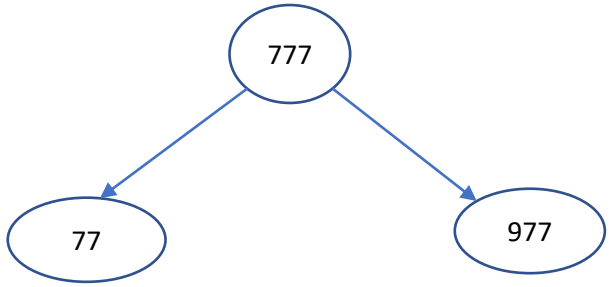
Binary Search Tree

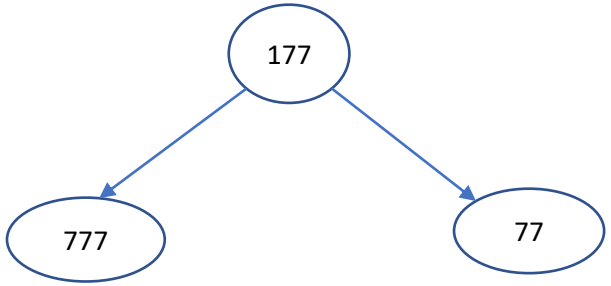
Binary Search Tree

- A binary **search** tree is a very commonly used data structure.
  - Think of it as a special case of a binary tree.

- A *Binary Search Tree* (or *BST*) is a tree where
  - a node has between 0 to 2 children.
  - The children are the left and right subtree (just as we saw in a Binary Tree earlier)
  - The values held by the left subtree are **less** than or **equal** to parent node's value.
  - The values held by the right subtree are **greater** than parent node's value.

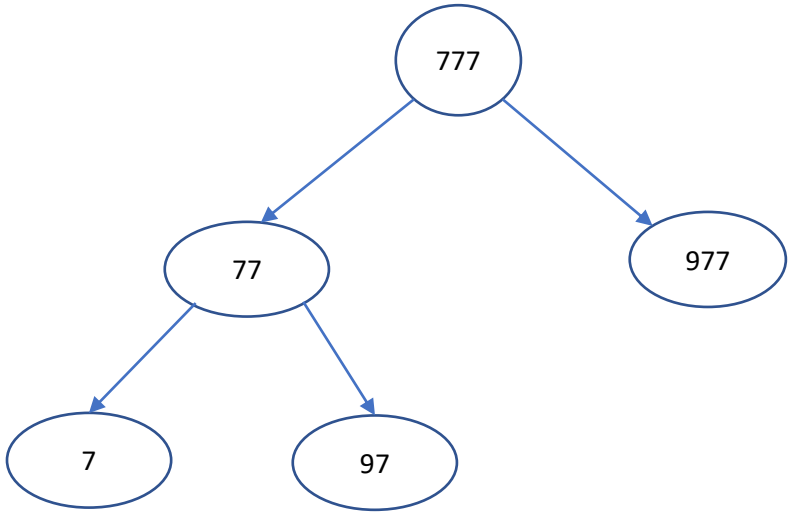
- Which of these is a binary search tree?

