Binary Search Trees

Presenter: Dr Hamed Vahdat-Nejad
Pervasive and Cloud Computing Lab
University of Birjand
http://perlab.birjand.ac.ir/

Binary Search Trees = BST

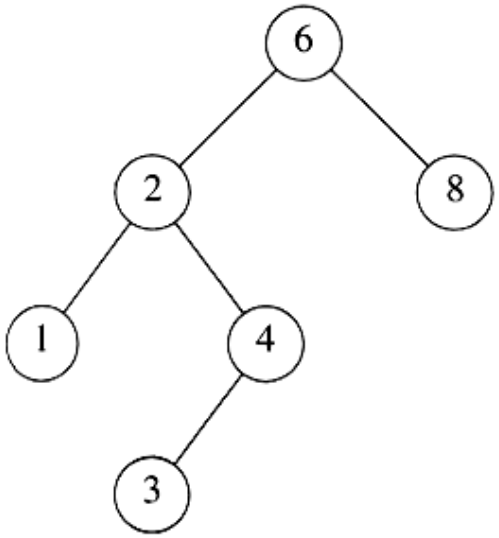
Expected Time Complexity (Average case)			
Operation	Binary Search Tree	Array-based List	Linked List
Search	O(logN)*	-	O(N)
Insert	O(logN)*	O(N)	-
Delete	O(logN)*	O(N)	O(N)

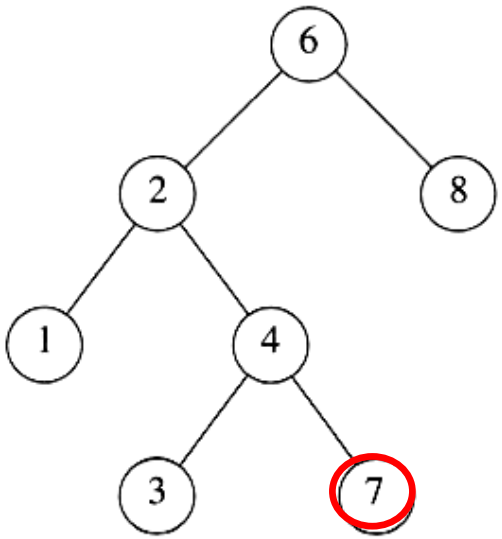
A BST is a data structure that performs searching, insertion and deletion, efficiently.

Binary Search Tree

- A binary tree.
- For every node x, all keys in the left subtree of x are smaller than that in x.
- For every node x, all keys in the right subtree of x are greater than that in x.
- The left and right subtrees are also binary search trees.

