Data Structures & Algorithms

Binary Search Trees (BST)

NUST
SCHOOL OF ELECTRICAL ENGINEERING &
COMPUTER SCIENCE (SEECS)

Binary Search Trees

- When we store ordered data in an array, we have a very efficient search algorithm, the binary search

- However, we have very inefficient insertion and deletion algorithms that require shifting data in the array

Binary Search Trees

- To provide for efficient insertions and deletions, we developed the linked list

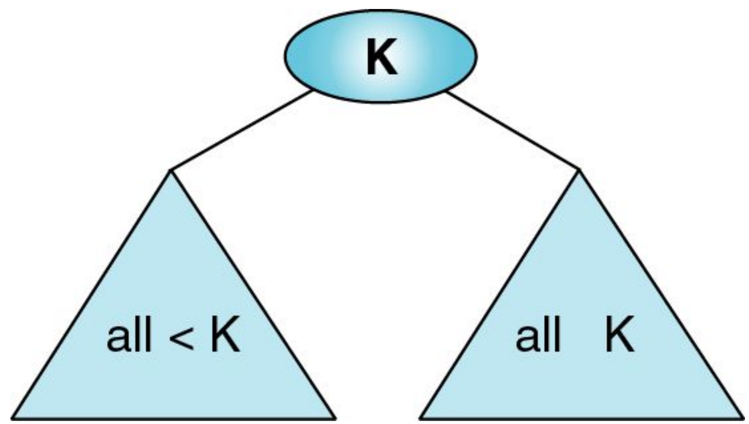
- The problem with linked lists, however, is that their search algorithms are sequential searches, which are very inefficient

- Can't we get the best of both worlds (i.e., efficient search, insertion, and deletion algorithms)?

Binary Search Trees

- The ***binary search tree*** is a binary tree with the following properties:
  - All items (keys) in the left subtree are less than the root's key
  - All items (keys) in the right subtree are greater than the root's key
  - Each subtree is, itself, a binary search tree

