Binary Search Trees

Search trees are the data structures that support many dynamic-set operations, including SEARCH, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE. Thus a search tree can be used both as a dictionary and as a priority queue.

Basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with n nodes, such operations run in Θ (log n) worst-case time. If the tree linear chain of nodes, however, the same operations of a randomly built binary search tree is only O (log n), so that basic dynamic-set operations on such a tree take Θ (log n) time on average.

In practice, we can't always guarantee that binary search trees are built randomly, but there are variations of binary search trees whose worst-case performance on basic operations can be guaranteed good. After presenting the basic properties of binary search trees, the following sections show how to walk on a binary search tree to print its values in a sorted order, how to search for a value in a binary search tree, how to find the minimum or maximum element, how to find the predecessor of an element, and how to insert into or delete from a binary search tree.

BASIC DEFINITIONS:

Binary tree: A binary tree T is defined as a finite set of elements, called nodes, such that:
1. T is empty or
2. T contains a distinguished node R, called root, and the remaining nodes form two disjoint binary trees T1 and T2.

Note: if T contains a root R, then the two trees T1 and T2 are called, respectively, the left and right sub trees of R.

Successor:

If left sub tree of a tree is nonempty, then the root of lest sub tree is called the "left successor" of R.

If right sub tree of a tree is nonempty, then the root of right sub tree is called the "right successor" of R.

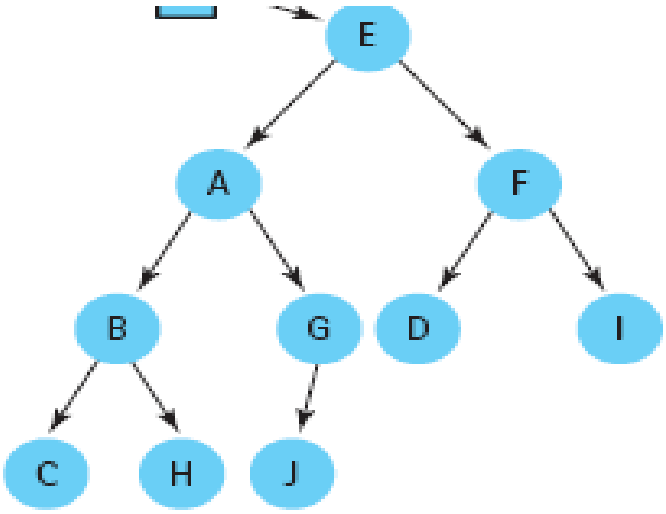
Predecessor:

Suppose N is a node in T with left successor s1 and right successor s2 .Then N is called the parent of s1 and s2.Every node N in a binary tree T, except the root, has a unique parent, called the predecessor of N.

Binary tree with root E

1. A and F are the left and right successor of root E.

2. A is predecessor of B and G



Reference: Introduction to Algorithmics, Brassard and Bratley

Level:

The distance of a node from the root is called the level. The root is level 0.

Height:

The maximum level of a tree is called the height of the tree.

Leaf:

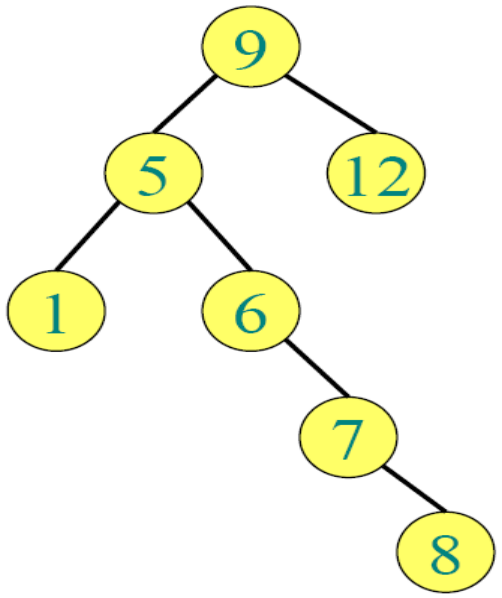
A node is called a leaf if it has no left and right sub tree (children).

In the above example C, H, J, D and I are the leaf nodes.
Height of the tree is 3.

Binary search tree:

A binary tree is called binary search tree if each node N of T has the property: The value at N is greater than every value in the left sub tree of N and less than or equal to every value in the right sub tree of N.

Example: 9, 5,1,12,6,7,8



Reference: Introduction to Algorithmics, Brassard and Bratley

CONSTRUCTION OF BINARY SEARCH TREE

The following algorithm describes how to construct a binary search tree from a given linear array.