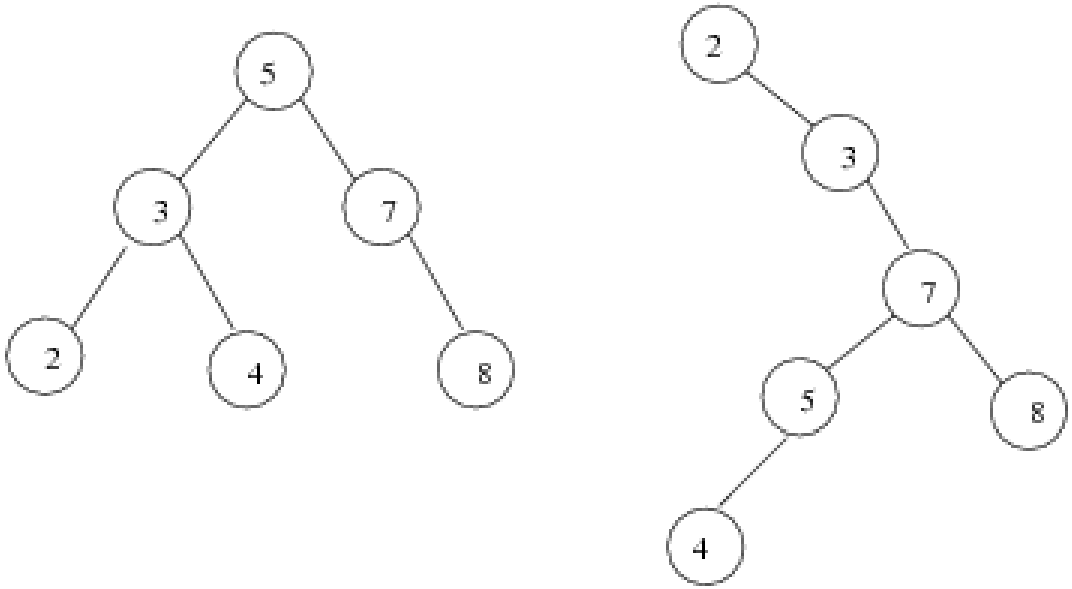
Examples





More examples

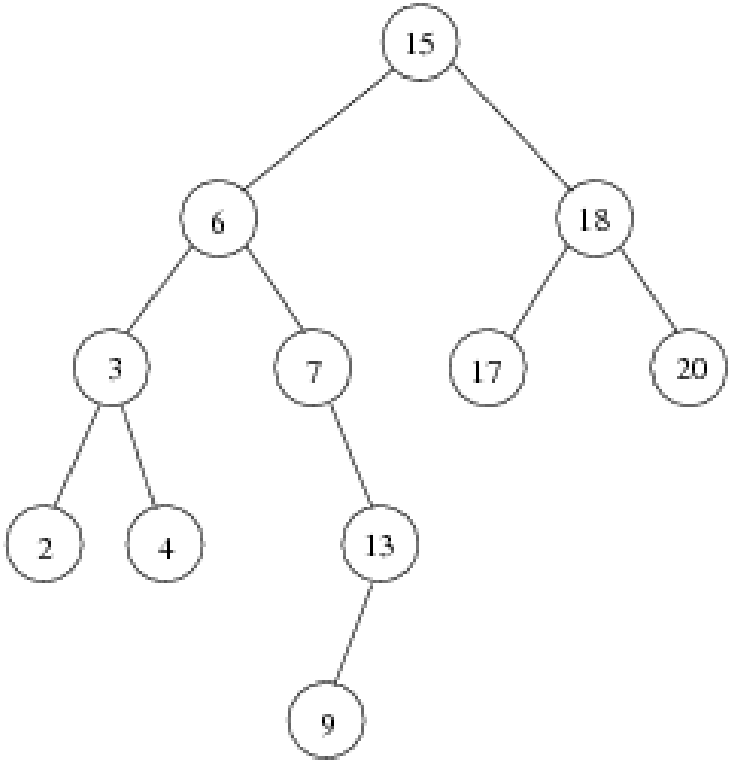
Two binary search trees representing the same set:



- Average depth of a node is O(log N); maximum depth of a node is O(N)

In-order traversal of BST

- Print out all the keys in sorted order



Inorder: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

Implementing a node in BST

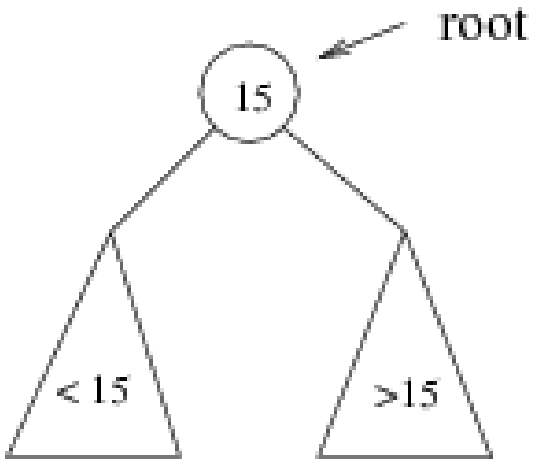
```
Class Node
{
        Node * left;
        int data;
        Node * right;
}
```

Implementing BST

```
Class BST
{Public:
        BST();
        ~BST();
        int NumberOfNodes();
        Node * succ(Node * t);
        Node * pred(Node * t);
        ……
        Node *search(int item);
        void insert(int x);
        void delete(int key);
 Private: Node * root
}
```

Searching BST

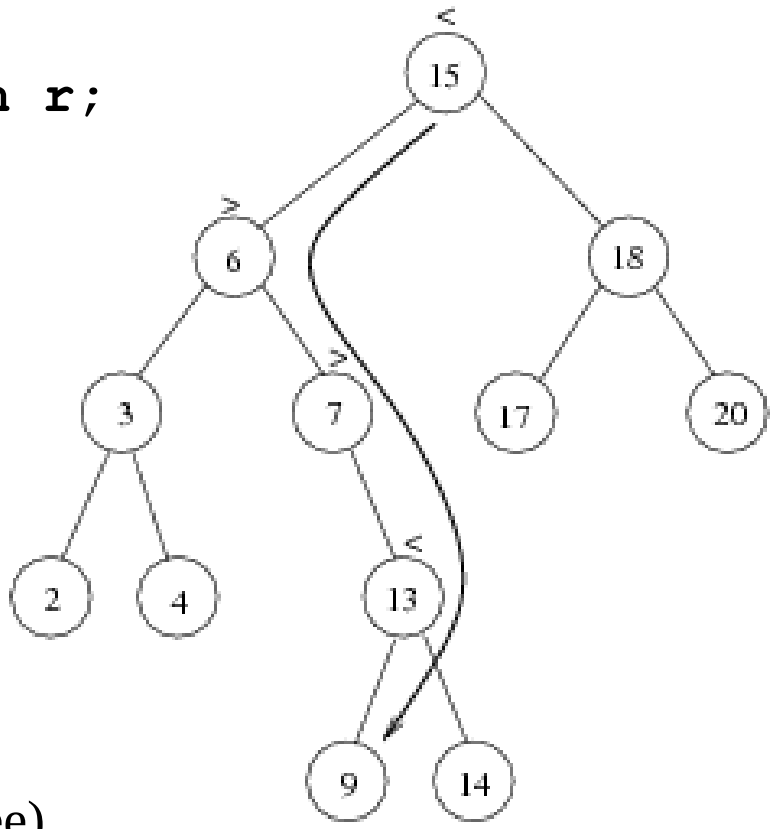
- If we are searching for 15, then we are done.
- If we are searching for a key < 15, then we should search in the left subtree.
- If we are searching for a key > 15, then we should search in the right subtree.



