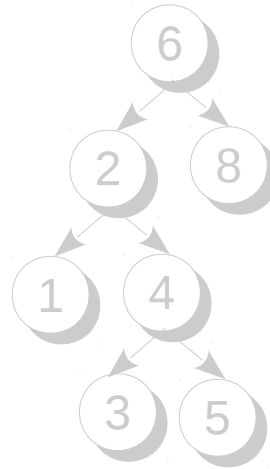


Binary Search Trees



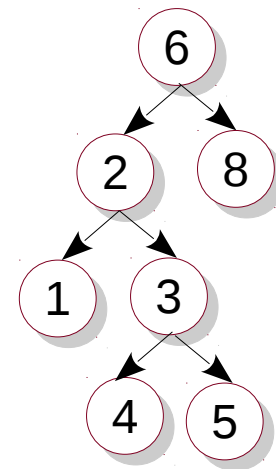
Rupesh Nasre.
rupesh@iitm.ac.in

Web: ~rupesh/teaching/pds/spw2019

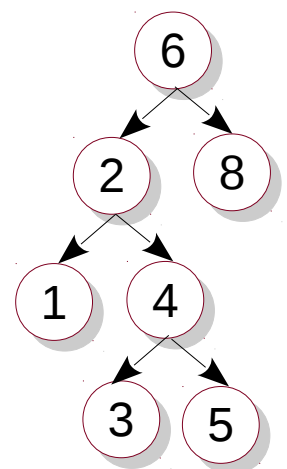
Summer Projects and Workshop
June 2019

Definition

- A BST is a binary tree.
- If it is non-empty, the **value** at the root is larger than any value in the left-subtree, and
- the value at the root is smaller than any value in the right-subtree.
- The left and the right subtrees are BSTs.
- Assumption: All values are unique.



Not a BST



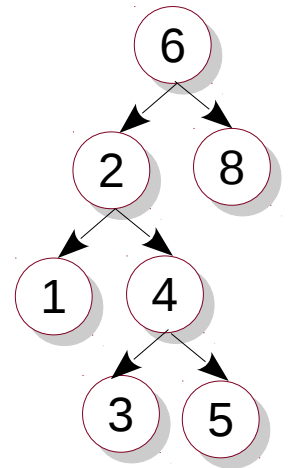
BST

BST Operations

```
class BST {  
    ...  
public:  
    ...  
    PtrToNode search(DataType element);  
    PtrToNode findMin();  
    PtrToNode findMax();  
    PtrToNode insert(DataType element);  
    void remove();  
};
```

Search

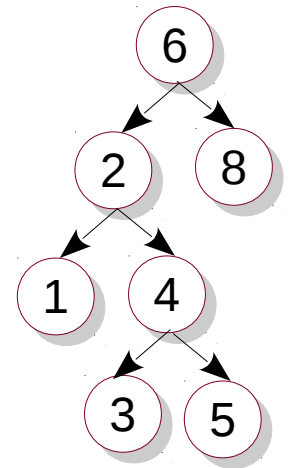
```
bool Tree::search(DataType data, PtrToNode rr) {  
    if (rr == NULL) return false;  
    if (data == rr->data) return true;  
    if (data < rr->data) return search(data, rr->left);  
    return search(data, rr->right);  
}  
  
bool Tree::search(DataType data) {  
    bool present = search(data, root);  
    if (present)  
        std::cout << data << " present." << std::endl;  
    else  
        std::cout << data << " NOT present." << std::endl;  
    return present;  
}
```



Switch to bst.cpp.

FindMin

```
DataType Tree::findmin(PtrToNode rr) {  
    if (rr) {  
        if (rr->left) return findmin(rr->left);  
        return rr->data;  
    }  
    return -1;  
}  
DataType Tree::findmin() {  
    return findmin(root);  
}
```

