Trees and Tree Algorithms

1. The Tree Abstract Data Type

2. Tree Traversals

3. Priority Queue and Binary Heap

4. Binary Search Trees

5. AVL tree

6.2 Introduction

Trees are used in many areas of computer science, including operating systems, graphics, database systems, and computer networking. A tree data structure has a root, branches, and leaves. The difference between a tree in nature and a tree in computer science is that a tree data structure has its root at the top and its leaves on the bottom!

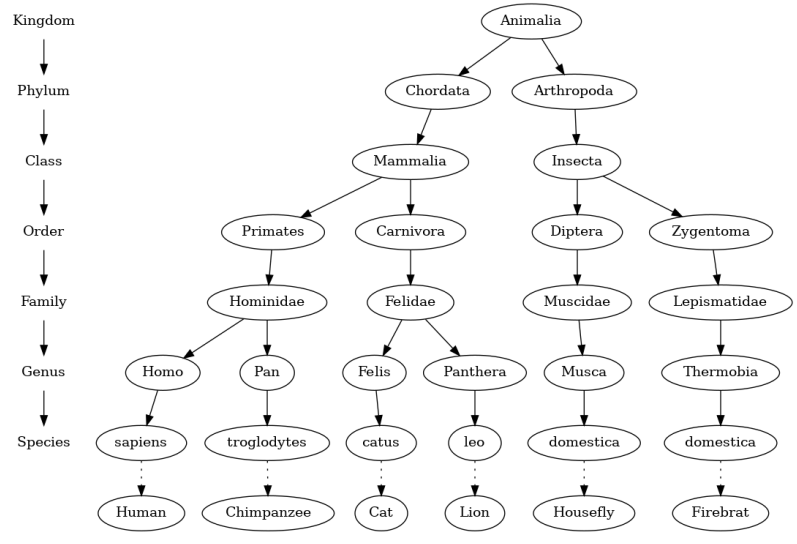
Trees are used in many areas of computer science, including operating systems, graphics, database systems, and computer networking. A tree data structure has a root, branches, and leaves. The difference between a tree in nature and a tree in computer science is that a tree data structure has its root at the top and its leaves on the bottom!

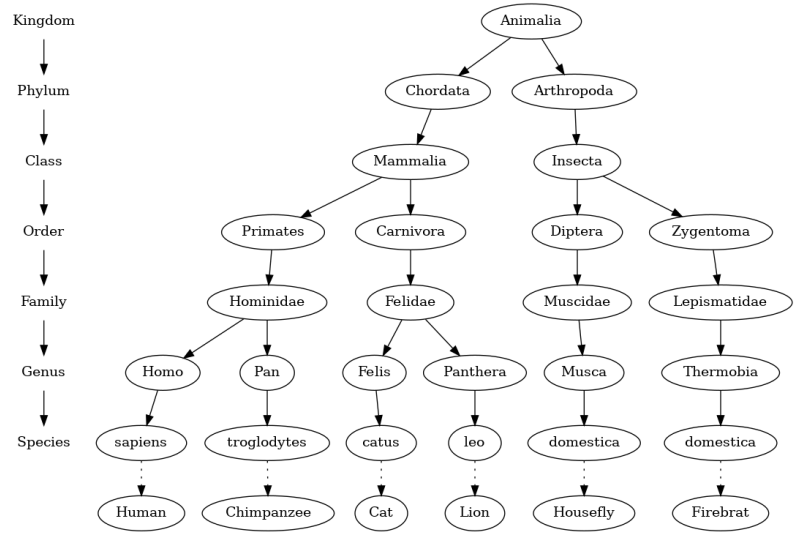
Our first example of a tree is a classification tree from biology which shows an example of the biological classification of some animals.

Trees are used in many areas of computer science, including operating systems, graphics, database systems, and computer networking. A tree data structure has a root, branches, and leaves. The difference between a tree in nature and a tree in computer science is that a tree data structure has its root at the top and its leaves on the bottom!

Our first example of a tree is a classification tree from biology which shows an example of the biological classification of some animals.

This example demonstrates that trees are **hierarchical**. By hierarchical, we mean that trees are structured in layers with the more general things near the top and the more specific things near the bottom. The top of the hierarchy is the kingdom, the next layer of the tree (the "children" of the layer above) is the phylum, then the class, and so on.


Kingdom
Phylum
Class
Order
Family
Genus
Species
Animalia
Chordata
Arthropoda
Mammalia
Insecta
Primates
Carnivora
Diptera
Zygentoma
Hominidae
Felidae
Muscidae
Lepismatidae
Homo
Pan
Felis
Panthera
Musca
Thermobia
sapiens
troglodytes
catus
leo
domestica
domestica
Human
Chimpanzee
Cat
Lion
Housefly
Firebrat

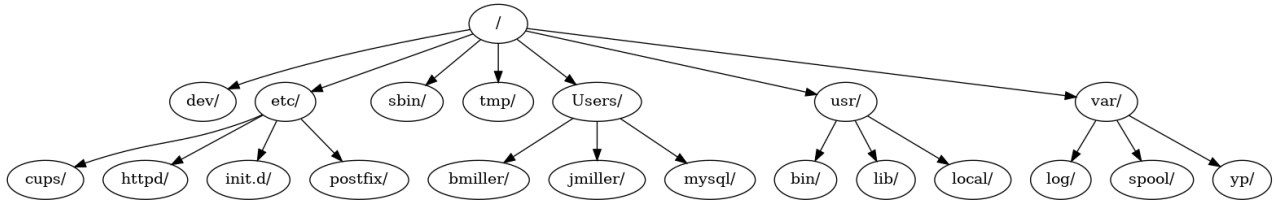


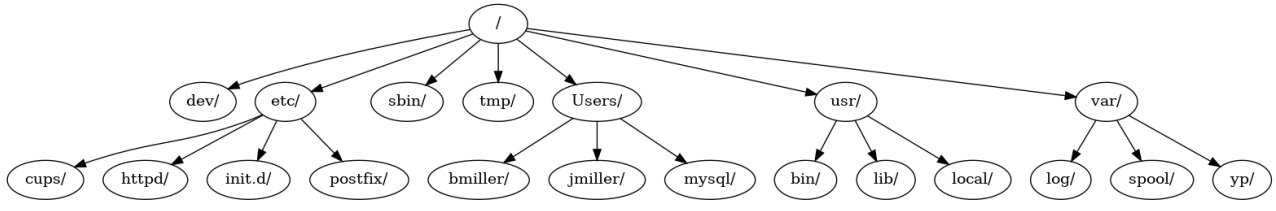
Notice that you can start at the top of the tree and follow a path made of circles and arrows all the way to the bottom. At each level of the tree we might ask ourselves a question and then follow the path that agrees with our answer.

A second property of trees is that all of the **children of one node are independent of the children of another node** while the third property is that **each leaf node is unique**.

A second property of trees is that all of the **children of one node are independent of the children of another node** while the third property is that **each leaf node is unique**.

Another example of a tree structure that you probably use every day is a file system. In a file system, directories, or folders, are structured as a tree. The file system tree enables you to follow a path from the root to any directory. That path will uniquely identify that subdirectory


/
dev/
etc/
sbin/
tmp/
Users/
usr/
var/
cups/
httpd/
init.d/
postfix/
bmiller/
jmiller/
mysql/
bin/
lib/
local/
log/
spool/
yp/



Note that we could take the entire subtree staring with `/etc/`, detach `etc/` from the root and reattach it under `usr/`. This would change the unique pathname to httpd from `/etc/httpd` to `/usr/etc/httpd`, but would not affect the contents or any children of the httpd directory!

6.3. Vocabulary and Definitions

- Node: A node is a fundamental part of a tree. It can have a name, which we call the **key**. A node may also have additional information. We call this additional information the value or payload.

- Node: A node is a fundamental part of a tree. It can have a name, which we call the **key**. A node may also have additional information. We call this additional information the value or payload.
- Edge: An edge is another fundamental part of a tree. An edge connects two nodes to show that there is a relationship between them. Every node (except the root) is connected by **exactly one incoming edge from another node**. Each node may have several outgoing edges.

- Node: A node is a fundamental part of a tree. It can have a name, which we call the **key**. A node may also have additional information. We call this additional information the value or payload.

- Edge: An edge is another fundamental part of a tree. An edge connects two nodes to show that there is a relationship between them. Every node (except the root) is connected by **exactly one incoming edge from another node**. Each node may have several outgoing edges.

- Root: The root of the tree is the only node in the tree that has no incoming edges.

- Path: A path is an ordered list of nodes that are connected by edges, for example, Mammalia -> Carnivora -> Felidae -> Felis -> catus is a path.

- Children: The set of nodes that have incoming edges from the same node are said to be the children of that node.
- Parent: A node is the parent of all the nodes it connects to with outgoing edges.

- Children: The set of nodes that have incoming edges from the same node are said to be the children of that node.
- Parent: A node is the parent of all the nodes it connects to with outgoing edges.

- Sibling: Nodes in the tree that are children of the same parent are said to be siblings. The nodes `etc/` and `usr/` are siblings in the file system tree shown in Figure 2.
- Subtree: A subtree is a set of nodes and edges comprised of a parent and all the descendants of that parent.

- Children: The set of nodes that have incoming edges from the same node are said to be the children of that node.
- Parent: A node is the parent of all the nodes it connects to with outgoing edges.

- Sibling: Nodes in the tree that are children of the same parent are said to be siblings. The nodes `etc/` and `usr/` are siblings in the file system tree shown in Figure 2.
- Subtree: A subtree is a set of nodes and edges comprised of a parent and all the descendants of that parent.

- Leaf Node: A leaf node is a node that has no children.
- Level: The level of a node n is the number of edges on the path from the root node to n.
- Height: The height of a tree is equal to the maximum level of any node in the tree.

Definition One: A tree consists of a set of nodes and a set of edges that connect pairs of nodes. A tree has the following properties:

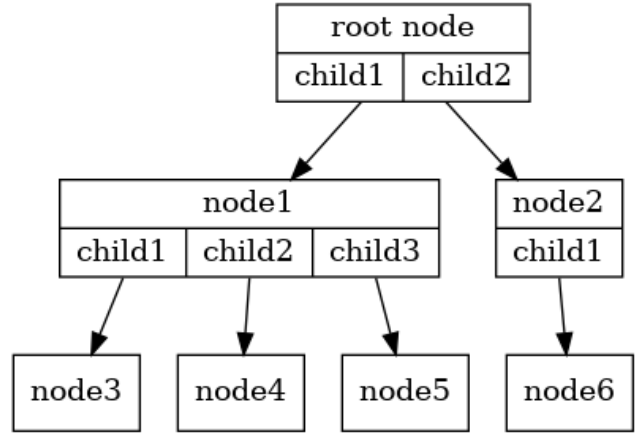
Definition One: A tree consists of a set of nodes and a set of edges that connect pairs of nodes. A tree has the following properties:

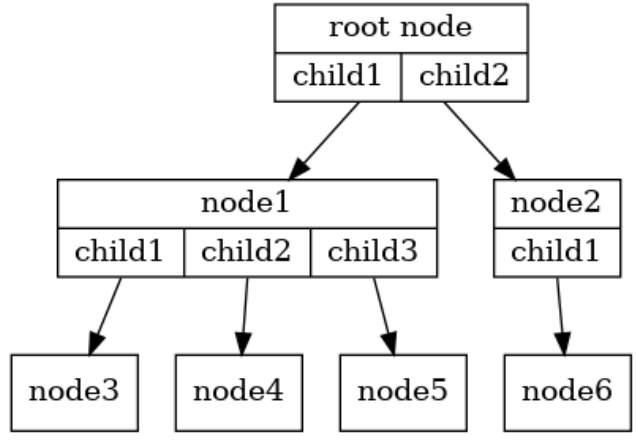
1. One node of the tree is designated as the root node.

2. Every node n, except the root node, is connected by an edge from exactly one other node p, where p is the parent of n.

Definition One: A tree consists of a set of nodes and a set of edges that connect pairs of nodes. A tree has the following properties:

1. One node of the tree is designated as the root node.
2. Every node n, except the root node, is connected by an edge from exactly one other node p, where p is the parent of n.
3. A unique path traverses from the root to each node.
4. If each node in the tree has a maximum of two children, we say that the tree is a binary tree.


root node
child1
child2
node1
child1
child2
child3
node2
child1
node3
node4
node5
node6



The arrowheads on the edges indicate the direction of the connection.

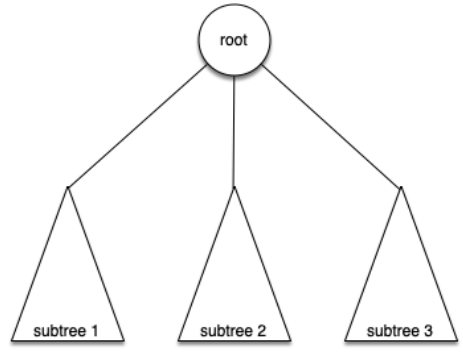
Definition Two: A tree is either empty or consists of a root and zero or more subtrees, each of which is also a tree. The root of each subtree is connected to the root of the parent tree by an edge.

Definition Two: A tree is either empty or consists of a root and zero or more subtrees, each of which is also a tree. The root of each subtree is connected to the root of the parent tree by an edge.

Figure below illustrates this recursive definition of a tree. Using the recursive definition of a tree, we know that the tree below has at least four nodes (if it is not empty) since each of the triangles representing a subtree must have a root. It may have many more nodes than that, but we do not know unless we look deeper into the tree!

Definition Two: A tree is either empty or consists of a root and zero or more subtrees, each of which is also a tree. The root of each subtree is connected to the root of the parent tree by an edge.

Figure below illustrates this recursive definition of a tree. Using the recursive definition of a tree, we know that the tree below has at least four nodes (if it is not empty) since each of the triangles representing a subtree must have a root. It may have many more nodes than that, but we do not know unless we look deeper into the tree!



6.4. Tree ADT

We can use the following functions to create and manipulate a binary tree:

We can use the following functions to create and manipulate a binary tree:

- `BinaryTree()`: creates a new instance of a binary tree.
- `get_root_val()`: returns the value stored in the current node.
- `set_root_val(val)`: stores the value in parameter `val` in the current node.

We can use the following functions to create and manipulate a binary tree:

- `BinaryTree()` : creates a new instance of a binary tree.
- `get_root_val()` : returns the value stored in the current node.
- `set_root_val(val)` : stores the value in parameter `val` in the current node.

- `get_left_child()` : returns the binary tree corresponding to the left child of the current node.
- `get_right_child()` : returns the binary tree corresponding to the right child of the current node.

- `insert_left(val)` : creates a new binary tree and installs it as the left child of the current node.
- `insert_right(val)` : creates a new binary tree and installs it as the right child of the current node.

- `insert_left(val)` : creates a new binary tree and installs it as the left child of the current node.
- `insert_right(val)` : creates a new binary tree and installs it as the right child of the current node.

