Binary Search Trees

Data and File Structures Laboratory

http://www.isical.ac.in/~dfslab/2020/index.html

Binary search trees

Definition

Binary tree in which following property holds for all nodes:

- *key values in left subtree are less than key value in the node*
- *key values in right subtree are greater than key value in the node*

Main operations

- **Insertion**
- **Search**
- **Deletion**

I/O operations

- **Read / construct**
- **Print**

Auxiliary operations

- **Find successor**
- **Find predecessor**

Binary tree implementation (from earlier class)

- All nodes of the tree are stored in a list.
- Each node is a dictionary with 3 or 4 keys: data, left, right, parent (optional).
- For a node, 'left' and 'right' store the indices of the left and right child of that node.
- If a node has no left / right child, the corresponding field is set to -1.

Binary tree implementation: revisited

```
class Node():
    """
    A single node in a binary tree.
    Attributes: data, left, right, parent.
    """

    def __init__(self, data, left, right, parent):
        self.data = data
        self.left = left
        self.right = right
        self.parent = parent
```

Binary tree implementation: revisited

```
class BinaryTree():
    """
    Binary tree, with nodes stored in a list.
    left, right, parent attribute of each node contains the index in
    the list of the left / right child / parent node.
    If a node has no left / right child, the corresponding field is
    set to -1.
    """

    def __init__(self, capacity):
        self.capacity = capacity
        self.nodelist = [ Node(0, -1, -1, -1) for i in range(capacity) ]
        self.num_nodes = 0
        self.root = -1
```

instead of dictionary keys

pre-allocate a list, so that we don't have to call append all the time

Binary search trees

```
class BinarySearchTree(BinaryTree):
    # If no init is provided, the init inherited from the superclass'
    # will be called. Since that is all we're doing here, we don't
    # really need this __init__.
    # NOTE: If we do provide an __init__, that init will not
    # automatically call super().__init__()
    def __init__(self, capacity):
        super().__init__(capacity)

    def search(self, start_index, d) :
        if start_index == -1 : return -1
        if d < self.nodelist[start_index].data :
            return self.search(self.nodelist[start_index].left, d)
        elif d > self.nodelist[start_index].data :
            return self.search(self.nodelist[start_index].right, d)
        else : return start_index
```

BST insertion

```
def insert(self, start_index, d) :
```

```
    ...
```

```
    current_value = self.nodelist[start_index].data
    if d < current_value :
        # insert recursively in left if less
        left = self.nodelist[start_index].left
        self.nodelist[start_index].left = self.insert(left, d)
    elif d > current_value :
        # insert recursively in right if more
        right = self.nodelist[start_index].right
        self.nodelist[start_index].right = self.insert(right, d)
    # ignore if present
    return start_index
```

BST insertion

```
def insert(self, start_index, d) :
```

```
    ...
```

```
    current_value = self.nodelist[start_index].data
    if d < current_value :
        # insert recursively in left if less
        left = self.nodelist[start_index].left
        self.nodelist[start_index].left = self.insert(left, d)
    elif d > current_value :
        # insert recursively in right if more
        right = self.nodelist[start_index].right
        self.nodelist[start_index].right = self.insert(right, d)
    # ignore if present
    return start_index
```

Why do we need an assignment instead of just a recursive call?

BST insertion

```
def insert(self, start_index, d) :
```

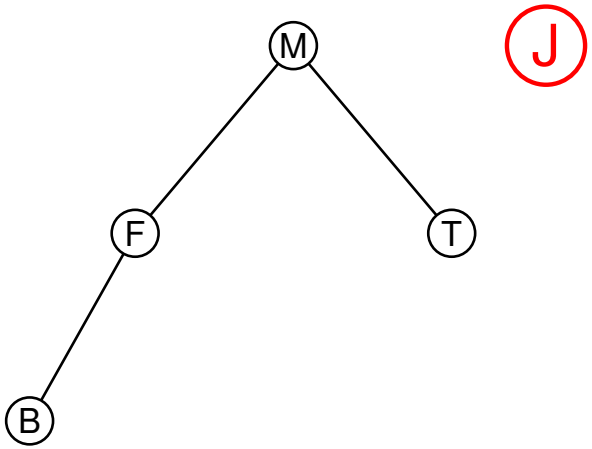
```
    ...
```

```
    current_value = self.nodelist[start_index].data
    if d < current_value :
        # insert recursively in left if less
        left = self.nodelist[start_index].left
        self.nodelist[start_index].left = self.insert(left, d)
    elif d > current_value :
        # insert recursively in right if more
        right = self.nodelist[start_index].right
        self.nodelist[start_index].right = self.insert(right, d)
    # ignore if present
    return start_index
```