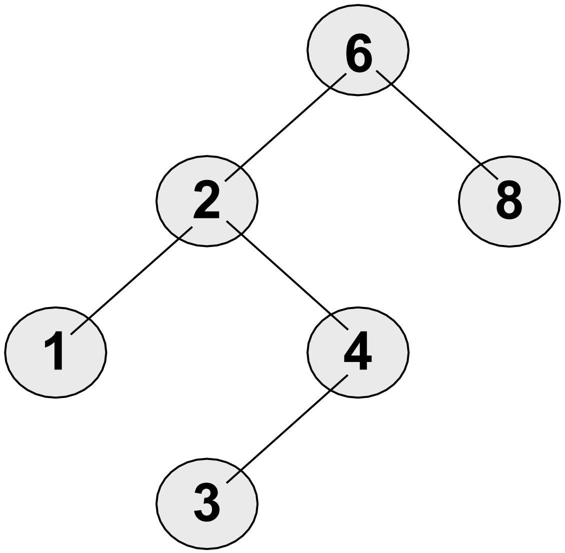
Binary Search Trees

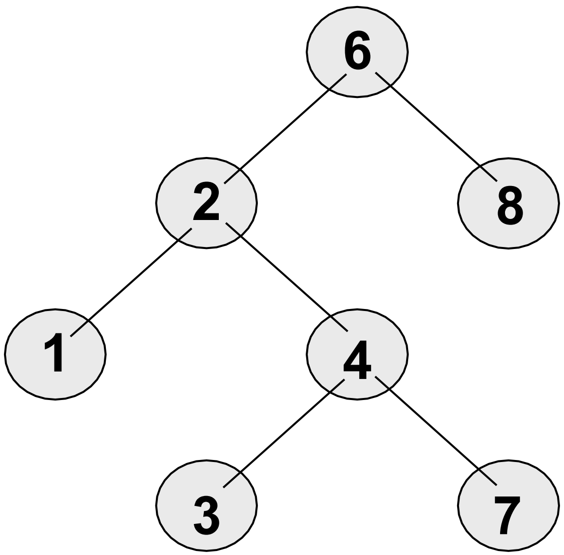


Here you can see the basic structure of a binary search tree

Binary Search Trees vs Binary Trees

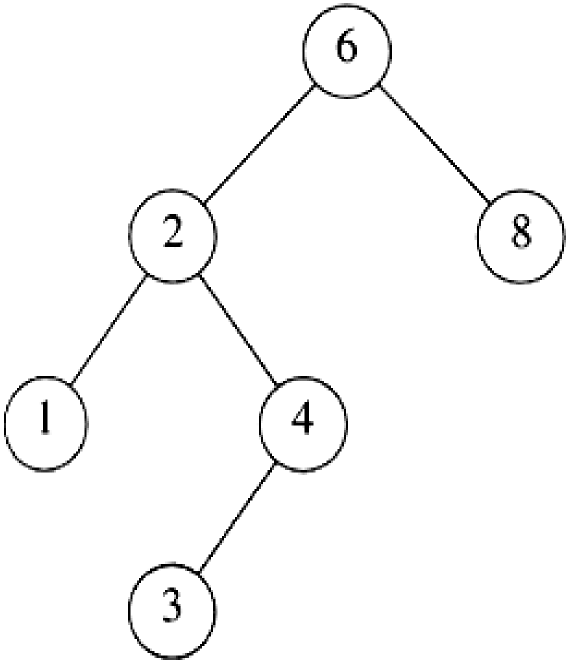
A *binary search tree*



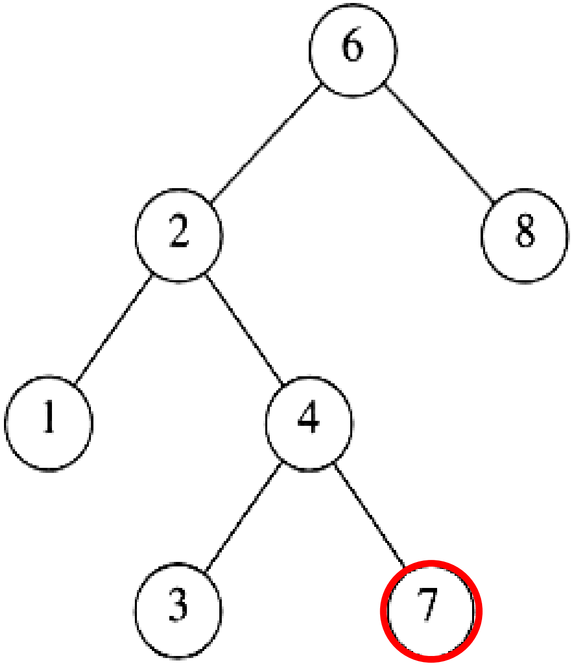


**Not a *binary search tree*,
but a *binary tree***

Examples



A binary search tree



Not a binary search tree

Representation of BSTs

- BST can be represented by a linked data structure where each node is an object
  - In addition to a key field, each node contains fields *left*, and *right*, that correspond to its left child and right child
  - If a child or parent is missing, the appropriate field contains the value NULL
  - The root is only node in the tree, whose parent field is NULL

Binary Search Trees

- **Note:** we have written this definition in a way that ensures that no two entries in a binary search tree can have equal keys
- Although it is possible to modify the definition to allow entries with duplicate keys, it makes the algorithms somewhat more complicated
- If duplicates need to be accounted for, they can be stored in another data structure (e.g., list)

Search Tree Property: Duplicate Keys

NUST SEECS

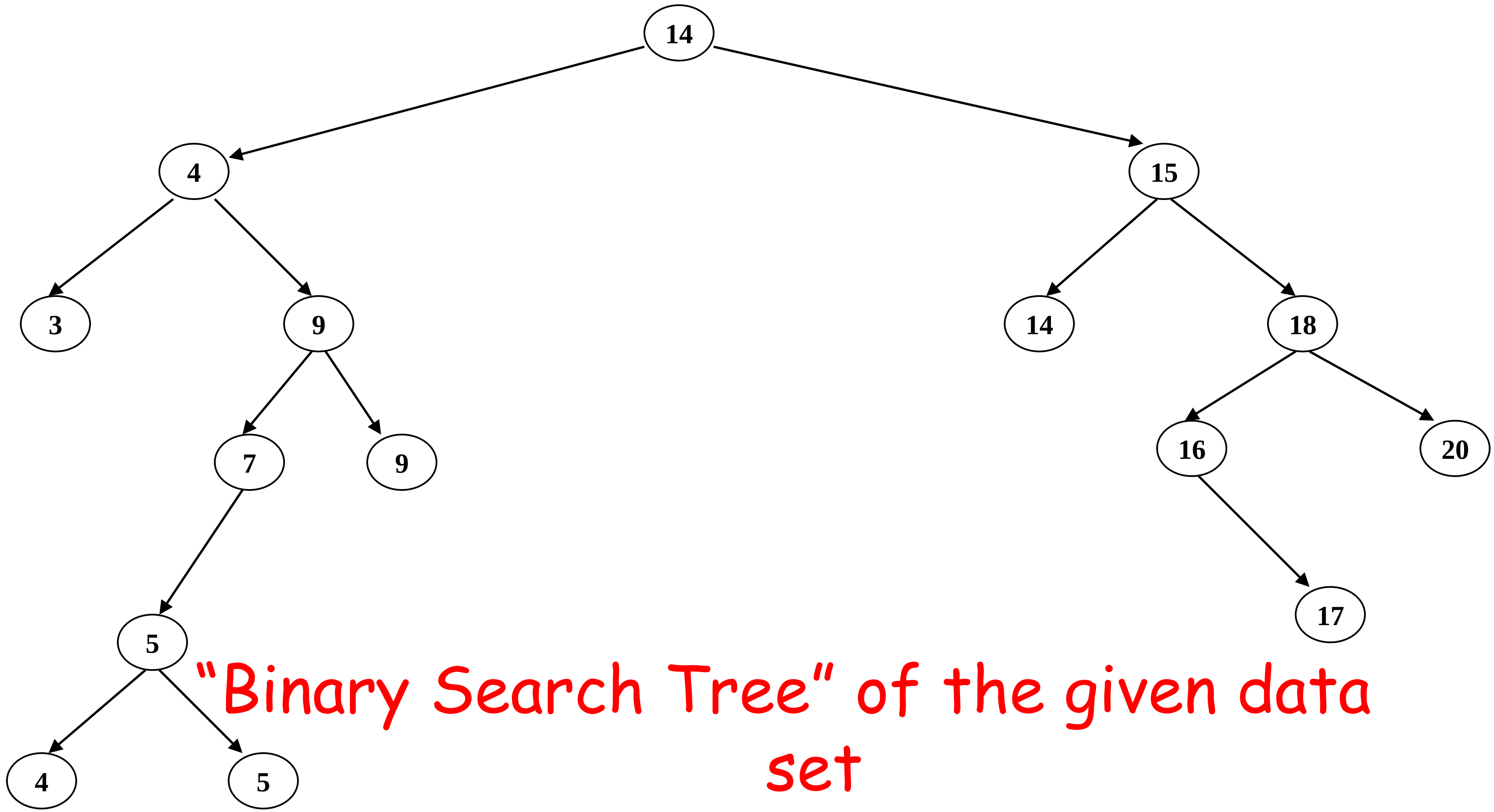
- The duplicate keys can all be kept in the left subtree, or all in the right subtree. It doesn't matter which we choose, but it matters that the choice is the same for the whole implementation.