The key decision in implementing a tree is choosing a good internal storage technique. We have two very interesting possibilities, and we will examine both before choosing one. We call them list of lists and nodes and references.

6.6. Nodes and References

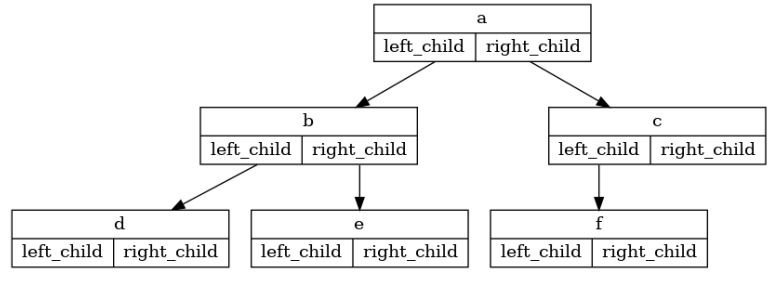
Our second method to represent a tree uses nodes and references. In this case we will define a class that has attributes for the root value as well as the left and right subtrees.

Our second method to represent a tree uses nodes and references. In this case we will define a class that has attributes for the root value as well as the left and right subtrees.

Using nodes and references, we might think of the tree as being structured like the one shown below:

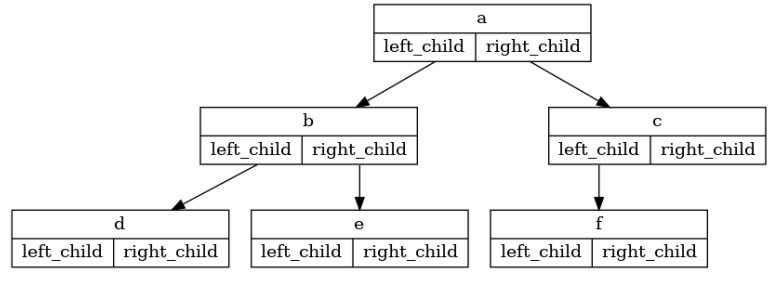
Our second method to represent a tree uses nodes and references. In this case we will define a class that has attributes for the root value as well as the left and right subtrees.

Using nodes and references, we might think of the tree as being structured like the one shown below:



Our second method to represent a tree uses nodes and references. In this case we will define a class that has attributes for the root value as well as the left and right subtrees.

Using nodes and references, we might think of the tree as being structured like the one shown below:



Since this representation more closely follows the object-oriented programming paradigm, we will continue to use this representation for the remainder of the chapter.

We will start out with a simple class definition for the nodes and references approach as shown below. The important thing to remember about this representation is that the attributes `left_child` and `right_child` will become references to other instances of the `BinaryTree` class. For example, when we insert a new left child into the tree, we create another instance of `BinaryTree` and modify `self.left_child` in the root to reference the new tree.

We will start out with a simple class definition for the nodes and references approach as shown below. The important thing to remember about this representation is that the attributes `left_child` and `right_child` will become references to other instances of the `BinaryTree` class. For example, when we insert a new left child into the tree, we create another instance of `BinaryTree` and modify `self.left_child` in the root to reference the new tree.

In [6]:
```
class BinaryTree:
    def __init__(self, root_obj):
        self.key = root_obj
        self.left_child = None
        self.right_child = None
```

We will start out with a simple class definition for the nodes and references approach as shown below. The important thing to remember about this representation is that the attributes `left_child` and `right_child` will become references to other instances of the `BinaryTree` class. For example, when we insert a new left child into the tree, we create another instance of `BinaryTree` and modify `self.left_child` in the root to reference the new tree.

In [6]:
```
class BinaryTree:
    def __init__(self, root_obj):
        self.key = root_obj
        self.left_child = None
        self.right_child = None
```

Just as you can store any object you like in a list, the root object of a tree can be a reference to any object. For our early examples, we will store the name of the node as the root value. Using nodes and references to represent the tree above, we would create six instances of the `BinaryTree` class.

Next let's look at the functions we need to build the tree beyond the root node. To add a left child to the tree, we will create a new binary tree object and set the `left_child` attribute of the root to refer to this new object.

Next let's look at the functions we need to build the tree beyond the root node. To add a left child to the tree, we will create a new binary tree object and set the `left_child` attribute of the root to refer to this new object.

In [7]:
```
def insert_left(self, new_node):
    if self.left_child is None:
        self.left_child = BinaryTree(new_node)
    else:
        new_child = BinaryTree(new_node)
        new_child.left_child = self.left_child
        self.left_child = new_child
```

Next let's look at the functions we need to build the tree beyond the root node. To add a left child to the tree, we will create a new binary tree object and set the `left_child` attribute of the root to refer to this new object.

In [7]:
```
def insert_left(self, new_node):
    if self.left_child is None:
        self.left_child = BinaryTree(new_node)
    else:
        new_child = BinaryTree(new_node)
        new_child.left_child = self.left_child
        self.left_child = new_child
```

Note that we consider two cases for insertion. The first case is characterized by a node with no existing left child. When there is no left child, simply add a node to the tree. The second case is characterized by a node with an existing left child. In the second case, we insert a node and push the existing child down one level in the tree.

In [8]:
```
def insert_right(self, new_node):
    if self.right_child == None:
        self.right_child = BinaryTree(new_node)
    else:
        new_child = BinaryTree(new_node)
        new_child.right_child = self.right_child
        self.right_child = new_child
```

In [8]:
```
def insert_right(self, new_node):
    if self.right_child == None:
        self.right_child = BinaryTree(new_node)
    else:
        new_child = BinaryTree(new_node)
        new_child.right_child = self.right_child
        self.right_child = new_child
```

To round out the definition for a simple binary tree data structure, we will write accessor methods for the left and right children and for the root values:

In [8]:
```
def insert_right(self, new_node):
    if self.right_child == None:
        self.right_child = BinaryTree(new_node)
    else:
        new_child = BinaryTree(new_node)
        new_child.right_child = self.right_child
        self.right_child = new_child
```

To round out the definition for a simple binary tree data structure, we will write accessor methods for the left and right children and for the root values:

In [9]:
```
def get_root_val(self):
    return self.key
def set_root_val(self, new_key):
    self.key = new_key

def get_left_child(self):
    return self.left_child
def get_right_child(self):
    return self.right_child
```

Now that we have all the pieces to create and manipulate a binary tree, let's use them to check on the structure a bit more. Let's make a simple tree with node a as the root, and add nodes "b" and "c" as children

Now that we have all the pieces to create and manipulate a binary tree, let's use them to check on the structure a bit more. Let's make a simple tree with node a as the root, and add nodes "b" and "c" as children

In [10]:
```
import sys
sys.path.append("./pythonds3/")
from pythonds3.trees import BinaryTree
```

Now that we have all the pieces to create and manipulate a binary tree, let's use them to check on the structure a bit more. Let's make a simple tree with node a as the root, and add nodes "b" and "c" as children

In [10]:
```
import sys
sys.path.append("./pythonds3/")
from pythonds3.trees import BinaryTree
```

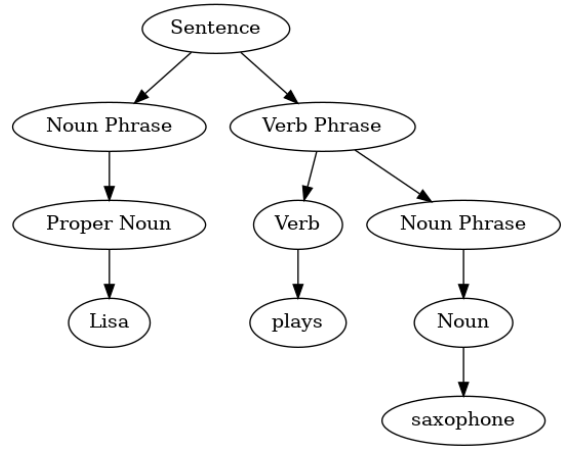
In [11]:
```
a_tree = BinaryTree("a")
print(a_tree.get_root_val())
print(a_tree.get_left_child())
a_tree.insert_left("b")
print(a_tree.get_left_child())
print(a_tree.get_left_child().get_root_val())
a_tree.insert_right("c")
print(a_tree.get_right_child())
print(a_tree.get_right_child().get_root_val())
a_tree.get_right_child().set_root_val("hello")
print(a_tree.get_right_child().get_root_val())
```

```
a
None
<pythonds3.trees.binary_tree.BinaryTree object at 0x0000027F9965DE80>
b
<pythonds3.trees.binary_tree.BinaryTree object at 0x0000027F9965DEE0>
c
hello
```

6.7. Parse Tree

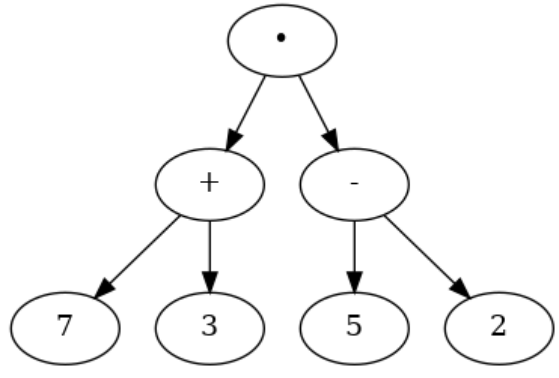
Parse trees can be used to represent real-world constructions like sentences or mathematical expressions. Figure below shows the hierarchical structure of a simple sentence. Representing a sentence as a tree structure allows us to work with the individual parts of the sentence by using subtrees.

Parse trees can be used to represent real-world constructions like sentences or mathematical expressions. Figure below shows the hierarchical structure of a simple sentence. Representing a sentence as a tree structure allows us to work with the individual parts of the sentence by using subtrees.

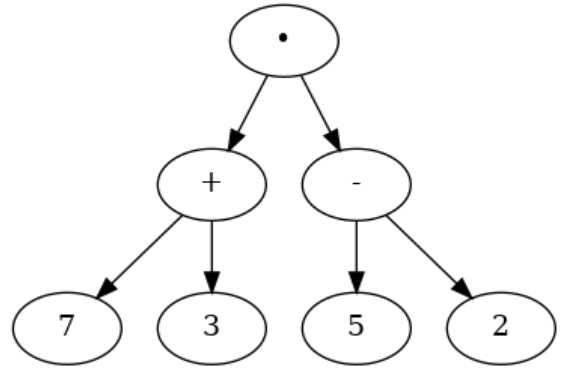


We can also represent a mathematical expression such as $((7+3)\cdot(5-2))$ as a parse tree, as shown below:

We can also represent a mathematical expression such as $((7+3)\cdot(5-2))$ as a parse tree, as shown below:



We can also represent a mathematical expression such as $((7 + 3) \cdot (5 - 2))$ as a parse tree, as shown below:



We know that multiplication has a higher precedence than either addition or subtraction. Because of the parentheses, we know that before we can do the multiplication we must evaluate the parenthesized addition and subtraction expressions. The hierarchy of the tree helps us understand the order of evaluation for the whole expression!

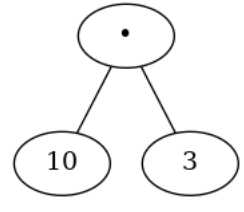
Before we can evaluate the top-level multiplication, we must evaluate the addition and the subtraction in the subtrees. The addition, which is the left subtree, evaluates to 10. The subtraction, which is the right subtree, evaluates to 3.

Before we can evaluate the top-level multiplication, we must evaluate the addition and the subtraction in the subtrees. The addition, which is the left subtree, evaluates to 10. The subtraction, which is the right subtree, evaluates to 3.

Using the hierarchical structure of trees, we can simply replace an entire subtree with one node once we have evaluated the expressions in the children. Applying this replacement procedure gives us the simplified tree shown below:

Before we can evaluate the top-level multiplication, we must evaluate the addition and the subtraction in the subtrees. The addition, which is the left subtree, evaluates to 10. The subtraction, which is the right subtree, evaluates to 3.

Using the hierarchical structure of trees, we can simply replace an entire subtree with one node once we have evaluated the expressions in the children. Applying this replacement procedure gives us the simplified tree shown below:



In the rest of this section we are going to examine parse trees in more detail. In particular we will look at

- How to build a parse tree from a fully parenthesized mathematical expression.

In the rest of this section we are going to examine parse trees in more detail. In particular we will look at

- How to build a parse tree from a fully parenthesized mathematical expression.
- How to evaluate the expression stored in a parse tree.

In the rest of this section we are going to examine parse trees in more detail. In particular we will look at

- How to build a parse tree from a fully parenthesized mathematical expression.
- How to evaluate the expression stored in a parse tree.
- How to recover the original mathematical expression from a parse tree.

In the rest of this section we are going to examine parse trees in more detail. In particular we will look at

- How to build a parse tree from a fully parenthesized mathematical expression.
- How to evaluate the expression stored in a parse tree.
- How to recover the original mathematical expression from a parse tree.

The first step in building a parse tree is to break up the expression string into a list of tokens. There are four different kinds of tokens to consider: **left parentheses, right parentheses, operators, and operands.**

We know that whenever we read a left parenthesis we are starting a new expression, and hence we should create a new tree to correspond to that expression. Conversely, whenever we read a right parenthesis, we have finished an expression.

We know that whenever we read a left parenthesis we are starting a new expression, and hence we should create a new tree to correspond to that expression. Conversely, whenever we read a right parenthesis, we have finished an expression.

We also know that operands are going to be leaf nodes and children of their operators. Finally, we know that every operator is going to have both a left and a right child. Using the information from above we can define four rules as follows:

We know that whenever we read a left parenthesis we are starting a new expression, and hence we should create a new tree to correspond to that expression. Conversely, whenever we read a right parenthesis, we have finished an expression.

We also know that operands are going to be leaf nodes and children of their operators. Finally, we know that every operator is going to have both a left and a right child. Using the information from above we can define four rules as follows:

1. If the current token is a `(`, add a new node as the left child of the current node, and descend to the left child.

2. If the current token is in the list `["+", "-", "/", "*"]`, set the root value of the current node to the operator represented by the current token. Add a new node as the right child of the current node and descend to the right child.

2. If the current token is in the list `["+", "-", "/", "*"]`, set the root value of the current node to the operator represented by the current token. Add a new node as the right child of the current node and descend to the right child.

3. If the current token is a number, set the root value of the current node to the number and return to the parent.

2. If the current token is in the list `["+", "-", "/", "*"]`, set the root value of the current node to the operator represented by the current token. Add a new node as the right child of the current node and descend to the right child.

3. If the current token is a number, set the root value of the current node to the number and return to the parent.

4. If the current token is a `)`, go to the parent of the current node.

2. If the current token is in the list `["+", "-", "/", "*"]`, set the root value of the current node to the operator represented by the current token. Add a new node as the right child of the current node and descend to the right child.

