# <2T>

# Binary Search Tree Description

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 in which each node is an object. In addition to a key and satellite data, 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. If a child or the parent is missing, the appropriate attribute contains the value NIL. The root node is the only node in the tree whose parent is NIL.

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 <= x.key. If y is a node in the right subtree of x, then y.key => x.key.

Searching

We use the following procedure to search for a node w ith a given key in a binary

search tree. Given a pointer to the root of the tree and a key k, TREE-SEARCH returns a pointer to a node with key k if one exists; otherwise, it returns NIL.

The procedure begins its search at the root and traces a simple path downward in

the tree, as shown in Figure 12.2. For each node x it encounters, it compares the key

k with x.key .If the two keys are equal, the search terminates. If k is smaller than x.key , the search continues in the left subtree of x, since the binary-search-tree property implies that k

could not be stored in the right subtree. Symmetrically, if k is larger than x.key , the search continues in the right subtree. The nodes encountered during the recursion form a simple path downward from the root of the tree, and thus the running time of TREE-SEARCH is O(h)

, where h is the height of the tree.

We can rewrite this procedure in an iterative fashion by “unrolling” the recursion into a

While loop. On most computers, the iterative version is more efficient.

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 NIL. If a node x has no left subtree, then since every key in the right subtree of x is at least as large as x.key, the minimum key in the subtree rooted at x is x.key. If node x has a left subtree, then since no key in the right subtree is smaller than x.key and every key in the left subtree is not larger than x.key, the minimum key in the subtree rooted at x resides in the subtree rooted at x.left. Similar logic follows for maximum and the right most child.

# Binary Search Tree Pseudocode

Inorder-Tree-Walk(x)

1    Inorder-Tree-Walk(x.left)

2    print x.key

3    Inorder-tree-walk(x.right)

Tree-Search(x, k)

1    if x == NIL or k == x.key

2        return x

3    if k < x.key

4        return Tree-Search(x.left, k)

5    else return Tree-Search(x.right, k)

Iterative-Tree\_Search(x, k)

1    while x != NIL and k != x.key

2        if k < x.key

3            x = x.left

4        else x = x.right

5    return x

Tree-Minimum(x)

1    while x.left != NIL

2        x = x.left

3    return x

Tree-Maximum(x)

1    while x.right != NIL

2        x = x.right

3    return x

Tree-Successor(x)

1    if x.right != NIL

2        return Tree-Minimum(x.right)

3    y = x.p

4    while y != NIL and x == y.right

5        x = y

6        y = y.p

7     return y

Tree-Insert(T, z)

1    y = NIL

2    x = T.root

3    while x != NIL

4        y = x

5        if z.key < x.key

6            x = x.left

7        else x = x.right

8    z.p = y

9    if y == NIL

10        T.root = z  //tree T was empty

11    elseif z.key < y.key

12        y.left = z

13    else y.right = z

Transplant(T, u, v)

1    if u.p == NIL

2        T.root = v

3    elseif u == u.p.left

4        u.p.left = v

5    else u.p.right = v

6    if v != NIL

7        v.p = u.p

Tree-Delete(T, z)

1    if z.left == NIL

2        Transplant(T, z, z.right)

3    elseif z.right == NIL

4        Transplant(T, z, z.left(

5    else y = Tree-minimum(z.right)

6        if y.p != z

7            Transplant(T, y, y.right)

8            y.right = z.right

9            y.right.p = y

10        Transplant(T, z, y)

11        y.left = z.left

12        y.left.p = y

# Binary Search Tree Code

class TreeNode:

    def \_\_init\_\_(self,key,val,left=None,right=None,parent=None):

        self.key = key

        self.payload = val

        self.leftChild = left

        self.rightChild = right

        self.parent = parent

    def hasLeftChild(self):

        return self.leftChild

    def hasRightChild(self):

        return self.rightChild

    def isLeftChild(self):

        return self.parent and self.parent.leftChild == self

    def isRightChild(self):

        return self.parent and self.parent.rightChild == self

    def isRoot(self):

        return not self.parent

    def isLeaf(self):

        return not (self.rightChild or self.leftChild)

    def hasAnyChildren(self):

        return self.rightChild or self.leftChild

    def hasBothChildren(self):

        return self.rightChild and self.leftChild

    def replaceNodeData(self,key,value,lc,rc):

        self.key = key

        self.payload = value

        self.leftChild = lc

        self.rightChild = rc

        if self.hasLeftChild():

            self.leftChild.parent = self

        if self.hasRightChild():

            self.rightChild.parent = self

class BinarySearchTree:

    def \_\_init\_\_(self):

        self.root = None

        self.size = 0

    def length(self):

