Splay Trees

Abhijit Mohanty, Shaikh Jamil Ahemad

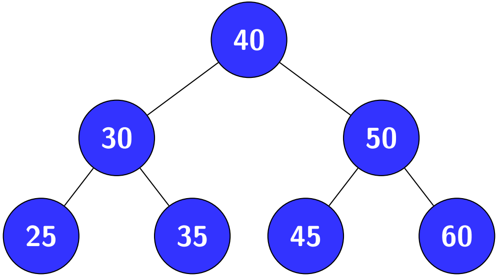
Department of Computer Science and Engineering
IIIT Bhubaneswar

Contents

- **BST Recap** – Basic operations and structure of Binary Search Trees.
- **Splay Tree** – Introduction and concept of self-adjusting trees.
- **Splaying** – Bringing an accessed node to the root using rotations.
- **Rotations** – Zig, Zig–Zig, and Zig–Zag operations.
- **Operations** – Insertion, Deletion, and Searching in Splay Trees.
- **Analysis of Splay Trees** – Time complexity and amortized analysis.

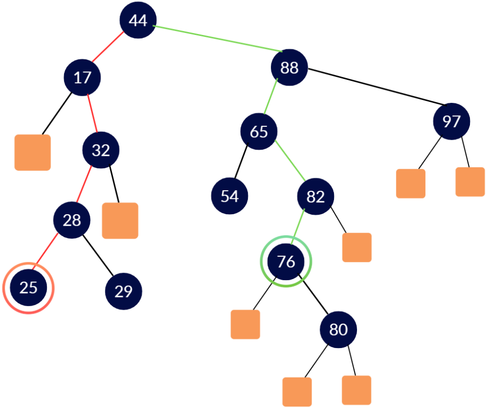
Binary Search Trees Recap

- Every node contains:
  - smaller values in left subtree
  - larger values in right subtree
- Exists a unique path from root to every node
- Successor nodes of a node: children
- Predecessor node of a node: parent
- Node with no children: Leaf



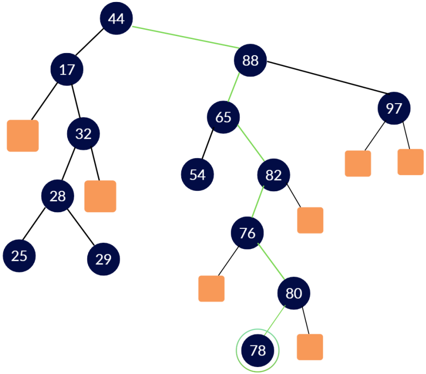
BST: Search

- **Search Operation** in a BST follows the property:
  - If key $<$ node $\rightarrow$ go to left subtree
  - If key $>$ node $\rightarrow$ go to right subtree
- Path taken depends on comparisons at each step.
- In the example:
  - Path for Search(25) is highlighted in red.
  - Path for Search(76) is highlighted in green.



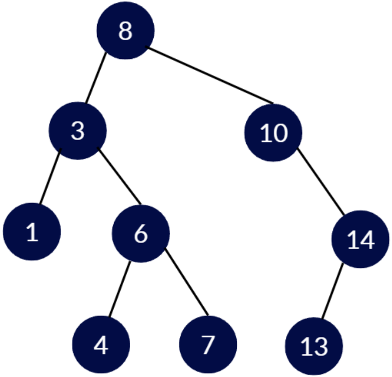
BST: Insertion

- **Insertion** in a BST always follows the binary search property:
  - Compare the new key with the root.
  - If smaller $\rightarrow$ go to the left subtree.
  - If larger $\rightarrow$ go to the right subtree.
  - Continue until a null position is reached.
- Insert the new node at that null position as a leaf.
- The structure of the tree changes locally no rearrangements elsewhere.
- In this example:
  - The key 78 is inserted.
  - The path from the root to the insertion point is highlighted in green.