3. If the current token is a number, set the root value of the current node to the number and return to the parent.

4. If the current token is a `)`, go to the parent of the current node.

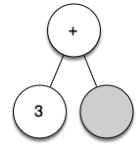
Let's look at an example of the rules outlined above in action. We will use the expression $(3 + (4 * 5))$. We will parse this expression into the following list of character tokens: `["(", "3", "+", "(", "4", "*", "5", ")", ")"]`. Initially we will start out with a parse tree that consists of an empty root node.

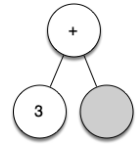
Figure below illustrates the structure and contents of the parse tree as each new token is processed:

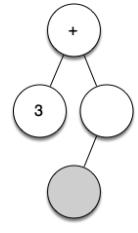
Figure below illustrates the structure and contents of the parse tree as each new token is processed:

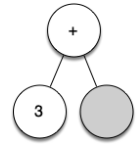
Figure below illustrates the structure and contents of the parse tree as each new token is processed:

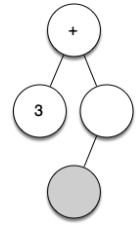
Figure below illustrates the structure and contents of the parse tree as each new token is processed:

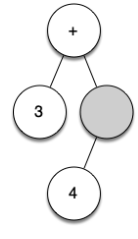

+
3

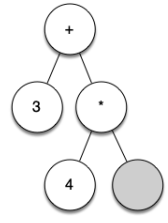

+
3

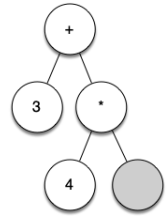

+
3

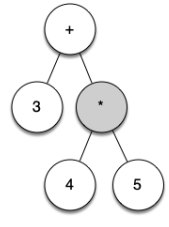

+
3


+
3


+
3
4


+
3
*
4


+
3
*
4


+
3
*
4
5

From the example above, it is clear that we need to keep track of the current node as well as the parent of the current node. The tree interface provides us with a way to get children of a node, through the `get_left_child()` and `get_right_child()` methods, but how can we keep track of the parent?

From the example above, it is clear that we need to keep track of the current node as well as the parent of the current node. The tree interface provides us with a way to get children of a node, through the `get_left_child()` and `get_right_child()` methods, but how can we keep track of the parent?

A simple solution to keeping track of parents as we traverse the tree is to **use a stack**. Whenever we want to descend to a child of the current node, we first push the current node on the stack. When we want to return to the parent of the current node, we pop the parent off the stack!

In [12]:

```
from pythonds3.basic import Stack
from pythonds3.trees import BinaryTree

def build_parse_tree(fp_expr):
    fp_list = fp_expr.split()
    p_stack = Stack()
    expr_tree = BinaryTree("")
    p_stack.push(expr_tree)
    current_tree = expr_tree

    for i in fp_list:
        if i == "(":
            current_tree.insert_left("")
            p_stack.push(current_tree)
            current_tree = current_tree.left_child
        elif i in ["+", "-", "*", "/"]:
            current_tree.root = i
            current_tree.insert_right("")
            p_stack.push(current_tree)
            current_tree = current_tree.right_child
        elif i.isdigit():
              current_tree.root = int(i)
              parent = p_stack.pop()
              current_tree = parent
        elif i == ")":
              current_tree = p_stack.pop()
        else:
              raise ValueError(f"Unknown operator '{i}'")
    return expr_tree
```

The four rules for building a parse tree are coded as the first four clauses of the `if..elif` statements. In each case you can see that the code implements the rule, as described above, with a few calls to the `BinaryTree` or `Stack` methods. The only error checking we do in this function is in the `else` clause where a `ValueError` exception will be raised if **we get a token from the list that we do not recognize.**

The four rules for building a parse tree are coded as the first four clauses of the `if..elif` statements. In each case you can see that the code implements the rule, as described above, with a few calls to the `BinaryTree` or `Stack` methods. The only error checking we do in this function is in the `else` clause where a `ValueError` exception will be raised if **we get a token from the list that we do not recognize.**

In [13]:
```
pt = build_parse_tree("( 3 + ( 4 * 5 ) )")
pt.inorder()  # defined and explained in the next section
```

```
3 + 4 * 5
```

The four rules for building a parse tree are coded as the first four clauses of the `if..elif` statements. In each case you can see that the code implements the rule, as described above, with a few calls to the `BinaryTree` or `Stack` methods. The only error checking we do in this function is in the `else` clause where a `ValueError` exception will be raised if **we get a token from the list that we do not recognize.**

In [13]:
```
pt = build_parse_tree("( 3 + ( 4 * 5 ) )")
pt.inorder()  # defined and explained in the next section
```

```
3 + 4 * 5
```

Now that we have built a parse tree, what can we do with it? As a first example, we will write a function to evaluate the parse tree and return the numerical result. To write this function, we will make use of the **hierarchical nature** of the tree.

Recall that we can replace the original tree with the simplified tree shown in above Figure. This suggests that we can write an algorithm that evaluates a parse tree by **recursively evaluating each subtree**.

Recall that we can replace the original tree with the simplified tree shown in above Figure. This suggests that we can write an algorithm that evaluates a parse tree by **recursively evaluating each subtree**.

A natural base case for recursive algorithms that operate on trees is to check for a leaf node. In a parse tree, the leaf nodes will always be operands. Since numerical objects like integers and floating points require no further interpretation, the evaluate function can simply return the value stored in the leaf node.

Recall that we can replace the original tree with the simplified tree shown in above Figure. This suggests that we can write an algorithm that evaluates a parse tree by **recursively evaluating each subtree**.

A natural base case for recursive algorithms that operate on trees is to check for a leaf node. In a parse tree, the leaf nodes will always be operands. Since numerical objects like integers and floating points require no further interpretation, the evaluate function can simply return the value stored in the leaf node.

The recursive step that moves the function toward the base case is to call evaluate on both the left and the right children of the current node. The recursive call effectively moves us down the tree, toward a leaf node.

To put the results of the two recursive calls together, we can simply apply the operator stored in the parent node to the results returned from evaluating both children. The code for a recursive evaluate function is shown below:

To put the results of the two recursive calls together, we can simply apply the operator stored in the parent node to the results returned from evaluating both children. The code for a recursive evaluate function is shown below:

In [14]:
```
import operator

def evaluate(parse_tree):
    operators = {
        "+": operator.add,
        "-": operator.sub,
        "*": operator.mul,
        "/": operator.truediv,
    }

    left_child = parse_tree.left_child
    right_child = parse_tree.right_child

    if left_child and right_child:
        fn = operators[parse_tree.root]
        return fn(evaluate(left_child), evaluate(right_child))
    else:
        return parse_tree.root
```

To implement the arithmetic, we use a dictionary with the keys `+`, `-`, `*`, and `/`. **The values stored in the dictionary are functions from Python's operator module.** The operator module provides us with the function versions of many commonly used operators. When we look up an operator in the dictionary, the corresponding function object is retrieved. Since the retrieved object is a function, we can call it in the usual way: `function(param1, param2)`. So the lookup operators `["+"](2, 2)` is equivalent to `operator.add(2, 2)`.

To implement the arithmetic, we use a dictionary with the keys `+`, `-`, `*`, and `/`. **The values stored in the dictionary are functions from Python's operator module.** The operator module provides us with the function versions of many commonly used operators. When we look up an operator in the dictionary, the corresponding function object is retrieved. Since the retrieved object is a function, we can call it in the usual way: `function(param1, param2)`. So the lookup operators `["+"](2, 2)` is equivalent to `operator.add(2, 2)`.

Finally, we will trace the evaluate function on the parse tree we created before. When we first call `evaluate()`, we pass the root of the entire tree as the parameter `parse_tree()`. Then we obtain references to the left and right children to make sure they exist. The recursive call takes place on line 16.

To implement the arithmetic, we use a dictionary with the keys `+`, `-`, `*`, and `/`. **The values stored in the dictionary are functions from Python's operator module.** The operator module provides us with the function versions of many commonly used operators. When we look up an operator in the dictionary, the corresponding function object is retrieved. Since the retrieved object is a function, we can call it in the usual way: `function(param1, param2)`. So the lookup operators `["+"](2, 2)` is equivalent to `operator.add(2, 2)`.

Finally, we will trace the evaluate function on the parse tree we created before. When we first call `evaluate()`, we pass the root of the entire tree as the parameter `parse_tree()`. Then we obtain references to the left and right children to make sure they exist. The recursive call takes place on line 16.

In [15]:
```
evaluate(pt)
```

Out[15]:
```
23
```

6.8. Tree Traversals

Now it is time to look at some additional usage patterns for trees. These usage patterns can be divided into three commonly used patterns to **visit all the nodes in a tree.**

Now it is time to look at some additional usage patterns for trees. These usage patterns can be divided into three commonly used patterns to **visit all the nodes in a tree.**

The difference between these patterns is the order in which each node is visited. We call this visitation of the nodes a tree traversal. The three traversals we will look at are called preorder, inorder, and postorder. Let's start out by defining these three traversals more carefully, then look at some examples where these patterns are useful.

Now it is time to look at some additional usage patterns for trees. These usage patterns can be divided into three commonly used patterns to **visit all the nodes in a tree.**

The difference between these patterns is the order in which each node is visited. We call this visitation of the nodes a tree traversal. The three traversals we will look at are called preorder, inorder, and postorder. Let's start out by defining these three traversals more carefully, then look at some examples where these patterns are useful.

- Preorder: In a preorder traversal, we visit the root node first, then recursively do a preorder traversal of the left subtree, followed by a recursive preorder traversal of the right subtree.

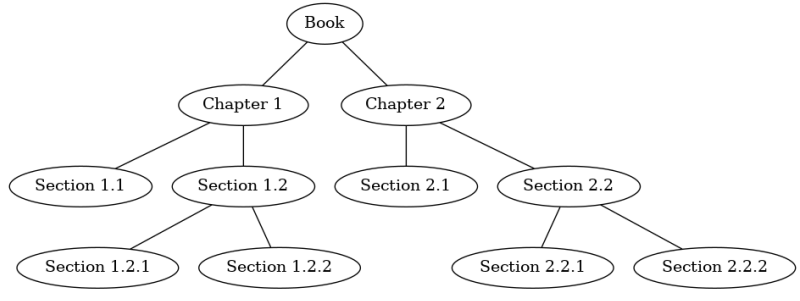
- Inorder: In an inorder traversal, we recursively do an inorder traversal on the left subtree, visit the root node, and finally do a recursive inorder traversal of the right subtree.

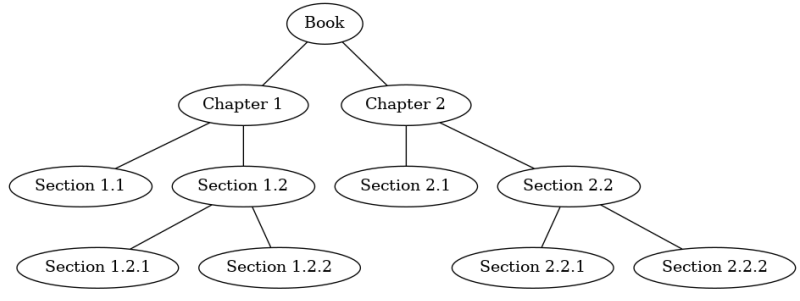
- Inorder: In an inorder traversal, we recursively do an inorder traversal on the left subtree, visit the root node, and finally do a recursive inorder traversal of the right subtree.
- Postorder: In a postorder traversal, we recursively do a postorder traversal of the left subtree and the right subtree followed by a visit to the root node.

- Inorder: In an inorder traversal, we recursively do an inorder traversal on the left subtree, visit the root node, and finally do a recursive inorder traversal of the right subtree.

- Postorder: In a postorder traversal, we recursively do a postorder traversal of the left subtree and the right subtree followed by a visit to the root node.

First let's look at the preorder traversal using a book as an example tree. The book is the root of the tree, and each chapter is a child of the root. Each section within a chapter is a child of the chapter, each subsection is a child of its section, and so on.


Book
Chapter 1
Chapter 2
Section 1.1
Section 1.2
Section 2.1
Section 2.2
Section 1.2.1
Section 1.2.2
Section 2.2.1
Section 2.2.2



Suppose that you wanted to read this book from front to back. The preorder traversal gives you exactly that ordering.

The code for writing tree traversals is surprisingly elegant, largely because the traversals are written recursively. The code belwo shows a version of the preorder traversal written as an external function that takes a binary tree as a parameter. The external function is particularly elegant because our base case is simply to check if the tree exists. If the tree parameter is `None`, then the function returns without taking any action.

The code for writing tree traversals is surprisingly elegant, largely because the traversals are written recursively. The code belwo shows a version of the preorder traversal written as an external function that takes a binary tree as a parameter. The external function is particularly elegant because our base case is simply to check if the tree exists. If the tree parameter is `None`, then the function returns without taking any action.

In [16]:
```
def preorder(tree):
    if tree:
        print(tree._key, end=" ")
        preorder(tree._left_child)
        preorder(tree._right_child)
```

The code for writing tree traversals is surprisingly elegant, largely because the traversals are written recursively. The code belwo shows a version of the preorder traversal written as an external function that takes a binary tree as a parameter. The external function is particularly elegant because our base case is simply to check if the tree exists. If the tree parameter is `None`, then the function returns without taking any action.

In [16]:
```
def preorder(tree):
    if tree:
        print(tree._key, end=" ")
        preorder(tree._left_child)
        preorder(tree._right_child)
```

In [17]:
```
preorder(pt)
```

```
+ 3 * 4 5
```

Implementing preorder as an external function is probably better in this case. The reason is that you very rarely want to just traverse the tree. In most cases you are going to want to accomplish something else while using one of the basic traversal patterns.

Implementing preorder as an external function is probably better in this case. The reason is that you very rarely want to just traverse the tree. In most cases you are going to want to accomplish something else while using one of the basic traversal patterns.

In fact, we will see in the next example that the postorder traversal pattern follows very closely with the code we wrote earlier to evaluate a parse tree. Therefore we will write the rest of the traversals as external functions.

Implementing preorder as an external function is probably better in this case. The reason is that you very rarely want to just traverse the tree. In most cases you are going to want to accomplish something else while using one of the basic traversal patterns.

In fact, we will see in the next example that the postorder traversal pattern follows very closely with the code we wrote earlier to evaluate a parse tree. Therefore we will write the rest of the traversals as external functions.

In [20]:
```
def postorder(tree):
    if tree:
        postorder(tree._left_child)
        postorder(tree._right_child)
        print(tree._key, end=" ")
```

Implementing preorder as an external function is probably better in this case. The reason is that you very rarely want to just traverse the tree. In most cases you are going to want to accomplish something else while using one of the basic traversal patterns.

In fact, we will see in the next example that the postorder traversal pattern follows very closely with the code we wrote earlier to evaluate a parse tree. Therefore we will write the rest of the traversals as external functions.