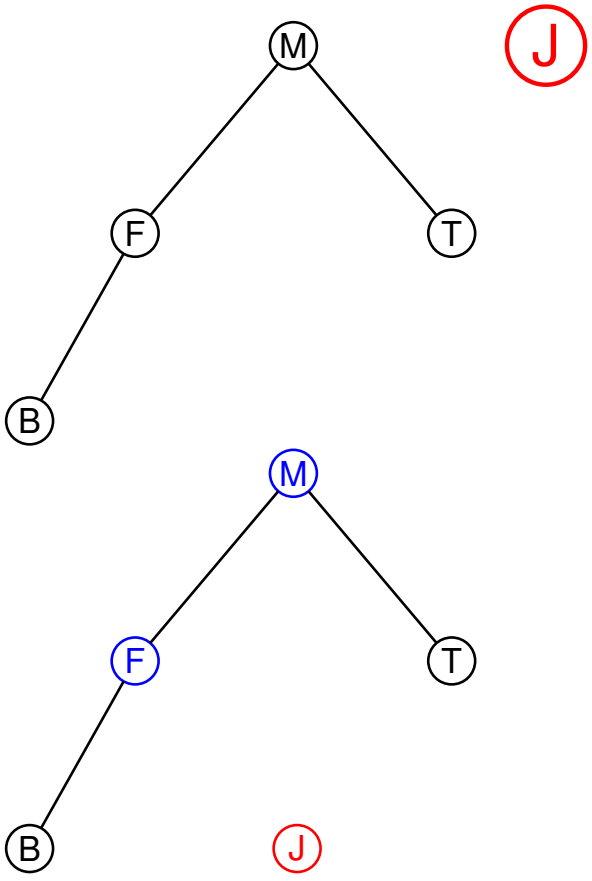
Why do we need an assignment instead of just a recursive call?

Answer: because insertion generally results in a new node being created. The index of this new node within `nodelist` will be the new left / right child of some node, or the new root (for the first insertion into an empty tree). See following example.

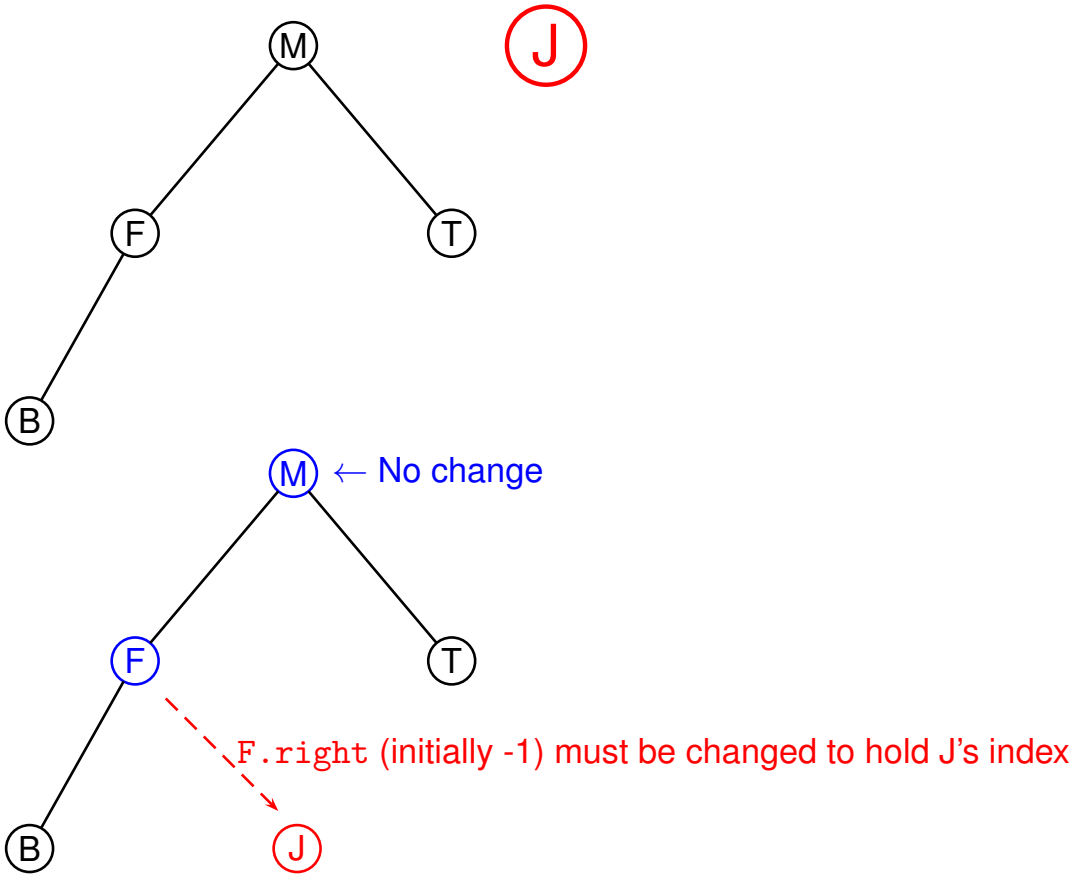
BST insertion (contd.)



