Array

An array is a data structure that stores elements of the same type in contiguous memory locations. Each element in an array is identified by an index or a key. Arrays provide a straightforward way to represent and manipulate collections of data.

Random Access:

- Elements in an array can be directly accessed using their index. This allows for constant-time access to any element, making random access very efficient.

Memory Efficiency:

- Arrays are memory-efficient due to their contiguous memory allocation. The elements are stored in adjacent locations, which reduces memory overhead.

Simple Implementation:

- Arrays are easy to understand and implement. The concept of indexing and direct access simplifies the manipulation of data.

Efficiency, Cost, and Time Complexity:

Efficiency:

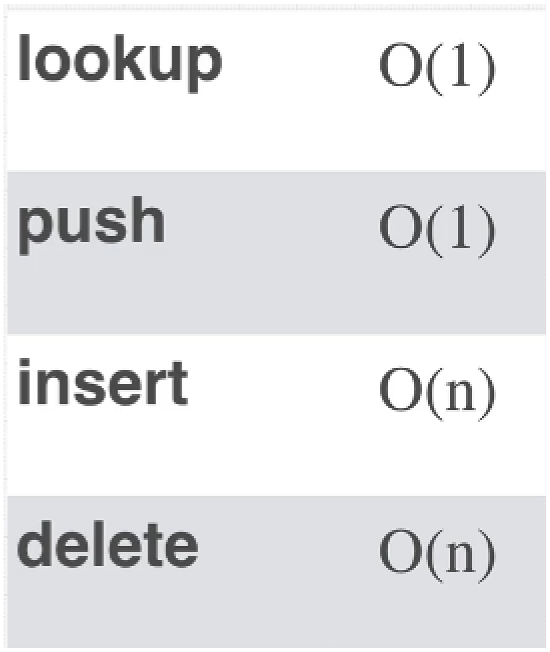
- Arrays are efficient for random access and simple operations. Accessing an element by index is fast (O(1)).

Cost:

- Arrays are generally cost-effective in terms of memory usage, especially when the size is known in advance.

Time Complexity:

- Accessing an element in an array has a time complexity of O(1). However, insertion and deletion operations in the middle of the array have a time complexity of O(n) due to the need to shift elements.



lookup	O(1)
push	O(1)
insert	O(n)
delete	O(n)

In [100…

```python
class MyCustomList:

    def __init__(self):
        self.length = 0
        self.data = {}

    def get(self, index):
        """Lookup | O(1)"""
        if not self.length:
            return
        return self.data[index]

    def push(self, item):
        """Push | O(1)"""
        self.data[self.length] = item
        self.length += 1

    def pop(self):
        """Pop | O(1)"""
        last_item = self.data[self.length-1]
        del self.data[self.length-1]
        self.length -= 1
        return last_item

    def delete(self, index):
        """Delete | O(n)"""
        self._shift_items(index)

    def insert(self, index, value):
        """Insert | O(n)"""
        for i in range(self.length, index, - 1): # Make a room for the new value
            self.data[i] = self.data[i - 1]
        self.data[index] = value
        self.length += 1

    def _shift_items(self, index):
        for i in range(index, self.length - 1):
            self.data[i] = self.data[i+1]
        del self.data[self.length-1]
        self.length -= 1

    def __repr__(self):
        return repr(self.data)

custom_list = MyCustomList()
custom_list.push('hi')
custom_list.push('how')
custom_list.push('are')
custom_list.push('you?')

custom_list.delete(0)
custom_list.delete(1)
custom_list.delete(2)
custom_list.delete(3)

custom_list.insert(0, 'whats')
custom_list.insert(1, 'up')
custom_list.insert(2, 'dog')

print(custom_list)
```

```
{0: 'whats', 1: 'up', 2: 'dog'}
```

Hash Map

A hash table, also known as a hash map, is a data structure that allows the efficient storage and retrieval of key-value pairs. It is based on the idea of using a hash function to map keys to indices in an array, where the associated values are stored. This mapping allows for constant-time average case complexity for search, insert, and delete operations.

Key Components:

- Array: Hash tables use an array to store data. Each index in the array corresponds to a "bucket" where key-value pairs may be stored.
- Hash Function: A hash function takes a key as input and returns an index in the array where the corresponding value should be stored. The goal is to distribute the keys uniformly across the array, minimizing collisions (two keys mapping to the same index).
- Collision Resolution: Collisions occur when two keys hash to the same index. Various techniques, such as chaining (using linked lists at each index) or open addressing (finding the next available slot in the array), are employed to resolve collisions.

Operations:

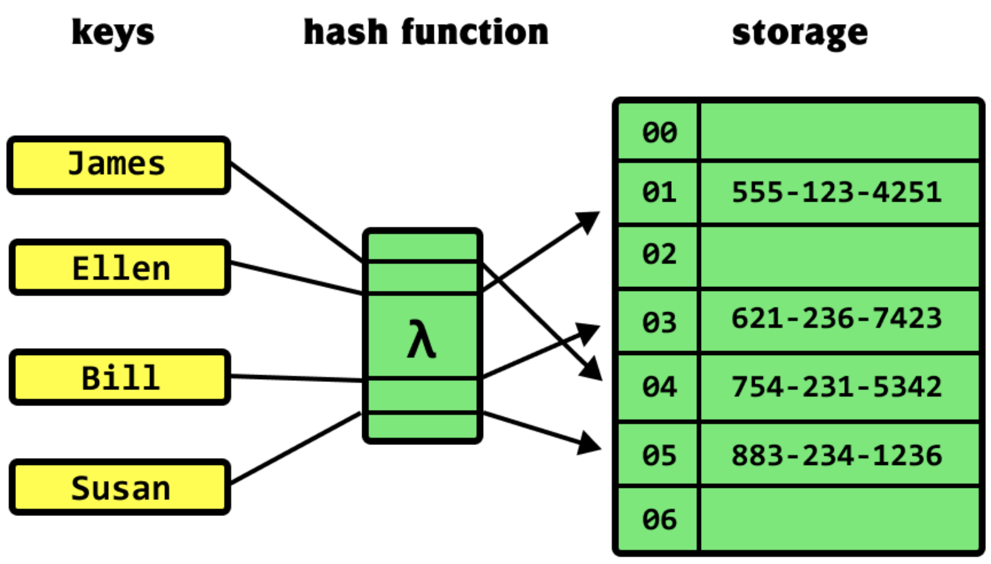
- Insertion (or Update): Compute the hash of the key, find the corresponding index in the array, and insert the key-value pair at that index.
- Retrieval: Given a key, compute its hash, find the corresponding index, and retrieve the associated value. This operation has an average-case time complexity of O(1).
- Deletion: Similar to retrieval, compute the hash of the key, find the corresponding index, and remove the key-value pair from that index.

Advantages:

- Fast Average-Case Performance: Hash tables provide constant-time average-case complexity for key-based operations, making them highly efficient for search, insert, and delete operations.
- Flexible Key Types: Hash tables can handle a wide range of key types, including integers, strings, and more complex objects.

Challenges:

- Collisions: Collisions can occur when two keys hash to the same index. Efficient collision resolution strategies are crucial for maintaining performance.
- Deterministic Hash Function: For hash tables to work effectively, the hash function should be deterministic, meaning the same input key will always produce the same hash value.
- Memory Usage: Hash tables may consume more memory than other data structures due to the need for an array large enough to accommodate potential keys.