In [20]:
```
def postorder(tree):
    if tree:
        postorder(tree._left_child)
        postorder(tree._right_child)
        print(tree._key, end=" ")
```

In [21]:
```
postorder(pt)
```

```
3 4 5 * +
```

We have already seen a common use for the postorder traversal, namely **evaluating a parse tree**. Assuming our binary tree is going to store only expression tree data, rewrite the evaluation function, but model it even more closely on the postorder code, we have:

We have already seen a common use for the postorder traversal, namely **evaluating a parse tree**. Assuming our binary tree is going to store only expression tree data, rewrite the evaluation function, but model it even more closely on the postorder code, we have:

In [23]:
```
def postordereval(tree):
    operators = {
        "+": operator.add,
        "-": operator.sub,
        "*": operator.mul,
        "/": operator.truediv,
    }
    result_1 = None
    result_2 = None
    if tree:
        result_1 = postordereval(tree._left_child)
        result_2 = postordereval(tree._right_child)
        if result_1 and result_2:
            return operators[tree._key](result_1, result_2)
        return tree._key
```

We have already seen a common use for the postorder traversal, namely **evaluating a parse tree**. Assuming our binary tree is going to store only expression tree data, rewrite the evaluation function, but model it even more closely on the postorder code, we have:

In [23]:
```
def postordereval(tree):
    operators = {
        "+": operator.add,
        "-": operator.sub,
        "*": operator.mul,
        "/": operator.truediv,
    }
    result_1 = None
    result_2 = None
    if tree:
        result_1 = postordereval(tree._left_child)
        result_2 = postordereval(tree._right_child)
        if result_1 and result_2:
            return operators[tree._key](result_1, result_2)
        return tree._key
```

In [24]:
```
postordereval(pt)
```

Out[24]:
```
23
```

Note that except that instead of printing the key at the end of the function, we return it. This allows us to save the values returned from the recursive calls. We then use these saved values along with the operator.

Note that except that instead of printing the key at the end of the function, we return it. This allows us to save the values returned from the recursive calls. We then use these saved values along with the operator.

The final traversal we will look at in this section is the inorder traversal. In the inorder traversal we visit the left subtree, followed by the root, and finally the right subtree.

Note that except that instead of printing the key at the end of the function, we return it. This allows us to save the values returned from the recursive calls. We then use these saved values along with the operator.

The final traversal we will look at in this section is the inorder traversal. In the inorder traversal we visit the left subtree, followed by the root, and finally the right subtree.

In [25]:
```
def inorder(tree):
    if tree:
        inorder(tree._left_child)
        print(tree._key, end=" ")
        inorder(tree._right_child)
```

Note that except that instead of printing the key at the end of the function, we return it. This allows us to save the values returned from the recursive calls. We then use these saved values along with the operator.

The final traversal we will look at in this section is the inorder traversal. In the inorder traversal we visit the left subtree, followed by the root, and finally the right subtree.

In [25]:
```
def inorder(tree):
    if tree:
        inorder(tree._left_child)
        print(tree._key, end=" ")
        inorder(tree._right_child)
```

In [26]:
```
inorder(pt)
```

```
3 + 4 * 5
```

If we perform a simple inorder traversal of a parse tree, we get our original expression back without any parentheses. Let's modify the basic inorder algorithm to allow us to recover the fully parenthesized version of the expression. The only modifications we will make to the basic template are as follows.

If we perform a simple inorder traversal of a parse tree, we get our original expression back without any parentheses. Let's modify the basic inorder algorithm to allow us to recover the fully parenthesized version of the expression. The only modifications we will make to the basic template are as follows.

Print a left parenthesis before the recursive call to the left subtree, and print a right parenthesis after the recursive call to the right subtree:

If we perform a simple inorder traversal of a parse tree, we get our original expression back without any parentheses. Let's modify the basic inorder algorithm to allow us to recover the fully parenthesized version of the expression. The only modifications we will make to the basic template are as follows.

Print a left parenthesis before the recursive call to the left subtree, and print a right parenthesis after the recursive call to the right subtree:

In [28]:
```
def print_exp(tree):
    result = ""
    if tree:
        result = "(" + print_exp(tree._left_child)
        result = result + str(tree._key)
        result = result + print_exp(tree._right_child) + ")"
    return result
```

If we perform a simple inorder traversal of a parse tree, we get our original expression back without any parentheses. Let's modify the basic inorder algorithm to allow us to recover the fully parenthesized version of the expression. The only modifications we will make to the basic template are as follows.

Print a left parenthesis before the recursive call to the left subtree, and print a right parenthesis after the recursive call to the right subtree:

In [28]:
```
def print_exp(tree):
    result = ""
    if tree:
        result = "(" + print_exp(tree._left_child)
        result = result + str(tree._key)
        result = result + print_exp(tree._right_child) + ")"
    return result
```

In [29]:
```
print_exp(pt)
```

Out[29]:
```
'((3)+((4)*(5)))'
```

Exercise 1: Clean up the `print_exp()` function so that it does not include an extra set of parentheses around each number.

Exercise 1: Clean up the `print_exp()` function so that it does not include an extra set of parentheses around each number.

In [30]:
```
def print_exp(tree):
    result = ""
    if tree:
        result = "(" + print_exp(tree._left_child)
        result = result + str(tree._key)
        result = result + print_exp(tree._right_child) + ")"
    return result
```

Exercise 1: Clean up the `print_exp()` function so that it does not include an extra set of parentheses around each number.

In [30]:
```
def print_exp(tree):
    result = ""
    if tree:
        result = "(" + print_exp(tree._left_child)
        result = result + str(tree._key)
        result = result + print_exp(tree._right_child) + ")"
    return result
```

In [31]:
```
print_exp(pt)
```

Out[31]:
```
'((3)+((4)*(5)))'
```

6.9. Priority Queues with Binary Heaps

You can probably think of a couple of easy ways to implement a priority queue using sorting functions and lists. However, inserting into a list is $O(n)$ and sorting a list is $O(n \log n)$.

You can probably think of a couple of easy ways to implement a priority queue using sorting functions and lists. However, inserting into a list is $O(n)$ and sorting a list is $O(n \log n)$.

We can do better. The classic way to implement a priority queue is using a data structure called a binary heap. A binary heap will allow us both to enqueue and dequeue items in $O(\log n)$.

You can probably think of a couple of easy ways to implement a priority queue using sorting functions and lists. However, inserting into a list is $O(n)$ and sorting a list is $O(n \log n)$.

We can do better. The classic way to implement a priority queue is using a data structure called a binary heap. A binary heap will allow us both to enqueue and dequeue items in $O(\log n)$.

The binary heap is interesting to study because when we diagram the heap it looks a lot like a tree, but when we implement it we use only a single list as an internal representation. The binary heap has two common variations: the min heap, in which the smallest key value is always at the front, and the max heap, in which the largest key value is always at the front. In this section we will implement the min heap.

6.10. Binary Heap Operations

- `BinaryHeap()` : creates a new empty binary heap.
- `insert(k)` : adds a new item to the heap.
- `get_min()` : returns the item with the minimum key value, leaving the item in the heap.

- `BinaryHeap()` : creates a new empty binary heap.
- `insert(k)` : adds a new item to the heap.
- `get_min()` : returns the item with the minimum key value, leaving the item in the heap.

- `delete()` : returns the item with the minimum key value, removing the item from the heap.
- `is_empty()` : returns `True` if the heap is empty, `False` otherwise.
- `size()` : returns the number of items in the heap.
- `heapify(list)` : builds a new heap from a list of keys.

In [32]:
```
from pythonds3.trees import BinaryHeap

my_heap = BinaryHeap()
my_heap.insert(5)
my_heap.insert(7)
my_heap.insert(3)
my_heap.insert(11)

print(my_heap.delete())
print(my_heap.delete())
print(my_heap.delete())
print(my_heap.delete())
```

```
3
5
7
11
```

In [32]:
```
from pythonds3.trees import BinaryHeap

my_heap = BinaryHeap()
my_heap.insert(5)
my_heap.insert(7)
my_heap.insert(3)
my_heap.insert(11)

print(my_heap.delete())
print(my_heap.delete())
print(my_heap.delete())
print(my_heap.delete())
```

```
3
5
7
11
```

Notice that no matter what order we add items to the heap, the smallest is removed each time!

6.11. Binary Heap Implementation

In order to make our heap work efficiently, we will take advantage of the logarithmic nature of the binary tree to represent our heap.

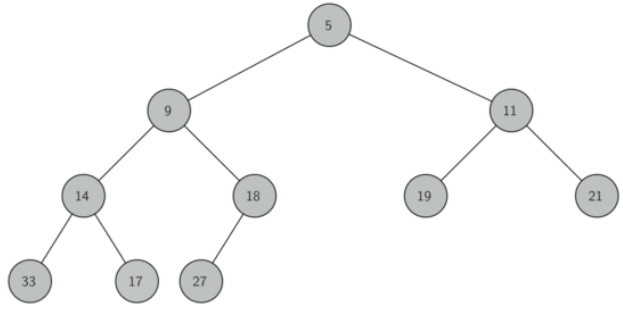
In order to make our heap work efficiently, we will take advantage of the logarithmic nature of the binary tree to represent our heap.

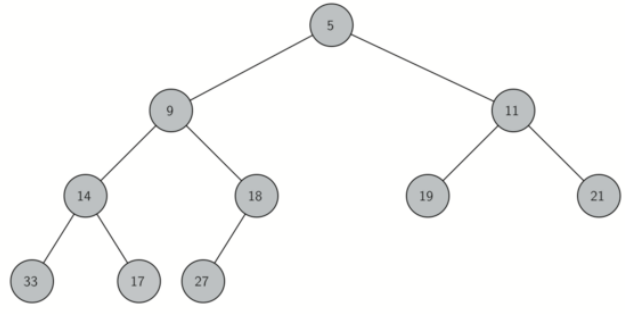
In order to guarantee logarithmic performance, we must keep our tree **balanced**. A balanced binary tree has roughly the same number of nodes in the left and right subtrees of the root. In our heap implementation we keep the tree balanced by creating a complete binary tree.

In order to make our heap work efficiently, we will take advantage of the logarithmic nature of the binary tree to represent our heap.

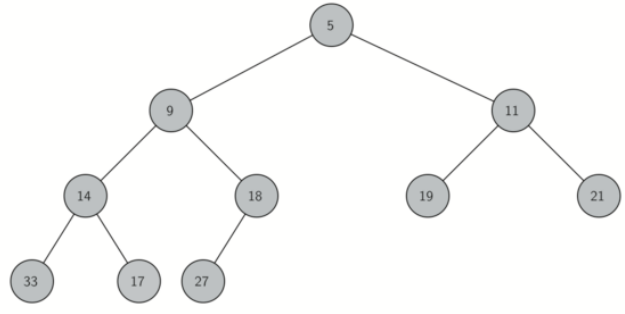
In order to guarantee logarithmic performance, we must keep our tree **balanced**. A balanced binary tree has roughly the same number of nodes in the left and right subtrees of the root. In our heap implementation we keep the tree balanced by creating a complete binary tree.

A complete binary tree is a tree in which each level has all of its nodes. The exception to this is the bottom level of the tree, which we fill in from left to right. Figure below shows an example of a complete binary tree.


5
9
11
14
18
19
21
33
17
27



Another interesting property of a complete tree is that we can represent it using a **single list**. We do not need to use nodes and references or even lists of lists. Because the tree is complete, the left child of a parent (at position p) is the node that is found in position $2p + 1$ in the list!



Another interesting property of a complete tree is that we can represent it using a **single list**. We do not need to use nodes and references or even lists of lists. Because the tree is complete, the left child of a parent (at position p) is the node that is found in position $2p + 1$ in the list!

Similarly, the right child of the parent is at position $2p + 2$ in the list. To find the parent of any node in the tree, we can simply use integer division. Given that a node is at position n in the list, the parent is at position $(n - 1)//2$.

Figure below shows a complete binary tree and also gives the list representation of the tree. Note the $2p + 1$ and $2p + 2$ relationship between parent and children.

Figure below shows a complete binary tree and also gives the list representation of the tree. Note the $2p + 1$ and $2p + 2$ relationship between parent and children.

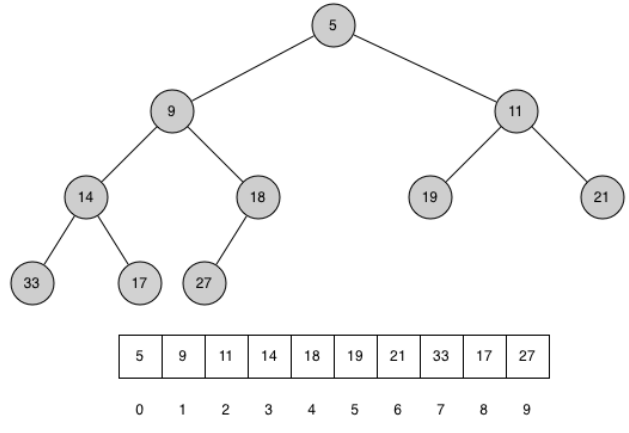
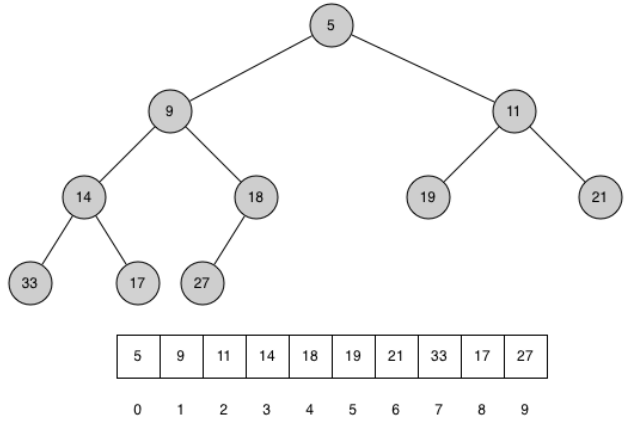


Figure below shows a complete binary tree and also gives the list representation of the tree. Note the $2p + 1$ and $2p + 2$ relationship between parent and children.



The list representation of the tree, along with the full structure property, allows us to efficiently traverse a complete binary tree using only a few simple mathematical operations.

The method that we will use to store items in a heap relies on maintaining the heap order property. The heap order property is as follows: in a heap, for every node x with parent p, the key in p is smaller than or equal to the key in x. Figure above also illustrates a complete binary tree that has the heap order property.

The method that we will use to store items in a heap relies on maintaining the heap order property. The heap order property is as follows: in a heap, for every node x with parent p, the key in p is smaller than or equal to the key in x. Figure above also illustrates a complete binary tree that has the heap order property.

We now begin our implementation of a binary heap with the constructor. Since the entire binary heap can be represented by a single list, all the constructor will do is initialize the list:

The method that we will use to store items in a heap relies on maintaining the heap order property. The heap order property is as follows: in a heap, for every node x with parent p, the key in p is smaller than or equal to the key in x. Figure above also illustrates a complete binary tree that has the heap order property.