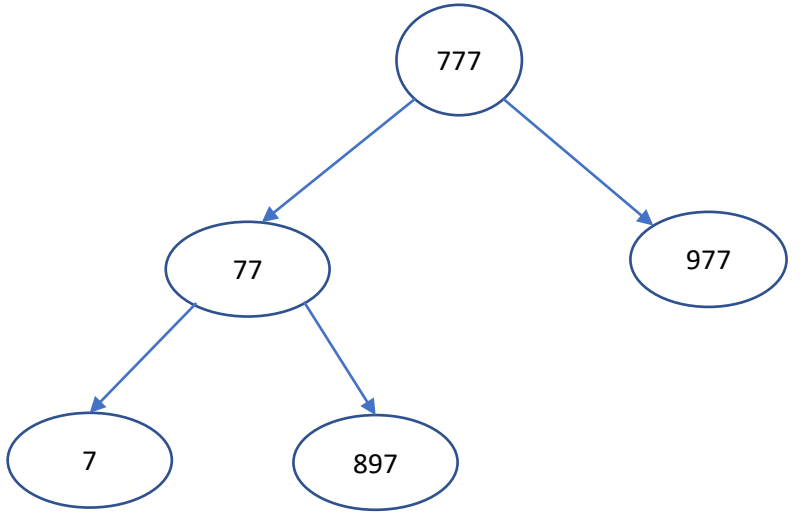




Binary Search Tree

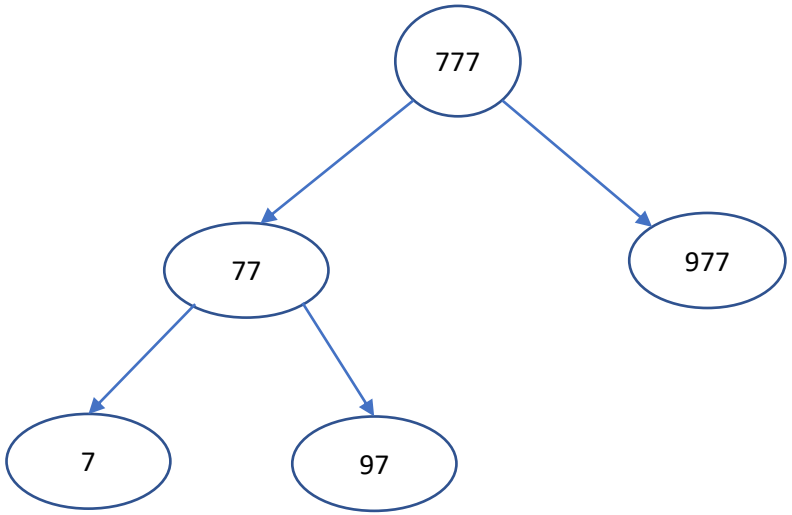
- Which of these is a binary search tree?