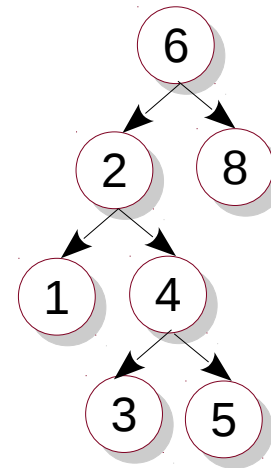
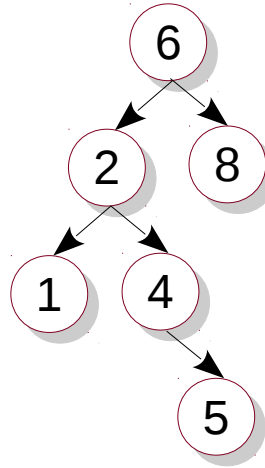
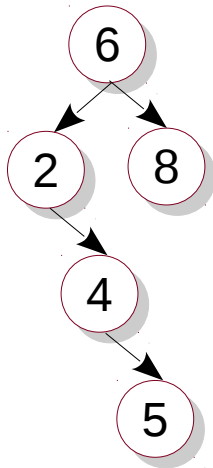
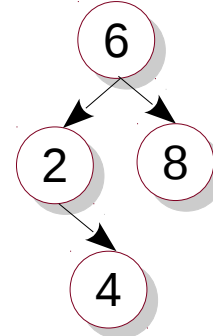
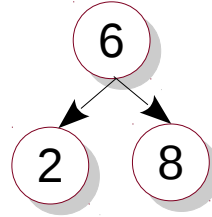
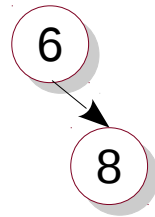


**Write iterative
findMax.**

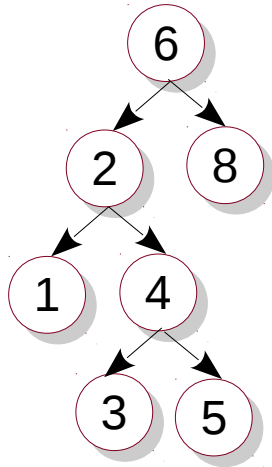
```
DataType Tree::findminiterative() {  
    PtrToNode ptr = root;  
    if (ptr) {  
        while (ptr->left) ptr = ptr->left;  
        return ptr->data;  
    }  
    return -1;  
}
```