We now begin our implementation of a binary heap with the constructor. Since the entire binary heap can be represented by a single list, all the constructor will do is initialize the list:

In [34]:
```
class BinaryHeap:
    def __init__(self):
        self._heap = []
```

The next method we will implement is `insert()`. The easiest, and most efficient, way to add an item to a list is to simply append the item to the end of the list. The good news about appending is that it guarantees that we will maintain the complete tree property. The bad news about appending is that we will very likely **violate the heap structure property.**

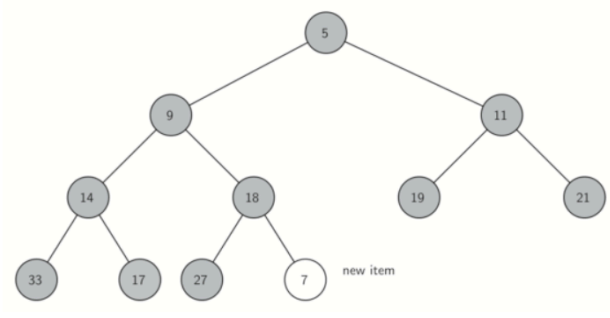
The next method we will implement is `insert()`. The easiest, and most efficient, way to add an item to a list is to simply append the item to the end of the list. The good news about appending is that it guarantees that we will maintain the complete tree property. The bad news about appending is that we will very likely **violate the heap structure property.**

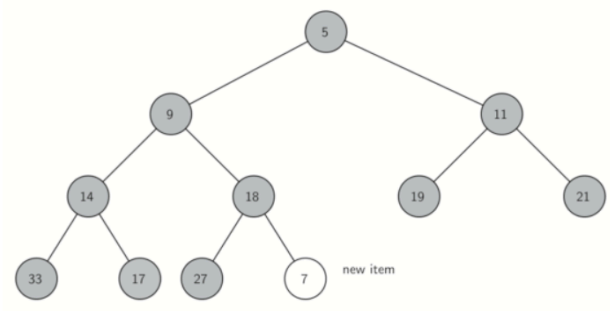
However, it is possible to write a method that will allow us to regain the heap structure property by comparing the newly added item with its parent. If the newly added item is less than its parent, then we can swap the item with its parent.

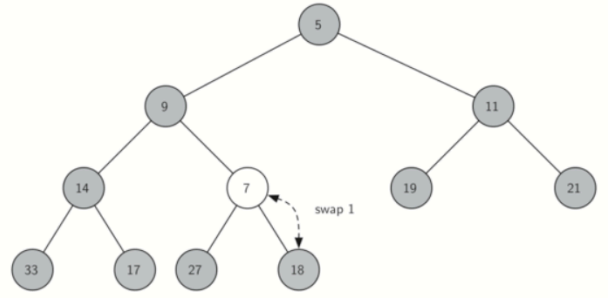
The next method we will implement is `insert()`. The easiest, and most efficient, way to add an item to a list is to simply append the item to the end of the list. The good news about appending is that it guarantees that we will maintain the complete tree property. The bad news about appending is that we will very likely **violate the heap structure property.**

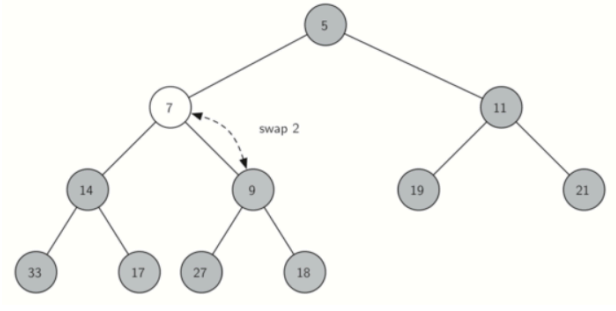
However, it is possible to write a method that will allow us to regain the heap structure property by comparing the newly added item with its parent. If the newly added item is less than its parent, then we can swap the item with its parent.

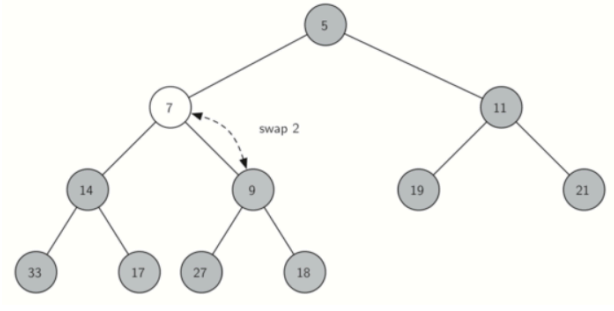
Figure below shows the series of swaps needed to percolate the newly added item up to its proper position in the tree.


5
9
11
14
18
19
21
33
17
27
7
new item


5
9
11
14
18
19
21
33
17
27
7
new item


5
9
11
14
7
19
21
swap 1
33
17
27
18


5
7
swap 2
11
14
9
19
21
33
17
27
18



Notice that when we percolate an item up, we are restoring the heap property between the newly added item and the parent. We are also preserving the heap property for any siblings. Of course, if the newly added item is very small, we may still need to swap it up another level. In fact, we may need to keep swapping until we get to the top of the tree!

In [35]:

```
def _perc_up(self, i):
    while (i - 1) // 2 >= 0:
        parent_idx = (i - 1) // 2
        if self._heap[i] < self._heap[parent_idx]:
            self._heap[i], self._heap[parent_idx] = (
                self._heap[parent_idx],
                self._heap[i],
            )
        i = parent_idx
```

In [35]:
```
def _perc_up(self, i):
    while (i - 1) // 2 >= 0:
        parent_idx = (i - 1) // 2
        if self._heap[i] < self._heap[parent_idx]:
            self._heap[i], self._heap[parent_idx] = (
                self._heap[parent_idx],
                self._heap[i],
            )
        i = parent_idx
```

We are now ready to write the `insert()` method. Most of the work in the `insert()` method is really done by `_perc_up()`. Once a new item is appended to the tree, `_perc_up()` takes over and positions the new item properly.

In [37]:
```
def insert(self, item):
    self._heap.append(item)
    self._perc_up(len(self._heap) - 1)
```

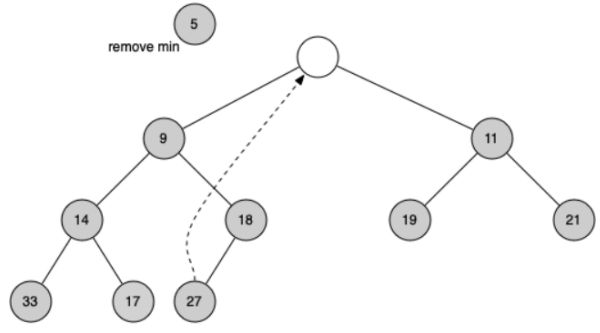
In [37]:
```
def insert(self, item):
    self._heap.append(item)
    self._perc_up(len(self._heap) - 1)
```

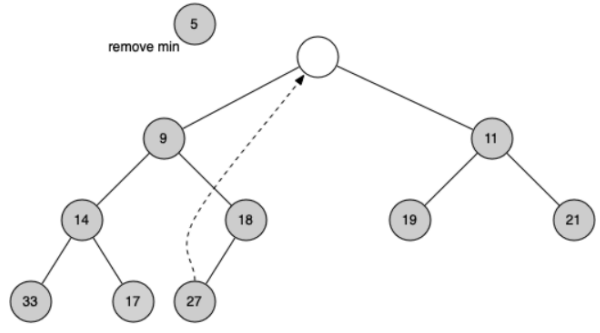
With the `insert()` method properly defined, we can now look at the `delete()` method. Since the heap property requires that the root of the tree be the smallest item in the tree, finding the minimum item is easy.

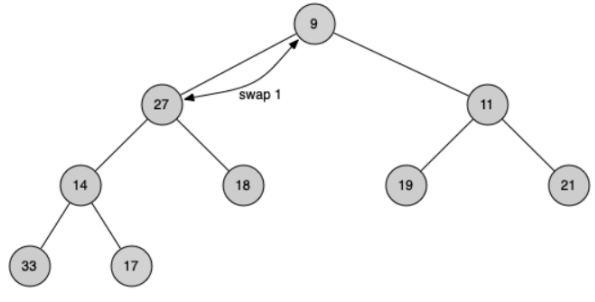
In [37]:
```
def insert(self, item):
    self._heap.append(item)
    self._perc_up(len(self._heap) - 1)
```

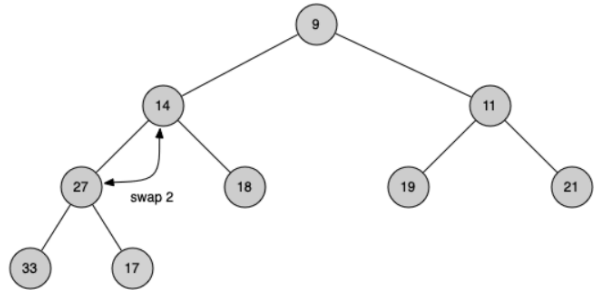
With the `insert()` method properly defined, we can now look at the `delete()` method. Since the heap property requires that the root of the tree be the smallest item in the tree, finding the minimum item is easy.

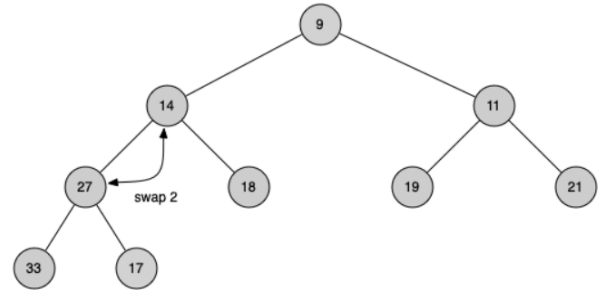
The hard part of delete is restoring full compliance with the heap structure and heap order properties after the root has been removed. We can restore our heap in two steps.

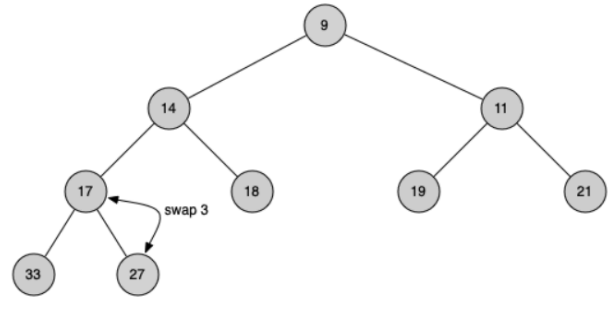

5
remove min
9
11
14
18
19
21
33
17
27


5
remove min
9
11
14
18
19
21
33
17
27


9
swap 1
27
11
14
18
19
21
33
17


9
14
11
27
swap 2
18
19
21
33
17


9
14
11
27
swap 2
18
19
21
33
17


9
14
11
17
swap 3
18
19
21
33
27

In order to maintain the heap order property, all we need to do is swap the root with its smaller child that is less than the root. After the initial swap, we may repeat the swapping process with a node and its children until the node is swapped into a position on the tree where it is already less than both children.

In order to maintain the heap order property, all we need to do is swap the root with its smaller child that is less than the root. After the initial swap, we may repeat the swapping process with a node and its children until the node is swapped into a position on the tree where it is already less than both children.

In [38]:
```
def _perc_down(self, i):
    while 2 * i + 1 < len(self._heap):
        sm_child = self._get_min_child(i)
        if self._heap[i] > self._heap[sm_child]:
            self._heap[i], self._heap[sm_child] = (
                self._heap[sm_child],
                self._heap[i],
            )
        else:
            break
        i = sm_child

def _get_min_child(self, i):
    if 2 * i + 2 > len(self._heap) - 1:
        return 2 * i + 1
    if self._heap[2 * i + 1] < self._heap[2 * i + 2]:
        return 2 * i + 1
    return 2 * i + 2
```

The code for the delete operation is in below. Note that once again the hard work is handled by a helper function, in this case `_perc_down()`.

The code for the delete operation is in below. Note that once again the hard work is handled by a helper function, in this case `_perc_down()`.

In [39]:
```
def delete(self):
    self._heap[0], self._heap[-1] = self._heap[-1], self._heap[0]
    result = self._heap.pop()
    self._perc_down(0)
    return result
```

To finish our discussion of binary heaps, we will look at a method to build an entire heap from a list of keys. The first method you might think of may be like the following. Given a list of keys, you could easily build a heap by inserting each key one at a time. Since you are starting with an empy list, it is sorted and you could use binary search to find the right position to insert the next key at a cost of approximately $O(\log n)$ operations.

To finish our discussion of binary heaps, we will look at a method to build an entire heap from a list of keys. The first method you might think of may be like the following. Given a list of keys, you could easily build a heap by inserting each key one at a time. Since you are starting with an empy list, it is sorted and you could use binary search to find the right position to insert the next key at a cost of approximately $O(\log n)$ operations.

However, remember that inserting an item in the middle of the list may require $O(n)$ operations to shift the rest of the list over to make room for the new key.

To finish our discussion of binary heaps, we will look at a method to build an entire heap from a list of keys. The first method you might think of may be like the following. Given a list of keys, you could easily build a heap by inserting each key one at a time. Since you are starting with an empy list, it is sorted and you could use binary search to find the right position to insert the next key at a cost of approximately $O(\log n)$ operations.

However, remember that inserting an item in the middle of the list may require $O(n)$ operations to shift the rest of the list over to make room for the new key.

Therefore, to insert n keys into the heap would require a total of $O(n^2)$ operations. However, if we start with an entire list then we can build the whole heap in $O(n)$ operations.

In [41]:

```
def heapify(self, not_a_heap):
    self._heap = not_a_heap[:]
    i = len(self._heap) // 2 - 1
    while i >= 0:
        self._perc_down(i)
        i = i - 1
```

In [41]:
```
def heapify(self, not_a_heap):
    self._heap = not_a_heap[:]
    i = len(self._heap) // 2 - 1
    while i >= 0:
        self._perc_down(i)
        i = i - 1
```

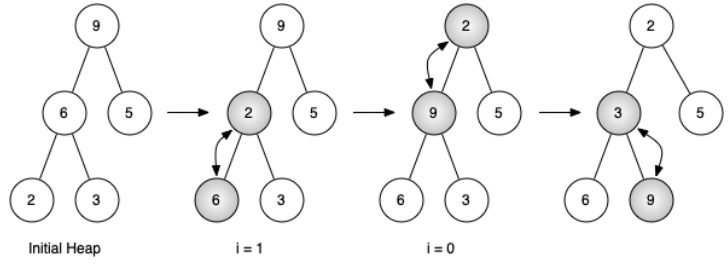


Figure above shows the swaps that the `hepify()` method makes as it moves the nodes in an initial tree of `[9, 6, 5, 2, 3]` into their proper positions. Although we start out in the middle of the tree and work our way back toward the root, the `_perc_down()` method ensures that the largest child is always moved down the tree. Because the heap is a complete binary tree, any nodes past the halfway point will be leaves and therefore have no children!

