Binary Search Trees

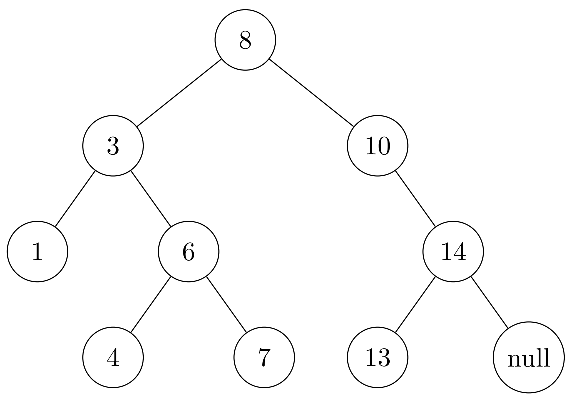
A **Binary Search Tree (BST)** is a binary tree where each node contains a key such that:

`left subtree keys` $<$ `node key` $<$ `right subtree keys`

This ordering enables efficient searching, insertion, and deletion operations.

Similar to hashtables, they are used to store key-value pairs. However unlike hashtables, BSTs maintain keys in sorted order.

1 Example BST Structure



This BST contains keys ordered such that for every node:

`left subtree keys` $<$ `node key` $<$ `right subtree keys`

2 Use Cases

- Symbol tables in compilers
- Maintaining sorted datasets
- Searching and dynamic set operations

3 Time Complexity

Operation	**Average Case**	**Worst Case (Unbalanced)**
Search	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Insert	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Delete	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$

3.1 Terminology

- **BST:** A binary tree with ordered keys
- **Leaf:** A node with no children
- **Internal node:** Has at least one child
- **Height:** Number of edges on the longest downward path
- **Balanced Tree:** Height is approximately $\log n$

4 Why BST Operations Are $\mathcal{O}(\log n)$

In a **balanced** BST, each level of the tree halves the search space. With n nodes, the maximum height is approximately $\log_2 n$.

- At each comparison, we eliminate half the remaining nodes.
- This is analogous to binary search on a sorted array.

Hence, insertion, search, and deletion all take $\mathcal{O}(\log n)$ time on average for balanced trees.

4.1 C++ Implementation of a BST

```cpp
#include
using namespace std;

struct BSTNode {
  int key;
  BSTNode* left;
  BSTNode* right;

  BSTNode(int val) : key(val), left(nullptr), right(nullptr) {}
};

class BST {
public:
  BST() : root(nullptr) {}

  void insert(int key) {
    root = insert(root, key);
  }

  bool search(int key) const {
    return search(root, key);
  }

  int min() const {
    BSTNode* node = min(root);
    if (node) return node->key;
      throw runtime_error("Tree is empty");
  }

  int max() const {
    BSTNode* node = max(root);
    if (node) return node->key;
      throw runtime_error("Tree is empty");
  }

private:
  BSTNode* root;

  BSTNode* insert(BSTNode* node, int key) {
```

```
    if (!node) return new BSTNode(key);
    if (key < node->key)
      node->left = insert(node->left, key);
    else if (key > node->key)
      node->right = insert(node->right, key);
    return node;
  }

  bool search(BSTNode* node, int key) const {
    if (!node) return false;
    if (key == node->key) return true;
    if (key < node->key)
      return search(node->left, key);
    else
      return search(node->right, key);
  }

  BSTNode* min(BSTNode* node) const {
    if (!node) return nullptr;
    while (node->left)
        node = node->left;
    return node;
  }

  BSTNode* max(BSTNode* node) const {
    if (!node) return nullptr;
    while (node->right)
        node = node->right;
    return node;
  }

};
```

Inorder Traversal (Left → Root → Right):

```
void inorder(BSTNode* node) const {
  if (!node) return;
  inorder(node->left);
  cout << node->key << " ";
  inorder(node->right);
}
```

Preorder Traversal (Root → Left → Right):

```
void preorder(BSTNode* node) {
  if (!node) return;
  cout << node->key << " ";
  preorder(node->left);
  preorder(node->right);
}
```

Postorder Traversal (Left → Right → Root):

```
void postorder(BSTNode* node) {
  if (!node) return;
  postorder(node->left);
  postorder(node->right);
  cout << node->key << " ";
}
```

5 Deletion in BSTs

To delete a node, we consider three cases:

- **Case 1: Leaf Node (0 children)** Simply remove the node.
- **Case 2: One Child** Replace the node with its only child.

