# Data Structures

March 4, 2020

## 0.1 Generals data structures

Data structues are divided into two categories

**Linear data structures:** These are the data structues which store the data elemments in a sequential manner.

- Array : It is a sequencial arrangement of data elements paired with the index of the data element.
- Linked list: Each data element contains a linke to another element along with the data present in it.
- Stack: It is a data structure which follows only to specific order of operation. LIFO (last in first out ) or FILO (first in last out).
- Queue: It is simmilar to Stack but the order of operation is only FIFO (first in first out).
- Matric: It is two dimensional data structure in which the data element is referred by a pair of indices.

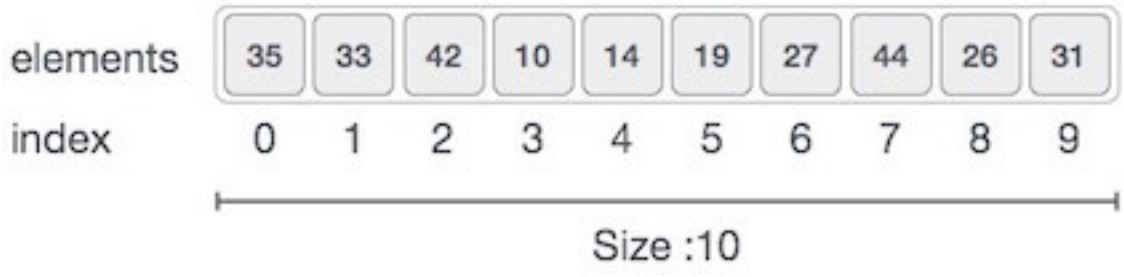
**Non-linear data sctructures:** These are the data structures in which there is no sequential linking of data elements. Any pair or group of data elements can be linked to each other and can be accessed without a strict sequence.

- Binary Tree: It is a data structure where each data element can be connected to maximum two other data elements and it starts with a root node.
- Heap: It is a special case of Tree data structure where the data in the parent node is either strictly greater than/equal tothe child nodes or strictly less than it's child nodes.
- Hash Table: It is a data structure which is made of arrays assocciated with each other using a hash function. It retrives values using keys rather than index from a data element.
- Graph: It is a arrangement of vertices and nodes where some of the nodes are connected to each other through links.

**Python specific data structures:**

- List: It is similar to array with the exception that the data elements can be of different types.
- Tuple: Tuples are similar to lists but they are immutable which means the values in a tuple cannot be modified they can only be read.
- Dictionary: The dictionary contains key-value pairs as its data elements.

# 1 Arrays




**Basic operations**:

- Traverse: - Print all the array elements one by one.
- Insertion: - Adds an element at the given index.
- Deletion: - Deletes an element at the given index.
- Search: - Searches an element using the given index or by the value.
- Update: - Updates an element at the given index.

[1]:
```
import array
arrayName = array.array('i',[10,20,30,40,50])

print("Print all the elements in the array")
for i in arrayName:
    print(i)
print("Access first and third elements of the array")
print(arrayName[0])
print(arrayName[2])
print("Insert new element 60 in indix 1")
arrayName.insert(1,60)
for i in arrayName:
    print(i)
print("Delement 40 form array")
arrayName.remove(40)
for i in arrayName:
    print(i)
print("Search element 50 using python buildin index() method")
print(" 50 is found in index number:",arrayName.index(50))
print("Update element 50 by 0")
arrayName[4] = 0
for i in arrayName:
    print(i)
```

```
Print all the elements in the array
10
20
30
40
50
```

```
Access first and third elements of the array
10
30
Insert new element 60 in indix 1
10
60
20
30
40
50
Delement 40 form array
10
60
20
30
50
Search element 50 using python buildin index() method
 50 is found in index number: 4
Update element 50 by 0
10
60
20
30
0
```

**Note: index() method can be used to find the index of an element in the sequence.But if an element occured more than one time it returns the first index**

[2]:
```
mylist = [1,2,3,43,6,7,1,2]
print("my list is : ",mylist)
print("1 is found in indices : ",mylist.index(1) )
print("43 is found in indices : ",mylist.index(43) )
```

```
my list is :  [1, 2, 3, 43, 6, 7, 1, 2]
1 is found in indices :  0
43 is found in indices :  3
```

