/*

这是 CSC3100 2025 Spring Final 的回忆改编版和详细解答。

形式为 10 选择+5 判断+5 简答+6 大题。

选择题和判断题是笔者另出的，和原试题**完全不同**。

原题在选择和判断部分主要考察了**基础数据结构、排序算法基础、各种树、复杂度、BFS、Kruskal 算法**等知识点，

此卷这部分的题目也同样基于它们，且附有详细解答，强烈建议一做。

简答题包含**排序算法对比、DFS/BFS 原理、哈希碰撞、栈的实现；**

大题包含**手排 AVL、哈希、最大堆，手算 DFS/BFS、Kruskal/Prim 算法，手写 LCA 算法和第 k 大数算法**。

此卷的简答题与大题和原卷相比可达 99%相似，仅不包含“手算 DFS/BFS、Kruskal/Prim 算法”一题。另附有详细解答和思路解析，并非单纯给出答案，强烈建议阅读。

此卷结构为(题目+解析+答案)* n，便于即时查阅解答。所有题目与解答均做分页处理，题目页为 2-4，11，13，20，27，30，34，39。

所有原卷的代码题均只要求伪代码，此卷则均另附了 C++与 Python 实现。此卷中的所有代码都已通过编译与测试，读者若想观察实际效果，可自行添加 include/import 与 main（）/test 以运行。

所有题目皆为笔者手搓+GPT o3 检查与润色，如有错误请在 issue 中联系我修改，且请优先相信自己的解答和教授的授课内容。

此卷免费开源在 https://github.com/ZiyinLin/CSC3100_Review ，

如果你是从付费渠道获得的，请举报倒狗并咒骂其 4000+。

希望对你有帮助，给项目点个 Star 可助力笔者产出更多内容。

最后修改：5.17.2025

*/

1. Single choice problems

① Which traversal of a binary search tree yields the keys in strictly increasing order?

A. Preorder

B. Inorder

C. Postorder

D. Level-order

② Which comparison-based sorting algorithm has worst-case time complexity $\Theta(n \log n)$, and requires only $O(1)$ extra space? (in-place implementation)

A. Merge Sort

B. Quick Sort

C. Heap Sort

D. Bubble Sort

③ In a depth-first search (DFS) on a graph, the order in which vertices are first visited can be described as:

A. Visiting vertices in order of non-decreasing distance from the source

B. Always exploring the neighbor with the smallest label first

C. Following one path as deep as possible before backtracking

D. Visiting all neighbors of the current vertex before any grandchildren

④ Which of the following statements about breadth-first search (BFS) on an unweighted graph is true?

A. It always finds a minimum spanning tree.

B. It computes shortest-path distances (in number of edges) from the start vertex.

C. It visits vertices in non-decreasing order of their labels.

D. It requires a stack as its primary auxiliary data structure.

⑤ In a hash table using chaining with n elements and m buckets, the load factor $\alpha = n/m$ represents:

A. The expected number of elements per bucket

B. The maximum number of elements in any bucket

C. The probability of a collision on any insertion

D. The ratio of filled buckets to total buckets

⑥ Which of the following is not a guaranteed property of a binary max-heap with n elements?

A. It is a complete binary tree.

B. Each parent node's key ≥ its children's keys.

C. Its height is $\lfloor \log_2 n \rfloor$.

D. All leaves are at the same depth.

⑦ What is the worst-case time complexity of Kruskal's algorithm on a graph with V vertices and E edges, assuming edges are pre-sorted in O(E log E) and union-find with path compression is used?

A. $O(E \log E)$

B. $O(V \log V + E)$

C. $O(E\, \alpha(V))$

D. $O(V^2)$

⑧ Prim's algorithm for finding a minimum spanning tree relies on which fundamental property of MSTs?

A. Cut Property: the minimum-weight edge crossing any cut belongs to some MST

B. Cycle Property: the maximum-weight edge in any cycle cannot belong to an MST

C. Subtree Property: every subtree of an MST is itself a minimum spanning tree

D. Optimal Substructure: an MST of the whole graph contains MSTs of subgraphs

⑨ What is the average-case time complexity of searching for a key in a balanced binary search tree with n nodes?

A. O(1)

B. O(log n)

C. O(n)

D. O(n log n)

⑩ Which graph representation uses $\Theta(V^2)$ space for a graph with V vertices, regardless of the number of edges?

