Disjoint Sets Data Structures

Sets Repersention:

The term set is used to refer to any collection of objects, which are called members or elements of the set. A set is called finite if it contains n elements, for some constant $n \geq 0$, and infinite otherwise. Examples of infinite sets include the set of natural numbers {1; 2; …} and the sets of integers, rationales and reals.

1- Disjoint Sets A disjoint set data structure is an efficient way of keeping track of such a partition. There are two operations on a disjoint set:

1. union - merge two sets of the partition into one, changing the partition structure
2. Find - determine which set of the partition contains a given element e, returning a canonical element of that set.

A data structure that is both simple and leads to efficient implementation of these two operations is to represent each set as a rooted tree with data elements stored in its nodes. Each element x other than the root has a pointer to its parent p(x) in the tree. The root has a null pointer, and it serves as the name or set representative of the set.

Fig 1(a) shows four trees corresponding to the four sets *{1; 7; 10; 11}*, *{2; 3; 5; 6}*, *{4; 8}* and *{9}*.
Fig 2(b) shows their array representation. Clearly, since the elements are consecutive integers, the array representation is preferable. However, in developing the algorithms for the *union* and *find o*perations, we will assume the more general representation, that is, the tree representation

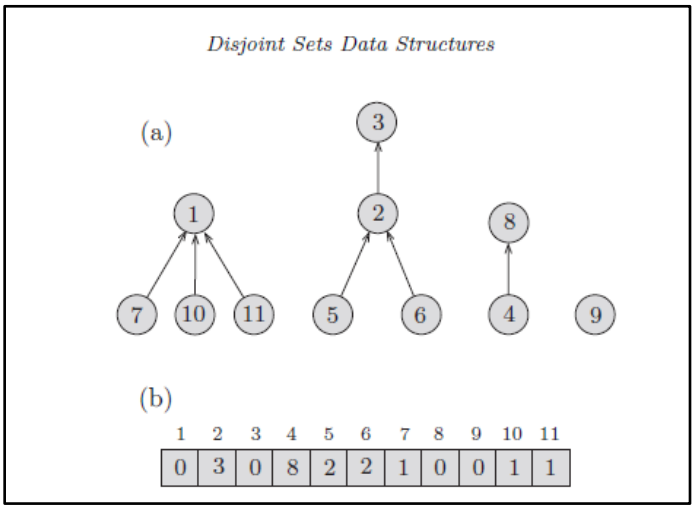


Fig.1 An example of the representation of disjoint sets. (a) Tree representation.
(b) Array representation when *S {*1,2,... , *n}*.

1-2 The union Operation: the operation union(x; y), let the link of root(i) point to root(j), i.e., if root(i) is u and root(j) is v, then let v be the parent of u.

Algorithm union:
Input: Two elements i and j
Output: The union of the two trees containing x and y. The original trees are destroyed.
{
p (v)← u
}

Example 1: Let S = {1, 2, …, 5} and consider applying the following sequence of unions: union(1,3), union(2,5), union(1,2)

Solution:

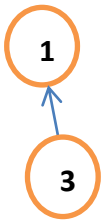
- Initially parent array conations zero

0	0	0	0	0
1	2	3	4	5

- After performing Union(1,3) operation

Parent (3) ← 1

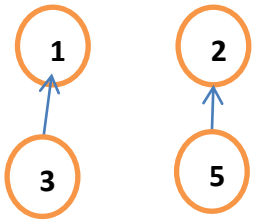
0	0	1	0	0
1	2	3	4	5



- After performing Union(2,5) operation

Parent (5) ← 2

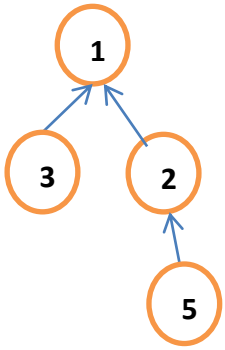
0	0	1	0	2
1	2	3	4	5



- After performing Union(1,2) operation

Parent (2) ← 1

0	1	1	0	2
1	2	3	4	5



1.3Find Operation: Find (i), Implies that it finds the root node ithnode, in other words, it returns the name of the set i

