Sample Problems and Solutions
Data Structures

[Algorithm analysis.]

1. [15 points] What is the order of complexity of the following block(s) of code? Show detailed steps to prove your answer.

```
CodeBlock-1($n$)
count ← 0
for $i \leftarrow n; i > 0; i \leftarrow i/4$ do
    for $j \leftarrow 0; j < i; j \leftarrow j + 1$ do
        count ← count + 1
```

```
CodeBlock-2($n$)
count ← 0; $i \leftarrow 1$
while $i \leq n$ do
    $j \leftarrow n$
    while $j \geq 1$ do
        count ← count + 1; $j \leftarrow j/3$
    $i \leftarrow i \times 2$
```

Solution:

Time for CodeBlock-1:

$= \sum_{i \in \left\{ \frac{n}{4^0}, \frac{n}{4^1}, \frac{n}{4^2}, \ldots, \frac{n}{4^{\lfloor \log_4 n \rfloor}} \right\}} \sum_{j \in \{0,1,2,\ldots,i\}} 1$

$= \sum_{i \in \left\{ \frac{n}{4^0}, \frac{n}{4^1}, \frac{n}{4^2}, \ldots, \frac{n}{4^{\lfloor \log_4 n \rfloor}} \right\}} (i + 1)$

$= \sum_{i \in \left\{ \frac{n}{4^0}, \frac{n}{4^1}, \frac{n}{4^2}, \ldots, \frac{n}{4^{\lfloor \log_4 n \rfloor}} \right\}} i + \sum_{i \in \left\{ \frac{n}{4^0}, \frac{n}{4^1}, \frac{n}{4^3}, \ldots, \frac{n}{4^{\lfloor \log_4 n \rfloor}} \right\}} 1$

$= \left(\frac{n}{4^0} + \frac{n}{4^1} + \frac{n}{4^2} + \cdots + \frac{n}{4^{\lfloor \log_4 n \rfloor}} \right) + \log_4 n$

$= \Theta(n) + \Theta(\log n)$ (using geometric series sum)

$= \Theta(n)$

Time for CodeBlock-2:

$= \sum_{i \in \left\{ 2^0, 2^1, 2^2, \ldots, 2^{\lfloor \log n \rfloor} \right\}} \sum_{j \in \left\{ \frac{n}{3^0}, \frac{n}{3^1}, \frac{n}{3^2}, \ldots, \frac{n}{3^{\lfloor \log_3 n \rfloor}} \right\}} 1$

$= \sum_{i \in \left\{ 2^0, 2^1, 2^2, \ldots, 2^{\lfloor \log n \rfloor} \right\}} 1 \times \sum_{j \in \left\{ \frac{n}{3^0}, \frac{n}{3^1}, \frac{n}{3^2}, \ldots, \frac{n}{3^{\lfloor \log_3 n \rfloor}} \right\}} 1$ (independent sums)

$= (\lfloor \log_2 n \rfloor + 1) \times (\lfloor \log_3 n \rfloor + 1)$

$= \Theta(\log n) \times \Theta(\log n)$

$= \Theta\left(\log^2 n\right)$

2. [10 points] Write the following functions $f(n)$ in the form $f(n) \in \Theta(g(n))$, where $g(n)$ is the simplified function. (i) $f(n) = n^{1/\log n}$, (ii) $f(n) = \log \sqrt{4^n}$, (iii) $f(n) = 2^{\log_4 n}$, (iv) $f(n) = 7^{\log_{49 \times 49} n}$, and (v) $f(n) = (\log \log n)^{1/\log \log \log n}$

Solution:

Please add the reasoning for individual steps

1. $f(n) = n^{1/\log n} = (2^n)^{1/\log n} = 2$ (because $n = 2^{\log_2 n}$)
   $= \Theta(1)$
2. $f(n) = \log\sqrt{4^n} = \log 4^{\frac{n}{2}} = \log(2^2)^{\frac{n}{2}} = \log 2^n$
   $= \Theta(n)$
3. $f(n) = 2^{\log_4 n} = 2^{\frac{\log n}{\log 4}} = 2^{\frac{\log n}{2}} = (2^{\log n})^{1/2} = n^{1/2} = \sqrt{n}$
   $= \Theta(\sqrt{n})$
4. $f(n) = 7^{\log_{49\times 49} n} = 7^{\log_{7^4} n} = n^{\log_{7^4} 7} = n^{\frac{\log_7 7}{\log_7 7^4}} = n^{\frac{\log_7 7}{4\log_7 7}} = n^{\frac{1}{4}}$
   $= \Theta\left(n^{1/4}\right)$
5. $f(n) = (\log\log n)^{1/\log\log\log n} = \left(2^{\log\log\log n}\right)^{1/\log\log\log n} = 2$
   $= \Theta(1)$

3. [10 points] Rank the following functions by order of growth. That is, arrange the functions in asymptotically nondecreasing order using $\prec$ and $\asymp$ relations. Give reasons for your answers. Assume that the log function has base 2 unless explicitly mentioned otherwise.

   $2^{\sqrt{2\log n}}$, $(\log n)!$, $\log^2 n$, $(\log n)^{\log n}$, $\log(n!)$

4. [20 points] Assume that you want to implement the following ADT using several different data structures. Simply mention the worst-case time complexity for different methods using different data structures given in an empty table. Provide the complexities using $\Theta()$ notation wherever feasible and use $\mathcal{O}$ notation in the remaining places. Give the amortized complexity for dynamic array implementations and average case complexity for hash table implementations assuming that the load factor is small and the hash function distributes keys uniformly. The symbols $\star$ represents amortized case complexity and $\dagger$ represents average/expected case complexity.