A. Adjacency list

B. Adjacency matrix

C. Edge list

D. Incidence list

1. Solutions

Quick check: BCCBA DCABB

① Which traversal of a binary search tree yields the keys in strictly increasing order?

A. Preorder

B. Inorder

C. Postorder

D. Level-order

② Which comparison-based sorting algorithm has worst-case time complexity $\Theta(n \log n)$, and requires only $O(1)$ extra space? (in-place implementation)

A. Merge Sort

Has $\Theta(n \log n)$ worst-case time, but requires $\Theta(n)$ auxiliary space—not in-place.

B. Quick Sort

Average-case is $\Theta(n \log n)$, but worst-case is $\Theta(n^2)$. Also uses $O(n)$ stack space in the worst case. Even after optimization can only reach $\Theta(\log n)$ rather than $O(1)$.

C. Heap Sort

Has worst-case $\Theta(n \log n)$, and it's an in-place algorithm using only constant extra space.

D. Bubble Sort

In-place, but its time complexity is $\Theta(n^2)$, not $\Theta(n \log n)$.

③ In a depth-first search (DFS) on a graph, the order in which vertices are first visited can be described as:

A. Visiting vertices in order of non-decreasing distance from the source

This describes breadth-first search (BFS), not DFS.

B. Always exploring the neighbor with the smallest label first

This may be used as a tie-breaker, but is not fundamental to DFS. Besides it doesn't emphasize it follows label order.

C. Following one path as deep as possible before backtracking

D. Visiting all neighbors of the current vertex before any grandchildren

This describes BFS behavior, not DFS.

④ Which of the following statements about breadth-first search (BFS) on an unweighted graph is true?

A. It always finds a minimum spanning tree.

BFS builds a shortest-path tree, not a minimum spanning tree (which depends on edge weights).

B. It computes shortest-path distances (in number of edges) from the start vertex.

On unweighted graphs, BFS explores the graph in increasing order of distance (in edge count) from the source.

C. It visits vertices in non-decreasing order of their labels.

BFS has no concern for vertex labels—only layer-by-layer expansion.

D. It requires a stack as its primary auxiliary data structure.

BFS uses a queue. A stack is typically used in DFS.

⑤ In a hash table using chaining with n elements and m buckets, the load factor $\alpha = n/m$ represents:

A. The expected number of elements per bucket

The load factor $\alpha = n/m$ directly measures the average length of each chain.

B. The maximum number of elements in any bucket

This refers to the maximum chain length, not the average.

C. The probability of a collision on any insertion

While related to α, it is not the definition of the load factor.

D. The ratio of filled buckets to total buckets

That’s bucket utilization, not load factor.

⑥ Which of the following is not a guaranteed property of a binary max-heap with n elements?

A. It is a complete binary tree.

This is a defining property of heaps.

B. Each parent node’s key $\geq$ its children’s keys.

This is the heap-order property of a max-heap.

C. Its height is $\lfloor \log_2 n \rfloor$.

A complete binary tree with n nodes has height $\lfloor \log_2 n \rfloor$.

D. All leaves are at the same depth.

Complete binary trees may have leaves at two adjacent levels.

⑦ What is the worst-case time complexity of Kruskal’s algorithm on a graph with V vertices and E edges, assuming edges are pre-sorted in O(E log E) and union-find with path compression is used?

A. O(E log E)

B. O(V log V + E)

This is the time for Prim’s algorithm using a binary heap.

C. O(E α(V))

With pre-sorted edges, Kruskal’s run-time is dominated by the α(V)-amortised union–find operations, giving Θ(E α(V))

D. $O(V^2)$

This is the time complexity of Prim’s algorithm using an adjacency matrix.

⑧ Prim's algorithm for finding a minimum spanning tree relies on which fundamental property of MSTs?

A. Cut Property

Prim always selects the minimum-weight edge crossing the current tree boundary, illustrating the cut property.

B. Cycle Property

States that the heaviest edge in any cycle cannot be in the MST; not used in Prim.

C. Subtree Property

No such formal axiom.

D. Optimal Substructure

While MST problems have this, Prim's step-by-step selection is more directly based on the cut property.

⑨ What is the average-case time complexity of searching for a key in a balanced binary search tree with n nodes?

A. O(1)

Not possible for trees; would require a hash table.

B. O(log n)

Balanced BSTs maintain height Θ(log n), so searches follow that path length.

C. O(n)

This applies to unbalanced BSTs or worst-case skewed trees.

D. O(n log n)

More typical of sorting or building a tree, not searching.