Insert

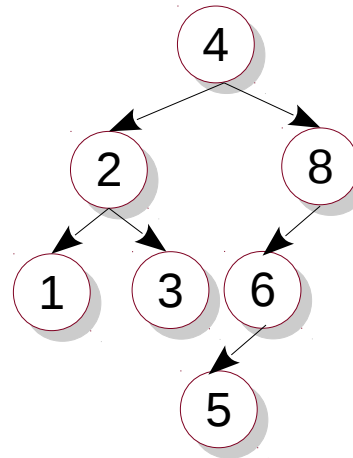
Insert 6, 8, 2, 4, 5, 1, 3



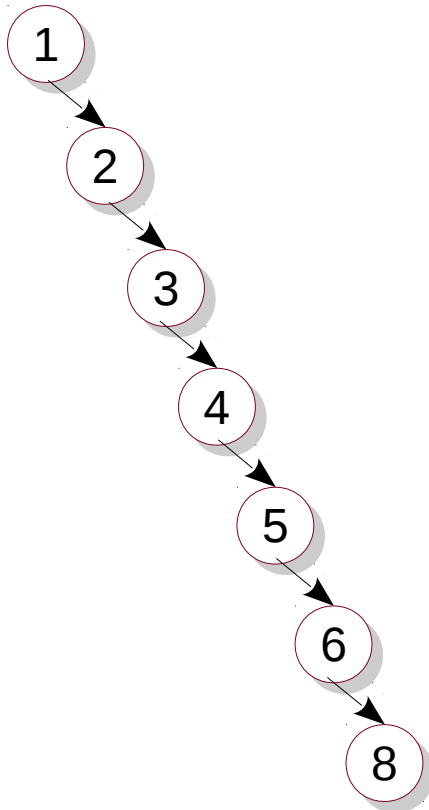
Insert 6, 8, 2, 4, 5, 1, 3



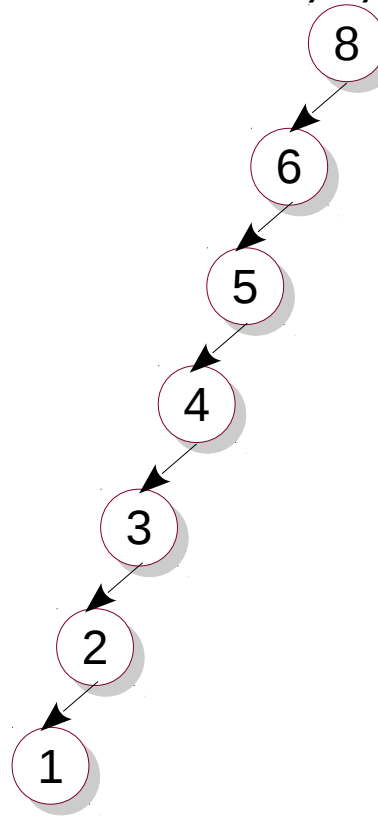
Insert 4, 8, 2, 6, 5, 1, 3



Insert 1, 2, 3, 4, 5, 6, 8



Insert 8, 6, 5, 4, 3, 2, 1



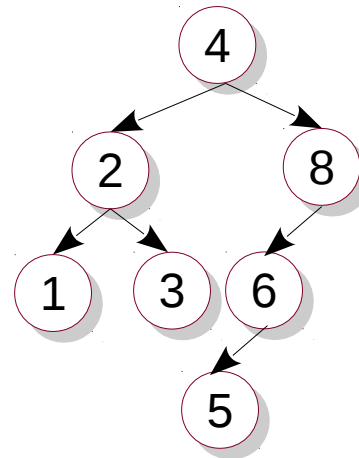
**Perform
inorder
traversal on
the last BST.**

Insert

- Insertion order may change the tree structure.
 - Different insertion orders may form the same tree.
- Inorder traversal prints the values in exactly the **same order**, irrespective of the BST structure.
 - This is also the sorted order.
- The first insertion forms the root.
- Insertions always happen at leaves.
 - New node cannot be added as an intermediate node.
- Insertion order decides the tree height.
 - Tree height affects efficiency/complexity of operations.

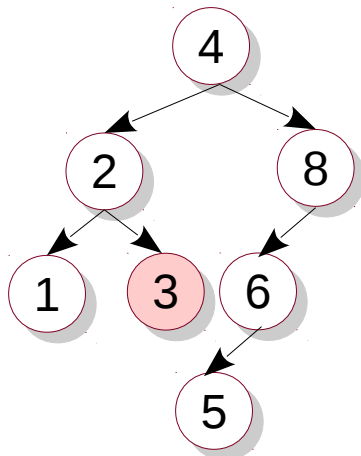
Insertion Orders

- For this BST, find three different insertion orders.
 - 4, 2, 8, 1, 3, 6, 5
 - 4, 8, 2, 6, 1, 3, 5
 - 4, 2, 1, 3, 8, 6, 5
 - ...

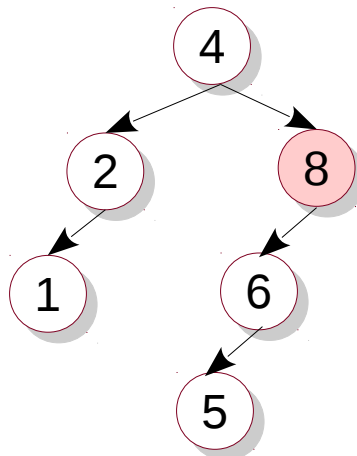


Remove(value)

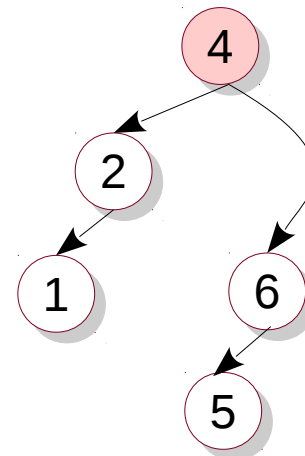
- Search for the node **n** to be removed.
- If **n** is a **leaf**, remove **n** from its parent.
- If **n** has **one** child **c**, make **c** the child of **n**'s parent.
- If **n** has **two** children, scratch your head.