Algorithm Find (i)

{

Integer i,j;

j ⟵ i

While parent (j)>0 do

j ⟵parent (j);

Repeat;

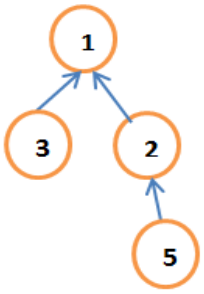
Return (j)

}

Example 2:-

Find (5),i=5, j⟵ i,j=5

0	1	1	0	2
1	2	3	4	5



While parent (5)>0 do condition true

j ⟵parent (j); j=2

While parent (2)>0 do condition true

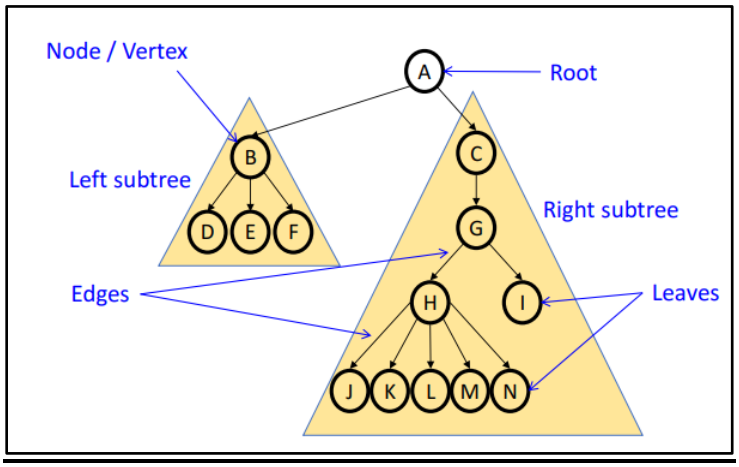
j ⟵parent (j); j=1

While parent (i)>0 do condition false

j ⟵parent (j);

Return (j), so 1 is root node of node 5

Tree terminology



Binary Search Trees (BST) Data Structure

A binary search tree is a data structure composed of nodes. Each node has a key, which determines the node's position in the tree. (The node may also have a "value" field, where additional data is stored.) The top of the tree is the "root," and the nodes contain pointers to other nodes. Specifically, each node has a left child, a right child, and a parent (some of which may be NIL) In Figure 1(b), the left child of 7 is 6 and the left child of 5 is NIL. Also, the parent of 5 is 2, and since 2 is the root of the tree, the parent of 2 is NIL

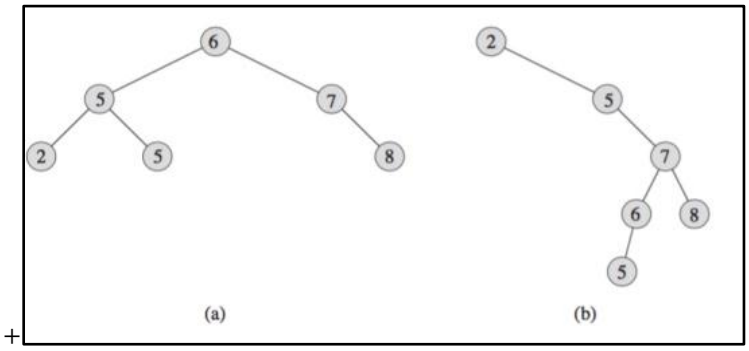


Figure 1**:** Binary Search Trees

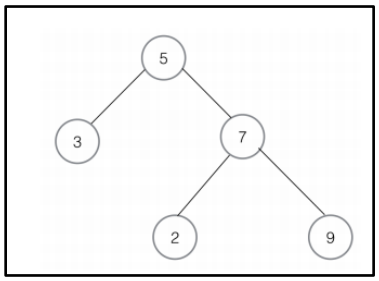
- Hierarchical data structure with a single reference to root node
- Each node has at most two child nodes (a left and a right child)
- Nodes are organized by the Binary Search property:

  • Every node is ordered by some key data field(s)

