Chapter 12 Binary Search Trees

Introduction to Algorithms

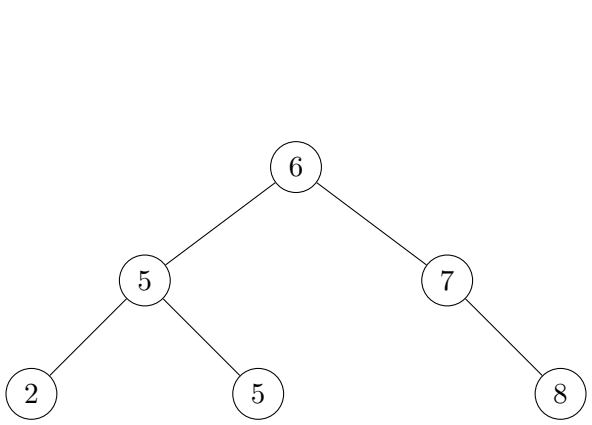
Mina Gabriel

12 Binary Search Trees

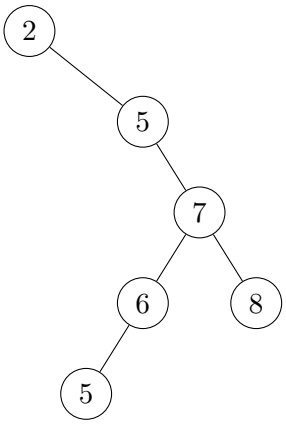
The search tree data structure supports many dynamic-set operations, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE. Thus, we can use a search tree both as a dictionary and as a priority queue.

12.1 What is a binary search tree?

A binary search tree is organized, as the name suggests, in a binary tree, We can represent such a tree by a linked data structure, each node contains attributes left, right, and p that point to the nodes corresponding to its left child, its right child, and its parent, respectively.



(a) BST



(b) Less efficient BST

The keys in a binary search tree are always stored in such a way as to satisfy the **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 $y.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$.

Thus, in Figure (a), the key of the root is 6, the keys 2, 5, and 5 in its left subtree are no larger than 6, and the keys 7 and 8 in its right subtree are no smaller than 6. The same property holds for every node in the tree. For example, the key 5 in the root's left child is no smaller than the key 2 in that node's left subtree and no larger than the key 5 in the right subtree.

The binary-search-tree property allows us to print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an inorder tree walk.

[1] PrintInOrdernode node is not null `// Traverse the left subtree` PrintInOrdernode.left

`// Print the current node's value` printnode.value

`// Traverse the right subtree` PrintInOrdernode.right

`// Assuming the root of the BST is called 'root'` PrintInOrderroot

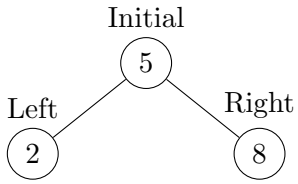
Listing 1: inorder usage

```
A = inorder(r,[])
```

As an example, the inorder tree walk return a list of roots object in each of the two binary search trees from last shown tree in the order 2, 5, 5, 6, 7, 8.

It takes $\Theta(n)$ time to walk an n-node binary search tree, since after the initial call, the procedure calls itself recursively exactly twice for each node in the tree—once for its left child and once for its right child, Take for example the following tree where $n = 3$, the previous method will be called three times:

- Initial Call
- Left Recursive Call
- Right Recursive Call



Theorem

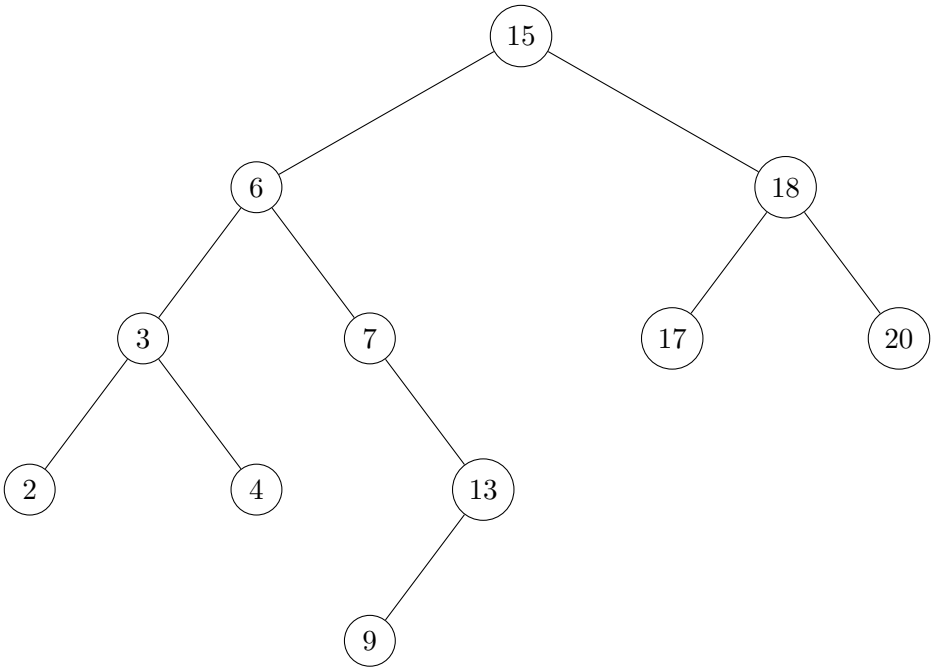
If x is the root of an n-node subtree, then the call INORDER(x) takes $\Theta(n)$ time

12.2 Querying a binary tree

We often need to search for a key stored in a binary search tree, with other queries as stated above.

12.2.1 Searching

We use the following procedure to search for a node with a given key in a binary search tree.



To search for the key 13 in the tree, we follow the path $15 \to 6 \to 7 \to 13$ from the root. The minimum key in the tree is 2, which is found by following left pointers from the root. The maximum key 20 is found by following right pointers from the root.

Listing 2: TREE-SEARCH(x)

```
def search(root, k):
  if root == None:
    return None
  elif root.val == k:
    return root
  if k < root.val:
    return search(root.left, k)
  else:
    return search(root.right, k)
```

12.2.2 Minimum and Maximum

We can always find an element in a binary search tree whose key is a minimum by following left child pointers from the root until we encounter a None.

Listing 3: TREE-MINIMUM(x)

```
def tree_min(root):
  while root.left != None:
    root = root.left
  return root.val
```

Listing 4: TREE-MAXIMUM(x)

```
def tree_max(root):
  while root.right != None:
    root = root.right
  return root.val
```

12.2.3 Successor and predecessor

Given a node in a binary search tree, sometimes we need to find its successor in the sorted order determined by an inorder tree walk. If all keys are distinct, the successor of a node x is the node with the smallest key greater than x.key. The structure of a binary search tree allows us to determine the successor of a node without ever comparing keys.

Listing 5: TREE-SUCCESSOR(x)

```
def successor(root, k):
  root = search(root, k)
  if root.right != None:
    return tree_min(root.right)
  y = root.p
  while y != None and root == y.right:
    root = y
    y = root.p
```

