 的指针指向自己.

61、在长度为n(n>=1)的双链表中删除一个结点（非尾结点）要修改（）个指针。
A、 1
B、 2
C、 3
D、 4

> 答案 : B

62、与非循环单链表相比，循环单链表的主要优点是
A、 不再需要头指针
B、 已知某一结点的位置后，能够容易找到它的前驱结点
C、 在进行插入、删除操作时，能更好地保持链表不断开
D、 在表中任意结点出发都能扫描到整个链表

> 答案 : D

63、将两个分别含有m、n个结点的有序单链表`归并`成一个有序单链表, 对应算法的时间复杂度是（）。
A、 O(n)
B、 O(m)
C、 O(n+m)
D、 O(MIN(n+m)）

> 答案 : C

64、对于长度位n(n>=1)的`双链表`L, 在 p 所指结点之前插入一个新结点的算法时间复杂度为（ ）。
A、 O(1)
B、 O(n)
C、 O(n^2)
D、 O(nlog2n)

> 答案 : A
>
> 解析 : 因双链表较单链表有前驱指针, 所以在指定节点前进行的插入操作只需修改指针即可哈.

65、在长度为n（n>=1）的`循环双链表`L中，删除尾结点的时间复杂度为（ ）。
A、 O(1)
B、 O(n)
C、 O(n^2)
D、 O(nlog2n)

> 答案 : A
>
> 解析 : 在非空的循环双向链表中, 头节点的前驱指针指向的节点即为尾结点.

66、将两个分别含有m、n个结点的有序单链表`归并`成一个有序单链表, 要求不破坏原有的单链表, 对应算法的时间复杂度是（ ）。
A、 O(n)
B、 O(n)
C、 O(n+m)
D、 O(MIN(m,n))

> 答案 : C

67、在长度为n（n>=1）的`双链表` L 中, `删除 p 所指结点`的时间复杂度为（ ）。
A、 O(1)
B、 O(n)
C、 O(n^2)
D、 O(nlog2n)

> 答案 : A
>
> 解析 : 先遍历`O(n²)`后删除`O(1)`. 应选 C, 答案有误？

82、(`重复题目`) 以下说法错误的是( )。
A、 求表长、定位这两种运算在采用顺序存储结构时实现的效率不比采用链式存储结构时实现的效率低
B、 顺序存储的线性表可以随机存取
C、 由于顺序存储要求连续的存储区域,所以在存储管理上不够灵活
D、 线性表的链式存储结构优于顺序存储结构

> 答案 : D
>
> 解析 : 链式存储结构和顺序存储结构各有优缺点, 有不同的适用场合.

栈

86、若进栈序列为1,2,3,4,5,6, 且进栈和出栈可以`穿插`进行, 则可能出现的出栈序列为( )。
A、 3,2,6,1,4,5
B、 3,4,2,1,6,5
C、 1,2,5,3,4,6
D、 5,6,4,2,3,1

> 答案 : B
>
> 解析 : Stack 的特点为 First In Last Out / Last In First Out 哈.
>
> - A 应改写为 : 3, 2, 1, 6, 5, 4
> - C 应改写为 : 1, 2, 5, 4, 3, 6
> - D 应该写为 : 5, 6, 4, 3, 2, 1

87、若一个栈的输人序列是1,2,3,...,n,输出序列的第一个元素是n,则第k个输出元素是( )。
A、 k
B、 n-k-1
C、 n-k+1
D、 不确定

> 答案 : C
>
> 解析 : 因为输出的第一个数是`n`, 则此时 1, 2, 3, ..., n-1已入栈, 进而第二个输出的是`n - 1`, 第三个输出的是`n - 2`, 以此类推, 第 i 个输出的是 `n - i + 1`.

88、栈和队列的共同点是( )。
A、 都是先进先出
B、 都是先进后出
C、 只允许在端点处插入和删除元素
D、 没有共同点

> 答案 : C
>
> 解析 : 栈在栈顶操作元素, 而队列在队尾添加(push)元素, 在队头删除(pop)元素.

92、若已知一个栈的入栈序列是 1, 2, 3, ..., n, 其输出序列为p1, p2, p3, ..., pn, 若 p1 = n, 则 pi 为( )
A、 i
B、 n-i
C、 n-i+1
D、 不确定

> 答案 : C
> 解析 : 因为输出的第一个数是`n`, 则此时 1, 2, 3, ..., n-1已入栈, 进而第二个输出的是`n - 1`, 第三个输出的是`n - 2`, 以此类推, 第 i 个输出的是 `n - i + 1`.

93、(`重复题目`)栈和队列的共同点是( )
A、 都是先进后出
B、 都是先进先出
C、 只允许在端点处插入和删除元素

D、 没有共同点

> 答案 : C

94、若依次输入数据元素序列{a, b, c, d, e, f, g}进栈,出栈操作可以和入栈操作`间隔`进行, 则下列哪个元素序列可以由出栈序列得到?( )
A、 {d,e,c,f,b,g,a}
B、 { f,e,g,d,a,c,b}
C、 {e,f,d,g,b,c,a}
D、 { c,d,b,e,g,a,f}

> 答案 : A
>
> 解析 : Stack 的特点为 First In Last Out / Last In First Out 哈.

95、一个栈的入栈序列是1, 2, 3, 4, 5,则下列序列中不可能的出栈序列是( )
A、 2,3,4,1,5
B、 5,4,1,3,2
C、 2,3,1,4,5
D、 1,5,4,3,2

> 答案 : B
>
> 解析 : Stack 的特点为 First In Last Out / Last In First Out 哈.
>
> B 选项中元素 1 比元素3, 2先出栈, 其违背了栈的后进先出原则.

96、栈的插入与删除是在( )进行。
A、 栈顶
B、 栈底
C、 任意位置
D、 指定位置

> 答案 : A

97、假设顺序栈的定义为:

```
typedef struct {
selemtype *base; /* 栈底指针 */
selemtype *top; /* 栈顶指针 */
int stacksize; /* 当前已分配的存储空间,以元素为单位 */
}sqstack;
```

变量 st 为 sqstack 型, 则栈 st 为空的判断条件为( )。
A、 st.base == NULL
B、 st.top == st.stacksize
C、 st.top-st.base>= st.stacksize
D、 st.top == st.base

> 答案 : D
>
> 解析 : 即当栈顶与栈底指针指向同一块区域时(`st.top == st.base`) 则表示栈为空.

98、假设顺序栈的定义为:

```
typedef struct {
selemtype *base; /* 栈底指针 */
selemtype *top; /* 栈顶指针*/
int stacksize; /* 当前已分配的存储空间,以元素为单位*/
}sqstack;
```

变量 st 为 sqstack 型, 则栈 st 为满的判断条件为( )。
A、 st.base == NULL
B、 st.top == st.stacksize
C、 st.top-st.base>= st.stacksize
D、 st.top == st.base

> 答案 : C 题目所给的条件不足, 答案有争议 !

99、链式栈结点为 : (data,link), top指向栈顶. 若想摘除栈顶结点, 并将删除结点的值保存到x中, 则应执行操作( )。
A、 x=top->data;top=top->link;
B、 top=top->link;x=top->link;
C、 x=top;top=top->link;
D、 x=top->link;

> 答案 : A
> 解析 :
>
> 1. `x = top -> data;` 将结点的值保存到 x 中,
> 2. `top = top -> link;` 将栈顶指针指向栈顶下一结点, 即摘除栈顶结点.

101、在一个链队列中, 假定front和rear分别为头指针和尾指针, 删除一个结点的操作是( )。
A、 front=front->next
B、 rear=rear->next
C、 rear->next=front
D、 front->next=rear

> 答案 : A

104、数组 [Qn]用来表示一个循环队列, f为当前队列头元素的前一位置, r为队尾元素的位置, 假定队列中元素的个数小于n, 计算队列中元素个数的公式为( ).
A、 r-f
B、 (n+f-r)%n
C、 n+r-f
D、 (n+r-f)%n

> 答案 : D
>
> 解析 : 综下两种情况所述, 计算队列中元素个数的公式为 : `(n + r - f) % n`
>
> 1. 当 rear > front 时: 队列中元素的个数为 : `r - f`
> 2. 当 front > rear 时: 队列中元素的格式为 `n + r - front`

> 对于非循环队列, 尾指针和头指针的差值便是队列的长度. 而对于循环队列, 差值可能为负数, 所以需要将差值加上 MAXSIZE (本题为n), 然后与 MAXSIZE 求余, 即(n + r - f) % n.

105、为解决计算机主机与打印机间速度不匹配问题, 通常设一个打印数据缓冲区. 主机将要输出的数据依次写入该缓冲区, 而打印机则依次从该缓冲区中取出数据. 该缓冲区的逻辑结构应该是( )。

A、 队列
B、 栈
C、 线性表
D、 有序表

> 答案 : A
> 解析 : 解决缓冲区问题应利用一种先进先出的线性表, 而队列正是一种先进先出的线性表.

106、设栈S和队列Q的初始状态为空,元素e1、e2、e3、e4、e5和e6依次进入栈S, 一个元素出栈后即进入Q, 若6个元素出队的序列是e2、e4、e3、e6、e5和e1,则栈S的容量至少应该是( )。

A、 2
B、 3
C、 4
D、 6

> 答案 : B
> 解析 : 元素出队的序列是`e2、e4、e3`、e6、e5和e1, 可知元素入队的序列是`e2、e4、e3`、e6、e5和e1, 即元素出栈的序列也是`e2、e4、e3`、e6、e5和e1, 而元素`e1、e2、e3`、e4、e5和e6依次进入栈,易知栈 S 中最多同时存在 3 个元素, 故栈 S 的容量至少为 3.

108、用链接方式存储的队列, 在进行删除运算时( )。

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

> 答案 : D
> 解析 : 在队首进行操作, 由于是链式存储, 删除结点后, 需要修改队首的指针, 使其指向下一个结点. 但如果队列中只有这一个结点, 这时候头、尾指针都指向这个结点, 在删除结点后, 头、尾指针都需要修改.

109、`循环队列`存储在数组A[0..m]中, 则入队时的操作为( )。

A、 rear=rear+1
B、 rear=(rear+1)%(m-1)
C、 rear=(rear+1)%m
D、 rear=(rear+1)%(m+1)

> 答案 : D
> 解析 : 数组 A[0..m] 中共含有 m + 1 个元素, 代表有m + 1个空间, 因为有个 0 在前面.
>
> - 入队: `rear = (rear + 1) % (m + 1)`
> - 出队: `front =（front + 1）% (m + 1)`

111、用单链表表示的`链式队列`的队头在链表的( )位置。
A、 表头
B、 表尾
C、 表中
D、 不确定

> 答案 : A
>
> 解析 : 单链表所表示的`链式队列`的队头在链表的表头, 队尾在链表的表尾.

112、栈的“先进后出”特性是指（ ）
A、 最后进栈的元素总是先出栈
B、 同时进行进栈和出栈操作时，总是进栈优先
C、 每当有出栈操作时，总要先进行一次进栈操作
D、 每次出栈的元素总是最先进栈的元素

> 答案 : A
>
> 解析 : 栈的特性为先进后出(`FILO`) / 后进先出(`LIFO`).

113、一个栈的进栈序列是a, b, c, d, e,则栈的不可能的输出序列是（ ）
A、 edcba
B、 decba
C、 dceab
D、 adcde

> 答案 : C
>
> 解析 : C选项应改成 d, c, e, b, a

115、若一个栈用数组data[1..n]存储, 初始栈顶指针 top 为n+1, 则以下元素 x 进栈的正确操作是( ).
A、 top++;data[top]=x;
B、 data[top]=x;top++
C、 top--;data[top]=x;
D、 data[top]=x;top--

> 答案 : C
>
> 1. `top--;` : 首先下移栈顶指针, 使其指向 [1 ~ n] 中的尾元素, 因为初始栈顶指针 top 为 n + 1.
> 2. `data[top] = x;` : 其次将进栈元素存储到 top 指针所指向的内存空间哈.

116、若一个栈用数组data[1..n]存储, 初始栈顶指针 top 为0, 则以下元素 x 进栈的正确操作是( ).
A、 top++;data[top]=x;
B、 data[top]=x;top++
C、 top--;data[top]=x;
D、 data[top]=x;top--

> 答案 : A
>
> 1. `top++;` : 首先上移栈顶指针, 使其指向 [1 ~ n] 中的首元素, 因为初始栈指针 top 为 0.
> 2. `data[top] = x;` : 其次将进栈元素存储到 top 指针所指向的内存空间哈.