Searching BST

```
Node * search (int item)
{
  Node *r=root;
  while (r) {
    if (item == r->data) return r;
    if (item < r->data)
        r = r->left;
    else r = r->right;
  }
  return NULL;
}
```

Example: Search for 9 ...

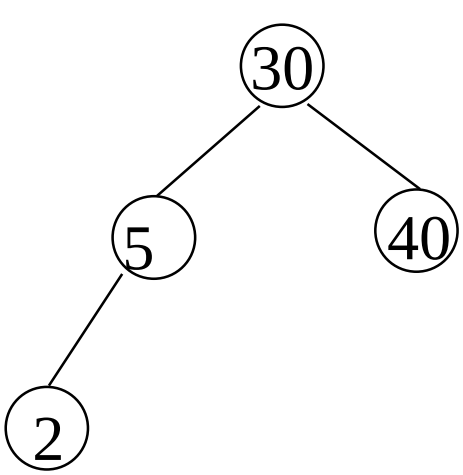


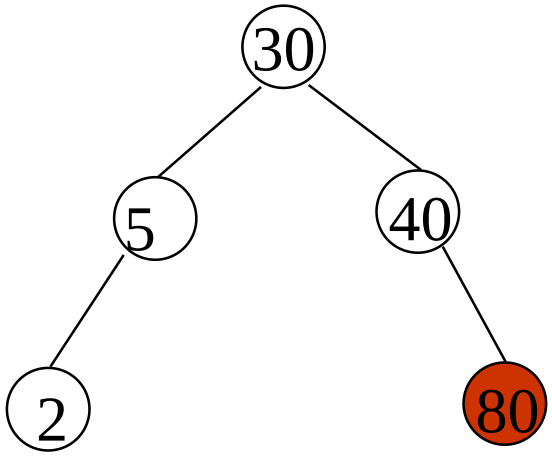
- Time complexity:

-Average and Worst cases: O(height of the tree)

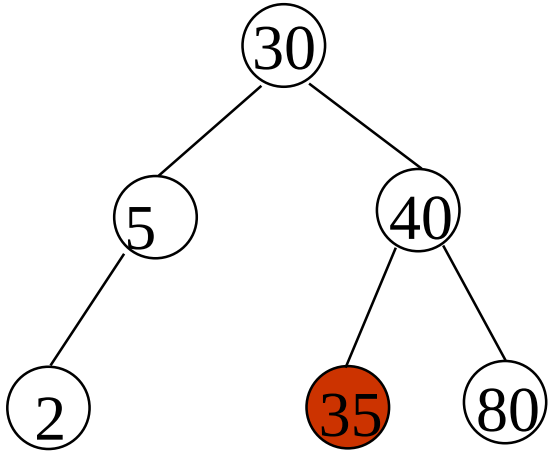
-Best case: O(1)

Inserting a Node in Binary Search Tree





Insert 80



Insert 35

Insert

Concept:

- Proceed down tree as in search
- If new key not found, then insert a new node at last spot traversed

Time complexity:

-Average and Worst cases: O(Height of the tree)

-Best case: O(1)

```
void insert(int x)
{
   Node *t=root;
  if ( t == NULL ) {
    t = new Node(x);
  }
   else if (x < t->data) {
    insert( x, t->left );

  } else if (x > t->data) {
    insert( x, t->right );

  } else {
    // duplicate
    cout<< "The item is already available";
}
```