⑩ Which graph representation always uses $\Theta(V^2)$ space for a graph with V vertices, regardless of the number of edges?

A. Adjacency list

For a sparse graph (few edges), the space is much less than V^2.

For a dense graph (many edges), it can approach V^2, but not always.

B. Adjacency matrix

C. Edge list

O(E), where E is the number of edges. If the graph is sparse, this can be much less than V^2.

D. Incidence matrix

This is similar to adjacency list or edge list in terms of growth with the number of edges, not always V^2.

2. T/F problems

(1) Memoization in recursive algorithms trades increased space usage for reduced running time by caching and reusing intermediate results.

(2) A red‑black tree with n internal nodes has height at most $2 \cdot \lfloor \log_2(n) \rfloor$

(3) You can perform binary search in O(log n) time on any sorted data structure—regardless of whether it supports random access.

(4) Extracting the maximum element from a max-heap of size *n* takes O(log n) time.

(5) A tree is an undirected, connected, acyclic graph.

2. Solutions

(1) Memoization in recursive algorithms trades increased space usage for reduced running time by caching and reusing intermediate results. **T**

1. Memoization stores the results of expensive function calls (subproblem solutions) in a lookup table.

2. Subsequent calls with the same parameters fetch the result in O(1) rather than recomputing.

3.* This eliminates exponential duplicate work at the cost of O(number of distinct subproblems) extra space.

(2) A red–black tree with n internal nodes has height at most $2\cdot\lfloor \log_2(n)\rfloor$ **F**

1. By red–black properties, every path from root to leaf has at most twice as many nodes as the shortest path (since red nodes can only appear between blacks).

2. The shortest possible black-height is $\lfloor \log_2(n+1)\rfloor$.

3. Therefore the overall height $\leq 2\cdot\lfloor \log_2(n+1)\rfloor$ rather than $2\cdot\lfloor \log_2(n)\rfloor$.

(3) You can perform binary search in O(log n) time on any sorted data structure—regardless of whether it supports random access. **F**

1. Binary search repeatedly jumps to the middle—requiring random (direct) access.

2. Structures like linked lists only support sequential access—finding the middle takes O(n).

3. So binary search runs in O(n) on a linked list, not O(log n).

(4) Extracting the maximum element from a max-heap of size n takes O(log n) time. **T**

1. Swap root (max) with the last element—O(1).

2. Heapify down from the root to restore heap order; each level you may swap—height $\approx \log_2 n$—so O(log n).

(5) A tree is an undirected, connected, acyclic graph. **T**

3. Short answer questions

(a) Please compare merge sort algorithm and heap sort algorithm, including their core ideas, stability, time and space complexity.

(b) Describe the core idea of radix sort.

(c) What is the basic data structure of DFS and BFS?

(d) Please implement stack with queues.

(e) What is a hash collision? What methods can we use when it occurs? And list the pros and cons of these methods.

3. Solutions

(a) Merge Sort vs. Heap Sort

Feature	Merge Sort	Heap Sort
Core Idea	Divide-and-conquer: recursively split and merge sorted subsequences	Build a max-heap, repeatedly swap root with end and reheapify
Stability	Stable	Unstable
Time Complexity	Best/Average/Worst: O(n log n)	Best/Average/Worst: O(n log n)
Space Complexity	O(n) extra space	O(1) in-place

(b) Core Idea of Radix Sort

Sort multiple times based on each digit using a stable sorting algorithm (commonly counting sort):

LSD (Least Significant Digit first): sort from the least to the most significant digit

MSD (Most Significant Digit first): sort recursively from the most significant digit

(c) DFS and BFS

DFS: Stack

BFS: Queue

(d) Implementing a Stack Using Two Queues

Key Insight

- A stack always pops the most recently pushed element.
- A queue always pops the least recently pushed element.

To reconcile these, we'll re-order elements on each push so that the newest element always sits at the front of our main queue q1.

Then:

- pop() can be a simple pop_front() from q1.
- top() is just q1.front().

Thus our invariant is:

After every push, q1.front() holds the stack's top element, and the rest of q1 is in stack order behind it.

Designing push(x)

1. **Enqueue** the new element x into our empty auxiliary queue q2.
2. **Drain** every existing element from q1 into q2 (by repeatedly:
   - q2.push_back(q1.front())
   - q1.pop_front()

).