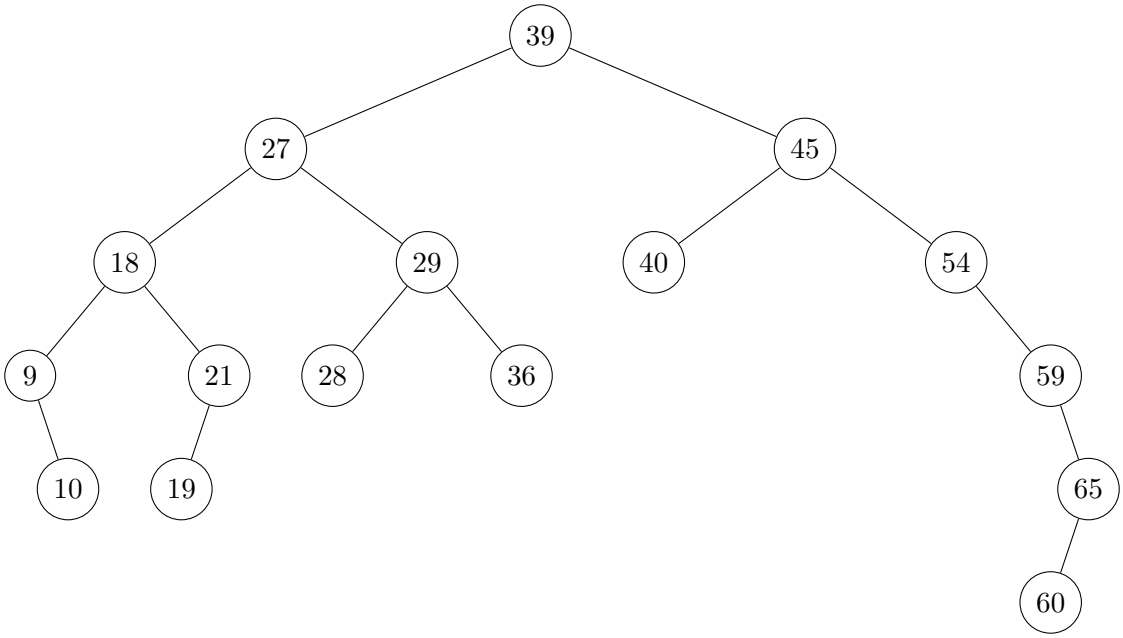
```
return y
```

The running time of TREE-SUCCESSOR on a tree of height h is $O(h)$, since we either follow a simple path up the tree or follow a simple path down the tree. The procedure TREE-PREDECESSOR, which is symmetric to TREE-SUCCESSOR, also runs in time $O(h)$.

Working Example

Consider the following tree, answer the following:

- Print all the elements of this tree in an ascending order.
- Print the element whose key is the minimum.
- Print the element whose key is the maximum.
- Print the s if key = 2 is in the tree.
- print the key successor of 29.



Listing 6: TREE-SUCCESSOR(x)

```
class Node:
    def __init__(self,key):
        self.left = None
        self.right = None
        self.val = key
```

```
        self.p = None

# A utility function to insert a new node with the given key
def insert(root,node):
    if root is None:
        root = node
    else:
        if root.val < node.val:
            if root.right is None:
                root.right = node
                node.p = root
            else:
                insert(root.right, node)
        else:
            if root.left is None:
                root.left = node
                node.p = root
            else:
                insert(root.left, node)

# A utility function to do inorder tree traversal
def inorder(root, A):
    if root:
        inorder(root.left, A)
        A.append(root)
        inorder(root.right,A)
    return A

def search(root, k):
  if root == None:
    return None
  elif root.val == k:
    return root
  if k < root.val:
    return search(root.left, k)
  else:
    return search(root.right, k)

def tree_min(root):
  while root.left != None:
    root = root.left
  return root

def tree_max(root):
  while root.right != None:
    root = root.right
  return root
```

```
def successor(root, k):
  root = search(root, k)
  if root.right != None:
    return tree_min(root.right)
  y = root.p
  while y != None and root == y.right:
    root = y
    y = root.p
  return y



r = Node(39)
#use the random list generator if needed
'''
for i in range(0, 100):
  insert(r, Node(random.randint(1,101)))
'''
nodes = [27,45,18,29,40,54,9,21,28,36,59,10,19,65,60]
for i in nodes:
  insert(r, Node(i))
A = inorder(r,[])
l = []
for i in A:
  l.append(i.val)
print l
print "Find key:", search(r,29).val
print "Successor is:", successor(r, 18).val
print "Tree Min:", tree_min(r).val
print "Tree Max:",  tree_max(r).val
```