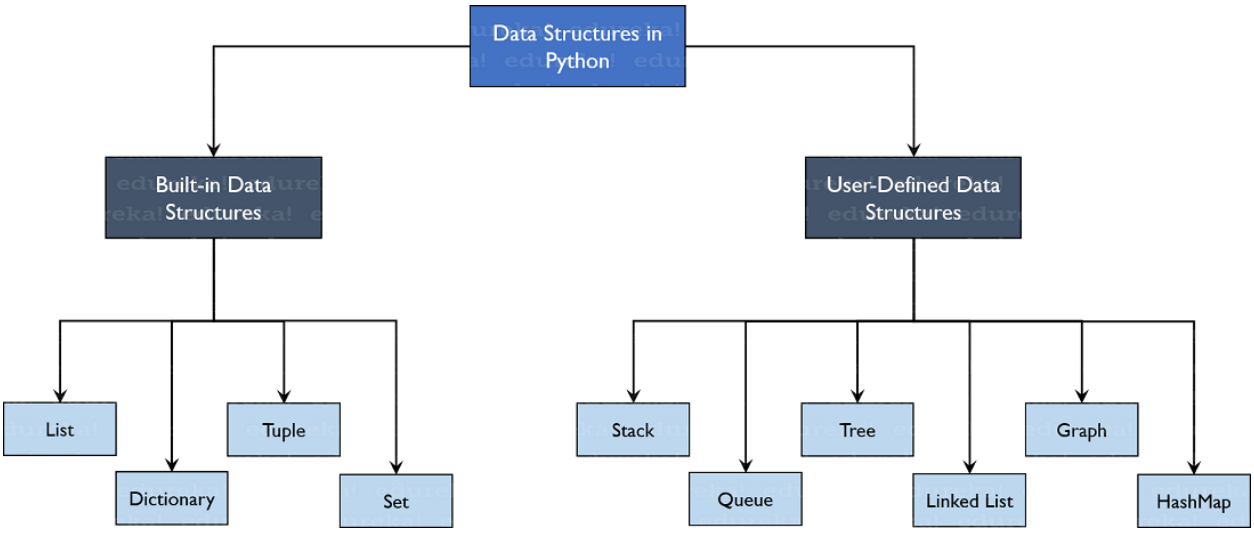
LAB2 Introduction to Python – II

1.1 Data Structures in Python:



Built-in Data-structures:

- **Lists**: Stores indexed elements that are changeable and can contain duplicate items
- **Tuples**: Stores indexed, unchangeable elements that can have duplicate copies
- **Dictionaries**: Store key-value pairs that are changeable
- **Sets**: Contains unordered, unique elements that are mutable

User-defined Data-structures:

- **Arrays**: Similar to Lists, but store single type of elements
- **Stack**: Linear LIFO (Last-In-First-Out) Data structure
- **Queues**: Linear FIFO (First-In-First-Out) data structure
- **Trees**: Non-Linear data structures having a root and nodes
- **Linked Lists**: Linear data structures that are linked with pointers
- **Graphs**: Store a collection of points or nodes along with edges
- **Hash Maps**: In Python, Hash Maps are the same as Dictionaries

1.2 List

List is a type of sequence – we can use it to store multiple values, and access them sequentially, by their position, or index, in the list. We define a list literal by putting a comma-separated list of values inside square brackets:

```
# a list of strings
animals = ['cat', 'dog', 'fish', 'bison']

# a list of integers
numbers = [1, 7, 34, 20, 12]

# an empty list
my_list = []

# a list of variables we defined somewhere else
things = [
    one_variable,
    another_variable,
    third_variable, # this trailing comma is legal in Python
]
```

To refer to an element in the list, we use the list identifier followed by the index inside square brackets. Indices are integers which start from zero:

```
print(animals[0]) # cat
print(numbers[1]) # 7

# This will give us an error, because the list only has four elements
print(animals[6])
```

We can also count from the end:

```
print(animals[-1]) # the last element -- bison
print(numbers[-2]) # the second-last element -- 20
```

We can extract a subset of a list, which will itself be a list, using a slice. This uses almost the same syntax as accessing a single element, but instead of specifying a single index between the square brackets we need to specify an upper and lower bound. Note that our sublist will include the element at the lower bound, but exclude the element at the upper bound:

```
print(animals[1:3]) # ['dog', 'fish']
print(animals[1:-1]) # ['dog', 'fish']
```

How do we check whether a list contains a particular value? We use in or not in, the membership operators:

```
numbers = [34, 67, 12, 29]
my_number = 67

if number in numbers:
    print("%d is in the list!" % number)

my_number = 90
if number not in numbers:
    print("%d is not in the list!" % number)
```

The list data type has some more methods. An example that uses most of the list methods:

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)  # Find next banana starting a position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```

1.3 Stack and Queues:

A stack (sometimes called a “push-down stack”) is an ordered collection of items where the addition of new items and the removal of existing items always takes place at the same end. This ordering principle is sometimes called LIFO, last-in first-out. It provides an ordering based on length of time in the collection. Newer items are near the top, while older items are near the base. There are two primary operations that are done on stacks: `push` and `pop`. When an element is added to the top of the stack, it is pushed onto the stack. When an element is taken off the top of the stack, it is popped off the stack.