Figure above shows the swaps that the `hepify()` method makes as it moves the nodes in an initial tree of `[9, 6, 5, 2, 3]` into their proper positions. Although we start out in the middle of the tree and work our way back toward the root, the `_perc_down()` method ensures that the largest child is always moved down the tree. Because the heap is a complete binary tree, any nodes past the halfway point will be leaves and therefore have no children!

Notice that when `i = 0`, we are percolating down from the root of the tree, so this may require multiple swaps. As you can see in the rightmost two trees of the figure, first the 9 is moved out of the root position, but after 9 is moved down one level in the tree, `_perc_down()` ensures that we check the next set of children farther down in the tree to ensure that it is pushed as low as it can go.

Figure above shows the swaps that the `hepify()` method makes as it moves the nodes in an initial tree of `[9, 6, 5, 2, 3]` into their proper positions. Although we start out in the middle of the tree and work our way back toward the root, the `_perc_down()` method ensures that the largest child is always moved down the tree. Because the heap is a complete binary tree, any nodes past the halfway point will be leaves and therefore have no children!

Notice that when `i = 0`, we are percolating down from the root of the tree, so this may require multiple swaps. As you can see in the rightmost two trees of the figure, first the 9 is moved out of the root position, but after 9 is moved down one level in the tree, `_perc_down()` ensures that we check the next set of children farther down in the tree to ensure that it is pushed as low as it can go.

In this case it results in a second swap with 3. Now that 9 has been moved to the lowest level of the tree, no further swapping can be done.

In [43]:
```
from pythonds3.trees import BinaryHeap

a_heap = BinaryHeap()
a_heap.heapify([10, 4, 9, 8, 12, 15, 3, 5, 14, 18])

#while not a_heap.is_empty():
#    print(a_heap.delete())
print(a_heap)
```

```
[3, 4, 9, 5, 12, 15, 10, 8, 14, 18]
```

In [43]:
```
from pythonds3.trees import BinaryHeap

a_heap = BinaryHeap()
a_heap.heapify([10, 4, 9, 8, 12, 15, 3, 5, 14, 18])

#while not a_heap.is_empty():
#    print(a_heap.delete())
print(a_heap)
```

```
[3, 4, 9, 5, 12, 15, 10, 8, 14, 18]
```

The assertion that we can build the heap in $O(n)$ may seem a bit mysterious at first, and a proof is beyond the scope of this course. However, the key to understanding that you can build the heap in $O(n)$ is to remember that the $\log n$ factor is derived from the height of the tree. For most of the work in `heapify()`, the tree is shorter than $\log n$.

Exercise 2: Using the `heapify()` and `delete()` method, write a sorting function that can sort a list in $O(n \log n)$ time.

Exercise 2: Using the `heapify()` and `delete()` method, write a sorting function that can sort a list in $O(n \log n)$ time.

In [44]:
```
def heap_sort(unsorted_list) :
    """Sorts a list using heap sort."""
    heap = BinaryHeap()
    # 1. Build the heap in O(n) time

    # 2. The main sorting loop involves calling delete on the heap
    # to get the smallest element and adding this to the sorted list.
    # Each delete operation is O(log n)
```

In [45]:
```python
# Example list to be sorted
unsorted_list = [10, 3, 5, 1, 15, 7, 9, 2, 8]

# Sort the list using heap sort
sorted_list = heap_sort(unsorted_list)

# Print the sorted list
print("Sorted list:", sorted_list)
```

```
Sorted list: None
```

6.12. Binary Search Trees

We have already seen two different ways to get key-value pairs in a collection. Recall that these collections implement the **map** abstract data type. The two implementations of the map ADT that we have discussed were binary search on a list and hash tables.

We have already seen two different ways to get key-value pairs in a collection. Recall that these collections implement the **map** abstract data type. The two implementations of the map ADT that we have discussed were binary search on a list and hash tables.

In this section we will study binary search trees as yet another way to map from a key to a value. In this case we are not interested in the exact placement of items in the tree, but we are interested in using the binary tree structure to provide for efficient searching.

6.13. Search Tree Operations

Before we look at the implementation, let's review the interface provided by the map ADT. You will notice that this interface is very similar to the dictionary.

- `Map()`: creates a new empty map.
- `put(key, val)`: adds a new key–value pair to the map. If the key is already in the map, it replaces the old value with the new value.

Before we look at the implementation, let's review the interface provided by the map ADT. You will notice that this interface is very similar to the dictionary.

- `Map()` : creates a new empty map.
- `put(key, val)` : adds a new key–value pair to the map. If the key is already in the map, it replaces the old value with the new value.

- `get(key)` : takes a key and returns the matching value stored in the map or `None` otherwise.
- `del` : deletes the key–value pair from the map using a statement of the form `del map[key]` .
- `size()` : returns the number of key–value pairs stored in the map.
- `in` : return `True` for a statement of the form key in map if the given key is in the map, `False` otherwise.

6.14. Search Tree Implementation

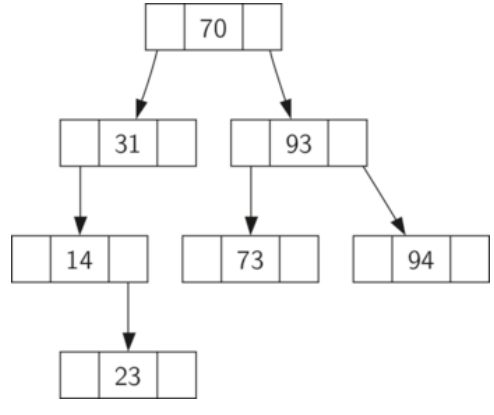
A binary search tree (BST) relies on the property that keys that are less than the parent are found in the left subtree, and keys that are greater than the parent are found in the right subtree. We will call this the BST property.

A binary search tree (BST) relies on the property that keys that are less than the parent are found in the left subtree, and keys that are greater than the parent are found in the right subtree. We will call this the BST property.

Figure below illustrates this property of a binary search tree, showing the keys without any associated values.

A binary search tree (BST) relies on the property that keys that are less than the parent are found in the left subtree, and keys that are greater than the parent are found in the right subtree. We will call this the BST property.

Figure below illustrates this property of a binary search tree, showing the keys without any associated values.



Now that you know what a binary search tree is, we will look at how a binary search tree is constructed. The search tree above represents the nodes that exist after we have inserted the following keys in the order shown: $70, 31, 93, 94, 14, 23, 73$. Since 70 was the first key inserted into the tree, it is the root.

Now that you know what a binary search tree is, we will look at how a binary search tree is constructed. The search tree above represents the nodes that exist after we have inserted the following keys in the order shown: $70, 31, 93, 94, 14, 23, 73$. Since 70 was the first key inserted into the tree, it is the root.

Next, 31 is less than 70, so it becomes the left child of 70. Next, 93 is greater than 70, so it becomes the right child of 70. Now we have two levels of the tree filled, so the next key is going to be the left or right child of either 31 or 93.

Now that you know what a binary search tree is, we will look at how a binary search tree is constructed. The search tree above represents the nodes that exist after we have inserted the following keys in the order shown: $70, 31, 93, 94, 14, 23, 73$. Since 70 was the first key inserted into the tree, it is the root.

Next, 31 is less than 70, so it becomes the left child of 70. Next, 93 is greater than 70, so it becomes the right child of 70. Now we have two levels of the tree filled, so the next key is going to be the left or right child of either 31 or 93.

Since 94 is greater than 70 and 93, it becomes the right child of 93. Similarly 14 is less than 70 and 31, so it becomes the left child of 31. 23 is also less than 31, so it must be in the left subtree of 31. However, it is greater than 14, so it becomes the right child of 14.

To implement the binary search tree, we will use the **nodes and references** approach similar to the one we used to implement the linked list and the expression tree.

To implement the binary search tree, we will use the **nodes and references** approach similar to the one we used to implement the linked list and the expression tree.

However, because we must be able create and work with a binary search tree that is empty, our implementation will use two classes. The first class we will call `BinarySearchTree`, and the second class we will call `TreeNode`.

To implement the binary search tree, we will use the **nodes and references** approach similar to the one we used to implement the linked list and the expression tree.

However, because we must be able create and work with a binary search tree that is empty, our implementation will use two classes. The first class we will call `BinarySearchTree`, and the second class we will call `TreeNode`.

The `BinarySearchTree` class has a reference to the `TreeNode` that is the root of the binary search tree. In most cases the external methods defined in the outer class simply check to see if the tree is empty. If there are nodes in the tree, the request is just passed on to a private method defined in the `BinarySearchTree` class that takes the root as a parameter.

In the case where the tree is empty or we want to delete the key at the root of the tree, we must take special action.

In the case where the tree is empty or we want to delete the key at the root of the tree, we must take special action.

In [47]:
```
class BinarySearchTree:
    def __init__(self):
        self.root = None
        self.size = 0

    def __len__(self):
        return self.size

    def __iter__(self):
        return self.root.__iter__()
```

In the case where the tree is empty or we want to delete the key at the root of the tree, we must take special action.

In [47]:
```
class BinarySearchTree:
    def __init__(self):
        self.root = None
        self.size = 0

    def __len__(self):
        return self.size

    def __iter__(self):
        return self.root.__iter__()
```

The `TreeNode` class provides many helper methods that make the work done in the `BinarySearchTree` class methods much easier.

In [48]:

```
class TreeNode:
    def __init__(self, key, value, left=None, right=None, parent=None):
        self.key = key
        self.value = value
        self.left_child = left
        self.right_child = right
        self.parent = parent

    def is_left_child(self):
        return self.parent and self.parent.left_child is self
    def is_right_child(self):
        return self.parent and self.parent.right_child is self
    def is_root(self):
        return not self.parent
    def is_leaf(self):
        return not (self.right_child or self.left_child)
    def has_any_child(self):
        return self.right_child or self.left_child
    def has_children(self):
        return self.right_child and self.left_child

    def replace_value(self, key, value, left, right):
        self.key = key
        self.value = value
        self.left_child = left
        self.right_child = right
        if self.left_child:
            self.left_child.parent = self
        if self.right_child:
            self.right_child.parent = self
```

The `TreeNode` class will also explicitly keep track of the parent as an attribute of each node. You will see why this is important when we discuss the implementation for the `del` operator.

The `TreeNode` class will also explicitly keep track of the parent as an attribute of each node. You will see why this is important when we discuss the implementation for the `del` operator.

Another interesting aspect is that we use optional parameters which make it easy for us to create a `TreeNode` under several different circumstances.

The `TreeNode` class will also explicitly keep track of the parent as an attribute of each node. You will see why this is important when we discuss the implementation for the `del` operator.

Another interesting aspect is that we use optional parameters which make it easy for us to create a `TreeNode` under several different circumstances.

Now that we have the `BinarySearchTree` shell and the `TreeNode`, it is time to write the `put()` method that will allow us to build our binary search tree. The method is a method of the `BinarySearchTree` class. This method will check to see if the tree already has a root. If there is not a root, then `put()` will create a new `TreeNode` and install it as the root of the tree.

If a root node is already in place, then `put()` calls the private recursive helper method `_put()` to search the tree according to the following algorithm.

If a root node is already in place, then `put()` calls the private recursive helper method `_put()` to search the tree according to the following algorithm.

- Starting at the root of the tree, search the binary tree comparing the new key to the key in the current node. If the new key is less than the current node, search the left subtree. If the new key is greater than the current node, search the right subtree.

If a root node is already in place, then `put()` calls the private recursive helper method `_put()` to search the tree according to the following algorithm.

- Starting at the root of the tree, search the binary tree comparing the new key to the key in the current node. If the new key is less than the current node, search the left subtree. If the new key is greater than the current node, search the right subtree.

- When there is no left or right child to search, we have found the position in the tree where the new node should be installed.

- To add a node to the tree, create a new `TreeNode` object and insert the object at the point discovered in the previous step.

In [49]:

```
def put(self, key, value):
    if self.root:
        self._put(key, value, self.root)
    else:
        self.root = TreeNode(key, value)
    self.size = self.size + 1

def _put(self, key, value, current_node):
    if key < current_node.key:
        if current_node.left_child:
            self._put(key, value, current_node.left_child)
        else:
            current_node.left_child = TreeNode(key, value,
                                               parent=current_node)
    else:
        if current_node.right_child:
            self._put(key, value, current_node.right_child)
        else:
            current_node.right_child = TreeNode(key, value,
                                                parent=current_node)
```

In [49]:
```
def put(self, key, value):
    if self.root:
        self._put(key, value, self.root)
    else:
        self.root = TreeNode(key, value)
    self.size = self.size + 1

def _put(self, key, value, current_node):
    if key < current_node.key:
        if current_node.left_child:
            self._put(key, value, current_node.left_child)
        else:
            current_node.left_child = TreeNode(key, value,
                                               parent=current_node)
    else:
        if current_node.right_child:
            self._put(key, value, current_node.right_child)
        else:
            current_node.right_child = TreeNode(key, value,
                                                parent=current_node)
```

With the `put()` method defined, we can easily overload the `[]` operator for assignment. This allows us to write statements like `my_zip_tree['Plymouth'] = 55446`, just like a Python dictionary!

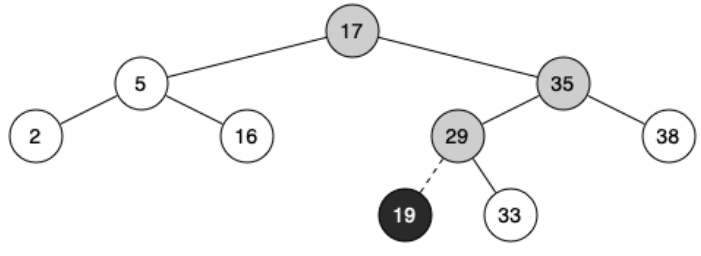
In [50]:
```
def __setitem__(self, key, value):
    self.put(key, value)
```

In [50]:
```
def __setitem__(self, key, value):
    self.put(key, value)
```

Figure below illustrates the process for inserting a new node into a binary search tree. The lightly shaded nodes indicate the nodes that were visited during the insertion process.

In [50]:
```
def __setitem__(self, key, value):
    self.put(key, value)
```

Figure below illustrates the process for inserting a new node into a binary search tree. The lightly shaded nodes indicate the nodes that were visited during the insertion process.



The `get()` method is even easier than the `put()` method because it simply searches the tree recursively until it gets to a non-matching leaf node or finds a matching key. When a matching key is found, the value stored in the payload of the node is returned.

The `get()` method is even easier than the `put()` method because it simply searches the tree recursively until it gets to a non-matching leaf node or finds a matching key. When a matching key is found, the value stored in the payload of the node is returned.