117、在设计链栈时, 通常采用单链表作为链栈, 而不采用双链表作为链栈, 其准确原因是（ ）
A、 栈中元素是顺序存取的，用单链表就足够了
B、 栈中元素是随机存取的，用单链表就足够了
C、 双链表运算较单链表更复杂
D、 双链表较单链表存储密度低

> 答案 : A

118、(重复题目)栈和队列的相同点是
A、 都是线性表
B、 都是链表
C、 只允许在端点处插入和删除元素
D、 没有共同点

> 答案 : C

119、若某循环队列有队首指针front和队尾指针rear, 在队`不空`时出队操作仅会改变（ ）。
A、 front
B、 rear
C、 front和rear
D、 以上都不对

> 答案 : A

121、设`循环队列`的存储空间为 a[0..20], 且当前队头指针 (front指向队首元素) 和队尾指针(rear指向队尾元素)的值分别是是8和3, 则该队列中元素个数为（ ）。
A、 5
B、 6
C、 16
D、 17

> 答案 : C
>
> 解析 : 假设循环队列的队尾指针是rear, 队头是front, 其中 QueueSize 为循环队列的最大长度. 则队列长度为 `(rear - front + QueueSize) % QueueSize`.
>
> 由题目已知 rear = 3, front = 8, QueueSize = 21, 故由公式可得队列中元素得个数为 (3 - 5 + 21) % 21 = 16.

122、(重复题目)假设用一个不带头结点的单链表表示队列, 队头在链表的（ ）位置。
A、 表头
B、 表尾
C、 表中
D、 以上都可以

> 答案 : A

123、与顺序队列相比, 链队（ ）。
A、 优点是可以实现无限长队列
B、 优点是进队和出队时间性能较好
C、 缺点是不能进行顺序访问
D、 缺点是不能根据队首和队尾指针计算队的长度

> 答案 : D
>
> 解析 : 链队也可以顺序存储, 但无法根据公式`(rear - front + QueueSize) % QueueSize`计算队的长度.

124、若用一个大小为 6 的数组实现`循环队列`, 且当前 rear 和 front 的值分别是0, 3, 当从队列中删除一个元素, 再加上两个元素后, rear和front的值分别是（ ）
A、 1，5
B、 2，4
C、 4，2
D、 5，1

> 答案 : B
>
> 解析 :
>
> 方法一 : 题中队列的初始状态为 0(`rear`), 1, 2, 3(`front`), 4, 5
>
> 1. 删除一个元素, front后移, 变成了 4.
> 2. 增加两个元素, 从rear后移, 变成了 2.
> 3. 最终队列的状态为 0, 1, 2(`rear`), 3, 4(`font`), 5
>
> 方法二 : 假设循环队列的队尾指针是rear, 队头是front, 其中QueueSize为循环队列的最大长度, 则 :
>
> - 入队时队尾指针前进1 : `(rear + 1) % QueueSize`
> - 出队时队头指针前进1 : `(front + 1) % QueueSize`
> - 队列长度 : `(rear - front + QueueSize) % QueueSize`
>
> 进而可得删除一个元素时 front = (3 + 1) % 6 = 4, 增加两个元素时 rear = (0 + 2) % 6 = 2.

125、通常设置`循环队列`sq 的队空条件(front队首指针指向队首元素, rear队尾指针指向队尾元素的下一个元素)是（ ）。
A、 (sq.rear+1)%M==(sq.front+1)%M
B、 (sq.rear+1)%M==sq.front+1
C、 (sq.rear+1)%M==sq.front
D、 sq.rear==sq.front

> 答案 : D
>
> 解析 : 即当循环队列 sq 的头尾指针都指向自己时(`sq.rear == sq.front`), 则表示队空.

串

126、串下面关于串的的叙述中, ( )是不正确的。
A、 串是字符的有限序列
B、 空串是由空格构成的串
C、 模式匹配是串的一种重要运算
D、 串既可以采用顺序存储, 也可以采用链式存储

> 答案 : B
> 解析 : 空格常常是串的字符集合中的一个元素, 有一个或多个空格组成的串成为空格串, 零个字符的串称为空串, 其长度为零.

127、已知串S = ‘aaab’, 其 Next 数组值为( )
A、 0123
B、 1123
C、 1231
D、 1211

> 答案 : A
> 解析 :
> a a a b
> -1 0 1 2
> 每个 next 值 + 1
> a a a b
> 0 1 2 3

128、串“ababaaababaa”的next数组为( )。
A、 012345678999
B、 012121111212
C、 011234223456
D、 0123012322345

> 答案 : C
> 解析 : next数组
> a b a b a a a b a b a a
> -1 0 0 1 2 3 1 1 2 3 4 5
> 每个 next 值 + 1
> a b a b a a a b a b a a
> 0 1 1 2 3 4 2 2 3 4 5 6

130、串的长度是指( )。
A、 串中所含不同字母的个数
B、 串中所含字符的个数
C、 串中所含不同字符的个数
D、 串中所含非空格字符的个数

> 答案 : B
> 解析 : 串中字符的数目称为串的长度.

131、若串S = ‘software’, 其子串的数目是( )。
A、 8
B、 37
C、 36
D、 9

> 答案 : B
> 解析 : 子串的定义是为 : 串中任意个连续的字符组成的子序列, 并规定空串是任意串的子串, 任意串是其自身的子串.
>
> 字符串的子串, 就是字符串中的某一个连续片段. 截取一个字符串长度需要一个起始位置和结束位置. 字符串 “software” 有8个字符, 可是设置间隔的位置有9个, 使用`C(9,2) = 36`(其中 2 代表起始位置和结束位置哈)即可求得字符串 “software” 的所有子串. 因空串也是子串, 故还需要加上1, 总共 37 个子串.
>
> - 子串(包括空串)为 : `n * (n + 1) / 2 + 1`
> - 非空真子子串(不包括空串和跟自己一样的子串)为 : `n *（n + 1）/ 2 - 1`
> - 含有 n 个不同字符的字符串的非空子串的个数为 : `C(n + 1, 2) = n * (n + 1) / 2`

132、设有两个串 p 和 q, 其中 q 是 p 的子串, 求 q 在 p 中首次出现的位置的算法称为( )。
A、 求子串
B、 连接
C、 模式匹配
D、 求串长

> 答案 : C
> 解析 :
>
> - `求子串` : 在一个主字符串中按一定的规则取出任意个连续的字符得到一个新串.
> - `匹配` : 求一个字符串是另一个字符串的子串, 并返回子串首次出现的位置.
> - `连接` : 把一个字符串的内容连接到另一个足够大的字符串的末尾.
> - `求串长` : 统计字符串中有效字符的个数.

134、串与普通的线性表相比较, 它的特殊性体现在( )。
A、 顺序的存储结构
B、 链接的存储结构
C、 数据元素是一个字符
D、 数据元素可以任意

> 答案 : C
>
> 解析 : 串又称为字符串, 是一种特殊的线性表, 其特殊性体现在数据元素是一个字符, 即串是一种内容受限的线性表, 而栈和队列则是操作受限的线性表.

135、串的长度是指( )。
A、 串中所含不同字母的个数
B、 串中所含字符的个数
C、 串中所含不同字符的个数
D、 串中所含非空格字符的个数

> 答案 : B
>
> 解析 :

> A / C : 忽视了即使种类数相同的字母 / 字符也会被多次记录到串长中.
>
> D : 空格也会被算入到串长当中去.

138、(重复题目)串下面关于串的的叙述中, ( )是不正确的。
A、 串是字符的有限序列
B、 空串是由空格构成的串
C、 模式匹配是串的一种重要运算
D、 串既可以采用顺序存储, 也可以采用链式存储

> 答案 : B
>
> 解析 : 空格常常是串的字符集合中的一个元素, 有一个或多个空格组成的串成为空格串, 零个字符的串成为空串, 其长度为零.

144、(重复题目)设有两个串 p 和 q, 其中 q 是 p 的子串, 求 q 在 p 中首次出现的位置的算法称为( )。
A、 求子串
B、 联接
C、 模式匹配
D、 求串长

> 答案 : C

145、下面关于串的叙述中, 正确的是
A、 串是一种特殊的线性表
B、 串中元素只能是字母
C、 空串就是空白串
D、 串的长度必须大于零

> 答案 : A
>
> 解析 : 串又称为字符串, 是一种特殊的线性表, 其特殊性体现在数据元素是一个字符, 即串是一种内容受限的线性表, 而栈和队列则是操作受限的线性表.

147、(重复题目)若串 str = "SoftWare", 其子串的个数是( )
A、 8
B、 9
C、 36
D、 37

> 答案 : D

150、对于一个链串s, 查找第一个字符值为 x 的算法的时间复杂度为（ ）
A、 O(1)
B、 O(n)
C、 O(n^2)
D、 以上都不对

> 答案 : B
>
> 解析 : 串是一种特殊的线性表, 故查找指定元素的时间复杂度与线性表相同.

151、(`重复题目`)设有两个串 p 和q, 其中 q 是 p 的子串, 则求 q 在 p 中首次出现位置的算法称为（ ）
A、 求子串
B、 串联接
C、 模式匹配
D、 求串长

> 答案 : C

152、在串的 BF 模式匹配中, 当模式串位 j 与目标串位 i 比较时, 两字符不相等, 则 i 的位移方式为（ ）
A、 i++
B、 i=j+1
C、 i=i-j+2
D、 i=j-i+2

> 答案 : C
>
> 解析： 模式匹配中两个经典的算法为 : BF(`Boy Friend ?`) 算法和 KMP(`看毛片 ?`).
>
> 好吧, 其实 BF 的全称为 `Brute Force`, 即暴力破解的意思, 是指用穷举法, 举出所有可能的结果, 然后逐一检验是否正确.
>
> 每当出现不匹配的现象时, 简单模式匹配算法的做法是 : 一律将 i 赋值为 `i - j + 2`, j 赋值为 `1`, 然后重新开始比较.

树

154、把一棵树转换为二叉树后,这棵二叉树的`形态`是( )。
A、 唯一的
B、 有多种
C、 有多种,但根结点都没有左孩子
D、 有多种,但根结点都没有右孩子

> 答案 : A
> 解析 : 因为二叉树有左孩子、右孩子之分, 故一棵树转换为二叉树后, 这棵二叉树的形态是唯一的, 且根结点没有右孩子.

155、由 3 个结点可以构造出多少种不同的二叉树?( )
A、 2
B、 3
C、 4
D、 5

> 答案 : D
>
> 解析 : 根据 Catalan(卡特兰数) 可得 n 个节点的二叉树有 `C(n, 2n) / (n + 1)` 种形态.

156、一棵`完全二叉树`上有 1000 个结点, 其中叶子结点的个数是( )。
A、 250
B、 500
C、 254
D、 501

> 答案 : B
>
> 解析 :
>
> 方法一 : 设度为 0 结点(叶子结点)个数为 `n0`, 度为 1 的结点个数为 `n1`, 度为 2 的结点个数为 `n2`,
>
> 由 `n0 = n2 + 1`, `n0 + n1 + n2 = 1000` 可得 `n0 + n0 - 1 + n1 = 1000`, 即 `2 * n0 + n1 - 1 = 1000`, 再由完全二叉树的性质可得 n1 = 0 或 1, 又因为 n0 为整数, 所以当 n1 = 1, n0 = 500.
>
> 方法二 : 由题得完全二叉树的最后一个结点的编号一定是1000, 进而它的父结点的编号为 1000 / 2 = 500, 叶子结点个数为 1000 - 500 = 500. 即若完全二叉树的最后一个结点的编号是n, 则它的父结点的编号为`[n / 2]`, 叶子结点个数为`n - [n / 2]`.

157、一个具有 1024 个结点的二叉树的高 h 为( )。
A、 11
B、 10
C、 11至1025之间
D、 11至1024之间

> 答案 : D
> 解析 : 若每层仅有一个结点, 则树高 h 为1024. 而最小树高为 $\log_2 1024 + 1 = 11$, 即 h 在 11 ~ 1024 之间.

158、满 m 叉树, 其深度为 h, 则该树第 i 层有( )个结点。(1=<i<=h)
A、 m^(i-1)
B、 m^i-1
C、 m^(h-1)
D、 m^h-1

> 答案 : A
> 解析 : 深度为 h 的满 m 叉树共有 `mh - 1` 个结点, 第 i 层有 `m ^ (i - 1)` 个结点, 可以和二叉树性质联系起来帮助理解.

159、利用二叉链表存储树,则根结点的右指针是( )。
A、 指向最左孩子
B、 指向最右孩子
C、 空
D、 非空

> 答案 : C
> 解析 : 利用二叉链表存储树时, 右指针指向兄弟结点, 因为根结点没有兄弟结点, 故根结点的右指针指向空。

160、对二叉树的结点从 1 开始进行连续编号, 要求每个结点的编号大于其左、右孩子的编号, 同一结点的左右孩子中, 其左孩子的编号小于其右孩子的编号, 可采用( )遍历实现编号。

A、 先序

B、 中序

C、 后序

D、 从根开始按层次遍历

> 答案 : C
> 解析 : 根据题意可知按照先左孩子、再右孩子、最后双亲结点的顺序遍历二叉树, 即后序遍历二叉树.

161、若二叉树采用二叉链表存储结构, 要交换其所有分支结点左、右子树的位置, 利用( )遍历方法最合适。

A、 前序

B、 中序

C、 后序

D、 按层次

> 答案 : C
> 解析 : 后续遍历和层次遍历均可实现左右子树的交换, 不过层次遍历的实现消耗比后续大, 后序遍历方法最合适.

162、在下列存储形式中, ( )不是树的存储形式?

A、 双亲表示法

B、 孩子链表表示法

C、 孩子兄弟表示法

D、 顺序存储表示法

> 答案 : D
> 解析 : 树的存储结构有三种, 即 `双亲表示法`、`孩子表示法`、`孩子兄弟表示法`, 其中孩子兄弟表示法是常用的表示法, 任意一棵树都能通过孩子兄弟表示法转换为二叉树进行存储.