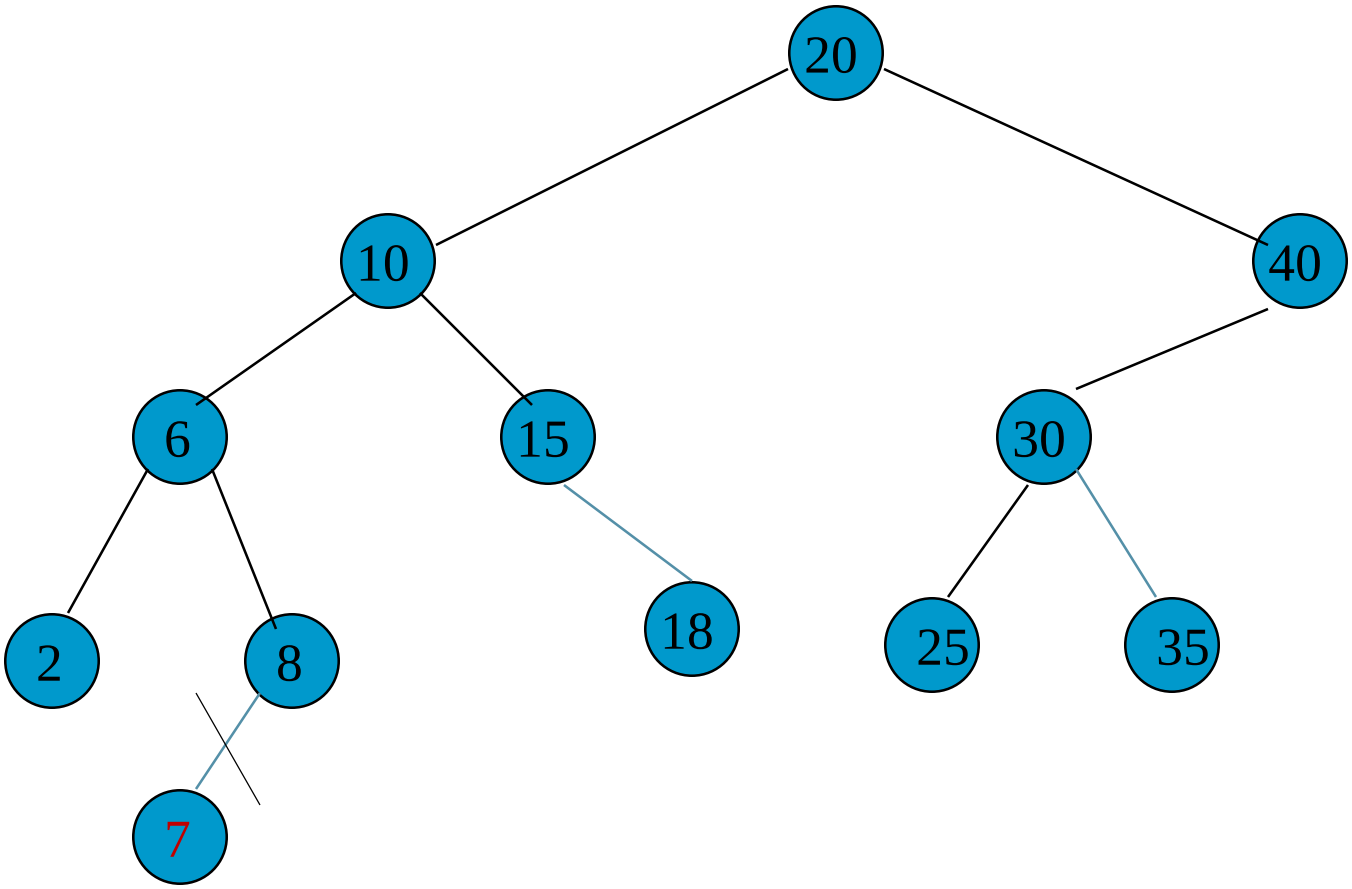
Delete a Node in Binary Search Tree()

- When we delete a node, we need to consider how we take care of the children of the deleted node.
