Homework 4 - Binary Search Tree

```
2/c Joram Stith
COM 212 Data Structures
Dr. Parker
10FEB2022
```

Problem 4a

```
Problem #4a due on 10 Feb.
Describe how to implement a binary search tree using a set of nodes (this tree
will have no number of element limit).
Do the six BST functions.
Can you determine how to implement a binary search tree using an array (assume it
will never have a depth of more than 7)?
Consider the six BST functions.
```

This additional information is provided about the definition of a binary search tree and functions for a binary search tree:

```
Binary Search Tree

Tree T is a binary search tree made up of n elements: x0 x1 x2 x3 ... xn-1

Functions:
createEmptyTree()  returns a newly created empty binary tree
delete(T, p)       removes the node pointed to by p from the tree T
insert(T, p)       returns T with the node pointed to by p added in
                   the proper location
search(T, key)     returns a pointer to the node in T that has a key
                   that matches key
           returns null if the node is not found
traverse(T)    prints the contents of T in order
isEmptyTree(T)     returns true if T is empty and false if it is not
```

Functions	Definitions
createEmptyTree()	returns a newly created empty binary tree
delete(T, p)	removes the node pointed to by p from the tree T
insert(T, p)	returns T with the node pointed to by p added in the proper location

Functions	Definitions
search(T, key)	returns a pointer to the node in T that has a key that matches key, returns null if the node is not found
traverse(T)	prints the contents of T in order.
isEmptyTree(T)	returns true if T is empty and false if it is not

Variable Size with Pointers and Nodes

Declaration

To declare a binary tree, you start with an empty pointer. Once the first node is created, the pointer representing the tree object will point to the first node (the trunk). That node will be populated with a value and encapsulated with two other pointers.

createEmptyTree()

returns a newly created empty binary tree

Like the declaration paragraph, create a node with a value of null, and two pointers with that node that also point to null.

```
def createEmptyTree():
    create node trunk
    trunk.value = null
  # a and b are the pointers to the next value
  trunk.a = null
    trunk.b = null
    return trunk
```

delete(T, p)

removes the node pointed to by p from the tree T

To remove a node, you need to find the node above it and re-assign it's pointer. This can be done in O(lg[n]) using a binary search tree. Start at the trunk, and perform a greater than or less than comparison to each value, traversing the appropriate branch (pointer) until you reach a value pointing to p, then change that pointer to null.

```
def delete(T, p):
  current_node = T.trunk
  while True:
    if current_node.a == p:
      current_node.a = null
      return
    if current_node.b == p:
      current_node.b = null
```

```
      return
    if p > current_node:
      current_node = current_node.a
    elif p < current_node:
      current_node = current_node.b
    else:
      # Edge case, p is the trunk
      T = null
      return
```

insert(T, p)

returns T with the node pointed to by p added in the proper location

To insert a value, we follow the same procedure as deleting to find the appropriate location, but rather than set the pointer to null we set it to the passed value p.

```
def insert(T, p):
  current_node = T.trunk
  while True:
    if p > current_node and p < current_node.a:
      current_node.b = p
      return
    elif p < current_node and p > current_node.b:
      current_node.a = p
      return
    elif p > current_node:
      current_node = current_node.a
    elif p < current_node:
      current_node = current_node.b
    else:
      # Edge case, p is equal to a value that already exists
      return
```

search(T, key)

returns a pointer to the node in T that has a key that matches key, returns null if the node is not found

To search for a value, we will once again go through the tree comparing each value for greater than or less than the key. Once we find the key, we return a pointer to that key.

```
def search(T, key):
  current_node = T.trunk
  while True:
    if current_node.a == key:
      return current_node.a
    if current_node.b == key:
      return current_node.b
```

```
    if key > current_node:
      current_node = current_node.a
    elif key < current_node:
      current_node = current_node.b
    else:
      # Edge case, p is equal to current node
      return T
```

traverse(T)

prints the contents of T in order.

To print all of the contents of T in order, We can use a recursive loop to go through each value in the tree one at a time. This function is necessarily >= O(n) because n values get accessed. I'm not sure this is the *best* way to do this, but I wanted to make something work with recrusion so here's my best shot (consider it a make up for my poor attempt at the towers of hanoi problem).

```
def printNode(T, p):
  if(p.a == null and p.b == null):
    print(p)
    return
  elif(p.a == null):
    return printNode(T, p.b)
  elif(p.b == null):
    return printNode(T, p.a)
  else:
    printNode(T, p.b)
    return printNode(T, p.a)

def traverse(T):
  current_node = T.trunk
  printNode(T, current_node)
  return
```

isEmptyTree(T)

returns true if queue is empty

To determine if a queue is empty, simple check if the passed value of T is pointing to null.

```
def isEmptyTree(T):
    return T == null
```

Other Notes

A lot of these functions feel like they're not O(lg[n]), and they wouldn't be if elements were unsorted. But, since all items in the binary tree are necessarily ordered and sorted, we can use loops and still achieve

O(lg[n]).

Set Size with Arrays

Methodology

To implement a binary tree of a set size with an array, you would follow the same logic, just with much more confusing indexing. I would put the trunk node at the start of the array, then we know the next two array values are the children on that trunk. Then, we know the next four array values are the children of those trunks, etc.

```
[TRUNK, Branch0, Branch1, Branch00, Branch01, Branch10, Branch11, Branch000,
Branch001, Branch010, Branch 011, ...]
```

These values could be accessed by using `pow(2,n)` to see how far down the array you must traverse to get to the next node, but you would be jumping from high values to low values very often, and it would not be more efficient than a node and pointer based system.

Considering the Functions

createEmptyTree()

Declare the array of the set size (128 if we limit to 2^7 like the instructions say).

delete(T, p)

Traverse the binary tree with the same methodology as with nodes (get to an element, compare if the element is greater or less than the passed value, then jump to the according child node). Use an index-ish variable `n` to count how many levels deep in the tree you are, then you can use `pow(2,n) + offset_from_comparison` to get to the next set of child nodes to compare. Once you reach the index that gets deleted, set its value to null.

insert(T, p)

Follow the same process as the delete function, but instead of setting the value in the array to null you set it to the passed value p.

search(T, key)

Not shockingly, search follows a similar function, too. However, when you find the key (should be done in O(lg[n])), return the index of the array rather than the value at that index.

traverse(T)

This function will behave similarly to the pointer based function except rather than jumping nodes it will jump values of the array. Since it's set size, you don't need the recursive function, you can use a for loop with 7 iterations and use `pow(2,7) + offset` to get to the children of nodes.

isEmptyTree(T)

Return true if T is null.

Other Notes

This does not seem like a very practical way to implement binary trees, and I assume it's not used much in modern computing (since arrays are really just sets of pointers when you get deep enough anyway). Interesting assignment, I like that things are getting more complex!