163、一棵非空的二叉树的先序遍历序列与后序遍历序列正好相反, 则该二叉树一定满足( )。

A、 所有的结点均无左孩子

B、 所有的结点均无右孩子

C、 只有一个叶子结点

D、 是任意一棵二叉树

> 答案 : C
> 解析 : 因为先序遍历次序的是 “`根左右`”, 后序遍历的次序是 “`左右根`”, 所以当没有左子树时, 遍历次序就是 “根右” 和 “右根” ; 当没有右子树时, 遍历次序就是 “左根” 和“ 根左”.
>
> 进而可得所有的结点均无左孩子或右孩子均可, 故A、B错误. 又所有的结点均无左孩子或右孩子时, 均只有一个叶子结点, 故选C.

168、树最适合用来表示（　）

A、 有序数据元素

B、 元素之间具有层次关系的数据

C、 无序数据元素

D、 元素之间无联系的数据

> 答案 : B

169、一棵结点个数为n、高度为 h 的m(m>=3)次树中, 其分支数为（ ）。
A、 nh
B、 n-1
C、 n+h
D、 h-1

> 答案 : B
>
> 解析 : `n - 1` 中的 1 指的是 root 节点哈.

170、若一棵 3 叉树中有 2 个度为 3 个结点, 1个度为 2 的结点, 2个度为 1 的结点, 该树一共有（ ）个结点。
A、 10
B、 11
C、 8
D、 5

> 答案 : B
> 解析 : 由题已知 n3 = 2, n2 = 1, n1 = 2, 故总分支数 = 1*n1* + *2*n2 + 3*n3 = 10, 总节点数 = 总分支数 + 1(`根节点`) = 10 + 1 = 11.

171、设树 T 的度为4, 其中度为1、2、3、4的结点个数分别是4、2、1、1, 则 T 中的叶子结点个数是（ ）。
A、 7
B、 6
C、 5
D、 8

> 答案 : D
> 解析 : 由题已知 : n1 = 4, n2 = 2, n3 = 1, n4 = 1, 分支数 = n1 + 2n2 + 3n3 + 4n4 = 15, 总节点数 = 总分支数 + 1 = 15 + 1 = 16. 进而叶子节点 n0 = n - n1 - n2 - n3 - n4 = 16 - (4 + 2 + 1 + 1) = 8.

172、假设每个结点值为单个字符, 而一棵树的后根遍历序列为ABCDEFGHIJ, 则其根结点值是（ ）。
A、 J
B、 以上都不对
C、 B
D、 A

> 答案 : A
>
> 解析 : 树的后序遍历次序为 `左 -> 右 -> 根`, 故遍历结果的尾元素为原树的根节点.

173、一棵度为5, 结点个数为 n 的树采用`孩子链表存储结构`时, 其中空指针域的个数是（ ）
A、 4n
B、 5n
C、 4n-1
D、 4n+1

> 答案 : D
> 解析 : 总指针数 = 5 * n, 非空指针域个数 = 分支数 = n - 1, 故空指针域的个数 = 5n - (n - 1) = 4n + 1.

174、有一棵三叉树, 其中n3 = 2, n2 = 2, n1 = 1, 该树采用`孩子兄弟链表`存储结构时, 则总的指针域数为（ ）
A、 16
B、 24
C、 36
D、 10

> 答案 : B
> 解析 : 分支数 = n - 1 = n1 + 2n2 + 3n3 = 11, 故总结点数 n = 12, 又每个结点有两个指针域, 所以总的指针域数为n *2 = 24.

177、判断线索二叉树中 p 结点为叶子结点的条件是（ ）
A、 p->ltag==0&&p->ltag==0
B、 p->ltag==1
C、 p->lchild==NULL&&p->rchild==NULL
D、 p->ltag==1&&p->ltag==1

> 答案 : D
>
> 线索二叉树的结点结构如下所示 :

lchild	ltag	data	rtag	rchild

> 在二叉树线索化的过程中会把树中的空指针利用起来作为寻找当前节点前驱或后继的线索.
>
> 1. 若 `ltag = 0` 则表示 lchild 为指针, 指向节点的左孩子; 若 `ltag = 1` 则表示 lchild 为线索, 指向节点的直接前驱.
> 2. 若 `rtag = 0` 则表示 rchild 为指针, 指向节点的右孩子; 若 `rtag = 1` 则表示 rchild 为线索, 指向节点的直接后继.
>
> 因叶子节点无左右孩子, 所以其节点的 ltag 与 rtag 的值都为 1, 即分别指向该节点的直接前驱与直接后继.

178、n个结点的线索二叉树上含有的线索个数为（ ）
A、 2n
B、 n+1
C、 n
D、 n-1

> 答案 : B
>
> 解析 : 无论是什么样的二叉链表, 空线索数均由度为 1 和度为 0 的节点所提供, 其中一个度为 1 的节点提供 1 个空线索数, 一个度为 0 的节点提供 2 个空线索数, 即总的空线索数为: `2*n0 + n1`, 又因 `n0 = n2 + 1`, 所以 n 个结点的线索二叉树上含有的线索个数为 :
>
> 1. 2 * n0 + n1
>
> | |

2. 2 * (n2 + 1) + n1

   | |

3. 2n2 + 2 + n1

   | |

4. (n0 - 1) + n2 + 2 + n1

   | |

5. n0 + n1 + n2 + 1

   | |

6. n + 1

图

185、具有 n 个顶点的`有向图`最多有( )条边。
A、 n
B、 n(n-1)
C、 n(n+1)
D、 n2

> 答案 : B
> 解析 : 简单图是没有平行边且没有自环的图.
>
> 没有平行边且没有自环, 但是两个顶点可以互指, 所以最多的情况为这个图中任意两个顶点两两相连, 且都是互相指向. 进而最多有 $2C_n^2 = n(n-1)$ 条边.

186、n个顶点的`连通图`用邻接距阵表示时, 该距阵至少有( )个非零元素.
A、 n
B、 2(n-1)
C、 n/2
D、 n2

> 答案 : B
>
> 解析 : 连通图一定是无向图, 有向的叫做强连通图. `n`个顶点的无向连通图至少有 `n - 1` 条边.
>
> 由于无向图的每条边同时关联两个顶点, 故邻接矩阵中每条边被存储了两次(对称矩阵), 进而至少有 `2(n - 1)` 个非零元素.

187、G是一个`非连通无向图`, 共有 28 条边, 则该图至少有( )个顶点。
A、 7
B、 8
C、 9
D、 10

> 答案 : C
> 解析 : n个顶点的无向图(完全连通图)最多有 `n(n - 1) / 2` 条边, 再添加一个点(n + 1)即构成非连通无向图.

> 故由题列得 n(n - 1) / 2 = 28, 解得 n = 8, 进而至少有 n + 1 = 9 个顶点.

188、若从无向图的任意一个顶点出发进行一次`深度优先搜索`可以访问图中所有的顶点, 则该图一定是( )图。
A、 非连通
B、 连通
C、 强连通
D、 有向

> 答案 : B
> 解析 : 若从无向图的任意一个顶点出发, 都有到各个顶点的路径, 则称该无向图是连通图.

189、下面( )算法适合构造一个稠密图G的最小生成树。
A、 Prim算法
B、 Kruskal算法
C、 Floyd算法
D、 Dijkstra算法

> 答案 : A
> 解析 :
>
> - `Prim算法` : 适合构造一个稠密图 G 的最小生成树
> - `Kruskal算法` : 适合构造一个稀疏图 G 的最小生成树
> - `Floyd算法` : 是解决任意两点间的最短路径的一种算法
> - `Dijkstra算法` : 典型最短路径算法, 用于计算一个节点到其他各个节点的最短路径

190、用邻接表表示图进行广度优先遍历时, 通常借助( )来实现算法。
A、 栈
B、 队列
C、 树
D、 图

> 答案 : B
> 解析 : 广度优先遍历(`BFS`)通常借助队列来实现算法, 而深度优先遍历(`DFS`)通常借助栈来实现算法.

191、用邻接表表示图进行深度优先遍历时, 通常借助( )来实现算法。
A、 栈
B、 队列
C、 树
D、 图

> 答案 : A

195、已知图的邻接矩阵如图 6.30 所示, 则从顶点 v0 出发按深度优先遍历的结果是( )。

图6.30 邻接矩阵
A、 0 2 4 3 1 5 6
B、 0 1 3 6 5 4 2
C、 0 1 3 4 2 5 6

D、 0 3 6 1 5 4 2

> 答案 : C

196、已知图的邻接表如图6.31所示, 则从顶点 v0 出发按广度优先遍历的结果是( ), 按深度优先遍历的结果是( )。

图6.31 邻接表
A、 0 1 3 2
B、 0 2 3 1
C、 0 3 2 1
D、 0 1 2 3

> 答案 : D

197、下面( )方法可以判断出一个有向图是否有环。
A、 深度优先遍历
B、 拓扑排序
C、 求最短路径
D、 求关键路径

> 答案 : B (答案有误，应为AB)
> 解析 : 如果存在回路, 则必存在一个子图, 是一个环路, 且环路中所有顶点的度 >= 2.
>
> 有向图 是否有环的判定算法主要有 深度优先 和 拓扑排序 两种.

204(重复题目)、下面( )算法适合构造一个稠密图G的最小生成树。
A、 Prim算法
B、 Kruskal算法
C、 Floyd算法
D、 Dijkstra算法

> 答案 : A
> 解析 : Prim算法适合构造一个稠密图G的最小生成树.

209、图的 BFS 生成树的树高比 DFS 生成树的树高( )。
A、 小
B、 相等
C、 小或相等
D、 大或相等

> 答案 : C
> 解析 : 树高可以理解为顶点到结点的距离, 对于 BFS 生成树, 每个结点到根结点都是最短距离, 而 DFS 没有这个限制, 因此一般情况下 DFS 生成树的树高 > BFS 生成树的树高. 而对于一些特殊的图, 比如只有一个顶点的图, 其 BFS 生成树的树高和 DFS 生成树的树高相等.

查找

215、对n个元素的表做`顺序查找`时, 若查找每个元素的概率相同, 则平均查找长度为( )。

A、 (n-1)/2
B、 n/2
C、 (n+1)/2
D、 n

> 答案 : C
> 解析 : 第一个数的比较次数为1, 第二个数的比较次数为2, 以此类推第 n 个数的比较次数为n, 所以总的比较次数为等差数列求和(首项1, 末项n) : 1 + 2 + ... + n = n(n + 1) / 2, 故平均比较次数为 [n(n + 1) / 2] / n, 进而平均查找长度为 `(n + 1) / 2`.

216、适用于折半查找的表的存储方式及元素排列要求为( )。

A、 链接方式存储,元素无序
B、 链接方式存储,元素有序
C、 顺序方式存储,元素无序
D、 顺序方式存储,元素有序

> 答案 : D
> 解析 : 折半查找也称二分查找(Binary Search), 要求线性表必须采用顺序存储结构, 而且表中元素按关键字有序排列.

217、如果要求一个线性表既能较快的查找, 又能适应`动态变化`的要求, 最好采用( )查找法。

A、 顺序查找
B、 折半查找
C、 分块查找
D、 哈希查找

> 答案 : C
> 解析 : 分块查找也叫索引查找, 是折半查找和顺序查找的一种改进方法, 其的优点是 : "块间有序, 块内无序", 因此适合做频繁的插入删除.

218、折半查找有序表(4,6,10,12,20,30,50,70,88,100)。若查找表中元素58, 则它将依次与表中( )比较大小, 查找结果是失败。

A、 20,70,30,50
B、 30,88,70,50
C、 20,50
D、 30,88,50

> 答案 : A
> 解析 : 二分查找中确定中间值的代码是 `middle = (low + high) / 2`, 判别查找失败的代码为 `high < low`.
>
> 1. 第一轮两端点为[0, 9], mid为4, 58 > 20, 区间右移
> 2. 第二轮两端点为[5, 9], mid为7, 58 < 70, 区间左移
> 3. 第三轮两端点为[5, 6], mid为5, 58 > 30, 区间右移
> 4. 第四轮两端点为[6, 6], mid为6, 58 > 50, 区间右移
> 5. 此时high(6) < low(7), 查找失败, 跳出循环

219、对 22 个记录的有序表作折半查找, 当查找失败时, 至少需要比较( )次关键字。
A、 3
B、 4
C、 5
D、 6

> 答案 : B
> 解析 : 22个记录的有序表, 其折半查找的判定树深度为 $\log_2 22 + 1 = 5$, 且该判定树不是满二叉树(除最后一层无任何子节点外, 每一层上的所有结点都有两个子结点的二叉树), 即查找失败时至多比较 5 次, 至少比较 4 次.

