Week 1

1. **Python program to Use and demonstrate basic data structures.**

Algorithm:

Step 1:[Input Operation & Import time]

import time

start = time.time()

List=[1,2,"ABC", 3, "xyz", 2.3]

Dict={"a":1,"b":2,"c":3}

Tup=(1,2,3,4,5)

S={1,1,2,2,3,3,4,4,5,5}

Step 2:[Output operation]

Print list("List")

Print ("\nDictionary")

Print ("\n Tuples")

Print ("\n Sets")

Print(list)

Print (Dict)

Print (Tup)

Print (s)

end = time.time()

print(f"Runtime of the program is {end - start}")

Python Code

```
import time
start = time.time()
```

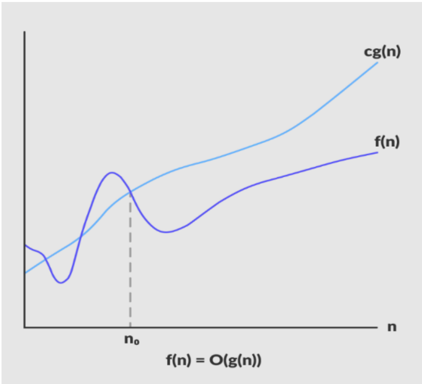
```
print("List")
List = [1, 2,"ABC", 3, "xyz", 2.3]
print(List)
print("\nDictionary")
Dict={"a":1,"b":2,"c":3}
print(Dict)
print("\n Tuples")
Tup=(1,2,3,4,5)
print (Tup)
print("\n Sets")
s={1,1,2,2,3,3,4,4,5,5}
print(s)
end = time.time()
print(f"Runtime of the program is {end - start}")
```

Output:

```
List
[1, 2, 'ABC', 3, 'xyz', 2.3]

Dictionary
{'a': 1, 'b': 2, 'c': 3}

 Tuples
(1, 2, 3, 4, 5)

 Sets
{1, 2, 3, 4, 5}
Runtime of the program is 0.1248924732208252
```

Time Complexity For open addressing:

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)
Insertion	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)



- List. Insert: O(n) Get Item: O(1) ...
- Dictionary. Get Item: O(1) Set Item: O(1) ...
- Set. Check for item in set: O(1) Difference of set A from B: O(length of A) ...
- Tuples. Tuples support all operations that do not mutate the data structure (and they. have the same complexity classes).

2. Implement an ADT with all its operations

Algorithm

Step 1: [Create class of stack & import time]

```
import time

start = time.time()

class Stack:
```

Step 2: [Define functions of basic operations under class stack]

```
def __init__(self):
        self.items = []
    def isEmpty(self):
        return self.items == []
    def push(self, item):
        self.items.append(item)
        print(item)
    def pop(self):
        return self.items.pop()
    def peek(self):
        return self.items[len(self.items) - 1]
    def size(self):
        return len(self.items)
s=Stack()
print(s.isEmpty()," - because stack is empty")
print("elements are pushed into stack for Operation")
s.push(11)
s.push(12)
s.push(13)
```

Step 3: [Output Operation]

```
print("Size",s.size())
print("Peek",s.peek())
print("Pop Operation")
print(s.pop())
print(s.pop())
```

print("Size",s.size())

print(40* '*')

Step 4: [Create class of Queue]

class Queue:

Step 5: [Define functions of basic operations class queue]

```
def __init__(self):
    self.items = []
  def isEmpty(self):
    return self.items == []
   def enqueue(self,item):
    self.items.append(item)
    print(item)
    def dequeue(self):
    return self.items.pop(0)
   def front(self):
    return self.items[len(self.items)-1]
   def size(self):
    return len(self.items)
```

