UNIT–2 : Advanced Data Structures (PYQ Solution)

AKTU · Design and Analysis of Algorithm (BCS503) · Solutions by **Chirag Jain**, Ycotes

SECTION A Answers (≈70 Words Each)

Q1. Derive the time complexity of Heap Sort. (Jan 2025)

Heap sort has two phases on an array of n elements.

1. **BUILD-MAX-HEAP: calls MAX-HEAPIFY on nodes ⌊n/2⌋ down to 1. A node at height h costs O(h) and there are at most ⌈n/2^(h+1)⌉ such nodes, so the cost is Σ (n/2^(h+1))·O(h) = O(n).**
2. **Sorting phase: n − 1 times, swap the root with the last element and call MAX-HEAPIFY on the root; each call costs O(log n) because the heap height is ⌊log n⌋.**

Total = O(n) + (n − 1)·O(log n) = O(n log n) in best, average and worst cases. Space is O(1) (in place).

Q2. List the properties of a Binomial Heap. (Jan 2025)

A binomial heap is a collection of binomial trees satisfying these properties:

- **Each binomial tree is min-heap ordered: a parent's key is ≤ its children's keys.**
- **For any k ≥ 0, there is at most one binomial tree B_k in the heap, so a heap of n nodes has at most ⌊$\log_2 n$⌋ + 1 trees.**
- **B_k has 2^k nodes, height k, and its root has degree k.**
- **The trees' sizes follow the binary representation of n (n = 11 = 1011_2 → B_3, B_1, B_0).**
- **Roots are kept in a linked list in increasing order of degree; the minimum is among the roots.**
- **Insert, find-min, extract-min, union and decrease-key take O(log n) time.**

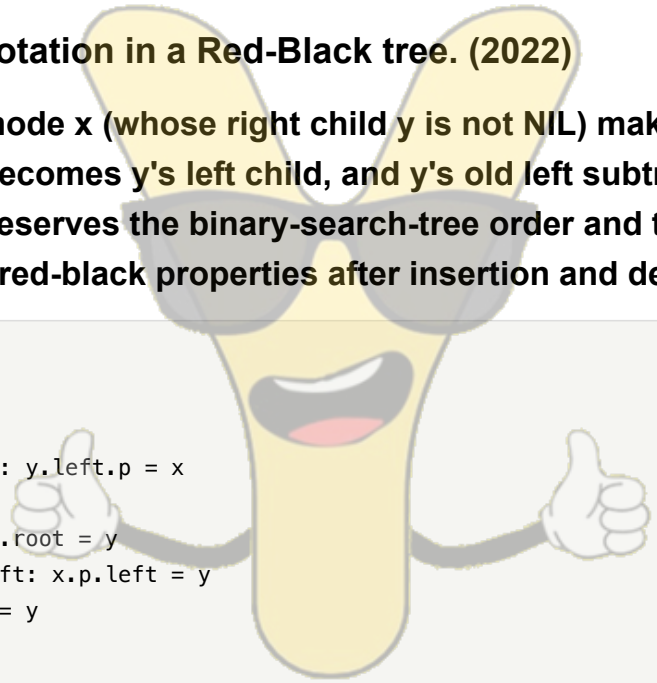
Q3. What is a Persistent Data Structure? (Jan 2026)

A persistent data structure keeps all previous versions of itself when it is updated, instead of destroying the old one. Every update creates a new version, while older versions can still be accessed (and sometimes modified).

Types: partial persistence (all versions readable, only the newest updatable), full persistence (every version readable and updatable), and confluent persistence (versions can be merged). Techniques: path copying (copy only the nodes from the root to the changed node, sharing the rest) and the fat node method. Example: persistent balanced trees used in version control and functional programming.

Q4. Explain left rotation in a Red-Black tree. (2022)

A left rotation on node x (whose right child y is not NIL) makes y the new root of the subtree, x becomes y's left child, and y's old left subtree becomes x's right subtree. It preserves the binary-search-tree order and takes O(1) time. It is used to restore red-black properties after insertion and deletion.



```
LEFT-ROTATE(T, x)
1  y = x.right
2  x.right = y.left
3  if y.left != NIL: y.left.p = x
4  y.p = x.p
5  if x.p == NIL: T.root = y
6  elif x == x.p.left: x.p.left = y
7  else: x.p.right = y
8  y.left = x
9  x.p = y
```

Q5. Write down the properties of a Fibonacci Heap. Write a short note on Fibonacci Heap. (2022, Jan 2024)

A Fibonacci heap is a collection of min-heap-ordered trees that gives very good amortized bounds.