- Another issue: with duplicate keys, it is important to have extra operations in the interface: getAll, and removeAll

Example

Input list of numbers:

14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



"Binary Search Tree" of the given data set

Implementing a BST

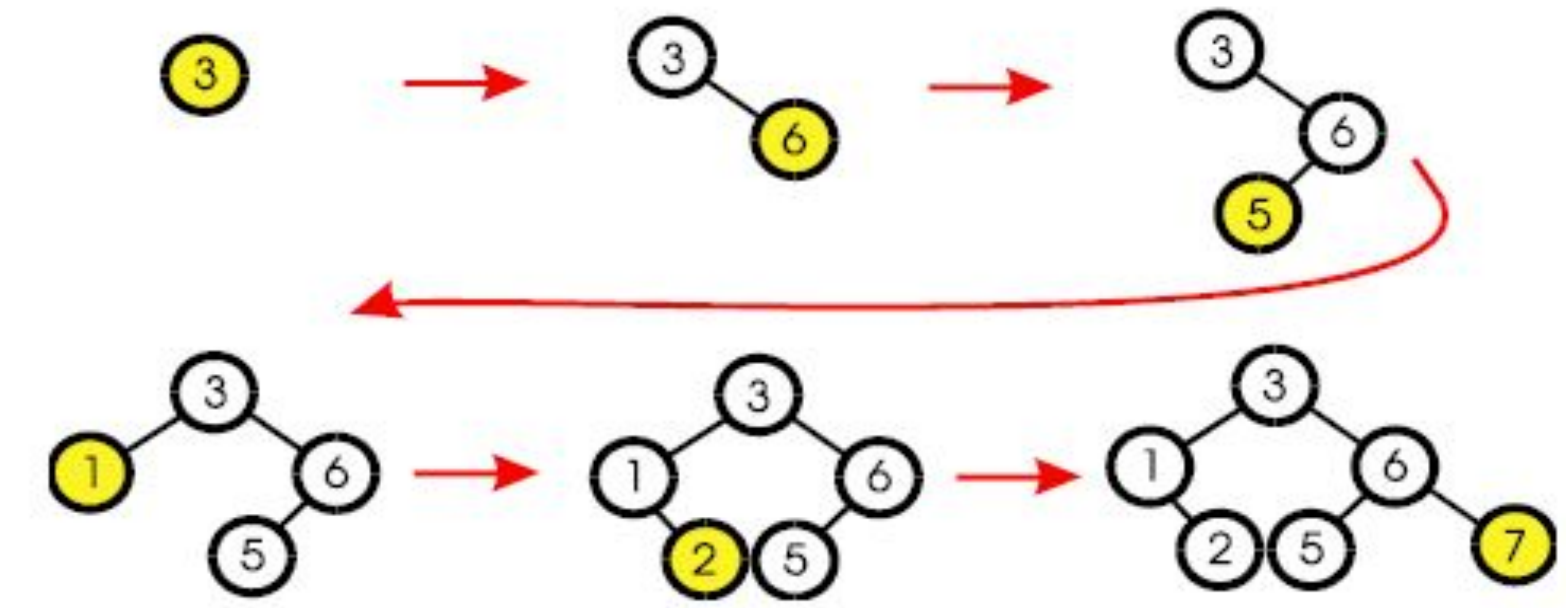
```
class Node
  {
   public:
      int data;
      Node * left,
             * right;

  };
```

Binary Search Trees

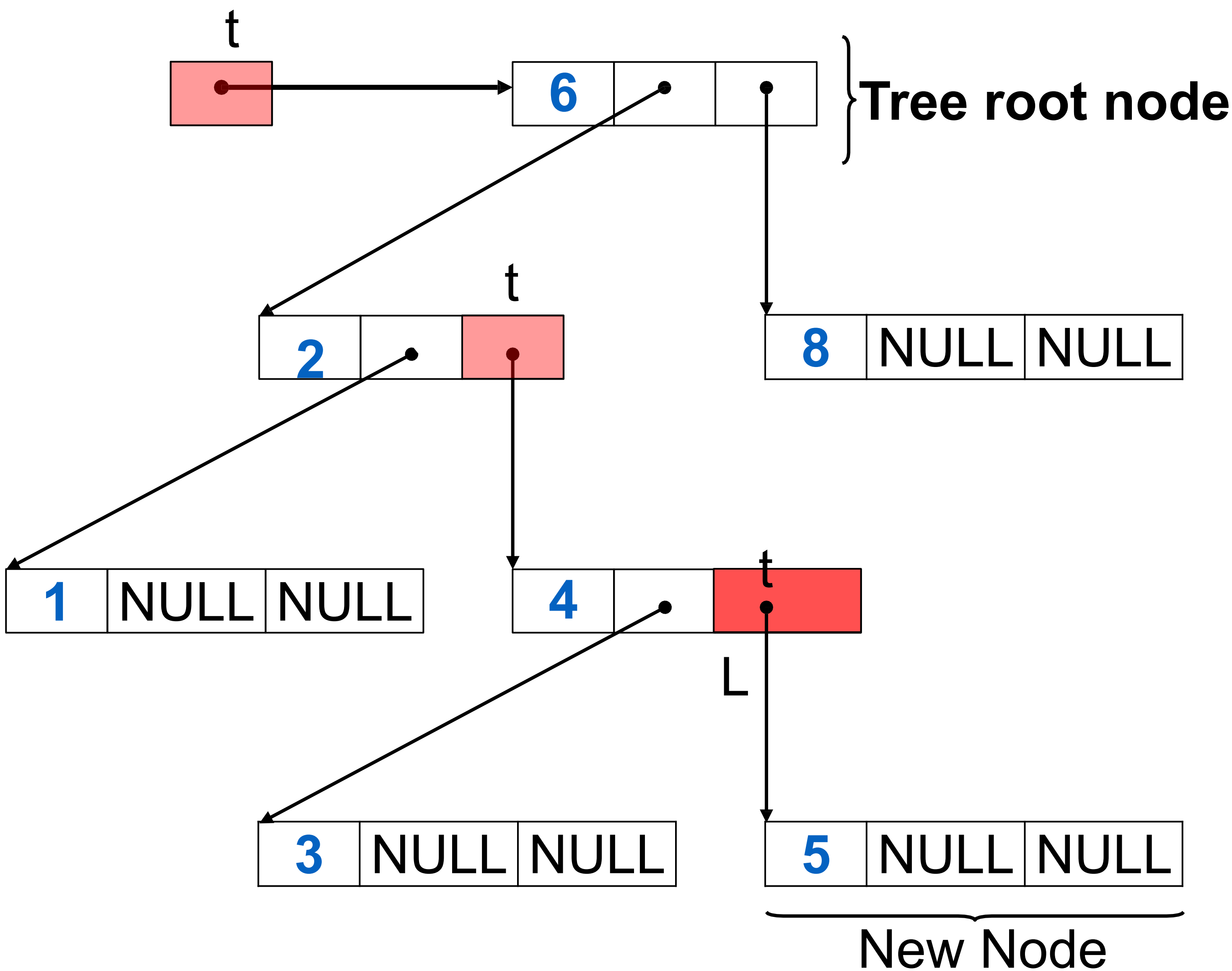
- Operations
  - Insertion
  - Search
  - Traversal
  - Deletion

