Binary Search Tree

★. log operations when balanced (If not balance all become O(n))

Why?

	Array (unsorted)	Array (Sorted)	LL	BST	Hash Table.
Search	O(n)	O(logn)	O(n)	O(logn)	O(1)
Insert	O(1)	O(n)	O(1) & (O(n) for sorted)	O(logn)	O(1)
Delete	O(n)	O(n)	O(n)	O(logn)	O(1)
Find closest.	O(n)	O(logn)	O(n)	O(logn)	O(n)
Sorted traversal.	O(nlogn)	O(n)	O(nlogn)	O(n)	O(nlogn)

∴ If only search Insert and delete then hashtable is best but additional one of the last two req, then BST can be used.

Intro

(1) ⇒ Tree + Binary Search. ⇒ BST.

⇒ It is implemented using linked list.

⇒ Implemented directly in Java → TreeSet & TreeMap.

Operations in BST

1) Search operation in BST

NOTE:
⇒ Class considered → class Node{
int key;
Node left, right;
Node int(x){ key = x;}
}

Search operation. (O(H)).

⇒ boolean Search (Node root, int x)
{
if(root == null)
return false;
else if (root.key == x)
return True;
else if (root.key < x)
return (Search (root.right, x));
else return search (root.left, x);
}

(iterative).

while(root != null)
{
if (root.key == x)
true
else if (root.key > x)
root = root.left
else. . . .
}

2) Insert operation in BST

⇒ Very similar to search operation
⇒ If key is there do nothing.
⇒ If not present, then we end up with a leafnode.

Special case ⇒ (root == null) → root = x;
↳ if (root == null) { return new Node(x);}

```
Node insert (Node root, int x)
{
    if (root == null)
        return new Node(x);

    if (root.key > x)
        root.left = insert(root.left, x);
    else if (root.key < x)
        root.right = insert(root.right, x);

    return root;
}
```

root, something imp as it connects root & our new node.

Insert operation in BST (Iterative).

```
Node insert (Node root, int x)
{
    Node temp = new Node(x);
    Node parent = null, curr = root;

    while (curr != null)
    {
        parent = curr;
        if (curr.key > x)
            curr = curr.left;
        else if (curr.key < x)
            curr = curr.right;
        else
            return root;
    }

    if (parent == null)
        return temp;
    if (parent.key > x)
        parent.left = temp;
    else { parent.right = temp; }
}
```

for forming connection

④ Delete operation in BST

- Search and delete.
- if not present return false.
  else delete and return true.

- Cases → leaf node
  internal node. → one child.
  → 2 childsen.

See code in Binary Tree section, same code.

---

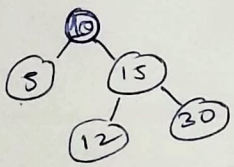
⑨ Note:

Inorder traversal of BST is sorted.

---

⑤ Floor & Ceil Operations in BST.

I/P: x = 14
Root of tree.



O/P: 12 (closet lower value).
if not smaller then given value return null.

- We can easily see that smaller will lie in left subtree.

```
Node floor (root, x)
{   Node res = null;
    while (root != null)
    {  if (root.key == x)
          return root;
       else if (root.key > x)
          root = root.left;
       else {
          res = root;
          root = root.right;
       }
    }
    return res;
}
```

- * Similarly you can program for ceil too.

(6) BST (Self Balancing)

* Idea in self balancing BST is to keep height in check so all operations stay as O(logn).

→ If you knows keys in adv, can simply sort and add values and it will be a balanced tree.

→ After that we can do rotations to keep it balanced when nodes added. (O(1) operation)

(7) AVL & Red Black Trees } All discussed in Additional nodes.

Applications of BST.

1) Maintain sorted stream of data.

2) Implementing doubly ended priority Queue.

3) To solve problems like :-

a) Largest Smaller/Greater in a Stream.

b) Floor/Ceiling in a stream.

(key) (key value)

TreeSet & TreeMap in Java

```
import java.util.*

public static void main(String[] args)

{ TreeSet<String> s = new TreeSet<String>();

  s.add("gfg");
  s.add("courses");
  s.add("ide");

  System.out.println(s);

  System.out.println(s.contains("ide"));

  Iterator<String> i = s.iterator();
while (i.hasNext()) { System.out.println(i.next());

}
```

[also has lower, higher, floor, remove, ceiling].
size, isEmpty - O(1)

O/P: [courses gfg ide] } will be in sorted order.

true

courses
gfg
courses.

TreeMap.

```
public static void main (String[] args)
{
    TreeMap<Integer, String> t = new TreeMap<Integer, String>();
    t.put(10, "geeks");
    t.put(15, "ide");
    t.put(5, "courses");

    System.out.println(t);

    for (Map.Entry<Integer, String> e : t.entrySet())
        System.out.println(e.getKey() + " " +
                           e.getValue());
}
```

higherKey
lowerKey
floorKey
ceilingKey

O/P:

{5=courses, 10=geeks, 15=ide}

5 courses
10 geeks
15 ide

* Sorts acc to keys.

Extra.

To get a value acc to key

→ t.get(key);

(t.higherEntry(10).getValue()); → gets closest higher value to key & print its value