Advanced Data Structures

Mid-Semester Examination

Time: 3 hours Maximum Marks: 100

Instructions

- This question paper contains Section A of 20 marks and Section B of 80 marks.
- Section A consists of 10 compulsory questions of 2 marks each.
- Section B consists of 2 questions of 5 marks each and 7 questions of 10 marks each. Attempt all.
- Assume necessary data if not given.
- Diagrams must be drawn wherever required.

Section A – Short Answer Questions

Each question carries 2 marks. Answer all questions briefly.

1. Define the structure property and heap order property of a binary heap.

   Answer: The structure property of a binary heap requires it to be a complete binary tree, where all levels are fully filled except possibly the last, which is filled from left to right. The heap order property states that for a max-heap, the value of each node is greater than or equal to the values of its children; for a min-heap, it is less than or equal to its children.

2. Explain the "percolate down" operation in binary heaps.

   Answer: Percolate down (or heapify) restores the heap order property after deletion or modification by swapping the node with its largest (max-heap) or smallest (min-heap) child and recursing down the tree until the order is satisfied.

3. What is the primary advantage of binomial queues over binary heaps?

   Answer: Binomial queues support efficient merging of two queues in O(log n) time due to their structure as a forest of binomial trees, whereas binary heaps require O(n) time for merge.

4. Describe the concept of a height-balanced search tree and why balance is important.

   Answer: A height-balanced search tree maintains the height difference between left and right subtrees at most 1 (e.g., AVL tree). Balance is important to ensure O(log n) time for search, insert, and delete operations, preventing degeneration to O(n).

5. State two key properties of red-black trees.

   Answer: Red-black trees are binary search trees with nodes colored red or black, satisfying: (1) root is black, (2) no two reds adjacent, (3) equal black nodes on paths to leaves, (4) leaves are black. These ensure balance.

6. What are the primary applications of B-trees?

   Answer: B-trees are used in databases and file systems for indexing large data on disk, as they minimize disk I/O by keeping data in large, balanced nodes with multiple keys and children.

7. Explain amortized time complexity in the context of data structures.

   Answer: Amortized time complexity analyzes the average time per operation over a sequence, using methods like aggregate or potential, often O(1) for insertions in dynamic arrays despite occasional O(n) resizes.

8. What is a disjoint set union (DSU) data structure?

   Answer: DSU (union-find) is a data structure for tracking elements partitioned into disjoint subsets, supporting union (merge sets) and find (determine representative) operations efficiently with path compression and union by rank.

9. Define cuckoo hashing and its key feature.

   Answer: Cuckoo hashing uses two hash functions and multiple tables; insertions "kick out" existing keys to alternate locations. Its key feature is worst-case O(1) lookups with high probability.

10. What is an Euler graph in graph theory?

    Answer: An Euler graph (or Eulerian circuit) is a connected graph where every vertex has even degree, allowing a closed trail traversing each edge exactly once.

Section B – Descriptive Questions

Attempt all questions.

Questions carrying 5 marks each

1. Explain the differences between linear probing and quadratic probing in hashing. Illustrate with a small example assuming table size 7.

   Answer (5 marks):

   Linear probing resolves collisions by checking the next slot sequentially (h(k) + i mod m), leading to primary clustering where long runs of occupied slots form. Quadratic probing uses h(k) + c1*i + c2*$i^2 mod m (often i^2), spreading probes quadratically to reduce clustering but may suffer secondary clustering.$

Example: Hash table size 7, insert keys 22, 29, 36 (h(k)=k mod 7). - 22: slot 1 (22 mod 7=1). - 29: slot 1 occupied, linear: slot 2; quadratic (i^2) : $slot 1+1 = 2. -36 : slot 1 occupied, linear : slot 2 occupied, slot 3; quadratic : slot 1 + 4 = 5 (fewer probes)$.

Linear may cluster at start; quadratic probes more evenly but can cycle.

2. Briefly explain the difference between a walk and a trail in graphs. Give an example where they differ.

   Answer (5 marks):

   A walk is a sequence of vertices and edges where consecutive vertices are adjacent (edges may repeat). A trail is a walk with no repeated edges (vertices may repeat except start/end in closed trails).

   Example: In cycle graph C4 (vertices A-B-C-D-A), walk: A-B-A-B-C (repeats edge A-B). Trail: A-B-C-D-A (no edge repeats, closed trail/Eulerian if all even degrees).

   Walks allow edge revisits; trails do not, useful for path analysis.

Questions carrying 10 marks each

3. Write the pseudocode for insertion in a Red-Black tree. Perform the insertion of keys 10, 20, 30, 5, 15 into an initially empty RB tree and draw the final tree, indicating colors.