- This has to be done such that the property of the search tree is maintained.

Four cases:

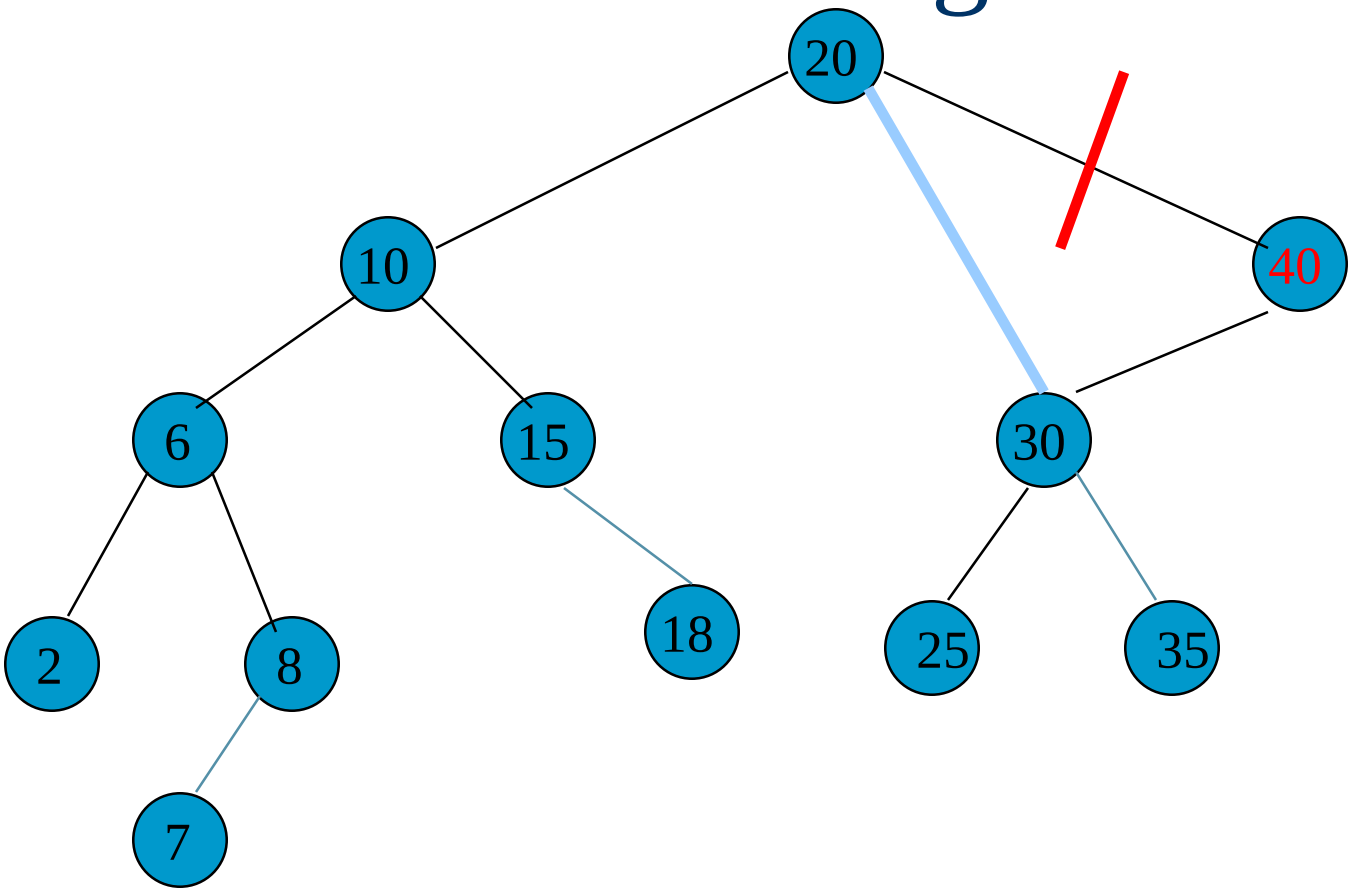
- No node with the key.
- Key is in a leaf.
- Key is in a degree 1 node.
- Key is in a degree 2 node.