- **Case 3: Two Children** Replace the node's value with either:
  - The **inorder successor** (smallest value in right subtree)
  - The **inorder predecessor** (largest value in left subtree)

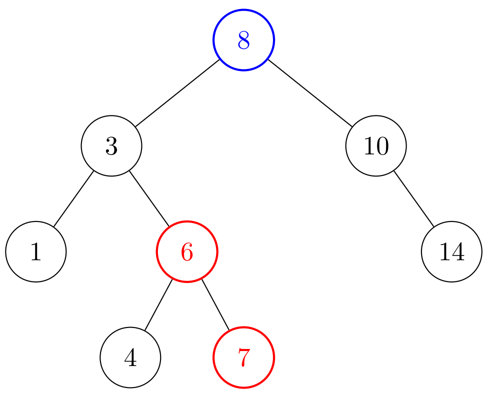
  Then delete the successor or predecessor node.

Finding the Inorder Successor

To find the inorder successor of a node in a BST, we consider two cases:

- If the node has a right subtree: successor is the leftmost node in the right subtree
- If the node has no right subtree: successor is the lowest ancestor for which the node lies in its left subtree

5.1 Visualizing Inorder Successor Cases



Red	Target Node
Blue	Inorder Successor

Case 1: Node **6** has a right child (7). Its inorder successor is the leftmost node in its right subtree: **7**.
Case 2: Node **7** has no right subtree. To find its inorder successor:

- We move up the tree from node 7 to its parent (node 6), but since 7 is a **right child**, we continue upward.
- We then move from 6 to its parent (node 3), where 6 is a **right child** again, so we continue.
- We then move from 3 to node 8, where 3 is a **left child**.
- Hence, node **8** is the lowest ancestor where node 7 (through its lineage) lies in the left subtree.

Therefore, the inorder successor of node **7** is **8**.

5.2 Degenerate Cases and Balancing

A poorly structured BST can degrade to a linked list, resulting in $\mathcal{O}(n)$ time operations. To prevent this:

- Use self-balancing trees like AVL or Red-Black Trees
- Randomize insertions or rebalance periodically

Balanced BSTs ensure height $\mathcal{O}(\log n)$, which guarantees logarithmic performance for search-related operations.

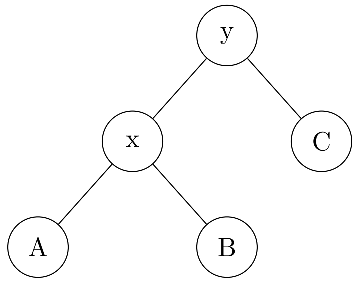
6 Tree Rotations

Tree rotations are fundamental operations used in self-balancing binary search trees (e.g., AVL and Red-Black Trees) to restore height balance after insertions or deletions. A rotation is a local restructuring operation that preserves the BST ordering.

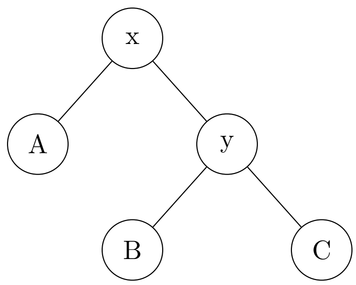
6.1 Right Rotation (`rotateRight`)

Right rotation is applied at a node y with a left child x to reduce the height of the left subtree.

