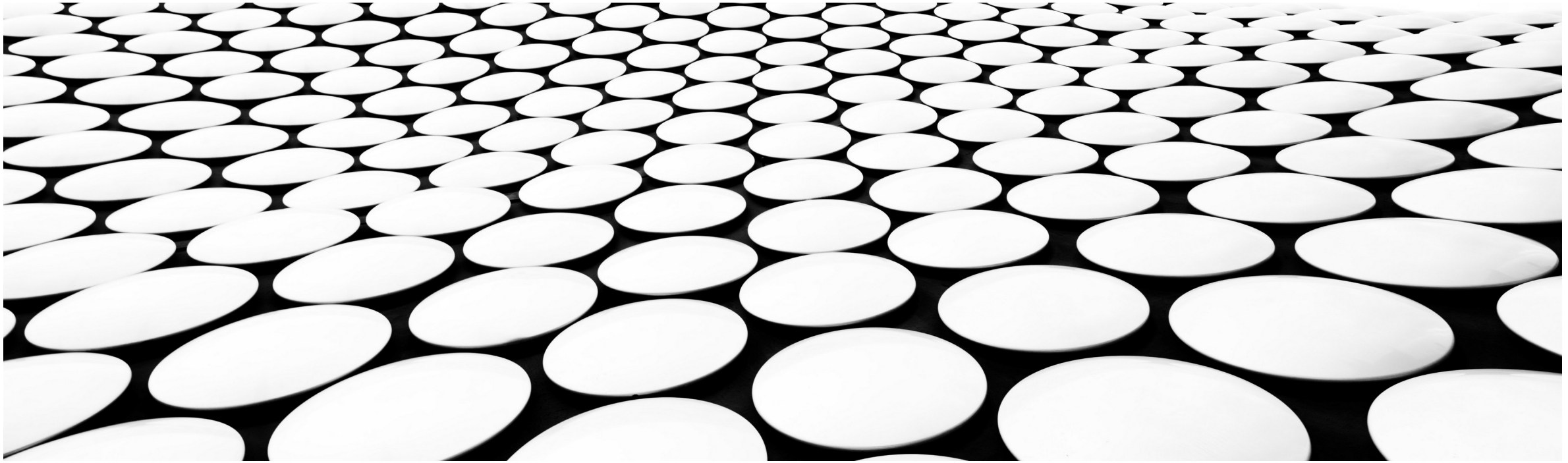
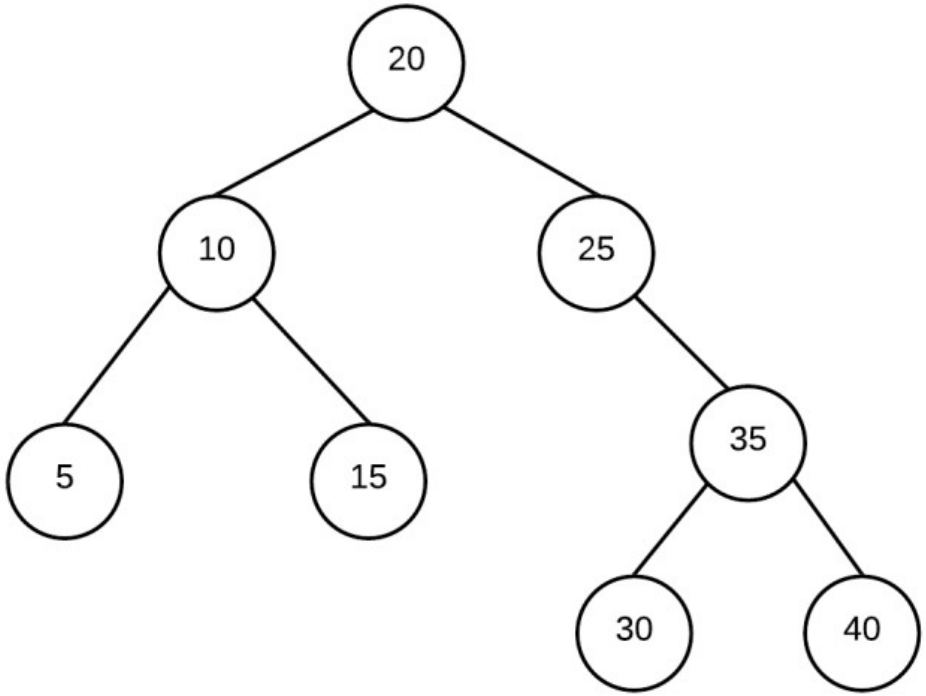
DSA 456 – WEEK 11

BINARY SEARCH TREES



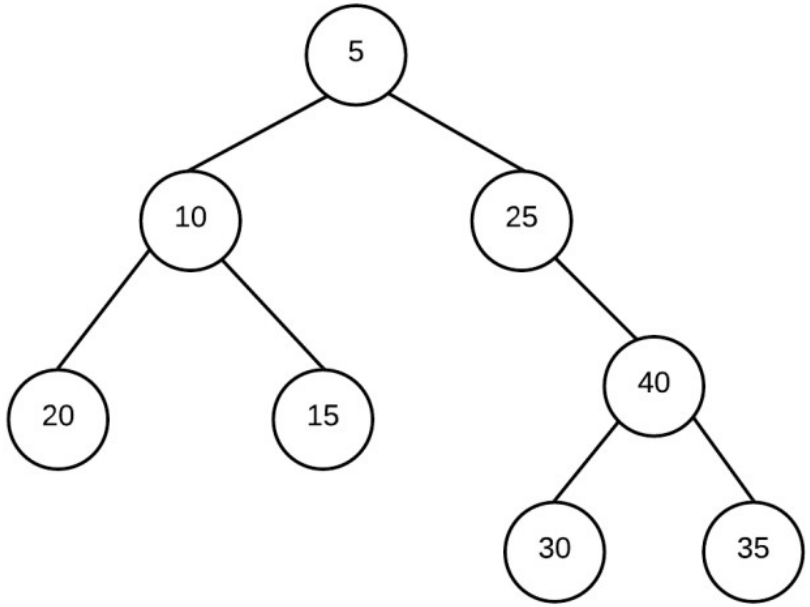
BINARY SEARCH TREES

Binary search trees (BST) are binary trees where values are placed in a way that supports efficient searching. In a BST, all values in the left subtree value in current node < all values in the right subtree. This rule must hold for EVERY subtree, ie every subtree must be a binary search tree if the whole tree is to be a binary tree. The following is a binary search tree:



BINARY SEARCH TREES

The following is NOT a binary search tree as the values are not correctly ordered:

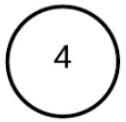


A binary search tree allows us to quickly perform a search on a linking structure (under certain conditions which we will explore later). To find a value, we simply start at the root and look at the value. If our key is less than the value, search left subtree. If key is greater than value search right subtree. It provides a way for us to do a "binary search" on a linked structure which is not possible with a linear linked list. During the search, we will never have to search the subtrees we eliminate in the search process... thus at worst, searching for a value in a binary search tree is equivalent to going through all the nodes from the root to the furthest leaf in the tree.