Stack Operation	Stack Contents	Return Value
s.is_empty()	[]	True
s.push(4)	[4]	
s.push('dog')	[4,'dog']	
s.peek()	[4,'dog']	'dog'
s.push(True)	[4,'dog',True]	
s.size()	[4,'dog',True]	3
s.is_empty()	[4,'dog',True]	False
s.push(8.4)	[4,'dog',True,8.4]	
s.pop()	[4,'dog',True]	8.4
s.pop()	[4,'dog']	True
s.size()	[4,'dog']	2

Table 3.1: Sample Stack Operations

For example, if s is a stack that has been created and starts out empty, then Table 3.1 shows the results of a sequence of stack operations. Under stack contents, the top item is listed at the far right.

1.3.1 Using List as Stack:

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved ("last-in, first-out"). To add an item to the top of the stack, use `append()`. To retrieve an item from the top of the stack, use `pop()` without an explicit index. For example:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

1.3.2 Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved ("first-in, first-out"); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one). To implement a queue, use `collections.deque` which was designed to have fast appends and pops from both ends. For example:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")          # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

1.4 Tuples

Python has another sequence type which is called tuple. Tuples are similar to lists in many ways, but they are immutable. We define a tuple literal by putting a comma-separated list of values inside round brackets:

```
WEEKDAYS = ('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday',
↪'Sunday')
```

What are tuples good for? We can use them to create a sequence of values that we don't want to modify. For example, the list of weekday names is never going to change. If we store it in a tuple, we can make sure it is never modified accidentally in an unexpected place.

1.5 Sets

The Python set type is called set. A set is a collection of unique elements. If we add multiple copies of the same element to a set, the duplicates will be eliminated, and we will be left with one of each element. To define a set literal, we put a comma-separated list of values inside curly brackets.

```
animals = {'cat', 'dog', 'goldfish', 'canary', 'cat'}
print(animals) # the set will only contain one cat
```

Note: to create an empty set you have to use set(), not {}; the latter creates an empty dictionary, a data structure that we discuss in the next section. Here is a brief demonstration:

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                      # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket                 # fast membership testing
True
>>> 'crabgrass' in basket
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                  # unique letters in a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                              # letters in a but not in b
{'r', 'd', 'b'}
>>> a | b                              # letters in a or b or both
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                              # letters in both a and b
{'a', 'c'}
>>> a ^ b                              # letters in a or b but not both
{'r', 'd', 'b', 'm', 'z', 'l'}
```

1.6 Dictionaries

The Python dictionary type is called dict. We can use a dictionary to store key-value pairs. To define a dictionary literal, we put a comma-separated list of key-value pairs between curly brackets.

```
marbles = {"red": 34, "green": 30, "brown": 31, "yellow": 29 }

personal_details = {
    "name": "Jane Doe",
    "age": 38, # trailing comma is legal
}

print(marbles["green"])
print(personal_details["name"])

# This will give us an error, because there is no such key in the dictionary
print(marbles["blue"])

# modify a value
marbles["red"] += 3
personal_details["name"] = "Jane Q. Doe"
```

We use a colon to separate each key from its value. We access values in the dictionary in much the same way as list or tuple elements, but we use keys instead of indices. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys.

1.7 Two-dimensional sequences

What if we want to use a sequence to represent a two-dimensional data structure, which has both rows and columns? The easiest way to do this is to make a sequence in which each element is also a sequence. For example, we can create a list of lists:

```
my_table = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
    [10, 11, 12],
]
```

The outer list has four elements, and each of these elements is a list with three elements (which are numbers). To access one of these numbers, we need to use two indices – one for the outer list, and one for the inner list:

```
print(my_table[0][0])