- **Roots of the trees are stored in a circular doubly linked list; a pointer min points to the root with the smallest key.**

- **Each node stores parent, child pointer, left/right siblings, degree and a mark bit (set when a node loses a child).**
- **Trees need not be binomial; structure is lazy and is cleaned up only in EXTRACT-MIN (consolidation, linking trees of equal degree).**
- **Maximum degree D(n) = O(log n); a node of degree k has at least F_{k+2} descendants (Fibonacci numbers).**
- **Amortized costs: insert, find-min, union and decrease-key O(1); extract-min and delete O(log n).**

Q6. Explain Binary Search Tree. (Jan 2024)

A binary search tree (BST) is a binary tree in which every node holds a key such that all keys in its left subtree are smaller and all keys in its right subtree are larger than the node's key (no duplicates).

Operations SEARCH, INSERT, DELETE, MIN and MAX take O(h) time, where h is the tree height. An inorder traversal visits keys in sorted order. For a balanced tree h = O(log n); for a skewed tree (keys inserted in sorted order) h = n, so the worst case is O(n). Self-balancing variants such as red-black trees and AVL trees keep h = O(log n).

Example: inserting 50, 30, 70, 20, 40 gives root 50, left child 30 (children 20, 40) and right child 70.

SECTION B Answers (≈250 Words Each)

Q1. Define Binomial Heap. Write an algorithm for the union of two binomial heaps. Also take a suitable example that clearly illustrates the merging of two binomial heaps. (Jan 2025)

A **binomial tree B_k** is defined recursively: B_0 is a single node; B_k is formed by linking two B_{k−1} trees so that the root of one becomes the leftmost child of the other. B_k has 2^k nodes. A **binomial heap** is a set of min-heap-ordered binomial trees with at most one tree of each degree.

Algorithm BINOMIAL-HEAP-UNION(H_1, H_2):

```
1  H = MAKE-BINOMIAL-HEAP()
2  H.head = MERGE(H1, H2)        // merge root lists by increasing degree
3  x = H.head; prev = NIL; next = x.sibling
4  while next != NIL
5      if degree(x) != degree(next) or
6         (next.sibling != NIL and degree(next.sibling) == degree(x))
7          prev = x; x = next                // Case 1 and 2: move on
8      elif x.key <= next.key
9          x.sibling = next.sibling          // Case 3: next becomes child of x
10         LINK(next, x)
11     else                                  // Case 4: x becomes child of next
12         if prev == NIL: H.head = next else prev.sibling = next
13         LINK(x, next); x = next
14     next = x.sibling
15 return H
```

LINK(y, z) makes y the leftmost child of z and increments degree(z).

Example. H_1 = B_0(10) and B_1(3 with child 8) (3 nodes). H_2 = B_0(6) and B_2 rooted at 2 with children 4 and 9(15) (5 nodes).

1. Merged root list by degree: B_0(10), B_0(6), B_1(3), B_2(2).
2. Two B_0 trees: 6 < 10, so 10 becomes a child of 6 → B_1 rooted at 6.
3. Two B_1 trees (3 and 6): 3 < 6, so the tree of 6 becomes a child of 3 → B_2 rooted at 3 (children 6(10), 8).
4. Two B_2 trees (2 and 3): 2 < 3, so the tree of 3 becomes a child of 2 → **B_3 rooted at 2**.

Result: a single B_3 with 8 nodes = 3 + 5 ✓ (binary 011 + 101 = 1000).

Time: the root lists have O(log n) trees, so union takes **O(log n)**.

Q2. Let P be a Quick Sort program that sorts ascending using the first element as pivot. Let t_1 and t_2 be the number of comparisons made by P for inputs {1, 2, 3, 4, 5} and {4, 1, 5, 3, 2}. Find t_1 and t_2. (Jan 2026)

In each partition step, the pivot is compared with every other element of the sub-array, so a sub-array of size m needs m − 1 comparisons.