        return self.size

    def \_\_len\_\_(self):

        return self.size

    def put(self,key,val):

        if self.root:

            self.\_put(key,val,self.root)

        else:

            self.root = TreeNode(key,val)

        self.size = self.size + 1

    def \_put(self,key,val,currentNode):

        if key < currentNode.key:

            if currentNode.hasLeftChild():

                   self.\_put(key,val,currentNode.leftChild)

            else:

                   currentNode.leftChild = TreeNode(key,val,parent=currentNode)

        else:

            if currentNode.hasRightChild():

                   self.\_put(key,val,currentNode.rightChild)

            else:

                   currentNode.rightChild = TreeNode(key,val,parent=currentNode)

    def \_\_setitem\_\_(self,k,v):

       self.put(k,v)

    def get(self,key):

       if self.root:

           res = self.\_get(key,self.root)

           if res:

                  return res.payload

           else:

                  return None

       else:

           return None

    def \_get(self,key,currentNode):

       if not currentNode:

           return None

       elif currentNode.key == key:

           return currentNode

       elif key < currentNode.key:

           return self.\_get(key,currentNode.leftChild)

       else:

           return self.\_get(key,currentNode.rightChild)

    def \_\_getitem\_\_(self,key):

       return self.get(key)

    def \_\_contains\_\_(self,key):

       if self.\_get(key,self.root):

           return True

       else:

           return False

    def delete(self,key):

      if self.size > 1:

         nodeToRemove = self.\_get(key,self.root)

         if nodeToRemove:

             self.remove(nodeToRemove)

             self.size = self.size-1

         else:

             raise KeyError('Error, key not in tree')

      elif self.size == 1 and self.root.key == key:

         self.root = None

         self.size = self.size - 1

      else:

         raise KeyError('Error, key not in tree')

    def \_\_delitem\_\_(self,key):

       self.delete(key)

    def spliceOut(self):

       if self.isLeaf():

           if self.isLeftChild():

                  self.parent.leftChild = None

           else:

                  self.parent.rightChild = None

       elif self.hasAnyChildren():

           if self.hasLeftChild():

                  if self.isLeftChild():

                     self.parent.leftChild = self.leftChild

                  else:

                     self.parent.rightChild = self.leftChild

                  self.leftChild.parent = self.parent

           else:

                  if self.isLeftChild():

                     self.parent.leftChild = self.rightChild

                  else:

                     self.parent.rightChild = self.rightChild

                  self.rightChild.parent = self.parent

    def findSuccessor(self):

      succ = None

      if self.hasRightChild():

          succ = self.rightChild.findMin()

      else:

          if self.parent:

                 if self.isLeftChild():

                     succ = self.parent

                 else:

                     self.parent.rightChild = None

                     succ = self.parent.findSuccessor()

                     self.parent.rightChild = self

      return succ

    def findMin(self):

      current = self

      while current.hasLeftChild():

          current = current.leftChild

      return current

    def remove(self,currentNode):

         if currentNode.isLeaf(): #leaf

           if currentNode == currentNode.parent.leftChild:

               currentNode.parent.leftChild = None

           else:

               currentNode.parent.rightChild = None

         elif currentNode.hasBothChildren(): #interior

           succ = currentNode.findSuccessor()

           succ.spliceOut()

           currentNode.key = succ.key

           currentNode.payload = succ.payload

         else: # this node has one child

           if currentNode.hasLeftChild():

             if currentNode.isLeftChild():

                 currentNode.leftChild.parent = currentNode.parent

                 currentNode.parent.leftChild = currentNode.leftChild

             elif currentNode.isRightChild():

                 currentNode.leftChild.parent = currentNode.parent

                 currentNode.parent.rightChild = currentNode.leftChild

             else:

                 currentNode.replaceNodeData(currentNode.leftChild.key,

                                    currentNode.leftChild.payload,

                                    currentNode.leftChild.leftChild,

                                    currentNode.leftChild.rightChild)

           else:

             if currentNode.isLeftChild():

                 currentNode.rightChild.parent = currentNode.parent

                 currentNode.parent.leftChild = currentNode.rightChild

             elif currentNode.isRightChild():

                 currentNode.rightChild.parent = currentNode.parent

                 currentNode.parent.rightChild = currentNode.rightChild

             else:

                 currentNode.replaceNodeData(currentNode.rightChild.key,

                                    currentNode.rightChild.payload,

                                    currentNode.rightChild.leftChild,

                                    currentNode.rightChild.rightChild)

mytree = BinarySearchTree()

mytree[3]="red"

mytree[4]="blue"

mytree[6]="yellow"

mytree[2]="at"

print(mytree[6])

print(mytree[2])