3. Now **q2** holds [x, old_top, next, …, bottom] in that exact order.
4. **Swap** the roles of q1 and q2 (so q1→q2, q2→q1).
   - Now q1 is ready for future pops, and q2 is empty for the next push.
5. Because we moved all old elements behind the new one, q1.front() is the newest element—exactly the stack's top.

Implementations:
In C++:

```
    void push(const T& x) {
        // q2 is auxiliary queue
        q2.push(x);
        // Move all elements from the main queue q1 to q2
        while (!q1.empty()) {
            q2.push(q1.front());
            q1.pop();
        }
        std::swap(q1, q2);
    }

    void pop() {
        if (q1.empty()) throw std::out_of_range("Stack is empty");
        q1.pop();
    }

    T& top() {
        if (q1.empty()) throw std::out_of_range("Stack is empty");
        return q1.front();
    }

    bool empty() const {
        return q1.empty();
    }

    size_t size() const {
        return q1.size();
```

In Python:

```python
    def __init__(self):
        self.q1 = deque()
        self.q2 = deque()

    def push(self, x):
        """
        Push element x onto stack.
        Steps:
        1. Enqueue x to q2.
        2. Move all elements from q1 to q2.
        3. Swap q1 and q2.
        Time complexity: O(n)
        """
        self.q2.append(x)
        while self.q1:
            self.q2.append(self.q1.popleft())
        # Swap the queues
        self.q1, self.q2 = self.q2, self.q1

    def pop(self):
        if not self.q1:
            raise IndexError("pop from an empty stack")
        return self.q1.popleft()

    def top(self):
        if not self.q1:
            raise IndexError("top from an empty stack")
        return self.q1[0]

    def empty(self):
        return not self.q1

    def size(self):
        return len(self.q1)
```

(e) Hash Collision and Solutions

Definition:

A hash collision occurs when **distinct** keys are mapped to **the same slot** by the hash function.

Resolution Strategies:

Chaining

Aspect	Description
Pros	- Simple to implement - Load factor can exceed 1
Cons	- Extra memory overhead due to pointers/linked lists - Performance degrades with long chains

Open Addressing

Variant	Pros	Cons
Linear Probing	- Simple implementation - Cache-friendly (contiguous memory access)	- Suffers from primary clustering
Quadratic Probing	- Reduces primary clustering	- Still has secondary clustering - Load factor must remain low
Double Hashing	- Best key distribution - Avoids clustering effectively	- More complex - Requires two good hash functions

Rehashing with Table Expansion

Aspect	Description
Pros	- Maintains low load factor - Prevents performance degradation
Cons	- Expensive: needs to rehash all keys - May cause latency spikes during resizing

4. Given a sequence of numbers [3, 17, 0, 18, 19, 0, 4, 5, 13, 12, 6, 7, 2].

   (a) Build an AVL tree by inserting the numbers in the given order.

   (b) Show the preorder, inorder, and postorder traversal of it.

4. Solutions

(a) Final AVL tree:

```
            12
          /     \
        5         17
      /   \      /   \
    3      6   13     18
   / \      \           \
  0   4      7           19
   \
    2
```

Details (each node is labeled "key (BF)"):

1. Insert 3, 0, 17, 18, 19

   // After inserting 19, node 17 goes RR-unbalanced

```
      3 (0)
      /  \
   0(0)   17(-2)
            \
            18(-1)
               \
               19(0)
```

→ rotation yields:

```
        3 (0)
        /   \
    0 (0)   18 (0)
            /   \
       17 (0)   19(0)
```

2. Insert 0, 4

```
   // Ignore repeat insertion 0.
  // After inserting 4, node 3 goes RL-unbalanced
           3 (–2)
           /    \
   0 (0)      18 (+1)
               /       \
      17 (+1)      19(0)
           /
   4 (0)
  →    rotation yields:
                     17(0)
                  /            \
             3(0)           18(-1)
            /     \                  \
        0(0)      4(0)             19(0)
```

3. Insert 5, 13

```
   // After inserting 13, node 4 goes RR-unbalanced
        17(+2)       // Don't rotate this node since we operate on the nearest
        /        \      // unbalanced node from new inserted node
   3(-1)      18(-1)
  /     \              \
0(0)      4(-2)        19(0)
            \
             5(-1)
                \
                 13(0)
```

```
→   rotation yields:
                17(+1)
                /     \
            3(-1)      18(-1)
           /    \          \
        0(0)    5(0)        19(0)
               /  \
            4(0)   13(0)
```