Input 1 = {1, 2, 3, 4, 5} (already sorted, worst case):

- Pivot 1: 4 comparisons → left {}, right {2, 3, 4, 5}
- Pivot 2: 3 comparisons → right {3, 4, 5}
- Pivot 3: 2 comparisons → right {4, 5}
- Pivot 4: 1 comparison → right {5}
- Total $t_1 = 4 + 3 + 2 + 1 =$ **10**

Input 2 = {4, 1, 5, 3, 2}:

- Pivot 4: 4 comparisons → left {1, 3, 2}, right {5}
- Left {1, 3, 2}: pivot 1: 2 comparisons → left {}, right {3, 2}
- {3, 2}: pivot 3: 1 comparison → left {2}, right {}
- {5} and {2}: single elements, 0 comparisons
- Total $t_2 = 4 + 2 + 1 =$ **7**

Answer: t_1 = 10, t_2 = 7 ($t_1 > t_2$).

Q3. What is a Binomial Heap? Write the algorithm for the Decrease-Key operation in a binomial heap and write its time complexity. (2022)

A **binomial heap** is a collection of min-heap-ordered **binomial trees**, with at most one tree of each order B_k (B_k has 2^k nodes, root degree k). A heap of n nodes has at most $\lfloor \log_2 n \rfloor + 1$ trees, and the roots form a linked list in increasing degree.

Decrease-Key lowers the key of a node x to a new value k and restores heap order by moving the node upwards (bubble up), exactly as in an ordinary binary heap.

```
BINOMIAL-HEAP-DECREASE-KEY(H, x, k)
1  if k > x.key
2      error "new key is greater than current key"
3  x.key = k
4  y = x
5  z = y.p
6  while z != NIL and y.key < z.key
7      exchange y.key with z.key       // also exchange satellite data
8      y = z
9      z = y.p
```

Working: after assigning the smaller key, compare it with the parent; while the node is smaller than its parent, swap the two keys and move one level up. Stop when the parent is smaller or the root is reached.

Example: in B_2 with root 2, child 9 and grandchild 15, decreasing 15 to 1 gives: 15→1 is below 9, swap → 1 and 9 exchange; 1 < 2, swap again → root becomes 1.

Time complexity: a binomial tree has height at most $\lfloor \log_2 n \rfloor$, so the loop runs at most that many times. Decrease-Key takes **O(log n)** time.

Q4. Prove all four properties of a Binomial Tree. (Jan 2024)

For the binomial tree B_k: (1) it has 2^k nodes; (2) its height is k; (3) there are exactly C(k, i) nodes at depth i, for i = 0, 1, …, k; (4) the root has degree k, which is greater than that of any other node, and the children of the root, from left to right, are roots of B_{k−1}, B_{k−2}, …, B_0.

Definition: B_0 is one node; B_k is two copies of B_{k−1} with the root of one made the leftmost child of the root of the other. We prove each by induction on k.

(1) Nodes = 2^k. Basis: B_0 has $1 = 2^0$ node. Step: B_k consists of two B_{k−1}, so it has 2·2^(k−1) = 2^k nodes.

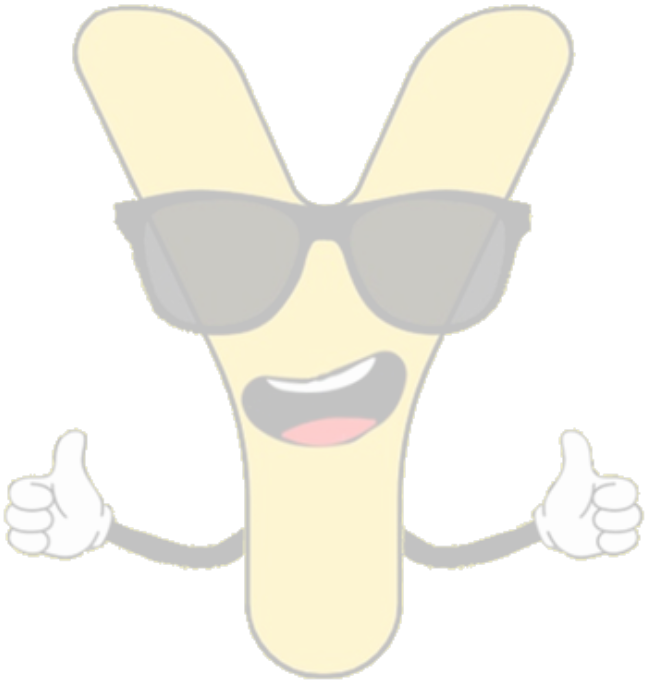
(2) Height = k. Basis: B_0 has height 0. Step: the height of B_k is one more than that of B_{k−1} (the linked tree hangs one level lower) = (k − 1) + 1 = k.