Inserting a value into a BST

- The first value inserted goes at the root
- Every node inserted becomes a leaf
- Descend left or right depending on value



Inserting Item 5 to the Tree



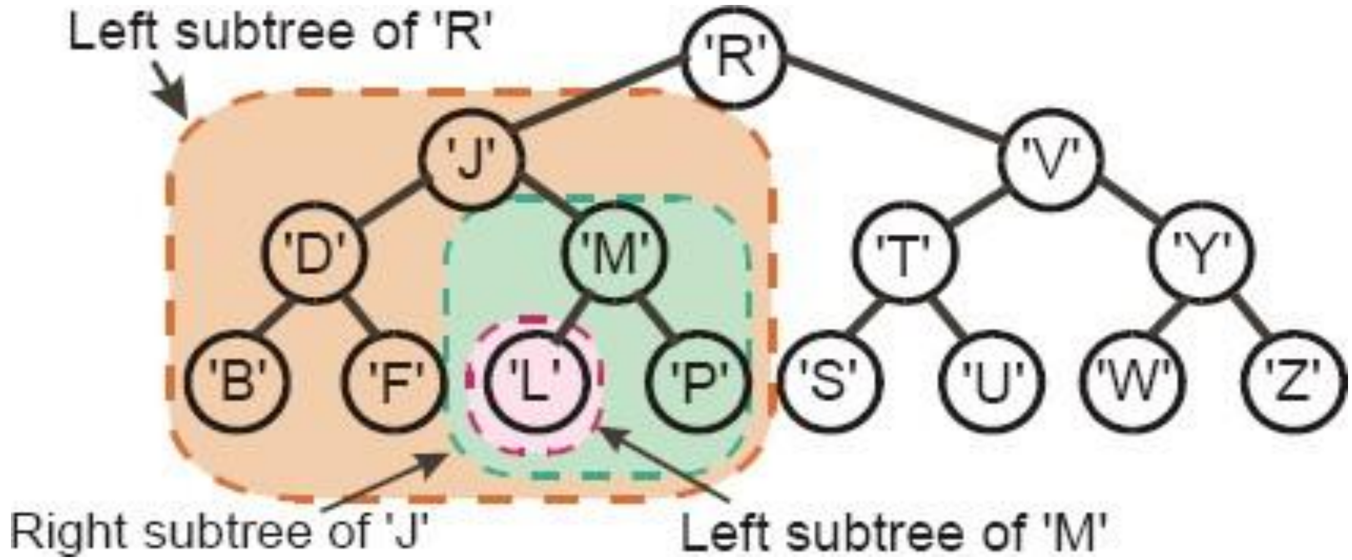
```
void Insert(Node *ptr, int value){
Node * prev = 0;
while (ptr!=0){
	prev = ptr;
	if (value <  ptr->data)
		ptr = ptr->left;
	else if(value > ptr->data)
		ptr = ptr->right;
	else{
		cout<<"Value already exist";return ;}
}
Node * temp = new Node;
temp->data=value; temp->left=0; temp->right=0;

if(prev==0)
	root = temp;
else if (value < prev->data)
	prev->left = temp;
else
prev->right = temp;
}
```

Searching the tree

- Start at the root
- Is target = value at current node?
  - We’re done

- Is target < value at current node?
  - Yes: search left subtree
  - No: search right subtree

Tree Search uses Sub-trees



Search Starts from the Root

Iterative Solution

Implementing bst search

```
bool Search(Node *ptr, int data)
{
        bool found = false;

        while(1)
        {
        If(found || ptr == NULL)
                break;
        If(data < ptr->data)
                ptr = ptr->left;
        else if (data > ptr->data)
                ptr = ptr->right;
        else
         found = true;
         }
    return found;
}
```

Implementing bst search – recursive solution

```
bool Search(Node *temp, int num)
{
        if(temp==NULL)
        return false;
  else if(temp->data == num)
        return true;
  else if(temp->data < num)
        return Search(temp->right, num);
  else if(temp->data > num)
       return Search(temp->left, num);
}
```

Deletion in BST

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.

Implementing a BST

<u>Deletion</u>

To delete a node x from a BST, we have three cases:

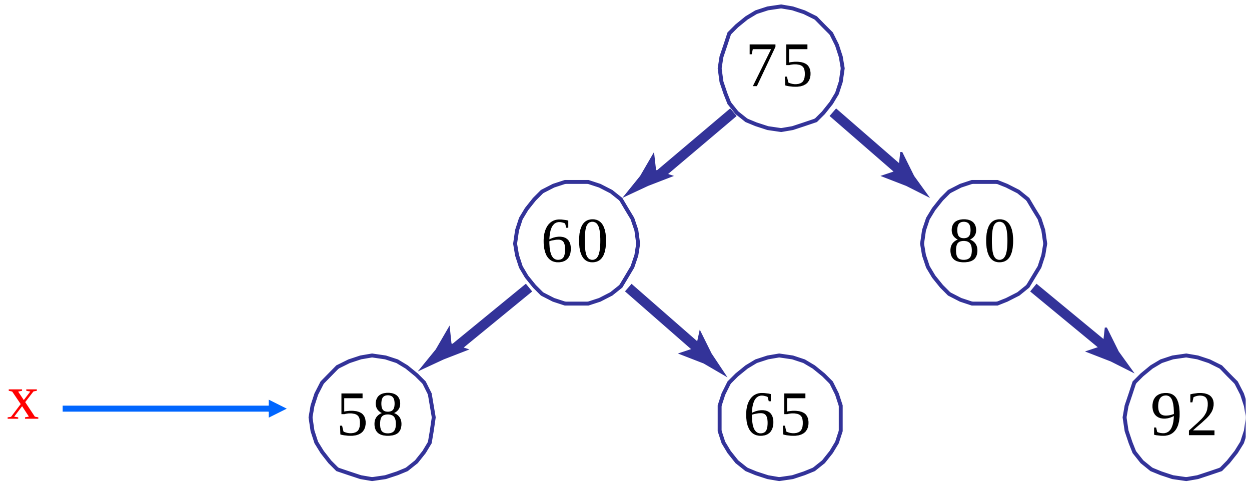
- ✓x is leaf
- ✓x has one child
- ✓x has two children