- **Time Complexity:**

insert	$O(1)$
lookup	$O(1)$
delete	$O(1)$
search	$O(1)$

Simple Hash Function

In [1]:
```python
def get_hash(key):
    h = 0
    for char in key:
        h += ord(char)
    return h % 100

get_hash('march 6')
```

Out[1]:
```
9
```

Simple Hash Table

In [4]:
```python
class HashTable:
    """Hash Table | O(1)"""

    def __init__(self):
        self.MAX = 100
        self.arr = [None for i in range(self.MAX)]

    def get_hash(self, key):
        h = 0
        for char in key:
            h += ord(char)
        return h % self.MAX

    def __setitem__(self, key, val):
        h = self.get_hash(key)
        self.arr[h] = val

    def __getitem__(self, key):
        h = self.get_hash(key)
        return self.arr[h]

    def __delitem__(self, key):
        h = self.get_hash(key)
        self.arr[h] = None
```

Extracting the hash value from a key

```
In [5]: t = HashTable()
        t.get_hash('march 6')
```

```
Out[5]: 9
```

Creating key and value for hash table

```
In [ ]: t['march 6'] = 130
        t['march 1'] = 20
        t.arr
```

Accessing value from key

```
In [19]: t['march 6']
```

```
Out[19]: 130
```

Handling collisions

Collision handling in hash maps refers to the situation where two or more keys hash to the same location, causing a conflict. In simple terms, it's like having multiple keys trying to occupy the same slot in a storage space.

To handle collisions, there are different strategies:

1. Chaining: Instead of storing a single value in a hash table slot, we store a linked list of values. When a collision occurs, new elements are added to the linked list at that slot.

2. Open Addressing: In this approach, when a collision happens, the algorithm looks for the next available slot in the vicinity. It might be the next slot or found by a specific probing sequence.

These methods help ensure that even if multiple keys hash to the same location, the hash map can still store and retrieve the values associated with each key. It's like having a backup plan for when two or more keys want to occupy the same address in the hash map.

Chaining

```
In [127…: class HashTable2:
    """Hash Table with Chaining | O(n) or O(1)"""

    def __init__(self):
        self.MAX = 10
        self.arr = [[] for i in range(self.MAX)]

    def get_hash(self, key):
        h = 0
        for char in key:
            h += ord(char)
        return h % self.MAX

    def __setitem__(self, key, val):
        h = self.get_hash(key)
        found = False
        for idx, element in enumerate(self.arr[h]):
            if len(element) == 2 and element[0] == key:
                self.arr[h][idx] = (key, val)
                found = True
        if not found:
            self.arr[h].append((key, val))

    def __getitem__(self, key):
        h = self.get_hash(key)
        if self.arr[h]:
            for element in self.arr[h]:
                if element[0] == key:
                    return element[1]

    def __delitem__(self, key):
        h = self.get_hash(key)
        for idx, element in enumerate(self.arr[h]):
            if element[0] == key:
                del self.arr[h][idx]
        self.arr[h] = None
```

In [129…

```
hash_table = HashTable2()
hash_table['abcd'] = 12
hash_table['dbac'] = 15

hash_table.arr
```

Out[129]:

```
[[], [], [], [], [('abcd', 12), ('dbac', 15)], [], [], [], [], []]
```

Linear Probing | Open Addressing

Solution 1

In [152…

```
class HashTable3:
    """Hash Table with linear probing | O(n) or O(1)"""

    def __init__(self):
        self.MAX = 10
        self.arr = [[] for i in range(self.MAX)]
        self.position = 0

    def get_hash(self, key):
        h = 0
        for char in key:
            h += ord(char)
        return h % self.MAX

    def __setitem__(self, key, val):
        h = self.get_hash(key)
        filled = False
        while h < self.MAX:
            if self.arr[h]:
                if all(sub for sub in self.arr):
                    raise Exception("HashMap Full")

                if key in self.arr[h]:
                    break

                filled = True
                h += 1
                if h == self.MAX:
                    h = 0

            else:
                filled = False
                break

        if not filled:
            self.arr[h] = (key, val)


    def __getitem__(self, key):
        h = self.get_hash(key)
        if self.arr[h]:
            for element in self.arr:
                if element[0][0] == key:
                    return element[1]

    def __delitem__(self, key):
        h = self.get_hash(key)
        for idx, element in enumerate(self.arr):
            if element[0][0] == key:
                del self.arr[h][idx]
        self.arr[h] = None
```

In [153…

```
hash_table = HashTable3()
hash_table["abcxd"] = 1
hash_table["bxdca"] = 9
hash_table["caxdb"] = 3
hash_table["adxbc"] = 2
hash_table["cbdax"] = 11
hash_table["xabdc"] = 13
hash_table["cxbda"] = 15

hash_table.arr
```

Out[153]:

```
[('cxbda', 15),
 [],
 [],
 [],
 ('abcxd', 1),
 ('bxdca', 9),
 ('caxdb', 3),
 ('adxbc', 2),
 ('cbdax', 11),
 ('xabdc', 13)]
```

Solution 2

In [155…

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

def get_prob_range(index, arr):
    return [*range(index, len(arr))] + [*range(0,index)]

print(get_prob_range(5, arr))
```

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

In [164…

```
class HashTable4:
    """Hash Table with linear probing | O(n) or O(1)"""

    def __init__(self) -> None:
```

```python
        self.MAX = 10
        self.arr = [None for i in range(self.MAX)]

    def __setitem__(self, key, value) -> None:
        h = self._get_hash(key)
        if self.arr[h] is None:
            self.arr[h] = (key, value)
        else:
            new_h = self._find_slot(key, h)
            self.arr[new_h] = (key, value)

    def __getitem__(self, key):
        h = self._get_hash(key)
        if self.arr[h] is None:
            return
        prob_range = self._get_prob_range(key)
        for idx in prob_range:
            element = self.arr[idx]
            if element is None:
                return
            if element[0] == key:
                return element[1]

    def __delitem__(self, key):
        h = self._get_hash(key)
        prob_range = self._get_prob_range(h)
        for idx in prob_range:
            if self.arr in prob_range:
                if self.arr[prob_range] is None:
                    raise Exception("Key not found")
                if self.arr[idx][0] == key:
                    self.arr[idx] = None
        print(self.arr)

    def _get_hash(self, key):
        hash = 0
        for idx in key:
            hash += ord(idx)
        return hash % self.MAX

    def _get_prob_range(self, index):
        return [*range(index, len(self.arr))] + [*range(0,index)]

    def _find_slot(self, key, h):
        prob_range = self._get_prob_range(h)
        for idx in prob_range:
            if self.arr[idx] is None:
                return idx
            if self.arr[idx][0] == key:
                return idx
        raise Exception("HashMap Full")
```

In [166…

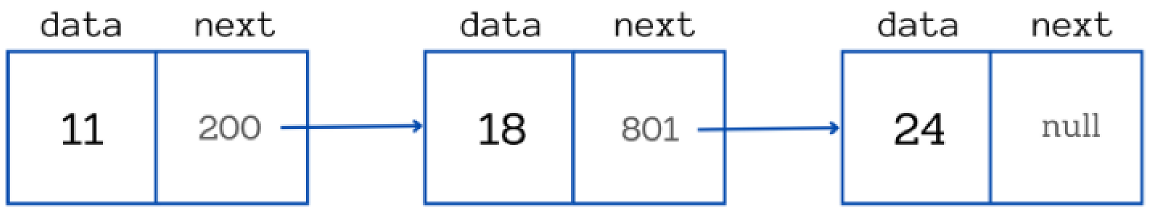
```python
hash_table = HashTable4()
hash_table["abcxd"] = 1
hash_table["bxdca"] = 9
hash_table["caxdb"] = 3
hash_table["adxbc"] = 2
hash_table["cbdax"] = 11
hash_table["xabdc"] = 13
hash_table["cxbda"] = 17
```