(3) Nodes at depth i = C(k, i). Basis: B_0 has C(0,0) = 1 node at depth 0. Step: let D(k, i) be the count. Nodes at depth i in B_k come from the first B_{k−1} (D(k−1, i) of them) and from the second B_{k−1}, whose depths are shifted by 1 (D(k−1, i−1)). So D(k, i) = C(k−1, i) + C(k−1, i−1) = C(k, i) by Pascal's rule.

(4) Root degree = k. Basis: B_0 has root degree 0. Step: the root of B_k is the root of one B_{k−1} (degree k − 1) plus one extra child, so its degree is k. No other node

can exceed this, since in the sub-trees all degrees are at most those of B_{k−1}, namely k − 1. By induction the root of B_{k−1} had children B_{k−2}, …, B_0; adding the other B_{k−1} as the leftmost child gives children B_{k−1}, B_{k−2}, …, B_0.

Corollary: the maximum degree in an n-node binomial tree is $\log_2 n$.



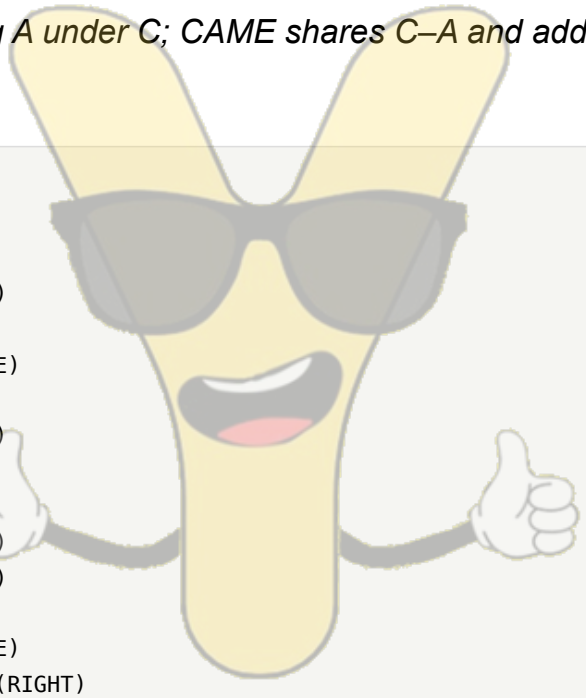
SECTION C Answers (≈500 Words Each)

Q1. Insert the following strings into an initially empty trie: DOG, DONE, CAT, CAN, RIGHT, DO, JUG, DAA, CA, CAME. Also make a compressed trie of it. (Jan 2025)

A **trie** (prefix tree) stores strings character by character; every edge is a character, common prefixes share a path, and a node is marked * if a word ends there. Insertion of a word of length L takes O(L).

Insertions: DOG creates D–O–G*; DONE shares D–O and adds N–E*; CAT creates C–A–T*; CAN shares C–A and adds N*; RIGHT creates R–I–G–H–T*; DO only marks the existing O node as a word end; JUG creates J–U–G*; DAA shares D and adds A–A*; CA marks the existing A under C; CAME shares C–A and adds M–E*.

Standard trie:



```
root
├─ D
│  ├─ O*            (DO)
│  │  ├─ G*         (DOG)
│  │  └─ N
│  │     └─ E*      (DONE)
│  └─ A
│     └─ A*         (DAA)
├─ C
│  └─ A*            (CA)
│     ├─ T*         (CAT)
│     ├─ N*         (CAN)
│     └─ M
│        └─ E*      (CAME)
├─ R ─ I ─ G ─ H ─ T*   (RIGHT)
└─ J ─ U ─ G*           (JUG)
```

Compressed trie: every node that has only one child and is not a word end is merged with that child, so edges carry strings. Chains C–A, N–E, M–E, A–A, R–I–G–H–T and J–U–G collapse.

