数据结构与算法期末复习样卷解析
Final

尹超

中国科学院大学，北京 100049

Carter Yin

University of Chinese Academy of Sciences, Beijing 100049, China

2025.6

序言

本文为笔者数据结构与算法的期末样卷解析。

望老师批评指正。

目录

题目 数据结构与算法期末样卷解析

一、选择题（共 60 分）

(1) 当输入非法时，一个“好”的算法会进行适当处理，而不是产生难以理解的输出结果。这称为算法的（B）。

A. 可读性 B. 健壮性 C. 正确性 D. 有穷性

Solution 1.
- **可读性 *(Readability):*** 指算法的代码易于阅读、理解和维护。
- **健壮性 *(Robustness):*** 指算法在遇到非法的、错误的或意外的输入数据时，能够做出适当的处理（如报错、返回特定值），而不会崩溃或产生不可预料的结果。这与题干描述完全相符。
- **正确性 *(Correctness):*** 指算法对于合法的输入，能够在有限时间内产生满足要求的结果。
- **有穷性 *(Finiteness):*** 指算法必须在执行有限的步骤后终止。

(2) 在一个初始为空的向量上依次执行：insert(0, 2), insert(1, 6), put(0, 1), remove(1), insert(0, 8) 后的结果是（C）。

A. {6, 2, 8} B. {2, 6, 0, 8} C. {8, 1} D. {2, 1, 8}

Solution 2. 我们逐步跟踪向量的状态：
- 初始状态*: ‘V = ‘*
- *‘insert(0, 2)‘:* 在索引 *0* 处插入 *2*。*‘V = 2‘*
- *‘insert(1, 6)‘:* 在索引 *1* 处插入 *6*。*‘V = 2, 6‘*
- *‘put(0, 1)‘:* 将索引 *0* 处的值替换为 *1*。*‘V = 1, 6‘*
- *‘remove(1)‘:* 删除索引 *1* 处的元素（即 *6*）。*‘V = 1‘*
- *‘insert(0, 8)‘:* 在索引 *0* 处插入 *8*。*‘V = 8, 1‘*

最终结果为 *{8, 1}*。

(3) 对于插入过程排序中的已排序子序列（设其长度为 k），下列说法正确的是（C）。

A. 其中的元素是整个序列中最小的 k 个元素
B. 其中的元素是整个序列中最大的 k 个元素
C. 其中的元素是原序列中位于前方的 k 个元素
D. 其中的元素是原序列中位于后方的 k 个元素

Solution 3. 插入排序的工作方式是，从第二个元素开始，依次将每个元素插入到其左侧已经排好序的子序列中。因此，在处理到第 k 个元素后，左侧的已排序子序列是由原序列的前 k 个元素组成的，只是它们的顺序被打乱重排了。它并不一定是整个序列中最小或最大的 k 个元素。例如，对 *‘5, 1, 4‘*排序，处理完 *‘5‘*和 *‘1‘*后，已排序子序列是 *‘1, 5‘*，它们是原序列的前两个元素。

(4) 选择排序算法的哪种实现是稳定的：（A）

A. 每一趟将最小元素移到前方，对于多个相等的最小元素，选取其中位置最靠前者。
B. 每一趟将最大元素移到后方，对于多个相等的最大元素，选取其中位置最靠前者
C. 每一趟将最小元素移到前方，对于多个相等的最小元素，选取其中位置最靠后者。
D. 以上实现皆稳定。

Solution 4. 稳定性要求相等的元素在排序后保持其原始的相对顺序。标准的选择排序（通过交换）是不稳定的。要使其稳定，关键在于处理相等元素。
- 考虑序列 *‘[..., X, ..., Y, ...]‘*，其中 *‘X‘* 和 *‘Y‘* 相等，且 *‘X‘* 在 *‘Y‘* 前面。
- 在某一趟排序中，如果 *‘X‘* 和 *‘Y‘* 是当前未排序部分的共同最小值。

- **选项 A**: 选取位置最靠前的 'X'。如果 'X' 恰好是当前趟次的第一个元素，则不发生交换，'Y' 的相对位置不变。如果 'X' 不是第一个元素，它会与第一个元素交换，但它仍然在 'Y' 的前面。这样，'X' 和 'Y' 的相对顺序得以保持。
- **选项 C**: 选取位置最靠后的 'Y'。如果将 'Y' 与当前趟次的第一个元素交换，'Y' 就会跑到 'X' 的前面，破坏了稳定性。