# lists are mutable, so we can do this
my_table[0][0] = 42
```

When we use a two-dimensional sequence to represent tabular data, each inner sequence will have the same length, because a table is rectangular – but nothing is stopping us from constructing two-dimensional sequences which don't have this property:

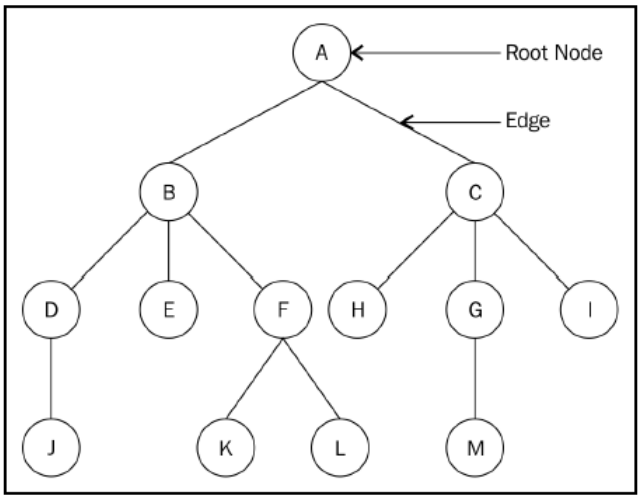
```
my_2d_list = [
    [0],
    [1, 2, 3, 4],
    [5, 6],
]
```

We can also make a three-dimensional sequence by making a list of lists of lists:

```
my_3d_list = [
    [[1, 2], [3, 4]],
    [[5, 6], [7, 8]],
]

print(my_3d_list[0][0][0])
```

1.8 Trees:

A tree is a hierarchical form of data structure. When we dealt with lists, queues, and stacks, items followed each other. But in a tree, there is a parent-child relationship between items. At the top of every tree is the so-called root node. This is the ancestor of all other nodes in the tree.



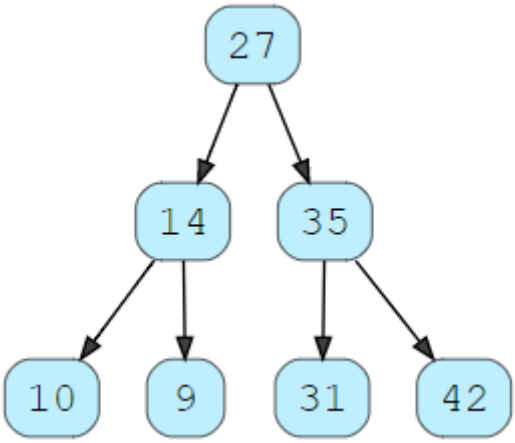
To understand trees, we need to first understand the basic ideas on which they rest. The following figure contains a typical tree consisting of character nodes lettered A through to M.

Here is a list of terms associated with a Tree:

- **Node**: Each circled alphabet represents a node. A node is any structure that holds data.
- **Root node**: The root node is the only node from which all other nodes come. A tree with an undistinguishable root node cannot be considered as a tree. The root node in our tree is the node A.
- **Sub-tree**: A sub-tree of a tree is a tree with its nodes being a descendant of some other tree. Nodes F, K, and L form a sub-tree of the original tree consisting of all the nodes.
- **Degree**: The number of sub-trees of a given node. A tree consisting of only one node has a degree of 0. This one tree node is also considered as a tree by all standards. The degree of node A is 2.
- **Leaf node**: This is a node with a degree of 0. Nodes J, E, K, L, H, M, and I are all leaf nodes.
- **Edge**: The connection between two nodes. An edge can sometimes connect a node to itself, making the edge appear as a loop.
- **Parent**: A node in the tree with other connecting nodes is the parent of those nodes. Node B is the parent of nodes D, E, and F.
- **Child**: This is a node connected to its parent. Nodes B and C are children of node A, the parent and root node.
- **Sibling**: All nodes with the same parent are siblings. This makes the nodes B and C siblings.
- **Level**: The level of a node is the number of connections from the root node. The root node is at level 0. Nodes B and C are at level 1.
- **Height of a tree**: This is the number of levels in a tree. Our tree has a height of 4.
- **Depth**: The depth of a node is the number of edges from the root of the tree to that node. The depth of node H is 2.

1.8.1 Binary tree

A tree whose elements have at most two children is called a binary tree. Each element in a binary tree can have only two children. A node's left child must have a value less than its parent's value, and the node's right child must have a value greater than its parent value.



Implementation: Here we have created a “node” class and assigned a value to the node.