   Solution: See Tables 1 and 2.

Operation	AdjacencyList	AdjacencyMatrix
CheckEdgeExistence(u, v)	$\mathcal{O}(\text{degree}(u))$	$\Theta(\lvert 1\rvert)$
GetNeighbors(v)	$\mathcal{O}(\text{degree}(u))$	$\Theta(\lvert V\rvert)$
Degree(v)	$\Theta(1)$	$\mathcal{O}(\lvert V\rvert)$
Traversal (BFS/DFS)	$\Theta(\lvert V\rvert + \lvert E\rvert)$	$\Theta(\lvert V\rvert^2)$
ListAllEdges	$\Theta(\lvert V\rvert + \lvert E\rvert)$	$\Theta(\lvert V\rvert^2)$
Space	$\Theta(\lvert V\rvert + \lvert E\rvert)$	$\Theta(\lvert V\rvert^2)$

Table 1: Time complexities for different operations on graphs.

Data Structure	Operation 1	Operation 2	Operation 3	Operation 4
Basic Data Structures	AddFirst	AddLast	RemoveFirst	RemoveLast
DynamicArray	$\Theta(n)$	$\Theta(1)^\star$	$\Theta(n)$	$\Theta(1)^\star$
SinglyLinkedList	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
CircularlySinglyLinkedList	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
DoublyLinkedList	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Stack ADT	Push	Pop	Top	
DynamicArray	$\Theta(1)^\star$	$\Theta(1)^\star$	$\Theta(1)$	
SinglyLinkedList	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	
CircularlySinglyLinkedList	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	
DoublyLinkedList	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	
Queue ADT	Enqueue	Dequeue	Front	
CircularDynamicArray	$\Theta(1)^\star$	$\Theta(1)^\star$	$\Theta(1)$	
SinglyLinkedList	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	
CircularlySinglyLinkedList	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	
DoublyLinkedList	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	
Deque ADT	AddFirst	AddLast	RemoveFirst	RemoveLast
CircularDynamicArray	$\Theta(1)^\star$	$\Theta(1)^\star$	$\Theta(1)^\star$	$\Theta(1)^\star$
SinglyLinkedList	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
CircularlySinglyLinkedList	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
DoublyLinkedList	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Sorted Set ADT	Add	Remove	Search	SortedOrder
DynamicArray	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\Theta(n\log n)$
SortedDynamicArray	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\Theta(n)$
BinarySearchTree	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\Theta(n)$
(2-4)-Tree	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\Theta(n)$
B-Tree	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\Theta(n)$
RedBlack-Tree	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\Theta(n)$
AVL-Tree	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\Theta(n)$
Set ADT	Add	Remove	Search	
DynamicArray	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	
SortedDynamicArray	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	
HashTable-Chaining-List	$\Theta(1)^\dagger$	$\Theta(1)^\dagger$	$\Theta(1)^\dagger$	
HashTable-LinearProbing	$\Theta(1)^\dagger$	$\Theta(1)^\dagger$	$\Theta(1)^\dagger$	
HashTable-QuadraticProbing	$\Theta(1)^\dagger$	$\Theta(1)^\dagger$	$\Theta(1)^\dagger$	
PriorityQueue ADT	Add	RemoveMin	Minimum	
SinglyLinkedList	$\Theta(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	
SortedSinglyLinkedList	$\mathcal{O}(n)$	$\Theta(1)$	$\Theta(1)$	
MinHeap-DynamicArray	$\mathcal{O}(\log n)^\star$	$\mathcal{O}(\log n)^\star$	$\Theta(1)$	
MinHeap-LinkedTree	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\Theta(1)$	

Table 2: Time complexities for different operations on various data structures.

[Summarizing data structures.]

1. [0 points] Write a beautiful poem on data structures

   Solution: ChatGPT: Here it is.

Coding the Invisible: A Journey Through Data Structures

Introduction
To hold and shape the thoughts we code,
We need smart ways to share the load.
Data structures show the way—
To store, to search, to build, to play.
From lines and trees to queues that wait,
They help machines communicate.

Array
In rows, each item knows its place,
Fixed in size, in tight embrace.
Fast to find by numbered call,
But shifting things can stall it all.
(Used in: images, matrices, and storing fixed lists.)

Singly linked list
One by one, they point ahead,
Each node links what's just been said.
Add with ease, but searching's slow—
You walk the chain to make it flow.
(Used in: music players, undo steps, and low-memory storage.)

Circularly linked list
No clear start and no true end,
It loops around just like a friend.
Great for games and endless tracks,
It cycles on and never slacks.
(Used in: multiplayer games, round-robin tasks.)

Doubly linked list
With two hands reaching front and back,
It makes its moves on any track.
Quick to add or take away,
From either side, it clears the way.
(Used in: web browsers (back/forward), LRU caches.)

Stack
Last in, first out—it's how it goes,

Like plates in piles or nested shows.
You push it in, then pop it out,
A helpful friend in code, no doubt.
(Used in: undo features, parsing code, call stacks.)

