AKTU · 3RD YEAR · SEMESTER 5 · BCS503

Design and Analysis of Algorithm

Unit 2 — Advanced Data Structures

Red-Black Trees | B – Trees | Binomial Heaps | Fibonacci Heaps | Tries | Skip List



DECODED BY

Chirag Jain

Founder, Ycotes · Gold Medalist in Artificial Intelligence, IIT Madras · taught 10,000+ students

theycotes.com

Red-Black Trees: Definition and Properties

RED-BLACK TREE

A Red-Black Tree is a self-balancing binary search tree where each node contains an extra color bit (Red or Black) used to ensure balanced tree height during insertions and deletions.

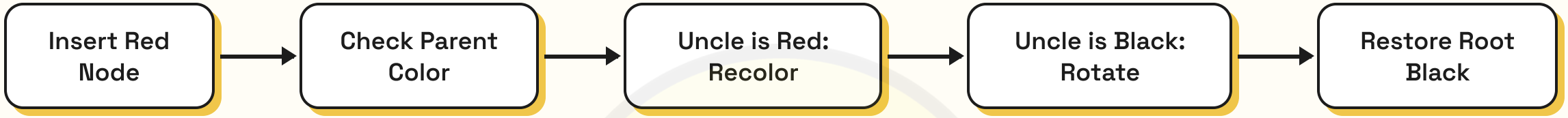
- Every node is colored either red or black, and the root node is always black.
- Every leaf node (NIL) is black, and no two consecutive red nodes can exist on a path.
- Every path from a node to descendant NIL leaves contains equal black nodes (black-height).
- Height of a Red-Black Tree with n internal nodes is bounded by $h \leq 2\log_2(n+1)$.

Red-Black Tree Operations and Rotations

Insertion in Red-Black Trees maintains balance through recoloring and tree rotations.

- New nodes are initially inserted as red nodes into the standard binary search tree location.
- Double-red violation occurs when a red parent gains a red child node.
- Uncle node color dictates the fix: if uncle is red, recolor parent, uncle, and grandparent.
- If uncle is black, perform appropriate single or double tree rotations to balance.
- Tree rotations execute in $O(1)$ time by adjusting structural node pointers.
- Total worst-case search, insertion, and deletion time complexity is $O(\log n)$.

Red-Black Tree Rotation Flow



Flow diagram for resolving double-red violations during Red-Black Tree insertion.

B-Trees: Concepts and Properties

B-TREE

A B-Tree of order m is a self-balancing, multi-way search tree designed for high-throughput disk systems, maintaining sorted data with logarithmic search times.

- Every internal node can contain up to m children and at most $m - 1$ keys.
- All non-root nodes contain at least $\lceil m/2 \rceil - 1$ keys and $\lceil m/2 \rceil$ children.
- All leaf nodes reside at the exact same depth level, ensuring uniform search paths.
- Keys inside each node are stored in strictly ascending order for binary search efficiency.

B-Tree Key Insertion Example

Insert keys 10, 20, 30, 40 into an initially empty B-Tree of order $m = 3$ (max 2 keys per node).

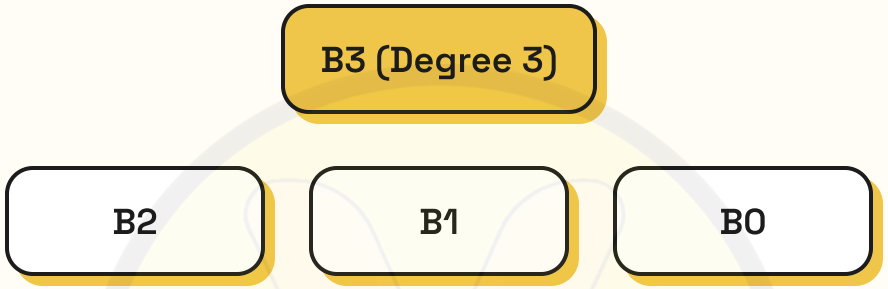
1. Step 1: Insert 10 and 20 into the root node. Node keys = [10, 20].
2. Step 2: Insert 30. Root overflows with keys [10, 20, 30].
3. Step 3: Split overflowed node at median key 20. Key 20 moves up as root.
4. Step 4: Left child contains [10], right child contains [30].
5. Step 5: Insert key 40 into right leaf node [30], resulting in [30, 40].