  • For every node in the tree, its key is greater than its left child's key and less than its right child's key

  All nodes in a binary search tree must satisfy the binary search tree property:

Binary-search-tree property: Let **x** be a node in a binary search tree. If **y** is a node in the left subtree of **x**, then **y.key ≤ x.key**. If **y** is a node in the right subtree of **x**, then **y.key ≥ x.key.** This means, for example, that the following tree is not a binary search tree:



Even though $2 \leq 7 \leq 9$ and $3 \leq 5 \leq 7$, this tree does not satisfy the binary search tree property, because 2 is in the right subtree of 5, despite being smaller than 5

1.1Operations of Binary search trees

Binary search trees support several operations, including: **Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete.**

1.1.1 Search Operation:

The Search routine searches for a node in the tree with a given key. It compares the node to the root.

Rules-

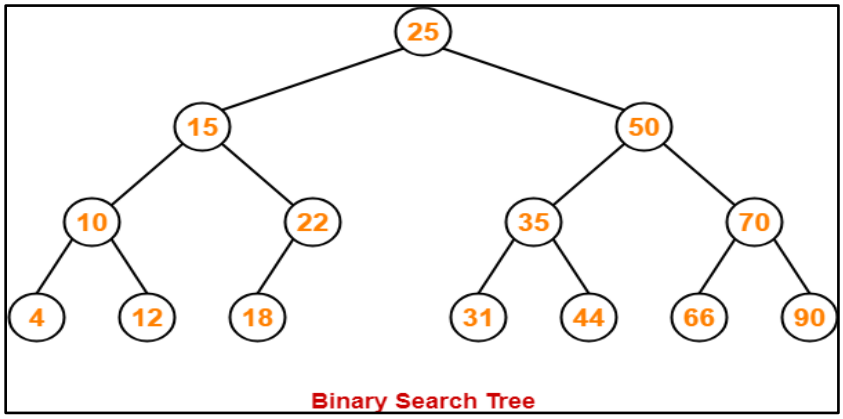
- If the keys are the same, it just returns that node.
- If the node's key is smaller, then the result (if it exists) will be in the left subtree, by the binary search tree property.
- If the node's key is larger, then the result (if it exists) will be in the right subtree.

Eventually, we either find the node, or we reach NIL, which tells us that the node can't be found in the tree.

Algorithm TREE-SEARCH(x,k)

```
if x==NILL or k===x.key
    return x
if  k<x.key
   return TREE-SEARCH(x.left,k)
else return Tree-Search(x.right,k)
```

Example- Consider key = 45 has to be searched in the given BST-



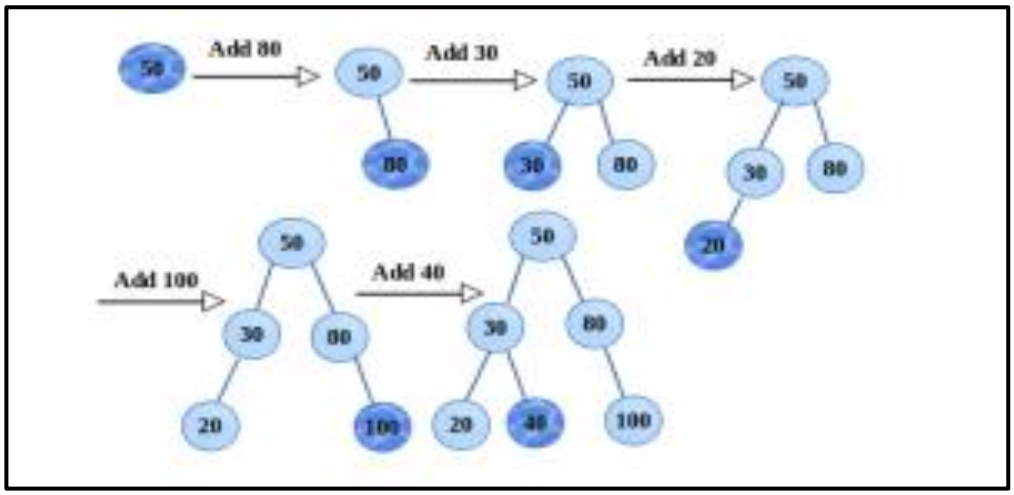
- Start our search from the root node 25.
- As 45 > 25, so we search in 25's right subtree.
- As 45 < 50, so we search in 50's left subtree.
- As 45 > 35, so we search in 35's right subtree.
- As 45 > 44, so we search in 44's right subtree but 44 has no subtrees.
- So, we conclude that 45 is not present in the above BST.