因此，选项 A 描述的策略可以实现稳定的选择排序。

(5) 利用栈结构进行中缀表达式求值，什么时候进行实际的运算?（D）

A. 每遇到一个新的操作数

B. 每遇到一个新的操作符

C. 当前的操作符比栈顶的操作符优先级高

D. 当前的操作符比栈顶的操作符优先级低

Solution 5. 在使用双栈（一个操作数栈，一个操作符栈）求值时，运算的时机由操作符的优先级决定。

- 当遇到一个操作符时，需要与操作符栈顶的已有操作符比较优先级。
- **选项 C**: 如果当前操作符优先级**更高**（例如，遇到 '*'，栈顶是 '+'），则应先处理更高优先级的运算，所以直接将当前操作符入栈，等待后续处理。
- **选项 D**: 如果当前操作符优先级**更低或相等**（例如，遇到 '+'，栈顶是 '*' 或 '+'），则说明栈顶的操作符可以并且必须先进行运算。此时，从操作符栈弹出一个操作符，从操作数栈弹出两个数，进行运算，并将结果压回操作数栈。这个过程会一直持续，直到栈顶操作符的优先级低于当前操作符。

因此，运算的触发条件是当前操作符的优先级不高于（即低于或等于）栈顶操作符的优先级。选项 D 是这个条件的主要情况。

(6) 分别采用每次追加固定内存空间和每次内存空间翻倍两种扩容策略，规模为 n 的向量插入元素的分摊时间复杂度分别为:（A）

A. O(n), O(1) B. O (n), O (n) C. O (1), O (1) D. O (n), O(log2n)

Solution 6.

- **追加固定内存空间**: 假设每次扩容增加固定大小 C。当向量大小从 0 增长到 n 时，会发生大约 n/C 次扩容。第 k 次扩容的成本是 $O(k \times C)$。总成本是 $O(C + 2C + \ldots + (n/C)C) = O(C \times (1 + 2 + \ldots + n/C)) = O(C \times (n/C)^2) = O(n^2/C) = O(n^2)$。将 $O(n^2)$ 的总成本分摊到 n 次插入操作上，每次插入的分摊成本是 $O(n)$。
- **内存空间翻倍**: 假设容量从 1 开始，每次翻倍。当向量大小增长到 n 时，扩容发生的时刻是大小为 $1, 2, 4, 8, \ldots, 2^k$（直到 $2^k \geq n$）。每次扩容的成本与当前大小成正比。总成本为 $O(1 + 2 + 4 + \ldots + n) \approx O(2n) = O(n)$。将 $O(n)$ 的总成本分摊到 n 次插入操作上，每次插入的分摊成本是 $O(1)$。

(7) 给定序列 {1, 2, 3...i...j...k...n}。下列哪个序列一定不是该序列的栈混洗:（C）

A. {⋯i⋯j⋯k⋯} B. {⋯k⋯j⋯i⋯} C. {⋯k⋯i⋯j⋯} D. {⋯j⋯k⋯i⋯}

Solution 7. 栈混洗有一个基本规则：如果 'j' 在 'i' 之后出栈，且 '$j > i$'，那么所有在 'i' 和 'j' 之间的元素 'k' ('$i < k < j$') 必须在 'j' 之前出栈。换言之，不可能出现 '$i < k < j$' 且出栈顺序为 '...j...k...' 的情况。一个更强的、更易于判断的规则是：不可能存在 '$i < j$'，使得出栈序列中 'k' 在 'i' 和 'j' 之间，且 '$k > j$'。即不可能有 '...i...k...j...' 这样的序列。让我们检查选项 C: '⋯k⋯i⋯j⋯'。假设 '$i < j < k$'。

- 为了让 'k' 先出栈，元素 'i, j, k' 必须都已入栈，此时栈的状态（从顶到底）是 'k, j, i'。

- 接下来 'k' 出栈。
- 此时栈顶是 'j'。为了让 'i' 出栈，必须先将 'j' 出栈。
- 但序列要求 'i' 在 'j' 之前出栈，这是矛盾的。因此，这种序列是不可能产生的。

(8) V={1, 2, 3, 4, 5, 6, 7}，在 V 中用 Fibonacci 查找元素 1，被选取为轴点 mi 的元素依次是（D）
A. 4, 3, 2, 1　B. 4, 2, 1　C. 5, 2, 1　D. 5, 3, 2, 1

Solution 8. *Fibonacci* 查找，目标元素为 *1*，序列长度 *n=7*。

- 首先找到最小的斐波那契数 *F(k) >= n+1=8*。*F(0)=0, F(1)=1, F(2)=1, F(3)=2, F(4)=3, F(5)=5, F(6)=8*。所以 *k=6*。
- **第 *1* 步：** *'lo=0, hi=7'*。轴点 *'mi = lo + F(k-1) - 1 = 0 + F(5) - 1 = 0 + 5 - 1 = 4'*。*V[4]=5*。因为 *1 < 5*，所以到左子序列查找。*'hi'* 更新为 *'mi=4'*，*'k'* 更新为 *'k-1=5'*。轴点元素为 ***5***。
- **第 *2* 步：** *'lo=0, hi=4'*。轴点 *'mi = lo + F(k-1) - 1 = 0 + F(4) - 1 = 0 + 3 - 1 = 2'*。*V[2]=3*。因为 *1 < 3*，所以到左子序列查找。*'hi'* 更新为 *'mi=2'*，*'k'* 更新为 *'k-1=4'*。轴点元素为 ***3***。
- **第 *3* 步：** *'lo=0, hi=2'*。轴点 *'mi = lo + F(k-1) - 1 = 0 + F(3) - 1 = 0 + 2 - 1 = 1'*。*V[1]=2*。因为 *1 < 2*，所以到左子序列查找。*'hi'* 更新为 *'mi=1'*，*'k'* 更新为 *'k-1=3'*。轴点元素为 ***2***。
- **第 *4* 步：** *'lo=0, hi=1'*。轴点 *'mi = lo + F(k-1) - 1 = 0 + F(2) - 1 = 0 + 1 - 1 = 0'*。*V[0]=1*。因为 *1 == 1*，查找成功。轴点元素为 ***1***。