4. Insert 12

```
    // After inserting 12, node 3 goes RR-unbalanced
                17(+2)
                /     \
            3(-2)      18(-1)
           /    \          \
        0(0)    5(-1)       19(0)
               /    \
            4(0)    13(+1)
                    /
                 12(0)
→   rotation yields:
                17(+1)
                /       \
            5(0)         18(-1)
           /     \           \
        3(0)      13(+1)      19(0)
       /    \      /
    0(0)    4(0)  12(0)
```

5. Insert 6

// After inserting 6, node 13 goes LL-unbalanced

```
                17(+2)
              /         \
          5(-1)          18(-1)
        /        \            \
     3(0)       13(+2)      19(0)
   /     \         /
0(0)      4(0)  12(+1)
              /
          6(0)
```

→ rotation yields:

```
                    17(+1)
                  /           \
              5(0)            18(-1)
           /         \            \
        3(0)         12(0)       19(0)
      /    \        /    \
   0(0)   4(0)   6(0)   13(0)
```

6. Insert 7

// After inserting 7, node 17 goes LR-unbalanced

```
                 17(+2)
               /        \
            5(-1)        18(-1)
          /        \            \
     3(0)          12(0)          19(0)
   /    \        /    \
0(0)   4(+1)   6(0)    13(0)
                 \
                  7(0)
```

→ rotation yields:

```
                  12(0)
               /          \
            5(0)            17(-1)
          /    \           /      \
     3(0)       6(-1)    13(0)    18(-1)
   /    \          \                  \
0(0)    4(0)       7(0)               19(0)
```

7. Insert 2

```
                     12(+1)
                  /            \
              5(+2)          17(-1)
            /     \            /      \
       3(+1)   6(-1)   13(0)     18(-1)
      /       \          \                  \
 0(-1)      4(0)      7(0)               19(0)
   \
      2(0)
```

And it's the final answer.

(b)

Pre-order:	12, 5, 3, 0, 2, 4, 6, 7, 17, 13, 18, 19
In-order:	0, 2, 3, 4, 5, 6, 7, 12, 13, 17, 18, 19
Post-order:	2, 0, 4, 3, 7, 6, 5, 13, 19, 18, 17, 12

5. Given a sequence of numbers [10, 22, 31, 4, 15, 28, 17, 88, 59].

(a) Consider two initially empty hash tables T_1, T_2 of size 11 (indices 0 through 10). The primary hash function is $h_1(k) = k \bmod 11$. Please insert the sequence of integer keys into T_1 in the given order using $h_1(k)$. Collisions are resolved by linear probing in (a)

(b) $h_2(k) = 1 + (k \bmod 10)$. Please insert the sequence of integer keys into T_2 in the given order using $h_2(k)$. Collisions are resolved by double hashing in (b).

5. Solutions

(a)

// x (y) means the key being inserted hashes to index x, but x is already occupied by key y.

Key	$h_1(k)$	Probe sequence	Final index
10	10	10 → empty	10
22	0	0 → empty	0
31	9	9 → empty	9
4	4	4 → empty	4
15	4	4 (occupied by 4) → 5 (empty)	5
28	6	6 → empty	6
17	6	6 (occupied by 28) → 7 (empty)	7
88	0	0 (22) → 1 (empty)	1
59	4	4 (4) → 5 (15) → 6 (28) → 7 (17) → 8 (empty)	8

Index	0	1	2	3	4	5	6	7	8	9	10
Key	22	88	-	-	4	15	28	17	59	31	10

(b)

Key	$h_1(k)$	$h_2(k)$	Probe i=0	i=1	i=2	Final index
10	10	1	10 → empty			10
22	0	3	0 → empty			0
31	9	2	9 → empty			9
4	4	5	4 → empty			4
15	4	6	4 (4)	(4+6)%11=10 (10)	(4+2·6)%11=5	5
28	6	9	6 → empty			6
17	6	8	6 (28)	(6+8)%11=3		3
88	0	9	0 (22)	(0+9)%11=9 (31)	(0+2·9)%11=7	7
59	4	10	4 (4)	(4+10)%11=3 (17)	(4+2·10)%11=2	2

Index	0	1	2	3	4	5	6	7	8	9	10
Key	22	-	59	17	4	15	28	88	-	31	10

6. Given a sequence of numbers [4, 1, 3, 2, 16, 9, 10, 14, 8, 7]**(1-indexed)**.

(a) Build a Max Heap in given order(use the standard bottom-up Build-Max-Heap procedure). Show the final structure.