# 2 Sets

### 2.0.1 Adding element to Sets

[3]:
```
Days = set(["MON","Tues",'Thurs'])
Days.add("SUN")
print(Days)
```

```
{'Thurs', 'Tues', 'SUN', 'MON'}
```

### 2.0.2 Removing element from sets

[4]:
```
Days.discard("SUN")
print(Days)
```

```
{'Thurs', 'Tues', 'MON'}
```

### 2.0.3 Union, intersection, difference of sets

[5]:
```
set_A = set([1,2,3,34,5])
set_B = set([1,3,45,676])
print("A unoion B is : A|B or A.union(B)",set_A|set_B)
print("A intesection B is : A&B",set_A&set_B)
print("A diffrence B is : A-B",set_A-set_B)
```

```
A unoion B is : A|B or A.union(B) {1, 2, 3, 34, 5, 676, 45}
A intesection B is : A&B {1, 3}
A diffrence B is : A-B {2, 34, 5}
```

### 2.0.4 Check for subsets or supersets

[6]:
```
set_A = set([1,2,3,34,5])
set_B = set([1,3,45,676])
print(set_A <= set_B)
```

```
False
```

[7]:
```
set_A = set([1,2,3])
set_B = set([1,3])
subsets = set_A >= set_B
supersets = set_B <= set_A
print(subsets)
print(supersets)
```

```
True
True
```

# 3 Maps

Python maps also called ChainMap is a type of data structure to manage multiple dictionaries together as one unit. The combined dictionary contains key and value pairs in a specific sequence eliminating any duplicate keys.

[8]:
```
from collections import ChainMap

dict_1 = {'day1':'Mon','day2':'Tue'}
dict_2 = {'day3':'Wed','day1':'Thu'}

res = ChainMap(dict_1,dict_2)
print(res)
```

```
ChainMap({'day1': 'Mon', 'day2': 'Tue'}, {'day3': 'Wed', 'day1': 'Thu'})
```

[9]:
```
print(res.maps)
```

```
[{'day1': 'Mon', 'day2': 'Tue'}, {'day3': 'Wed', 'day1': 'Thu'}]
```

[10]:
```
print(res.keys())
```

```
KeysView(ChainMap({'day1': 'Mon', 'day2': 'Tue'}, {'day3': 'Wed', 'day1':
'Thu'}))
```

[11]:
```
print("Keys = {}".format(list(res.keys())))
```

```
Keys = ['day3', 'day1', 'day2']
```

[12]:
```
print("Values = {}".format(list(res.values())))
```

```
Values = ['Wed', 'Mon', 'Tue']
```

[13]:
```
for key, val in res.items():
    print("{} = {}".format(key,val))
```

```
day3 = Wed
day1 = Mon
day2 = Tue
```

# 4 Nodes

There are situations when the allocation of memory to store the data cannot be in a continous block of memeory. So we take help of pointers where the along with the data, the address of the next location of data element is also store. So we know the address of the next data element from the values or current data element. In general such structures are known as pointers. But in python we refer them as Nodes.

Nodes are the foundations on which various other data structures liked lists and trees can be handled in python.

**Creation of Nodes**

The nodes aer created by implementing a class which will hold the pointers along with the data element. In the below example we create a class named daynames to hold the name of the weekdays. The nextval pointer is initialized to null and three nodes and initialized with values as shown.

[14]:
```
class daynames:
    def __init__(self,dataval=None):
        self.dataval = dataval
        self.nextval = None

    def __repr__(self):
        return "Day: {}, Next: ({})".format(self.dataval,self.nextval)
```

[15]:
```
e1 = daynames('Mon')
e1
```

[15]:
```
Day: Mon, Next: (None)
```

[16]:
```
e2 = daynames('Tue')
e3 = daynames('Wed')
```

[17]:
```
e1.nextval = e2
```

[18]:
```
e1
```

[18]:
```
Day: Mon, Next: (Day: Tue, Next: (None))
```

[19]:
```
e2.nextval = e3
```

[20]:
```
e2
```

[20]:
```
Day: Tue, Next: (Day: Wed, Next: (None))
```

[21]:
```
e1
```

[21]:
```
Day: Mon, Next: (Day: Tue, Next: (Day: Wed, Next: (None)))
```

### Traversing the Node elements

We can traverse the elements of the node created above by creating a variable and assigning the first element to it. Then we use a while loop and nextval pointer to print out all the node elements. Note that we have one more additional data element and the nextval pointers are properly arranged to get the output as a days of a week in a proper sequence.