因此，被选为轴点的元素依次是 *5, 3, 2, 1*。

(9) 用父节点 + 孩子节点的方法存储 n 个节点的树, 需要的空间是（B）。
A. O(1)　B. O(n)　C. O(nlogn)　D. O(n2)

Solution 9. 这种存储方式通常指每个节点包含一个指向其父节点的指针，以及一个数据结构（如链表或动态数组）来存储指向其所有孩子节点的指针。

- 每个节点都有一个父指针（根节点除外，可为 *null*），共 *n* 个指针，空间为 *O(n)*。
- 树中总共有 *n-1* 条边，每条边对应一个从父节点到孩子节点的指针。所有孩子指针的总数是 *n-1*，空间为 *O(n)*。

两者相加，总空间复杂度为 *O(n) + O(n) = O(n)*。

(10) 二叉树的中序遍历中第一个被访问的节点是（A）。
A. 最左的节点　B. 最右的节点　C. 根节点　D. 左侧分枝的叶节点

Solution 10. 中序遍历的顺序是"左-根-右"。为了访问第一个节点，算法会从根节点开始，不断地沿着左孩子指针向下，直到到达一个没有左孩子的节点。这个节点就是整棵树中"最靠左"的节点，它将是第一个被访问的节点。

(11) 对下图二叉树进行层次遍历，节点 F 正欲出队时队列中的元素从队头到队尾为（B）。
A. F　B. F, G　C. E, F　D. E, F, G

Solution 11. 层次遍历使用队列。我们跟踪队列的状态：

(1) 初始: *Q = {}*
(2) *Enqueue(A). Q = {A}*
(3) *Dequeue(A), Enqueue(B). Q = {B}*
(4) *Dequeue(B), Enqueue(C, D). Q = {C, D}*
(5) *Dequeue(C). Q = {D}*
(6) *Dequeue(D), Enqueue(E, F). Q = {E, F}*

(7) Dequeue(E), Enqueue(G). Q = {F, G}

在第 *7* 步之后，节点 *F* 位于队头，正准备出队。此时队列中的元素从队头到队尾是 *F, G*。

(12) 下列关于树的命题中错误的是:（D）

A. 顶点数为 n 的树的边数为 n-1。

B. 树中任意两顶点之间存在唯一路径。

C. 在树中添加任一条边都会破坏树的结构。

D. 在树中删除任一条边得到的还是树。

Solution 12.
- *A, B, C* 都是树的基本性质，是正确的。树是无环的连通图。
- *D* 是错误的。树是连通的，删除任意一条边都会导致图不再连通，从而分裂成两个独立的连通分量（即两棵树）。结果是一个森林，而不是一棵树。

(13) 关于二叉树遍历序列之间关系的说法错误的是:（D）

A. 已知先序遍历序列和中序遍历序列可以确定后序遍历序列

B. 已知中序遍历序列和后序遍历序列可以确定先序遍历序列

C. 已知中序遍历序列和后序遍历序列可以确定层次遍历序列

D. 已知先序遍历序列和后序遍历序列可以确定中序遍历序列

Solution 13.
- *A, B, C* 都是正确的。只要有中序遍历序列，再配合先序或后序遍历序列，就可以唯一地重建出二叉树的结构。一旦树的结构确定，任何一种遍历序列（包括后序、先序、层次）都可以生成。
- *D* 是错误的。只知道先序和后序遍历序列，无法唯一确定一棵二叉树。例如，一个根节点 *A*，左孩子 *B* 的树，和根节点 *A*，右孩子 *B* 的树，它们的先序遍历都是*"AB"*，后序遍历都是*"BA"*，但它们的中序遍历不同（*"BA" vs "AB"*）。因为无法唯一确定树的结构，所以也无法唯一确定中序遍历序列。

(14) 对平衡二叉树进行插入操作，对待插入的目标元素 e 进行查找后，若查找失败，_hot 指向的节点为:（B）

A. 待插入的节点　B. 被插入后的父亲　C. 被插入后的左孩子　D. 根节点

Solution 14. 在二叉搜索树的插入操作中，通常会用一个指针（题目中为 `_hot`）来跟踪当前搜索指针的父节点。当搜索因为指针变为 `NULL` 而失败时，`_hot` 正好指向新元素应该被插入的位置的父节点。因此，`_hot` 指向的是待插入节点的父亲。