Delete From A Leaf



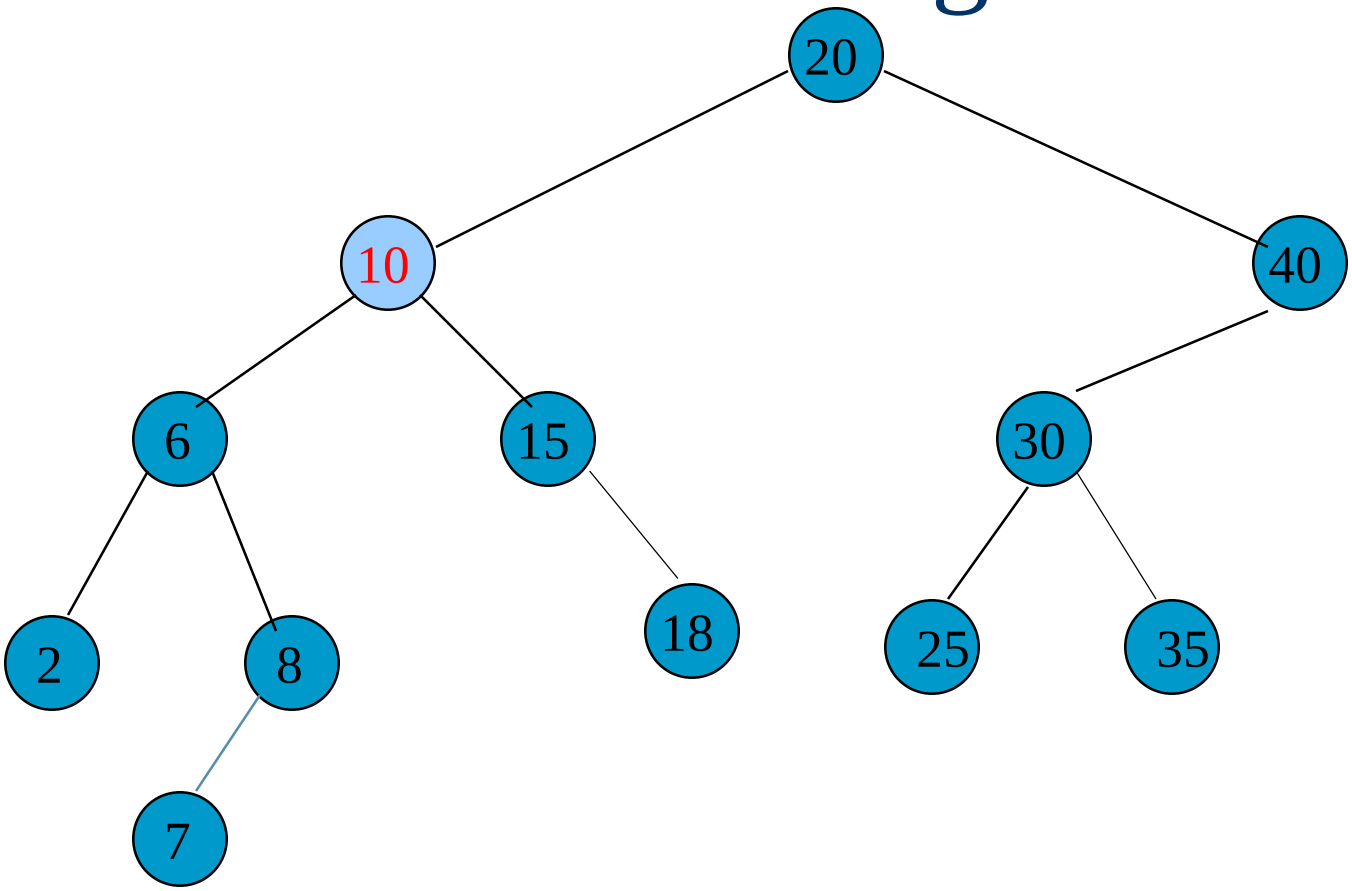
Delete a leaf element. key = 7

Delete From A Degree 1 Node



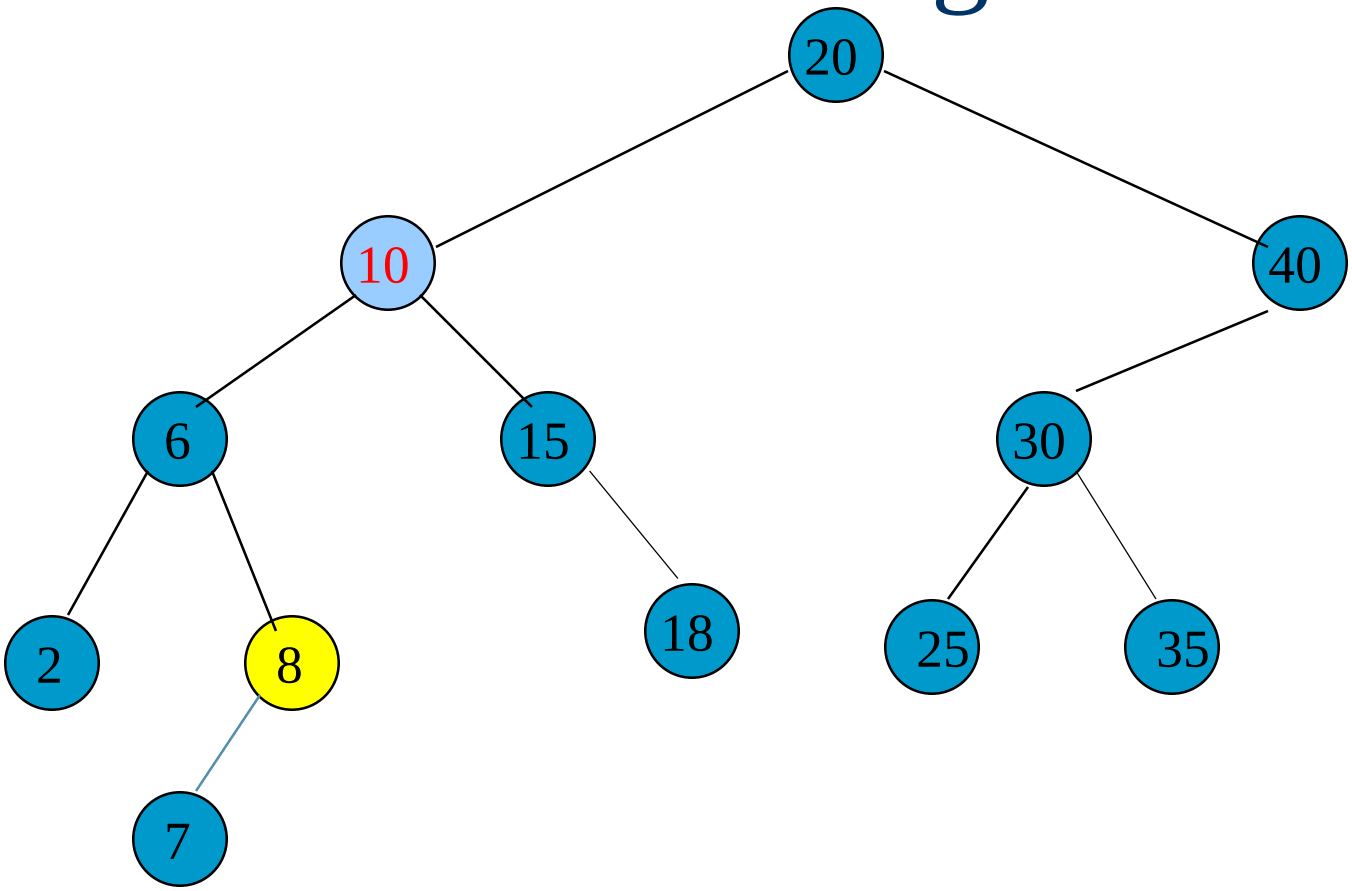