```
hash_table.arr
```

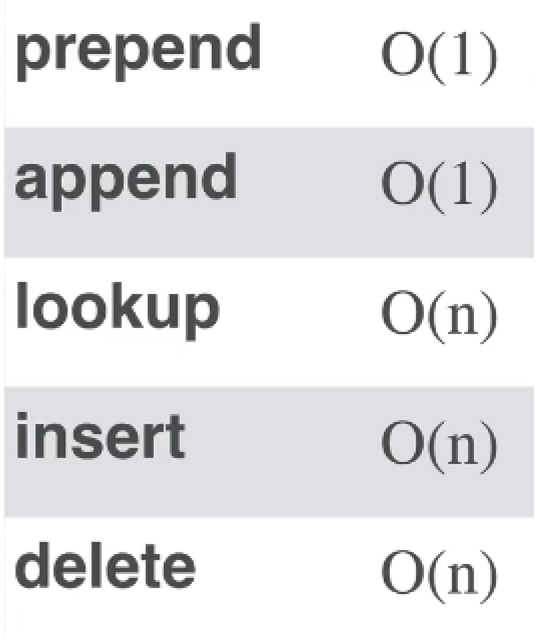
```
Out[166]: [('cxbda', 17),
           None,
           None,
           None,
           ('abcxd', 1),
           ('bxdca', 9),
           ('caxdb', 3),
           ('adxbc', 2),
           ('cbdax', 11),
           ('xabdc', 13)]
```

Linked Lists

Linked list is a linear and dynamic data structure. It is composed of a sequence of nodes or cells that contain their data and also one or two pointers ("links") pointing to the previous or next node. Linked lists are useful for representing dynamic sets of data. In other words, you do not need to define a maximum size for a linked list.



Time Complexity:



prepend	O(1)
append	O(1)
lookup	O(n)
insert	O(n)
delete	O(n)

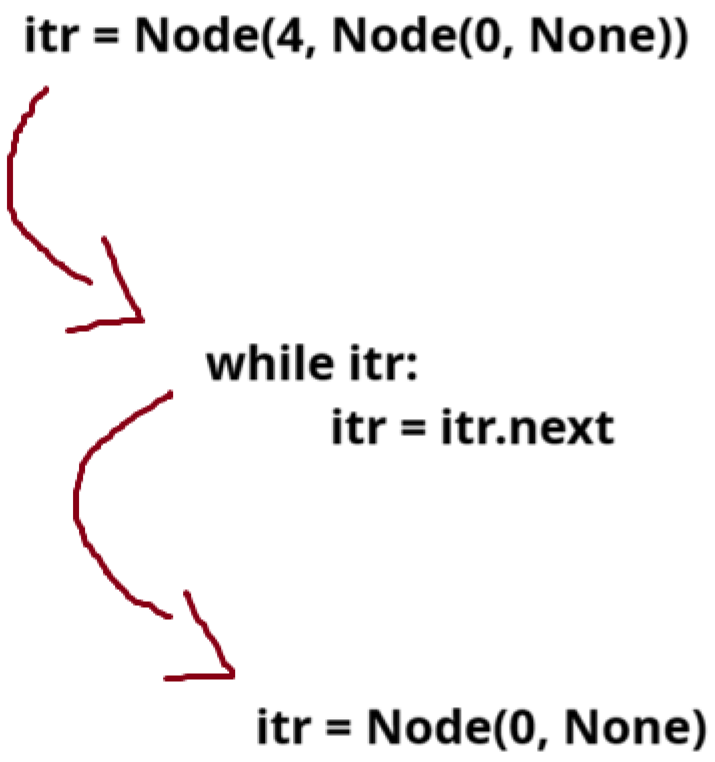
Simple Linked List

In [41]:

```python
class Node:
    def __init__(self, data=None, next=None):
        self.data = data
        self.next = next

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

    def add(self, data):
        node = Node(data, self.head)
        self.head = node
        # head -> Node(4, Node(0, None))

    def print(self):
        itr = self.head
        llstr = ''
        while itr:
            llstr += str(itr.data) + ' --> '
            itr = itr.next
        print(llstr)

ll = LinkedList()
ll.add(0)
ll.add(4)
ll.print()
```

```
4 --> 0 -->
```

Node object iteration process:



```python
itr = Node(4, Node(0, None))

while itr:
        itr = itr.next

itr = Node(0, None)
```

Linked List | Solution 1

In [51]:
```python
class Node:
    def __init__(self, d, n) -> None:
        self.data = d
        self.next = n

class LinkedList:
    def __init__(self) -> None:
        self.head = None

    def prepend(self, value):
        """O (1)"""
        node = Node(value, self.head)
        self.head = node

    def append(self, value):
        """O (n)"""
        if self.head is None:
            self.head = Node(value, None)
            return

        itr = self.head
        while itr.next:
            itr = itr.next

        itr.next = Node(value, None)

    @property
    def size(self):
        """O (n)"""
        if self.head is None:
            return None

        count = 0
        itr = self.head
        while itr:
            itr = itr.next
            count += 1
        return count

    def __repr__(self):
        """Lookup | O(n)"""
        if self.head is None:
            return "Gatcha!"

        acc_items = ()
        itr = self.head
        while itr:
            acc_items += (itr.data,)
            itr = itr.next
        return repr(acc_items)

    def insert(self, index, value):
        """O (n)"""
        if index < 0 or index > self.size:
            raise Exception('Invalid Index')

        if index == 0:
            self.prepend(value)

        count = 0
        itr = self.head
        while itr:
            if count == index - 1:
                node = Node(value, itr.next)
                itr.next = node
                break

            itr = itr.next
            count += 1
```

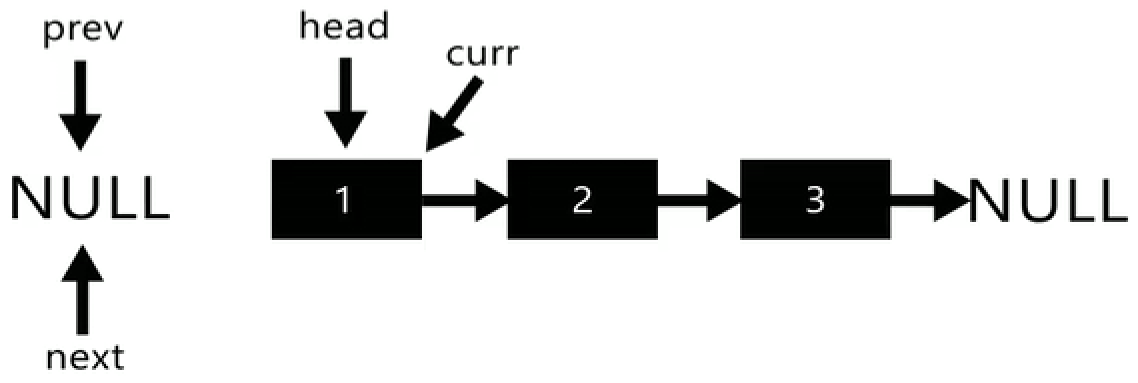
In [55]:
```
my_linked_list = LinkedList()
my_linked_list.append(1)
my_linked_list.prepend(2)
my_linked_list.append(8)
my_linked_list
```

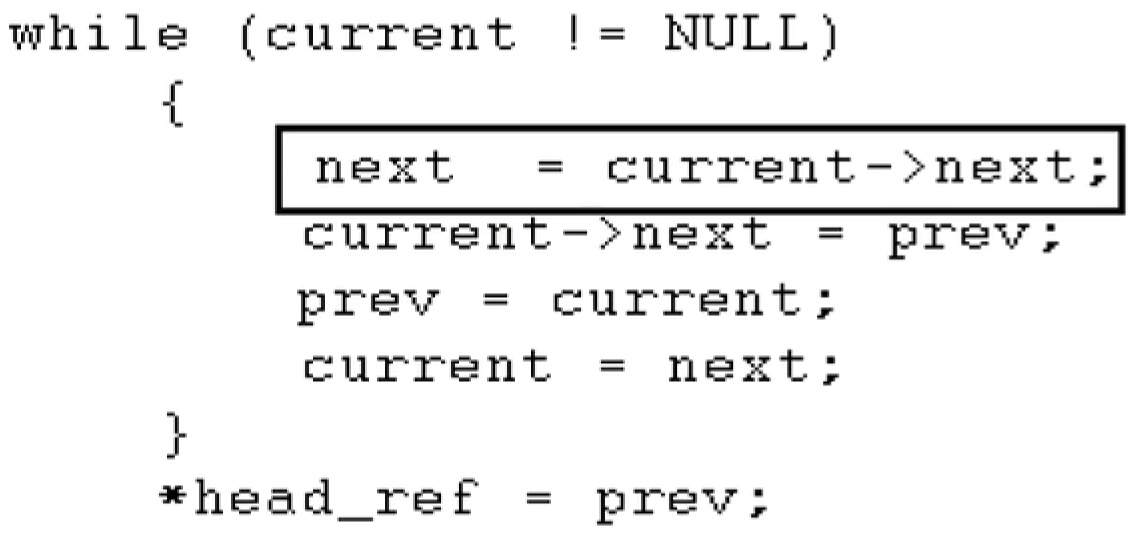
Out[55]:
```
(2, 1, 8)
```

Reversed Linked List

Process:

In [11]:
```python
class Node:
    def __init__(self, d, n) -> None:
        self.data = d
        self.next = n