Implementing a BST

CASE 1:

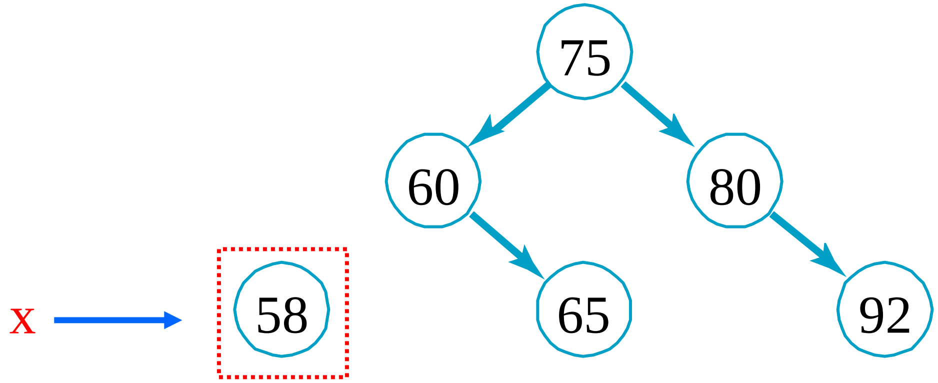
- x is a leaf

Simply make the appropriate pointer in x's parent a null pointer



Implementing a BST

Make the left pointer of x's parent null
Free x

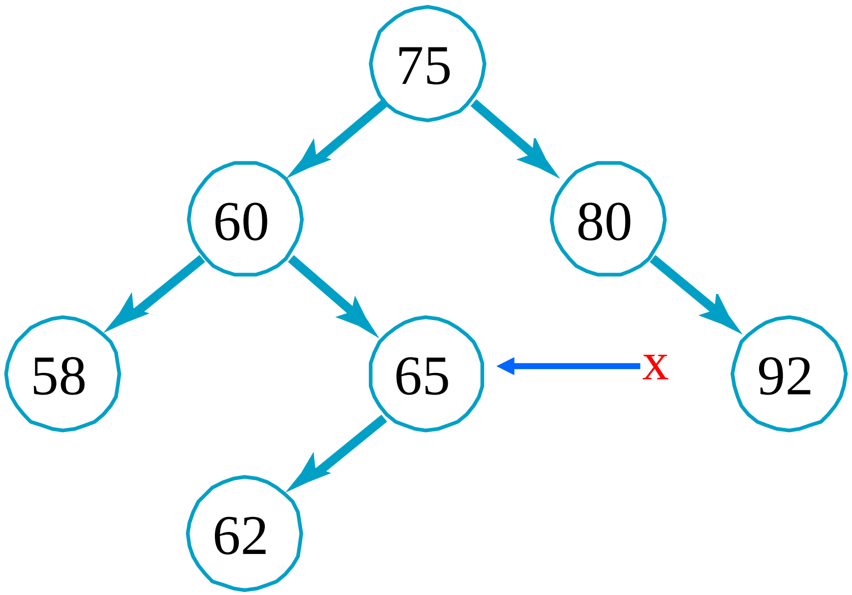


Implementing a BST

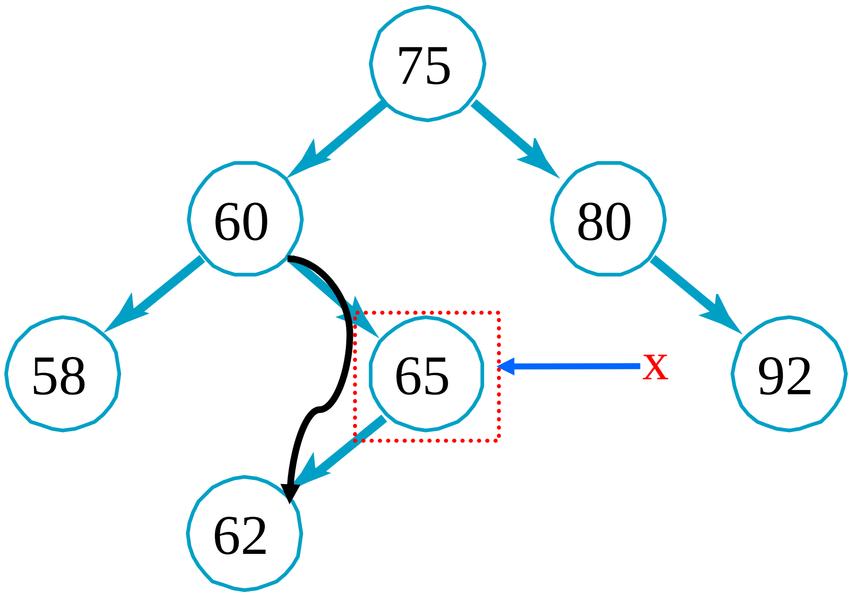
<u>CASE II</u>:

- x is has one child

Set the appropriate pointer in x's parent to point to this child



Implementing a BST



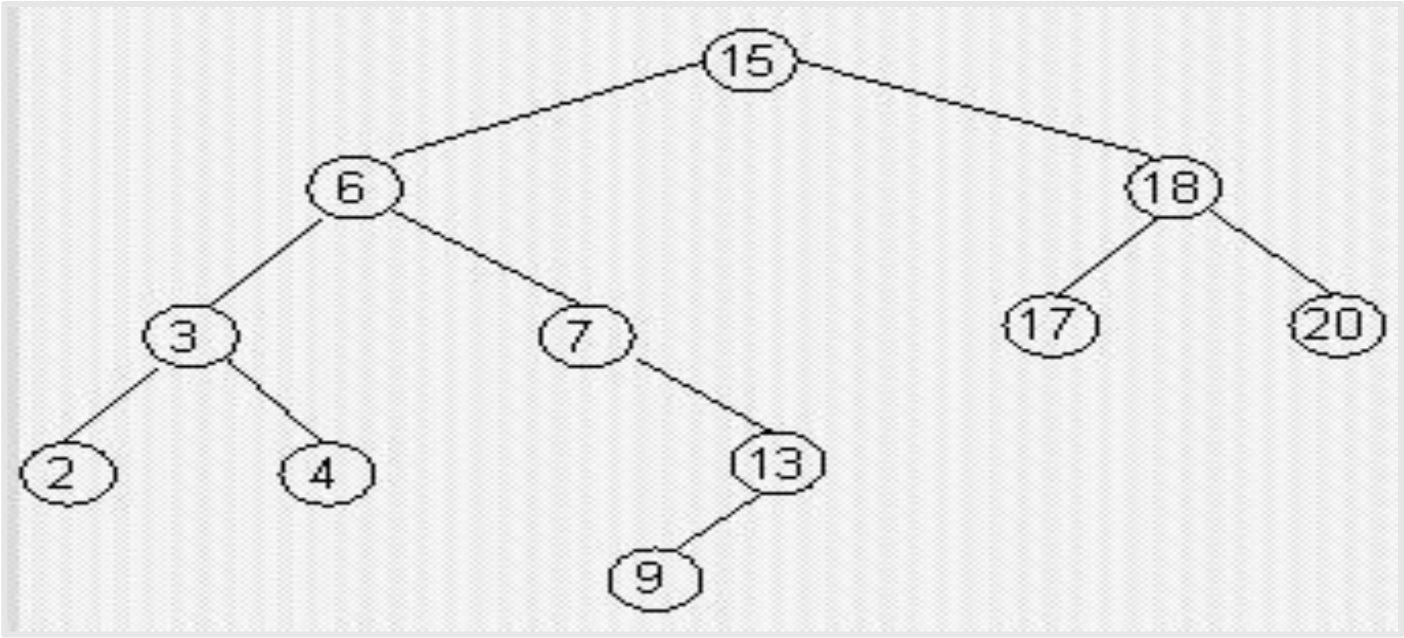
Deleting from a BST

- If a node z has no children, we modify its parent to replace z with NULL as child
- If node z has one child, we splice out z by making a new link between its child and its parent
- If node z has two children, we splice out z's **successor** y, which has no left child and replace the contents of z with the contents of y

Successor

- The successor of a node *x*, is node *y*, that has the smallest key greater than that of *x*
  - If *x* has a **right subtree**, then *successor(x)* is the left most element in that sub tree
  - If *x* has **no right sub tree**, then *successor(x)* is the lowest **ancestor** of *x* (above x on the path to the root) that has *x* in its left sub tree

Examples - Successor

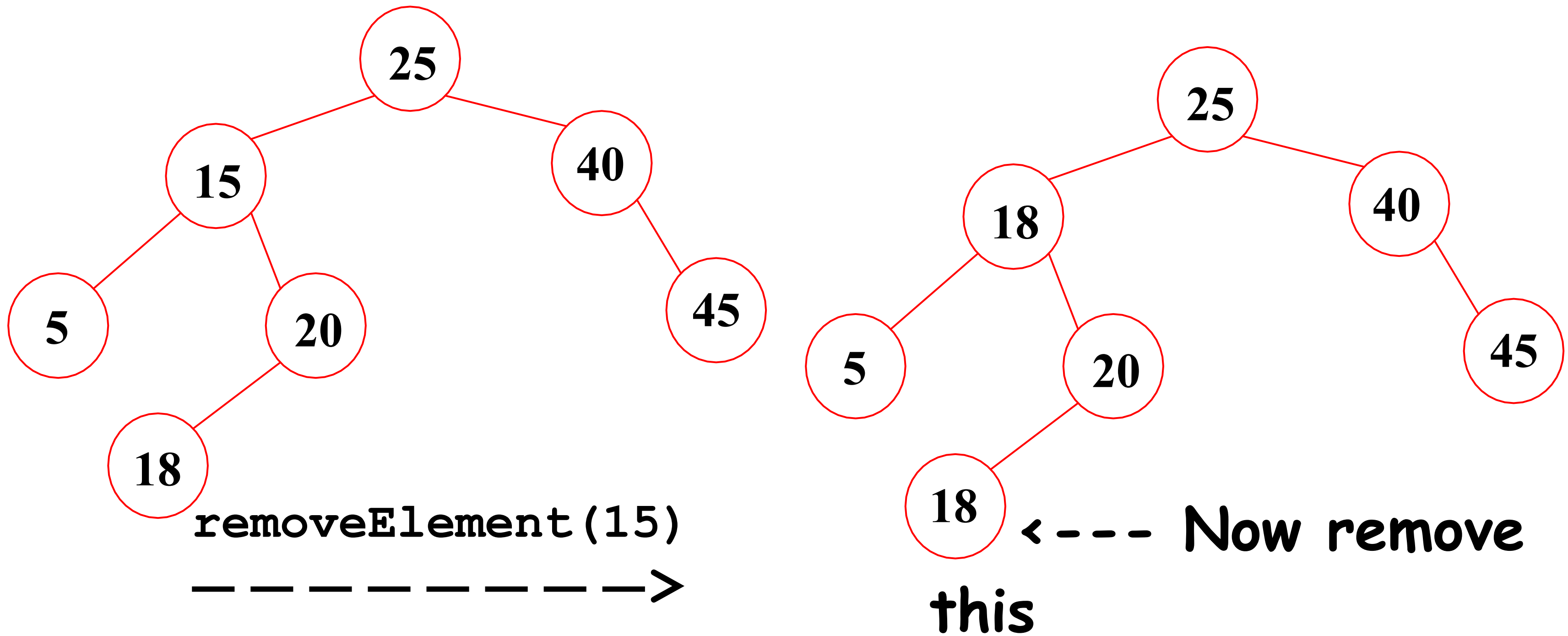


- The successor of node with key 15 is node with key 17
- The successor of node with key 7 is node with key 9
- The successor of node with key 13 is node with key 15

