Data Structures

SCIS 313: Data Structures and Algorithm Analysis

Prof. Antonio Khalil Moretti
Spelman College

Reference: Complexity Summary

Structure	Access	Search	Insert	Delete
Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Linked List	$O(n)$	$O(n)$	$O(1)^*$	$O(1)^*$
Stack / Queue	—	—	$O(1)$	$O(1)$
BST (balanced)	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Heap	—	$O(n)$	$O(\log n)$	$O(\log n)$
Hash Table (avg)	$O(1)$	$O(1)$	$O(1)$	$O(1)$

*If you have a pointer to the location.

Problems

Problem 1: Choosing the Right Structure

For each scenario below, identify the most appropriate data structure and briefly justify your choice in 1–2 sentences. Be specific about which operation(s) drive your decision.

(a) A web browser needs to support a “back” button that returns the user to the previously visited page.

(b) A hospital emergency room needs to always treat the most critical patient next, regardless of arrival order.

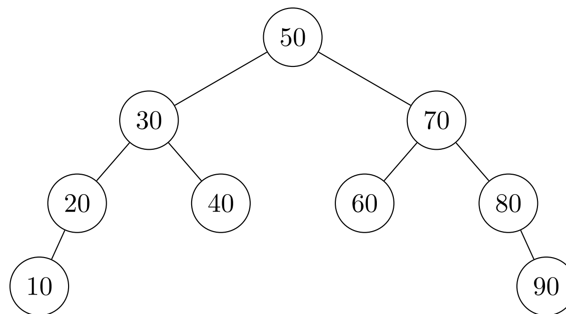
(c) A spell-checker needs to look up whether a word exists in a dictionary of 100,000 words as fast as possible. The dictionary never changes.

(d) A music streaming service needs to maintain a playlist where songs can be added or removed from either end in constant time.

(e) A compiler needs to match opening and closing parentheses, brackets, and braces in source code.

Problem 2: Tree Traversals

Consider the following binary search tree:



(a) Write the **inorder** traversal of this tree. What do you notice about the result?

(b) Write the **preorder** traversal of this tree.

(c) Write the **postorder** traversal of this tree.

(d) Write the **level-order** traversal of this tree.

(e) Suppose you wanted to make an exact copy of this tree. Which traversal order would you use, and why?

Hint

For part (e), think about what information you need to reconstruct a node — you need to create a node *before* you can attach its children. Which traversal visits the root before its children?

Problem 3: BST Operations

Start with an empty BST and insert the following values **in order**:

$$15, \quad 8, \quad 23, \quad 4, \quad 12, \quad 19, \quad 30$$

(a) Draw the resulting BST.

(b) What is the height of the tree? (Height = number of edges on the longest root-to-leaf path.)

(c) Now insert the values in sorted order instead: $4, 8, 12, 15, 19, 23, 30$. Draw the resulting BST. What is its height?

(d) What does part (c) tell you about the relationship between insertion order and BST performance? What is the worst-case search time for each tree?

Hint

For part (c), follow the BST insertion rule exactly: each new value goes left if it is smaller than the current node, right if it is larger. Notice what shape the tree takes when values arrive in sorted order.

Problem 4: Heaps

Consider the following array, which represents a complete binary tree:

```
[25, 16, 18, 9, 12, 6, 10]
```

(a) Draw the array as a complete binary tree. Is it a valid min-heap? If not, identify which node(s) violate the heap property.

(b) Apply heapify to build a valid min-heap. Show the state of the array after **each bubble-down step**. Which index do you start at, and why?

(c) Once you have a valid min-heap, insert the value **5**. Show the array after the insertion and after each bubble-up swap.

(d) Perform an extract-min on the heap from part (c). Show the array at each step of the bubble-down process.

Hint

For part (b): the last non-leaf node is at index $\lfloor n/2 \rfloor - 1$. With $n = 7$, that is index 2. Work *right to left* from there to index 0, bubbling down each node as needed.
For part (c): always append to the end of the array, then compare with the parent at index $\lfloor (i-1)/2 \rfloor$ and swap upward while the heap property is violated.