(b) Starting from the heap you built, delete the root node and show the final structure.

6. Solutions

(a)

We start at i = └ 10 / 2 ┘ = 5

i = 5

A[5] = 16; left = A[10] = 7; right out of range

16 > 7 → **no swap**

Array remains

i = 4

A[4] = 2; left = A[8] = 14; right = A[9] = 8

Largest is 14 → **swap A[4] ↔ A[8]**

[4, 1, 3, 14, 16, 9, 10, 2, 8, 7]

Heapify at index 8: no children → stop

i = 3

A[3] = 3; left = A[6] = 9; right = A[7] = 10

Largest is 10 → **swap A[3] ↔ A[7]**

[4, 1, 10, 14, 16, 9, 3, 2, 8, 7]

Heapify at index 7: no children → stop

i = 2

A[2] = 1; left = A[4] = 14; right = A[5] = 16

Largest is 16 → **swap A[2] ↔ A[5]**

[4, 16, 10, 14, 1, 9, 3, 2, 8, 7]

Heapify at index 5:

A[5] = 1; left = A[10] = 7

7 > 1 → **swap A[5] ↔ A[10]**

[4, 16, 10, 14, 7, 9, 3, 2, 8, 1]

Heapify at index 10: no children → stop

i = 1 (1-indexed)

A[1] = 4; left = A[2] = 16; right = A[3] = 10

Largest is 16 → **swap A[1] ↔ A[2]**

[16, 4, 10, 14, 7, 9, 3, 2, 8, 1]

Heapify at index 2:

A[2] = 4; left = A[4] = 14; right = A[5] = 7

Largest is 14 → **swap A[2] ↔ A[4]**

[16, 14, 10, 4, 7, 9, 3, 2, 8, 1]

Heapify at index 4:

A[4] = 4; left = A[8] = 2; right = A[9] = 8

Largest is 8 → **swap A[4] ↔ A[9]**

[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]

Heapify at index 9: no children → stop

Resulting Max-Heap

[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]

```
              16
          /         \
        14           10
     /      \       /  \
    8        7     9    3
  /  \     /
 2    4   1
```

(b)

Starting array:

[16, 14, 10, 8, 7, 9, 3, 2, 4, 1]

Step 1:

Swap A[1] and A[10],

[1, 14, 10, 8, 7, 9, 3, 2, 4, 16]

Then extracte root = 16

Step 2:

A[1] = 1; left = A[2] = 14; right = A[3] = 10 → **swap A[1] ↔ A[2]**

[14, 1, 10, 8, 7, 9, 3, 2, 4]

A[2] = 1; left = A[4] = 8; right = A[5] = 7 → **swap A[2] ↔ A[4]**

[14, 8, 10, 1, 7, 9, 3, 2, 4]

A[4] = 1; left = A[8] = 2; right = A[9] = 4 → **swap A[4] ↔ A[9]**

[14, 8, 10, 4, 7, 9, 3, 2, 1]

Heapify at index 9: no children → stop

Final Heap

[14, 8, 10, 4, 7, 9, 3, 2, 1]

```
              14
           /      \
         8          10
      /   \        /  \
     4     7      9    3
   /  \
  2    1
```

7. Design an algorithm to find LCA of two nodes n1 and n2. Both n1 and n2 are nodes in the same BST, and their keys are distinct.
   (a) Show your code or pseudocode.
   (b) Analyze the time complexity of your algorithm.

7. Approach

Key Insight

If both target keys lie **to the left** of the current node, then their LCA must lie in the left subtree.

If both lie **to the right**, the LCA must lie in the right subtree.

Otherwise—the moment one key is ≤ current key and the other ≥ current key—the current node "splits" the two targets, so it is their LCA.

Visually:

```
                    ┌──────────┐
                    │   cur    │
                    │ key = K  │
                    └──────────┘

                 /                  \
        targets both <K      targets both >K
           ↓ left                ↓ right
```

Once we reach the split (one ≤ K ≤ the other), we stop.