220、折半搜索与二叉排序树的时间性能( )。
A、 相同
B、 完全不同
C、 有时不相同
D、 数量级都是O(log2n)

> 答案 : C
>
> 解析 : 不一定相同.
>
> - `折半查找` : 必须要求记录有序, 采用顺序存储, 利用这个特点, 折半查找的效率也比顺序查找高, 其时间复杂度为`O(logN)`.
> - `二叉查找树` : 若它的左子树不为空, 则左子树上所有节点的值均小于根节点. 若它的右子树不为空, 则右子树上所有节点的值均大于根节点. 它的左右子树都是二叉查找树.
>
> 二叉排序树不一定是平衡树, 因为它是只要求了左右子树与根结点存在大小关系, 但是对左右子树之间没有层次差异的约束, 进而通过二叉排序树进行查找不一定能够满足`O(logN)`, 例如一棵只有多层左子树的而叉排序树. 只有是一棵平衡的二叉排序树时, 其查找时间性能才和折半查找类似.

221、分别以下列序列构造二叉排序树, 与用其它三个序列所构造的结果不同的是( )。
A、 (100,80, 90, 60, 120,110,130)
B、 (100,120,110,130,80, 60, 90)
C、 (100,60, 80, 90, 120,110,130)
D、 (100,80, 60, 90, 120,130,110)

> 答案 : C
> 解析 : A、B、C、D选项中构造的二叉排序树都以 100 为根, 又知A、B、D三个序列中第一个比 100 小的关键字为80, 即 100 的左孩子为80, 而 C 选项中100的左孩子为60, 故选C.
>
> 二叉排序树的定义 : 又称为二叉查找树. 二叉排序树或者是一棵空树, 或者是具有以下性质的二叉树 :
>
> - 若其左子树不为空, 则左子树上的所有节点的值均小于它的根结点的值
> - 若其右子树不为空, 则右子树上的所有节点的值均大于它的根结点的值
> - 左右子树又分别是二叉排序树

13、采用排序算法对 n 个元素进行排序, 其排序趟数肯定为 n-1 趟的排序方法是（ ）。
A、简单选择和直接插入
B、直接插入和快速
C、简单选择和冒泡
D、冒泡和快速

> 正确答案 : A
> 解析 :
>
> - 冒泡排序为 `1 ～ n-1` 趟
> - 快速排序为 `log2n ～ n-1` 趟
> - 简单选择和直接插入为 `n - 1` 趟

11、单选快速排序在（ ）情况下最不利于发挥其长处。
A、排序的数据个数为奇数
B、排序的数据中含有多个相同值
C、排序的数据量太大
D、排序的数据已基本有序

> 答案 : D
> 解析 : 平均情况下快速排序的时间复杂度是`O(nlogn)`, 空间复杂度是`O(logn)`.
>
> 若排序的数据已基本有序, 快速排序的时间复杂度则为`O(n²)`.

8、单选冒泡排序最少元素移动的次数是（ ）。
A、1
B、0
C、3n(n-1)/2
D、n

> 答案 : B
> 解析 : 数据有序的情况下冒泡排序无需移动元素.

9、对有 n 个记录的表进行`直接插入`排序, 在最好情况下需比较（ ）次关键字。
A、n(n-1)/2
B、n+1
C、n/2
D、n-1

> 正确答案 : D
> 解析 : 直接插入排序在初始数据为正序时效率最好, 此时仅需要进行 `n - 1` 次关键字比较即可.

10、以下关于排序的叙述中正确的是（ ）。
A、对同一个顺序表使用不同的排序方法进行排序，得到的排序结果可能不同
B、排序方法都是在顺序表上实现的，在链表上无法实现排序方法
C、在顺序表上实现的排序方法在链表上也同样适合
D、稳定的排序方法优于不稳定的排序方法，因为稳定的排序方法效率较高

> 答案 : A
> 解析 :
>
> A : 由于排序方法具有不同的稳定性, 所以对同一个顺序表(存在相同的多个关键字记录)使用不同的排序方法进行排序, 得到的排序结果可能不同.
>
> B / C : 有些排序方法既可以上顺序表上实现，也可以在链表上实现，但不是所有的排序方法都如此。
>
> D : 稳定的排序方法的效率不一定都比不稳定的排序方法高。

判断题

1. 线性表的逻辑顺序总是与其物理顺序一致。（ ）

> 答案 : 错
>
> 解析 : 线性表有两种存储方式, 即顺序与链式存储, 只有用顺序表时, 逻辑顺序才和物理顺序一致.

2. 线性表的顺序存储优于链式存储。（ ）

> 答案 : 错
>
> 解析 : 不一定哟, 各有所长哈.

3. 在长度为 n 的顺序表中, 求第 i 个元素的直接前驱算法的时间复杂度为0(1)。（ ）

> 答案 : 对
>
> 解析 : 假设顺序表L, 长度为n, 则第 i 个节点为L[i], 直接前驱为L[i-1], 故为O(1).

4. 若一棵二叉树中的结点均无右孩子, 则该二叉树的中根遍历和后根遍历序列正好相反。（ ）

> 答案 : 错
>
> 解析 : 均无`左`孩子的情况下刚好相反哈.

5. 顺序表和一维数组一样, 都可以按下标随机(或直接)访问。（ ）

> 答案 : 对
>
> 扩 : 顺序表支持随机访问, 但单链表不行哒.

6. 任何一棵二叉树的叶结点在三种遍历中的相对次序是不变的。（ ）

> 答案 : 对
>
> 解析 : 因为根据三个遍历的次序和特点为 : 前序是`根左右`、中序是`左根右`、后序是`左右根`, 因此相对次序发生变化的都是子树的根, 即分支结点.

7. 对平衡二叉树进行中根遍历, 可得到结点的有序排列。（ ）

> 答案 : 对
>
> 解析 : 平衡二叉树的特点为 :
>
> 1. 非叶子节点最多拥有两个子节点
> 2. `非叶子节值大于左边子节点、小于右边子节点` (故中序遍历为有序排列)
> 3. 树的左右两边的层级数相差不会大于1

> 4. 没有值相等重复的节点

8. 在一棵二叉树中，假定每个结点只有左子女, 没有右子女, 对它分别进行前序遍历和中根遍历, 则具有相
同的结果。( )

> 答案 : 错
>
> 解析 : 若每个节点只有右子女, 则前序与中序遍历的结果刚好相反.

9. 拓扑排序是指结点的值是有序排序的。（ ）

> 答案 : 错

10. 在散列法中采取开散列(链地址)法来解决冲突时, 其装载因子的取值一定在(0,1)之间。( )

> 答案 : 错
>
> 解析 : `装载因子 = 关键字个数 / 表长`

11. 图的深度优先搜索是一种典型的回溯搜索的例子, 可以通过递归算法求解。( )

> 答案 : 对

12. 对二叉排序树进行中根遍历，可得到结点的有序排列。（ ）

> 答案 : 对
>
> 解析 : 二叉排序树又称为二叉查找树, 它要么是一棵空树, 要么是具有下列性质的二叉树 :
>
> 1. 若它的左子树不为空, 则左子树上所有结点的值均小于它的根结点的值
> 2. 若它的右子树不为空, 则右子树上所有结点的值均大于它的根结点的值
> 3. 它的左右子树也分别为二叉排序树

13. (`重复题目`)任何一棵二叉树的叶结点在三种遍历中的相对次序是不变的。（ ）

> 答案 : 对

14. 边数很少的稀疏图，适宜用邻接矩阵表示。（ ）

> 答案 : 错
>
> 解析 :
>
> - `稀疏图`用`邻接表`存储
> - `稠密图`用`邻接矩阵`存储
>
> 邻接表只存储非零节点, 而邻接矩阵则要把所有的节点信息(非零节点与零节点)都存储下来.
>
> 稀疏图的非零节点不多, 所以选用邻接表效率高, 如果选用邻接矩阵则效率很低, 因为矩阵中大多都会是零节点. 而稠密图的非零界点多, 零节点少, 故适合使用邻接矩阵存储.

15. 二叉树是一棵无序树。( )

> 答案 : 错

16. 对于一棵具有 n 个结点, 其高度为 h 的二叉树, 进行任一种次序遍历的时间复杂度为O(n)。( )

> 答案 : 对

17. 当待排序序列初始有序时, 快速排序的时间复杂性为O(n)。（ ）

> 答案 : 错
> 解析 : 平均情况下快速排序的时间复杂度是`O(nlogn)`, 空间复杂度是`O(logn)`.
> 若排序的数据已基本有序, 则其时间复杂度则为`O(n²)`.

18. 顺序表的空间利用率高于链表。（ ）

> 答案 : 对
> 解析 : 毕竟链表中的每个节点都需要花费格外的空间来存储指针.

19. 采用不同的遍历方法, 所得到的无向图的生成树是不同的。（ ）

> 答案 : 对

20. 单链表可以实现随机存取。（ ）

> 答案 : 错
> 扩 : 顺序表与数组则支持随机访问哈.

21. 理想情况下哈希查找的等概率查找成功的平均查找长度是O(1)。( )

> 答案 : 对

22. 边数很少的稀疏图, 适宜用邻接表表示。（ ）

> 答案 : 对
> 解析 : `稀疏图`适合用`邻接表`存储, 而`稠密图`适合用`邻接矩阵`存储.

23. 强连通分量是有向图中的极大强连通子图。( )

> 答案 : 对

24. 顺序查找法适用于存储结构为顺序或链接存储的线性表。（ ）

> 答案 : 对

25. 若让元素1, 2, 3依次进栈, 则出栈次序1,3,2是不可能出现的情况。（ ）

> 答案 : 错
>
> 解析 : Stack 的特性为 FILO / LIFO. 故出栈次序 1, 3, 2 的步骤如下 :
>
> 1. 1 先进栈, 然后立即弹出
> 2. 2 进栈
> 3. 3 进栈, 然后立即弹出
> 4. 最后弹出栈中唯一的元素 2

26. 对任何用顶点表示活动的网络(AOV网)进行拓扑排序的结果都是唯一的。( )

> 答案 : 错

27. 邻接矩阵适用于稠密图(边数接近于顶点数的平方), 邻接表适用于稀疏图(边数远小于顶点数的平方)。( )

> 答案 : 对
>
> 解析 : 邻接表只存储非零节点, 而邻接矩阵则要把所有的节点信息(非零节点与零节点)都存储下来.
>
> 稀疏图的非零节点不多, 所以选用邻接表效率高, 如果选用邻接矩阵则效率很低, 因为矩阵中大多都会是零节点. 而稠密图的非零界点多, 零节点少, 故适合使用邻接矩阵存储.

28. 算法和程序原则上没有区别, 在讨论数据结构时二者是通用的。( )

> 答案 : 错
>
> 解析 : `程序 = 算法 + 数据结构` (好好体会吧, 愿你某天也能成为大牛嘿嘿)

29. 循环链表的结点与单链表的结点结构完全相同, 只是结点间的连接方式不同。（ ）

> 答案 : 对
>
> 解析 : 循环链表中尾节点的 next 指针指向头节点哈.

30. 能够在链接存储的有序表上进行折半查找, 其时间复杂度与在顺序存储的有序表上相同。( )

> 答案 : 错
>
> 解析 : 因链表不支持随机访问, 故查找的时间复杂度为`O(n)`, 而顺序表查找的时间复杂度则为`O(1)`.

31. 一个无向连通图的生成树是图的极大的连通子图。（ ）

> 答案 : 错

32. 对稀疏矩阵进行压缩存储是为了节省存储空间。（ ）

> 答案 : 对

33. 快速排序的时间复杂性不受数据初始状态影响，恒为O(nlog2n)。（ ）

> 答案 : 错
>
> 解析 : 平均情况下快速排序的时间复杂度是`O(nlogn)`, 空间复杂度是`O(logn)`.
>
> 若排序的数据已基本有序, 则其时间复杂度则为`O(n²)`.

34. 只有用面向对象的计算机语言才能描述数据结构算法。（ ）

> 答案 : 错
>
> 解析 : 伪代码和自然语言也可以哈.

35. 如果无向图中每个顶点的度都大于等于2，则该图中必有回路。( )

> 答案 : 对

36. 邻接表只能用于有向图的存储, 邻接矩阵对于有向图和无向图的存储都适用。( )

> 答案 : 错

37. 折半查找所对应的判定树, 既是一棵二叉查找树, 又是一棵理想平衡二叉树。( )

> 答案 : 对

38. 存储无向图的邻接矩阵是对称的, 因此可以只存储邻接矩阵的下(上)三角部分。( )

> 答案 : 对

39. (`重复题目`)对稀疏矩阵进行压缩存储是为了便于进行矩阵运算。（ ）

> 答案 : 错

40. 在任意一棵二叉树的前序序列和后序序列中, 各叶子之间的相对次序关系都相同。（ ）

> 答案 : 对
>
> 解析 : 前序序列为 `根左右`, 后序序列为 `左右根`, 其中`左右`的相对次序并未放生改变哈.

41. 采用不同的遍历方法, 所得到的无向图的生成树总是相同的。（ ）

> 答案 : 错
>
> 解析 : 采用不同的遍历方法, 所得到的无向图的生成树是不同的.

42. 对于同一组记录, 生成二叉搜索树的形态与插入记录的次序无关。( )

> 答案 : 错

43. 对一个有向图进行拓扑排序, 一定可以将图的所有顶点按其关键码大小排列到一个拓扑有序的序列中。( )

> 答案 : 错

44. 链式栈与顺序栈相比, 一个明显的优点是通常不会出现栈满的情况。（ ）

> 答案 : 对

45. 边数很少的稀疏图, 适宜用邻接矩阵表示。（ ）

> 答案 : 错
>
> 解析 : `稀疏图`适合用`邻接表`存储, 而`稠密图`适合用`邻接矩阵`存储.