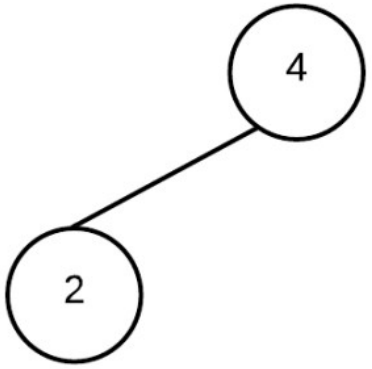
INSERTION

To insert into a binary search tree, we must maintain the nodes in sorted order. There is only one place an item can go.
Example Insert the numbers 4,2,3,5,1, and 6 in to an initially empty binary search tree in the order listed. Insertions always occur by inserting into the first available empty subtree along the search path for the value:

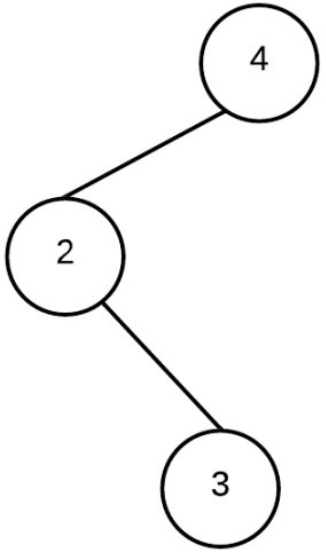
Insert 4:



Insert 2: 2 is < 4 so it goes left

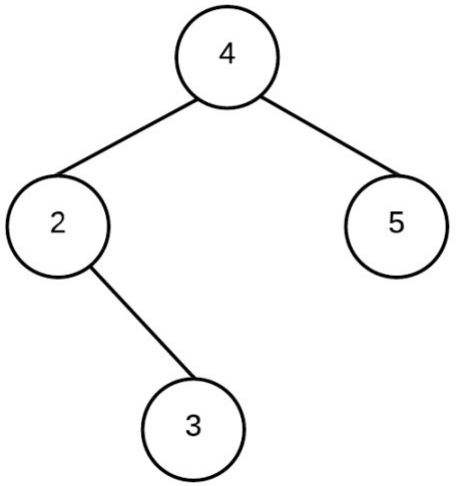


Insert 3: 3 is less than 4 but more than 2 so it goes to left of 4, but right of 2

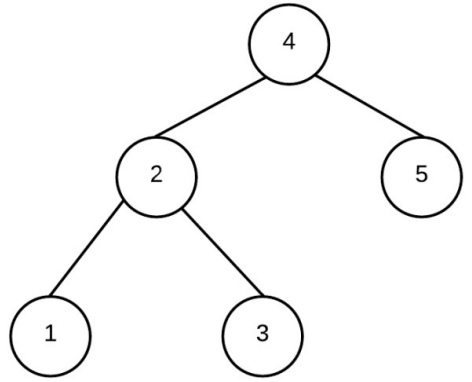


INSERTION

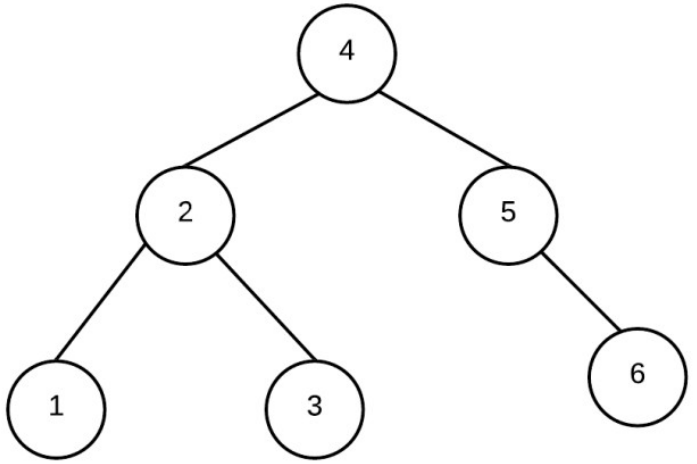
Insert 5: 5 is > 4 so it goes right of 4



Insert 1: 1 is < 4 so it goes to left of 4. 1 is also < 2 so it goes to left of 2



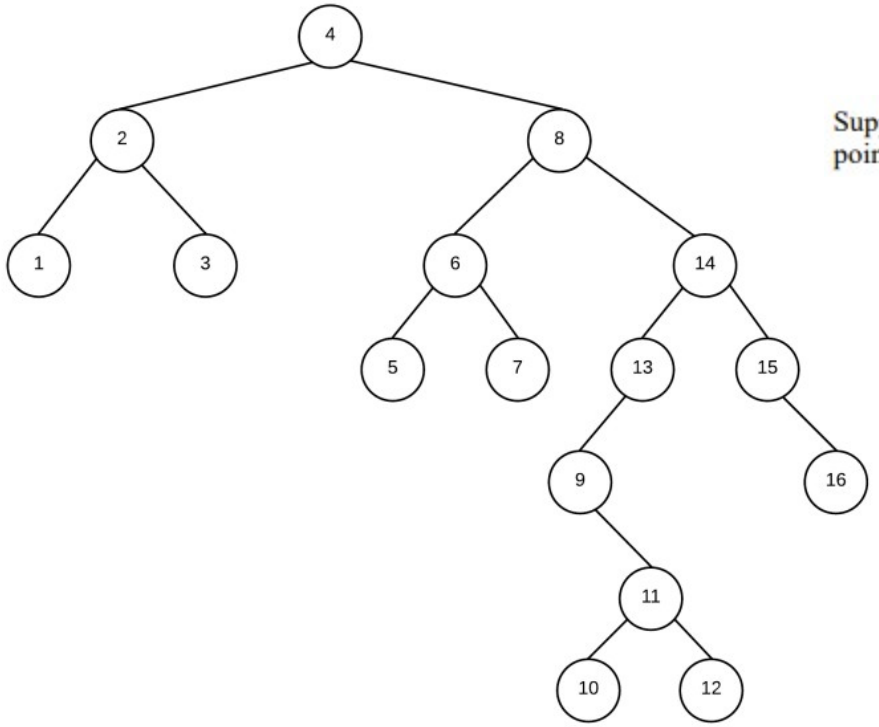
Insert 6: 6 is > 4 so it goes to right of 4. 6 is also > 5 so it goes to right of 5



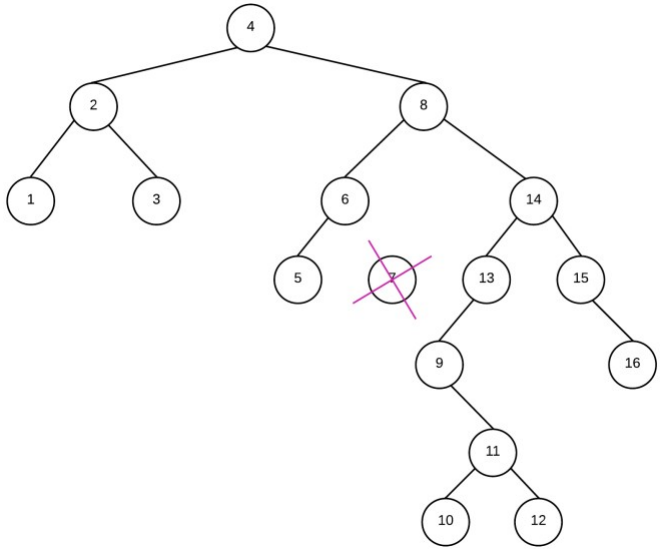
REMOVAL

In order to delete a node, we must be sure to link up the subtree(s) of the node properly. Let us consider the following situations.