[22]:
```
class daynames:
    def __init__(self, dataval=None):
        self.dataval = dataval
        self.nextval = None
    def __repr__(self):
```

```
        return "Day: {}, Next: ({})".format(self.dataval,self.nextval)

e1 = daynames('Mon')
e2 = daynames('Wed')
e3 = daynames('Tue')
e4 = daynames('Thu')

e1.nextval = e3
e3.nextval = e2
e2.nextval = e4

thisvalue = e1
```

[23]:
```
thisvalue
```

[23]:
```
Day: Mon, Next: (Day: Tue, Next: (Day: Wed, Next: (Day: Thu, Next: (None))))
```

[24]:
```
while thisvalue:
    print(thisvalue.dataval)
    thisvalue = thisvalue.nextval
```

```
Mon
Tue
Wed
Thu
```

## 5 Linked List

A linked list is a sequenceo of data elements, which are connected together via links. Each data element contains a connection to another data element in form a pointer.

[25]:
```
class Node:
    def __init__(self,dataval=None):
        self.dataval = dataval
        self.nextval = None

    def __repr__(self):
        return "Data: {}, Next: ({})".format(self.dataval,self.nextval)
```

[26]:
```
class SlinkedList:
    def __init__(self):
        self.headval = None

    def __repr__(self):
        return "Head: {}".format(self.headval)
```

[27]:
```
list1 = SlinkedList()
```

[28]:
```
list1
```

[28]:
```
Head: None
```

[29]:
```
list1.headval = Node('Mon')
```

[30]:
```
list1
```

[30]:
```
Head: Data: Mon, Next: (None)
```

[31]:
```
list1.headval.nextval = Node('Tue')
```

[32]:
```
list1
```

[32]:
```
Head: Data: Mon, Next: (Data: Tue, Next: (None))
```

[33]:
```
e1 = Node('Wed')
```

[34]:
```
e2 = Node('Thu')
```

[35]:
```
list1.headval.nextval = e1
```

[36]:
```
list1
```

[36]:
```
Head: Data: Mon, Next: (Data: Wed, Next: (None))
```

[37]:
```
e1.nextval = e2
```

[38]:
```
list1
```

[38]:
```
Head: Data: Mon, Next: (Data: Wed, Next: (Data: Thu, Next: (None)))
```

### 5.0.1 Traversing a linked list

Singly linked lists can be traversed in only forward direction starting from the first data element. We simply print the value of the next data element by assigning the pointer of the next node to the current data element.

[39]:
```
class Node:
    def __init__(self, dataval = None):
        self.dataval = dataval
        self.nextval = None

class SlinkedList:
```

```
    def __init__(self):
        self.headval = None

    def listprint(self):
        printval = self.headval
        while printval:
            print(printval.dataval)
            printval = printval.nextval
```

[40]:
```
mylist = SlinkedList()
e1 = Node('Mon')
e2 = Node('Tue')
e3 = Node('Wed')
e4 = Node('Thu')
```

[41]:
```
mylist.headval = e1
```

[42]:
```
e1.nextval = e2
```

[43]:
```
e2.nextval = e3
```

[44]:
```
e3.nextval = e4
```

[45]:
```
mylist.listprint()
```

```
Mon
Tue
Wed
Thu
```

### 5.0.2 Insertion in a linked list

#### Inserting at the beginning of the linked list

This involves pointing the next pointer of the new data node to the current head of the linked list. So the current head of the linked list becomes the second data element and the new node becomes the head of the linked list.