Queue

One joins the line, one leaves from start,
It moves like people in a mart.
First come, first served—it's fair and clear,
The queue stays calm, both far and near.
(Used in: printer tasks, customer service lines, CPUs.)

Deque

Like queue and stack, but more than both,
It adds and pulls from either growth.
From front or end, you're free to choose,
It gives more ways, so less you lose.
(Used in: sliding window problems, task scheduling.)

Binary search tree (BST)

Left for less and right for more,
Each node unlocks a sorted door.
To add or search, just take a dive,
The structure helps the values thrive.
(Used in: databases, auto-suggestion tools.)

Balanced search tree

But if one side grows far too tall,
The search slows down, not fast at all.
Balance keeps the height in check,
So time is saved, and things don't wreck.
(Used in: file systems, search engines, maps.)

Priority queue

Not first in line, but most in need,
The biggest task gets done with speed.
It sorts by weight, not time alone—
The urgent ones are quickly shown.
(Used in: emergency systems, CPU scheduling.)

Hash table

No rows, no chains, just keys that map,
A clever trick, a fast-hand clap.
Jump to data using codes,
No need to walk through all the loads.
(Used in: dictionaries, password storage, caches.)

Graph
With nodes and edges all around,
It models maps and friends we’ve found.
No single path, just many ways,
It shapes the web in modern days.
(Used in: GPS, social networks, internet links.)

Closing
Each form has logic, clean and smart,
A vital piece of coding art.
From simple lists to complex graphs,
They build the world in code and math.
So learn their ways, and you will see—
They hold the key to mastery.

[Arrays and lists.]

1. [10 points] Given a string $S[1 \ldots n]$, write an algorithm code to check whether the given string is a palindrome or not. A palindrome is a string that reads the same backward as forward. E.g. madam, malayalam, and racecar. Give the algorithm's time and space complexity.

```
IsPalindrome($S[1 \ldots n]$)
Write the pseudocode
```

Solution:

```
IsPalindrome($S[1 \ldots n]$)
for $i \leftarrow 1$ to $\lfloor n/2 \rfloor$ do
    if $S[i] \neq S[n-i+1]$ then
        return false
return true
```

This algorithm compares the characters at the beginning and end of the string, moving towards the center. If any characters don't match, it returns false. If it completes the loop, the string is a palindrome.

$\boxed{\text{Time} = \mathcal{O}(n), \text{Space} = \mathcal{O}(1)}$

2. [10 points] Given an array $a[1 \ldots n]$ of non-negative integers, write a recursive algorithm code to compute the greatest common divisor GCD$(a[1 \ldots n])$ of these n numbers. Assume that $gcd(x, y)$ computes the greatest common divisor of non-negative integers x and y.

```
GCD($a[1 \ldots n]$)
Write the pseudocode
```

Solution:

```
GCD($a[1 \ldots n]$)
if $n = 1$ then return $a[1]$
else return $gcd(a[n], \text{GCD}(a[1 \ldots (n-1)]))$
```

The recursive algorithm computes the GCD of the last element and the GCD of the rest of the array. The base case is when there's only one element, in which case the GCD is the element itself. The time complexity is $\mathcal{O}(n \cdot \log M)$, where M is the maximum value in the array, due to the recursive nature of the GCD function.

$\boxed{\text{Time} = \mathcal{O}(n \log M), \text{Space} = \mathcal{O}(n)}$

[Stacks, queues, and deques.]

1. [10 points] Fill Table 3 with the contents of the data structures after executing different operations starting from the initial configuration. Note that when you remove an element from a data structure, you return the removed element to the caller. You do not need to worry about the data overflow problem. The top element in the stack, the first element in the queue, and the first element in the deque are highlighted using a box.

   Solution: See Table 4.

Operations	Stack S	Queue Q	Deque D
Initially	$1, 4, \boxed{8}$	$\boxed{7}, 3, 6$	$\boxed{9}, 5, 2$
D.AddLast(S.Pop() $+$ Q.Dequeue())			
Q.Enqueue(D.RemoveFirst() $-$ S.Pop())			
S.Push(D.RemoveLast() $-$ Q.Dequeue())			
D.AddFirst(S.Pop() $-$ Q.Dequeue())			
D.AddFirst(D.RemoveFirst() $+$ D.RemoveLast())			

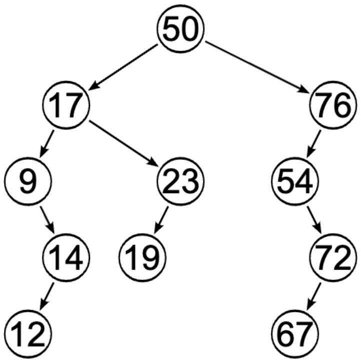
Table 3: Fill in the table with contents of the data structures.

Operations	Stack S	Queue Q	Deque D
Initially	$1, 4, \boxed{8}$	$\boxed{7}, 3, 6$	$\boxed{9}, 5, 2$
D.AddLast(S.Pop() $+$ Q.Dequeue())	1, 4	3, 6	9, 5, 2, 15
Q.Enqueue(D.RemoveFirst() $-$ S.Pop())	1	3, 6, 5	5, 2, 15
S.Push(D.RemoveLast() $-$ Q.Dequeue())	1, 12	6, 5	5, 2
D.AddFirst(S.Pop() $-$ Q.Dequeue())	1	5	6, 5, 2
D.AddFirst(D.RemoveFirst() $+$ D.RemoveLast())	1	5	8, 5

Table 4: Table filled with contents of the data structures.