Suppose we start with the following binary search tree:

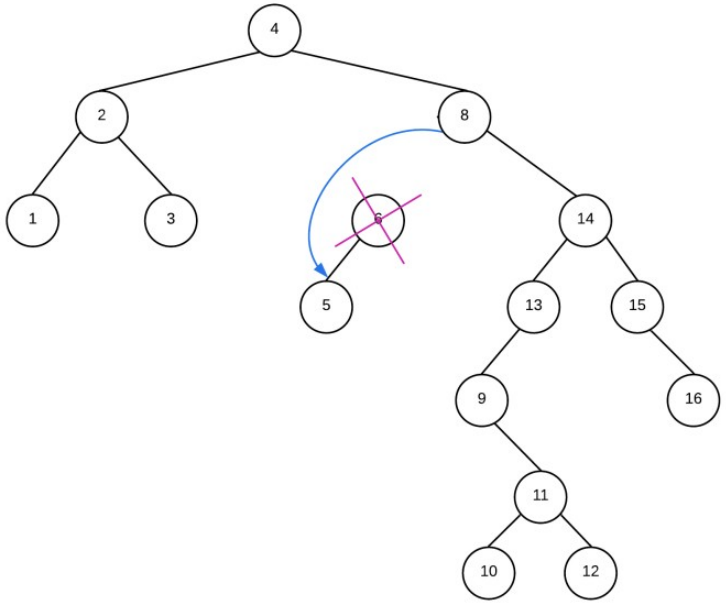


Suppose we were to remove 7 from the tree. Removing this node is relatively simple, we simply have to ensure that the pointer that points to it from node 6 is set to nullptr and delete the node:

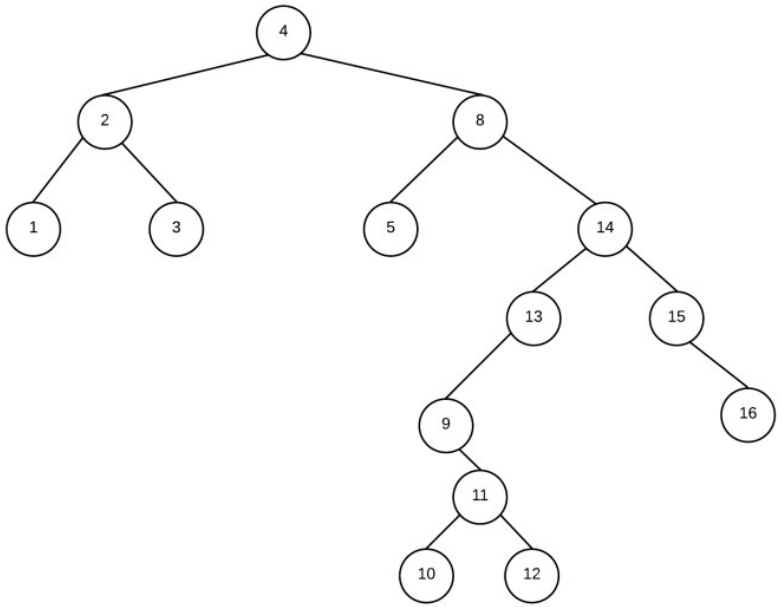


REMOVAL

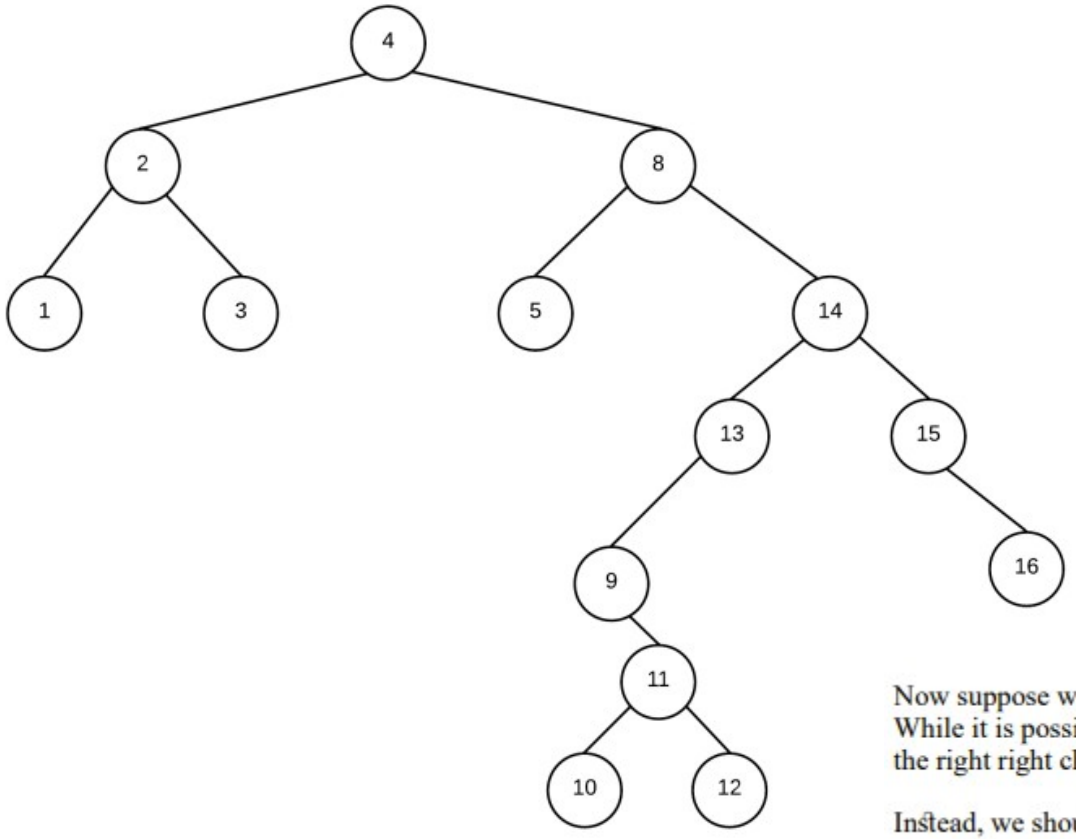
Now, Lets remove the node 6 which has only a left child but no right child. This is also easy. all we need to do is make the pointer from the parent node point to the left child.



Thus our tree is now:



REMOVAL

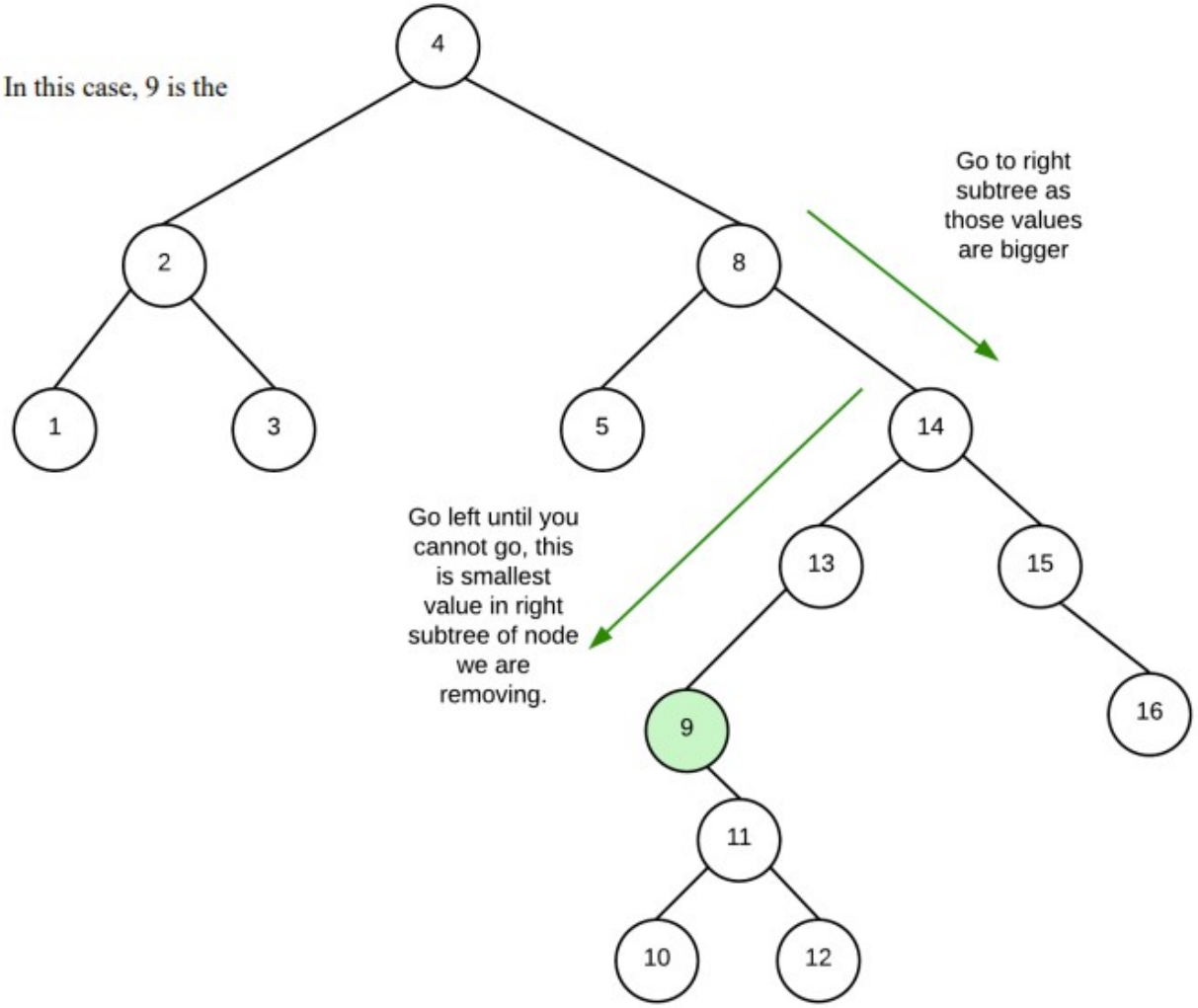


Now suppose we wanted to remove a node like 8. We cannot simply make 4 point to 8's child as there are 2 children. While it is possible to do something like make parent point to right child then attach left child to the left most subtree of the right right child, doing so would cause the tree to be bigger and is not a good solution.

Instead, we should find the inorder successor (next biggest descendent) to 8 and promote it so that it replaces 8.

REMOVAL

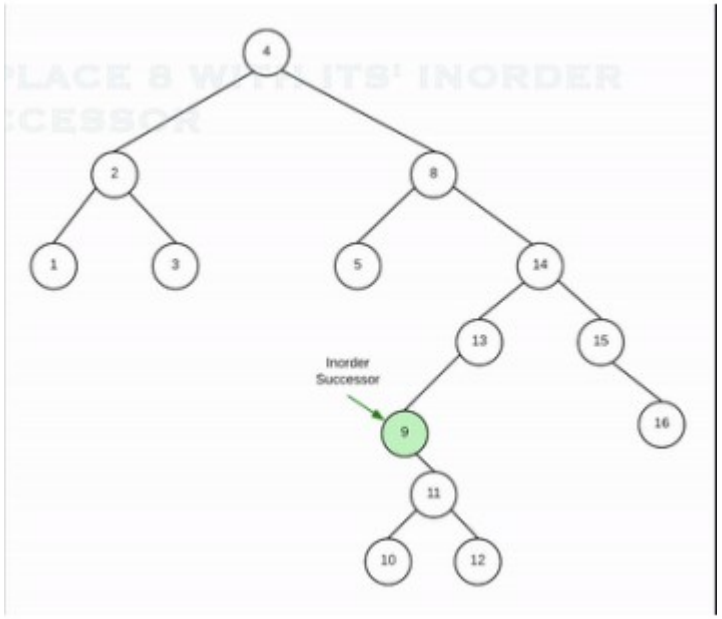
The inorder successor can be found by going to the right child of 8 then going as far left as possible. In this case, 9 is the inorder sucessor to 8:



REMOVAL

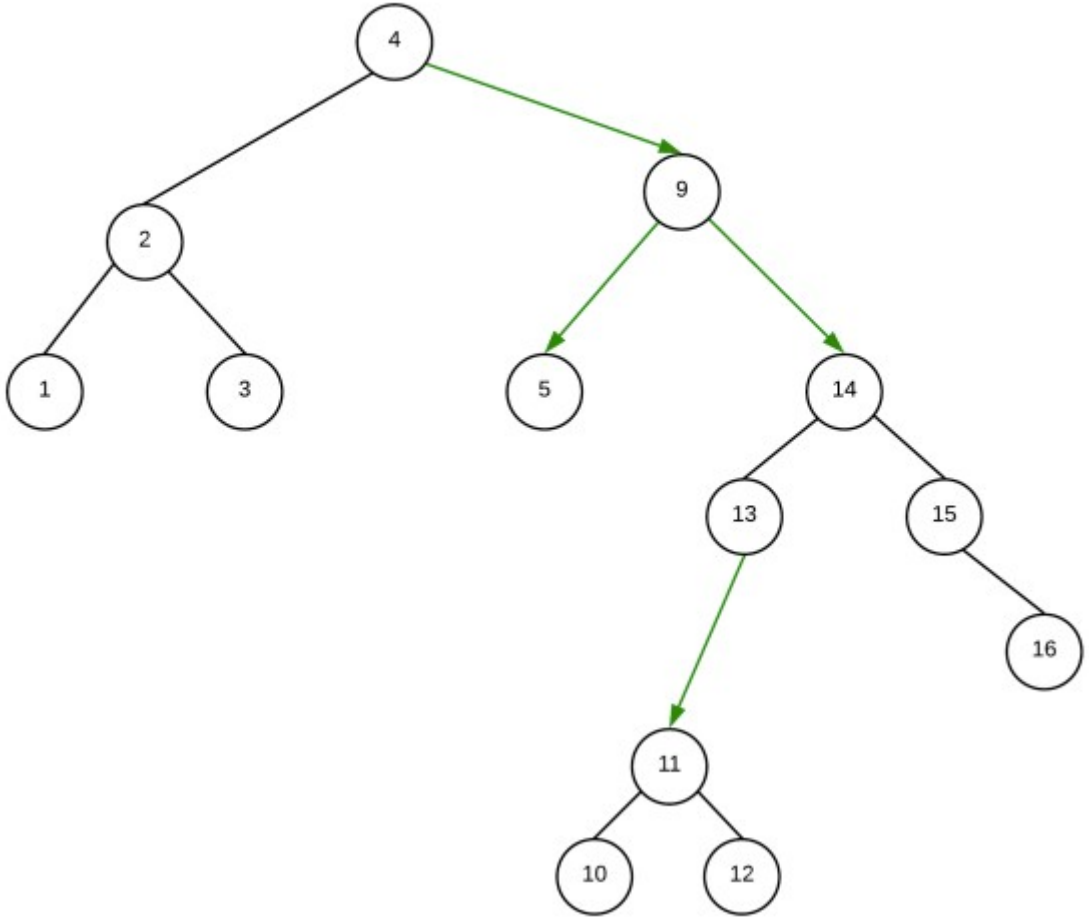
The inorder successor will either be a leaf node or it will have a right child only. It will never have a left child because we found it by going as far left as possible.

Once found, we can promote the inorder successor to take over the place of the node we are removing. The parent of inorder successor must link to the right child of the inorder succesor.



REMOVAL

In the end we have:



Traversals

There are a number of functions that can be written for a tree that involves iterating through all nodes of the tree exactly once. As it goes through each node exactly 1 time, the runtime should not exceed O(n)

These include functions such as print, copy, even the code for destroying the structure is a type of traversal.