(15) AVL 树中插入节点引发失衡，经旋转调整后重新平衡，此时包含节点 g, p, v 的子树高度（B）

A. 减小 1　B. 不变　C. 增加 1　D. 有可能不变也有可能增加 1

Solution 15. *AVL* 树的一个关键特性是，因插入而导致的失衡，在经过一次旋转（单旋或双旋）调整后，失衡子树的高度会恢复到其**插入前的高度**。也就是说，相对于插入前，子树高度不变。相对于插入后、旋转前，子树高度减小 *1*。题目问的是调整后的最终状态，通常是与操作前的稳定状态比较，故高度不变。

(16) 使用 Dijkstra 算法求下图中从顶点 A 到其他各顶点的最短路径，依次得到的各最短路径的目标顶点是（A）

A. {E, B, C, F, D}　B. {E, B, F, C, D}　C. {E, B, D, C, F}　D. {E, B, C, D, F}

Solution 16. *Dijkstra* 算法每次从未包含在最短路径集 *S* 中的顶点里，选取距离源点 *A* 最近的顶点 *u*，加入 *S*，并更新 *u* 的邻居到 *A* 的距离。根据选项 *A* 的顺序，可以反推出一个可能的执行过程和图的权重关系：

(1) *A* 到 *E* 的直接距离最短，选 ***E***。
(2) 更新后，*A* 经 *E* 到 *B* 的距离最短，选 ***B***。
(3) 更新后，*A* 经 *E*、*B* 到 *C* 的距离最短，选 ***C***。
(4) 更新后，*A* 经 *E*、*B*、*C* 到 ***F*** 的距离比到 ***D*** 的距离更短，选 ***F***。
(5) 最后选 ***D***。

这要求图的权重满足在第 *4* 步时，*'dist(F) < dist(D)'*。

(17) B 树高度的减少只会发生于（A）

A. 根节点的两个孩子合并　B. 根节点被删除　C. 根节点发生旋转　D. 根节点有多个关键码

Solution 17. *B* 树的高度只在删除操作中才可能减少。当删除导致某节点下溢，并需要与其兄弟合并时，这种合并可能会向上传播。如果合并一直传播到根节点的两个孩子，它们合并成一个新节点，此时原根节点中的一个关键码会下移到合并后的新节点中。如果这个关键码是根节点唯一的关键码，那么根节点就会变空，此时树的高度会减少 *1*，合并后的节点成为新的根。因此，根节点的两个孩子合并是高度减少的直接原因。

(18) 对下图进行拓扑排序，可以得到的不同的拓扑序列的个数是（B）

A．4　B. 3　C. 2　D. 1

Solution 18. 拓扑排序的序列个数取决于在算法执行的每一步中，有多少个入度为 *0* 的节点可供选择。假设存在一个图，其拓扑排序过程如下：

(1) 初始时只有一个节点 *A* 入度为 *0*。序列：*A...*
(2) *A* 出度后，节点 *B* 和 *C* 的入度变为 *0*。此时有两个选择。
(3) ***分支 1***：选择 *B*。*A* 出度后，节点 *D* 的入度仍不为 *0*。序列：*AB...*。此时 *C* 入度为 *0*，*D* 入度不为 *0*。只能选 *C*。序列：*ABC...*。*C* 出度后，*D* 入度为 *0*。最后选 *D*。得到序列 ***ABCD***。
(4) ***分支 2***：选择 *C*。*A* 出度后，*B* 和 *D* 的入度都不为 *0*。序列：*AC...*。此时 *B* 入度为 *0*。只能选 *B*。序列：*ACB...*。*B* 出度后，*D* 入度为 *0*。最后选 *D*。得到序列 ***ACBD***。

等一下，上述分析有误。让我们重新考虑一个能产生 *3* 个序列的图，例如 *A->B, A->C, B->D*。

(1) 只能先选 *A*。
(2) 接下来 *B* 和 *C* 的入度都为 *0*。
(3) ***分支 1***：选 *B*，再选 *C*。此时 *D* 的入度为 *0*。得到序列 ***ABCD***。
(4) ***分支 2***：选 *C*，再选 *B*。此时 *D* 的入度为 *0*。得到序列 ***ACBD***。
(5) ***分支 3***：选 *B*，但 *B* 出度后，*D* 的入度为 *0*，*C* 的入度也为 *0*。可以先选 *D* 再选 *C*。得到序列 ***ABDC***。

因此，对于图 *A->B, A->C, B->D*，可以得到 *3* 个不同的拓扑序列。

(19) 伸展树每次访问过某节点后都会把该节点（C）

A. 删除　B. 上移一层　C. 移动到根节点　D. 再次访问该节点

Solution 19. 伸展树（*Splay Tree*）的核心操作是"伸展"（*splaying*）。每当一个节点被访问（查找、插入、删除）后，都会通过一系列特定的旋转操作（*zig, zig-zig, zig-zag*）将其移动到树的根节点位置。这是伸展树的定义性特征，旨在优化后续对该节点的访问。