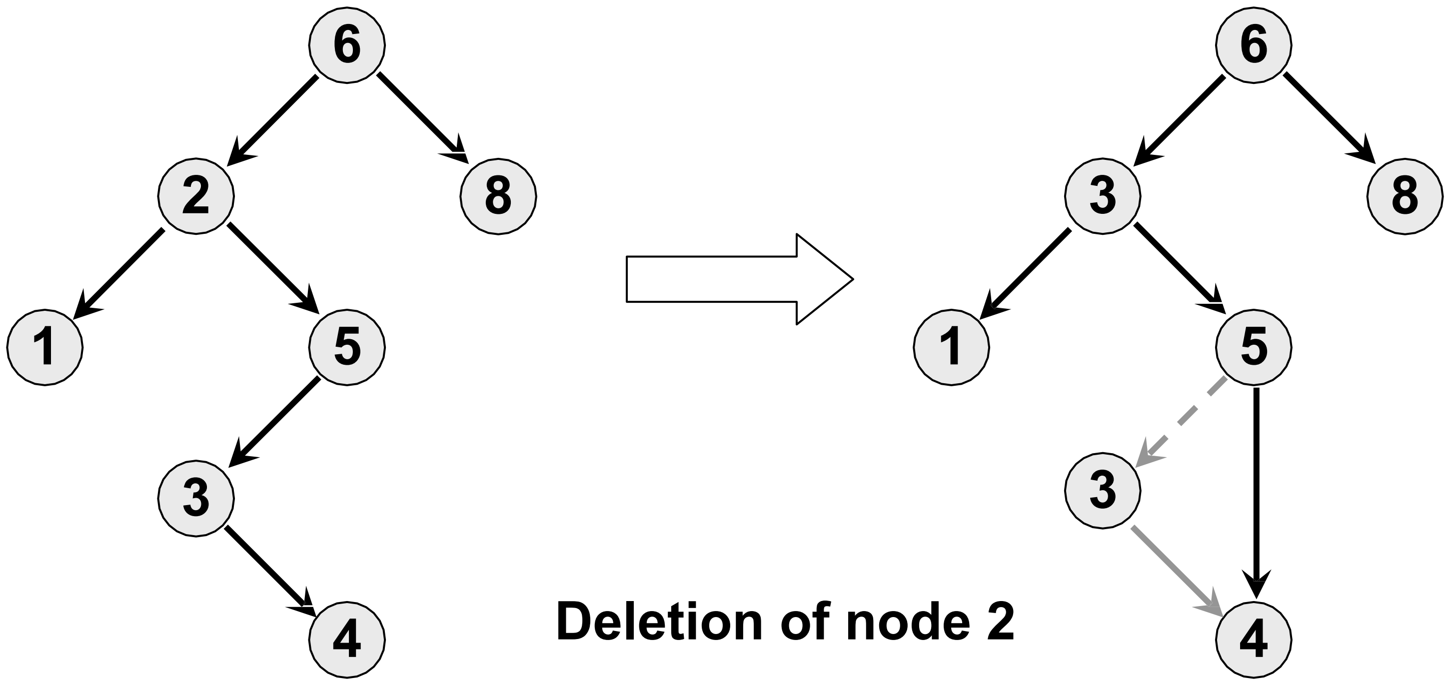
`remove` in a binary search tree

Case 3: `x` occurs at a node with two children

First replace `x` with smallest value in right subtree of `x`. which is the successor of x. This value occurs at a node with no left child. So we can delete this node using one of the two previous cases



Deleting a node with two children



Deletion of node 2

Minimum Function

NUST SEECS

- The minimum element of BST is the left most node of left sub tree
- Therefore the minimum can always be found by tracking the left child pointers until an empty sub tree is reached
- If there is no left sub tree then minimum key is at root (i.e. key[x])

```
Tree-Minimum(tree)
{
        while( tree->left != NULL)
                tree = tree->left;
        return tree
}
```

Maximum Function

- The maximum element of BST is the right most node of right sub tree
- Therefore the maximum can always be found by tracking the right child pointers until an empty sub tree is reached

```
Tree-Maximum( tree)
{
   while( tree->right!= NULL) tree =
        tree->right;
return tree

}
```