Step-by-Step Algorithm Design

1. Initialize a pointer cur = root

2. Loop while **cur** is not null:

    Case 1: n1.key < cur.key and n2.key < cur.key

        → Both targets lie left ⇒ set **cur = cur -> left**

    Case 2: n1.key > cur.key and n2.key > cur.key

        → Both lie right ⇒ set **cur = cur ->right**

    Else:

        → We have found the split point ⇒ return cur

3. If we ever exit the loop without returning, the nodes were not both in the tree ⇒ return nullptr.

Pseudocode

```
function LCA_BST(root, n1, n2):
    cur ← root
    while cur ≠ null:
        if n1.key < cur.key AND n2.key < cur.key:
            cur ← cur.left
        else if n1.key > cur.key AND n2.key > cur.key:
            cur ← cur.right
        else:
            return cur
    return null
```

Solutions

(a)

In C++:

```cpp
struct Node {
    int key;
    Node *left, *right;
    Node(int k) : key(k), left(nullptr), right(nullptr) {}
};

Node* LCA_BST(Node* root, Node* n1, Node* n2) {
    Node* cur = root;
    while (cur) {
        // both targets smaller → go left
        if (n1->key < cur->key && n2->key < cur->key) {
            cur = cur->left;
        }
        // both targets greater → go right
        else if (n1->key > cur->key && n2->key > cur->key) {
            cur = cur->right;
        }
        // otherwise cur splits the paths → it's the LCA
        else {
            return cur;
        }
    }
    return nullptr;
}
```

In Python:

```python
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

def lca_bst(root: Node, n1: Node, n2: Node) -> Node:
    cur = root
    while cur:
        if n1.key < cur.key and n2.key < cur.key:
            cur = cur.left
        elif n1.key > cur.key and n2.key > cur.key:
            cur = cur.right
        else:
            return cur
    return None
```

(b)

Time: $O(h)$, where h = tree height

Balanced BST $\Rightarrow$ $h = O(\log n)$

Worst-case (degenerate) $\Rightarrow$ $h = O(n)$

8. Design an algorithm to find the k-th largest number in a large unsorted array with n elements (n > k).

(a) Show your code or pseudocode.

(b) Analyze the complexity of your algorithm.

8. Approach

Min-Heap

Key Insight

- We don’t need the full order of all elements—just the top .
- Maintain a min-heap of the largest seen so far.
  - The heap root is the smallest among those top .
  - Whenever we see a new element larger than the root, it belongs in the top —we replace.

Step-by-Step Design

1. Initialize an empty min-heap H
2. Scan each element in the array:

   Insert into H

   If H.size > k, pop the smallest (the root)
3. After one pass, **H** contains exactly the **k** largest elements, and its root is the k-th largest.
4. Return H.top( )

Pseudocode

```
function kth_largest_minheap(arr, k):
    // Maintain a min-heap of size ≤ k
    heap ← empty min-heap
    for x in arr:
        heap.push(x)
        if heap.size() > k:
            heap.pop()        // remove the smallest
    // At the end, the root of the heap is the k-th largest
    return heap.top()
```

Complexity Analysis

Each of the insertions costs O(log k) in a heap of size ≤ k.

Total time O(n log k), space O(k).

Quickselect (Partition-Based)

Key Insight

Quickselect is a selection algorithm derived from quick-sort’s partitioning:

1. Choose a pivot
2. Partition the array so that:
    - Left side < pivot
    - Pivot in its final position
    - Right side > pivot
3. Let = pivot’s final index
    - If m = t (target index), pivot is the answer
    - If t < m , recurse on left
    - If t > m, recurse on right
    - Here **target index** t = n - k (zero-based) for the k-th largest.

Step-by-Step Design

```
1. Compute t = n - k
2. Define quickselect(A, lo, hi, t)    // lower index(left boundary), higher index
       If lo == hi, return A[lo]            // (right boundary
       Pick random pivot index p in [lo..hi]    // random pivot to avoid worst case
       m = partition(A, lo, hi, p)          // p = pivot index
       If m == t, return A[m]
       Else if t < m, recurse on [lo..m-1]
       Else, recurse on [m+1..hi]
3. Call quickselect(A, 0, n-1, t) and return its result.
4.* In practice, you can rewrite the algorithm using iteration (a while loop)
       instead of recursion to avoid stack overflow.
```

Pseudocode

```
function kth_largest_quickselect(arr, k):
    # convert “k-th largest” to zero-based target index
    target_index ← arr.length - k
    return quickselect(arr, 0, arr.length - 1, target_index)

function quickselect(A, lo, hi, t):
    if lo == hi:
        return A[lo]
    # 1. Pick a pivot at random
    p ← RANDOM(lo, hi)
    # 2. Partition around A[p], returns pivot final position m
    m ← partition(A, lo, hi, p)
    # 3. Recurse into the side containing t
    if t == m:
        return A[m]
    else if t < m:
        return quickselect(A, lo, m - 1, t)
    else:
        return quickselect(A, m + 1, hi, t)

function partition(A, lo, hi, p):
    pivot ← A[p]
    swap(A[p], A[hi])
    store ← lo
    for i from lo to hi - 1:
        if A[i] < pivot:
            swap(A[i], A[store])
            store ← store + 1
    swap(A[store], A[hi])
    return store
```

Complexity Analysis

Average Case:

The expected total number of elements processed across all partition steps is O(n), because each recursive call works on a progressively smaller subarray (geometric shrinkage).