[Trees and priority queues.]

1. [10 points] Traverse the following binary tree using the four traversal algorithms: Preorder traversal, inorder traversal, postorder traversal, and level-order (or breadth-first) traversal.



Solution

Inorder: 9, 12, 14, 17, 19, 23, 50, 54, 67, 72, 76

Preorder: 50, 17, 9, 14, 12, 23, 19, 76, 54, 72, 67

Postorder: 12, 14, 9, 19, 23, 17, 67, 72, 54, 76, 50

Level order: 50, 17, 76, 9, 23, 54, 14, 19, 72, 12, 67

2. Draw each of the following evolving data structures twice that result from the following operations in that order into an initially empty data structure.
   (a) [10 points] binary search tree
   insert [100, 10, 30, 50, 70, 40, 90, 60], draw tree, delete [30], insert [20, 30], delete [70], insert [80, 70], draw tree.
   (b) [10 points] (2,4) tree
   insert [100, 10, 30, 50, 70, 40], draw tree, insert [90, 60, 20, 80], draw tree.
   (c) [10 points] min heap (tree representation)
   insert [100, 10, 30, 50, 70, 40, 90, 60], draw tree, deletemin, deletemin, insert [20, 80], deletemin, draw tree.

   Solution: Please refer to Figures 1, 2, and 3.

3. [5 points] Check if the following array is a BST. Draw the tree and explain the reasoning.
   10, 5, 20, -, -, 9, 25

   Solution:

   Draw a tree diagram

   No. Because 9 is smaller than 10 yet it is on the right subtree of 10.

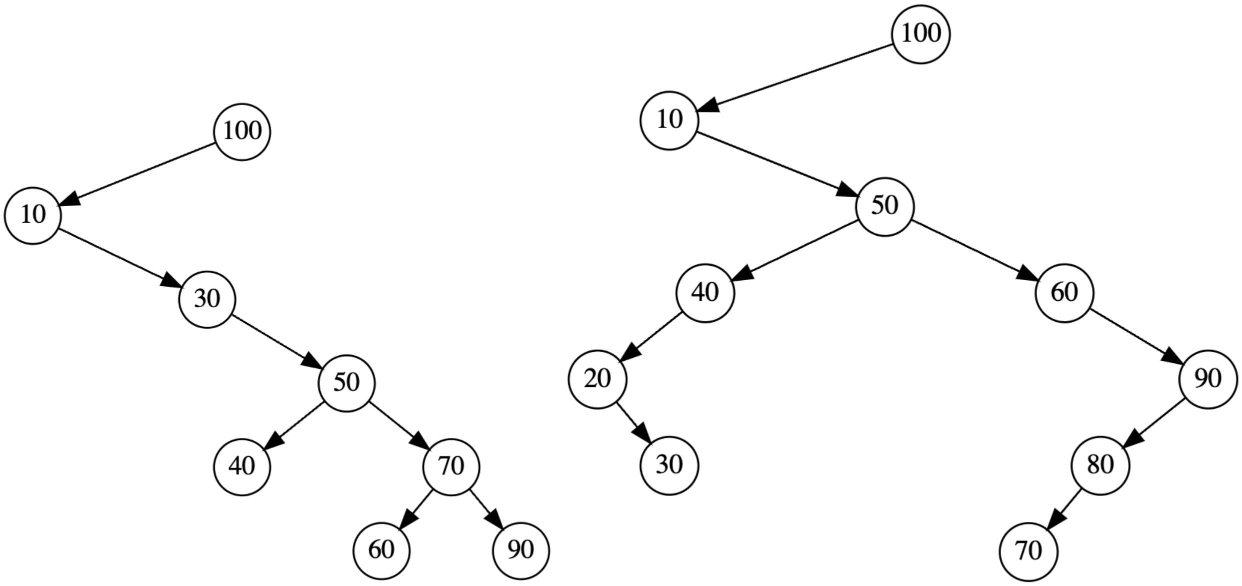


Figure 1: Binary search tree (BST).

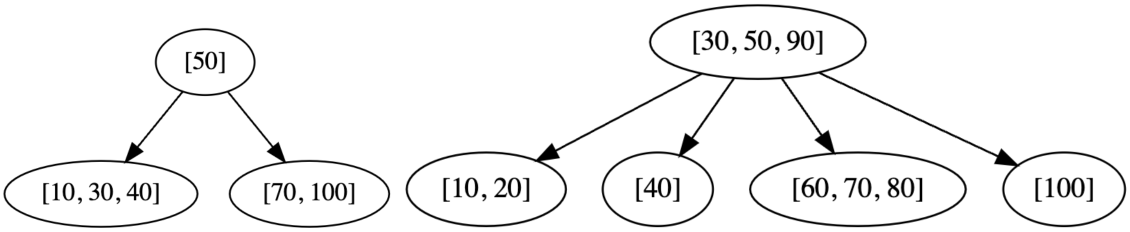


Figure 2: 2-4 tree.