In [52]:
```
def get(self, key):
    if self.root:
        result = self._get(key, self.root)
        if result:
            return result.value
    return None

def _get(self, key, current_node):
    if not current_node:
        return None
    if current_node.key == key:
        return current_node
    elif key < current_node.key:
        return self._get(key, current_node.left_child)
    else:
        return self._get(key, current_node.right_child)
```

We can implement two to related methods as follows:

We can implement two to related methods as follows:

In [53]:
```
def __getitem__(self, key):
    return self.get(key)

def __contains__(self, key):
    return bool(self._get(key, self.root))
```

Finally, we turn our attention to the most challenging operation on the binary search tree, the deletion of a key. The first task is to find the node to delete by searching the tree. If the tree has more than one node we search using the `_get()` method to find the `TreeNode` that needs to be removed.

Finally, we turn our attention to the most challenging operation on the binary search tree, the deletion of a key. The first task is to find the node to delete by searching the tree. If the tree has more than one node we search using the `_get()` method to find the `TreeNode` that needs to be removed.

If the tree only has a single node, that means we are removing the root of the tree, but we still must check to make sure the key of the root matches the key that is to be deleted!

Finally, we turn our attention to the most challenging operation on the binary search tree, the deletion of a key. The first task is to find the node to delete by searching the tree. If the tree has more than one node we search using the `_get()` method to find the `TreeNode` that needs to be removed.

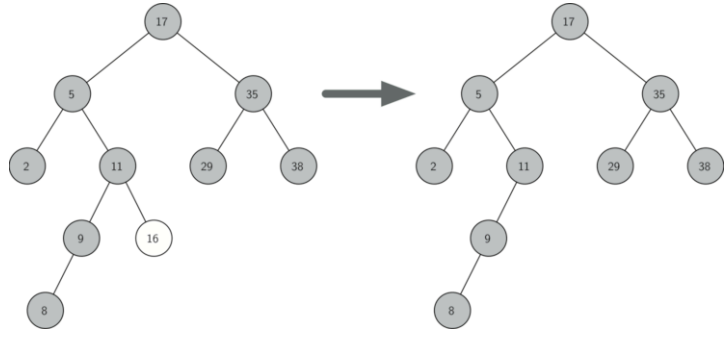
If the tree only has a single node, that means we are removing the root of the tree, but we still must check to make sure the key of the root matches the key that is to be deleted!

In [54]:
```
def delete(self, key):
    if self.size > 1:
        node_to_remove = self._get(key, self.root)
        if node_to_remove:
            self._delete(node_to_remove)
            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")
```

Once we've found the node containing the key we want to delete, there are three cases that we must consider:

Once we've found the node containing the key we want to delete, there are three cases that we must consider:

1. The node to be deleted has no children



The first case is straightforward. If the current node has no children, all we need to do is delete the node and remove the reference to this node in the parent.

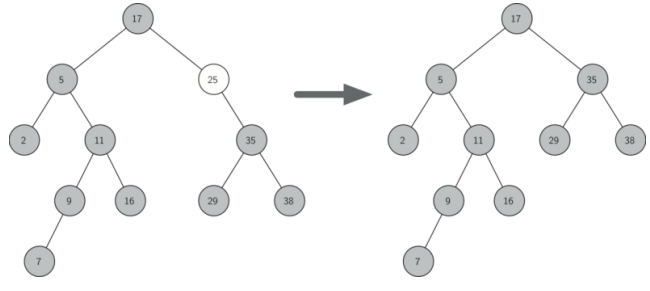
The first case is straightforward. If the current node has no children, all we need to do is delete the node and remove the reference to this node in the parent.

```
if current_node.is_leaf():
    if current_node == current_node.parent.left_child:
        current_node.parent.left_child = None
    else:
        current_node.parent.right_child = None
```

The first case is straightforward. If the current node has no children, all we need to do is delete the node and remove the reference to this node in the parent.

```
if current_node.is_leaf():
    if current_node == current_node.parent.left_child:
        current_node.parent.left_child = None
    else:
        current_node.parent.right_child = None
```

2. The node to be deleted has only one child



The second case is only slightly more complicated. If a node has only a single child, then we can simply promote the child to take the place of its parent. Since the six cases are symmetric with respect to either having a left or right child, we will just discuss the case where the current node has a left child.

The second case is only slightly more complicated. If a node has only a single child, then we can simply promote the child to take the place of its parent. Since the six cases are symmetric with respect to either having a left or right child, we will just discuss the case where the current node has a left child.

1. If the current node is a left child, then we only need to update the parent reference of the left child to point to the parent of the current node, and then update the left child reference of the parent to point to the current node's left child.

The second case is only slightly more complicated. If a node has only a single child, then we can simply promote the child to take the place of its parent. Since the six cases are symmetric with respect to either having a left or right child, we will just discuss the case where the current node has a left child.

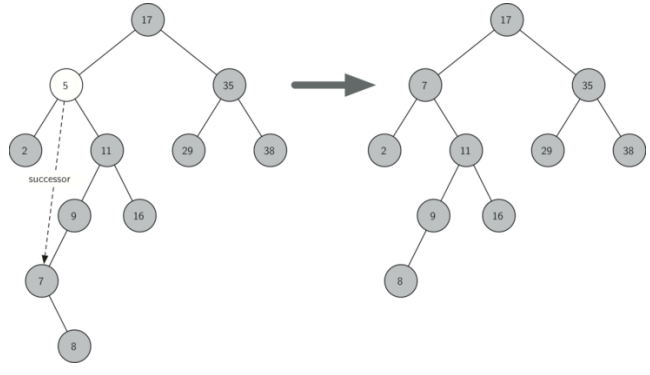
1. If the current node is a left child, then we only need to update the parent reference of the left child to point to the parent of the current node, and then update the left child reference of the parent to point to the current node's left child.

2. If the current node is a right child, then we only need to update the parent reference of the left child to point to the parent of the current node, and then update the right child reference of the parent to point to the current node's left child.

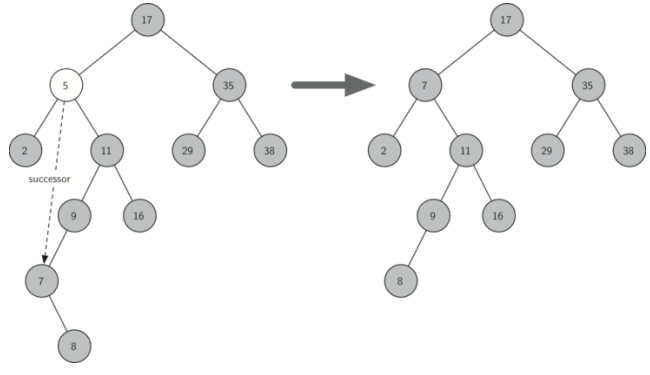
3. If the current node has no parent, it must be the root. In this case we will just replace the `key`, `value`, `left_child`, and `right_child` data by calling the replace_value method on the root.

```
else:  # removing a node with one child
    if current_node.get_left_child():
        if current_node.is_left_child():
            current_node.left_child.parent = current_node.parent
            current_node.parent.left_child = current_node.left_child
        elif current_node.is_right_child():
            current_node.left_child.parent = current_node.parent
            current_node.parent.right_child = current_node.left_child
        else:
            current_node.replace_value(
                current_node.left_child.key,
                current_node.left_child.value,
                current_node.left_child.left_child,
                current_node.left_child.right_child,
            )
    else:
```

3. The node to be deleted has two children



3. The node to be deleted has two children



The third case is the most difficult case to handle. If a node has two children, then it is unlikely that we can simply promote one of them to take the node's place!

We can, however, search the tree for a node that can be used to replace the one scheduled for deletion. What we need is a node that will preserve the bst relationships for both of the existing left and right subtrees. The node that will do this is the node that has the next-largest key in the tree. We call this node the successor.

We can, however, search the tree for a node that can be used to replace the one scheduled for deletion. What we need is a node that will preserve the bst relationships for both of the existing left and right subtrees. The node that will do this is the node that has the next-largest key in the tree. We call this node the successor.

The successor is guaranteed to have no more than one child, so we know how to remove it using the two cases for deletion that we have already implemented. Once the successor has been removed, we simply put it in the tree in place of the node to be deleted. We make use of the helper methods `find_successor()` and `splice_out()` to find and remove the successor. The reason we use `splice_out()` is that it goes directly to the node we want to splice out and makes the right changes.

We can, however, search the tree for a node that can be used to replace the one scheduled for deletion. What we need is a node that will preserve the bst relationships for both of the existing left and right subtrees. The node that will do this is the node that has the next-largest key in the tree. We call this node the successor.

The successor is guaranteed to have no more than one child, so we know how to remove it using the two cases for deletion that we have already implemented. Once the successor has been removed, we simply put it in the tree in place of the node to be deleted. We make use of the helper methods `find_successor()` and `splice_out()` to find and remove the successor. The reason we use `splice_out()` is that it goes directly to the node we want to splice out and makes the right changes.

```
elif current_node.has_children():  # removing a node with two children
    successor = current_node.find_successor()
    successor.splice_out()
    current_node.key = successor.key
    current_node.value = successor.value
```

In [55]:

```
def find_successor(self): # a method of the TreeNode class
    successor = None
    if self.right_child:
        successor = self.right_child.find_min()
    else:
        if self.parent:
            if self.is_left_child():
                successor = self.parent
            else:
                self.parent.right_child = None
                successor = self.parent.find_successor()
                self.parent.right_child = self
    return successor
```

In [55]:
```
def find_successor(self): # a method of the TreeNode class
    successor = None
    if self.right_child:
        successor = self.right_child.find_min()
    else:
        if self.parent:
            if self.is_left_child():
                successor = self.parent
            else:
                self.parent.right_child = None
                successor = self.parent.find_successor()
                self.parent.right_child = self
    return successor
```

This code makes use of the same properties of binary search trees that cause an **inorder traversal** to print out the nodes in the tree from smallest to largest. There are three cases to consider when looking for the successor:

1. If the node has a right child, then the successor is the smallest key in the right subtree.

In [55]:
```
def find_successor(self): # a method of the TreeNode class
    successor = None
    if self.right_child:
        successor = self.right_child.find_min()
    else:
        if self.parent:
            if self.is_left_child():
                successor = self.parent
            else:
                self.parent.right_child = None
                successor = self.parent.find_successor()
                self.parent.right_child = self
    return successor
```

This code makes use of the same properties of binary search trees that cause an **inorder traversal** to print out the nodes in the tree from smallest to largest. There are three cases to consider when looking for the successor:

1. If the node has a right child, then the successor is the smallest key in the right subtree.

2. If the node has no right child and is the left child of its parent, then the parent is the successor.

3. If the node is the right child of its parent, and itself has no right child, then the successor to this node is the successor of its parent, excluding this node.

In [56]:
```
def find_min(self):
    current = self
    while current.left_child:
        current = current.left_child
    return current
```

In [56]:
```
def find_min(self):
    current = self
    while current.left_child:
        current = current.left_child
    return current
```

The `find_min()` method is called to find the minimum key in a subtree. You should convince yourself that the minimum value key in any binary search tree is the leftmost child of the tree. Therefore the `find_min()` method simply follows the `left_child` references in each node of the subtree until it reaches a node that does not have a left child.

In [57]:

```
def splice_out(self):
    if self.is_leaf():
        if self.is_left_child():
            self.parent.left_child = None
        else:
            self.parent.right_child = None
    elif self.has_any_child():
        if self.left_child:
            if self.is_left_child():
                self.parent.left_child = self.left_child
            else:
                self.parent.right_child = self.left_child
            self.left_child.parent = self.parent
        else:
            if self.is_left_child():
                self.parent.left_child = self.right_child
            else:
                self.parent.right_child = self.right_child
            self.right_child.parent = self.parent
```

We need to look at one last interface method for the binary search tree. Suppose that we would like to simply iterate over all the keys in the tree in order. You already know how to traverse a binary tree in order, using the inorder traversal algorithm. However, writing an iterator requires a bit more work since an iterator should return only one node each time the iterator is called.

We need to look at one last interface method for the binary search tree. Suppose that we would like to simply iterate over all the keys in the tree in order. You already know how to traverse a binary tree in order, using the inorder traversal algorithm. However, writing an iterator requires a bit more work since an iterator should return only one node each time the iterator is called.

In [59]:
```
def __iter__(self):
    if self:
        if self.left_child:
            for elem in self.left_child:
                yield elem
        yield self.key
        if self.right_child:
            for elem in self.right_child:
                yield elem
```

We need to look at one last interface method for the binary search tree. Suppose that we would like to simply iterate over all the keys in the tree in order. You already know how to traverse a binary tree in order, using the inorder traversal algorithm. However, writing an iterator requires a bit more work since an iterator should return only one node each time the iterator is called.

In [59]:
```
def __iter__(self):
    if self:
        if self.left_child:
            for elem in self.left_child:
                yield elem
        yield self.key
        if self.right_child:
            for elem in self.right_child:
                yield elem
```

At first glance you might think that the code is not recursive. However, remember that `__iter__` overrides the `for ... in` operation for iteration, so it really is recursive!

In [61]:
```
from pythonds3.trees import BinarySearchTree

my_tree = BinarySearchTree()
my_tree["a"], my_tree["q"] = "a", "quick"
my_tree["b"], my_tree["f"] = "brown", "fox"
my_tree["j"], my_tree["o"] = "jumps", "over"
my_tree["t"], my_tree["l"] = "the", "lazy"
my_tree["d"] = "dog"

print(my_tree["q"])
print(my_tree["l"])
print("There are {} items in this tree".format(len(my_tree)))
my_tree.delete("a")
print("There are {} items in this tree".format(len(my_tree)))

for node in my_tree:
    print(my_tree[node], end=" ")
print()
```

```
quick
lazy
There are 9 items in this tree
There are 8 items in this tree
brown dog fox jumps lazy over quick the
```

Exercise 3: Using the `put()` and `in` method, write a sorting function that can sort a list in $O(n \log n)$ time in average case.

Exercise 3: Using the `put()` and `in` method, write a sorting function that can sort a list in $O(n \log n)$ time in average case.

In [62]:
```
def tree_sort(values):
    bst = BinarySearchTree()
    # 1. Insert all elements into the BST. Each insert is O(log n) on average,
    # leading to O(n log n) for all insertions if the tree is balanced.

    # 2. Perform an in-order traversal of the BST. This operation is O(n)
    # because each node is visited exactly once in sorted order.
```

Exercise 3: Using the `put()` and `in` method, write a sorting function that can sort a list in $O(n \log n)$ time in average case.

In [62]:
```
def tree_sort(values):
    bst = BinarySearchTree()
    # 1. Insert all elements into the BST. Each insert is O(log n) on average,
    # leading to O(n log n) for all insertions if the tree is balanced.

    # 2. Perform an in-order traversal of the BST. This operation is O(n)
    # because each node is visited exactly once in sorted order.
```

In [63]:
```
# Example list to be sorted
unsorted_list = [20, 1, 15, 22, 10, 3, 7, 5, 8, 12]

# Sort the list using tree sort
sorted_list = tree_sort(unsorted_list)

# Print the sorted list
print("Sorted list:", sorted_list)
```

```
Sorted list: None
```

6.15. Search Tree Analysis

Let's first look at the `put()` method. The limiting factor on its performance is the height of the binary tree. Recall from the vocabulary section that the height of a tree is the number of edges between the root and the deepest leaf node.