(20) G 是有向无环图，(u, v) 是 G 中的一条由 u 指向 v 的边。对 G 进行 DFS 的结果是：（C）

A. dTime(u) > dTime(v)　B. dTime(u) < dTime(v)

C. fTime(u) > fTime(v)　D. fTime(u) < fTime(v)

Solution 20. 在对有向图进行深度优先搜索（*DFS*）时，会记录每个顶点的发现时间 *‘dTime‘* 和完成时间 *‘fTime‘*。

- 因为 *‘(u, v)‘* 是一条边，所以在 *DFS* 过程中，如果从 *‘u‘* 访问到了 *‘v‘*（即 *‘(u,v)‘* 是树边或前向边），那么 *‘v‘* 的发现和完成都发生在 *‘u‘* 的发现之后和完成之前。
- 这意味着 *‘dTime(u) < dTime(v)‘* 且 *‘fTime(v) < fTime(u)‘*。
- 由于图 *G* 是无环图（*DAG*），不存在后向边，所以对于图中的任意一条边 *‘(u, v)‘*，都有 *‘fTime(u) > fTime(v)‘*。这个性质是拓扑排序算法的基础。
- 选项 *B* 和 *C* 都是正确的陈述，但 *‘fTime(u) > fTime(v)‘* 是一个更强的、用于判断 *DAG* 和进行拓扑排序的核心性质。

(21) 文本串的长度为 n，模式串的长度为 m，蛮力匹配的时间复杂度为（D ）

A. O(m) B. O(n) C. O(mlogn) D. O(mn)

Solution 21. 蛮力匹配算法的思路是，将模式串 *P* 与文本串 *T* 的所有可能起始位置对齐，然后进行比较。

- 文本串 *T* 中可能的起始位置有 *‘n-m+1‘* 个。这构成了外层循环，复杂度为 *O(n)*。
- 对于每一个起始位置，都需要将长度为 *‘m‘* 的模式串与文本串的对应子串进行比较。在最坏情况下，每次比较都需要进行 *‘m‘* 次字符对比。这构成了内层循环，复杂度为 *O(m)*。
- 总的时间复杂度是两者相乘，即 *O((n-m+1) * m) = O(nm)*。

(22) 对序列 A[0, n) 用快速排序算法进行排序，u 和 v 是该序列中的两个元素。在排序过程中，u 和 v 发生过比较，当且仅当（假定所有元素互异）:（C ）

A. u < v

B. u 在某次被选取为轴点

C. 对于所有介于 u 和 v 之间的元素（包括 u 和 v 本身），它们之中第一个被选为轴点的是 u 或者 v

D. 所有比 u 和 v 都小的元素都始终没有被选为轴点

Solution 22. 在快速排序中，元素之间的比较只发生在轴点（*pivot*）和当前子数组的其他元素之间。一旦划分完成，位于轴点两侧的元素将永不比较。

- 两个元素 *‘u‘* 和 *‘v‘* 要想发生比较，它们必须在某次划分中，一个作为轴点，另一个作为非轴点元素。
- 这要求在它们被比较之前，它们从未被任何一个值介于它们之间的轴点所分离。
- 如果第一个被选为轴点的、值介于 *‘u‘* 和 *‘v‘* 之间的元素是 *‘p‘ (‘u < p < v‘)*，那么 *‘u‘* 会被分到左边，*‘v‘* 会被分到右边，它们从此再无比较的机会。
- 因此，*‘u‘* 和 *‘v‘* 发生比较的充要条件是，在所有值介于 *‘u‘* 和 *‘v‘* 之间的元素中，第一个被选为轴点的是 *‘u‘* 或 *‘v‘* 本身。

(23) 允许对队列进行的操作是（C ）

A. 对队列中的元素排序

B. 取出最近进队的元素

C. 删除队头元素

D. 在队头元素之前插入元素

Solution 23. 队列（*Queue*）是一种先进先出（*FIFO*）的数据结构。

- *A.* 排序不是队列的基本操作。
- *B.* 取出最近进队的元素是栈（*LIFO*）的操作。

- *C.* 删除队头元素，即 *'dequeue()'* 或 *'pop()'*，是队列的核心出队操作。
- *D.* 在队头插入元素不符合队列的定义，插入操作（*'enqueue'*）应在队尾进行。

(24) 在使用独立链法解决冲突的散列表中查找某关键码，经过一系列的试探，最终查找成功。这些试探的元素（B ）。

A. 关键码的散列地址均不相同

B. 关键码的散列地址一定相同

C. 关键码的散列地址可能不同

D. 无法判断

Solution 24. 独立链法（*Separate Chaining*）是将所有散列到同一个地址（桶）的关键码存放在一个链表中。

- 查找过程首先计算关键码的散列地址，定位到对应的桶。
- 然后，遍历该桶所关联的链表，逐一比较链表中的元素，直到找到目标或到达链表末尾。
- 因此，所有被"试探"的元素都位于同一个链表中，它们当初被插入时，一定是计算出了相同的散列地址。