class LinkedList:
    def __init__(self) -> None:
        self.head = None

    def prepend(self, value):
        """O (1)"""
        node = Node(value, self.head)
        self.head = node

    def append(self, value):
        """O (n)"""
        if self.head is None:
            self.head = Node(value, None)
            return

        itr = self.head
        while itr.next:
            itr = itr.next

        itr.next = Node(value, None)

    @property
    def size(self):
        """O (n)"""
        if self.head is None:
            return None

        count = 0
        itr = self.head
        while itr:
            itr = itr.next
            count += 1
        return count

    def __repr__(self):
        """Lookup | O(n)"""
        if self.head is None:
            return

        acc_items = ()
        itr = self.head
        while itr:
            acc_items += (itr.data,)
            itr = itr.next
        return repr(acc_items)

    def insert(self, index, value):
        """O (n)"""
        if index < 0 or index > self.size:
            raise Exception('Invalid Index')

        if index == 0:
            self.prepend(value)

        count = 0
        itr = self.head
        while itr:
            if count == index - 1:
                node = Node(value, itr.next)
                itr.next = node
                break

            itr = itr.next
            count += 1

    @property
    def reversed(self):
        if self.head is None or self.head.next is None:
            return self.head

        self.head = self._reverse(self.head)
        return self.head

    def _reverse(self, head):
        prev = None
        curr = head
        next = None
        while curr is not None:
            next = curr.next # Get the next node
            curr.next = prev # Invert the pointer to the previous node
            prev = curr # Previous node becomes the current node
            curr = next # Current node becomes the next node
        return prev # Returns the last value
```





```
while (current != NULL)
      {
         next  = current->next;
         current->next = prev;
         prev = current;
         current = next;
      }
      *head_ref = prev;
```

In [12]:
```
my_linked_list = LinkedList()
my_linked_list.append(1)
my_linked_list.prepend(2)
my_linked_list.append(8)

print("original:")
print(my_linked_list)

my_linked_list.reversed

print("reversed:")
print(my_linked_list)
```

```
Lista original:
(2, 1, 8)
Lista invertida:
(8, 1, 2)
```

Extra: reverse linked lists with recursion

In [ ]:
```
def reverseListRecursive(self, head):

    def reverse(cur, prev):
        if cur is None:
            return prev
        else:
            next = cur.next
            cur.next = prev

            return reverse(next, cur)

    return reverse(head, None)
```

Complete Linked List | Solution 2 - Doubly Linked List

In [28]:
```
class Node:

    def __init__(self, d, n = None, p = None):
        self.data = d
        self.next_node = n
        self.prev_node = p

    def __str__(self):
        return ('(' + str(self.data) + ')')

class CompleteLinkedList:

    def __init__(self, r = None):
        self.root = r # or head
        self.size = 0

    def prepend(self, d):
        new_node = Node(d, self.root)
        self.root = new_node
        self.size += 1

    def append(self, d):
        if self.root is None:
            self.root = Node(d, None)
            self.size += 1
            return

        this_node = self.root
        while this_node.next_node is not None:
            this_node = this_node.next_node
        this_node.next_node = Node(d, None)
        self.size += 1
        return True

    def insert(self, idx, d):
        if idx < 0 or idx > self.size:
            raise Exception('Invalid Index')
        if idx == 0:
            self.prepend(d)

        this_node = self.root
        count = 0

        while this_node.next_node is not None:
            if count ==  idx - 1:
                node = Node(d, this_node.next_node)
                this_node.next_node = node
                break
            this_node = this_node.next_node
            count += 1

    def find(self, d):
        this_node = self.root
        while this_node is not None:
            if this_node.data == d:
                return d
            else:
                this_node = this_node.next_node
        return None

    def remove(self, d):
        this_node = self.root
        prev_node = None

        while this_node is not None:
            if this_node.data == d:
                if prev_node is not None:
                    prev_node.next_node = this_node.next_node
                else:
                    # This d corresponds to the first node
                    self.root = this_node.next_node
                self.size -= 1
                return True
            else:
                prev_node = this_node
                this_node = this_node.next_node

    def show(self):
        this_node = self.root
        while this_node is not None:
            print(this_node, end=' -> ')
            this_node = this_node.next_node
        print('None')
```

In [29]:
```
myList = CompleteLinkedList()
myList.append(5)
myList.append(10)
myList.prepend(1)
# Expected output: (1) -> (5) -> (10) -> None
myList.show()
```

```
(1) -> (5) -> (10) -> None
```

In [30]:
```
myList.size
myList.remove(5)
myList.show()
```

```
(1) -> (10) -> None
```

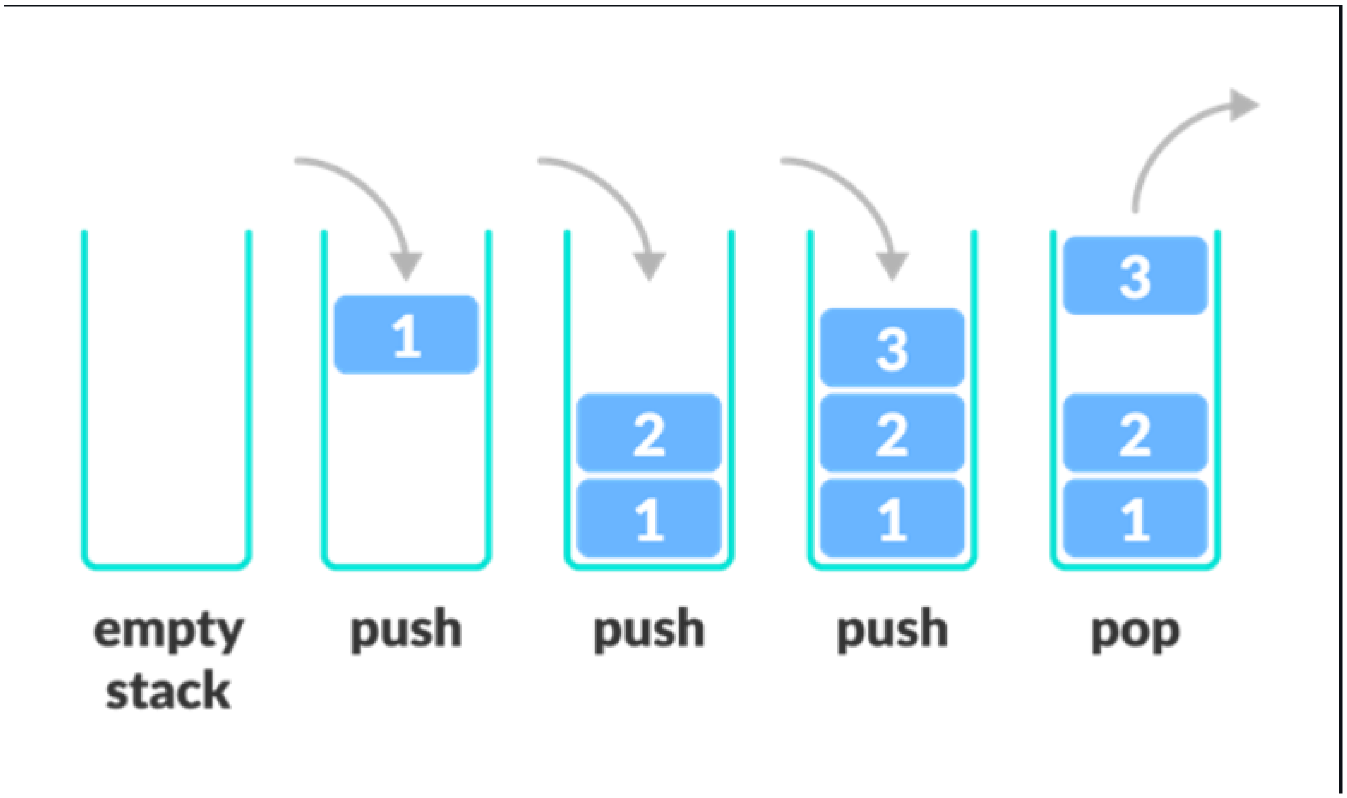
In [31]:
```
myList.insert(0, 5)
myList.show()
```