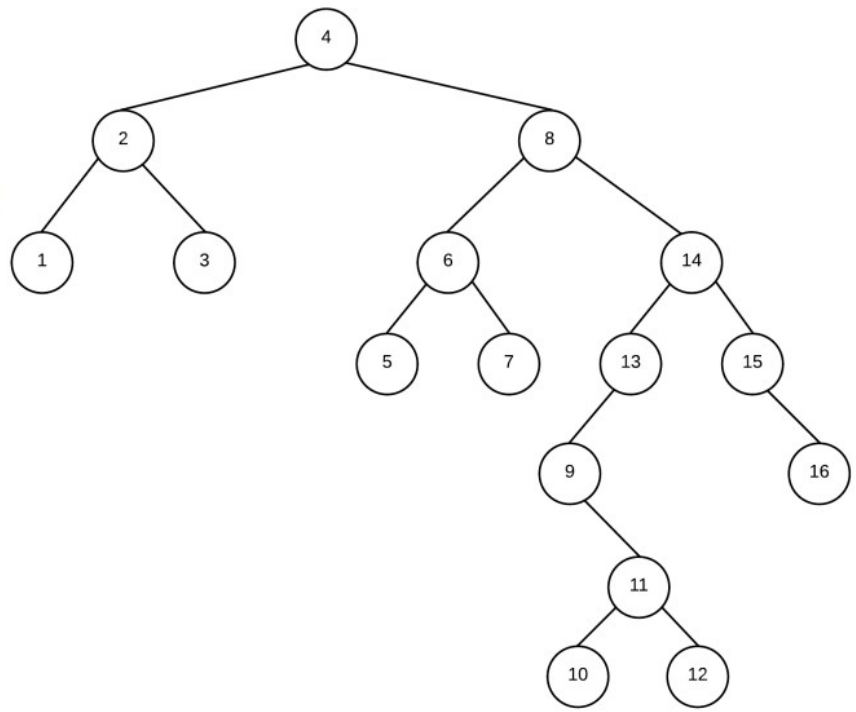
Traversals can be done either depth first (follow a branch as far as it will go before backtracking to take another) or breadfirst, go through all nodes at one level before going to the next.

Depth First Traversals

There are generally three ordering methods for depth first traversals. They are:

- preorder
- inorder
- postorder

In each of the following section, we will use this tree to describe the order that the nodes are "visited". A visit to the node, processes that node in some way. It could be as simple as printing the value of the node:



Preorder traversals

- visit a node
- visit its left subtree
- visit its right subtree

4, 2, 1, 3, 8, 6, 5, 7, 14, 13, 9, 11, 10, 12, 15, 16

Inorder traversals:

- visit its left subtree
- visit a node
- visit its right subtree

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16

Notice that this type of traversal results in values being listed in its sorted order

Postorder traversals:

- visit its left subtree
- visit its right subtree
- visit a node

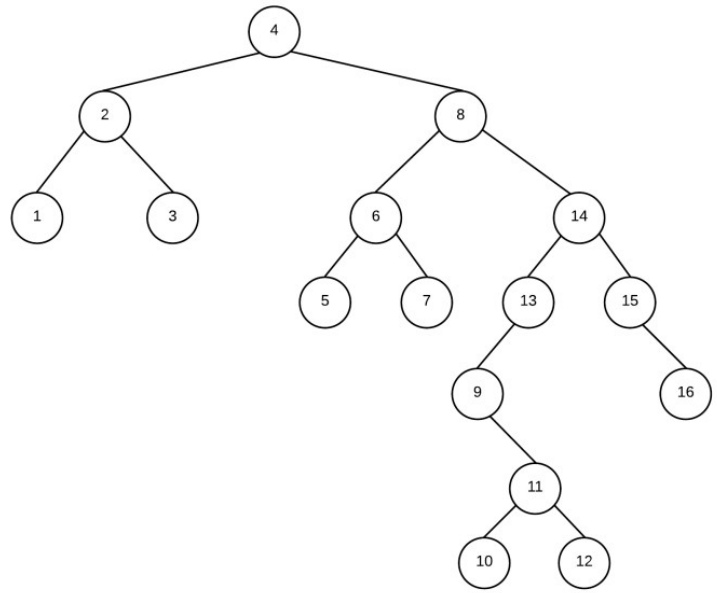
1, 3, 2, 5, 7, 6, 10, 12, 11, 9, 13, 16, 15, 14, 8, 4

This is the type of traversal you would use to destroy a tree.

Breadth-First Traversal

A breadfirst traversal involves going through all nodes starting at the root, then all its children then all of its children's children, etc. In otherwords we go level by level.

Given the following tree:



A breadthfirst traversal would result in:

4, 2, 8, 1, 3, 6, 14, 5, 7, 13, 15, 9, 16, 11, 10, 12

BST Implemenation

To implement a binary search tree, we are going to borrow some concepts from linked lists as there are some parts that are very similar. In these notes we will look a few of the functions and leave the rest as an exercise.

Similar to a linked list, A binary search tree is made up of nodes. Each node can have a left or right child, both of which could be empty trees. Empty trees are represented as nullptrs. The binary search tree object itself only stores a single pointer that points at the root of the entire tree. The data stored within the nodes must be of some type that is comparable. We will thus begin our binary search tree class declaration in a similar manner to that of a linked list. The code for each of the functions will be filled in later in this chapter.

- Python
- C++

```python
class BST:
    class Node:
        # Node's init function
        def __init__(self,data=None,left=None,right=None):
            self.data = data
            self.left = left
            self.right = right

    #BST's init function
    def __init__(self):
        self.root = None

    def insert(self, data):
        ...

    def search(self, data):
        ...

    def inorder_print(self):
        ...

    def pre_order_print(self):
        ...

    def breadthfirst_print(self):
        ...
```

If you rename the data members above you actually see that its pretty similar to that of a doubly linked list... The key to the operations of a BST lies not in what data is declared, but rather how we organize the nodes. The next 2 sections of the notes we will look at the implementation of the functions listed above. In some cases a function may be written both iteratively and recursively and both versions will be looked at

Constructor (C++)

When we create our tree, we are going to start with an empty tree. Thus, our constructor simply needs to initialize the data member to nullptr.

```
template <typename T>
class BST{
    struct Node{
        T data_;
        Node* left_;
        Node* right_;
        Node(const T& data, Node* left=nullptr, Node* right=nullptr){
            data_=data;
            left_=left;
            right_=right;
        }
    };
    //single data member pointing to root of tree
    Node* root_;
public:
    BST(){
        root_=nullptr;
    }
...
};
```

Iterative Methods

This section looks at the functions that are implemented iteratively (or the iterative version of the functions)

Insert - Iterative version

This function will insert data into the tree. There are two ways to implement this function, either iteratively or recursively. We will start by looking at the iterative solution. In this situation, we begin by taking care of the empty tree situation. If the tree is empty we simply create a node and make root_ point to that only node. Otherwise, we need to go down our tree until we find the place where we must do the insertion and then create the node.

- Python
- C++

```
class BST:
    class Node:
        # Node's init function
        def __init__(self, data=None, left=None, right=None):
            self.data = data
            self.left = left
            self.right = right

    # BST's init function
    def __init__(self):
        self.root = None

    def insert(self, data):
        if self.root is None:
            self.root = BST.Node(data)
        else:
            # curr points to the variable we are currently looking at
            curr = self.root
            inserted = False

            while not inserted:
                if data < curr.data:
                    if curr.left is not None:
                        curr = curr.left
                    else:
                        curr.left = BST.Node(data)
                        inserted = True
                else:
                    if curr.right is not None:
                        curr = curr.right
                    else:
                        curr.right = BST.Node(data)
                        inserted = True
```

Search - Iterative version

The key operation that is supported by a binary search tree is search. For our purposes we will simply look at finding out

whether or not a value is in the tree or not. The search operation should never look at the entire tree. The whole point of the binary search tree is to make this operation fast. We should search it so that we can eliminate a portion of the tree with every search operation.

To do this we start at the root and compare that node's data against what we want. If it matches, we have found it. If not, we go either left or right depending on how data relates to the current node. If at any point we have an empty tree (ie the pointer we are using for iterating through the tree becomes nullptr) we stop the search and return false. If we find a node that matches we stop and return true.