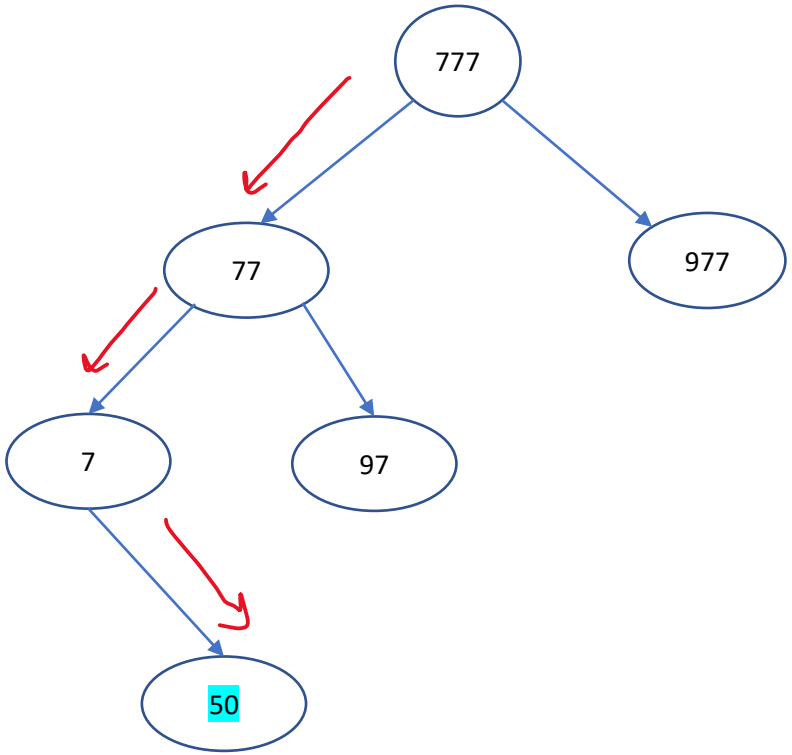
Binary Search Tree

- Where would you insert 50 ?
  - See answer diagram on next slide



Binary Search Tree

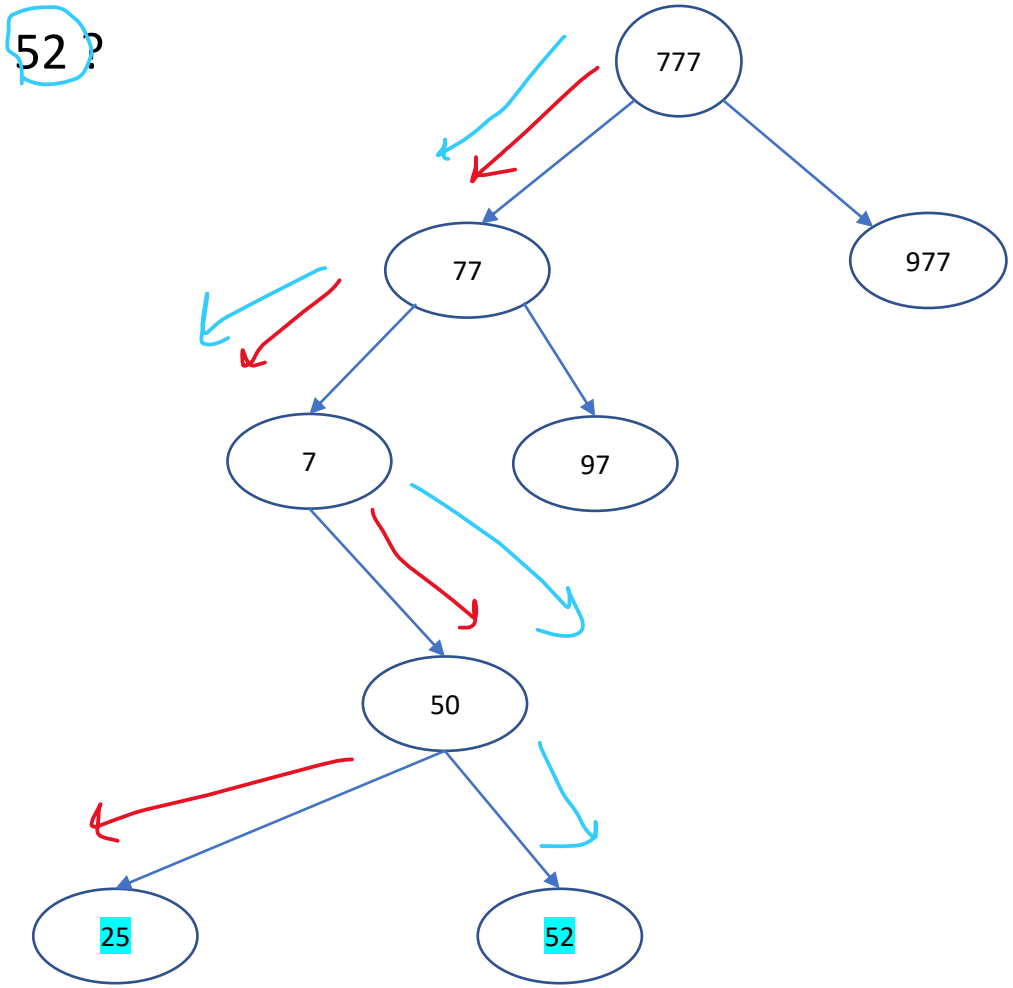
- Where would you insert 50 ?



- Now, where would you insert 25 and 52 ?
  - See answer next slide

Binary Search Tree

- Where would you insert 25 and 52 ?



Binary Tree

- So, the data is sorted in a binary search tree (but not necessarily in a binary tree).
- As a result, we can run efficient searches in a BST.

- The way u would search in a BST would be

  - Start at rootNode (or whatever node u want to start the search)
  - If rootNode.value is the value u r searching, we are done.
  - Else, the search will continue at:
    - Left subtree, if the value we are looking for is < rootNode.value
    - OR
    - Right subtree, if the value we are looking for is > rootNode.value

Binary Tree

- So, after each comparison:
  - if we find the value, we are done.
  - If not, then we are **dividing** the search space into ?

- Time complexity?

- Space complexity (just the search aspect, not the memory already taken by the BST)
  - Recursive search ?
  - Iterative search ?

Binary Search Tree

- Pseudocode for a BST search
- Lets say u r searching for node with value X.