Final B-Tree root is [20]; Left child is [10]; Right child is [30, 40].

Comparison: Red-Black Trees vs B-Trees

Property	Red-Black Tree	B-Tree (Order m)
Branching Factor	Binary (2 children max)	Multi-way (up to m children)
Primary Application	In-memory search maps	Disk and database indexing
Tree Height	Approx. $2\log_2(n)$	Approx. $\log_{\lceil m/2 \rceil}(n)$
Keys per Node	Exactly 1 key per node	Up to $m - 1$ keys per node
Node Splitting	No (uses pointer rotations)	Splits nodes on key overflow
Worst-case Search	$O(\log n)$	$O(\log_m n)$

Binomial Tree Recursive Hierarchy



A Binomial Tree B_k is formed recursively by linking two B_{k-1} trees.

Binomial Heap Definition and Structure

BINOMIAL HEAP

A Binomial Heap is a collection of binomial trees satisfying min-heap property, where each binomial tree degree is distinct and maps directly to binary digits of key count.

- A Binomial Tree B_k has degree k, total nodes 2^k, and height k.
- Root degree of B_k equals the number of children connected directly to root.
- For a heap with n elements, tree count is at most $\lfloor \log_2 n \rfloor + 1$.
- Supports fast union operations by combining tree roots of identical degree.

Binomial Heap Operations and Union Algorithm

Union is the primary primitive operation used to construct and update Binomial Heaps.

- Union merges two heaps by combining trees of equal degree B_k into B_{k+1}.
- Extract-Min removes minimum root, reverses its child list, and unions it back.
- Insert operation executes by performing a Union with a single new node.
- Decrease-Key bubbles up the reduced key until min-heap property is restored.
- Delete key performs Decrease-Key to $-\infty$ followed by Extract-Min operation.
- Worst-case time for Union, Insert, and Extract-Min is strictly $O(\log n)$.

Fibonacci Heap Structure and Properties

FIBONACCI HEAP

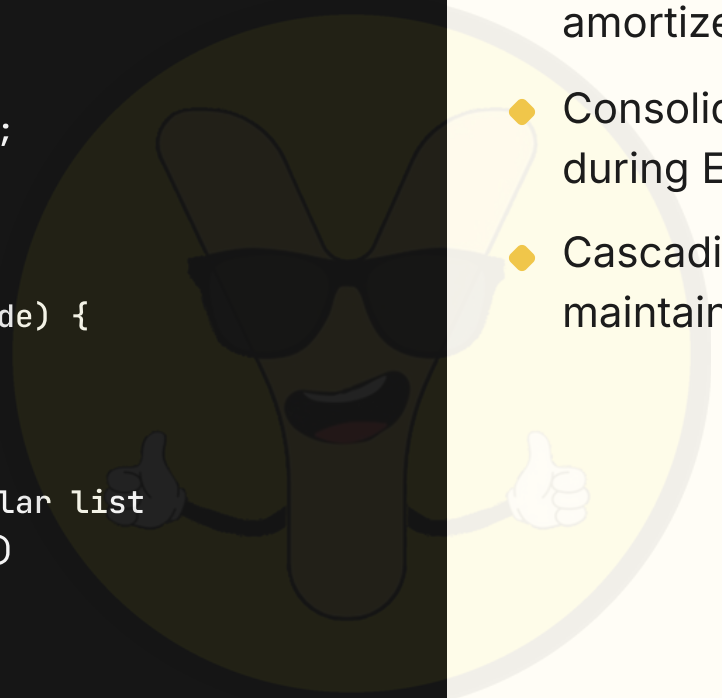
A Fibonacci Heap is a forest of heap-ordered trees that provides superior amortized time complexity for key priority-queue operations compared to standard heaps.