46. 递归的算法简单、易懂、容易编写, 而且执行效率也高。（ ）

> 答案 : 错

47. 链队列的出队操作总是需要修改尾指针。（ ）

> 答案 : 错
>
> 解析 : 链式队列出队时需要修改头指针, 入队时则需要修改尾指针.

48. 栈和队列都是顺序存取的线性表, 但它们对存取位置的限制不同。（ ）

> 答案 : 对
>
> 解析 : 栈在栈顶进行操作元素, 而队列在队头出队, 在队尾入队哈.

49. 数据的逻辑结构是指各数据元素之间的逻辑关系，是用户根据应用需要建立的。

> 答案 : 对

50. 直接选择排序是一种稳定的排序方法。( )

> 答案 : 错
>
> 解析 : 直接选择排序是不稳定的.
>
> 算法稳定性的定义为 : 对于待排序列中相同项的原来次序, 不被算法改变则称该算法稳定.
>
> `注意点` : 用数组实现的选择排序是不稳定的, 而用链表实现的选择排序是稳定的.

51. (重复题目)线性表的逻辑顺序总是与其物理顺序一致。（ ）

> 答案 : 错
>
> 解析 : 线性表有两种存储方式, 即顺序与链式存储, 只有用顺序表时, 逻辑顺序才和物理顺序一致.

52. (重复题目)对平衡二叉树进行中根遍历, 可得到结点的有序序列。（ ）

> 答案 : 对

53. 双向循环链表的结点与单链表的结点结构相同, 只是结点间的连接方式不同。（ ）

> 答案 : 错
>
> 解析 : 结点结构相同也不同, 因为双向循环链表中每个节点具有两个指针, 即指向前一节点的`previous`及指向后一节点的`next`指针, 而单链表仅有`next`指针哈.

54. 在顺序表中, 逻辑上相邻的元素在物理位置上不一定相邻。（ ）

> 答案 : 错
>
> 解析 : 线性表有两种存储方式, 即顺序与链式存储, 只有用顺序表时, 逻辑顺序才和物理顺序一致.

55. 对于一棵具有 n 个结点, 其高度为 h 的任何二叉树, 进行任一种次序遍历的时间复杂度均为O(h)。( )

> 答案 : 错
>
> 解析 : 对于一棵具有 n 个结点, 其高度为 h 的二叉树, 进行任一种次序遍历的时间复杂度为`O(n)`.

56. 当输入序列已经基本有序时, 起泡排序需要比较关键码的次数, 比快速排序还要少。( )

> 答案 : 对

57. (重复题目)顺序查找法适用于存储结构为顺序或链接存储的线性表。( )

> 答案 : 对

58. 插入与删除操作是数据结构中最基本的两种操作, 因此这两种操作在数组中也经常被使用。（ ）

> 答案 : 错
>
> 解析 : 这就好像说学校计算机整体教学水平很高, 因此我编程就很厉害噗嗤？!

59. 在一个有向图中，所有顶点的入度之和等于所有顶点的出度之和。（ ）

> 答案 : 对

60. 在用循环单链表表示的链式队列中, 可以不设队头指针, 仅在链尾设置队尾指针。（ ）

> 答案 : 对
>
> 解析 : 因为循环单链表中尾节点的`next`指针指向的是队头节点, 故出队或入队时仅需移动尾指针即可.

61. 在二叉排序树中插入新结点时, 新结点总是作为叶子结点插入。（ ）

> 答案 : 对

62. (`重复题目`)边数很少的稀疏图, 适宜用邻接表表示。（ ）

> 答案 : 对

63. 链队列的出队操作总是需要修改尾指针。（ ）

> 答案 : 错
>
> 解析 : 链队出队时需要修改头指针.

64. ( ) 链队列的出队操作是不需要修改尾指针的。（ ）

> 答案 : 错
>
> 解析 : 当出队节点为最后一个节点时, 则需要修改头尾指针哈.

65. 图的广度优先搜索算法通常采用非递归算法求解。( )

> 答案 : 对
>
> 解析 : 深度优先搜索借助辅助栈进行递归式遍历, 而广度优先搜索则借助队列进行层次式遍历.

66. 拓扑排序是指结点的值是有序排序的。（ ）

> 答案 : 错
>
> 解析 : 拓扑排序是对结点的先后逻辑关系进行排序, 和节点值没有关系.

67. 数据的逻辑结构与数据元素本身的内容和形式无关。

> 答案 : 对

68. 在树的存储中, 若使每个结点带有指向双亲结点的指针, 这为在算法中寻找双亲结点带来方便。( )

> 答案 : 对

69. 边数很多的稠密图，适宜用邻接表表示。（ ）

> 答案 : 错
>
> 解析 : `稀疏图`适宜用`邻接表`存储, 而`稠密图`适宜用`邻接矩阵`存储.

70. 对一个`连通图`进行一次深度优先搜索可以遍访图中的所有顶点。( )

> 答案 : 对

71. 用字符数组存储长度为 n 的字符串, 数组长度至少为n + 1。（ ）

> 答案 : 对
>
> 解析 : 因为数组的尾部需要添加字符串结束符 `\0`.

72. 线性表若采用链式存储表示时, 其存储结点的地址可连续也可不连续。（ ）

> 答案 : 对

73. (`重复题目`)在二叉排序树中插入新结点时，新结点总是作为叶子结点插入。（ ）

> 答案 : 对

74. 在线索二叉树中每个结点通过线索都可以直接找到它的前驱和后继。（ ）

> 答案 : 错
>
> 解析 : 若线索二叉树中的某一节点有左右孩子节点, 则该节点的 `ltag` 与 `rtag` 的值都为 0, 故无法指向该节点的直接前驱与直接后继.

75. 数据元素是数据的最小单位。

> 答案 : 错
>
> 解析 : 数据元素是数据的基本单位.
>
> 数据项是数据元素不可分割的最小单位, 一个数据元素可由若干个数据项(`Data Item`)组成.

76. (`重复题目`)在长度为 n 的顺序表中, 求第 i 个元素的直接前驱算法的时间复杂度为0（1）。（ ）

> 答案 : 对
>
> 解析 : 假设顺序表L, 长度为n, 则第 i 个节点为L[i], 直接前驱为L[i - 1], 故为O(1).

1．在决定选取何种存储结构时, 一般不考虑各结点的值如何。（）

> 答案 : 对

2．抽象数据类型(ADT)包括定义和实现两方面, 其中定义是独立于实现的, 定义仅给出一个 ADT 的逻辑特性,不必考虑如何在计算机中实现。（ ）

> 答案 : 对

3．线性表采用链式存储结构时, 结点和结点内部的存储空间可以是不连续的。（ ）

> 答案 : 错
>
> 解析 : 线性表中的结点和结点内部的存储空间是连续的.

7．顺序存储方式只能用于存储线性结构。（ ）

> 答案 : 错
>
> 解析 : 顺序存储方式可以存储线性结构与链式结构.

10．线性表就是顺序存储的表。（ ）

> 答案 : 错
>
> 解析 : 线性表有两种存储方式, 即顺序存储与链式存储.

12．循环链表不是线性表。（ ）

> 答案 : 错
>
> 解析 : 线性表包含顺序表与链表, 而链表又包含单链表、单向循环链表、双向循环链表.

13．链表是采用链式存储结构的线性表, 进行插入、删除操作时, 在链表中比在顺序表中效率高。（ ）

> 答案 : 对
>
> 链表相比较顺序表而言, 插入与删除操作仅需修改指针即可, 故空间复杂度为`O(1)`哈. 而顺序表在插入或删除操作后, 还需要移动元素, 故平均空间复杂度为`O(n)`哈.

14．双向链表可随机访问任一结点。（ ）

> 答案 : 错
>
> 解析 : 可随机访问是顺序表的特点哟, 而链表访问其中节点则需要遍历.

16．队列是一种插入和删除操作分别在表的两端进行的线性表, 是一种先进后出的结构。（ ）

> 答案 : 错
>
> 解析 : 队列的特点为 First In First Out / Last In Last Out.

25．二叉树的后序遍历序列中, 任意一个结点均处在其孩子结点的后面。（ ）

> 答案 : 对
>
> 解析 : 后序遍历的次序为 : `左 -> 右 -> 根`.

26．度为 2 的有序树是二叉树。（ ）

> 答案 : 错
>
> 解析 :
>
> 1. `度不同`
>
> 度为 2 的树要求每个节点最多只能有两棵子树, 并且至少有一个节点有两棵子树. 二叉树的要求是度不超过2, 节点最多有两个叉, 可以是 1 或者0.
>
> 在任意一棵二叉树中, 叶子结点总是比度为 2 的结点多一个.
>
> 2. `分支不同`
>
> 度为 2 的树有两个分支, 但分支没有左右之分.
> 一棵二叉树也有两个分支, 但有左右之分, 左右子树的次序不能随意颠倒.
>
> 3. `次序不同`
>
> 度为 2 的树从形式上看与二叉树很相似, 但它的子树是无序的. 而二叉树是有序的, 即在一般树中若某结点只有一个孩子, 就无需区分其左右次序, 而在二叉树中即使是一个孩子也有左右之分.

27．二叉树的前序遍历序列中, 任意一个结点均处在其孩子结点的前面。（ ）

> 答案 : 对
>
> 解析 : 前序遍历的次序为 `根 -> 左 -> 右`.

28．用一维数组存储二叉树时, 总是以前序遍历顺序存储结点。（ ）

> 答案 : 错

29．若已知一棵二叉树的前序遍历序列和后序遍历序列, 则可以恢复该二叉树。（ ）

> 答案 : 错
>
> 解析 : 恢复二叉树两种情况(其中必须要有`中序`)
>
> 1. 已知`前序`和`中序`
> 2. 已知`后序`和`中序`

40．具有 n 个结点的二叉排序树有多种, 其中树高最小的二叉排序树是最佳的。（ ）

> 答案 : 对

42、栈和队列都是限制存取点的线性结构。（ ）

> 答案 : 对
>
> 解析 : 栈在栈顶对元素进行操作, 而队列在队头出队, 在队尾入队.

44、若一个栈的输入序列是1, 2, 3…n, 输出序列的第一个元素是i, 则第 i 个输出元素不确定。（ ）

> 答案 : 对
>
> 解析 : 栈的入栈与出栈的次序是不固定的哈.

46、链队列与循环队列相比, 前者不会发生溢出。（ ）

> 答案 : 对

48、(重复题目)数据元素是数据的最小单位。（ ）

> 答案 : 错

52.数据的逻辑结构与数据元素本身的内容和形式无关。（ ）

> 答案 : 对

53.一个数据结构是由一个逻辑结构和这个逻辑结构上的一个基本运算集构成的整体。（ ）

> 答案 : 对

55.数据的逻辑结构和数据的存储结构是相同的。（ ）

> 答案 : 错
>
> 解析 : 数据的逻辑结构用于描述数据之间的关系, 而数据的存储结构与相应数据在内存中的物理地址有关, 它是数据的逻辑结构在计算机上的具体语言实现.

57.从逻辑关系上讲, 数据结构主要分为线性结构和非线性结构。（ ）

> 答案 : 对

58.数据的存储结构是数据的逻辑结构的存储映像。（ ）

> 答案 : 错
>
> 解析 : 数据的逻辑结构可以独立于存储结构来考虑, 反之不可以.

59.数据的物理结构是指数据在计算机内实际的存储形式。（ ）

> 答案 : 对

78. 数据的逻辑结构是依赖于计算机的。（ ）

> 答案 : 错
>
> 解析 : 数据的逻辑结构用于描述数据之间的关系, 而数据的存储结构与相应数据在内存中的物理地址有关, 它是数据的逻辑结构在计算机上的具体语言实现.

1. 79. 算法是对解题方法和的描述步骤。（ ）

> 答案 : 对

66. 设一棵二叉树的先序序列和后序序列, 则能够唯一确定出该二叉树的形状。（ ）

> 答案 : 错
>
> 解析 : 恢复 / 确定二叉树两种情况(其中必须要有`中序`)
>
> 1. 已知`前序`和`中序`
> 2. 已知`后序`和`中序`

69. 线性表的顺序存储结构比链式存储结构更好。（ ）

> 答案 : 错
>
> 解析 : 各有所长哈, 例如顺序存储支持随机存取. 链式存储较线性存储, 其插入及删除的时间复杂度较低.

70. 中序遍历二叉排序树可以得到一个有序的序列。（ ）

> 答案 : 对
>
> 解析 : 二叉排序树又称为二叉查找树, 它要么是一棵空树, 要么是具有下列性质的二叉树 :
>
> 1. 若它的左子树不为空, 则左子树上所有结点的值均小于它的根结点的值
> 2. 若它的右子树不为空, 则右子树上所有结点的值均大于它的根结点的值
> 3. 它的左右子树也分别为二叉排序树

72.不论是入队列操作还是入栈操作, 在顺序存储结构上都需要考虑“ 溢出 ”情况。（ ）

> 答案 : 对

73. ( )当向二叉排序树中插入一个结点，则该结点一定成为叶子结点。（ ）

> 答案 : 对

96. (`重复题目`)中序遍历一棵二叉排序树可以得到一个有序的序列。( )

> 答案 : 对

97. 入栈操作和入队列操作在链式存储结构上实现时不需要考虑栈溢出的情况。( )

> 答案 : 对
>
> 解析 : 链式存储结构是动态分配空间的, 而在顺序存储则需要考虑栈溢出的情况哈.