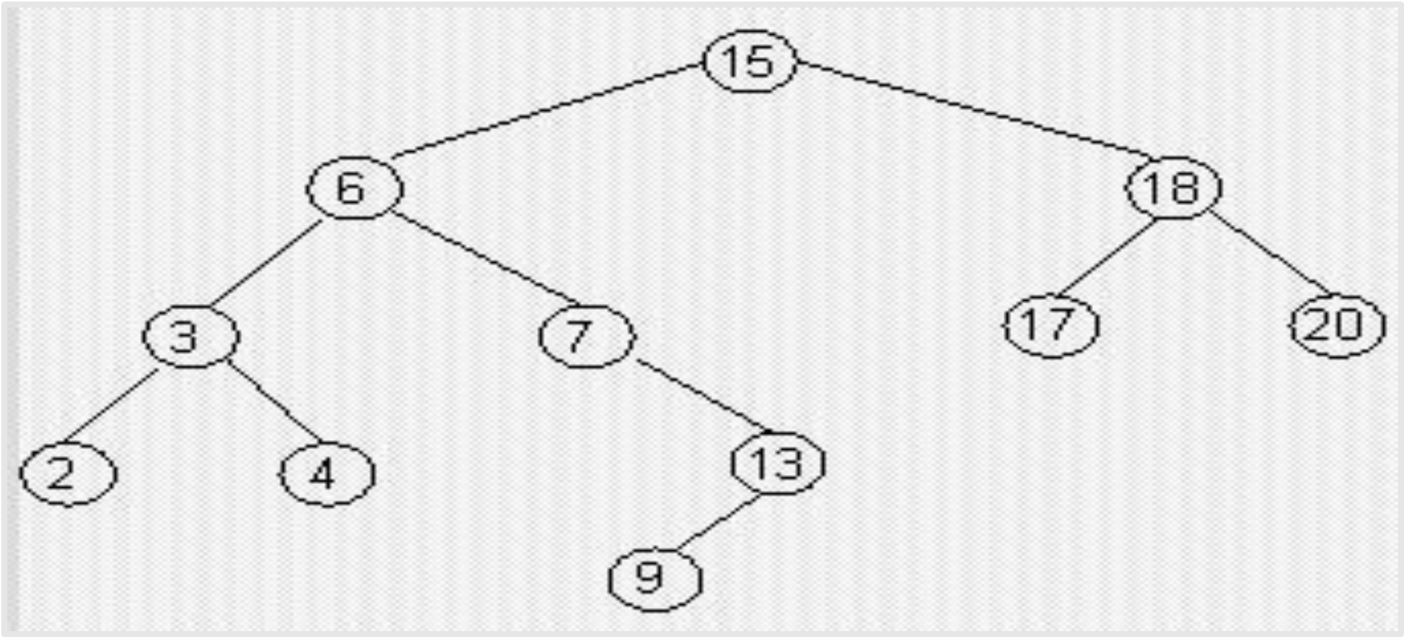
Predecessor

- The predecessor is the node that has the largest key smaller than that of *x*

  - If *x* has a left sub tree, then the predecessor must be the right most element of the left sub tree

  - If *x* has **no left subtree**, then predecessor (x) is the lowest **ancestor** of *x* (above *x* on the path to the root) that has x in its right subtree

Examples - Predecessor Function



- The predecessor of node with key 6 is node with key 4
- The predecessor of node with key 15 is node with key 13
- The predecessor of node with key 17 is node with key 15

ALGORITHM 7-3 Search BST

```
Algorithm searchBST (root, targetKey)
Search a binary search tree for a given value.
  Pre    root is the root to a binary tree or subtree
         targetKey is the key value requested
  Return the node address if the value is found
         null if the node is not in the tree
1 if (empty tree)
     Not found
  1  return null
2 end if
3 if (targetKey < root)
  1  return searchBST (left subtree, targetKey)
4 else if (targetKey > root)
  1  return searchBST (right subtree, targetKey)
5 else
     Found target key
  1  return root
6 end if
end searchBST
```

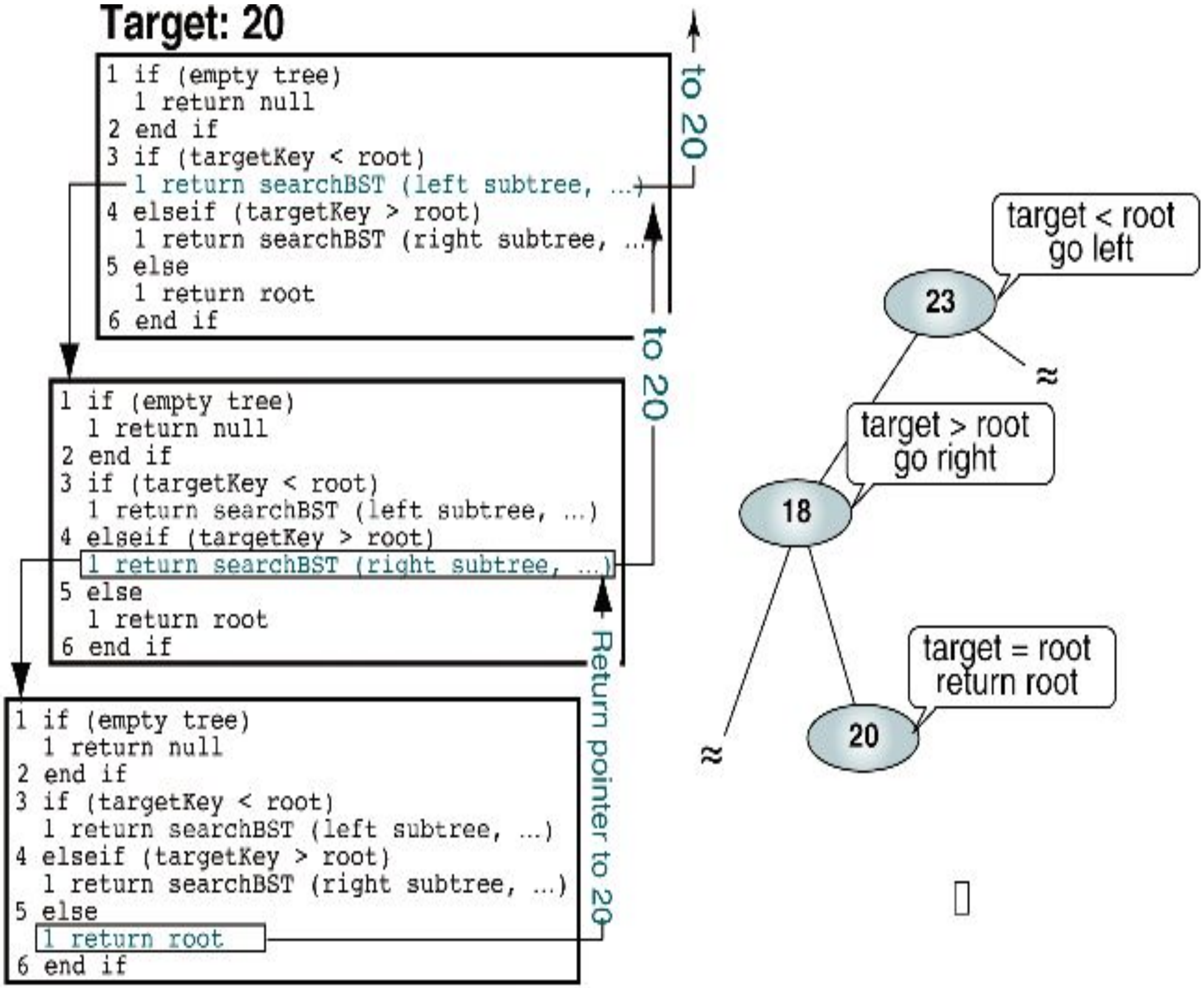


FIGURE 7-7 Searching a BST