```
(5) -> (1) -> (10) -> None
```

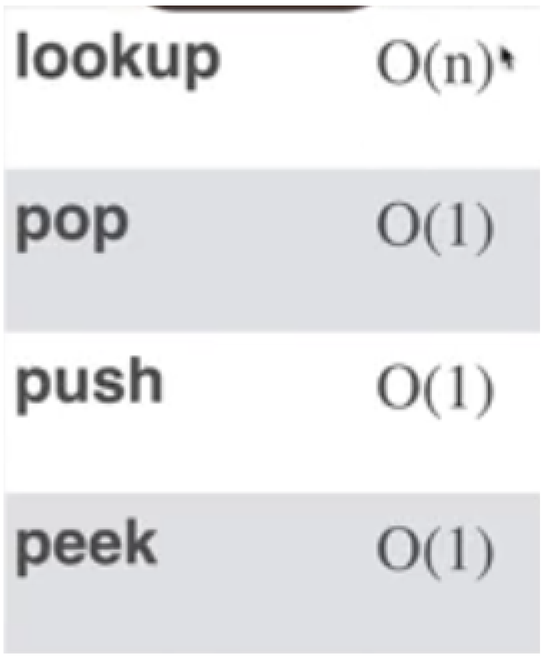
Stacks

A stack is a data structure that follows the Last In, First Out (LIFO) principle, meaning that the last element added is the first one to be removed. It operates like a collection of items with two main operations:

- Push: Adds an item to the top of the stack.
- Pop: Removes the item from the top of the stack.

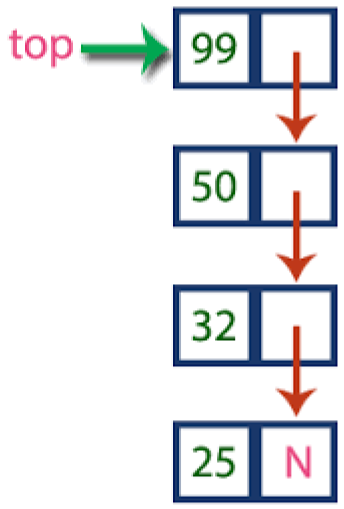


- Time complexity:



lookup	O(n)
pop	O(1)
push	O(1)
peek	O(1)

Stack using Linked Lists



In [183…

```python
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None


class Stack:
    def __init__(self):
        self.top = None
        self.bottom = None
        self.length = 0

    def peek(self):
        return self.top.value

    def push(self, value):
        new_node = Node(value)
        if self.length == 0:
            self.top = new_node
            self.bottom = new_node
        else:
           pointer = self.top # Store the current top value
           self.top = new_node # The top value becomes the new node
           self.top.next = pointer # The new node now points to the older top value
        self.length += 1
        return self

    def pop(self):
        if self.length == 0:
            return None

        if self.length == 1:
            self.top = None
            self.bottom = None
        else:
            self.top = self.top.next
        self.length -= 1

    def show(self):
        curr_node = self.top
        while curr_node is not None:
            print(curr_node.value)
            curr_node = curr_node.next
```

```
In [184…  mystack = Stack()
          myStack.push('google')
          myStack.push('amazon')
          myStack.push('discord')
          myStack.show()
```

```
discord
amazon
google
```

```
In [185…  myStack.peek()
```

```
Out[185]: 'discord'
```

```
In [186…  myStack.pop()
          myStack.show()
```

```
amazon
google
```

Stack using Arrays

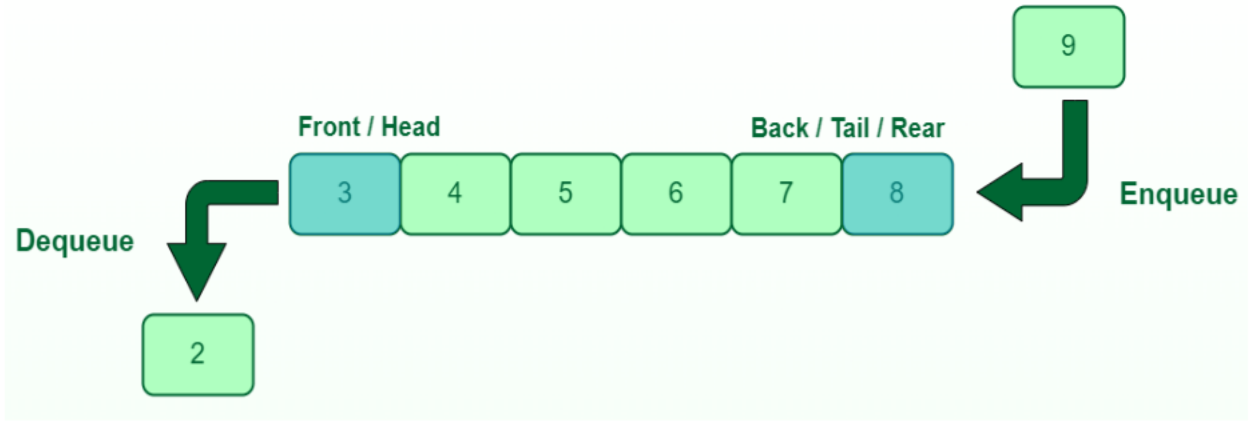
```
In [3]:  class Stack():
             def __init__(self) -> None:
                 self.stack = list()

             def push(self, item):
                 self.stack.append(item)

             def pop(self):
                 if len(self.stack) > 0:
                     return self.stack.pop()
                 else:
                     return None

             def peek(self):
                 if len(self.stack) > 0:
                     return self.stack[-1]