应用题

以下算法均使用 `C++` 实现, 代码 `注释详细`、`程序精简`、而且每个程序都是单独 `可运行` 的哟 (●'◡'●)~

> 不会 `C++` 的童鞋可以看下我的这个笔记哈 : 两小时从 C 过渡到 C++

1. 单链表的插入

```
/**
 * @Author: GoogTech
 * @Email: googtech@qq.com
 * @Website: https://algorithm.show
 * @Date: 2021,6,21
 *
 * 1.单链表的插入
 *  (1).从头节点进行插入
 *  (2).从尾节点进行插入
 *  (3).从指向节点前进行插入
 *  (4).从指定节点后进行插入
 */
#include "iostream"

using namespace std;

class List {
public:
    List() { createList(); }

    // 创建头节点
    void createList();

    // 从头部插入
    void insert(const int &d);

    // 从尾部插入
    void insertTail(const int &d);

    // 在指定位置之后插入
    void insertPosAfter(const int &d, const int &d1);

    // 在指定位置之前插入
    void insertPosBefore(const int &d, const int &d1);

    // 打印
    void print();

private:
    // 定义节点结构
    struct Node {
        int data;
        Node *next;

        Node(const int &d) : data(d), next(NULL) {}
```

```
    };

    // 声明链表头节点
    Node *head;

    // 查找 d 的节点位置
    Node *find(const int &d) {
        Node *p = head;
        while (p != NULL) {
            if (p->data == d) {
                break;
            }
            p = p->next;
        }
        return p;
    }

    // 查找 d 前一节点的位置
    Node *findBefore(const int &d) {
        Node *p = head, *preNode = NULL;
        while (p != NULL) {
            if (p->data == d) {
                break;
            }
            preNode = p;
            p = p->next;
        }
        return preNode;
    }

    // 查找尾部节点
    Node *findTail() {
        Node *p = head;
        while (p->next != NULL) {
            p = p->next;
        }
        return p;
    }
};

// 创建头节点
void List::createList() {
    head = new Node(0);
}

// 从头插入一个节点
void List::insert(const int &d) {
    Node *p = new Node(d);
    p->next = head->next;
    head->next = p;
}

// 从尾部插入一个节点
void List::insertTail(const int &d) {
    Node *p = new Node(d);
    Node *q = findTail();
    q->next = p;
}
```

```
// 在 d 位置之后插入 d1
void List::insertPosAfter(const int &d, const int &d1) {
    Node *p = find(d);
    Node *q = new Node(d1);
    q->next = p->next;
    p->next = q;
}

// 在 d 位置之前插入 d1
void List::insertPosBefore(const int &d, const int &d1) {
    Node *p = findBefore(d);
    Node *q = new Node(d1);
    q->next = p->next;
    p->next = q;
}

// 打印函数
void List::print() {
    cout << "print result be shown below: " << endl;
    Node *p = head->next;
    while (p != NULL) {
        cout << p->data << ", ";
        p = p->next;
    }
    cout << endl;
}

// 测试
int main() {
    // 从头部将节点据插入到链表中
    List list;
    list.insert(1);
    list.insert(3);
    list.insert(5);
    // 打印链表
    list.print();

    // 将新节点插入到指点节点之前
    list.insertPosBefore(1, 2);
    list.insertPosBefore(3, 4);
    list.insertPosBefore(5, 6);
    // 打印链表
    list.print();

    // 从尾部将节点插入到链表中
    list.insertTail(-1);
    list.insertTail(-3);
    list.insertTail(-5);
    // 打印链表
    list.print();


    // 将新节点插入到指定节点之后
    list.insertPosAfter(-1, -2);
    list.insertPosAfter(-3, -4);
    list.insertPosAfter(-5, -6);
    //打印链表
```

```
    list.print();

    return 0;
}
```

程序运行结果如下：

```
print result be shown below:
5, 3, 1,
print result be shown below:
6, 5, 4, 3, 2, 1,
print result be shown below:
6, 5, 4, 3, 2, 1, -1, -3, -5,
print result be shown below:
6, 5, 4, 3, 2, 1, -1, -2, -3, -4, -5, -6,
```

---

2. 单链表的删除

```
/**
 * @Author: GoogTech
 * @Email: googtech@qq.com
 * @Website: https://algorithm.show
 * @Date: 2021,6,21
 *
 * 2.单链表的删除
 *  (1).删除头节点
 *  (2).删除尾节点
 *  (3).删除指定节点
 *  (4).删除指定节点的前一个节点
 *  (5).删除指定节点的后一个节点
 */
#include "iostream"

using namespace std;

class List {
public:
    List() { createList(); }

    // 创建头节点
    void createList();

    // 从头部插入
    void insert(const int &d);

    // 删除头节点
    void delHead();

    // 删除尾节点
    void delTail();
```

```
    // 删除指定节点
    void del(const int &d);

    // 删除指定节点的前一个节点
    void delBefore(const int &d);

    // 删除指定节点的后一个节点
    void delAfter(const int &d);

    // 打印
    void print();

private:
    // 定义节点结构
    struct Node {
        int data;
        Node *next;

        Node(const int &d) : data(d), next(NULL) {}
    };

    // 声明链表头节点
    Node *head;

    // 查找 d 的节点位置
    Node *find(const int &d) {
        Node *p = head;
        while (p != NULL) {
            if (p->data == d) {
                break;
            }
            p = p->next;
        }
        return p;
    }

    // 查找 d 前一节点的位置
    Node *findBefore(const int &d) {
        Node *p = head, *preNode = NULL;
        while (p != NULL) {
            if (p->data == d) {
                break;
            }
            preNode = p;
            p = p->next;
        }
        return preNode;
    }

    // 查找 d 前一节点的前一个节点的位置
    Node *findBBefore(const int &d) {
        Node *p = head, *preNode = head, *ppreNode = head;
        while (p != NULL) {
            if (p->data == d) {
                break;
            }
            ppreNode = preNode;
            preNode = p;
```

```
            p = p->next;
        }
        return ppreNode;
    }

    // 查找尾部节点的前一个节点的位置
    Node *findTailBefore() {
        Node *p = head;
        while (p->next->next != NULL) {
            p = p->next;
        }
        return p;
    }

};

// 创建头节点
void List::createList() {
    head = new Node(0);
}

// 从头插入一个节点
void List::insert(const int &d) {
    Node *p = new Node(d);
    p->next = head->next;
    head->next = p;
}

// 删除头节点
void List::delHead() {
    Node *p = head->next; // 保存待删节点
    head->next = head->next->next;
    delete p;
}

// 删除尾节点
void List::delTail() {
    Node *p = findTailBefore();
    Node *q = p->next;
    p->next = NULL;
    delete q;
}

// 删除指定节点
void List::del(const int &d) {
    Node *p = findBefore(d);
    Node *q = p->next;
    p->next = p->next->next;
    delete q;
}

// 删除指定节点的前一个节点
void List::delBefore(const int &d) {
    Node *p = findBBefore(d);
    Node *q = p->next;
    p->next = q->next;
    delete q;
}
```

```
// 删除指定节点的后一个节点
void List::delAfter(const int &d) {
    Node *p = find(d);
    Node *q = p->next;
    p->next = q->next;
    delete q;
}

// 打印函数
void List::print() {
    cout << "print result be shown below: " << endl;
    Node *p = head->next;
    while (p != NULL) {
        cout << p->data << ", ";
        p = p->next;
    }
    cout << endl;
}

// 测试
int main() {
    // 从头部将节点据插入到链表中
    List list;
    list.insert(1);
    list.insert(2);
    list.insert(3);
    list.insert(4);
    list.insert(5);
    list.insert(6);
    list.insert(7);
    // 打印链表
    list.print();

    // 删除头节点
    list.delHead();
    // 打印链表
    list.print();

    // 删除尾节点
    list.delTail();
    // 打印链表
    list.print();

    // 删除指定节点
    list.del(3);
    // 打印链表
    list.print();

    // 删除指定节点的前一个节点
    list.delBefore(2);
    // 打印链表
    list.print();

    // 删除指定节点的后一个节点
    list.delAfter(6);
    // 打印链表
    list.print();
```

```
    return 0;
}
```

程序运行结果如下：

```
print result be shown below:
7, 6, 5, 4, 3, 2, 1,
print result be shown below:
6, 5, 4, 3, 2, 1,
print result be shown below:
6, 5, 4, 3, 2,
print result be shown below:
6, 5, 4, 2,
print result be shown below:
6, 5, 2,
print result be shown below:
6, 2,
```

---

3. 顺序有序表的合并

```
/**
 * @Author: GoogTech
 * @Email: googtech@qq.com
 * @Website: https://algorithm.show
 * @Date: 2021,6,21
 *
 * 3.合并两个有序顺序表
 */
#include "iostream"

#define MAXSIZE 100

using namespace std;

// 定义顺序表结构
typedef int elementType;
typedef struct {
    elementType *data;
    int length;
} sqList;

class Solution {
public:
    // 初始化顺序表
    bool initList(sqList &list) {
        list.length = 0;
        list.data = new int[MAXSIZE];
        return list.data == NULL ? false : true;
    }
```

```
    // 创建顺序表
    bool createlist(sqList &list, int len) {
        if (len < 1 || len > MAXSIZE) return false;
        list.length = len;
        for (int i = 0; i < len; i++) {
            cout << "Please input a element data: ";
            cin >> list.data[i];
        }
        cout << "The sequence list created successfully" << endl;
        return true;
    }

    // 打印顺序表
    void print(sqList &list) {
        cout << "The print result be shown below: " << endl;
        for (int i = 0; i < list.length; i++) {
            cout << list.data[i] << ", ";
        }
        cout << endl;
    }

    // 合并两个有序顺序表
    void mergeTwoList(sqList list1, sqList list2, sqList &newList) {
        newList.length = list1.length + list2.length;
        int i = 0, j = 0, k = 0, flag = 0;
        // 遍历顺序表
        while (true) {
            // 判断顺序表是否遍历完毕
            if (i == list1.length) break;
            if (j == list2.length) {
                flag = 1;
                break;
            }
            // 逐个比较两顺序表中元素值的大小，并通过下标值自增来构建新顺序表
            if (list1.data[i] <= list2.data[j]) {
                newList.data[k++] = list1.data[i++];
            } else {
                newList.data[k++] = list2.data[j++];
            }
        }
        // 若 flag 为 0 则说明 list1 先遍历完毕，反之 list2.
        if (flag == 0) {
            // 若 list1 先遍历完毕，则将 list2 中剩余的元素值依次添加到新顺序表的尾部
            while (k < newList.length) {
                newList.data[k++] = list2.data[j++];
            }
        } else {
            // 反之则将 list1 中剩余的元素值依次添加到新顺序表的尾部
            while (k < newList.length) {
                newList.data[k++] = list1.data[i++];
            }
        }
    }
};

int main() {
    Solution s;
```

```
    // 初始化顺序表
    sqList list1, list2, newList;
    if (!s.initList(list1) || !s.initList(list2) || !s.initList(newList)) {
        cout << "The sequence list inited unsuccessfully !";
        return 0;
    }

    // 创建顺序表，并将两顺序表的长度都设置为 3
    s.createlist(list1, 3);
    s.createlist(list2, 3);
    // 打印两顺序表中的元素值
    s.print(list1);
    s.print(list2);

    // 合并两个有序顺序表
    s.mergeTwoList(list1, list2, newList);
    // 打印合并后的顺序表中的元素值
    s.print(newList);

    return 0;
}
```

程序运行结果如下：

```
Please input a element data: 1
Please input a element data: 3
Please input a element data: 5
The sequence list created successfully
Please input a element data: 2
Please input a element data: 4
Please input a element data: 6
The sequence list created successfully
The print result be shown below:
1, 3, 5,
The print result be shown below:
2, 4, 6,
The print result be shown below:
1, 2, 3, 4, 5, 6,
```