(25) 已知运算符包括 +、-、*、/、(和)，将中缀表达式 3+2-6*((5+1)/2-3)+9 转换为逆波兰表达式时，符号栈最少应该能保存多少个符号?（A ）

A. 5 B. 6 C. 7 D. 8

Solution 25. 我们跟踪算法执行过程中操作符栈的状态和大小：

(1) 遇到 *'+': push(+)*。栈: *'[+]'*。大小 =*1*。

(2) 遇到 *'-': pop(+), push(-)*。栈: *'[-]'*。大小 =*1*。

(3) 遇到 *'*': push(*)*。栈: *'[-, *]'*。大小 =*2*。

(4) 遇到 *'(': push(()*。栈: *'[-, *, (]'*。大小 =*3*。

(5) 遇到 *'(': push(()*。栈: *'[-, *, (, (]'*。大小 =*4*。

(6) 遇到 *'+': push(+)*。栈: *'[-, *, (, (, +]'*。大小 =*5*。

(7) 遇到 *')': pop(+)*。栈: *'[-, *, (, (]'*。大小 =*4*。

(8) 遇到 *'/': push(/)*。栈: *'[-, *, (, /, (]'*。大小 =*5*。

(9) 遇到 *')': pop(/), pop(()*。栈: *'[-, *, (]'*。大小 =*3*。

(10) 遇到 *'-': push(-)*。栈: *'[-, *, (, -]'*。大小 =*4*。

(11) 遇到 *')': pop(-), pop(()*。栈: *'[-, *]'*。大小 =*2*。

(12) 遇到 *'+': pop(*), pop(-), push(+)*。栈: *'[+]'*。大小 =*1*。

在整个过程中，栈的最大深度达到了 *5*。

(26) 若元素 a、b、c、d、e、f 依次入栈，允许入栈、出栈操作交替进行，但不允许连续 3 次进行出栈操作，则不可能得到的出栈序列是（D ）

A. d, c, e, b, f, a B. c, b, d, a, e, f

C. b, c, a, e, f, d D. a, f, e, d, c, b

Solution 26. 我们分析选项 *D* 的可行性。

- 为了得到 *'a'* 作为第一个出栈元素，操作必须是：*'push(a), pop(a)'*。此时栈为空。
- 为了得到 *'f'* 作为下一个出栈元素，操作必须是：*'push(b), push(c), push(d), push(e), push(f), pop(f)'*。此时栈中有 *'[b, c, d, e]'*。
- 为了得到 *'e'* 作为下一个出栈元素，操作必须是：*'pop(e)'*。

- 为了得到 *'d'* 作为下一个出栈元素，操作必须是：*'pop(d)'*。
- 到目前为止，我们已经连续执行了 *'pop(f), pop(e), pop(d)'* 三次出栈操作。这违反了“不允许连续 3 次进行出栈操作”的规则。因此，序列 *D* 是不可能得到的。

(27) 若平衡二叉树的高度为 6，且所有非叶结点的平衡因子均为 1，则该平衡二叉树的结点总数为（B）
A. 12 B. 20 C. 32 D. 33

Solution 27. 一个所有非叶节点平衡因子均为 *+1* 的 *AVL* 树是节点数最少的 *AVL* 树。其节点数 *'N(h)'* 满足递推关系 *'N(h) = N(h-1) + N(h-2) + 1'*。根据对“高度”的不同定义，结果会不同。

- ***定义 1***：单个节点高度为 *0*。*'N(0)=1', 'N(1)=2', 'N(2)=4', 'N(3)=7', 'N(4)=12', 'N(5)=20', 'N(6)=33'*。
- ***定义 2***：单个节点高度为 *1*。*'N(1)=1', 'N(2)=2', 'N(3)=4', 'N(4)=7', 'N(5)=12', 'N(6)=20'*。

选项中出现了 *20* 和 *33*，这表明题目可能采用了其中一种定义。根据常见的题目设置，选项 *B(20)* 是基于高度定义 *2* 的结果，这在很多教材中被使用。

(28) 已知一棵 3 阶 B 树，如下图所示。删除关键码 78 后得到一棵新的 B 树，其最右叶子结点中的关键码是（D）
A. 60 B. 60、62 C. 62、65 D. 65

Solution 28. 此题缺少必要的图示，但我们可以根据 *B* 树的删除逻辑和选项进行推断。删除操作可能会引起下溢（*underflow*），需要通过向兄弟节点“借位”（旋转）或与兄弟节点“合并”来恢复平衡。