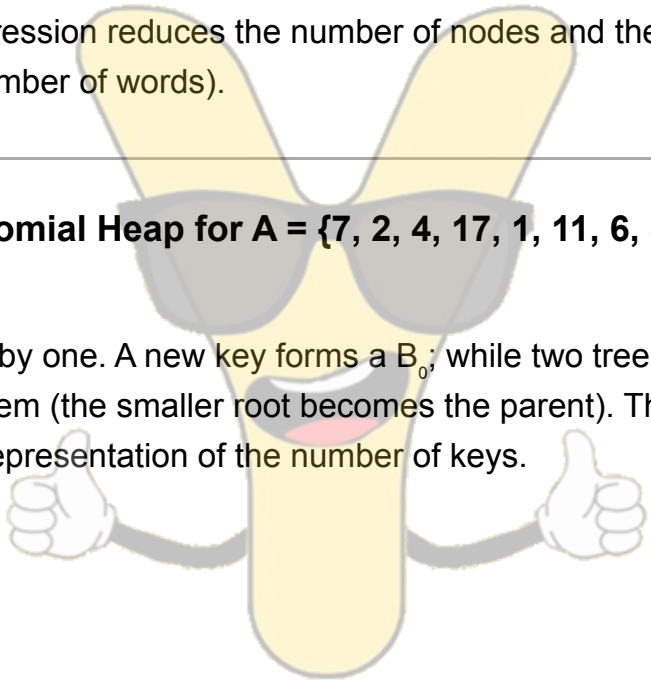
```
root
├─ D
│  ├─ O*
│  │  ├─ "G"*
│  │  └─ "NE"*
│  └─ "AA"*
├─ "CA"*
│  ├─ "T"*
│  ├─ "N"*
│  └─ "ME"*
├─ "RIGHT"*
└─ "JUG"*
```

The standard trie has 21 nodes besides the root; the compressed trie has 11. Node D remains because it has two children (O and A). Node O remains because DO is a word end.

Conclusion: compression reduces the number of nodes and the space from O(total characters) to O(number of words).

Q2. Design a Binomial Heap for A = {7, 2, 4, 17, 1, 11, 6, 8, 15, 10, 20}. (Jan 2025)

Insert the keys one by one. A new key forms a B_0; while two trees of the same degree exist, link them (the smaller root becomes the parent). The heap always mirrors the binary representation of the number of keys.



Insert	Trees after insertion (root with children)
7	B_0: 7
2	link 7, 2 → B_1: 2(7)
4	B_0: 4; B_1: 2(7)
17	4 + 17 → B_1: 4(17); with B_1 2(7) → B_2: 2(7, 4(17))
1	B_0: 1; B_2: 2(7, 4(17))
11	1 + 11 → B_1: 1(11); B_2: 2(…)
6	B_0: 6; B_1: 1(11); B_2: 2(7, 4(17))
8	6 + 8 → B_1: 6(8); with B_1 1(11) → B_2: 1(11, 6(8)); with B_2 2(…) → **B_3: 1(11, 6(8), 2(7, 4(17)))**
15	B_0: 15; B_3
10	15 + 10 → B_1: 10(15); B_3
20	B_0: 20; B_1: 10(15); B_3

Final heap ($11 = 1011_2$ → B_0, B_1, B_3):

- **B_0:** 20
- **B_1:** root 10, child 15
- **B_3:** root 1 with children (left to right) 11 (B_0), 6 with child 8 (B_1), and 2 with children 7 and 4 (4 has child 17) (B_2)

```
head → [20] → [10] → [1]
                |      ├─ 11
               15      ├─ 6 — 8
                       └─ 2 — 7
                            └─ 4 — 17
```

Check: 1 + 2 + 8 = 11 nodes ✓, every parent is smaller than its children ✓. The minimum is the smallest root, **1**.

Q3. In a skip list, if the probability of promoting a node to the next level is 1/2, what is the expected number of nodes at the second level if there are 100 nodes at the base level? (Jan 2026)

Treat the base level as level 1. Each base node is promoted to the next level independently with probability p = 1/2.

Expected nodes at the second level = 100 × p = 100 × 1/2 = **50**.