[46]:
```
class Node:
    def __init__(self,dataval = None):
        self.dataval = dataval
        self.nextval = None
class SlinkedList:
    def __init__(self):
        self.headval = None

    def listprint(self):
```

```
        printval = self.headval
        while printval:
            print(printval.dataval)
            printval = printval.nextval
    def insert_at_beginning(self,newNode):
        newNode.nextval = self.headval
        self.headval = newNode
```

[47]:
```
mylist = SlinkedList()
e1 = Node('Mon')
e2 = Node('Tue')
e3 = Node('Wed')
e4 = Node('Thu')
```

[48]:
```
mylist.headval = e1
```

[49]:
```
e1.nextval = e2
```

[50]:
```
e2.nextval = e3
```

[51]:
```
mylist.listprint()
```

```
Mon
Tue
Wed
```

[52]:
```
mylist.insert_at_beginning(e4)
```

[53]:
```
mylist.listprint()
```

```
Thu
Mon
Tue
Wed
```

### Inserting at the End of the linked list

This involves pointing the next pointer of the current last node of the linked list to the new data node. So the current last node of the linked list becomes the second last data node and the new node becomes the last node of the linked list.

[54]:
```
class Node:
    def __init__(self, dataval = None):
        self.dataval = dataval
        self.nextval = None

class SlinkedList:
    def __init__(self):
```

```
        self.headval = None

    def listprint(self):
        printval = self.headval
        while printval:
            print(printval.dataval)
            printval = printval.nextval
    def insert_at_ending(self,newNode):
        lastval = self.headval
        while lastval.nextval:
            lastval = lastval.nextval
        lastval.nextval = newNode
```

[55]:
```
mylist = SlinkedList()
e1 = Node('Mon')
e2 = Node('Tue')
e3 = Node('Wed')
e4 = Node('Thu')
mylist.headval = e1
e1.nextval = e2
e2.nextval = e3
```

[56]:
```
mylist.listprint()
```

```
Mon
Tue
Wed
```

[57]:
```
mylist.insert_at_ending(e4)
```

[58]:
```
mylist.listprint()
```

```
Mon
Tue
Wed
Thu
```

### Inserting in between two Data Nodes

It involves changing the pointer of a specific node to point to the new node. That is possible by passing in both the new node and the exiting node after which the new node will be inserted. So we define an additional class which will change the next pointer of the new node to the next pointer of middle node. Then assign the new node to next pointer of the middle node.

[59]:
```
class Node:
    def __init__(self,dataval = None):
        self.dataval = dataval
        self.nextval = None
```

```
class SlinkedList:
    def __init__(self):
        self.headval = None

    def listprint(self):
        printval = self.headval
        while printval:
            print(printval.dataval)
            printval = printval.nextval
    def insert_at_middle(self,middleNode,newNode):
        if middleNode is None:
            print("The mentioned node is absent")
        newNode.nextval = middleNode.nextval
        middleNode.nextval = newNode
```

[60]:
```
mylist = SlinkedList()
e1 = Node('Mon')
e2 = Node('Tue')
e3 = Node('Wed')
e4 = Node('Thu')
mylist.headval = e1
e1.nextval = e2
```

[61]:
```
mylist.listprint()
```

```
Mon
Tue
```

[62]:
```
mylist.insert_at_middle(e1,e3)
```

[63]:
```
mylist.listprint()
```

```
Mon
Wed
Tue
```

### 5.0.3 Removing an item from a linked list

We can remove an existing node using the key for that node. In the below program we locate the previous node of the node which is to be deleted. Then point the next pointer of this node to the next node of the node to be deleted

[64]:
```
class Node:
    def __init__(self,dataval = None):
        self.dataval = dataval
        self.nextval = None
class SlinkedList:
```

```
    def __init__(self):
        self.headval = None

    def listprint(self):
        printval = self.headval
        while printval:
            print(printval.dataval)
            printval = printval.nextval

    def insert_at_beginning(self,newNode):
        newNode = Node(newNode)
        newNode.nextval = self.headval
        self.headval = newNode

    def remove_node(self, remove_key):
        headval = self.headval

        if headval:
            if headval.dataval == remove_key:
                self.headval = headval.next
                headval = None
                return

        while headval:
            if headval.dataval == remove_key:
                break
            prev = headval
            headval = headval.nextval

        if headval == None:
            return
        prev.nextval = headval.nextval

        headval = None
```

[65]:
```
mylist = SlinkedList()
mylist.insert_at_beginning('Mon')
mylist.insert_at_beginning('Tue')
mylist.insert_at_beginning('Wed')
mylist.insert_at_beginning('Thu')
mylist.listprint()
```

```
Thu
Wed
Tue
Mon
```