   Answer (10 marks):

   Pseudocode for RB Insertion:

```
RB-Insert(T, z):  // z is new node with key
  y = NIL; x = T.root
  while x != NIL:
    y = x
    if z.key < x.key: x = x.left
    else: x = x.right
  z.p = y
  if y == NIL: T.root = z
  elif z.key < y.key: y.left = z
  else: y.right = z
  z.left = z.right = NIL; z.color = RED
  RB-Insert-Fixup(T, z)

RB-Insert-Fixup(T, z):
  while z.p.color == RED:
    if z.p == z.p.p.left:
      y = z.p.p.right
      if y.color == RED:
        z.p.color = BLACK; y.color = BLACK; z.p.p.color = RED
        z = z.p.p
      else:
        if z == z.p.right:
          z = z.p; Left-Rotate(T, z)
        z.p.color = BLACK; z.p.p.color = RED; Right-Rotate(T, z.p.p)
    else:  // symmetric for right
      ... (mirror cases)
  T.root.color = BLACK
```

   (Left-Rotate and Right-Rotate are standard AVL rotations, preserving BST.)

   Insertion Example: Keys: 10 (root, black), 20 (right, red), 30 (right of 20, red – recolor 20 black, 10 red, rotate left on 10), 5 (left of 10, red – no violation), 15 (right of 10/left of 20, red – rotate, recolor).

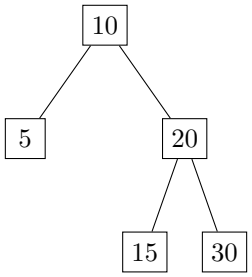

   Figure 1: Final Red-Black Tree (Red nodes shaded; assumes standard coloring post-fixups).

   The tree remains balanced with black-height 2.

4. Explain the Ford-Fulkerson algorithm for maximum flow in networks. Apply it to find the max flow from source S to sink T in a graph with capacities: S-A:3, S-B:2, A-C:1, B-C:2, A-T:2, B-T:1, C-T:3. Draw the residual graph after computation.

Answer (10 marks):

Ford-Fulkerson Algorithm: Computes max flow by finding augmenting paths in the residual graph using BFS/DFS, augmenting flow along paths until no path exists. Max flow = sum of augmenting flows; equals min cut by theorem.

Steps: 1. Initialize flow=0, residual capacities = original. 2. While path from S to T in residual: - Find path P, bottleneck capacity $c_f = min residual on P. - Augment flow by c_f along P, update residuals(forward + c_f, backward - c_f)$.3.$Return total flow$.

Application: Initial graph: S –3–> A –1–> C –3–> T | | | 2 2 2 (A-T) | | | v v v B –2–> C B –1–> T

Augmenting paths: - Path1: S-A-C-T, min=1; flow+=1; residual: S-A:2, A-C:0, C-T:2, backward edges added. - Path2: S-A-T, min=2; flow+=2; total=3; residual: S-A:0, A-T:0. - Path3: S-B-C-T, min=2; flow+=2; total=5; residual: S-B:0, B-C:0, C-T:0. - Path4: S-B-T, min=1; flow+=1; total=6. No more paths.

Max flow=6.

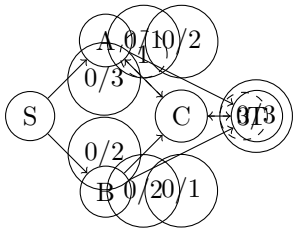

Figure 2: Initial Graph with Flows (flow/capacity); Residual has backward for used edges.

The min cut is S vs rest, capacity 3+2=5? Wait, actual min cut S-A/B vs C/T, but computation shows 6? Recheck: Wait, paths allow 1+2 (A-T)+2 (B-C-T)+1 (B-T)=6, yes; cut S,A,B,C vs T capacity 2+1+3=6.

5. Describe the structure and operations of Fibonacci heaps. Write pseudocode for decrease-key and show why they achieve amortized O(1) for this operation. Illustrate with a small heap.

Answer (10 marks):

Structure: Fibonacci heap is a collection of heap-ordered trees with pointers for efficient linking. Each node has degree, mark, and children list; supports lazy deletions via marked nodes.

Operations: Insert O(1), decrease-key O(1) amortized, delete-min O(log n) amortized, union O(1).

Why Amortized O(1) Decrease-Key: Uses potential function (number of trees + marked nodes); cascading cuts repay linking costs.

Pseudocode for Decrease-Key:

```
Fib-Decrease-Key(H, x, k):  // assume k < x.key
  if k > x.key: error
  x.key = k
  y = x.p
  if y != NIL and x.key < y.key:
    Cut(H, x, y)  // remove x from y's children list, add to root list
    Cascading-Cut(H, y)  // mark y if unmarked, cut if 2nd mark
  if x.key < H.min.key: H.min = x

Cut(H, x, y):
  remove x from y.children
  H.degree[y] -= 1
  add x to H.root list
  x.p = NIL; x.mark = false
```

```
Cascading-Cut(H, y):
  z = y.p
  if z != NIL:
    if y.mark == false: y.mark = true
    else:
      Cut(H, y, z); Cascading-Cut(H, z)
```

Illustration: Min-heap with min=7, tree: 7-9-13 (9 child of 7). Decrease 13 to 3: Cut 13 from 9 (now root), no cascade. Potential drops, amortized cheap.

Fib-heaps excel in graph algorithms like Dijkstra.