key = 40

Delete From A Degree 2 Node



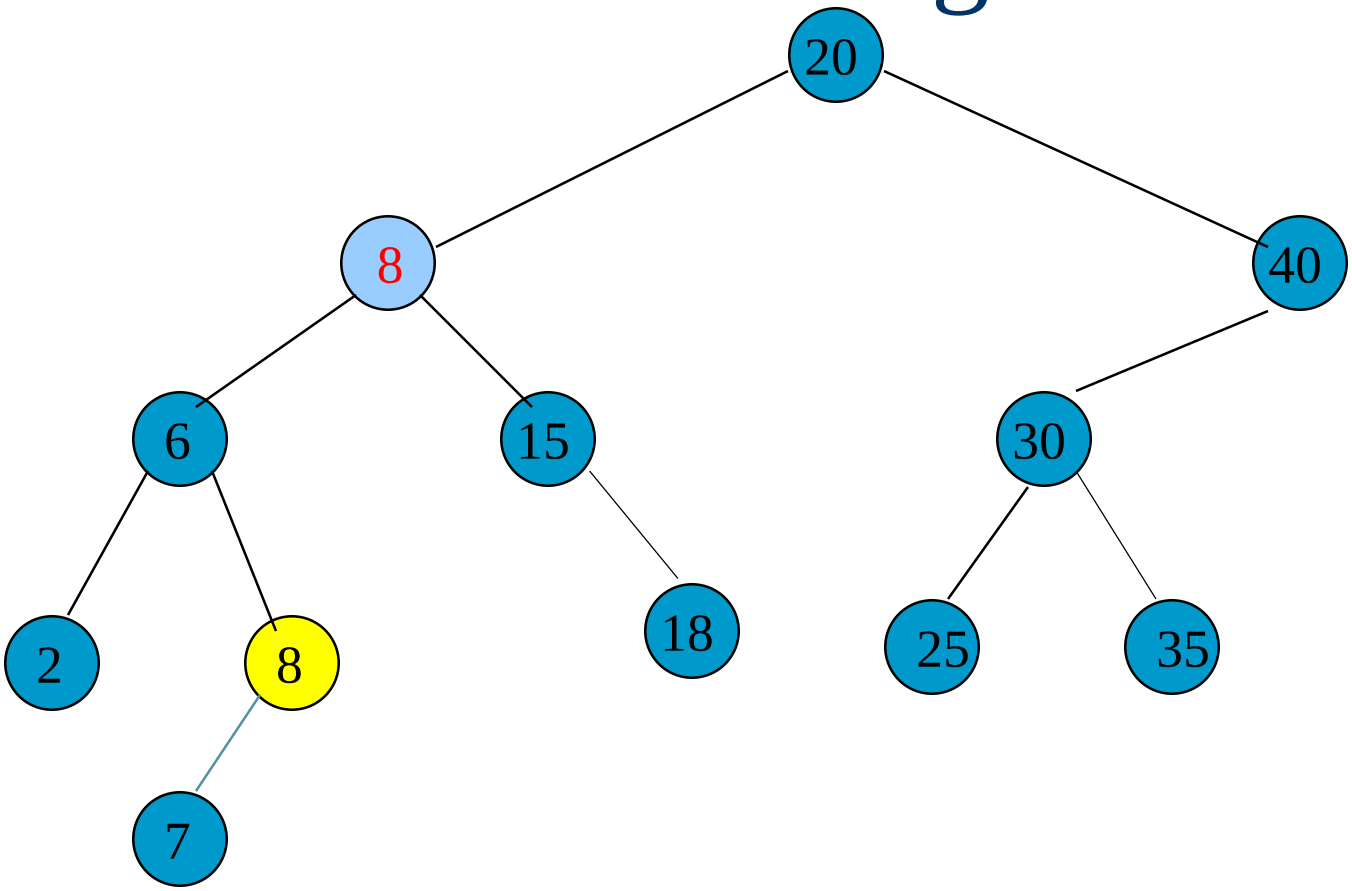
Replace with largest key in left subtree (or smallest in right subtree).