[66]:
```
mylist.remove_node('Tue')
```

[67]:
```
mylist.listprint()
```

```
Thu
Wed
Mon
```

# 6 Advanced linked list (doubly linked list)

### Features of doubly linked list

- Contains a link element called first and last.
- Each link carries a data and tow links called next and prev
- Each link is linked with its next link using its next link
- Eack link is linked with its previous link using its prev link
- The last link carries a link as null to mark the end of the list

### 6.0.1 Creating doubly linked list

[68]:
```
class Node:
    def __init__(self,data = None):
        self.data = data
        self.next = None
        self.prev = None

class DlinkedList:
    def __init__(self):
        self.head = None

    def push(self,new_val):
        new_node = Node(new_val)
        new_node.next = self.head
        if self.head:
            self.head.prev = new_node
        self.head = new_node

    def insert(self, prev_node, NewVal):
        if prev_node is None:
            return
        NewNode = Node(NewVal)
        NewNode.next = prev_node.next
        prev_node.next = NewNode
        NewNode.prev = prev_node
        if NewNode.next is not None:
```

```
            NewNode.next.prev = NewNode


    def listprint(self):
        printval = self.head
        while printval:
            print(printval.data)
            printval = printval.next
```

[69]:
```
my_list = DlinkedList()
my_list.push(12)
my_list.push(8)
my_list.push(62)
```

[70]:
```
my_list.listprint()
```

```
62
8
12
```

[71]:
```
my_list.insert(my_list.head.next,13)
```

[72]:
```
my_list.listprint()
```

```
62
8
13
12
```

### 6.0.2 Appending a doubly linked list

[73]:
```
class Node:
    def __init__(self,data = None):
        self.data = data
        self.next = None
        self.prev = None

class DlinkedList:
    def __init__(self):
        self.head = None

    def push(self,new_val):
        new_node = Node(new_val)
        new_node.next = self.head
        if self.head:
            self.head.prev = new_node
```

```
        self.head = new_node

    def append(self,new_val):
        new_node = Node(new_val)
        new_node.next = None

        if self.head is None:
            new_node.prev = None
            self.head = new_node
            return

        last = self.head

        while last.next:
            last = last.next
        last.next = new_node
        new_node.prev = last
        return

    def listprint(self):
        printval = self.head
        while printval:
            print(printval.data)
            printval = printval.next
```

[74]:
```
my_list = DlinkedList()
my_list.push(12)
my_list.push(8)
my_list.push(62)
```

[75]:
```
my_list.listprint()
```

```
62
8
12
```

[76]:
```
my_list.append(45)
```

[77]:
```
my_list.listprint()
```

```
62
8
12
45
```

# 7 Stack

In english dictionary the word stack means arranging objects one over another. It is the same way memory is allocated in this data structure. It stores teh data elements in a similar fashion as a bunch of plates are stored one above anotehr in the kitchen. so stack data structure allows operations at one end which can be called top of the stack. We can add element or remove elements only from this end of the stack.

In a stack the element inserted last in sequence will come out first as we can remove only from the top of the stack. Such feature is known as Last in First Out (LIFO) feature. The operations of adding and removing the elements is known as **PUSH** and **POP**.

### 7.0.1 Push into a stack

[78]:
```
class Stack:
    def __init__(self):
        self.stack = []

    def add(self,data):
        if data not in self.stack:
            self.stack.append(data)

    def peek(self):
        return self.stack[-1]
```

[79]:
```
my_stack = Stack()
my_stack.add("Mon")
my_stack.add("Tue")
my_stack.peek()
```

[79]:
```
'Tue'
```

[80]:
```
my_stack.add("Wed")
my_stack.add("Thu")
my_stack.peek()
```

[80]:
```
'Thu'
```

### 7.0.2 Pop from stack

[81]:
```
class Stack:
    def __init__(self):
        self.stack = []

    def add(self,data):
```

```
        if data not in self.stack:
            self.stack.append(data)

    def peek(self):
        return self.stack[-1]

    def remove(self):
        if len(self.stack) <= 0:
            print("No stack exits.")
        else:
            self.stack.pop()
```

[82]:
```
my_stack = Stack()
my_stack.add("Mon")
my_stack.add("Tue")
my_stack.peek()
```

[82]: 'Tue'

[83]:
```
my_stack.remove()
```

[84]:
```
my_stack.peek()
```

[84]: 'Mon'

**How inbuild pop function works**

[85]:
```
a = [1,2,3]
```

[86]:
```
a.pop()
```

[86]: 3

[87]:
```
a
```

[87]: [1, 2]

# 8 Queue

The queue data structure means the data elements are arranged in a queue. The uniqueness of queue lies in the way items are added and removed. The items are allowed at on end but removd from the other end. So it is a First-in-First out method. An queue can be implemented using python list where we can use the insert() and pop methods to add and remove elements. There is no insertiion as data elements are always added at the end of the queue.