```
# node class
class Node:

    def __init__(self, data):
        # left child
        self.left = None
        # right child
        self.right = None
        # node's value
        self.data = data

    # print function
    def PrintTree(self):
        print(self.data)

root = Node(27)

root.PrintTree()
```

Being a binary tree node, these references are to the left and the right children. To test this class out, we first create a few nodes:

```
n1 = Node("root node")
n2 = Node("left child node")
n3 = Node("right child node")
n4 = Node("left grandchild node")
```

Next, we connect the nodes to each other. We let **`n1`** be the root node with **`n2`** and **`n3`** as its children. Finally, we hook **`n4`** as the left child to **`n2`**, so that we get a few iterations when we traverse the left sub-tree:

```
n1.left_child = n2
n1.right_child = n3
n2.left_child = n4
```

Once we have our tree structure set up, we are ready to traverse it.

1.8.2 Binary search tree implementation

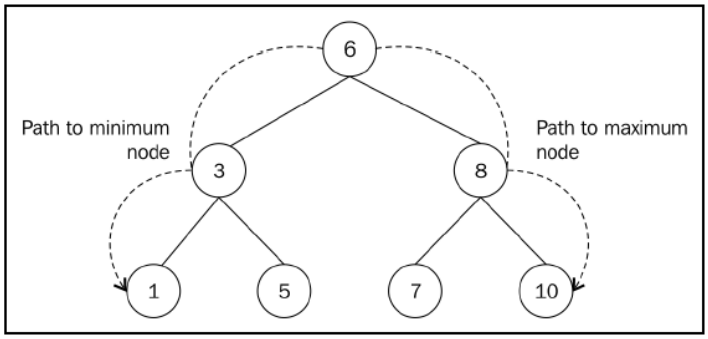
Let us begin our implementation of a BST. We will want the tree to hold a reference to its own root node:

```
class Tree:
    def __init__(self):
        self.root_node = None
```

That's all that is needed to maintain the state of a tree. Let's examine the main operations on the tree in the next section.

1.8.2.1 Finding the minimum and maximum nodes

The structure of the BST makes looking for the node with the maximum and minimum values very easy. To find the node with smallest value, we start our traversal from the root of the tree and visit the left node each time we reach a sub-tree. We do the opposite to find the node with the biggest value in the tree:



The method that returns the minimum node is as follows:

```
def find_min(self):
    current = self.root_node
    while current.left_child:
        current = current.left_child

    return current
```

The while loop continues to get the left node and visits it until the last left node points to None. It is a very simple method. The method to return the maximum node does the opposite, where `current.left_child` now becomes `current.right_child`.

1.8.2.2 Inserting nodes

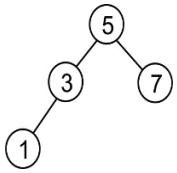
One of the operations on a BST is the need to insert data as nodes. Whereas in our first implementation, we had to insert the nodes ourselves, here we are going to let the tree be in charge of storing its data.

In order to make a search possible, the nodes must be stored in a specific way. For each given node, its left child node will hold data that is less than its own value, as already discussed. That node's right child node will hold data greater than that of its parent node.

We are going to create a new BST of integers by starting with the data 5. To do this, we will create a node with its data attribute set to 5.

Now, to add the second node with value 3, 3 is compared with 5, the root node. Since 5 is greater than 3, it will be put in the left sub-tree of node 5. The tree satisfies the BST rule, where all the nodes in the left sub-tree are less than its parent. To add another node of value 7 to the tree, we start from the root node with value 5 and do a comparison. Since 7 is greater than 5, the node with value 7 is situated to the right of this root. What happens when we want to add a node that is equal to an existing node? We will simply add it as a left node and maintain this rule throughout the structure.

If a node already has a child in the place where the new node goes, then we have to move down the tree and attach it. Let's add another node with value 1. Starting from the root of the tree, we do a comparison between 1 and 5. The comparison reveals that 1 is less than 5, so we move our attention to the left node of 5, which is the node with value 3. We compare 1 with 3 and since 1 is less than 3, we move a level below node 3 and to its left. But there is no node there. Therefore, we create a node with the value 1 and associate it with the left pointer of node 3 to obtain the following structure:



So far, we have been dealing only with nodes that contain only integers or numbers. For numbers, the idea of greater than and lesser than are clearly defined. Strings would be compared alphabetically, so there are no major problems there either. But if you want to store your own custom data types inside a BST, you would have to make sure that your class supports ordering.

Let's now create a function that enables us to add data as nodes to the BST. We begin with a function declaration:

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

By now, you will be used to the fact that we encapsulate the data in a node. This way, we hide away the node class from the client code, who only needs to deal with the tree:

```
node = Node(data)
```

A first check will be to find out whether we have a root node. If we don't, the new node becomes the root node (we cannot have a tree without a root node):

```
if self.root_node is None:
    self.root_node = node
else:
```

As we walk down the tree, we need to keep track of the current node we are working on, as well as its parent. The variable `current` is always used for this purpose:

```
current = self.root_node
parent = None
while True:
    parent = current
```

Here we must perform a comparison. If the data held in the new node is less than the data held in the current node, then we check whether the current node has a left child node. If it doesn't, this is where we insert the new node. Otherwise, we keep traversing:

```
if node.data < current.data:
    current = current.left_child
    if current is None:
        parent.left_child = node
        return
```

Now we take care of the greater than or equal case. If the current node doesn't have a right child node, then the new node is inserted as the right child node. Otherwise, we move down and continue looking for an insertion point:

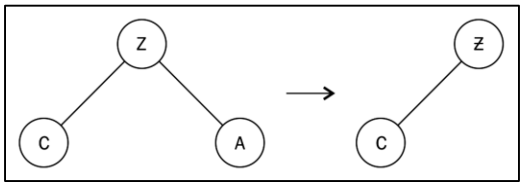
```
else:
    current = current.right_child
    if current is None:
        parent.right_child = node
        return
```

1.8.2.3 Deleting nodes

Another important operation on a BST is the deletion or removal of nodes. There are three scenarios that we need to cater for during this process. The node that we want to remove might have the following:

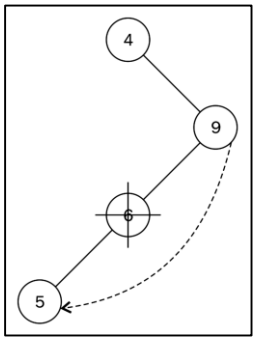
1. No children
2. One child
3. Two children

The first scenario is the easiest to handle. If the node about to be removed has no children, we simply detach it from its parent:



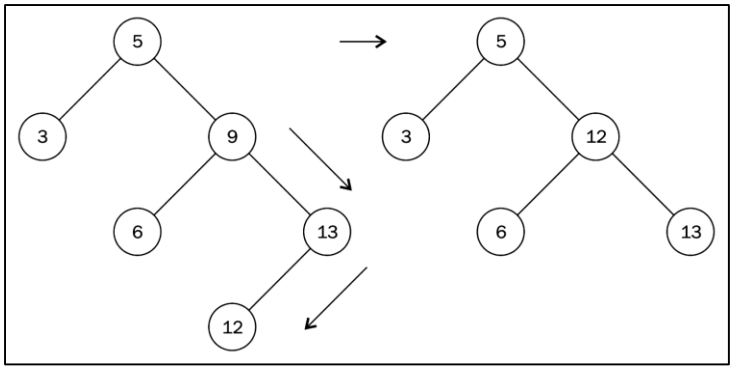
Because node A has no children, we will simply dissociate it from its parent, node Z.

On the other hand, when the node we want to remove has one child, the parent of that node is made to point to the child of that particular node:



In order to remove node 6, which has as its only child, node 5, we point the left pointer of node 9 to node 5. The relationship between the parent node and child has to be preserved. That is why we need to take note of how the child node is connected to its parent (which is the node about to be deleted). The child node of the deleted node is stored. Then we connect the parent of the deleted node to that child node.

A more complex scenario arises when the node we want to delete has two children:



We cannot simply replace node 9 with either node 6 or 13. What we need to do is to find the next biggest descendant of node 9. This is node 12. To get to node 12, we move to the right node of node 9. And then move left to find the leftmost node. Node 12 is called the in-order successor of node 9. The second step resembles the move to find the maximum node in a sub-tree.

We replace the value of node 9 with the value 12 and remove node 12. In removing node 12, we end up with a simpler form of node removal that has been addressed previously. Node 12 has no children, so we apply the rule for removing nodes without children accordingly.

Our **`node`** class does not have reference to a parent. As such, we need to use a helper method to search for and return the node with its parent node. This method is similar to the **`search`** method:

```
def get_node_with_parent(self, data):
    parent = None
    current = self.root_node
    if current is None:
        return (parent, None)
    while True:
        if current.data == data:
            return (parent, current)
        elif current.data > data:
            parent = current
            current = current.left_child
        else:
            parent = current
            current = current.right_child

    return (parent, current)
```

The only difference is that before we update the current variable inside the loop, we store its parent with `parent = current`. The method to do the actual removal of a node begins with this search:

```
def remove(self, data):
    parent, node = self.get_node_with_parent(data)

    if parent is None and node is None:
        return False

    # Get children count
    children_count = 0

    if node.left_child and node.right_child:
        children_count = 2
    elif (node.left_child is None) and (node.right_child is None):
        children_count = 0
    else:
        children_count = 1
```

We pass the parent and the found node to `parent` and `node` respectively with the line `parent, node = self.get_node_with_parent(data)`. It is helpful to know the number of children that the node we want to delete has. That is the purpose of the `if` statement.