For further levels: level 3 expects 100 × p² = 25, level 4 expects 12.5, and so on. In general level i expects n·p^(i−1) nodes, so the total expected nodes are n(1 + p + p² + …) = n/(1 − p) = 200, and the expected number of levels is about $\log_2 n \approx 7$.

(If the base level is called level 0, the "second level" means level 2 and the expected number would be 25.)

Q4. Explain Kruskal's algorithm for MST and discuss its time complexity. (Jan 2026)

A **minimum spanning tree (MST)** of a connected weighted graph is a spanning tree of least total weight. **Kruskal's algorithm** is greedy: it repeatedly adds the cheapest edge that does not form a cycle.

```
MST-KRUSKAL(G, w)
1  A = empty set
2  for each vertex v: MAKE-SET(v)
3  sort the edges of E by non-decreasing weight
4  for each edge (u, v) in sorted order
5      if FIND-SET(u) != FIND-SET(v)
6          A = A u {(u, v)}
7          UNION(u, v)
8  return A
```

Example. Edges: A–B 1, B–C 2, C–D 3, A–C 4, C–E 5, B–D 6, D–E 7 (5 vertices).

Edge	Weight	Decision
A–B	1	accept
B–C	2	accept
C–D	3	accept
A–C	4	reject (A, C already connected)
C–E	5	accept (4 edges = V − 1, stop)

MST weight = 1 + 2 + 3 + 5 = **11**.

Time complexity. Sorting the edges takes O(E log E). The loop performs O(E) FIND-SET and UNION operations; with union by rank and path compression each costs O(α(V)), almost constant. So the total is O(E log E). Since $E \leq V^2$, $\log E = O(\log V)$, hence **O(E log V)**. With an unsorted edge list and no efficient sets it would be worse.

Conclusion: Kruskal's algorithm works well for sparse graphs and always produces an MST.

Q5. Discuss the various cases for insertion of a key in a red-black tree for the sequence {15, 13, 12, 16, 19, 23, 5, 8} in an empty red-black tree. (2022)

Properties: every node is red or black; root is black; leaves (NIL) are black; a red node has black children; every path from a node to leaves has the same number of black nodes. A new node is inserted as **red**. Let z be the new node, p its parent, u the uncle:

- **Case 1:** p and u red → recolor p and u black, grandparent red; continue upward with the grandparent.
- **Case 2:** p red, u black, z is an inner child (triangle) → rotate at p to convert to case 3.
- **Case 3:** p red, u black, z is an outer child (line) → rotate at the grandparent, recolor p black, grandparent red.

Insertions:

1. **15:** root, coloured black.
2. **13:** red left child of 15 (parent black) – no violation.

3. **12:** parent 13 red, uncle NIL (black), outer left–left → **case 3**: right-rotate at 15, 13 black, 15 red. Tree: 13B (12R, 15R).
4. **16:** parent 15 red, uncle 12 red → **case 1**: 12 and 15 black, 13 red → root made black. Tree: 13B (12B, 15B (–, 16R)).
5. **19:** parent 16 red, uncle NIL, outer right–right → **case 3**: left-rotate at 15, 16 black, 15 red. Tree: 13B (12B, 16B (15R, 19R)).
6. **23:** parent 19 red, uncle 15 red → **case 1**: 15 and 19 black, 16 red; 16's parent 13 is black, stop.
7. **5:** left child of 12 (black parent) – no violation.
8. **8:** goes to the right of 5. Parent 5 red, uncle (12's right) NIL → inner child, **case 2**: left-rotate at 5; then **case 3**: right-rotate at 12, 8 black, 12 red.

Final tree:

```
13B
├─ 8B
│   ├─ 5R
│   └─ 12R
└─ 16R
    ├─ 15B
    └─ 19B
        └─ 23R (right child)
```

Black height from the root is 2 on every path ✓.

Q6. What is a skip list? Explain the Search operation in a skip list with a suitable example and write its algorithm. (2022)

A **skip list** is a probabilistic data structure made of several levels of sorted linked lists. Level 0 holds all keys; each higher level holds a random subset (each node is promoted with probability 1/2), acting as an "express lane". Search, insert and delete take **O(log n) expected** time; space is O(n).