Algorithm:

Sequential array to BST

1. Maketree(x)
   Info (1) = x
   Status (1) = 1
   For i is 2 to n
     Status (i) =0

2. Set left (insert x as left child of p)

```
q=2p
If (q>=N) “overflow”
Else
    If (status (q) =1)
       Then “invalid insertion”
    Else
       Info (q) =x
       Status (q) =1
```

3. Set right (insert x as right child of p)

```
q=2p+1
If (q>=N) “overflow”
Else
    If (status (q) =1)
       Then “invalid insertion”
    Else
       Info (q) =x
       Status (q) =1
```

4. Make tree (distinct)

```
    Make tree(x)
    While (input=1)
       P=1, q=1
       While (q≤ n & Status (q) =1 & no ≠ Info (p))
 {
    p = q
     If (no < Info (p)) then q = 2p
     Else q = 2p+1
   }
 If Info (p) = no
    “Number is a duplicate”
 Else
     If no< Info (p) then set left (p, no)
     Else Set right (p.no)
 }
}
```

Construction of binary search tree using linked list

```
Make tree(x)
  p=get node ()
  Info (p) =x
  Left (p) =null
  Right (p) =null

Set left (insert x as left child of p)
  If (p=null) "overflow"
  Else
      If (p→left! =null)
          Then "invalid insertion"
      Else
         p→left=make tree(x)
Set right (insert x as right child of p)
  If (p=null) "overflow"
  Else
      If (p→right! =null)
          Then "invalid insertion"
      Else
         p→right=make tree(x)
```

Note: complexity of construction of binary search tree is O(nlogn)

TRAVERSING BINARY SEARCH TREES

There are three standard ways of traversing a binary search tree T with root R. These three algorithms are called preorder, in order, post order.

Algorithms:

```
Preorder (p)
   If p! =null
      Visit info (p)
       Preorder (left (p))
       Preorder (right (p))

   In order (p)
    If p! =null
```

```
In order (left (p))
Visit info (p)
In order (right (p))

Post order (p)
If p! =null
  Post order (left (p))
  Post order (right (p))
  Visit info (p)
```

Note: complexity for traversing the binary search tree is O (n)
Construction of binary search tree in c language

SEARCHING

A common operation performed on a binary search tree is searching for a key stored in the tree. We use the following procedure to search for a node with a given key in a binary search tree. Given pointer to the root of the tree and a key k, tree search returns a pointer to a node with key k if exists; other wise it returns nil. The procedure begins its search at the root and traces a path downward in the tree. For each x it encounters, it compares the key k with key[x], if two are equal, the search terminates. If the k is smaller then key[x], the search continues in the left sub tree of x, since the binary-search-tree property implies that k could not be stored in right sub tree. Symmetrically, if k is larger than key[k], the search continues in the right sub tree.

Algorithm:

```
TREE-SEARCH (X, K)
{
  If x= nil or k = key[x] then return x
  If key < key [x]
     Then return TREE-SEARCH (left[x], k)
      Else return TREE-SEARCH (right[x], k)
}
```

ITERATIVE-TREE-SEARCH (X, K)

```
While x ± nil and k ± key[x]
      Do if k < key[x]
                Then x = left[x]
                 Else x = right[x]
Return x
```

Note: complexity for searching a element is O(h), where h is the height of the tree.

MAXIMUM AND MINIMUM

An element in a binary tree whose key is a minimum can always be found by the following left child pointers from the root until a NIL is encountered. The following procedure returns a pointer to the minimum element in the sub tree rooted at given node x. The binary search tree property guarantees that TREE-MINIMUM is correct. If a node has no left sub tree, then since every key in the right sub tree of x is at least as large as key[x], the minimum key in the sub tree rooted at x is key[x]. if a node x has a left sub tree, then since no key in the right sub tree is smaller than key[x] and every key in the left sub tree is not larger than key[x],the minimum key in the sub tree rooted at x can be found in the rooted at left[x]. Similar reasoning applies to the largest element also.

TREE-MINIMUM (X)

```
While left[x] ≠ NIL
            Do x = left[x]
Return x
```

TREE-MAXIMUM

```
 While right[x] ± NIL
        Do x = right[x]
  Return x
```

Note: complexity for finding minimum and maximum is O(h), where h is height of the tree

INSERTING AN ELEMENT

The insertion and deletion cause the dynamic set represented by a binary search tree to change. The data structure must be modified to reflect this change, but in such a way that binary-search-tree continues to hold.

To insert a new value v into a binary search tree T, we use the procedure TREE-INSERT. The procedure is passed a node z for which key[z] =v, left[z] =nil. And right[z] =nil.

```
TREE-INSERT (T, z)
    y←nil
    x←root [T]
    while x!=nil
      do y←x
          if key[z] <key[x]
          then x ←left[x]
          else  x ←right[x]
     p[z] ←y
     if y=nil
        then root[T] ←z
        else  if  key[z]<key[y]
                then left[y] ←z
                else  right ←z
```

Note: complexity for inserting an element is O (h), where h is height of the tree

DELETION OF AN ELEMENT

The procedure for deleting a given node z from a binary search tree has 3 cases

1. If z has no children, we modify its parent p[z] to replace z with nil as its child.
2. If the node has only a single child, we slice out by making a new link between its child and its parent.
3. If the node has two children, we slice out z's successor y, which has no left child and replace z's

```
TREE-DELETE (T, z)
    If left[z] =null or right[z] =null
      then y ← z
       else y ← TREE-SUCCESSOR (z)
    if left[y]!=null
```

```
    then x ←left[y]
    else x ← right[y]
 if x !=null
    then p[x] ←p[y]
 if p[y]=null
   then root [T] ← x
    else if y=left [p[y]]
          then left [p[y]] ←x
          else right [p[y]] ←x
if  y!=z
    then key[z] ←key[y]
          copy y's data into z
return y
```

Note: complexity for deleting a given node is O(h), where h is height of the tree