             def __str__(self):
                 return str(self.stack)
```

```
In [5]:  myStack = Stack()
         myStack.push('pizza')
         myStack.push('bread')
         myStack.push('banana')
         print(myStack)
```

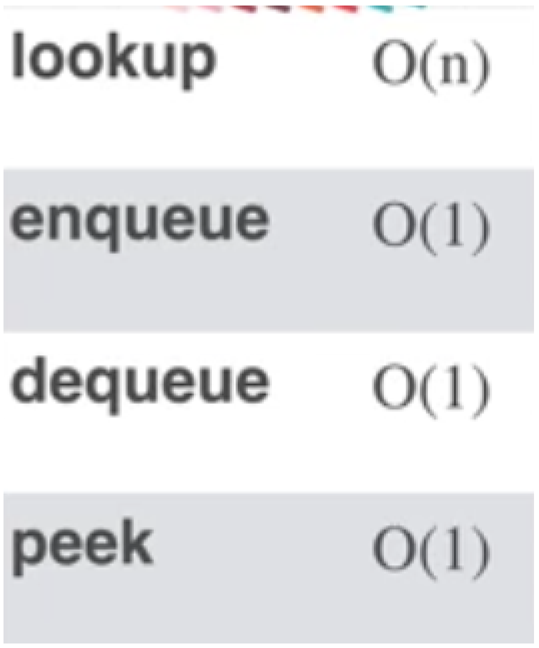
```
['pizza', 'bread', 'banana']
```

Queues

A queue is a data structure that follows the First In, First Out (FIFO) principle, meaning that the first element added to the queue is the first one to be removed. It operates much like a real-world queue or line, where individuals join at the back and leave from the front. In a queue, new elements are added at the rear, and removal takes place from the front. This ensures that the order of elements is preserved, and the oldest element is always the next to be processed or removed. Queues are commonly used in computer science and programming for tasks such as managing tasks in a print spooler, handling requests in a network, or implementing breadth-first search algorithms in graph traversal.



Queue Data Structure



lookup	O(n)
enqueue	O(1)
dequeue	O(1)
peek	O(1)

Queue with Linked List

In [16]:
```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class Queue:
    def __init__(self) -> None:
        self.first = None
        self.last = None
        self.length = 0

    def peek(self):
        if self.first is not None:
            return self.first.value
        return None

    def enqueue(self, value):
        new_node = Node(value)
        if not self.length:
            self.first = new_node
            self.last = new_node
        else:
            self.last.next = new_node
            self.last = new_node
        self.length += 1
        return self

    def dequeue(self):
        if self.first is None:
            return None

        if self.first == self.last:
            self.last = None
        self.first = self.first.next
        self.length -= 1
        return self

    def __str__(self):
        current = self.first
        elements = []
        while current:
            elements.append(current.value)
            current = current.next
        return f"Queue({', '.join(map(str, elements))})"
```

In [18]:
```
myQueue = Queue()
myQueue.enqueue('Joy')
myQueue.enqueue('Karl')
#myQueue.dequeue()
print(myQueue)
```

```
Queue(Joy, Karl)
```

Queues using Stacks

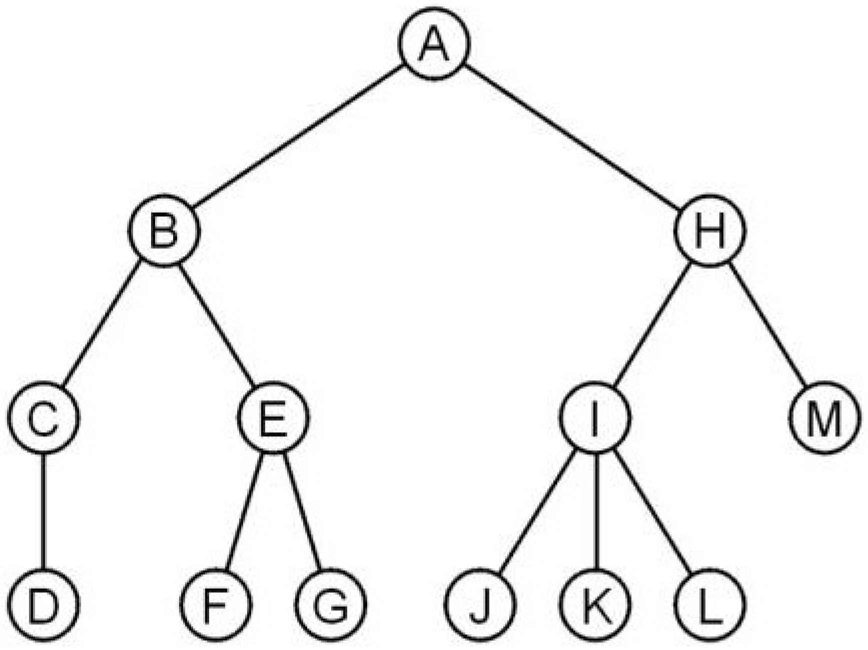
In [ ]:
```
from collections import deque

class CustomQueue:
    def __init__(self):
        self.queue = deque()
        self.size = 0

    def push(self, value):
        self.queue.append(value)
        self.size += 1

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

    def pop(self):
        if not self.queue:
            return None
        last_value = self.queue.popleft()
        self.size -= 1
        return last_value
```

```python
    def empty(self):
        return not self.queue
```

Trees

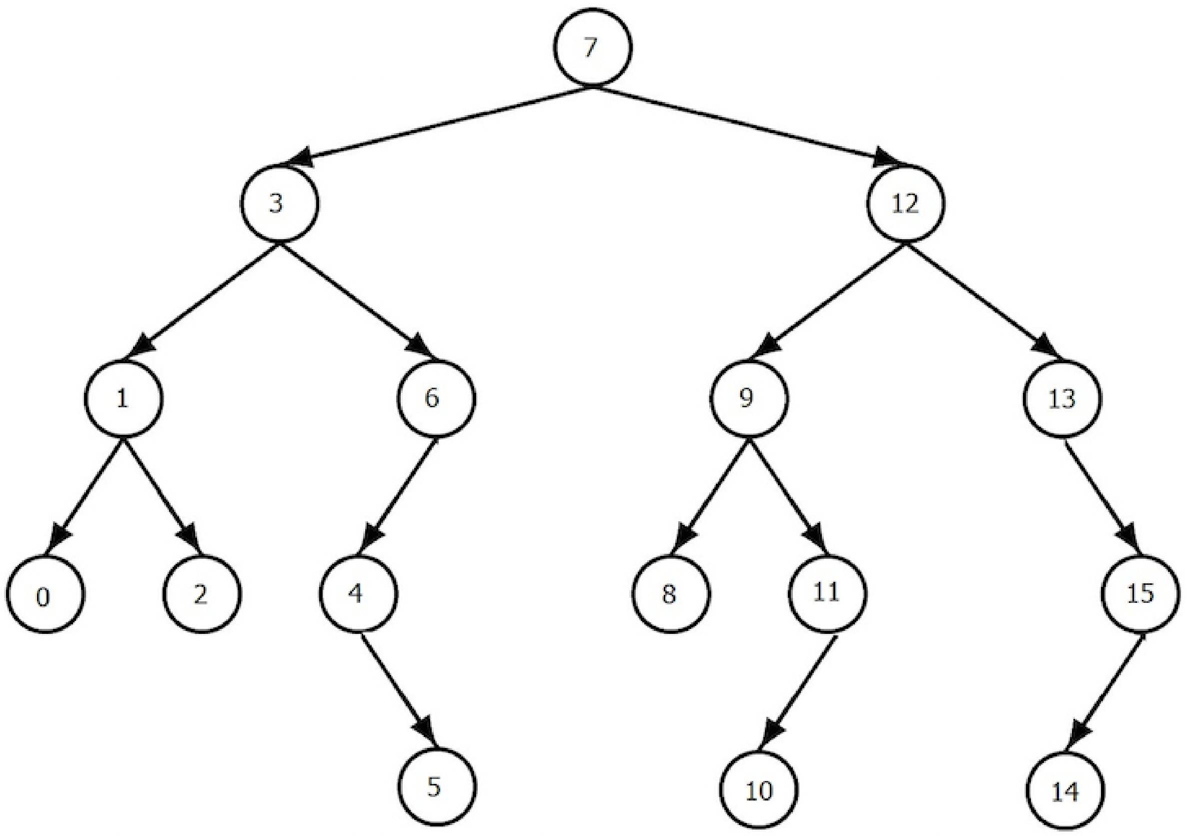
A tree is a widely used data structure in computer science that consists of nodes connected by edges. It is hierarchical in nature and typically has a single designated node called the "root." The nodes in a tree are organized in levels or layers, with each node having zero or more child nodes.

Trees are versatile and are used in various applications such as representing hierarchical relationships, organizing data efficiently (e.g., binary search trees), and facilitating efficient searching, sorting, and retrieval operations. Different types of trees include binary trees, AVL trees, B-trees, and others, each with specific properties and use cases.

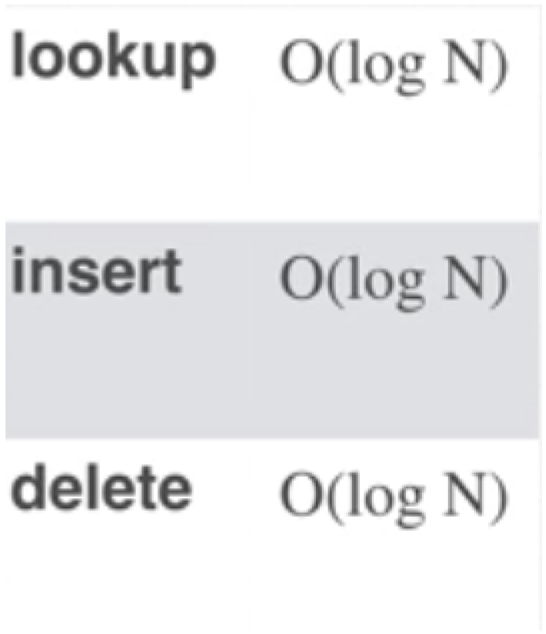


Binary Tree

A binary tree is a hierarchical data structure composed of nodes, where each node has at most two children, referred to as the left child and the right child. The topmost node in a binary tree is called the root. Nodes in a binary tree are organized in such a way that each node can have zero, one, or two children.



Time Complexity:



lookup	O(log N)
insert	O(log N)
delete	O(log N)

In [13]:
```python
class Node:
    def __init__(self, value):
        self.left = None
        self.right = None
        self.value = value