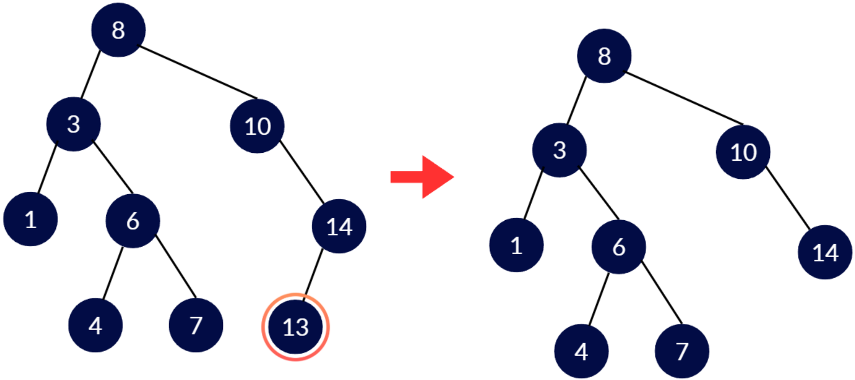


BST: Deletion

- **Deletion** in a BST must preserve the binary search property.
- The process depends on the type of node being deleted:
  - **Case 1:** Node is a **leaf node**
  - **Case 2:** Node is a **non-leaf node**
- Each case is handled differently to maintain BST structure.
- In the following slides, we'll use this BST to explore the cases

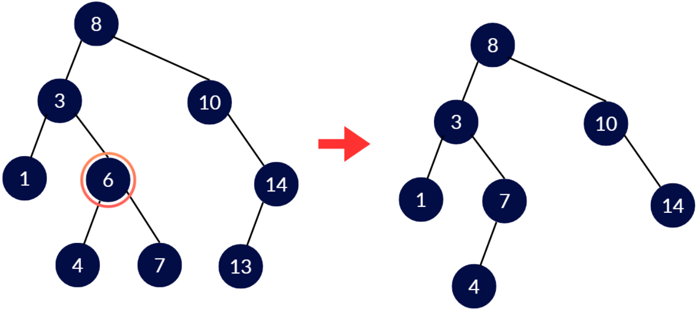


Deletion Case 1: Leaf Node



13 is a leaf node, so it is deleted directly without affecting the rest of the tree.

Deletion Case 2: Non-Leaf Node



In-order traversal: 1, 3, 4, 6, 7, 8, 10, 13, 14

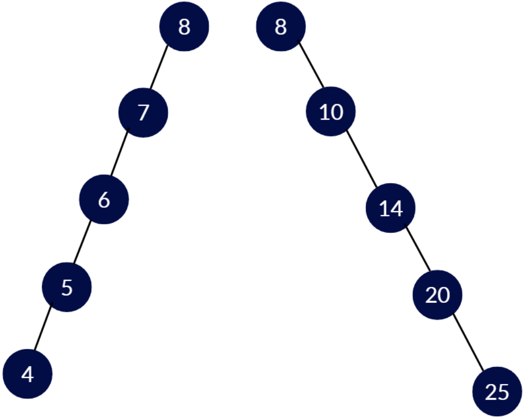
Predecessor of 6: 4 **Successor of 6: 7**

In deletion of a non-leaf node, the node is replaced by its in-order successor or predecessor.

Here, node 6 is replaced by its in-order successor 7.

Skewed Binary Search Trees

- A **Binary Search Tree (BST)** can become **skewed** when elements are inserted in sorted order.
- Two types of skewness:
  - **Left-skewed tree**: all nodes have only left children.
  - **Right-skewed tree**: all nodes have only right children.
- In such cases, the height of the tree becomes n.
- Hence, all operations (*search*, *insert*, *delete*) take $\mathcal{O}(n)$ time - i.e., **linear time.**
- This why we move towards Splay Trees.



What are Splay Trees?

- A **Splay Tree** is a self-adjusting binary search tree invented by Daniel Sleator and Robert Tarjan in 1985.
- **Key Property:** Recently accessed elements are quick to access again.
- Unlike AVL or Red-Black trees, splay trees do not maintain strict balance.
- Instead, they reorganize themselves using a special operation called **splaying**.
- After every operation (search, insert, delete), the accessed node is moved to the root.
- This ensures frequently accessed nodes stay near the top of the tree.
- **Amortized Time Complexity:** $\mathcal{O}(\log n)$ for all operations.
- **Worst-case Time Complexity:** $\mathcal{O}(n)$ for individual operations.