```
if children_count == 0:
    if parent:
        if parent.right_child is node:
            parent.right_child = None
        else:
            parent.left_child = None
    else:
        self.root_node = None
```

In the case where the node about to be deleted has only one child, the `elif` part of the `if` statement does the following:

```
elif children_count == 1:
    next_node = None
    if node.left_child:
        next_node = node.left_child
else:
    next_node = node.right_child

if parent:
    if parent.left_child is node:
        parent.left_child = next_node
    else:
        parent.right_child = next_node
else:
    self.root_node = next_node
```

`next_node` is used to keep track of where the single node pointed to by the node we want to delete is. We then connect `parent.left_child` or `parent.right_child` to `next_node`.

Lastly, we handle the condition where the node we want to delete has two children:

```
...
else:
    parent_of_leftmost_node = node
    leftmost_node = node.right_child
    while leftmost_node.left_child:
        parent_of_leftmost_node = leftmost_node
        leftmost_node = leftmost_node.left_child

    node.data = leftmost_node.data
```

In finding the in-order successor, we move to the right node with `leftmost_node = node.right_child`. As long as there exists a left node, `leftmost_node.left_child` will evaluate to `True` and the `while` loop will run. When we get to the leftmost node, it will either be a leaf node (meaning that it will have no child node) or have a right child.

We update the node about to be removed with the value of the in-order successor with `node.data = leftmost_node.data`:

```
if parent_of_leftmost_node.left_child == leftmost_node:
   parent_of_leftmost_node.left_child = leftmost_node.right_child
else:
   parent_of_leftmost_node.right_child = leftmost_node.right_child
```

The preceding statement allows us to properly attach the parent of the leftmost node with any child node. Observe how the right-hand side of the equals sign stays unchanged. That is because the in-order successor can only have a right child as its only child.

1.8.3 Searching the tree

Since the `insert` method organizes data in a specific way, we will follow the same procedure to find the data. In this implementation, we will simply return the data if it was found or `None` if the data wasn't found:

```
def search(self, data):
```

We need to start searching at the very top, that is, at the root node:

```
current = self.root_node
while True:
```

We may have passed a leaf node, in which case the data doesn't exist in the tree and we return `None` to the client code:

```
if current is None:
    return None
```

We might also have found the data, in which case we return it:

```
elif current.data is data:
    return data
```

As per the rules for how data is stored in the BST, if the data we are searching for is less than that of the current node, we need to go down the tree to the left:

```
elif current.data > data:
    current = current.left_child
```

Now we only have one option left: the data we are looking for is greater than the data held in the current node, which means we go down the tree to the right:

```
else:
    current = current.right_child
```

Finally, we can write some client code to test how the BST works. We create a tree and insert a few numbers between 1 and 10. Then we search for all the numbers in that range. The ones that exist in the tree get printed:

```
tree = Tree()
tree.insert(5)
tree.insert(2)
tree.insert(7)
tree.insert(9)
tree.insert(1)

for i in range(1, 10):
    found = tree.search(i)
    print("{}: {}".format(i, found))
```

1.8.4 Tree traversal

Visiting all the nodes in a tree can be done depth first or breadth first. These modes of traversal are not peculiar to only binary search trees but trees in general.

1.8.4.1 Depth-first traversal

In this traversal mode, we follow a branch (or edge) to its limit before recoiling upwards to continue traversal. We will be using the recursive approach for the traversal. There are three forms of depth-first traversal, namely `in-order`, `pre-order`, and `post-order`.

In-order traversal and infix notation

Most of us are probably used to this way of representing an arithmetic expression, since this is the way we are normally taught in schools. The operator is inserted (infixed) between the operands, as in `3 + 4`. When necessary, parentheses can be used to build a more complex expression: `(4 + 5) * (5 - 3)`.

In this mode of traversal, you would visit the left sub-tree, the parent node, and finally the right sub-tree.

The recursive function to return an in-order listing of nodes in a tree is as follows:

```
def inorder(self, root_node):
    current = root_node
    if current is None:
        return
    self.inorder(current.left_child)
    print(current.data)
    self.inorder(current.right_child)
```

We visit the node by printing the node and making two recursive calls with `current.left_child` and `current.right_child`.

Pre-order traversal and prefix notation

Prefix notation is commonly referred to as Polish notation. Here, the operator comes before its operands, as in `+ 3 4`. Since there is no ambiguity of precedence, parentheses are not required: `* + 4 5 - 5 3`.

To traverse a tree in pre-order mode, you would visit the node, the left sub-tree, and the right sub-tree node, in that order.

The recursive function for this traversal is as follows:

```
def preorder(self, root_node):
    current = root_node
    if current is None:
        return
    print(current.data)
    self.preorder(current.left_child)
    self.preorder(current.right_child)
```

Note the order in which the recursive call is made.

Post-order traversal and postfix notation.

Postfix or **reverse Polish notation** (**RPN**) places the operator after its operands, as in `3 4 +`. As is the case with Polish notation, there is never any confusion over the precedence of operators, so parentheses are never needed: `4 5 + 5 3 - *`.

In this mode of traversal, you would visit the left sub-tree, the right sub-tree, and lastly the root node.

The `post-order` method is as follows:

```
def postorder(self, root_node):
    current = root_node
    if current is None:
        return
    self.postorder(current.left_child)
    self.postorder(current.right_child)

    print(current.data)
```

1.8.4.2 Breadth-first traversal

This kind of traversal starts from the root of a tree and visits the node from one level of the tree to the other. This mode of traversal is made possible by using a queue data structure. Starting with the root node, we push it into a queue. The node at the front of the queue is accessed (dequeued) and either printed and stored for later use. The left node is added to the queue followed by the right node. Since the queue is not empty, we repeat the process.

The algorithm is as follows:

```
from collections import deque
class Tree:
    def breadth_first_traversal(self):
        list_of_nodes = []
        traversal_queue = deque([self.root_node])
```

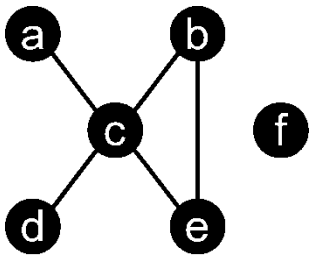
We enqueue the root node and keep a list of the visited nodes in the `list_of_nodes` list. The `dequeue` class is used to maintain a queue:

```
while len(traversal_queue) > 0:
    node = traversal_queue.popleft()
    list_of_nodes.append(node.data)

    if node.left_child:
        traversal_queue.append(node.left_child)

    if node.right_child:
        traversal_queue.append(node.right_child)
return list_of_nodes
```

If the number of elements in the `traversal_queue` is greater than zero, the body of the loop is executed. The node at the front of the queue is popped off and appended to the `list_of_nodes` list. The first `if` statement will `enqueue` the left child node of the node provided a left node exists. The second `if` statement does the same for the right child node. The `list_of_nodes` is returned in the last statement.

1.9 Graphs in Python

Before we start our possible Python representations of graphs, we want to present some general definitions of graphs and its components. A "graph" consists of "nodes", also known as "vertices". Nodes may or may not be connected with one another. The connecting line between two nodes is called an edge. If the edges between the nodes are undirected, the graph is called an undirected graph. If an edge is directed from one vertex (node) to another, a graph is called a directed graph. A directed edge is called an arc.



1.9.1 Graphs implementation using Dictionaries:

Python has no built-in data type or class for graphs, but it is easy to implement them in Python. One data type is ideal for representing graphs in Python, i.e. dictionaries. The graph in our illustration can be implemented in the following way:

```
graph = { "a" : ["c"],
          "b" : ["c", "e"],
          "c" : ["a", "b", "d", "e"],
          "d" : ["c"],
          "e" : ["c", "b"],
          "f" : []
        }
```

The keys of the dictionary above are the nodes of our graph. The corresponding values are lists with the nodes, which are connecting by an edge. There is no simpler and more elegant way to represent a graph. An edge can be seen as a 2-tuple with nodes as elements, i.e. ("a","b").

Function to generate the list of all edges:

```
def generate_edges(graph):
    edges = []
    for node in graph:
        for neighbour in graph[node]:
            edges.append((node, neighbour))

    return edges

print(generate_edges(graph))
```

This code generates the following output, if combined with the previously defined graph dictionary:

```
[('a', 'c'), ('c', 'a'), ('c', 'b'), ('c', 'd'), ('c', 'e'), ('b', 'c'), ('b', 'e'), ('e', 'c'), ('e', 'b'), ('d',
'c')]
```

As we can see, there is no edge containing the node "f". "f" is an isolated node of our graph. The following Python function calculates the isolated nodes of a given graph:

```
def find_isolated_nodes(graph):
    """ returns a list of isolated nodes. """
    isolated = []
    for node in graph:
        if not graph[node]:
            isolated += node
    return isolated
```

To generate the path from one node to the other node: Using Python dictionary, we can find the path from one node to the other in a Graph. The idea is similar to DFS in graphs. In the function, initially, the path is an empty list. In the starting, if the start node matches with the end node, the function will return the path. Otherwise the code goes forward and hits all the values of the starting node and searches for the path using recursion. If there is no such path, it returns “None”.