1.1.2. Insertion Operation:

To build a BST of the keys, 50, 80, 30, 20, 100, 40. It can be clearly seen below.

Rules- for inserting, first the key:

- is compared with the root,
- if smaller then goto Left subtree
- else Right subtree.
- The same step is repeated until all the keys are inserted.

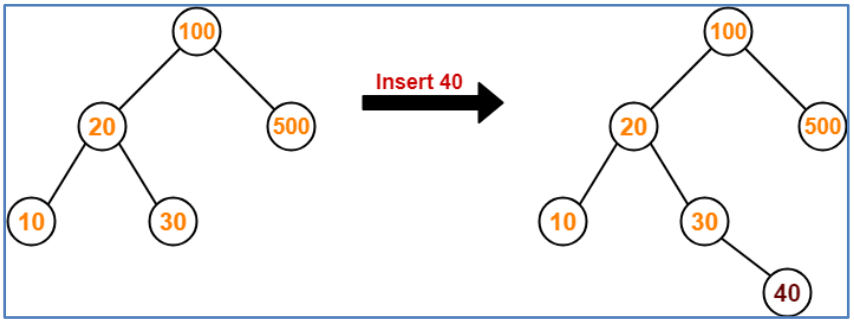


Insertion to BST

The insertion of a new key always takes place as the child of some leaf node.
For finding out the suitable leaf node,

- Search the key to be inserted from the root node till some leaf node is reached.
- Once a leaf node is reached, insert the key as child of that leaf node.

Example- Consider the following example where key = 40 is inserted in the given BST-



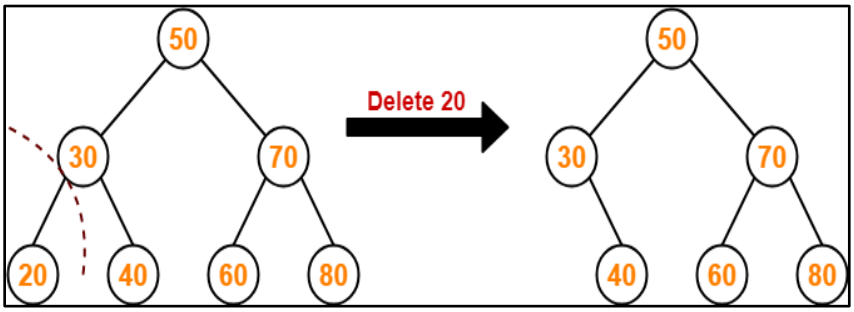
- Start searching for value 40 from the root node 100.
- As 40 < 100, so we search in 100's left subtree.
- As 40 > 20, so we search in 20's right subtree.
- As 40 > 30, so we add 40 to 30's right subtree.

1.1.3Deletion Operation- Deletion of a node is performed as follows**,**

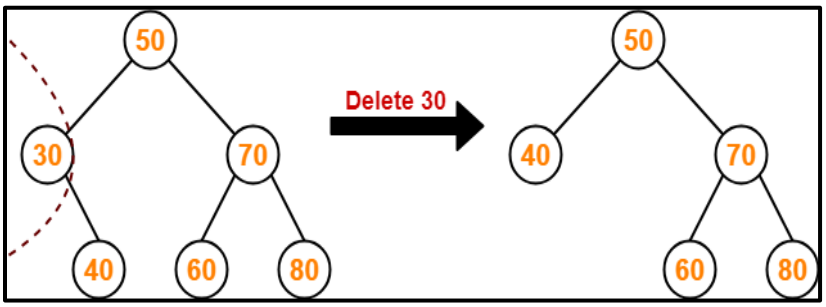
1. Deleting node has no child, therefore just delete the node(made to point NULL)
2. Deleting node has 1 child, swap the key with the child and delete the child.
3. Deleting node has 2 children, in this case swap the key with *inorder successor* of the deleting node. It should be noted, inorder successor will be the minimum key in the right subtree (of the deleting node).