- 原始的最右叶子节点包含 *78*。删除 *78* 后，该节点可能下溢。
- 如果发生下溢，它需要与其左兄弟合并或旋转。这个过程可能会级联，导致树的结构发生较大变化。
- 最终结果的最右叶子节点是 *'[65]'*，这暗示了原先包含 *78* 的整个右侧分支可能因为级联合并而被移除，使得原先的中间部分成为了新的最右部分。
- 一个可能的情景是：删除 *78* 导致其所在叶子节点下溢，进而与父节点和兄弟节点合并，这个合并又导致父节点下溢，最终一直合并到根节点，使得树的高度降低，原先包含 *'[...62, 65]'*的子树成为了新的最右叶子节点。

没有图的情况下，这是基于答案反推的最合理解释。

(29) 下列选项给出的是从根分别到达两个叶结点路径上的权值序列，能属于同一棵哈夫曼树的是（D）
A. 24, 10, 5 和 24, 10, 7 B. 24, 10, 5 和 24, 12, 7
C. 24, 10, 10 和 24, 14, 11 D. 24, 10, 5 和 24, 14, 6

Solution 29. 哈夫曼树的节点权值满足两个性质：*1)* 父节点权值等于其两个子节点权值之和。*2)* 因此，在从根到叶的路径上，权值必须是严格递减的。

- ***C*** 错误，因为路径 *'24, 10, 10'* 中有相等的权值，违反了性质 *2*。
- ***A*** 错误。路径 *1 ('24, 10, 5')* 意味着节点 *10* 的子节点是 *5* 和 *5 (*因为 *10=5+5)*。路径 *2 ('24, 10, 7')* 意味着节点 *10* 的子节点是 *7* 和 *3 (*因为 *10=7+3)*。同一个节点 *10* 不可能有两组不同的子节点。
- ***B*** 错误。路径 *1 ('24, 10, 5')* 意味着根节点 *24* 的子节点是 *10* 和 *14*。路径 *2 ('24, 12, 7')* 意味着根节点 *24* 的子节点是 *12* 和 *12*。同一个根节点 *24* 不可能有两组不同的子节点。
- ***D*** 正确。路径 *1 ('24, 10, 5')* 推断出：*'parent(5)=10', 'sibling(5)=5'*；*'parent(10)=24', 'sibling(10)=14'*。路径 *2 ('24, 14, 6')* 推断出：*'parent(6)=14', 'sibling(6)=8'*；*'parent(14)=24', 'sibling(14)=10'*。这两个推断是完全一致的，可以共存于同一棵哈夫曼树。

(30) 下列关于生成树的说法正确的是（C ）。

A. 一个图的最小生成树一定不唯一

B. 若图有 3 个最小生成树 T1、T2、T3，并且在 T1 中权值最小的边为 e，则 e 一定会出现在 T2 和 T3 中

C. 某个无向图存在相同权值的边，它的最小生成树仍然可能是唯一的

D. 一个图一定有大于等于一棵最小生成树

Solution 30.

- *A* 错误。如果图中所有边的权值都不同，则最小生成树（*MST*）是唯一的。
- *B* 错误。该陈述有歧义。如果 ‘*e*‘是图中所有边里权值最小的，且这个最小权值是唯一的，那么 ‘*e*‘一定在所有 *MST* 中。但如果存在多条权值相同的最小边，一个 *MST* 可能包含其中一条，另一个 *MST* 可能包含另一条。
- *C* 正确。即使图中存在权值相同的边，*MST* 也可能是唯一的。例如，在一个四边形 *ABCD* 中，边 *AB=1, CD=1, BC=2, AD=2*。*MST* 必须包含 *AB* 和 *CD*，然后为了连通，必须包含 *BC* 或 *AD* 中的一条。如果 *BC=2, AD=3*，那么 *MST* 必须包含 *BC*，此时 *MST* 是唯一的，即使图中存在权值为 *2* 的多条边。一个更简单的例子：*A-1-B, C-1-D, B-2-C*。*MST* 必须是 *A-B, C-D, B-C*，是唯一的。
- *D* 错误。只有连通图才有生成树。非连通图没有生成树，也就没有最小生成树。

二、填空题

(1)（8 分）本题关于堆排序，请在空白处填入对应的关键语句。

```
template <typename T> void Vector<T>::heapSort( Rank lo, Rank hi ) {
    T* A = _elem + lo; Rank n = hi - lo; heapify( A , n );
    while ( 0 < --n )
    { swap( A[0], A[n] ); percolateDown( A, n, 0 ); }
}

template <typename T> void heapify( T* A, const Rank n ) {
    for ( Rank i = n / 2 - 1; -1 < i; i-- ) // Corrected loop condition
        percolateDown( A, n, i );
}

template <typename T> Rank percolateDown( T* A, Rank n, Rank i ) {
    Rank j;
    while ( i != ( j = ProperParent( A, n, i ) ) )
        { swap( A[i], A[j] ); i = j; }
    return i;
}

// ------------------- 空白处 -------------------
```