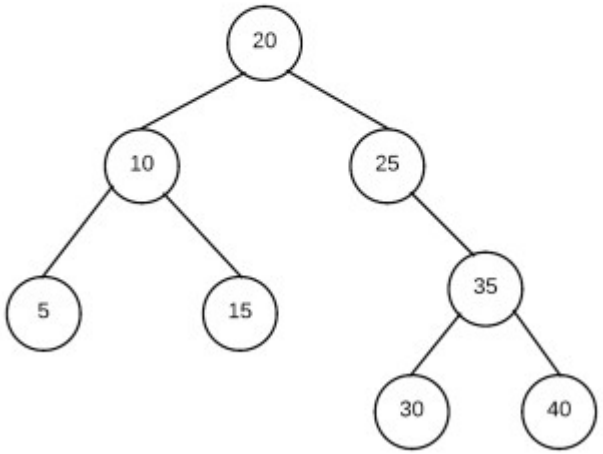
- Python
- C++

```python
class BST:
    class Node:
        # Node's init function
        def __init__(self, data=None, left=None, right=None):
            self.data = data
            self.left = left
            self.right = right

    # BST's init function
    def __init__(self):
        self.root = None

    def search(self, data):
        curr = self.root

        while curr is not None:
            if data < curr.data:
                curr = curr.left
            elif data > curr.data:
                curr = curr.right
            else:
                return curr

        return None
```

Breadth First Print

Writing a breadth-first traversal involves using the queue data structure to order what nodes to deal with next. You want to deal with the nodes from top to bottom left to right, and thus you use the queue to order the nodes. Here is an example of how we will do this.

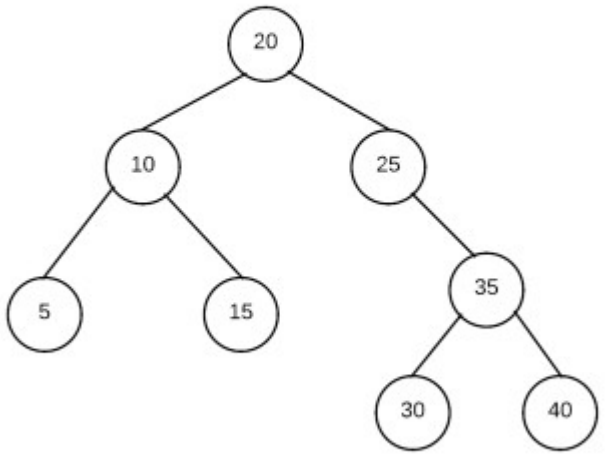
We begin by declaring a queue (initially empty)

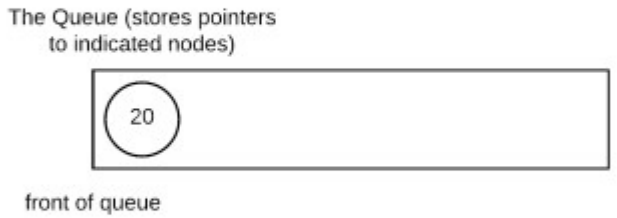


The Queue (stores pointers to indicated nodes)

Prime the Queue

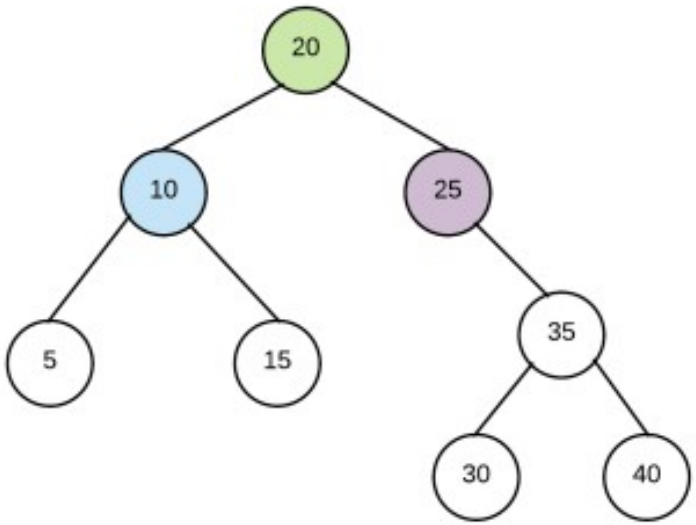
We start prime the queue by putting the root into the queue. In this example, we always check to ensure no nullptrs are ever added to the queue. Alternatively we allow the addition of nullptrs and decide how to deal with them when we dequeue.

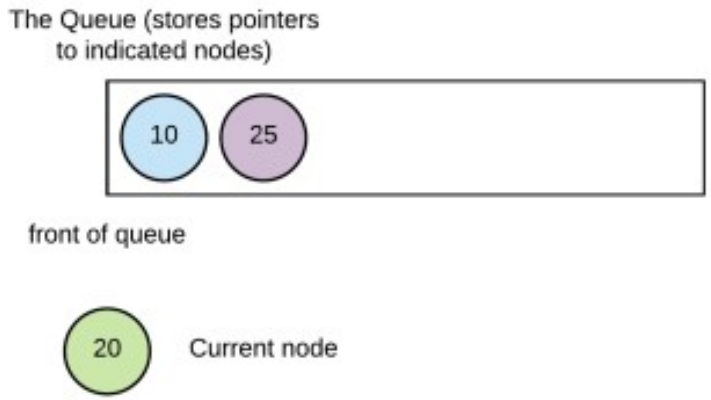




Dequeue front of node and process it by adding its non-nullptr children into the queue, print the node

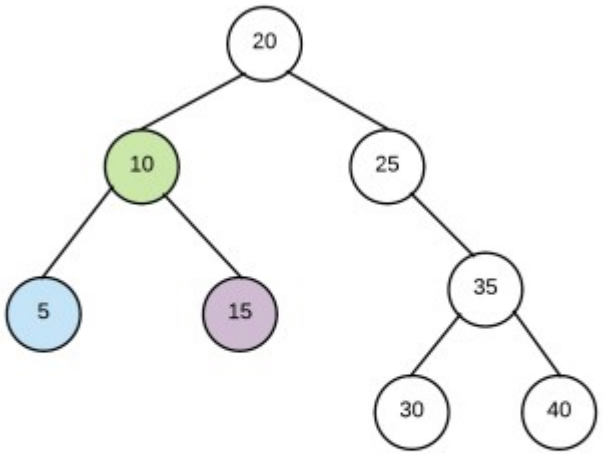
Dequeue front of node and process it by adding its non-nullptr children into the queue, print the node

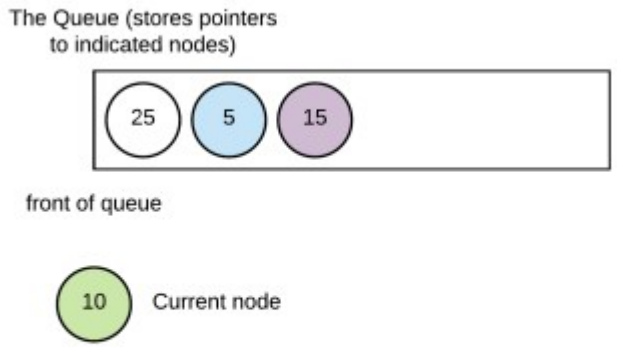




Output: **20**

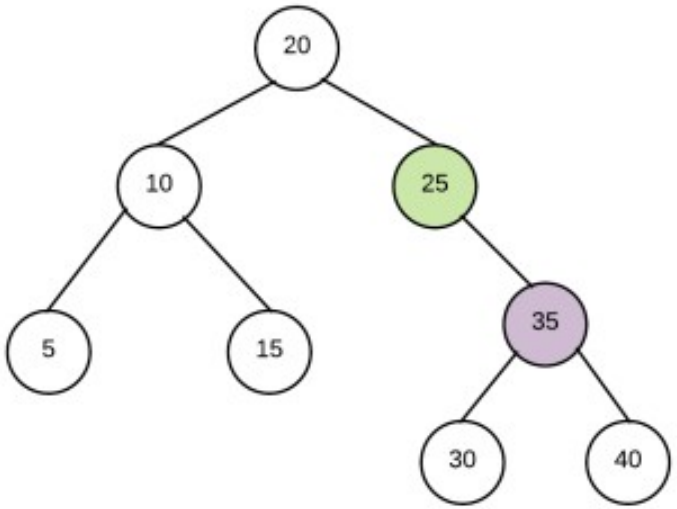
Continue by dequeuing front node and processing it in same manner