### 8.0.1 adding element in queue

[88]:
```
class Queue:
    def __init__(self):
        self.queue = []

    def add(self,data):
        if data not in self.queue:
            self.queue.insert(0,data)
    def size(self):
        return len(self.queue)
```

[89]:
```
my_queue = Queue()
my_queue.add("Mon")
my_queue.add("Tue")
my_queue.add("Wed")
my_queue.size()
```

[89]: 3

### 8.0.2 Removing element from queue

[90]:
```
class Queue:
    def __init__(self):
        self.queue = []

    def add(self,data):
        if data not in self.queue:
            self.queue.insert(0,data)
    def remove(self):
        self.queue.pop()

    def size(self):
        return len(self.queue)

    def __repr__(self):
        return "Queue([{}])".format(",".join(self.queue))
```

[91]:
```
my_queue = Queue()
my_queue.add("Mon")
my_queue.add("Tue")
my_queue.add("Wed")
my_queue.size()
```

[91]: 3

[92]:
```
my_queue
```

[92]: Queue([Wed,Tue,Mon])

[93]:
```
my_queue.remove()
```

[94]:
```
my_queue.size()
```

[94]: 2

[95]:
```
my_queue
```

[95]: Queue([Wed,Tue])

## 9 Dequeue

A double-ended queue or dequeue supports adding and removing elements from either end. The more commonly used stacks and queue are degenerate of deques, where the inputs and outputs are restricted to a single end.

[96]:
```
from collections import deque

my_deque = deque(["Mon","Tue","Wed"])
my_deque.append("Thu")
my_deque.appendleft("Sun")
my_deque
```

[96]: deque(['Sun', 'Mon', 'Tue', 'Wed', 'Thu'])

[97]:
```
my_deque.pop()
my_deque
```

[97]: deque(['Sun', 'Mon', 'Tue', 'Wed'])

[98]:
```
my_deque.popleft()
```

[98]: 'Sun'

[99]:
```
my_deque
```

[99]: deque(['Mon', 'Tue', 'Wed'])

# 10 Hash Table

Hash tables are a type of data structure in which the address or the index value of the data element is generated from a hash function. That makes accessing the data faster as the index value behaves as a key for the data value. In other words, Hash tables stores key-value pairs but the key is generated through a hasing function.

So, the search and insertion function of a data element becomes much faster as the key values themselves become the index of the array which stores the data.

In python, the dictionary data types represent the implementation of hash tables. The keys in the dictionary satisfy the following requirements.

- The keys of the dictionary are hashable. i.e. the age generated by hashing function which generates unique result for each unique value supplied to the hash function.
- The order of data elements in a dictionary is not fixed.

We implement hash table by using dictionary data types.

# 11 Binary Tree

Tree represents the nodes connected by edges. It is a non-linear data structure. It has the following properties.

- One node is marked as root node.
- Every node other than the root is associated withone parent node.
- Each node can have an arbitrary number of child node.

we create a tree data structure in python by using the concept of node. We designate one node as root node and then add more nodes as child nodes.

### 11.0.1 Create root

[100]:
```
class Tree:
    def __init__(self,node):
        self.node = node
        self.right = None
        self.left = None

    def __repr__(self):
        return "Tree (Node: {},Right: {},Left: {})".format(self.node,self.
 ↪right,self.left)
```

[101]:
```
my_tree = Tree(10)
```

[102]:
```
my_tree.right = Tree(5)
```

[103]:
```
my_tree.left = Tree(2)
```

[104]:
```
my_tree
```

[104]:
```
Tree (Node: 10,Right: Tree (Node: 5,Right: None,Left: None),Left: Tree (Node:
2,Right: None,Left: None))
```