```
# Python program to generate the first
# path of the graph from the nodes provided

graph = {
    'a': ['c'],
    'b': ['d'],
    'c': ['e'],
    'd': ['a', 'd'],
    'e': ['b', 'c']
}

# function to find path
def find_path(graph, start, end, path=[]):
    path = path + [start]
    if start == end:
        return path
    for node in graph[start]:
        if node not in path:
            newpath = find_path(graph, node, end, path)
            if newpath:
                return newpath
            return None


# Driver function call to print the path
print(find_path(graph, 'd', 'c'))
```

Program to generate all the possible paths from one node to the other: In the above discussed program, we generated the first possible path. Now, let us generate all the possible paths from the start node to the end node. The basic functioning works same as the functioning of the above code. The place where the difference comes is instead of instantly returning the first path, it saves that path in a list named as 'paths' in the example given below. Finally, after iterating over all the possible ways, it returns the list of paths. If there is no path from the starting node to the ending node, it returns "None".

```
# Python program to generate the all possible
# path of the graph from the nodes provided
graph ={
'a':['c'],
'b':['d'],
'c':['e'],
'd':['a', 'd'],
'e':['b', 'c']
}

# function to generate all possible paths
def find_all_paths(graph, start, end, path =[]):
    path = path + [start]
    if start == end:
        return [path]
    paths = []
    for node in graph[start]:
        if node not in path:
            newpaths = find_all_paths(graph, node, end, path)
        for newpath in newpaths:
            paths.append(newpath)
    return paths

# Driver function call to print all
# generated paths
print(find_all_paths(graph, 'd', 'c'))
```

Program to generate the shortest path: To get to the shortest from all the paths, we use a little different approach as shown below. In this, as we get the path from the start node to the end node, we compare the length of the path with a variable named as shortest which is initialized with the "None" value. If the length of generated path is less than the length of shortest, if shortest is not "None", the newly generated path is set as the value of shortest. Again, if there is no path, it returns "None".

```python
# Python program to generate shortest path

graph ={
'a':['c'],
'b':['d'],
'c':['e'],
'd':['a', 'd'],
'e':['b', 'c']
}

# function to find the shortest path
def find_shortest_path(graph, start, end, path =[]):
        path = path + [start]
        if start == end:
            return path
        shortest = None
        for node in graph[start]:
            if node not in path:
                newpath = find_shortest_path(graph, node, end, path)
                if newpath:
                    if not shortest or len(newpath) < len(shortest):
                        shortest = newpath
        return shortest

# Driver function call to print
# the shortest path
print(find_shortest_path(graph, 'd', 'c'))
```

1.10 Lab Tasks

Exercise 2.1.

Complete the following code which will perform a selection sort in Python. "..." denotes missing code that should be filled in:

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[]:


        arr[i], arr[min_idx] = arr[], arr[i]
    return arr
```

Exercise 2.2.

Write a Python function that implements merge sort. It may help to write a separate function which performs merges and call it from within your merge sort implementation.

Exercise 2.3.

Consider the list of characters: ['M','O','H','S','I','N']. Show how this list is sorted using the following algorithms:

- bubble sort
- selection sort
- insertion sort
- merge sort

Exercise 2.4.

Write a function to find mean, median, mode for the given set of numbers in a list.

1. Read the elements into a list.
2. Calculate the sum of list elements.
3. Calculate the mean, median.
4. Display the result.

Exercise 2.5.

Write a function duplicate to find all duplicates in the list.

1. Create a list to read elements.
2. Pass the list as parameter to dup function.
3. Define function dup to identify duplicate elements in the list.
4. Return the final list to called function.
5. Display the result

Exercise 2.6.

Implement the following instructions.

1. Create a list `a` which contains the first three odd positive integers and a list `b` which contains the first three even positive integers.
2. Create a new list **`c`** which combines the numbers from both lists (order is unimportant).
3. Create a new list `d` which is a sorted copy of `c`, leaving `c` unchanged.
4. Reverse `d` in-place.
5. Set the fourth element of `c` to 42.
6. Append 10 to the end of `d`.
7. Append 7, 8 and 9 to the end of `c`.
8. Print the first three elements of `c`.
9. Print the last element of `d` without using its length.
10. Print the length of `d`.

Exercise 2.7.

1. Create a list `a` which contains three tuples. The first tuple should contain a single element, the second two elements and the third three elements.
2. Print the second element of the second element of `a`.
3. Create a list `b` which contains four lists, each of which contains four elements.
4. Print the last two elements of the first element of `b`.

Exercise 2.8.

1. Write a program which uses a nested for loop to populate a three-dimensional list representing a calendar: the top-level list should contain a sub-list for each month, and each month should contain four weeks. Each week should be an empty list.
2. Modify your code to make it easier to access a month in the calendar by a human-readable month name, and each week by a name which is numbered starting from 1. Add an event (in the form of a string description) to the second week in July.