Let's first look at the `put()` method. The limiting factor on its performance is the height of the binary tree. Recall from the vocabulary section that the height of a tree is the number of edges between the root and the deepest leaf node.

The height is the limiting factor because when we are searching for the appropriate place to insert a node into the tree, we will need to do at most one comparison at each level of the tree!

Let's first look at the `put()` method. The limiting factor on its performance is the height of the binary tree. Recall from the vocabulary section that the height of a tree is the number of edges between the root and the deepest leaf node.

The height is the limiting factor because when we are searching for the appropriate place to insert a node into the tree, we will need to do at most one comparison at each level of the tree!

Note that if the keys are added in a random order, the height of the tree is going to be around $\log_2 n$ where n is the number of nodes in the tree. This is because if the keys are randomly distributed, about half of them will be less than the root and about half will be greater than the root.

Remember that in a binary tree there is one node at the root, two nodes in the next level, and four at the next. The number of nodes at any particular level is 2^d where d is the depth of the level. The total number of nodes in a perfectly balanced binary tree is $2^{h+1} - 1$, where h represents the height of the tree.

Remember that in a binary tree there is one node at the root, two nodes in the next level, and four at the next. The number of nodes at any particular level is 2^d where d is the depth of the level. The total number of nodes in a perfectly balanced binary tree is $2^{h+1} - 1$, where h represents the height of the tree.

A perfectly balanced tree has the same number of nodes in the left subtree as the right subtree. In a balanced binary tree, the worst-case performance of `put()` is $O(\log_2 n)$.

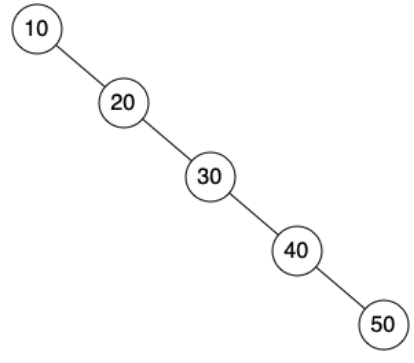
Remember that in a binary tree there is one node at the root, two nodes in the next level, and four at the next. The number of nodes at any particular level is 2^d where d is the depth of the level. The total number of nodes in a perfectly balanced binary tree is $2^{h+1} - 1$, where h represents the height of the tree.

A perfectly balanced tree has the same number of nodes in the left subtree as the right subtree. In a balanced binary tree, the worst-case performance of `put()` is $O(\log_2 n)$.

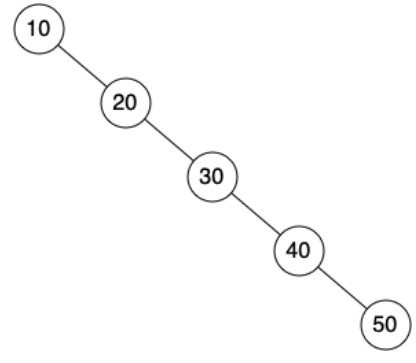
Notice that this is the inverse relationship to the calculation in the previous paragraph. So $\log_2 n$ gives us the height of the tree and represents the maximum number of comparisons that `put()` will need to do as it searches for the proper place to insert a new node.

Unfortunately it is possible to construct a search tree that has height n simply by inserting the keys in sorted order! An example of this is shown below:

Unfortunately it is possible to construct a search tree that has height n simply by inserting the keys in sorted order! An example of this is shown below:



Unfortunately it is possible to construct a search tree that has height n simply by inserting the keys in sorted order! An example of this is shown below:



In this case the performance of the put method is $O(n)$.

Now that you understand that the performance of the `put()` method is limited by the height of the tree, you can probably guess that other methods, `get()`, `in`, and `del`, are limited as well. Since `get()` searches the tree to find the key, in the worst case the tree is searched all the way to the bottom and no key is found.

Now that you understand that the performance of the `put()` method is limited by the height of the tree, you can probably guess that other methods, `get()`, `in`, and `del`, are limited as well. Since `get()` searches the tree to find the key, in the worst case the tree is searched all the way to the bottom and no key is found.

At first glance `del` might seem more complicated since it may need to search for the successor before the deletion operation can complete. But remember that the worst-case scenario to find the successor is also just the height of the tree which means that you would simply double the work. Since doubling is a constant factor, it does not change worst-case analysis of $O(n)$ for an unbalanced tree.

6.16. Balanced Binary Search Trees

In the previous section we looked at building a binary search tree. As we learned, the performance of the binary search tree can degrade to $O(n)$ for operations like `get()` and `put()` when the tree becomes unbalanced.

In the previous section we looked at building a binary search tree. As we learned, the performance of the binary search tree can degrade to $O(n)$ for operations like `get()` and `put()` when the tree becomes unbalanced.

In this section we will look at a special kind of binary search tree that automatically makes sure that the tree remains balanced at all times. This tree is called an AVL tree.

In the previous section we looked at building a binary search tree. As we learned, the performance of the binary search tree can degrade to $O(n)$ for operations like `get()` and `put()` when the tree becomes unbalanced.

In this section we will look at a special kind of binary search tree that automatically makes sure that the tree remains balanced at all times. This tree is called an AVL tree.

An AVL tree implements the `Map` abstract data type just like a regular binary search tree; the only difference is in how the tree performs. To implement our AVL tree we need to keep track of a balance factor for each node in the tree.

We do this by looking at the heights of the left and right subtrees for each node. More formally, we define the balance factor for a node as the difference between the height of the left subtree and the height of the right subtree.

We do this by looking at the heights of the left and right subtrees for each node. More formally, we define the balance factor for a node as the difference between the height of the left subtree and the height of the right subtree.

$$balance\_factor = height(left\_subtree) - height(right\_subtree)$$

We do this by looking at the heights of the left and right subtrees for each node. More formally, we define the balance factor for a node as the difference between the height of the left subtree and the height of the right subtree.

$$balance\_factor = height(left\_subtree) - height(right\_subtree)$$

Using the definition for balance factor given above, we say that a subtree is left-heavy if the balance factor is greater than zero. If the balance factor is less than zero, then the subtree is right-heavy. If the balance factor is zero, then the tree is perfectly in balance.

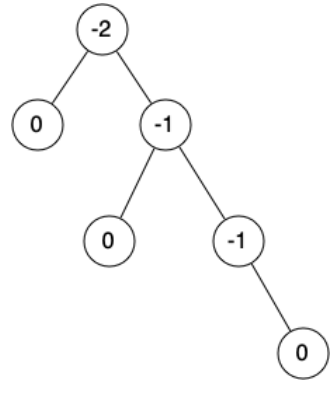
For purposes of implementing an AVL tree and gaining the benefit of having a balanced tree, we will define a tree to be in balance if the balance factor is -1, 0, or 1. Once the balance factor of a node in a tree is outside this range we will need to have a procedure to bring the tree back into balance.

For purposes of implementing an AVL tree and gaining the benefit of having a balanced tree, we will define a tree to be in balance if the balance factor is -1, 0, or 1. Once the balance factor of a node in a tree is outside this range we will need to have a procedure to bring the tree back into balance.

Below shows an example of an unbalanced right-heavy tree and the balance factors of each node:

For purposes of implementing an AVL tree and gaining the benefit of having a balanced tree, we will define a tree to be in balance if the balance factor is -1, 0, or 1. Once the balance factor of a node in a tree is outside this range we will need to have a procedure to bring the tree back into balance.

Below shows an example of an unbalanced right-heavy tree and the balance factors of each node:



6.17. AVL Tree Performance

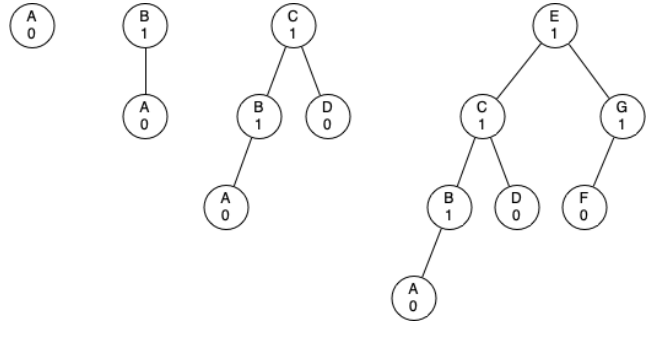
Before we proceed any further let's look at the result of enforcing this new balance factor requirement. Our claim is that by ensuring that a tree always has a balance factor of -1, 0, or 1 we can get better Big-O performance of key operations.

Before we proceed any further let's look at the result of enforcing this new balance factor requirement. Our claim is that by ensuring that a tree always has a balance factor of -1, 0, or 1 we can get better Big-O performance of key operations.

Let us start by thinking about how this balance condition changes the worst-case tree. There are two possibilities to consider, a left-heavy tree and a right-heavy tree. If we consider trees of heights 0, 1, 2, and 3, Figure below illustrates the most unbalanced left-heavy tree possible under the new rules.

Before we proceed any further let's look at the result of enforcing this new balance factor requirement. Our claim is that by ensuring that a tree always has a balance factor of -1, 0, or 1 we can get better Big-O performance of key operations.

Let us start by thinking about how this balance condition changes the worst-case tree. There are two possibilities to consider, a left-heavy tree and a right-heavy tree. If we consider trees of heights 0, 1, 2, and 3, Figure below illustrates the most unbalanced left-heavy tree possible under the new rules.



Looking at the total number of nodes in the tree we see that for a tree of height 0 there is 1 node, for a tree of height 1 there is $1 + 1 = 2$ nodes, for a tree of height 2 there are $1 + 1 + 2 = 4$, and for a tree of height 3 there are $1 + 2 + 4 = 7$. More generally the pattern we see for the number of nodes in a tree of height N_h is:

Looking at the total number of nodes in the tree we see that for a tree of height 0 there is 1 node, for a tree of height 1 there is $1 + 1 = 2$ nodes, for a tree of height 2 there are $1 + 1 + 2 = 4$, and for a tree of height 3 there are $1 + 2 + 4 = 7$. More generally the pattern we see for the number of nodes in a tree of height N_h is:

$$N_h = 1 + N_{h-1} + N_{h-2}$$

Looking at the total number of nodes in the tree we see that for a tree of height 0 there is 1 node, for a tree of height 1 there is $1 + 1 = 2$ nodes, for a tree of height 2 there are $1 + 1 + 2 = 4$, and for a tree of height 3 there are $1 + 2 + 4 = 7$. More generally the pattern we see for the number of nodes in a tree of height N_h is:

$$N_h = 1 + N_{h-1} + N_{h-2}$$

This recurrence may look familiar to you because it is very similar to the Fibonacci sequence. We can use this fact to derive a formula for the height of an AVL tree given the number of nodes in the tree.

Recall that for the Fibonacci sequence the i^{th} Fibonacci number is given by:

Recall that for the Fibonacci sequence the i^{th} Fibonacci number is given by:

$$
\begin{aligned}
F_0 &= 0 \\
F_1 &= 1 \\
F_i &= F_{i-1} + F_{i-2} \text{ for all } i \geq 2
\end{aligned}
$$

Recall that for the Fibonacci sequence the i^{th} Fibonacci number is given by:

$$
\begin{aligned}
F_0 &= 0 \\
F_1 &= 1 \\
F_i &= F_{i-1} + F_{i-2} \text{ for all } i \geq 2
\end{aligned}
$$

An important mathematical result is that as the numbers of the Fibonacci sequence get larger and larger the ratio of F_i/F_{i-1} becomes closer and closer to approximating the golden ratio Φ which is defined as $\Phi = \frac{1+\sqrt{5}}{2}$.

We will simply use this equation to approximate F_i as $F_i = \Phi^i/\sqrt{5}$. If we make use of this approximation we can rewrite the equation for N_h as:

We will simply use this equation to approximate F_i as $F_i = \Phi^i/\sqrt{5}$. If we make use of this approximation we can rewrite the equation for N_h as:

$$N_h = F_{h+3} - 1, h \geq 1$$

We will simply use this equation to approximate F_i as $F_i = \Phi^i/\sqrt{5}$. If we make use of this approximation we can rewrite the equation for N_h as:

$$N_h = F_{h+3} - 1, h \geq 1$$

By replacing the Fibonacci reference with its golden ratio approximation we get:

We will simply use this equation to approximate F_i as $F_i = \Phi^i/\sqrt{5}$. If we make use of this approximation we can rewrite the equation for N_h as:

$$N_h = F_{h+3} - 1, h \geq 1$$

By replacing the Fibonacci reference with its golden ratio approximation we get:

$$N_h = \frac{\Phi^{h+2}}{\sqrt{5}} - 1$$

If we rearrange the terms, take the base 2 log of both sides, and then solve for h, we get the following derivation:

If we rearrange the terms, take the base 2 log of both sides, and then solve for h, we get the following derivation:

$$\begin{aligned}
\log N_h + 1 &= (h+2)\log\Phi - \frac{1}{2}\log 5 \\
h &= \frac{\log\left(N_h+1\right) - 2\log\Phi + \frac{1}{2}\log 5}{\log\Phi} \\
h &= 1.44\log N_h
\end{aligned}$$

If we rearrange the terms, take the base 2 log of both sides, and then solve for h, we get the following derivation:

$$\begin{aligned}
\log N_h + 1 &= (h+2)\log\Phi - \frac{1}{2}\log 5 \\
h &= \frac{\log\left(N_h + 1\right) - 2\log\Phi + \frac{1}{2}\log 5}{\log\Phi} \\
h &= 1.44\log N_h
\end{aligned}$$

This derivation shows us that at any time the height of our AVL tree is equal to a constant (1.44) times the log of the number of nodes in the tree. This is great news for searching our AVL tree because it limits the search to $O(\log n)$.

6.19. Summary of Map ADT Implementations

Over the past two chapters we have looked at several data structures that can be used to implement the map abstract data type: a binary search on a list, a hash table, a binary search tree, and a balanced binary search tree. To conclude this section, let's summarize the performance of each data structure for the key operations defined by the map ADT:

Over the past two chapters we have looked at several data structures that can be used to implement the map abstract data type: a binary search on a list, a hash table, a binary search tree, and a balanced binary search tree. To conclude this section, let's summarize the performance of each data structure for the key operations defined by the map ADT:

operation	Sorted List	Hash Table	Binary Search Tree	AVL Tree
`put()`	$O(n)$	$O(1)$	$O(n)$	$O(\log_2 n)$
`get()`	$O(\log_2 n)$	$O(1)$	$O(n)$	$O(\log_2 n)$
`in()`	$O(\log_2 n)$	$O(1)$	$O(n)$	$O(\log_2 n)$
`del()`	$O(1)$	$O(1)$	$O(n)$	$O(\log_2 n)$

References

1. Textbook CH6