---

4. 链式有序表的合并

```
/**
 * @Author: GoogTech
 * @Email: googtech@qq.com
 * @Website: https://algorithm.show
 * @Date: 2021,6,21
 *
 * 4.合并两个有序链表( 迭代法 )
 */
#include "iostream"
```

```cpp
using namespace std;

// 定义节点结构
struct ListNode {
    int data;
    ListNode *next;

    ListNode(const int &d) : data(d), next(NULL) {}
};

class Solution {
public:
    // 合并两个有序链表(迭代法)
    static ListNode *mergeTwoLists(ListNode *l1, ListNode *l2) {
        // 判断链表是否为空
        if (l1 == NULL) return l2;
        if (l2 == NULL) return l1;
        // 初始化哨兵节点与其头指针
        ListNode *preHead = new ListNode(-1);
        ListNode *prev = preHead;
        // 遍历链表并逐个比较两节点大小，通过移动哨兵节点头指针来构建新链表
        while (l1 != nullptr && l2 != nullptr) {
            if (l1->data < l2->data) {
                prev->next = l1;
                l1 = l1->next;
            } else {
                prev->next = l2;
                l2 = l2->next;
            }
            // 每比较一次，哨兵头指针都需要往后移动一位
            prev = prev->next;
        }
        // l1 与 l2 合并结束后，最后还剩一个链表是非空的，这时需将链表末尾指向未合并完的链表
        prev->next = l1 == nullptr ? l2 : l1;
        return preHead->next;
    }

    // 打印链表
    static void print(ListNode *head) {
        cout << "print result be shown below: " << endl;
        ListNode *p = head->next;
        while (p != NULL) {
            cout << p->data << ", ";
            p = p->next;
        }
        cout << endl;
    }
};

int main() {
    // 初始第一个有序链表
    ListNode *preHead1 = new ListNode(-1);
    ListNode *prev1 = preHead1;
    for (int i = 1; i < 10; i += 2) {
        ListNode *tempNode = new ListNode(i);
        prev1->next = tempNode;
        prev1 = tempNode;
    }
```

```
    // 打印链表
    Solution::print(preHead1);

    // 初始化第二个有序链表
    ListNode *preHead2 = new ListNode(-1);
    ListNode *prev2 = preHead2;
    for (int j = 2; j <= 10; j += 2) {
        ListNode *tempNode = new ListNode(j);
        prev2->next = tempNode;
        prev2 = tempNode;
    }
    // 打印链表
    Solution::print(preHead2);

    // 合并两个有序链表
    ListNode *newHead = Solution::mergeTwoLists(preHead1, preHead2);
    // 打印链表
    Solution::print(newHead->next);

    return 0;
}
```

程序运行结果如下:

```
print result be shown below:
1, 3, 5, 7, 9,
print result be shown below:
2, 4, 6, 8, 10,
print result be shown below:
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

---

5. 循环队列的入队

6. 循环队列的出队

```
/**
 * @Author: GoogTech
 * @Email: googtech@qq.com
 * @Website: https://algorithm.show
 * @Date: 2021,6,23
 *
 * 5,6.循环队列的入队
 *  (1).初始化队列
 *  (2).在队尾插入节点
 *  (3).删除队头节点
 *  (4).判断队列是否为空
 *  (5).队列长度
 *  (6).销毁队列
 *  (7).清空队列
 *  (8).打印队列节点数据
 */
```

```cpp
#include "iostream"

using namespace std;

#define  MAX_QUEUE_SIZE 100

typedef int ElementType;

// 定义循环队列节点结构
typedef struct {
    ElementType *base;
    int front;
    int rear;
} SequenceQueue;

class CircularQueue {
public:
    // 初始化队列
    void initQueue();

    // 在队尾插入节点
    void enQueue(ElementType val);

    // 删除队头节点
    void deQueue();

    // 判断队列是否为空
    bool isEmpty();

    // 队列长度
    int queueLength();

    // 销毁队列
    void destroyQueue();

    // 清空队列
    void clearQueue();

    // 打印队列节点数据
    void print();

private:
    // 初始化队列对象
    SequenceQueue sequenceQueue;
};

// 初始化队列
void CircularQueue::initQueue() {
    sequenceQueue.base = (ElementType *) malloc(sizeof(ElementType) *
MAX_QUEUE_SIZE);
    if (!sequenceQueue.base) {
        cout << "The sequence queue inited unsuccessfully." << endl;
        return;
    }
    sequenceQueue.front = sequenceQueue.rear = 0;
}

// 在队尾插入节点
```

```cpp
void CircularQueue::enQueue(ElementType val) {
    if ((sequenceQueue.rear + 1) % MAX_QUEUE_SIZE == sequenceQueue.front) {
        cout << "The sequence queue is filled." << endl;
        return;
    }
    sequenceQueue.base[sequenceQueue.rear] = val;
    sequenceQueue.rear = (sequenceQueue.rear + 1) % MAX_QUEUE_SIZE;
}

// 删除队头节点
void CircularQueue::deQueue() {
    if (isEmpty()) {
        cout << "The sequence queue is empty." << endl;
        return;
    }
    ElementType val = sequenceQueue.base[sequenceQueue.front];
    sequenceQueue.front = (sequenceQueue.front + 1) % MAX_QUEUE_SIZE;
    cout << "The front node of sequence queue be deleted and the value: " << val
<< endl;
}

// 判断队列是否为空
bool CircularQueue::isEmpty() {
    return sequenceQueue.front == sequenceQueue.rear;
}

// 队列长度
int CircularQueue::queueLength() {
    return (sequenceQueue.rear - sequenceQueue.front + MAX_QUEUE_SIZE) %
MAX_QUEUE_SIZE;
}

// 销毁队列
void CircularQueue::destroyQueue() {
    free(sequenceQueue.base);
    sequenceQueue.front = sequenceQueue.rear = 0;
}

// 清空队列
void CircularQueue::clearQueue() {
    destroyQueue();
    initQueue();
    cout << "The sequence queue be deleted successfully." << endl;
}

// 打印队列节点数据
void CircularQueue::print() {
    if (isEmpty()) {
        cout << "The sequence queue is empty." << endl;
        return;
    }
    cout << "The print result be shown below: " << endl;
    for (int i = sequenceQueue.front; i < sequenceQueue.rear; i++) {
        cout << sequenceQueue.base[i] << ", ";
    }
    cout << endl;
}
```

```
int main() {
    // 初始化循环队列
    CircularQueue circularQueue;
    circularQueue.initQueue();

    // 在队尾插入节点
    circularQueue.enQueue(1);
    circularQueue.enQueue(2);
    circularQueue.enQueue(3);
    circularQueue.enQueue(4);
    circularQueue.enQueue(5);
    circularQueue.enQueue(6);
    //  打印队列节点数据
    circularQueue.print();

    // 打印队列长度
    cout << "The length of circular queue: " << circularQueue.queueLength() <<
endl;

    // 连续出队两个队头节点
    circularQueue.deQueue();
    circularQueue.deQueue();
    //  打印队列节点数据
    circularQueue.print();

    // 清空队列
    circularQueue.clearQueue();
    // 打印队列长度
    cout << "The length of circular queue: " << circularQueue.queueLength() <<
endl;

    return 0;
}
```

程序运行结果如下：

```
The print result be shown below:
1, 2, 3, 4, 5, 6,
The length of circular queue: 6
The front node of sequence queue be deleted and the value: 1
The front node of sequence queue be deleted and the value: 2
The print result be shown below:
3, 4, 5, 6,
The sequence queue be deleted successfully.
The length of circular queue: 0
```

7. 表达式中括号匹配的检验

```
/**
 * @Author: GoogTech
 * @Email: googtech@qq.com
 * @Website: https://algorithm.show
 * @Date: 2021,6,23
 *
 * 7.表达式中括号匹配的检验
 */

#include "iostream"
#include "stack"

using namespace std;

class Solution {
public:
    static bool isValid(string str) {
        // 判断字符的个数是否为奇数
        if (str.size() % 2 != 0) return false;

        // 初始化栈
        stack<int> s;

        // 遍历字符串
        for (int i = 0; i < str.size(); i++) {
            // 将右括号压入栈中
            if (str[i] == '(') s.push(')');
            else if (str[i] == '[') s.push(']');
            else if (str[i] == '{') s.push('}');
            else if (str[i] == '<') s.push('>');
                // 判断栈中是否有待匹配符号的右括号
            else if (s.empty() || s.top() != str[i]) return false;
                // 匹配成功则弹出栈中对应的右括号
            else s.pop();
        }
        // 若左右括号全部匹配完毕则栈为空
        return s.empty();
    }
};

int main() {
    string str;
    cout << "Please input a string : ";
    cin >> str;
    // 0 : false and 1 : true
    cout << "The string is valid ? : " << Solution::isValid(str) << endl;
}
```

程序运行结果如下：

```
Please input a string :()
The string is valid ? :1

Please input a string :<>()[]{}
```

```
The string is valid ? :1

Please input a string :{[(<>)]}
The string is valid ? : 1

Please input a string :(]
The string is valid ? : 0

Please input a string :([)]
The string is valid ? : 0

Please input a string :{[<)]}
The string is valid ? : 0
```

---

8. BF模式匹配算法

```cpp
/**
 * @Author: GoogTech
 * @Email: googtech@qq.com
 * @Website: https://algorithm.show
 * @Date: 2021,6,23
 *
 * 8.BF模式匹配算法
 */
#include "iostream"

using namespace std;

// 暴力(Brute Force)算法
int BFAlgorithm(string S, string T, int pos) {
    // 主串与模式串的起始位置
    int i = pos, j = 0;
    while (i < S.length() && j < T.length()) {
        // 若相等则继续逐个比较后续字符
        if (S[i] == T[j]) {
            i++;
            j++;
            // 否则从主串的下一个字符起，重新和模式的起始字符进行比较
        } else {
            i = i - j + 1;
            j = 0;
        }
    }
    // 若匹配成功，则返回模式 T 中第一个字符相等的字符
    // 在主串 S 中的序号
    if (j >= T.length()) return i - T.length();
    else return -1;
}

int main() {
    string t = "abcac";
    string s = "ababcabcacbab";
```

```
    cout << "The index: " << BFAlgorithm(s, t, 0) + 1; // + 1 表示位置

    return 0;
}
```

程序运行结果如下：

```
The index: 6
```

---

9. 中序遍历的非递归算法

```
/**
 * @Author: GoogTech
 * @Email: googtech@qq.com
 * @Website: https://algorithm.show
 * @Date: 2021,6,23
 *
 * 10.中序遍历的非递归算法(迭代法)
 *
 * 扩展: 层次遍历
 */
#include "iostream"
#include "vector"
#include "stack"
#include "queue"

using namespace std;

// 定义二叉树节点结构
struct BitreeNode {
    int data;
    struct BitreeNode *lchild, *rchild;
};

// 初始化二叉树节点
void InitTreeNode(BitreeNode &root, int data, BitreeNode *lchild, BitreeNode
*rchild) {
    root.data = data;
    root.lchild = lchild;
    root.rchild = rchild;
}

// 迭代法
vector<int> InorderTraversal(BitreeNode *root) {
    vector<int> result;
    stack<BitreeNode *> s;
    // 遍历辅助栈
    while (root != nullptr || !s.empty()) {
        // 遍历根节点的左孩子节点，并将其依次压入栈中，随后进入操作2
        if (root != nullptr) { // 操作1
            s.push(root);
```

```
            root = root->lchild;
            // 弹出栈顶元素，若其有右孩子，则将右孩子节点压入栈中，随后继续操作1
        } else { // 操作2
            root = s.top();
            s.pop();
            result.push_back(root->data); // 存储栈中弹出的节点值(左 -> 根 -> 右)
            root = root->rchild;
        }
    }
    return result;
}

// 层次遍历
vector<int> BFSTraversal(BitreeNode *root) {
    vector<int> result;
    queue<BitreeNode *> queue;
    // 若根节点非空则将其入队
    if (root == nullptr) return result;
    queue.push(root);
    // 遍历辅助队列
    while (!queue.empty()) {
        // 将队首元素出队
        root = queue.front();
        queue.pop();
        // 存储队列出队的该层的节点值
        result.push_back(root->data);
        // 把当前节点所有的左右孩子节点全部入队
        if (root->lchild != nullptr) queue.push(root->lchild);
        if (root->rchild != nullptr) queue.push(root->rchild);
    }
    return result;
}

// 打印遍历结果
void Print(vector<int> v) {
    cout << "The print result be shown below: " << endl;
    for (auto data : v) {
        cout << data << ", ";
    }
    cout << endl;
}

int main() {
    // 构建二叉树
    BitreeNode t1, t2, t3, t4, t5, t6, t7;
    InitTreeNode(t7, 7, nullptr, nullptr);
    InitTreeNode(t6, 6, nullptr, nullptr);
    InitTreeNode(t5, 5, nullptr, nullptr);
    InitTreeNode(t4, 4, nullptr, nullptr);
    InitTreeNode(t3, 3, &t6, &t7);
    InitTreeNode(t2, 2, &t4, &t5);
    InitTreeNode(t1, 1, &t2, &t3);

    // 迭代法中序遍历二叉树，并打印结果
    Print(InorderTraversal(&t1));

    // 层次遍历二叉树，并打印结果
    Print(BFSTraversal(&t1));
```

```
}
```

程序运行结果如下：

```
The print result be shown below:
4, 2, 5, 1, 6, 3, 7,
The print result be shown below:
1, 2, 3, 4, 5, 6, 7,
```

---

10. 先序遍历的顺序建立二叉链表

```
/**
 * @Author: GoogTech
 * @Email: googtech@qq.com
 * @Website: https://algorithm.show
 * @Date: 2021,6,23
 *
 * 11.先序遍历的顺序建立二叉链表
 */
#include "iostream"

using namespace std;

// 定义二叉链表结构体
typedef struct ListNode {
    char data;
    struct ListNode *lchild, *rchild;
} ListNode, *ListTree;