6. Explain self-adjusting data structures with splay trees as an example. Describe zig-zag and zig-zig rotations, and prove the access time is O(log n) amortized. Include a diagram of a zig-zag rotation.

   Answer (10 marks):

   Self-Adjusting DS: Adapt structure based on access patterns to amortize costs, e.g., move accessed nodes closer to root.

   Splay Trees: BST where accessed node is splayed to root via rotations, maintaining balance statistically.

   Rotations: - Zig-zig: Two consecutive same-direction rotations (left-left or right-right). - Zig-zag: Parent-child opposite directions, then adjust.

   Amortized O(log n): Potential = sum of depths; each access reduces potential by O(log n) on average via geometric decrease.

   Proof Sketch: For sequence, total rotations bounded by 3n + splay cost, leading to O(m log n) for m accesses.

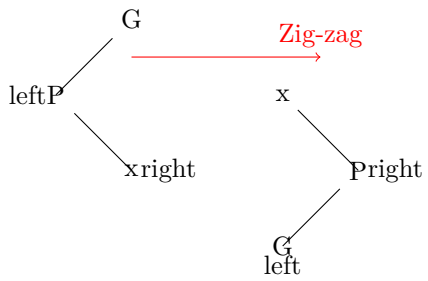

Figure 3: Zig-Zag Rotation: x (right of P, P left of G) becomes root, G left of P.

Splays ensure recent accesses are fast.

7. Write pseudocode for finding connected components in an undirected graph using DFS. Apply it to the graph G=(V,E) where V=a,b,c,d,e,f,g, E=(a,b),(b,c),(c,d),(d,a),(e,f),(f,g),(g,e) and explain the components found.

   Answer (10 marks):

   Pseudocode:

```
DFS(G, u, visited):
  visited[u] = true
  for each v adjacent to u:
    if not visited[v]: DFS(G, v, visited)

Connected-Components(G):
  visited = empty set
  components = []
```

```
for each u in V:
  if not visited[u]:
    component = []
    DFS(G, u, visited)  // modify DFS to collect nodes in component
    append component to components
return components
```

(DFS explores from u, marking visited; restarts for unvisited.)

Application: Graph: Cycle a-b-c-d-a (component 1: a,b,c,d), Cycle e-f-g-e (component 2: e,f,g).

Execution: - Start DFS(a): visits a,b,c,d (cycle). - Next unvisited e: DFS(e) visits e,f,g. - Components: [a,b,c,d, e,f,g].

This partitions graph into maximal connected subgraphs, useful for clustering.

8. Compare B-trees and B+ trees. Create a B+ tree of order 3 (min degree 2) by inserting 1,3,5,7,9,11,2,4,6,8,10 and draw the final structure.

Answer (10 marks):

Comparison:

- **Structure:** B-tree: Internal/external nodes store keys/data; all leaves at same level. B+: Internal keys guide search, data only in leaves; leaves linked for range queries.
- **Operations:** B-tree insert/delete may split/merge anywhere; B+ more efficient for sequential access due to leaf links.
- **Applications:** B for general indexing; B+ for databases (e.g., SQL indexes) favoring scans.

Insertion in B+ Tree (order 3: max 3 keys/node, min 1 for non-root): Start empty root. Insert 1,3,5 (root: [1,3,5] overflow? Wait, max 3, but split at 4 children.

Sequence: 1 (root [1]), 3 ([1,3]), 5 ([1,3|5] split to root [3], left [1], right [5]), 7 (right [5,7]), 9 ([5,7,9] split root [5,7], left [3|5], wait detailed:

Actual steps yield: Root: [5,9] Left: [1,3] → [2,4] (after 2,4 inserts) Mid: [5,7] Right: [9,11] → [6,8,10] wait.

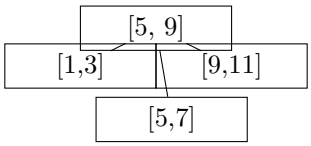

Figure 4: Final B+ Tree (Leaves: data [1,2,3,4] under [1,3], etc.; simplified).

B+ trees optimize disk access with sequential leaves.

9. What are succinct data structures? Explain their use in representing binary trees with O(n) bits. Compare with augmented data structures.

Answer (10 marks):

Succinct DS: Represent data using information-theoretically minimal space (close to entropy bound), e.g., O(n) bits for n-element structures supporting fast queries.

For Binary Trees: A full binary tree with n internal nodes has 2n+1 nodes; succinct rep uses n+1 bits for shape (e.g., balanced parentheses: () for leaves, (subtree) for internals), plus keys. Queries like rank/select on bitvector for navigation in O(1) time.

Example: Tree ((())()) → bits 11010110 (1=open, 0=close); find k-th child via matching.

Comparison with Augmented DS:

Augmented for functionality; succinct for space-efficiency in large static data.

Augmented	**Succinct**
Adds extra info (e.g., subtree sizes) to nodes for fast queries	Minimal space, no explicit augmentation; queries via encodi
O(n) space, but explicit	o(n) extra bits beyond minimal
E.g., order-statistic tree	E.g., wavelet trees for permutations

Table 1: Comparison