Remove(3)



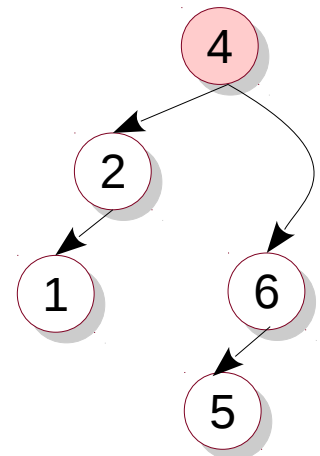
Remove(8)



Remove(4)

Remove(value)

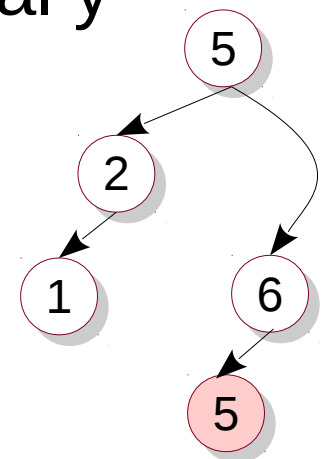
- We would like to convert this complicated remove into another simpler remove.
 - That is, convert this case of two children into a case of one child or zero children.
 - For instance, remove(4) can be converted to remove(5) or remove(6) or remove(2) or remove(1).
 - Which one would be the **best**, in general?
- **General strategy:**
 - **Copy** the smallest value from the right subtree here.
 - Recursively delete that smallest value.



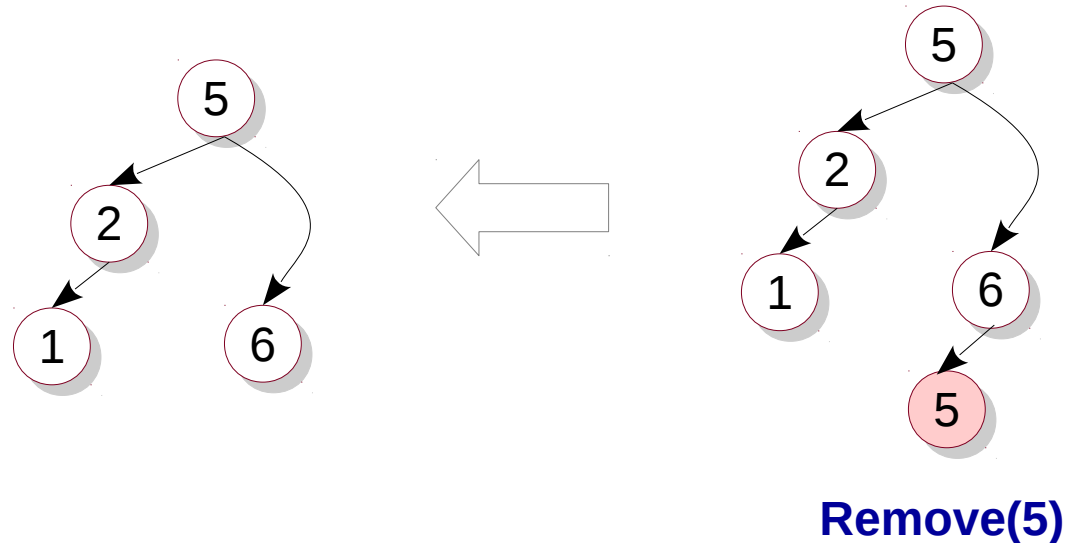
Remove(4)

Guarantees on the smallest value

- If x is the value to be deleted, and y is the smallest value in its right subtree,
 - There are no values in the BST between x and y .
 - The node with y value cannot have two children.
 - In fact, y cannot have a left child.
 - When y replaces x , after removing original y node, the BST structure is not affected.
 - Removal of a node with two children does not result in further removal of another node with two children.

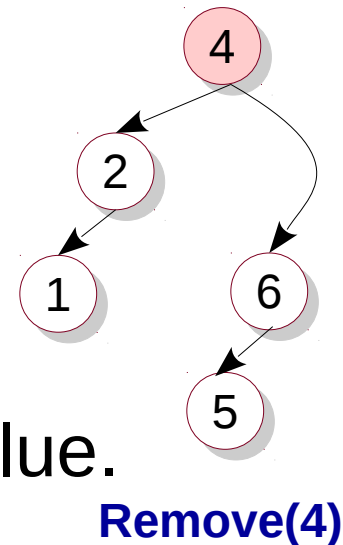


Remove(value)