class BinarySearchTree:
    def __init__(self) -> None:
        self.root = None
    def insert(self, value):
        new_node = Node(value)

        if self.root is None:
            self.root = new_node
        else:
            current_node = self.root

            while True:
                if value < current_node.value:
                    if current_node.left is None:
                        current_node.left = new_node
                        return self
                    current_node = current_node.left
                elif value > current_node.value:
                    if current_node.right is None:
                        current_node.right = new_node
                        return self
                    current_node = current_node.right

    def remove(self, data):
        if self.root == None: # Tree is empty
            return "Tree Is Empty"
        current_node = self.root
        parent_node = None
        while current_node!=None: # Traversing the tree to reach the desired node or the end of
            if current_node.data > data:
                parent_node = current_node
                current_node = current_node.left
            elif current_node.data < data:
                parent_node = current_node
                current_node = current_node.right
            else: # Match is found. Different cases to be checked
                # Node has left child only
                if current_node.right == None:
                    if parent_node == None:
                        self.root = current_node.left
                        return
                    else:
                        if parent_node.data > current_node.data:
                            parent_node.left = current_node.left
                            return
                        else:
                            parent_node.right = current_node.left
                            return

                # Node has right child only
                elif current_node.left == None:
                    if parent_node == None:
                        self.root = current_node.right
                        return
                    else:
                        if parent_node.data > current_node.data:
                            parent_node.left = current_node.right
                            return
                        else:
                            parent_node.right = current_node.right
                            return

                # Node has neither left nor right child
                elif current_node.left == None and current_node.right == None:
```

```
                if parent_node == None: #Node to be deleted is root
                    current_node = None
                    return
                if parent_node.data > current_node.data:
                    parent_node.left = None
                    return
                else:
                    parent_node.right = None
                    return

            # Node has both left and right child
            elif current_node.left != None and current_node.right != None:
                del_node = current_node.right
                del_node_parent = current_node.right
                while del_node.left != None: #Loop to reach the leftmost node of the right su
                    del_node_parent = del_node
                    del_node = del_node.left
                current_node.data = del_node.data #The value to be replaced is copied
                if del_node == del_node_parent: #If the node to be deleted is the exact right
                    current_node.right = del_node.right
                    return
                if del_node.right == None: #If the leftmost node of the right subtree of the
                    del_node_parent.left = None
                    return
                else: #If it has a right subtree, we simply link it to the parent of the del_
                    del_node_parent.left = del_node.right
                    return

        return "Not Found"

    def lookup(self, value):
        if self.root is None:
            return False
        current_node = self.root
        while current_node:
            if value < current_node.value:
                current_node = current_node.left
            elif value > current_node.value:
                current_node = current_node.right
            elif current_node.value == value:
                return current_node
        return False

    def _print_tree(self, node, level=0, prefix="Root: "):
            if node is not None:
                print(" " * (level * 4) + prefix + str(node.value))
                if node.left is not None or node.right is not None:
                    self._print_tree(node.left, level + 1, "L--- ")
                    self._print_tree(node.right, level + 1, "R--- ")

    def visualize(self):
        self._print_tree(self.root)
```

In [14]:

```
bst = BinarySearchTree()
values = [9, 4, 6, 20, 170, 15, 1]

for value in values:
    bst.insert(value)

bst.visualize()
```

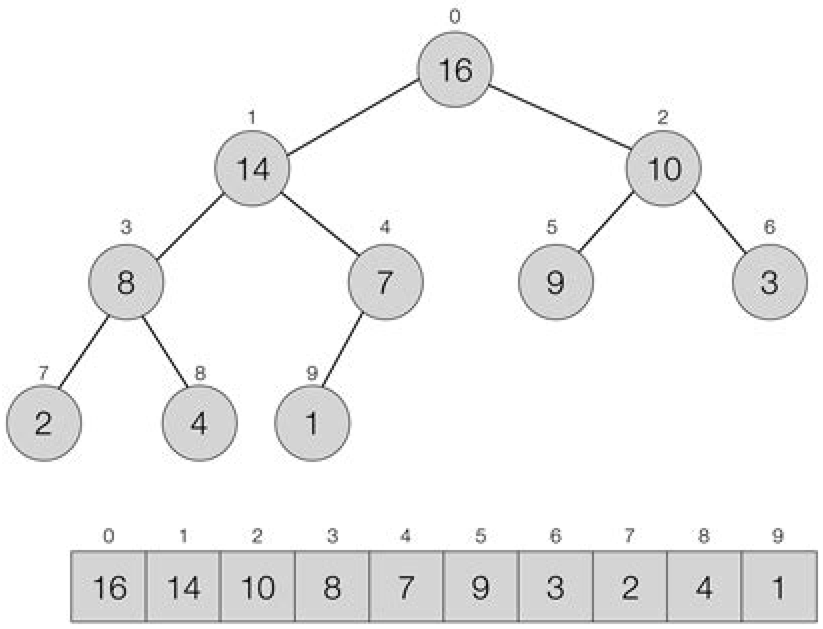
```
Root: 9
    L--- 4
        L--- 1
        R--- 6
    R--- 20
        L--- 15
        R--- 170
```

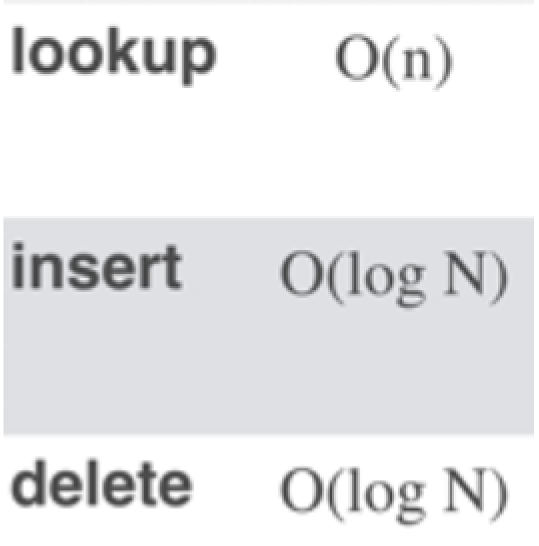
In [19]:
```
value = 9
if result := bst.lookup(value):
    print(result.value)
    print(result.left.value)
    print(result.right.value)
else:
    print('Value not found')
```