Therefore, the average-case time complexity is O(n).

Recursion Depth:

The recursion depth is O(log n) on average, as each partition typically splits the problem roughly in half.

However, each deeper recursive call handles a smaller subarray, so the total work remains O(n).

Worst Case:

If the pivot choices are consistently poor (e.g., always choosing the smallest/largest element),

the algorithm can degrade to O(n^2) time complexity.

Space Complexity:

The algorithm uses O(1) extra space, plus O(log n) stack space on average due to recursion (worst case O(n)).

8. Solutions

(a)

In C++

```
// MinHeap Solution
int findKthLargestMinHeap(const vector<int>& nums, int k) {
    priority_queue<int, vector<int>, greater<int>> minHeap;
    for (int x : nums) {
        minHeap.push(x);
        if ((int)minHeap.size() > k)
            minHeap.pop();
    }
    return minHeap.top();
}


// Quickselect Solution
int partition(vector<int>& nums, int lo, int hi, int pivotIndex) {
    int pivot = nums[pivotIndex];
    swap(nums[pivotIndex], nums[hi]);
    int store = lo;
    for (int i = lo; i < hi; ++i) {
        if (nums[i] < pivot) {
            swap(nums[store], nums[i]);
            ++store;
        }
    }
    swap(nums[store], nums[hi]);
    return store;
}

int quickselect(vector<int>& nums, int lo, int hi, int kSmallest) {
    if (lo == hi)
        return nums[lo];
    int pivotIndex = lo + rand() % (hi - lo + 1);
    pivotIndex = partition(nums, lo, hi, pivotIndex);

    if (kSmallest == pivotIndex)
        return nums[kSmallest];
    else if (kSmallest < pivotIndex)
        return quickselect(nums, lo, pivotIndex - 1, kSmallest);
    else
        return quickselect(nums, pivotIndex + 1, hi, kSmallest);
}
```

```
int findKthLargestQuickselect(vector<int> nums, int k) {
    int n = nums.size();
    // convert k-th largest to zero-based index for "k-th smallest"
    int target = n - k;
    return quickselect(nums, 0, n - 1, target); }
```

In Python:

```
# Min-Heap Solution
def kth_largest_minheap(nums, k):
    min_heap = []
    for x in nums:
        heapq.heappush(min_heap, x)
        if len(min_heap) > k:
            heapq.heappop(min_heap)
    return min_heap[0]

# Quickselect Solution
def partition(nums, lo, hi, pivot_idx):
    pivot = nums[pivot_idx]
    nums[pivot_idx], nums[hi] = nums[hi], nums[pivot_idx]
    store = lo
    for i in range(lo, hi):
        if nums[i] < pivot:
            nums[store], nums[i] = nums[i], nums[store]
            store += 1
    nums[store], nums[hi] = nums[hi], nums[store]
    return store

def quickselect(nums, lo, hi, k_smallest):
    if lo == hi:
        return nums[lo]
    pivot_idx = random.randint(lo, hi)
    pivot_idx = partition(nums, lo, hi, pivot_idx)
    if k_smallest == pivot_idx:
        return nums[k_smallest]
    elif k_smallest < pivot_idx:
        return quickselect(nums, lo, pivot_idx - 1, k_smallest)
    else:
        return quickselect(nums, pivot_idx + 1, hi, k_smallest)

def kth_largest_quickselect(nums, k):
    n = len(nums)
    return quickselect(nums[:], 0, n - 1, n - k)
```

(b)

MinHeap:

Each of the insertions costs O(log k) in a heap of size $\leqslant$ k.

Total time O(n log k), space O(k).

Quickselect:

Average-case time: O(n), since the total number of elements processed over all partition steps is O(n) in expectation.

Worst-case time: O(n^2), if the pivot always partitions poorly.

Recursion depth / stack space: O(log n) on average, O(n) in the worst case.

Extra space: O(1) aside from the recursion stack