- Trees inside a Fibonacci Heap are not restricted to binomial shapes.
- Roots are linked via a circular doubly linked list for constant time merging.
- Minimum element key is tracked continuously using a direct minimum root pointer.
- Marked bit on a node tracks whether the node lost a child since parent assignment.

Fibonacci Heap Operations Concept

```
struct Node {
    int key, degree;
    bool mark;
    Node *parent, *child, *left, *right;
};

// Fast Insert: Amortized O(1)
void insert(Node* &minNode, Node* newNode) {
    if (!minNode) {
        minNode = newNode;
    } else {
        // Append newNode to root circular list
        if (newNode→key < minNode→key)
            minNode = newNode;
    }
}
```



- Insert and Union execute in constant amortized time $O(1)$.
- Consolidation combines equal degree roots during Extract-Min in $O(\log n)$.
- Cascading cuts run during Decrease-Key to maintain balanced tree structure.

Comparison: Binomial Heap vs Fibonacci Heap

Operation	Binomial Heap (Worst-Case)	Fibonacci Heap (Amortized)
Find Minimum	$O(1)$	$O(1)$
Insertion	$O(\log n)$	$O(1)$
Union	$O(\log n)$	$O(1)$
Decrease-Key	$O(\log n)$	$O(1)$
Extract-Min	$O(\log n)$	$O(\log n)$
Delete Node	$O(\log n)$	$O(\log n)$
Structure	Strict Binomial Trees	Flexible Forest of Trees

Trie (Prefix Tree): Structure and Properties

TRIE DATA STRUCTURE

A Trie is an advanced tree-based search structure used to store strings where node paths represent character sequences and shared prefixes.

- Key strings are not stored directly in nodes; edge character paths define stored words.
- Each node contains child pointers (e.g., array size 26) and a boolean `isEndOfWord` flag.
- Root represents an empty string prefix, branching downwards per character.
- Time complexity for Search, Insert, and Delete depends only on string length m.