Solution 31. ***1.* 填空代码**

空白处应填入 ‘*ProperParent*‘ 函数的实现。该函数用于在节点 ‘*i*‘ 及其左右孩子中，找出值最大的节点的秩（索引）。这是下滤操作 ‘*percolateDown*‘ 的核心辅助函数。

```
template <typename T>
static Rank ProperParent( T* A, Rank n, Rank i ) {
    Rank lc = 2 * i + 1; // 左孩子索引
    Rank rc = 2 * i + 2; // 右孩子索引
    Rank largest = i;    // 假设父节点i是三者中最大的

    // 如果左孩子存在(lc < n)，且其值比当前最大者(A[largest])还大
    if ( lc < n && A[largest] < A[lc] ) {
        largest = lc; // 更新最大者索引为左孩子
    }
    // 如果右孩子存在(rc < n)，且其值比当前最大者(A[largest])还大
    if ( rc < n && A[largest] < A[rc] ) {
        largest = rc; // 更新最大者索引为右孩子
    }

    // 返回三者中值最大者的索引
    return largest;
}
```

2. 逐行代码解释

heapSort 函数

这是堆排序的入口函数，负责驱动整个排序流程。

```
template <typename T>
void Vector<T>::heapSort( Rank lo, Rank hi ) {
    T* A = _elem + lo; Rank n = hi - lo;
    heapify( A , n );
    while ( 0 < --n ) {
        swap( A[0], A[n] );
        percolateDown( A, n, 0 );
    }
}
```

- `T* A = _elem + lo; Rank n = hi - lo;`
  创建指针 ‘*A*‘ 指向待排序区间的起点。**`hi - lo`**: 计算待排序区间的长度 ‘*n*‘。‘*hi*‘ 是上界（不含），‘*lo*‘ 是下界（含），二者之差即为元素个数。
- `heapify( A , n );`
  调用 ‘*heapify*‘ 函数，将无序数组 ‘*A*‘ 转换成一个大顶堆。完成后，‘*A[0]*‘ 是序列中的最大元素。
- `while ( 0 < --n )`
  主循环，共执行 *n-1* 次。每次循环将一个元素放到其最终的有序位置。‘*–n*‘ 在循环开始前执行，表示堆的有效大小每次都减 *1*。

- `swap( A[0], A[n] );`
  (重点) 这是排序的关键一步。‘*A[0]*‘ 是当前堆中的最大元素，‘*A[n]*‘ 是堆的最后一个元素。交换后，最大元素被放置到当前有效序列的末尾，这个位置就是它在整个排序完成后的最终位置。
- `percolateDown( A, n, 0 );`
  由于上一步的交换，新的堆顶 ‘*A[0]*‘ 很可能不再是最大值，破坏了堆的性质。此行代码调用“下滤”操作，将新的堆顶元素 ‘*A[0]*‘ 向下调整，直到它找到合适的位置，从而使规模为 ‘*n*‘ 的新堆重新恢复大顶堆性质。

heapify 函数

该函数将一个任意数组原地转换成一个大顶堆。

```
template <typename T>
void heapify( T* A, const Rank n ) {
    for ( Rank i = n / 2 - 1; -1 < i; i-- )
        percolateDown( A, n, i );
}

```

- `for ( Rank i = n / 2 - 1; ... )`
  (重点) 这个循环从最后一个非叶子节点开始，自下而上、自右至左地对每个内部节点执行下滤操作。**`n / 2 - 1`**：对于一个从 *0* 开始索引的完全二叉树，‘*n/2 - 1*‘ 正是最后一个非叶子节点的索引。从这个节点开始向前处理，可以保证当处理节点 ‘*i*‘ 时，它的左右子树都已经是合法的堆了。

percolateDown 函数

下滤操作，将不满足堆性质的节点 ‘*i*‘ 向下调整。

```
template <typename T>
Rank percolateDown( T* A, Rank n, Rank i ) {
    Rank j;
    while ( i != ( j = ProperParent( A, n, i ) ) )
        { swap( A[i], A[j] ); i = j; }
    return i;
}

```

- `while ( i != ( j = ProperParent( A, n, i ) ) )`
  ‘*ProperParent*‘ 函数返回节点 ‘*i*‘ 和其左右孩子三者中值最大者的索引 ‘*j*‘。如果 ‘*i*‘ 本身不是最大的（即 ‘*i* != *j*‘），说明 ‘*i*‘ 的位置不正确，需要向下调整，循环继续。如果 ‘*i*‘ 本身就是最大的（‘*i* == *j*‘），则调整完成，循环终止。
- `swap( A[i], A[j] ); i = j;`
  (重点) 如果父节点 ‘*i*‘ 小于其孩子 ‘*j*‘，则交换它们。**`i = j`**：这是下滤过程的核心。交换后，原来的父节点移动到了位置 ‘*j*‘。现在需要从这个新位置 ‘*j*‘ 继续向下检查，以确保它在新的子树中也满足堆性质。因此，将 ‘*i*‘ 更新为 ‘*j*‘，以便在下一轮循环中继续从新位置开始下滤。

其他题见试卷