### 11.0.2 Inserting into a Tree

[105]:
```
class Tree:
    def __init__(self,node):
        self.node = node
        self.right = None
        self.left = None

    def insert(self,node):
        if self.node:
            if node < self.node:
                if self.left is None:
                    self.left = Tree(node)
                else:
                    self.left.insert(node)
            elif node > self.node:
                if self.right is None:
                    self.right = Tree(node)
                else:
                    self.right.insert(node)
        else:
            self.node = node

    def __repr__(self):
        return "Node: {}, (Right: {}, Left: {})".format(self.node,self.
↪right,self.left)
```

[106]:
```
my_tree = Tree(12)
```

[107]:
```
my_tree
```

[107]:
```
Node: 12, (Right: None, Left: None)
```

[108]:
```
my_tree.insert(6)
```

[109]:
```
my_tree
```

[109]:
```
Node: 12, (Right: None, Left: Node: 6, (Right: None, Left: None))
```

## 12 Search Tree

A binary search tree (BST) is a tree in which all the nodes follow the below-mentioned properties * the left sub-tree of a node has a key les than or equal to its parent node'e key. * the right sub-tree of a noe has a key greater than to tits parent node's key.

searching for a value in a tree involves comparing the incoming value with the value exiting nodes. We travers the nodes from left to right and then finally with the parent. If the searched for value does not match any of the existing vlaue, then we return not found message else the found message is returned.

[110]:
```
class Tree:
    def __init__(self,node):
        self.node = node
        self.right = None
        self.left = None

    def insert(self,node):
        if self.node:
            if node < self.node:
                if self.left is None:
                    self.left = Tree(node)
                else:
                    self.left.insert(node)
            elif node > self.node:
                if self.right is None:
                    self.right = Tree(node)
                else:
                    self.right.insert(node)
        else:
            self.node = node

    def search(self,value):
        if value < self.node:
            if self.left is None:
                return str(value)+" Not Found"
            return self.left.search(value)
        elif value > self.node:
            if self.right is None:
                return str(value)+" Not Found"
            return self.right.search(value)
        else:
            print(str(value)+" is found")

    def __repr__(self):
        return "Node: {}, (Right: {}, Left: {})".format(self.node,self.
↪right,self.left)
```

[111]:
```
my_tree = Tree(12)
my_tree.insert(6)
my_tree.insert(14)
my_tree.insert(3)
```

[112]:
```
my_tree
```

[112]:
```
Node: 12, (Right: Node: 14, (Right: None, Left: None), Left: Node: 6, (Right:
None, Left: Node: 3, (Right: None, Left: None)))
```

[113]:
```
my_tree.search(7)
```

[113]:
```
'7 Not Found'
```

[114]:
```
my_tree.search(14)
```

```
14 is found
```

# 13 Heaps

Heap is a special tree structue in which each parent node is less than or equal to its child node. Then it is called a Min Heap. If each parent node is greater than or equal to its child node then it is called Max heap.

### 13.0.1 Create a Heap

A heap is created by using python's inbuild librayry named **heapq**. This library has the relevent functions to carry out various operations on heap data structure. * heapify => This function converts a regular list to a heap. In the resulting heap the smallest element gets pushed to the index position 0. But rest of the data elements are not necessarily sorted. * heappush => This function adds an element to the heap without altering the current heap. * heappop => This function returns the smallest data element from the heap. * heapreplace => This function replaces the smallest data element with a new value supplied in the function.

[115]:
```
import heapq

H = [21,1,45,78,3,5]
heapq.heapify(H)
print(H)
```

```
[1, 3, 5, 78, 21, 45]
```

[116]:
```
heapq.heappush(H,8)
print(H)
```

```
[1, 3, 5, 78, 21, 45, 8]
```

[117]:
```
heapq.heappop(H)
print(H)
```

```
[3, 8, 5, 78, 21, 45]
```

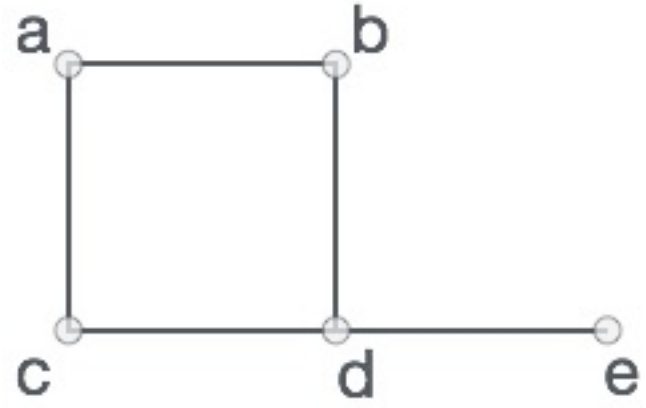
[118]:
```
heapq.heapreplace(H,6)
print(H)
```

```
[5, 8, 6, 78, 21, 45]
```