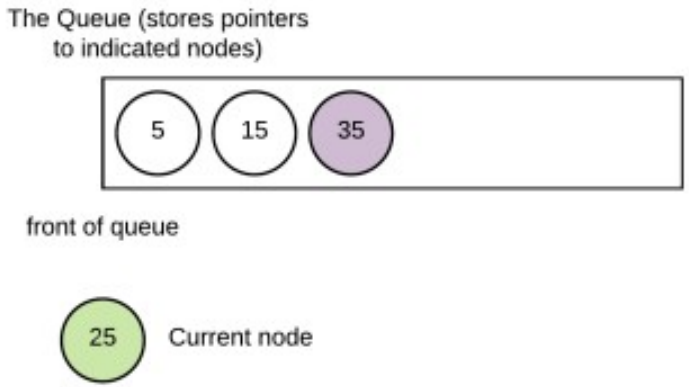




Output: 20 **10**

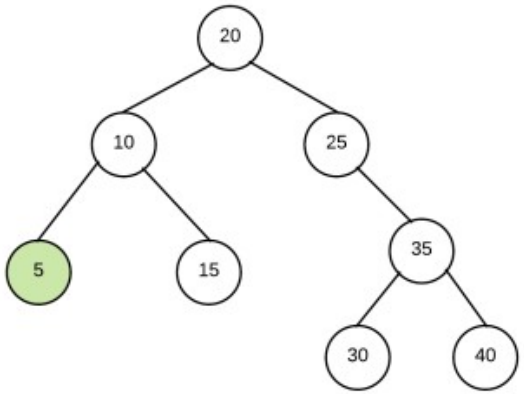
Repeat again as 25 only has a right child only it is added

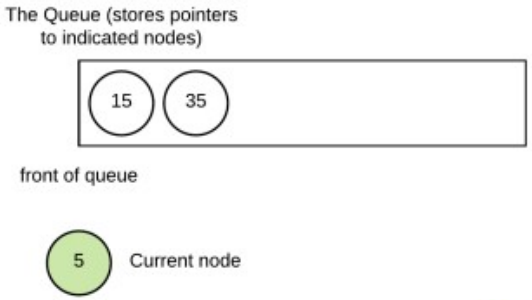




Output: 20 10 **25**

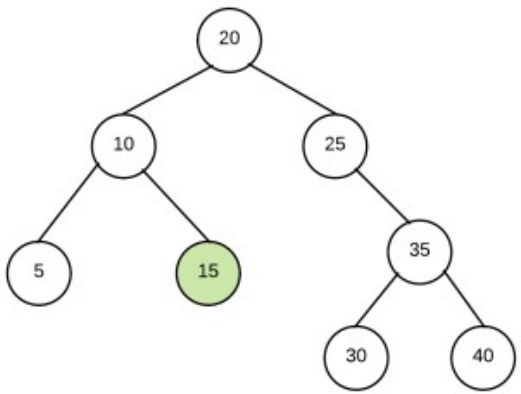
Repeat once again with 5 which has no children thus nothing is added to queue

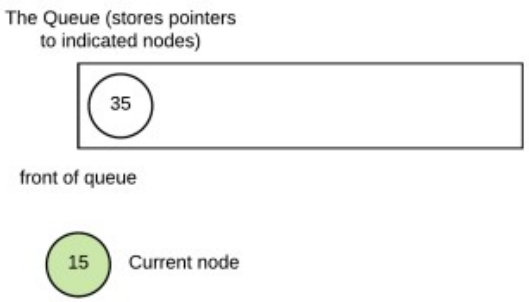




Output: 20 10 25 **5**

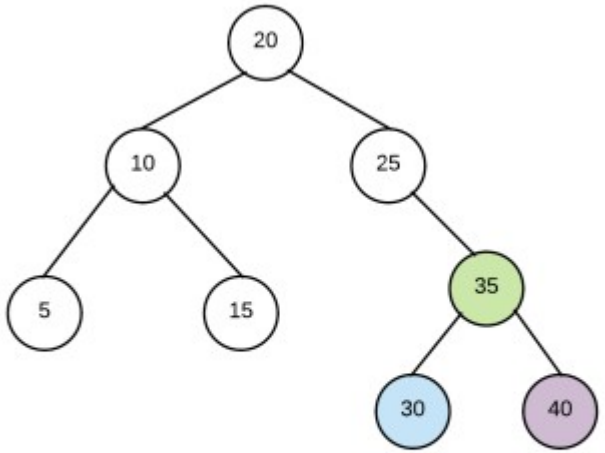
Repeat again with 15 (also no children)

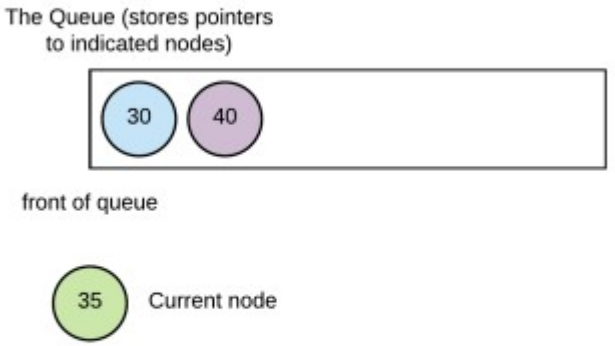




Output: 20 10 25 5 **15**

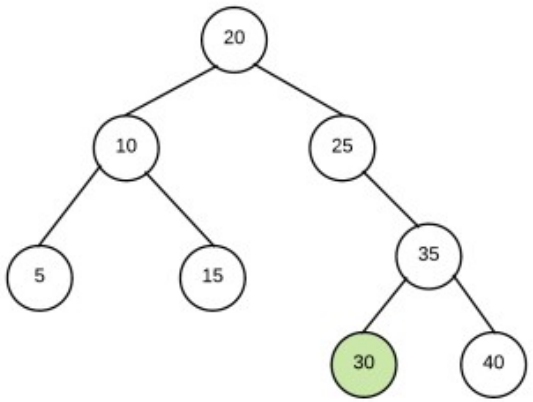
Repeat with 35 and add its children

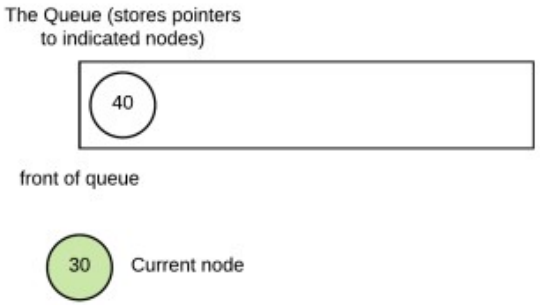




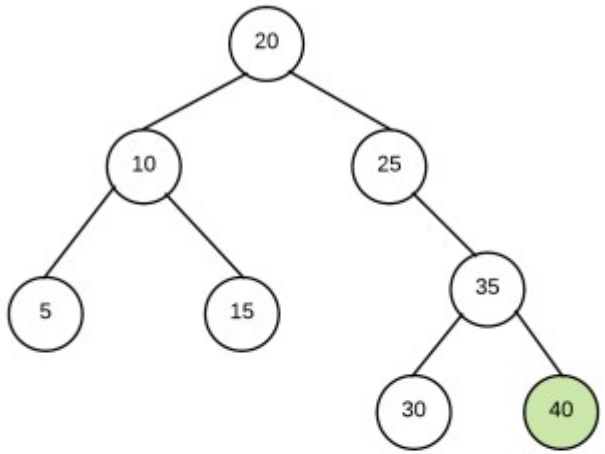
Output: 20 10 25 5 15 **35**

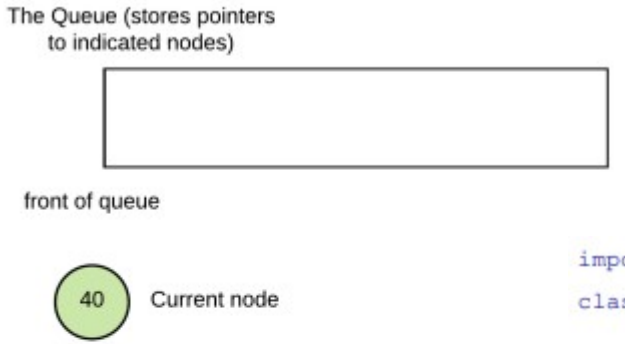
Continue by removing 30





Output: 20 10 25 5 15 35 **30**





Output: 20 10 25 5 15 35 30 40

At this point queue is empty and thus, we have completed our breadthfirst print of the tree.