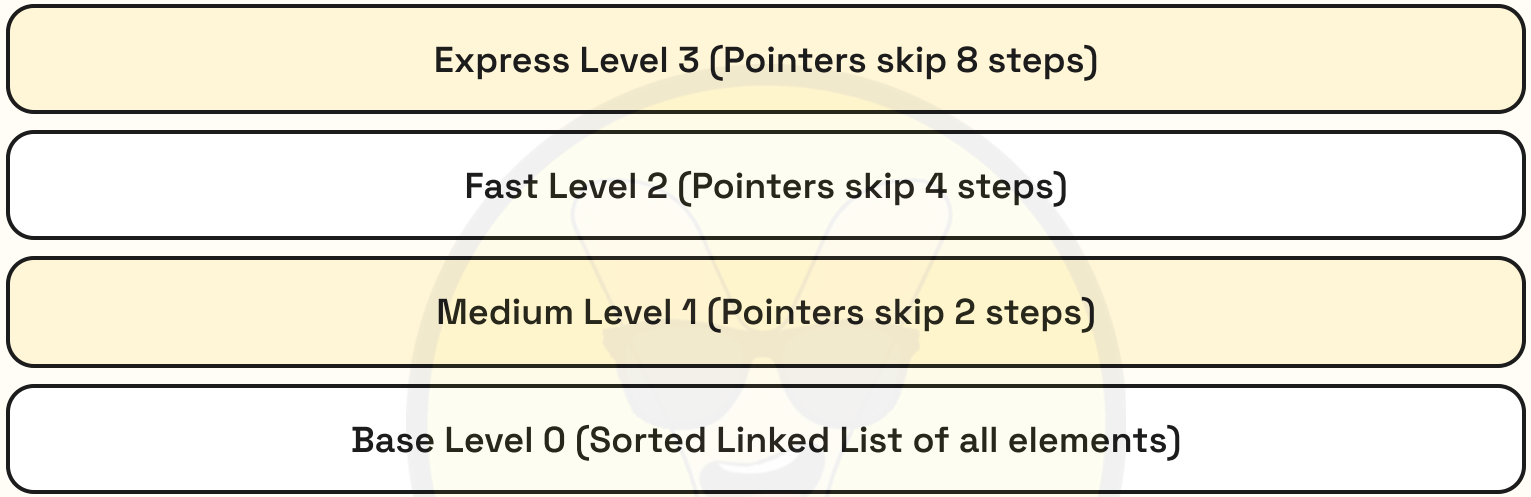
Trie Word Insertion and Search Example

Insert strings 'cat', 'car', 'dog' into a Trie and search for 'cat' and 'cap'.

1. Step 1: Insert 'cat': Root → 'c' → 'a' → 't' (set isEndOfWord = true at 't').
2. Step 2: Insert 'car': Root → 'c' → 'a' shared; branch to 'r' (isEndOfWord = true).
3. Step 3: Insert 'dog': Root → new branch 'd' → 'o' → 'g' (isEndOfWord = true).
4. Step 4: Search 'cat': Traverse 'c' → 'a' → 't'; reaches valid end node → Found.
5. Step 5: Search 'cap': Traverse 'c' → 'a'; 'p' link is null → Not Found.

Trie isolates search path directly to string length $m = 3$, independent of total key count.

Skip List Multi-Level Structure



Multi-level probabilistic linked structure enabling logarithmic search forward jumps.

Skip List Operations and Probabilistic Complexity

Skip List uses randomized coin-flips to promote node levels, replacing rigid tree balance algorithms.

- Layer 0 contains a fully sorted single linked list containing all key values.
- Node height is determined randomly using probability $p = 1/2$ per additional level.
- Search begins at top express layer and moves right or down based on comparison.
- Search, Insertion, and Deletion run in expected average time complexity $O(\log n)$.
- Space complexity is $O(n)$ in expectation with continuous memory efficiency.
- Avoids complex tree rebalancing rotations seen in AVL or Red-Black Trees.

Time Complexity Comparison Matrix

Data Structure	Search	Insert	Delete	Space Complexity
Red-Black Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
B-Tree (Order m)	$O(\log_m n)$	$O(\log_m n)$	$O(\log_m n)$	$O(n)$
Binomial Heap	$O(n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Fibonacci Heap	$O(n)$	$O(1)$	$O(\log n)$	$O(n)$
Trie (Length m)	$O(m)$	$O(m)$	$O(m)$	$O(\Sigma \cdot m \cdot n)$
Skip List (Expected)	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$

Exam Corner: Likely AKTU Questions

- **Q1** Define Red-Black Tree. Explain all properties and discuss insertion with double-red resolution steps.
- **Q2** What is a B-Tree of order m? Construct a B-Tree of order 3 for keys: 10, 20, 30, 40, 50, 60.
- **Q3** Explain the structure of Binomial Trees. Illustrate how Union operation combines Binomial Heaps.
- **Q4** Compare Binomial Heap and Fibonacci Heap with respect to structure and amortized complexities.
- **Q5** What is a Trie? Explain insert and search operations with an example and analyze time complexity.
- **Q6** Define Skip List. Explain probabilistic node level promotion and calculate expected search time.
- **Q7** Write short notes on: (a) Black-height in RB-Tree (b) Cascading Cut in Fibonacci Heap (c) Multi-way search in B-Trees.

Unit 2 done. You are exam-ready.

Revise the Exam corner once more, then open Unit 3 on theycotes.com. Every AKTU subject, unit by unit, free. Notes by Chirag Jain, Ycotes.