Problem 5: Hash Tables

Suppose you have a hash table of size $m = 7$ and the hash function $h(k) = k \bmod 7$.

(a) Insert the keys $\{23,\ 11,\ 30,\ 4,\ 17,\ 38\}$ into the hash table using **chaining**. Draw the resulting table, showing each chain.

(b) What is the load factor α after all insertions? What does this tell you about the expected search time?

(c) Now insert the same keys using **linear probing** (open addressing). Show which slot each key lands in, including any probing steps needed to resolve collisions.

(d) Compare your answers to (a) and (c). Which method placed more keys in their "natural" slot (i.e., $h(k)$ directly)? Which would you expect to perform better in practice at this load factor, and why?

Hint

For linear probing: if slot $h(k)$ is occupied, try $h(k)+1$, then $h(k)+2$, etc., wrapping around modulo m. For example, if $h(k) = 6$ and slot 6 is full, try slot 0 next.

Problem 6: Stacks and Queues

(a) Trace the following sequence of operations on an initially empty stack. After each operation, write the contents of the stack (top on the left):

`push(3), push(7), push(1), pop(), push(5), push(2), pop(), pop()`

(b) A classic interview question: implement a **queue** using **two stacks**. Describe in plain English how `enqueue` and `dequeue` would work. What is the amortized time complexity of each operation?

(c) Consider the string `"({[()]})"`. Describe how you would use a stack to verify that the brackets are balanced. Trace your algorithm on this string, showing the stack contents after each character.

Hint

For part (b): think of one stack as the “inbox” and one as the “outbox.” When does it make sense to transfer elements from one to the other?
For part (c): push opening brackets onto the stack. When you see a closing bracket, check whether the top of the stack is the matching opener.

Problem 7 (LeetCode)**: Two Sum**

LeetCode

LeetCode #1 — Two Sum (*Easy*)
`https://leetcode.com/problems/two-sum/`

Given an array of integers `nums` and an integer `target`, return the indices of the two numbers that add up to `target`. You may assume exactly one solution exists and you may not use the same element twice.

(a) Describe a naïve $O(n^2)$ solution. What data structure does it use (implicitly or explicitly)?

(b) Describe an $O(n)$ solution using a hash table. How does the hash table help? Trace your algorithm on the input `nums = [3, 2, 4]`, `target = 6`.

(c) What is the space complexity of your $O(n)$ solution? Is there a trade-off between time and space here?

Hint

For part (b): for each element `x`, you are looking for `target - x`. Before processing `x`, check whether `target - x` is already in the hash table. If not, store `x` and its index.

Problem 8 (LeetCode): Kth Largest Element

LeetCode

LeetCode #215 — Kth Largest Element in an Array (*Medium*)
`https://leetcode.com/problems/kth-largest-element-in-an-array/`

Given an integer array `nums` and an integer k, return the k-th largest element in the array (not the k-th distinct element).

(a) What is the simplest approach? Sort the array and return the k-th element from the end. What is its time complexity?

(b) Describe a more efficient $O(n \log k)$ solution using a **min-heap** of size k. Walk through the key idea: what invariant does the heap maintain as you process each element?

(c) Trace your heap-based algorithm on `nums = [3, 2, 1, 5, 6, 4]`, $k = 2$. Show the state of the heap after each element is processed.

(d) Why is $O(n \log k)$ better than $O(n \log n)$ when $k \ll n$? Give a concrete example with numbers.

Hint

For part (b): maintain a min-heap of the k largest elements seen so far. When you process a new element:

- If the heap has fewer than k elements, insert it.
- Otherwise, if the new element is larger than the heap's minimum, pop the minimum and insert the new element.

At the end, the root of the heap is the k-th largest element overall.

Submission Guidelines

For each problem, your solution should include:

- **Clear reasoning:** Don't just give an answer — explain why.
- **Diagrams where asked:** Draw trees and tables neatly and label them.
- **Step-by-step traces:** For heap and hash table operations, show the state after each step, not just the final result.
- **Complexity analysis:** When asked about time or space complexity, justify your answer — don't just state it.