BST insertion (contd.)



BST insertion (contd.)



BST insertion (contd.)

```
def insert(self, start_index, d) :
    if start_index == -1 : # empty tree or have reached insertion point
        if self.capacity == self.num_nodes : # self.nodelist is full
            self.nodelist = self.nodelist + \
                [ Node(0, -1, -1, -1) for i in range(capacity) ]
            self.capacity *= 2
        new_node_index = self.num_nodes
        self.nodelist[new_node_index].data = d
        self.num_nodes += 1
```

BST deletion

Let X be the node to be deleted.

Case I X is a leaf node.
Simply delete X.

Case II X has one child.
Replace the link to X with a link to its only child.

Case III X has 2 children.

1. Find S, the successor of X (node with smallest key in right subtree of X).
2. Replace the value in X by the value in S.
3. Delete node S from the tree (see Cases I and II above).

May also use X's predecessor, the largest key in left subtree of X in a similar fasion.

BST deletion: helper function

```
def detach_successor(self, start_index) :
    assert start_index != -1
    # Go to right child, then as far left as possible
    child = self.nodelist[start_index].right
    # If this function has been called, node has both children
    assert child != -1

    if self.nodelist[child].left == -1 :
        self.nodelist[start_index].right = self.nodelist[child].right
        return child

    while self.nodelist[child].left != -1 :
        current_node = child
        child = self.nodelist[child].left

    self.nodelist[current_node].left = self.nodelist[child].right
    return child
```

BST deletion

```
def delete(self, start_index, d) :
    if start_index == -1 : # not found
        print(d, " not found")
        return None
    current_value = self.nodelist[start_index].data
    if d < current_value :
        # delete recursively from left if less
        # print("Deleting", d, " from left subtree")
        left = self.nodelist[start_index].left
        status = self.delete(left, d)
        if status != None :
            self.nodelist[start_index].left = status
    elif d > current_value :
        # delete recursively from right if more
        # print("Deleting", d, " from right subtree")
        right = self.nodelist[start_index].right
        status = self.delete(right, d)
```

```
        ...
```

BST deletion (contd.)

```
else : # found it
    self.num_nodes -= 1
    print("Found", d)
    # Case I: value found at leaf node, just delete
    if self.nodelist[start_index].left == -1 and \
       self.nodelist[start_index].right == -1 :
        print("Deleting leaf node", d)
        return -1
    # Case II: only one child
    if self.nodelist[start_index].left == -1 :
        print("Deleted node has only one child")
        return self.nodelist[start_index].right
    if self.nodelist[start_index].right == -1 :
        print("Deleted node has only one child")
        return self.nodelist[start_index].left
    # Case III: both sub-trees present
    s = self.detach_successor(start_index)
    print("Successor:", self.nodelist[s].data)
    self.nodelist[start_index].data = self.nodelist[s].data
return start_index
```