Delete From A Degree 2 Node



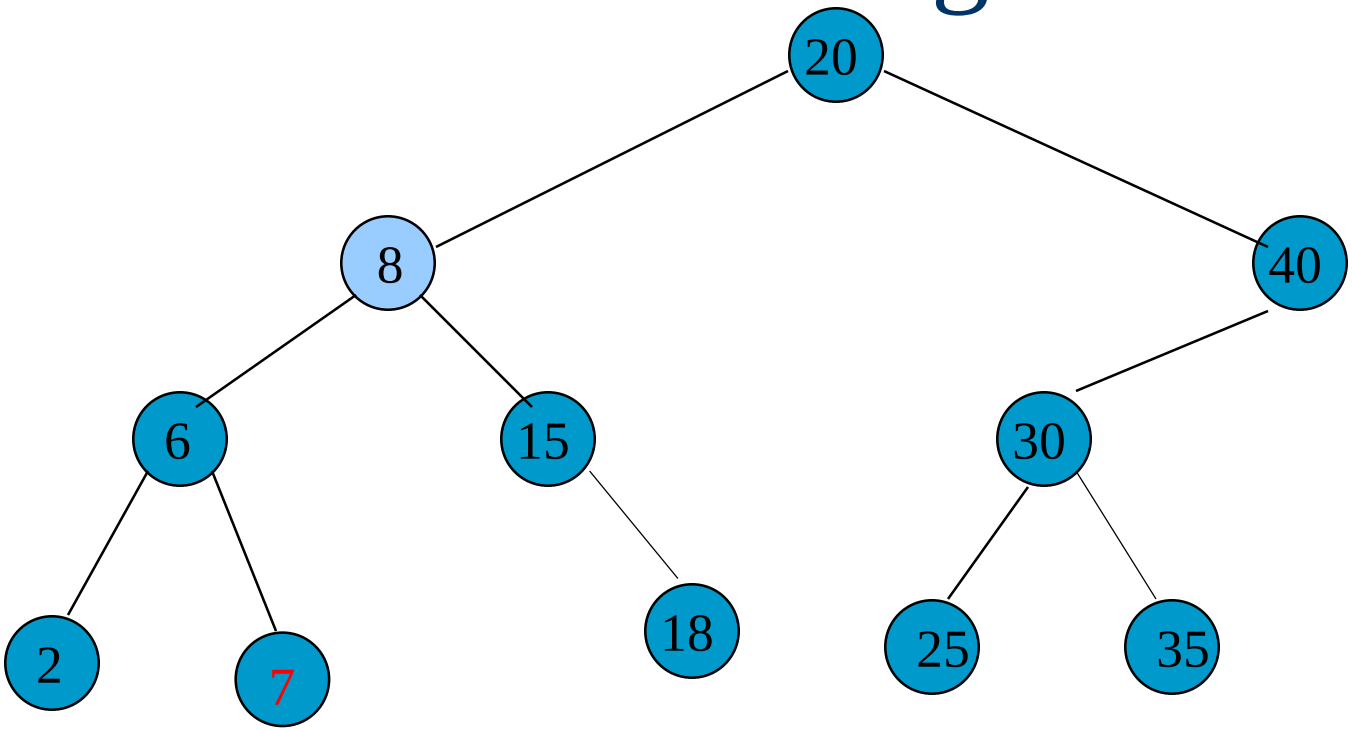
Replace with largest key in left subtree (Rightmost element) or smallest in right subtree(leftmost element).

Delete From A Degree 2 Node



Replace with largest key in left subtree (or smallest in right subtree).

Delete From A Degree 2 Node



Replace with largest key in left subtree (or smallest in right subtree).

Delete Algorithm

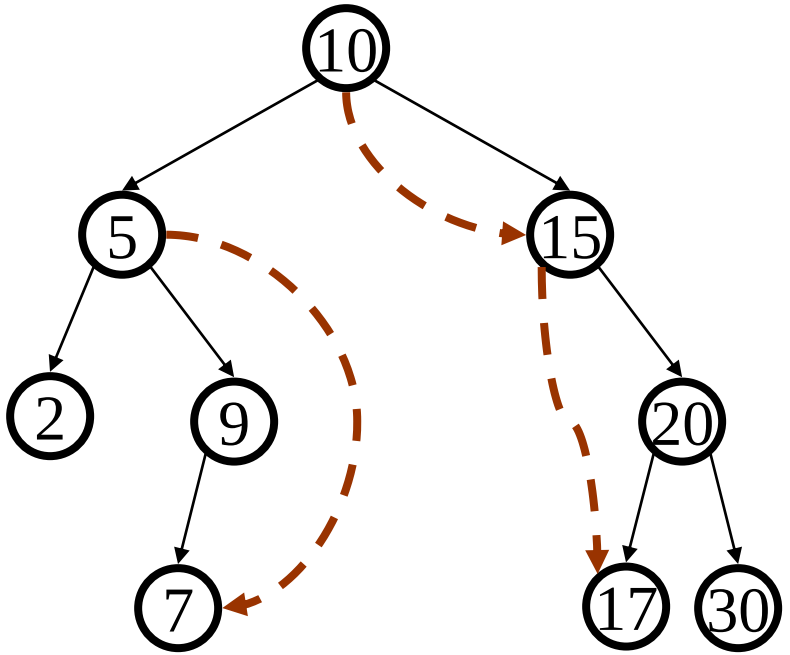
```
void delete(int key) {
  Node * handle=search(key);
  Node * toDelete = handle;
  if (handle != NULL) {
    if (handle->left == NULL) {          // Leaf or one child
      handle = handle->right;
      delete toDelete;
    } else if (handle->right == NULL) { // One child
      handle = handle->left;
      delete toDelete;
    } else {                             // Two children
      successor = succ(toDelete);
      handle->data = successor->data;
      delete(successor->data);
    }
  }
}
```

Successor Node

Next larger node

in this node's subtree

```
Node * succ(Node * t) {
  if (t->right == NULL)
    return NULL;
  else
    return min(t->right);
}
```



Applications of BST

- Removing duplicate elements from a list
- Sorting a list

Conclusion

Binary Search Trees are fast if they're balanced:

What matters?

- Problems occur when one branch is much longer than another
- i.e. when tree is out of balance