Splaying

- In splay trees, after performing an ordinary BST Search, Insert, or Delete, a **splay operation** is performed on some node x (as described later).
- The splay operation moves x to the root of the tree.
- The splay operation consists of sub-operations called **zig-zig**, **zig-zag**, and **zig**.
- These operations use tree rotations to move the node upward while maintaining the BST property.
- Splaying not only moves the target node to the root but also tends to balance the tree.

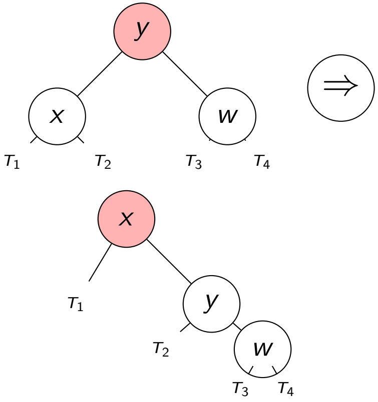
Zig Operation

When to use:

- Node x has **no grandparent**
- Parent y is the root
- This is the **last step** in a splay operation
- Occurs when x has **odd depth**

What happens:

- Perform a single rotation
- x becomes the new root
- Depth of x decreases by **1**



(Symmetric case also exists)

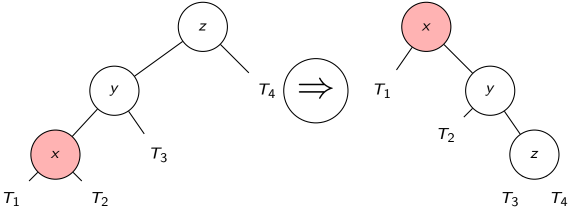
Zig-Zig Operation

When to use:

- Node x **has a grandparent** z
- x and parent y are in the **same direction**
- Both are left children or both are right children

What happens:

- Two rotations: first on y and z, then on x and y
- **Different** from zig-zag!
- x moves up by **2 levels**
- Makes tree more balanced
- Depth of x decreases by **2**



(Symmetric case also exists)

Why Zig-Zig Instead of Double Zig?

- **Question:** Why not just rotate x twice upward (double zig)?
- **Answer:** Zig-zig provides better balance!

Double Zig (Bad)

- Rotating in same direction twice
- Keeps tree unbalanced

Zig-Zig (Good)

- Rotates parent first, then child
- Improves balance of tree

Key Insight: Zig-zig not only moves x up, but also reduces the depth of nodes on the path from x to the root, making the tree more balanced overall.

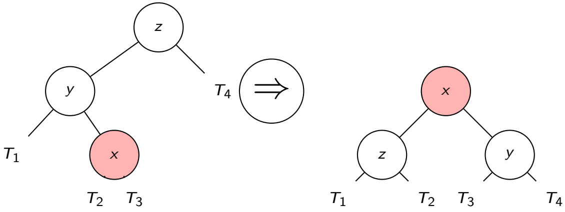
Zig-Zag Operation

When to use:

- Node x **has a grandparent** z
- x and parent y are in **opposite directions**
- x is left child of right child (or vice versa)

What happens:

- Two rotations: first on x and y, then on x and z
- Similar to AVL double rotation
- x moves up by **2 levels**
- Depth of x decreases by **2**

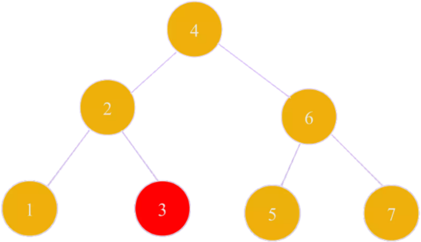


(Symmetric case also exists)

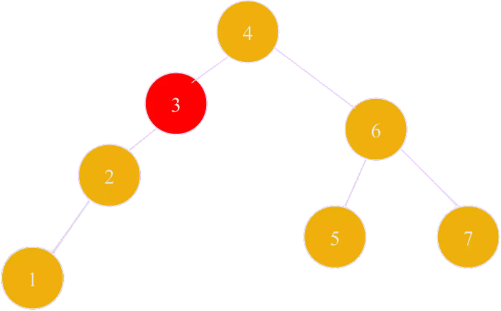
Rules of Splaying: Search

- When searching for key I, if I is found at node x, we splay x.
- If not successful, we splay the node which is accessed recently