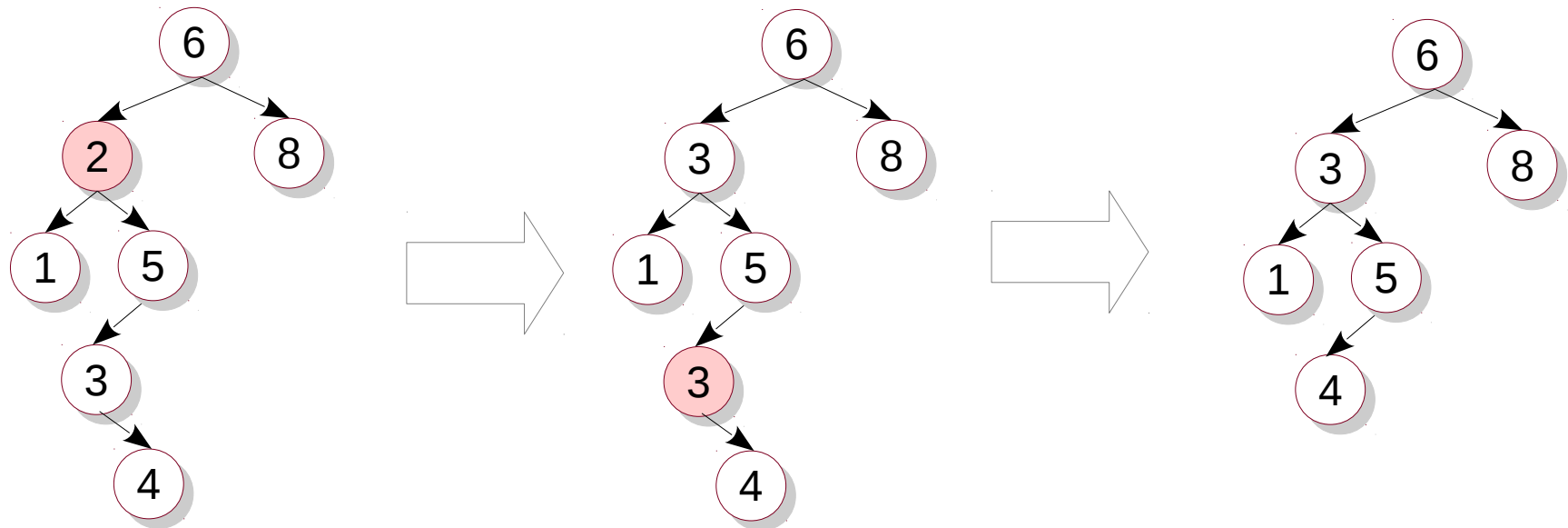


- **General strategy:**

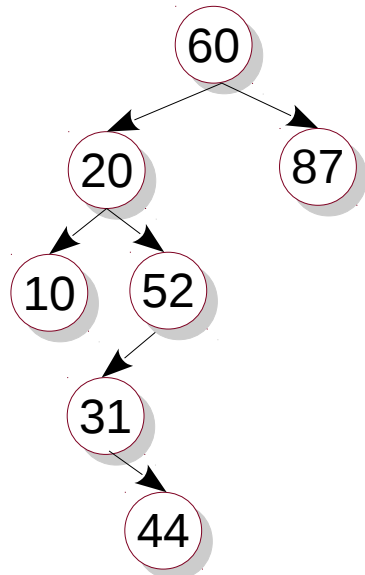
- Copy the smallest value in the right subtree here.
- Recursively delete that smallest value.



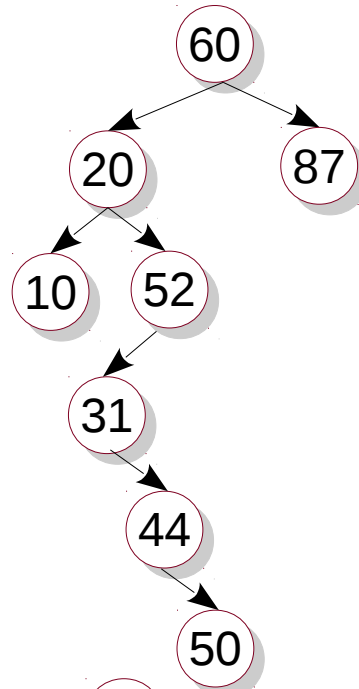
Remove(value)