1. Set currentNode = rootNode
2. Start at the currentNode
3. If currentNode is null, value X was not found. Return null.
4. Compare value at current node with search value X
   - If currentNode.value == X
     - Found the node, return it.
   - Else if currentNode.value > X // value we are looking for is smaller, so go down left child
     - Set currentNode = currentNode.Leftchild
   - Else if currentNode.value < X // value we are looking for is bigger, so go down right child
     - Set currentNode = currentNode.Rightchild
   - Go to step 3

Time complexity

- If you really think about it, when u search in a BST:
  - u start at the root node,
  - and then keep going down towards the leaf level
  - path taken to leaf level depends on values in the BST and the value being searched.

- So this means
  - Most that u would have to traverse down the BST would be the **height** of the BST
  - Since u start at root node, and in the worst case, u would reach a leaf.
  - When u reach a leaf,
    - either u find the value there,
    - or the value is not in BST.

- In other words, the worst case complexity of a BST search is the **height** of the BST.
  - Lets see on the next slide what this height is.

- We talked about the height of a binary tree in an earlier class (applies to a BST also)
  - Copying the lines below from the earlier slide deck:

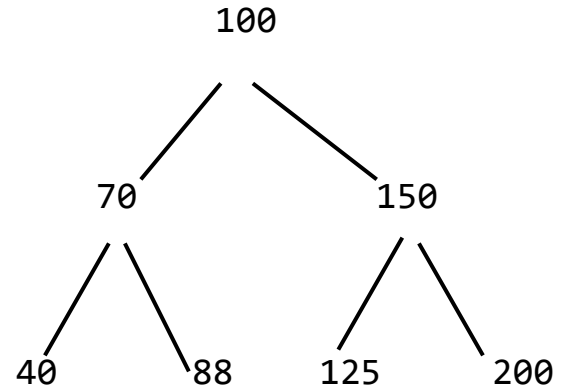
  - If there are N nodes in a binary tree (**assume** a complete tree).
  - Then, the height of the tree is $\log_2 N$

  - So we made two statements wrt to BST search time complexity so far:
    1. The worst case BST search complexity is the *height* of the BST
    2. And the height of the BST is $\log_2 N$

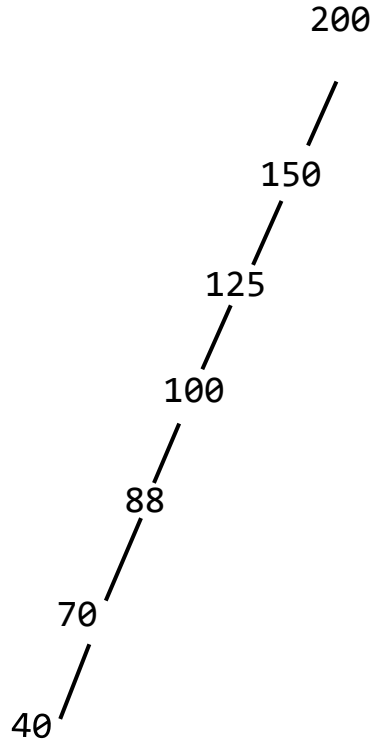
  - However, lets look at scenarios where the 2nd bullet above may not be true.

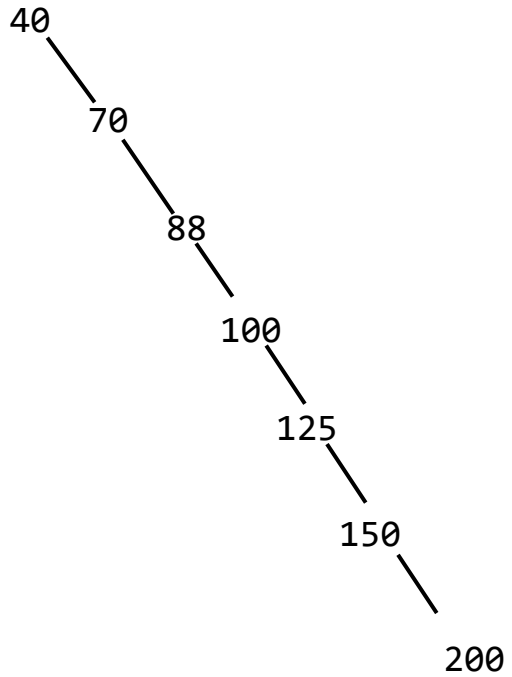
- Copying the 2nd bullet from previous slide:
  - height of the BST is $\log_2 N$

- I have three BSTs on this slide
  - All these BSTs have 7 nodes each.
  - And of course, they follow the BST property.