In code form, this is how we will write our code (we assume we have a template queue available:

- Python
- C++

```python
import queue

class BST:
    class Node:
        # Node's init function
        def __init__(self, data=None, left=None, right=None):
            self.data = data
            self.left = left
            self.right = right

    # BST's init function
    def __init__(self):
        self.root = None

    def breadth_first_print(self):
        the_nodes = queue.Queue()

        if self.root is not None:
            the_nodes.put(self.root)

        while not the_nodes.empty():
            curr = the_nodes.get()

            if curr.left:
                the_nodes.put(curr.left)
            if curr.right:
                the_nodes.put(curr.right)

            print(curr.data, end=" ")
```

Recursive Methods

One of the way to look at a binary search tree is to define the tree recursively. A binary search tree is either empty or consists of a root node who has two children each of which is a binary search tree. This recursive nature makes certain functions easy to write recursively. This portion of the implementation will look at how to write member functions recursively.

For all recursive functions involving a binary search tree, you typically need at least one argument that gives you access to the root of a subtree. The public interfaces for these recursive functions will typically start it off by passing in the root as the first argument.

When writing these functions, we look at the problem in terms of the subtree. Often the base case will involve dealing with an empty subtree (or doing nothing with an empty subtree). This section of the notes will deal with some details about how this can be accomplished and some typical ways of dealing with recursive solutions.

search - recursive function

This is the same function as the iterative version of the search (does the same thing). Only difference is that it is written recursively.

As stated earlier, recursive functions for our trees typically involve looking at it in terms of what to do with a tree (defined as root of the tree).

For the recursive search() function, we will need to write a recursive function and call it from the public search() function. The recursive function will not only have data for the argument but also the root of the subtree where we will be trying to find the data. Note that we cannot use the data member root because we will lose the tree if we do. We must actually pass in the argument so that it can change in each call without causing the root to be modified. Thus, our recursive function has the following prototype:

- Python
- C++

```
def r search(self, data, subtree)
```

The above function returns true if data is in the tree who's root is pointed to by subtree, false otherwise.

As with all recursive cases we always want to start by figuring out the base case of the recursive function

Under what circumstances would it be so easy to find the answer we can confidently return the result immediately? So, given a tree (where all we see is the address of the root node) what situations would allow us to know that we have a result.

In our case there are two base cases:

- the tree is empty. If the tree is empty, the value can't be in the tree, so we can return false immediately.
- the tree is not empty but the value we want is in the root of the tree. If that is the case we don't need to look at the rest of the tree. We already know that its there.

CAUTION

a tree being empty is usually a base case for recursive functions involving a binary search tree. If this is not a base case you are considering, you should ask why and what would happen if you got an empty tree and whether or not you have made a mistake.

So... what if its not a base case? Well if its not a base case, then we know the following:

- there is a tree with at least a root node
- the data isn't in the root node

Thus if the value exists, its either in the left subtree or the right subtree of the root. As the BST places data so that values in right subtree is bigger than value in current node and value in left is smaller, which subtree we look at depends on how the data compares against the root. If the data is smaller than value in root, then it could only be in left subtree if it there at all. Similarly if it is bigger, it could only be in right subtree. Therefore to find if value is in tree we simply make the recursive call to search() using either subtree's left or subtree's right.

INFO

For any function that involves looking for a value in the tree it is never more correct to look at both left and right. The difference will be O(log n) vs O(n)

- Python
- C++

```python
class BST:
    class Node:
        # Node's init function
        def __init__(self, data=None, left=None, right=None):
            self.data = data
            self.left = left
            self.right = right

    # BST's init function
    def __init__(self):
        self.root = None

    def r_search(self, data, subtree):
        if subtree is None:
            return None
        else:
            if data < subtree.data:
                return self.r_search(data, subtree.left)
            elif data > subtree.data:
                return self.r_search(data, subtree.right)
            else:
                return subtree

    def search_recursive(self, data):
        return self.r_search(data, self.root)
```

Insert - recursive version

Similar to search, we must write a separate private recursive insert() function that is called from the public insert function. Similar to search(), the recursive insert function also requires a pointer to the node we are trying to insert the data into. We think about this as inserting data into subtree. In python, there isn't really a pass by reference concept. As such, we will need to utilize the return statement in order to correctly attach our new tree.

- Python
- C++

```
def insert(self, data, subtree):
```

As with search, we want to start by figuring the base and recursive cases. For the base case, if the tree was empty we will simply make the current node the root of the tree. If not, we will want to insert the value either into the left or right subtree depending on how it compares to the data in the root of the subtree.

- Python
- C++

```
class BST:
    class Node:
        # Node's init function
        def __init__(self,data=None,left=None,right=None):
            self.data = data
            self.left = left
            self.right = right

    #BST's init function
    def __init__(self):
        self.root = None

    def ins(self, data,  subtree):
        if(subtree==None):
            return BST.Node(data)
        elif(data < subtree.data):
            subtree.left = self.ins(data,subtree.left)
            return subtree
        else:
            subtree.right = self.ins(data, subtree.right)
            return subtree

    def recursive_insert(self,data):
        self.root = self.ins(data,self.root)
```

InOrder Print function

To print all values in the tree from smallest to biggest, we can write the function recursively. This is an example of an inorder tree traversal. That is we will visit every node in the tree exactly one time and process it exactly one time.

This function is most easily written recursively. As with all other recursive function for bst we will pass in the root of the subtree. This function will be called from the public inOrder print() function.

The base case for this function is simply that of an empty tree. If the tree is empty, there is nothing to print so we do nothing and exit function.

if tree isn't empty we want to start by printing all values smaller than the current node (stored in left subtree), then the current node then all the values bigger than the current node (stored in right subtree). if its a tree, we'll use the inOrderPrint() function to print those so that they will be printed in order also.

```
class BST:
    class Node:
        # Node's init function
        def __init__(self,data=None,left=None,right=None):
            self.data = data
            self.left = left
            self.right = right

    #BST's init function
    def __init__(self):
        self.root = None


    def print_inorder(self, subtree):
        if(subtree != None):
            self.print_inorder(subtree.left)
            print(subtree.data, end = " ")
            self.print_inorder(subtree.right)

    def print(self):
        self.print_inorder(self.root)
        print("")
```

Pre Order Print

The pre-order print function is similar to the inorder print function except that we print the current node before printing its subtrees. This is the ordering that would be used if we were to do something like listing the contents of a file system.

The code for this is pretty much identical to inOrderPrint(). The only difference is in when we print the current node. What we want to do is print the current node, then its left and right subtrees

- Python
- C++

```python
class BST:
    class Node:
        # Node's init function
        def __init__(self,data=None,left=None,right=None):
            self.data = data
            self.left = left
            self.right = right

    #BST's init function
    def __init__(self):
        self.root = None


    def print_preorder(self, subtree):
        if(subtree != None):
            print(subtree.data, end = " ")
            self.print_preorder(subtree.left)
            self.print_preorder(subtree.right)

    def print(self):
        self.print_preorder(self.root)
        print("")
```