Example-(Deleting node has no child)

Consider the following example where node with value = 20 is deleted from the BST-



Example (Deleting node has 1 child) Consider the following example where node with value = 30 is deleted from the BST-

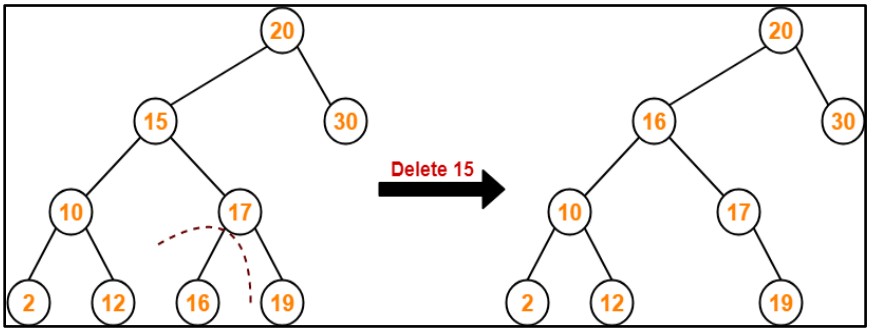


Example (Deletion Of A Node Having Two Children)- A node with two children may be deleted from the BST in the following two ways-

Method-01:

- Visit to the right subtree of the deleting node.
- Pluck the least value element called as inorder successor.
- Replace the deleting element with its inorder successor.

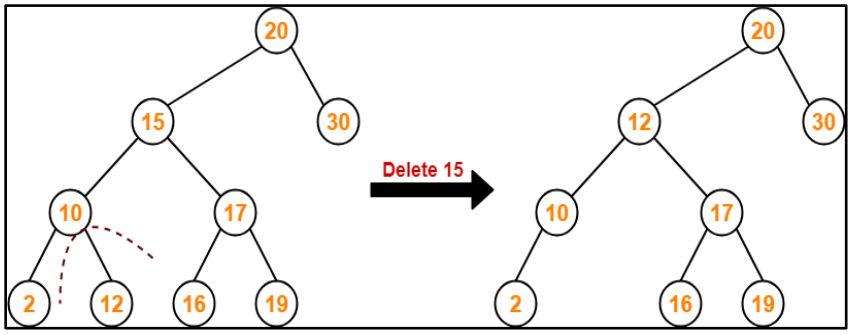
Consider the following example where node with value = 15 is deleted from the BST-



Method-02:

- Visit to the left subtree of the deleting node.
- Pluck the greatest value element called as inorder predecessor.
- Replace the deleting element with its inorder predecessor.

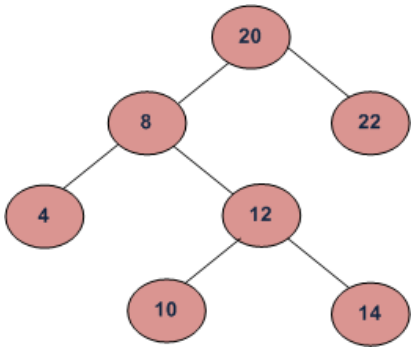
Consider the following example where node with value = 15 is deleted from the BST-



1.1.4Minimum& Maximum Operations:

1-For the node with minimum value: find the leftmost leaf node

2- For the node with maximum value: find the rightmost leaf node



For the above tree, we start with 20, then we move left 8, we keep on moving to left until we see NULL. Since left of 4 is NULL, 4 is the node with minimum value.

For the above tree, we start with 20, then we move right 22, we keep on moving to right until we see NULL. Since right of 22 is NULL, 22 is the node with maximum value.