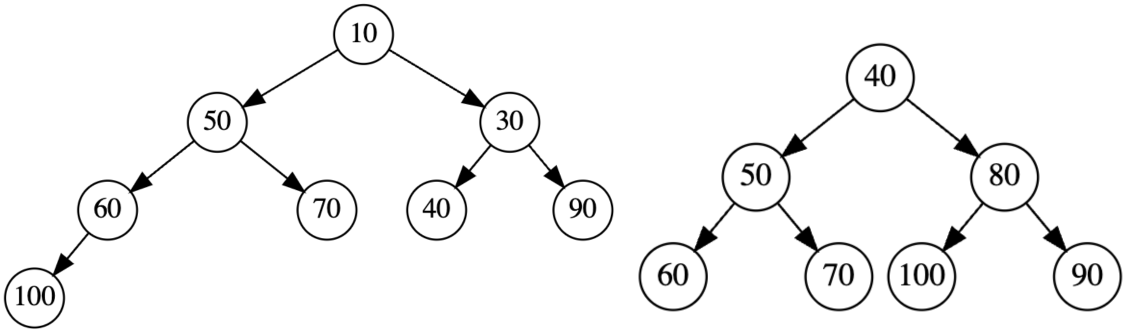


Figure 3: Min-heap.

4. [5 points] Check if the following array is a min-heap. Draw the tree and explain the reasoning.
   2, 4, 5, 7, 6, 4

   Solution:

   Draw a tree diagram

   No. Because 5 (at index 3) is bigger than its left child 4 (at index 6).

[Hash tables.]

1. Use the hash function $\textsc{HashCode}(key) = (2 \times key + 5)$ to fill a 11-entry hash table. Insert $[5, 16, 20, 39, 11, 94, 23, 88, 13, 44, 12]$, draw table, delete $[20, 44, 39]$, insert $[98, 14]$, draw table. Assume collisions are handled by:
   - (a) [5 points] Separate chaining → linked list
   - (b) [5 points] Open addressing → linear probing
   - (c) [5 points] Open addressing → quadratic probing

Solution:

Separate chaining → linked list:

Bucket	0	1	2	3	4	5	6	7	8	9	10
Keys	–	20	–	–	5, 16	11, 88, 44	39, 94	23, 12	–	13	–

Bucket	0	1	2	3	4	5	6	7	8	9	10
Keys	14	–	–	98	5, 16	11, 88	94	23, 12	–	13	–

Open addressing → linear probing:

Bucket	0	1	2	3	4	5	6	7	8	9	10
Keys	13	20	44	12	5	16	39	11	94	23	88

Bucket	0	1	2	3	4	5	6	7	8	9	10
Keys	13	14	×	12	5	16	98	11	94	23	88

Open addressing → quadratic probing:

(44 will never be inserted. So stop at this point)

Bucket	0	1	2	3	4	5	6	7	8	9	10
Keys		20		88	5	16	39	94	23	11	13

Bucket	0	1	2	3	4	5	6	7	8	9	10
Keys											

[Graphs.]

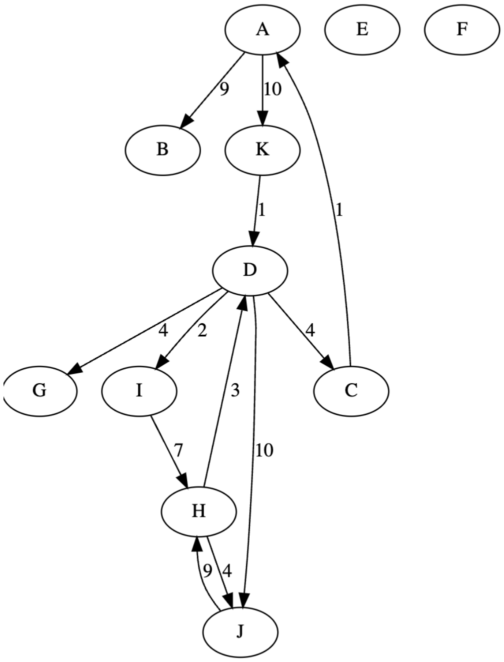


Figure 4: A directed weighted graph.

1. Consider a directed weighted graph as shown in Figure 4. Assume that the traversals are considered in alphabetical/increasing order and all adjacency lists are given in alphabetical/increasing order.

   Write the following graph representations:

   (a) [5 points] Adjacency-matrix representation

   (b) [5 points] Adjacency-list representation

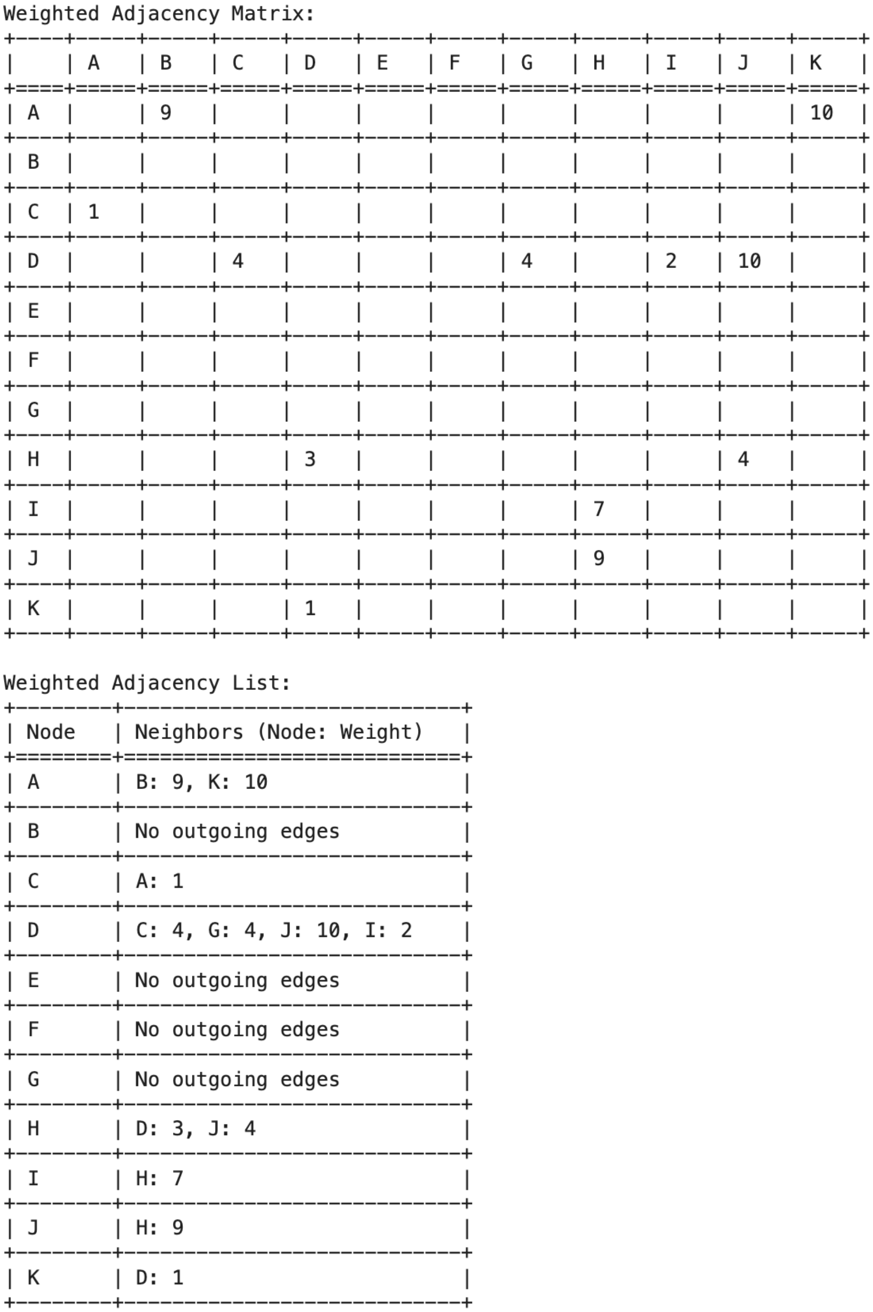
   Show the ordering of vertices produced by the following algorithms.

   (a) [5 points] Depth-first search

   (b) [5 points] Breadth-first search

   Solution:

   DFS: [A, B, K, D, C, G, I, H, J], [E], [F]
   BFS: [A, B, K, D, C, G, I, J, H], [E], [F]



```
Weighted Adjacency Matrix:
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|    | A   | B   | C   | D   | E   | F   | G   | H   | I   | J   | K   |
+====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+=====+
| A  |     | 9   |     |     |     |     |     |     |     |     | 10  |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| B  |     |     |     |     |     |     |     |     |     |     |     |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| C  | 1   |     |     |     |     |     |     |     |     |     |     |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| D  |     |     | 4   |     |     |     | 4   |     | 2   | 10  |     |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| E  |     |     |     |     |     |     |     |     |     |     |     |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| F  |     |     |     |     |     |     |     |     |     |     |     |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| G  |     |     |     |     |     |     |     |     |     |     |     |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| H  |     |     |     | 3   |     |     |     |     |     | 4   |     |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| I  |     |     |     |     |     |     |     | 7   |     |     |     |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| J  |     |     |     |     |     |     |     | 9   |     |     |     |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| K  |     |     |     | 1   |     |     |     |     |     |     |     |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+

Weighted Adjacency List:
+--------+----------------------------+
| Node   | Neighbors (Node: Weight)   |
+========+============================+
| A      | B: 9, K: 10                |
+--------+----------------------------+
| B      | No outgoing edges          |
+--------+----------------------------+
| C      | A: 1                       |
+--------+----------------------------+
| D      | C: 4, G: 4, J: 10, I: 2    |
+--------+----------------------------+
| E      | No outgoing edges          |
+--------+----------------------------+
| F      | No outgoing edges          |
+--------+----------------------------+
| G      | No outgoing edges          |
+--------+----------------------------+
| H      | D: 3, J: 4                 |
+--------+----------------------------+
| I      | H: 7                       |
+--------+----------------------------+
| J      | H: 9                       |
+--------+----------------------------+
| K      | D: 1                       |
+--------+----------------------------+
```

Figure 5: Adjacency matrix and adjacency list

[Sorting.]

1. Write the divide-and-conquer diagram to sort array $[3, 7, 5, 4, 2, 8, 1, 5]$ using:
   (a) [5 points] Merge sort algorithm
   (If there are odd number of elements in a subarray, the left half should have 1 more element than the right half)
   (b) [10 points] Quicksort algorithm

Solution:

Refer to Figures 6 and 7.