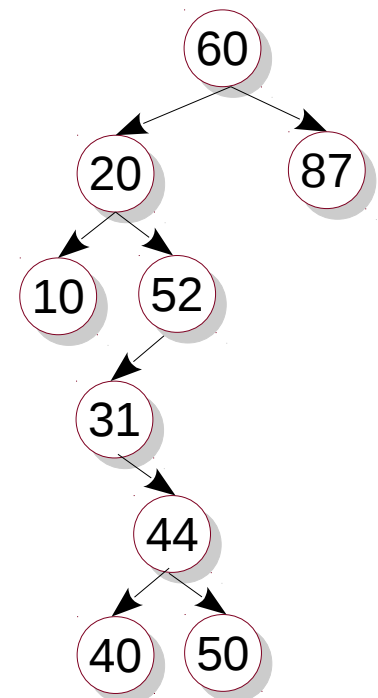
Classwork



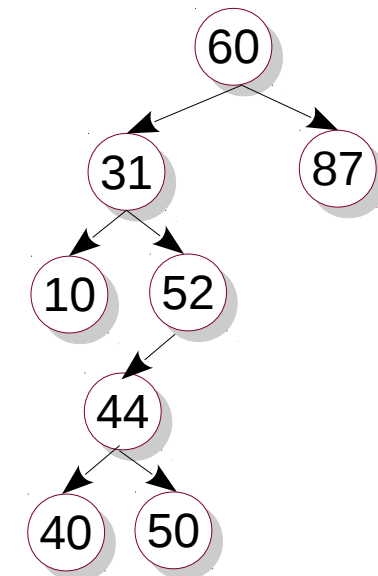
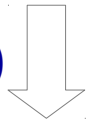
insert(50)



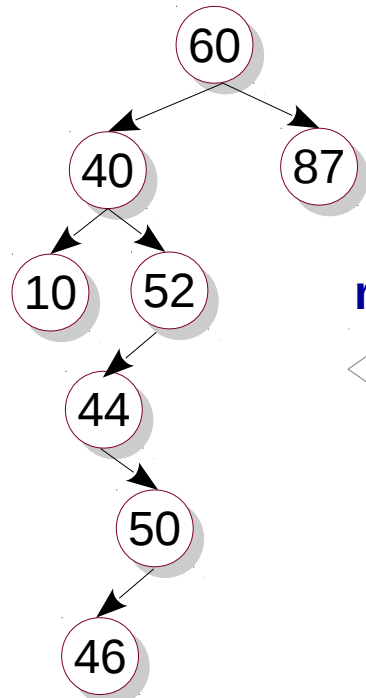
insert(40)



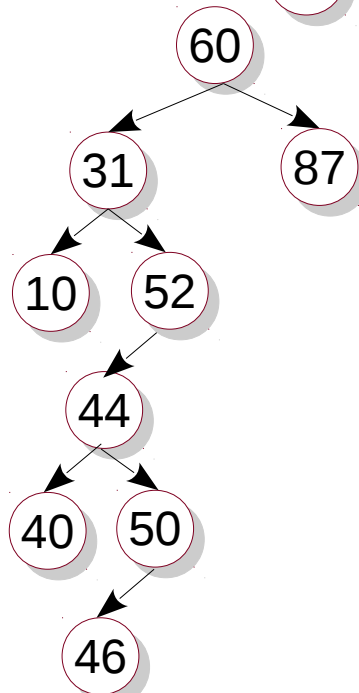
remove(20)



remove(31)



insert(46)



Time Complexity

- **Search**

- One may get tempted to conclude it to be $O(\log n)$.
- But it is $O(\text{tree height})$, which could be $O(n)$.

- **Insert**

- Same as that of search.

- **Remove**

- There may be two remove calls.
- Still the complexity does not change. It is same as that of search.

- All the complexities improve if the BST is **height-balanced** (AVL trees, Splay Trees, red-black trees, B trees, ...).

Some Questions?

- What if a BST has duplicates?
- Can a BST node contain strings? Other types?
- Can I store more pointers in a node?

Exercises

- Given a binary tree, find out if it is BST.
- Given an insertion sequence, how would you permute it to achieve the minimum height of the resultant BST?
 - See if your answer has a resemblance with binary search in a sorted array.
- Count the number of leaves in a BST.
- Print a BST in a level-order (breadth-first) manner.
- Write a program to print values in a BST in reverse-sorted order.

Learning Outcomes

- **General trees**

- Insertion, traversals
- Implementation in C++
- Application: Directory listing

- **Binary trees**

- Insertion, traversals, implementation
- Application: Expressions

- **Binary Search Trees**

- Insertion, search, removal
- Implementation