```
9
4
20
```

Binary Heaps

A binary heap is a complete binary tree that is used to store data efficiently to get the max or min element based on its structure 1. A binary heap is either a min heap or a max heap. In a min binary heap, the key at the root must be minimum among all keys present in the binary heap. The same property must be recursively true for all nodes in the binary tree. Max binary heap is similar to min heap 1.





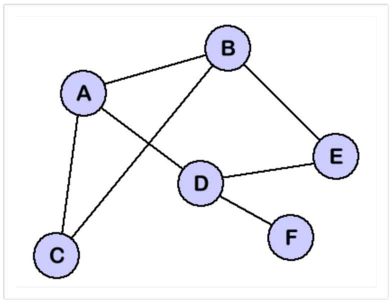
lookup	O(n)
insert	O(log N)
delete	O(log N)

In [3]:

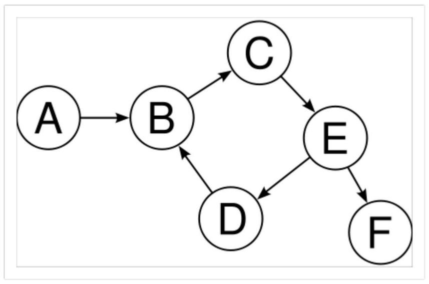
```python
class MaxHeap():
    def __init__(self, items=[]):
        self.heap = [0]
        for item in items:
            self.heap.append(item)
            self.__floatUp(len(self.heap) - 1)

    def push(self, item):
        self.heap.append(item)
        self.__floatUp(-len(self.heap) - 1)


    def peek(self):
        if self.heap[1]:
            return self.heap[1]
        else:
            return None

    def pop(self):
        if len(self.heap) > 2:
            self.__swap(1, len(self.heap) - 1)
            max = self.heap.pop()
            self.__bubbleDown(1)
        elif len(self.heap) == 2:
            max = self.heap.pop()
        else:
            max = False
        return max

    def __swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

    def __floatUp(self, index):
        parent = index//2
        if index <= 1:
            return
        elif self.heap[index] > self.heap[parent]:
            self.__swap(index, parent)
            self.__floatUp(parent)

    def __bubbleDown(self, index):
        left = index * 2
        right = index * 2 + 1
        largest = index
        # Se o valor de um dos nós filhos for maior que o do pai, eles trocam de lugar
        if len(self.heap) > left and self.heap[largest] < self.heap[left]:
            largest = left
        if len(self.heap) > right and self.heap[largest] < self.heap[right]:
            largest = right
        if largest != index:
            self.__swap(index, largest)
            self.__bubbleDown(largest)

    # Retorna a lista
    def __str__(self):
        return str(self.heap)
```

Graphs

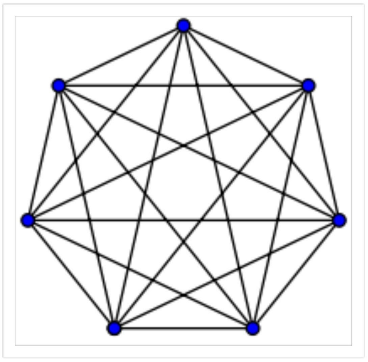
A graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally, a Graph is composed of a set of vertices (V) and a set of edges (E). The graph is denoted by G(E, V). Vertices are the fundamental units of the graph, and edges are drawn or used to connect two nodes of the graph. Graphs are used to solve many real-life problems, such as representing networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like LinkedIn, Facebook. For example, in Facebook, each person is represented with a vertex (or node). Each node is a structure and contains information like person id, name, gender, locale, etc.



undirected graph



directed graph



complete graph

Implementation 1

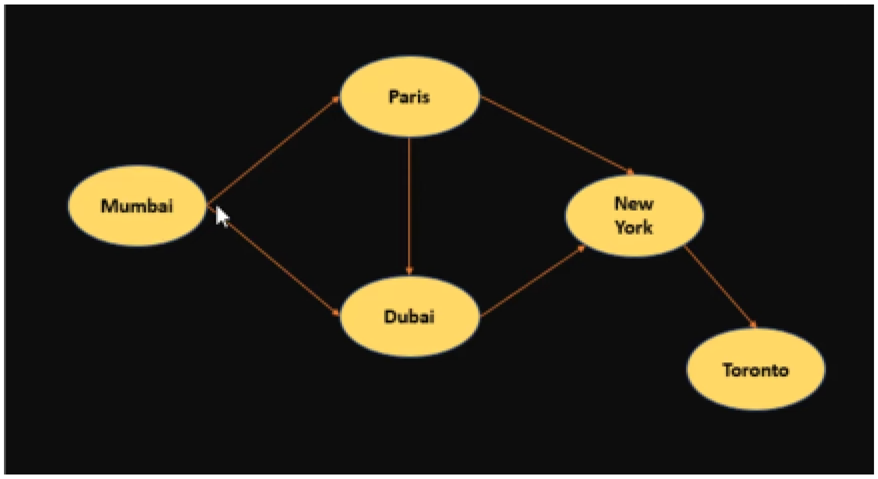
```
In [ ]: class Graph:
            """Unweighted and undirected"""
            def __init__(self):
                self.number_of_nodes = 0
                self.list_of_nodes = {}

            def add_node(self, node):
                self.list_of_nodes[node] = []
                self.number_of_nodes += 1
```

```
    def add_edge(self, node_1, node_2):
        self.list_of_nodes[node_1].append(node_2)
        self.list_of_nodes[node_2].append(node_1)
```

https://www.youtube.com/watch?v=tWVWeAqZ0WU

Implementation 2

Graph use case:



In [11]:
```
class Graph:
    def __init__(self, edges):
        self.edges = edges
        self.graph_dict = {}
        for start, end in self.edges:
            if start in self.graph_dict:
                self.graph_dict[start].append(end)
            else:
                self.graph_dict[start] = [end]
        print(f"Graph dict: {self.graph_dict}")

    def get_paths(self, start, end, path=[]):
        path = path + [start]

        if start == end:
            return [path]

        if start not in self.graph_dict:
            return []

        paths = []
        for node in self.graph_dict[start]:
            if node not in path:
                new_paths = self.get_paths(node, end, path)
                for p in new_paths:
                    paths.append(p)

        return paths
```

In [12]:
```
routes = [
        ("Mumbai", "Paris"),
        ("Mumbai", "Dubai"),
        ("Paris", "Dubai"),
        ("Paris", "New York"),
        ("Dubai", "New York"),
        ("New York", "Toronto"),
    ]

route_graph = Graph(routes)

start = "Mumbai"
end = "New York"

print(f"All paths between: {start} and {end}: ", route_graph.get_paths(start,end))
```

```
Graph dict: {'Mumbai': ['Paris', 'Dubai'], 'Paris': ['Dubai', 'New York'], 'Dubai': ['New Yor
k'], 'New York': ['Toronto']}
All paths between: Mumbai and New York:  [['Mumbai', 'Paris', 'Dubai', 'New York'], ['Mumbai',
'Paris', 'New York'], ['Mumbai', 'Dubai', 'New York']]
```