Search algorithm:

```
SKIPLIST-SEARCH(L, k)
1  x = L.head                 // sentinel at highest level
2  for i = L.level downto 0
3      while x.next[i] != NIL and x.next[i].key < k
4          x = x.next[i]       // move right
5      // else drop down one level
6  x = x.next[0]
7  if x != NIL and x.key == k return x
8  return NOT-FOUND
```

Example. Level 0: 3, 6, 7, 9, 12, 19, 21, 25, 26. Level 1: 6, 9, 19, 25. Level 2: 6, 25.

Search **19**: Level 2: head → 6 (6 < 19); next is 25 > 19, drop down. Level 1: at 6, next 9 < 19, move to 9; next 19 is not < 19, drop. Level 0: at 9, next is 12 < 19, move to 12; next 19 not < 19, stop. Step to next[0] = 19 → **found** after visiting 6, 9, 12 and 19 (4 nodes), instead of 6 nodes on a plain list.

Search **8**: Level 2 → 6; level 1: 6 → next 9 > 8, drop; level 0: 6 → 7 → next 9 ≠ 8 → not found.

Expected time: about $\log_2 n$ levels and about 2 nodes visited per level, so O(log n).

Q7. Insert the following elements using the properties of an RB tree: 61, 58, 51, 32, 39, 29. (Jan 2024)

Each new node is inserted red, and violations are fixed by recolouring (uncle red) or rotation (uncle black).

1. **61:** root → black. Tree: 61B.
2. **58:** red left child of 61.
3. **51:** left of 58. Parent 58 red, uncle NIL → left–left, right-rotate at 61; 58 black, 61 red. Tree: 58B (51R, 61R).
4. **32:** left of 51. Parent 51 red, uncle 61 red → recolor: 51 and 61 black, 58 red; root is recoloured black. Tree: 58B (51B (32R, –), 61B).
5. **39:** 39 < 58, 39 < 51, 39 > 32 → right child of 32. Parent 32 red, uncle (51's right) NIL → inner child: left-rotate at 32, then right-rotate at 51, 39 black, 51 red. Tree: 58B (39B (32R, 51R), 61B).
6. **29:** left child of 32. Parent 32 red, uncle 51 red → recolor: 32 and 51 black, 39 red. 39's parent 58 is black → stop.

Final tree:

```
58B
├─ 39R
│  ├─ 32B
│  │  └─ 29R
│  └─ 51B
└─ 61B
```

Black height = 2 on every path ✓; no red node has a red child ✓.

Q8. Explain B-Tree and its properties. Also write B-Tree deletion cases with examples. (Jan 2024)

A **B-tree** is a balanced multiway search tree used in databases and file systems, designed to minimise disk accesses. Its order is given by the **minimum degree t ≥ 2**.

Properties:

- All leaves are at the **same depth**.
- Every node except the root has **at least t − 1 keys** (t children if internal); the root has at least 1 key.
- Every node has **at most 2t − 1 keys** (2t children).
- Keys inside a node are sorted; an internal node with k keys has k + 1 children and keys separate the child ranges.
- Height h ≤ log_t((n + 1)/2), so search, insert and delete cost O(log n).

Deletion cases (t = 2, so a node has 1 to 3 keys). A node is never entered while it has only t − 1 keys; it is first made larger.

Case 1 – key in a leaf with at least t keys: just remove it. Leaf [15, 17], delete 17 → [15].

Case 2a – key k in internal node, the child before k has ≥ t keys: replace k with its predecessor and delete the predecessor recursively. Parent [30], children [10, 20] and [40]; delete 30 → replace with 20 → parent [20], children [10] and [40].

Case 2b – both neighbouring children have t − 1 keys: merge k and both children into one node, then delete k from it. Parent [30], children [10] and [40]; delete 30 → merged node [10, 40].

Case 3a – the child to descend into has t − 1 keys but a sibling has ≥ t: rotate a key through the parent. Parent [30], children [10] and [40, 50]; delete 10 → 30 moves down, 40 moves up: parent [40], children [10, 30] and [50]; then remove 10 → [30].

Case 3b – child and all siblings have t − 1 keys: merge the child with a sibling and the separating parent key. Parent [30, 60], children [10], [40], [70]; delete 10 → merge [10] + 30 + [40] = [10, 30, 40], parent [60]; remove 10 → [30, 40]. If the root becomes empty, the tree height decreases by one.