Before:



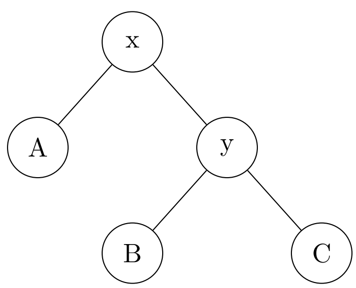
After `rotateRight(y)`:



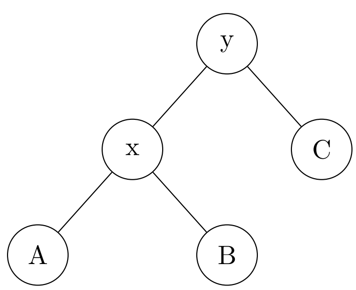
6.2 Left Rotation (`rotateLeft`)

Left rotation is applied at a node x with a right child y to reduce the height of the right subtree.

Before:



After `rotateLeft(x)`:



6.3 Preserving the BST Property During Rotations

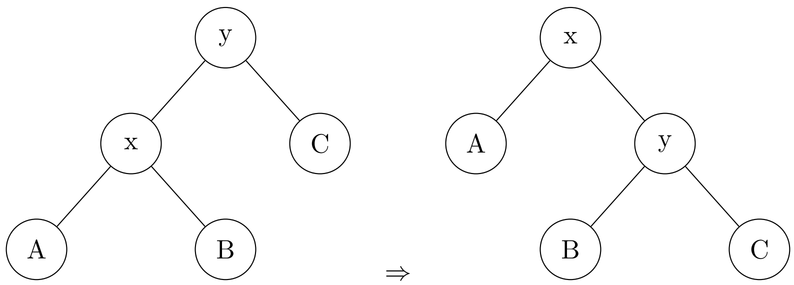
Tree rotations maintain the binary search tree property:

$$\texttt{left subtree} < \texttt{node} < \texttt{right subtree}$$

Consider the right rotation at node y (where x is its left child):
Before rotation:

- Subtree A: All keys < x.key
- Subtree B: x.key < keys < y.key
- Subtree C: All keys > y.key

Right Rotation:



After rotation:

- Subtree A remains in the left of x
- Subtree B becomes left of y, and since B's keys were > x.key and < y.key, they remain valid
- Subtree C remains to the right of y

All relative key orderings are preserved, satisfying:

$$\texttt{A} < x < \texttt{B} < y < \texttt{C}$$

Both left and right rotations preserve in-order traversal and relative ordering of keys. Hence, the BST property remains intact after any single rotation.

7 std::map and std::set in C++ STL

The C++ Standard Template Library (STL) provides `std::map` and `std::set` as associative containers that store elements in sorted order.

`std::map` stores key-value pairs where keys are unique and each key maps to a corresponding value. `std::set` stores unique keys without associated values.

Both containers are typically implemented using a balanced binary search tree, most commonly a *Red-Black Tree*. This ensures that insertions, deletions, and lookups have logarithmic time complexity: $O(\log n)$.

Key properties:

- Elements are kept sorted according to the comparison function (default: `operator<`).
- Iteration over `map` or `set` yields elements in sorted order.
- Efficient operations such as range queries, lower and upper bound searches, and equal range queries are supported.

Relation to Binary Search Trees. `std::map` and `std::set` are abstracted interfaces over balanced binary search trees. These containers provide the ordered behavior and logarithmic performance characteristics of binary search trees while hiding the tree operations from the user.

Contrast with Hash-Based Containers. Unlike `std::unordered_map` and `std::unordered_set`, which use hash tables:

- `map` and `set` maintain elements in order.
- Lookup, insertion, and deletion are $O(\log n)$ rather than average $O(1)$ (hash containers) but no ordering.
- `map` and `set` support ordered range operations; hash containers do not.

8 Practice problem

Leetcode Problem 108: *Convert Sorted Array to Binary Search Tree*
`https://leetcode.com/problems/convert-sorted-array-to-binary-search-tree/description/`