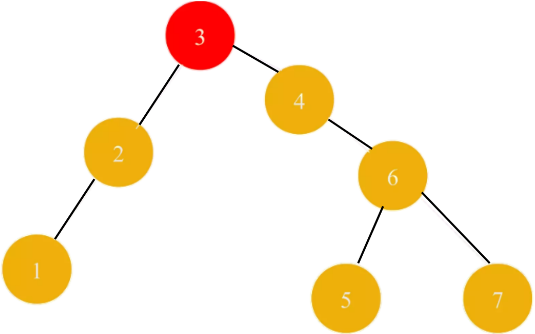
Splaying Example: Searching 3



Splaying Example: Searching 3

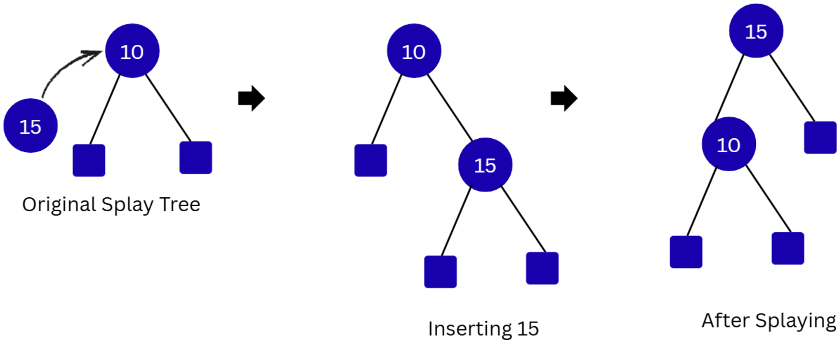


Splaying Example: Searching 3



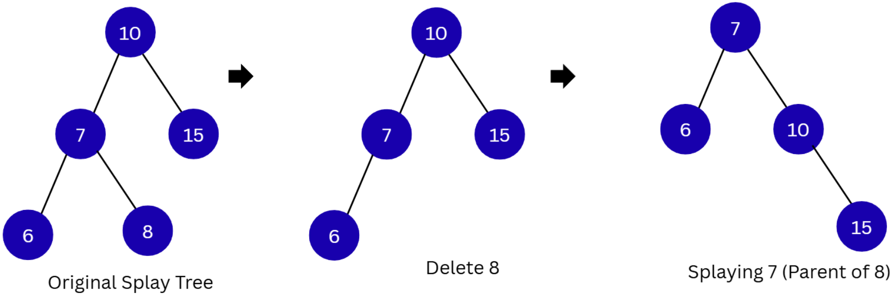
Rules of Splaying: Insertion

- When inserting a key I, we splay the newly created internal node where I was inserted.



Rules of Splaying: Deletion

1. Search for the node.
2. Delete that node.
3. Splay the parent of the deleted node.



Splay Tree Analysis

Meaning of Ranks

- The rank of a tree is a measure of how well balanced it is.
- A well-balanced tree has a low rank.
- A badly balanced tree has a high rank.
- Splaying operations tend to make the rank smaller, which balances the tree and makes other operations faster.
- Some operations near the root may make the rank larger and slightly unbalance the tree.
- Amortized analysis is used on splay trees, with the rank of the tree serving as the potential.

Rank and Potential Function

- T is a splay tree with n keys.
- **Definition:** The **size** of a node v in T, denoted $n(v)$, is the number of nodes in the subtree rooted at v.
  - **Note:** The root is of size $2n+1$.
- **Definition:** The **rank** of a node v, denoted $r(v)$, is defined as:

$$r(v) = \lg(n(v))$$

  - **Note:** The root has rank $\lg(2n+1)$.
- **Potential Function:** The potential of the entire tree T is defined as:

$$\Phi(T) = \sum_{v \in T} r(v)$$

Case 1: Zig–Zig

Only the ranks of x, y, **and** z **change.** Also, $r'(x) = r(z)$, $r'(y) \leq r'(x)$, and $r(y) \geq r(x)$. Thus,

$$\Delta\Phi = r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)$$

$$= r'(y) + r'(z) - r(x) - r(y)$$

$$\leq r'(x) + r'(z) - 2r(x) \quad (1)$$

Also, $n(x) + n'(z) \leq n'(x)$, which (by property of lg) implies:

$$r(x) + r'(z) \leq 2r'(x) - 2 \Rightarrow r'(z) \leq 2r'(x) - r(x) - 2 \quad (2)$$