## 14 Graphs

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices and the links that connect the vertices are called edges. Following are the basic operations we perform on graphs. * Display graph verices * Display graph edges * Add a vetex * Add an edge * Creating a graph

A graph can be easily presented using the python dictionary data types. We represent the verteices as keys of the dictionary and the connection between the vertices also called deges athe vlaues in the dictionary.




Vortices = {a,b,c,d,e} Edges = {ab,ac,bd,cd,de}

[119]:
```
graph = { "a" : ["b","c"],
          "b" : ["a", "d"],
          "c" : ["a", "d"],
          "d" : ["e"],
          "e" : ["d"]
        }
```

[120]:
```
graph
```

[120]:
```
{'a': ['b', 'c'], 'b': ['a', 'd'], 'c': ['a', 'd'], 'd': ['e'], 'e': ['d']}
```

### 14.0.1 Display graph vortices and edges

[121]:
```
class Graph:
    def __init__(self,graph=dict()):
        self.graph = graph

    def get_vortices(self):
        return list(self.graph.keys())

    def __edgenames(self): # To make it not accssible by a class instance.
        edges = []
        for vortex in self.graph:
            for next_vortex in self.graph[vortex]:
                if {vortex,next_vortex} not in edges:
                    edges.append({vortex,next_vortex})
        return edges

    def get_edges(self):
        return self.__edgenames()

    def add_vortex(self,vortex):
        if vortex not in self.graph:
            self.graph[vortex] = []

    def add_edge(self,edge):
        edge = set(edge)
        (vortex,next_vortex) = tuple(edge)
        if vortex in self.graph:
            self.graph[vortex].append(next_vortex)
        else:
            self.graph[vortex] = [next_vortex]


    def __repr__(self):
        return Graph.__name__
```

[122]:
```
elements = { "a" : ["b","c"],
          "b" : ["a", "d"],
          "c" : ["a", "d"],
          "d" : ["e"],
          "e" : ["d"]
         }
my_graph = Graph(elements)
```

[123]:
```
my_graph.get_vortices()
```

[123]:
```
['a', 'b', 'c', 'd', 'e']
```

### 14.0.2 Display graph edges

Finding the graph deges is little tricker than the vertices as w have to find each of the pairs of vertices which have an edge in between them. So we crate an empty list of edges then iterate through the edge values associated with each of the vertices.

[124]:
```
my_graph.get_edges()
```

[124]:
```
[{'a', 'b'}, {'a', 'c'}, {'b', 'd'}, {'c', 'd'}, {'d', 'e'}]
```

Try to access hidden method

[125]:
```
my_graph._Graph__edgenames()
```

[125]:
```
[{'a', 'b'}, {'a', 'c'}, {'b', 'd'}, {'c', 'd'}, {'d', 'e'}]
```

[126]:
```
my_graph.__edgenames()
```

```
␣
↪---------------------------------------------------------------------------

       AttributeError                                Traceback (most recent call␣
↪last)

       <ipython-input-126-763060aea40b> in <module>
   ----> 1 my_graph.__edgenames()


       AttributeError: 'Graph' object has no attribute '__edgenames'
```

### 14.0.3 Adding a vertex end edges

[127]:
```
my_graph.add_vortex('f')
```

[128]:
```
my_graph.get_vortices()
```

[128]:
```
['a', 'b', 'c', 'd', 'e', 'f']
```

[129]:
```
my_graph.add_edge({'a','e'})
```

[130]:
```
my_graph.get_edges()
```

[130]:
```
[{'a', 'b'}, {'a', 'c'}, {'a', 'e'}, {'b', 'd'}, {'c', 'd'}, {'d', 'e'}]
```