- Now, the search complexity in all of these will be
  - O (height of the BST)
    - This statement is true for all.
  - But if I say the search complexity is $O(\log_2 N)$
    - Which BST(s) would that apply to?





- What is the search complexity in the two unbalanced trees?

Balance

- It is important for BSTs to be balanced, or close to balanced.
- As we saw in the previous slide, just having a BST doesn't mean that searches will be efficient.
- In the worst case, the BST will be like a linked list, and the search will be linear.
    - Note that in this case, the statement below is still true ☺
        - Worst case search complexity is O (height of the BST)
    - Its just that the height of the BST is N in the extreme case.
    - But for a balanced BST, it would be $\log_2 N$

Traversal complexities

- We learnt about traversals of a binary tree.
- We had these:

    - *Depth first traversal*
        - Pre order
        - In order
        - Post order
    - Complexities:
        - Time ?
        - Space ? (Recursive)
            - Where do we get the space for these recursive traversals?

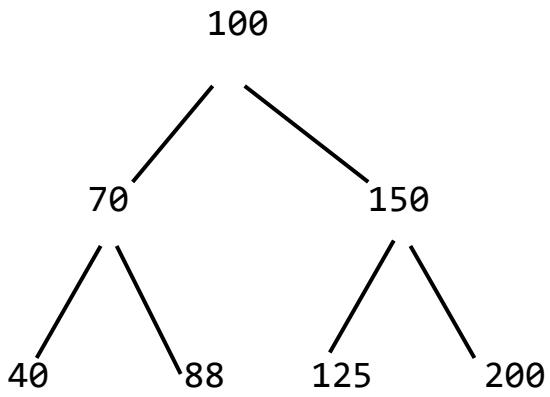
Breadth First Traversal – Level order

As the name suggests, we traverse the nodes along its “breadth”.

Thinking of it differently, we do the following:

- Traverse all nodes in the first level (which is only the root)
- Traverse nodes in the next level,
- An so on, until we get to the leaf nodes.

- In this tree, a level order traversal would result in:
  - 100 70 150 40 88 125 200



  - We will use a queue that we call Q

  - Q: 100 *starting point added to queue Q*
  - Run a while loop … while the Q is not empty
    - Pop from the queue
    - Print the value
    - Push the popped node’s left and right children onto the queue
    - Loop

Level order traversal

- Pseudocode for level order traversal
- Simplified version below, next slide has with all the checks.

```
void LevelOrder( Node node )                    // or BFS ( Node  node )
{
        Queue q;

        q.Enqueue( node );     // initialization, push the root node onto the queue

        while ( ! q.Empty() )
        {
                Node n = q.Dequeue();
                print n.value;
                q.Enqueue ( n.LeftChild );
                q.Enqueue ( n.RightChild );
        }
}
```

Level order traversal

- Pseudocode for level order traversal

```
void LevelOrder( Node node )    // or BFS ( Node  node )
{
        if (node == null)
                return;
        Queue q;
        q.Enqueue( node );     // initialization, push the root node onto the queue
        while ( ! q.Empty() )
        {
                Node n = q.Dequeue();
                print n.value;
                if (n has left child)
                        q.Enqueue ( n.LeftChild );
                if (n has right child)
                        q.Enqueue ( n.RightChild );
        }
}
```

Traversal complexities

Breadth first traversal

- Level order

- Complexities:
  - Time ?
  - Space ?

Min and max values in a BST

- Which node do u think would have the min value in a BST?

- Which node do u think would have the max value in a BST?

- **LAB**
- Write a **recursive** function to return the node with the **minimum** value in a BST
  - Same as above, but return node with **max** value.
- Same as above two, but write an **iterative** version.
- That's a total of 4 functions.