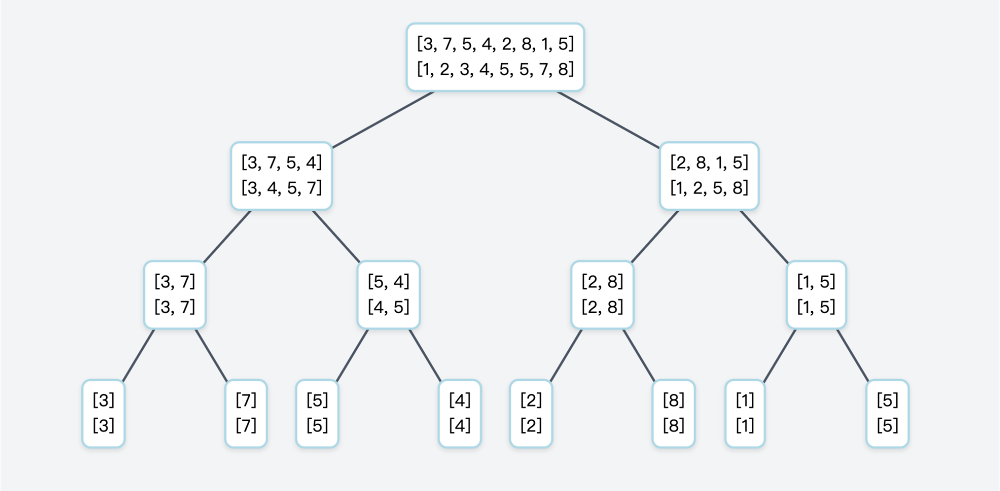


Figure 6: Merge sort

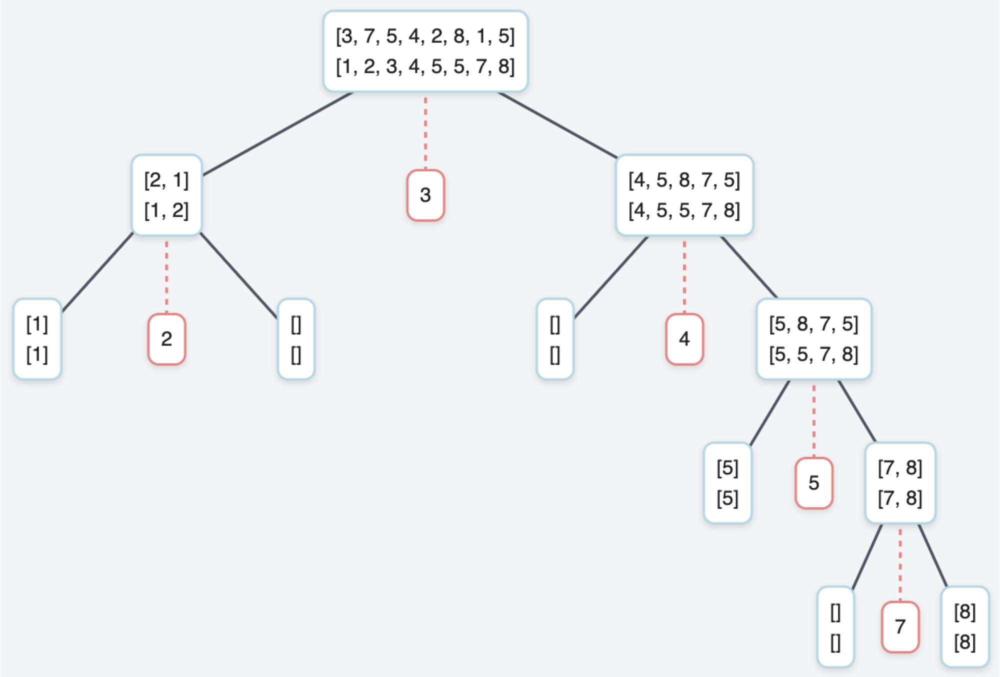
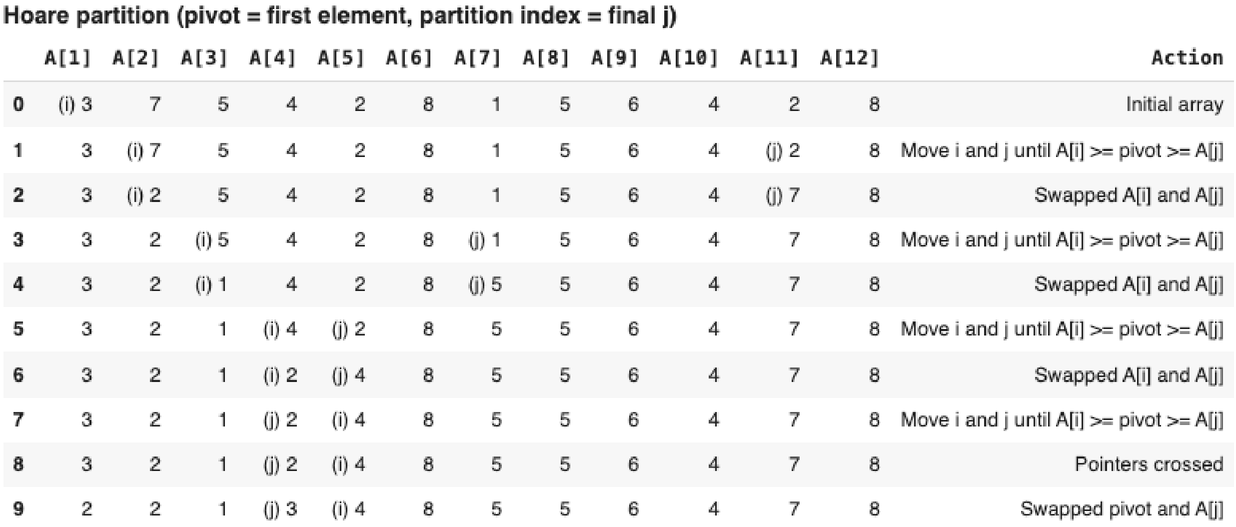


Figure 7: Quicksort

2. [10 points] Show step-by-step details of how to partition the array $A = [3, 7, 5, 4, 2, 8, 1, 5, 6, 4, 2, 8]$ using the first element of the array as the pivot element using Hoare partition

   Solution: Refer to Figure 8.