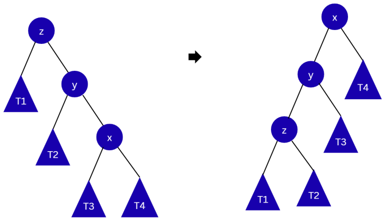
By (1) and (2):

$$\Delta\Phi \leq r'(x) + (2r'(x) - r(x) - 2) - 2r(x)$$

$$\Rightarrow \Delta\Phi \leq 3(r'(x) - r(x)) - 2$$

Hence, the amortized complexity for Zig–Zig (two rotations) is:

$$O(3(r'(x) - r(x)))$$



*Zig–Zig rotation example

If $a > 0$, $b > 0$, and $c \geq a + b$, then $\lg a + \lg b \leq 2 \lg c - 2$

Case 2: Zig–Zag

Only the ranks of x, y, **and** z **change.** Also, $r'(x) = r(z)$ and $r(x) \leq r(y)$. Thus,

$$\Delta\Phi = r'(x) + r'(y) + r'(z) - r(x) - r(y) - r(z)$$

$$= r'(y) + r'(z) - r(x) - r(y)$$

$$\leq r'(y) + r'(z) - 2r(x) \quad (1)$$

Also, $n'(y) + n'(z) \leq n'(x)$, which (by property of lg) implies:

$$r'(y) + r'(z) \leq 2r'(x) - 2 \quad (2)$$

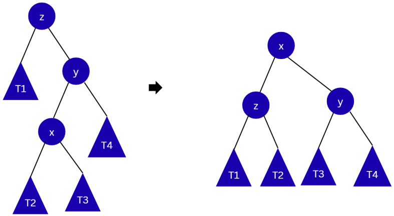
By (1) and (2):

$$\Delta\Phi \leq 2r'(x) - 2 - 2r(x)$$

$$\Rightarrow \Delta\Phi \leq 3(r'(x) - r(x)) - 2$$

Hence, the amortized complexity for Zig–Zag (two rotations) is:

$$O(3(r'(x) - r(x)))$$



Zig–Zag rotation example

If $a > 0$, $b > 0$, and $c \geq a + b$, then $\lg a + \lg b \leq 2 \lg c - 2$

Case 3: Zig

Only the ranks of x **and** y **change.** Also, $r'(y) \leq r(y)$ and $r'(x) \geq r(x)$. Thus,

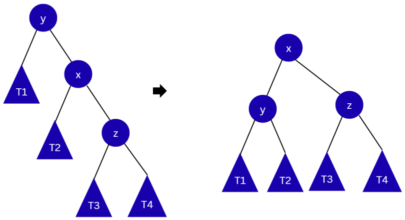
$$\Delta\Phi = r'(x) + r'(y) - r(x) - r(y)$$

$$\leq r'(x) - r(x)$$

$$\leq (r'(x) - r(x))$$

Hence, the amortized complexity for Zig (one rotation) is:

$$O((r'(x) - r(x)))$$



Zig rotation example

Final Time Complexity of Splay Trees

When we splay, we rotate the node upward and each rotation moves it closer to the root.

If the node is at a depth d, it can take at most d rotations to bring it to the root.

In the **worst case**, for a balanced tree:

$$d = \log n$$

Therefore, the number of steps (and the time taken) is at most proportional to $\log n$.

$$\boxed{\text{Time Complexity (per operation)} \;=\; O(\log n)}$$

Hence, for a sequence of m operations:

$$\boxed{\text{Total Time Complexity} \;=\; O(m \log n)}$$

Conclusion

- A balanced binary search tree.
- Doesn't need any extra information to be stored in the node, *i.e.*, color, level, etc.
- Running time is $O(m \log n)$ for m operations.
- Can adapt to the access pattern of items in a dictionary to achieve faster running times for frequently accessed elements ($O(1)$ in best cases, whereas AVL trees are about $O(\log n)$, etc.).

Thank You