// 先序建立二叉链表
void CreateListTree(ListTree &listTree) {
    char ch;
    cin >> ch;
    if (ch == '#') { // 一次读取一个字符哈
        listTree = NULL;
    } else {
        listTree = new ListNode;
        listTree->data = ch;
        CreateListTree(listTree->lchild);
        CreateListTree(listTree->rchild);
    }
}

// 先序遍历打印节点值
void PreOrderTraversal(ListTree listTree) {
    if (listTree) {
        cout << listTree->data << ", ";
        PreOrderTraversal(listTree->lchild);
        PreOrderTraversal(listTree->rchild);
    }
}
```

```
int main() {
    ListTree listTree;

    cout << "Please input the data of tree node: " << endl;
    CreateListTree(listTree);

    cout << "The result of Pre-Order Traversal: " << endl;
    PreOrderTraversal(listTree);

    return 0;
}
```

程序运行结果如下：

```
Please input the data of tree node:
ABC##DE#G##F###
The result of Pre-Order Traversal:
A, B, C, D, E, G, F,
```

---

11. 复制二叉树

```
/**
 * @Author: GoogTech
 * @Email: googtech@qq.com
 * @Website: https://algorithm.show
 * @Date: 2021,6,23
 *
 * 12.复制二叉树
 */
#include "iostream"
#include "vector"

using namespace std;

// 定义二叉树节点结构
struct BitreeNode {
    int data;
    struct BitreeNode *lchild, *rchild;
};

// 初始化二叉树节点
void InitTreeNode(BitreeNode &root, int data, BitreeNode *lchild, BitreeNode
*rchild) {
    root.data = data;
    root.lchild = lchild;
    root.rchild = rchild;
}

// 复制二叉树
void CopyBinaryTree(BitreeNode *root, BitreeNode *newRoot) {
```

```
    if (root == nullptr) return;
    // 复制根节点的值
    newRoot->data = root->data;
    // 判断左右子树是否为空
    if (root->lchild != nullptr) newRoot->lchild = new BitreeNode;
    if (root->rchild != nullptr) newRoot->rchild = new BitreeNode;
    // 递归复制左右子树
    CopyBinaryTree(root->lchild, newRoot->lchild);
    CopyBinaryTree(root->rchild, newRoot->rchild);
}

int main() {
    // 构建二叉树
    BitreeNode tree, tree2, tree3;
    InitTreeNode(tree3, 3, nullptr, nullptr);
    InitTreeNode(tree2, 2, nullptr, nullptr);
    InitTreeNode(tree, 1, &tree2, &tree3);

    // 复制二叉树
    BitreeNode newTree;
    CopyBinaryTree(&tree, &newTree);

    // 打印 newTree 的节点值( 偷懒~(•'◡'•)~ )
    cout << newTree.data << ", "
         << newTree.lchild->data << ", "
         << newTree.rchild->data;

    return 0;
}
```

程序运行结果如下:

```
1, 2, 3
```

---

12. 计算二叉树的深度

```
/**
 * @Author: GoogTech
 * @Email: googtech@qq.com
 * @Website: https://algorithm.show
 * @Date: 2021,6,23
 *
 * 13.计算二叉树的深度
 */
#include "iostream"

using namespace std;

// 定义二叉树节点结构
struct BitreeNode {
    int data;
```

```
    struct BitreeNode *lchild, *rchild;
};

// 初始化二叉树节点
void InitTreeNode(BitreeNode &root, int data, BitreeNode *lchild, BitreeNode
*rchild) {
    root.data = data;
    root.lchild = lchild;
    root.rchild = rchild;
}

// 后序遍历(DFS)获取最大子树的深度
int MaxDepth(BitreeNode *root) {
    if (root == nullptr) return 0;
    // 递归获取左右子树的深度
    int lDepth = MaxDepth(root->lchild) + 1;
    int rDepth = MaxDepth(root->rchild) + 1;
    // 返回最大子树的深度
    return lDepth > rDepth ? lDepth : rDepth;
}

int main() {
    // 构建二叉树
    BitreeNode t1, t2, t3, t4, t5;
    InitTreeNode(t5, 5, nullptr, nullptr);
    InitTreeNode(t4, 4, nullptr, nullptr);
    InitTreeNode(t3, 3, &t4, &t5);
    InitTreeNode(t2, 2, nullptr, nullptr);
    InitTreeNode(t1, 1, &t2, &t3);

    // 获取二叉树的深度
    cout << "The depth of binary tree: " << MaxDepth(&t1);

    return 0;
}
```

程序运行结果如下：

```
The depth of binary tree: 3
```

---

13. 冒泡排序

```
/**
 * @Author: GoogTech
 * @Email: googtech@qq.com
 * @Website: https://algorithm.show
 * @Date: 2021,6,23
 *
 * 18.冒泡排序
 * 依次比较相邻的关键字，若前者大于后者则相互交换位置，
 * 一直按这种方式进行下去，最终最大的那个关键字被交换到了最后，即一趟冒泡排序完成.
```

```
 * 经过多趟这样的排序，最终使整个序列有序.
 *
 * 时间复杂度: O(n²)，空间复杂度(1)，稳定.
 */
#include "iostream"
#include "vector"

using namespace std;

static void BubbleSort(vector<int> &R) {
    int temp;
    bool flag; // flag 用于标记本趟排序是否发生了交换
    for (int i = 0; i < R.size() - 1; i++) {
        flag = false;
        for (int j = i + 1; j <= R.size() - 1; j++) {
            // 相邻关键字两两对比
            if (R[i] > R[j]) {
                // 每次 i 后面的关键字比 R[i] 小就交换
                temp = R[j];
                R[j] = R[i];
                R[i] = temp;
                flag = true;
            }
        }
        // 若一趟排序过程没有发生关键字交换，则证明序列有序
        if (flag == false) return;
    }
}
```

程序运行结果如下：

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

---

14. 直接插入排序

```
/**
 * @Author: GoogTech
 * @Email: googtech@qq.com
 * @Website: https://algorithm.show
 * @Date: 2021,6,23
 *
 * 19.直接插入排序
 * 每趟将一个待排序的关键字按照其值的大小，
 * 插入到已经排好的部分有序序列的适当位置上，
 * 直到所有待排序关键字都被插入到有序序列中为止.
 *
 * 时间复杂度为: O(n²)，空间复杂度为: O(1)，稳定.
 */
#include "iostream"
#include "vector"
```

```
using namespace std;

static void InserSort(vector<int> &R) {
    for (int i = 1; i < R.size(); i++) {
        int cur = R[i]; // 将待插入关键字暂存在 cur 中
        int index = i - 1; // index为已排序元素的最后一个索引数
        // 从待排关键字之前的关键字开始扫描,
        while (index >= 0 && cur < R[index]) {
            // 如果大于待排关键字, 则后移动一位
            R[index + 1] = R[index];
            index--;
        }
        // 找到插入位置, 将 cur 中暂存的待排序关键字插入
        R[index + 1] = cur;
    }
}

int main() {
    vector<int> R = {10, 1, 9, 2, 8, 3, 7, 4, 6, 5};
    InserSort(R);
    for (auto r : R) {
        cout << r << ", ";
    }

    return 0;
}
```

程序运行结果如下:

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

---

15. 简单选择排序

```
/**
 * @Author: GoogTech
 * @Email: googtech@qq.com
 * @Website: https://algorithm.show
 * @Date: 2021,6,23
 *
 * 20.简单选择排序
 * 简单选择排序采用最简单的选择方式,
 * 从头到尾顺序扫描序列, 找出最小的一个记录,
 * 然后和第一个记录进行交换, 接着继续从剩下的无序序列中继续这种选择和交换, 最终使序列有序.
 *
 * 时间复杂度为: O(n²), 空间复杂度为: O(1).
 */

#include "iostream"
#include "vector"

using namespace std;
```

```
static void SelectSort(vector<int> &R) {
    int i, j, k;
    int temp;
    for (i = 0; i < R.size(); ++i) {
        k = i;
        // 从无序序列中挑出一个最小的关键字
        for (j = i + 1; j < R.size(); ++j) {
            if (R[k] > R[j]) {
                k = j;
            }
        }
        // 将最小关键字与无序序列中的第一个关键字进行交换
        temp = R[i];
        R[i] = R[k];
        R[k] = temp;
    }
}

int main() {
    vector<int> R = {10, 1, 9, 2, 8, 3, 7, 4, 6, 5};
    SelectSort(R);
    for (auto r : R) {
        cout << r << ", ";
    }

    return 0;
}
```

程序运行结果如下：

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

---

16. 快速排序

```
/**
 * @Author: GoogTech
 * @Email: googtech@qq.com
 * @Website: https://algorithm.show
 * @Date: 2021,6,24
 *
 * 20.快速排序
 *
 * 其算法思想如下:
 * 1.在待排序的元素中任取一个元素作为基准(通常取第一个，但最好随机取哈)，称为基准元素.
 * 2.将待排序的元素进行区分，比基准元素大的元素放在它的右边，比其小的放在它的左边.
 * 3.对左右两个分区重复以上步骤，直到所有元素都是有序的.
 *
 * 最坏的情况下，时间复杂度为 O(n²). 最好的情况下，空间复杂度为 O(log₂n). 不稳定.
 */
#include "iostream"
```

```cpp
using namespace std;

// 划分算法
int Partition(int A[], int low, int high) {
    int pivot = A[low]; // 将数组中的第一个元素设为基准元素，对数组进行划分
    // 数组左右两边交替扫描，直到 low = high
    while (low < high) {
        // 将比基准元素小的元素移动到左端
        while (low < high && A[high] >= pivot) --high;
        A[low] = A[high];
        // 将比基准元素大的元素移动到右端
        while (low < high && A[low] <= pivot) ++low;
        A[high] = A[low];
    }
    // 将基准元素存储到最终位置
    A[low] = pivot;
    // 返回基准元素 pivot 的最终位置
    return low;
}

// 快速排序算法
void QuickSort(int A[], int low, int high) {
    if (low < high) {
        // 获取基准元素 pivot 的最终位置
        int pivot = Partition(A, low, high);
        // 依次对划分的两个子数组进行递归排序
        QuickSort(A, low, pivot - 1);
        QuickSort(A, pivot + 1, high);
    }
}

// 打印数组元素
void Print(int arr[], int len) {
    cout << "The print result be shown blow: " << endl;
    for (int i = 0; i < len; i++) {
        cout << arr[i] << ", ";
    }
    cout << endl;
}

int main() {
    int arr[] = {10, 1, 9, 2, 8, 3, 7, 4, 6, 5};
    QuickSort(arr, 0, 9);
    Print(arr, 9);

    return 0;
}
```

程序运行结果如下：

```
The print result be shown blow:
1, 2, 3, 4, 5, 6, 7, 8, 9,
```

17. 散列(哈希)查找

基本概念

- `散列函数` 和 `散列地址` : 在记录的存储位置 p 和其关键字 key 之间建立一个确定的对应关系H, 使 `p = H( key)` , 称这个对应关系 H 为散列函数, p为散列地址.
- `散列表` : 一个有限连续的地址空间, 用以存储按散列函数计算得到相应散列地址的数据记录. 通常散列表的存储空间是一个一维数组, 散列地址是数组的下标.
- `散列查找` 又称 `哈希查找` , 利用散列函数进行查找的过程叫散列查找.

处理冲突的方法(开放地址法)

1. 线性探测法

   基本思想 : 可将散列表假想成一个循环表, 当发生冲突时, 从初次发生冲突的位置依次向后探测其它的地址.

2. 链地址法

   基本思想 : 链地址法是把所有的同义词用单链表连接起来的方法 .

3. 平方探测法

   基本思想 : 设发生冲突的地址为 d, 则用平方探测法所得的新的地址序列为 $d + 1^2$, $d - 1^2$, $d + 2^2$, $d - 2^2 \cdots$.

   例题 : 设散列表长为7, 记录关键字组为 14, 6, 20, 16, 61, 23, 散列函数 : `H(key) = key MOD 7`, 冲突处理采用线性探测法, 其过程如下所示 :

   (1). H(14) = 14 MOD 7 = 0

   (2). H(6) = 6 MOD 7 = 6

   (3). H(20) = 20 MOD 7 = 6 `冲突`

   (4). H(20) = (6 + 1^2) MOD 7 = 0 `又冲突`

   (5). H(20) = (6 - 1^2) MOD 7 = 5

   (6). H(16) =16 MOD 7 = 2

   (7). H(61) = 61 MOD 7 = 5 `冲突`

   (8). H(61) = (5 + 1^2) MOD 7 = 6 `又冲突`

   (9). H(61) = (5 - 1^2) MOD 7 = 4

   (10). H(23) = 23 MOD 7 = 2 `冲突`

   (11). H(23) = (23 + 1^2) MOD 7 = 3

14		**16**	**23**	**61**	**20**	**6**

---

18. 待实现

18. KMP模式匹配算法
19. 构造平衡二叉排序树
20. 普利姆最小生成树算法
21. 构造哈夫曼树并算出WPL
22. 先 / 中 / 后序遍历线索二叉树
23. 用邻接表来表示图的DFS
24. 广度优先搜索遍历连通图
25. 迪杰斯特拉最短路径算法

总结

求知若饥, 虚心若愚. 更多算法题解请关注 : *https://algorithm.show*