Hoare partition (pivot = first element, partition index = final j)

	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]	A[11]	A[12]	Action
0	(i) 3	7	5	4	2	8	1	5	6	4	2	8	Initial array
1	3	(i) 7	5	4	2	8	1	5	6	4	(j) 2	8	Move i and j until A[i] >= pivot >= A[j]
2	3	(i) 2	5	4	2	8	1	5	6	4	(j) 7	8	Swapped A[i] and A[j]
3	3	2	(i) 5	4	2	8	(j) 1	5	6	4	7	8	Move i and j until A[i] >= pivot >= A[j]
4	3	2	(i) 1	4	2	8	(j) 5	5	6	4	7	8	Swapped A[i] and A[j]
5	3	2	1	(i) 4	(j) 2	8	5	5	6	4	7	8	Move i and j until A[i] >= pivot >= A[j]
6	3	2	1	(i) 2	(j) 4	8	5	5	6	4	7	8	Swapped A[i] and A[j]
7	3	2	1	(j) 2	(i) 4	8	5	5	6	4	7	8	Move i and j until A[i] >= pivot >= A[j]
8	3	2	1	(j) 2	(i) 4	8	5	5	6	4	7	8	Pointers crossed
9	2	2	1	(j) 3	(i) 4	8	5	5	6	4	7	8	Swapped pivot and A[j]

Figure 8: Hoare partition.

[Algorithm codes.]

1. Learn how to write algorithm codes using 1-indexed arrays for the following algorithms. Learn also to write time and space complexities.
   - Recursion
     - Nonrecursive and recursive binary search
     - Recursive product (log complexity)
     - Recursive power function or exponentiation (log complexity)
     - Recursive divide-and-conquer array sum
     - Recursive towers of hanoi
   - Trees
     - Binary tree → Recursive preorder traversal
     - Binary tree → Recursive inorder traversal
     - Binary tree → Recursive postorder traversal
     - Binary tree → Nonrecursive level-order traversal
     - BST → Nonrecursive and recursive search
     - BST → Nonrecursive and recursive add
     - BST → Recursive remove
   - Priority queues
     - Nonrecursive add using upheap
     - Nonrecursive removemin using downheap
   - Graphs
     - Nonrecursive and recursive DFS
     - Nonrecursive BFS optimized
   - Sorting
     - Nonrecursive selection sort
     - Nonrecursive insertion sort
     - Nonrecursive heap sort (using maxheap)
     - Recursive merge sort
     - Recursive quick sort

[Applications.]

1. Consider the following ADTs (implemented in the best possible way): stack ADT, queue ADT, deque ADT, set ADT (implemented using hash tables), sorted set ADT (implemented using balanced search trees such as 2-4 trees), priority queue ADT (implemented as an array-based min-heap), and graph ADT (implemented as an adjacency list). For each of the scenarios below, mention the most suitable ADT (implemented in the best possible way).
   - (a) [1 point] Create a database designed to efficiently retrieve a specific ID and its associated information without the need for sorting.
   - (b) [1 point] Store browser history (i.e., a sequence of recently visited web URLs) and quickly find the most recently visited web URL.
   - (c) [1 point] Create a task scheduler, where each task is given a priority (i.e., a natural number in the range $[1, r]$ and r is known), and the highest-priority task is always selected first for execution.
   - (d) [1 point] Create a real-time leaderboard for a gaming application that allows update and display of the top players.
   - (e) [1 point] Design a music playlist system that enables users to add songs, remove songs, and play the next track with the closest longer duration.
   - (f) [1 point] Design an online library catalog where users can add new books, search for books by title, and print a sorted list of all the books available.
   - (g) [1 point] Create a social network application where users can make friends, and the application can quickly suggest potential new friends based on friends-of-friends connections.
   - (h) [1 point] Create a call center system that can effectively manage incoming calls and route them to available operators based on order of arrival.
   - (i) [1 point] Design a pathfinding service that helps users find the shortest route through a network of roads between two locations.
   - (j) [1 point] Develop a system so that students in a university can give printing jobs to printers in a fair manner.

   Solution:

   - (a) Set ADT (implemented using hash tables)
   - (b) Stack ADT
   - (c) Priority Queue ADT (key $= 1 \div priority$)
   - (d) Sorted Set ADT (using balanced search trees such as 2-4 trees)
   - (e) Sorted Set ADT (using balanced search trees such as 2-4 trees)
   - (f) Sorted Set ADT (using balanced search trees such as 2-4 trees)
   - (g) Graph ADT (implemented as an adjacency list)
   - (h) Queue ADT
   - (i) Graph ADT (implemented as an adjacency list)
   - (j) Queue ADT