Step 6: [Output Operation]

```
print(q.isEmpty(),"- because queue is empty")
print("Enquee")
q.enqueue(11)
q.enqueue(12)
q.enqueue(13)
print("Front",q.front())
```

```
    print("Dequee")
    print(q.dequeue())
    print(q.dequeue())
    print("Size",q.size())
    end = time.time()
    print(f"Runtime of the program is {end - start}")
```

Python Code

```
import time
start = time.time()
class Stack:
    def __init__(self):
        self.items = []
    def isEmpty(self):
        return self.items == []
    def push(self, item):
        self.items.append(item)
        print(item)
    def pop(self):
        return self.items.pop()
    def peek(self):
        return self.items[len(self.items) - 1]
    def size(self):
        return len(self.items)
s=Stack()
```

```
print(s.isEmpty()," - because stack is empty")
print("elements are pushed into stack for Operation")
s.push(11)
s.push(12)
s.push(13)
print("Size",s.size())
print("Peek",s.peek())
print("Pop Operation")
print(s.pop())
print(s.pop())
print("Size",s.size())
print((40* '*')
class Queue:
    def __init__(self):
        self.items = []
    def isEmpty(self):
         return self.items == []
    def enqueue(self,item):
        self.items.append(item)
        print(item)
    def dequeue(self):
        return self.items.pop(0)
    def front(self):
        return self.items[len(self.items)-1]
    def size(self):
```

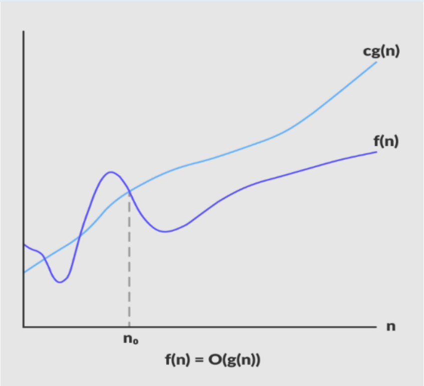
```
        return len(self.items)
q=Queue()
print(q.isEmpty(),"- because queue is empty")
print("Enquee")
q.enqueue(11)
q.enqueue(12)
q.enqueue(13)
print("Front",q.front())
print("Dequee")
print(q.dequeue())
print(q.dequeue())
print("Size",q.size())
end = time.time()
print(f"Runtime of the program is {end - start}")
```

OUTPUT:

```
True  - because stack is empty
elements are pushed into stack for Operation
11
12
13
Size 3
Peek 13
Pop Operation
13
12
Size 1
****************************************************
True - because queue is empty
Enquee
```

11
12
13
Front 13
Dequee
11
12
Size 1
Runtime of the program is 0.42179059982299805
Time Complexity For open addressing:

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)
Insertion	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)



Time Complexity · Best Case Scenario is O(1) as only one elements needs to be pushed onto the stack. · Average Case Scenario would be O(1).

Week 2

1. **Implement an ADT and Compute space and time complexities.**

Algorithm

Step 1:[input operation & import time]

```
Import time
Start = time.time()
Class stack:
    def __init__(self):
        self.items = []
    def isEmpty(self):
        return self.items == []
    def push(self, item):
        self.items.append(item)
        print(item)
    def pop(self):
        return self.items.pop()
    def peek(self):
        return self.items[len(self.items) - 1]
    def size(self):
        return len(self.items)
```

step 2:[output operation]

```
s=Stack()
print(s.isEmpty())
print("Push")
s.push(11)
s.push(12)
s.push(13)
print("Peek",s.peek())
```

```
print("Pop")
print(s.pop())
print(s.pop())
print("Size",s.size())
end = time.time()
print(f"Runtime of the program is {end - start}")
```

Python code

```
import time
start = time.time()
class Stack:
    def __init__(self):
        self.items = []
    def isEmpty(self):
        return self.items == []
    def push(self, item):
        self.items.append(item)
        print(item)
    def pop(self):
        return self.items.pop()
    def peek(self):
        return self.items[len(self.items) - 1]
    def size(self):
        return len(self.items)
s=Stack()
```

```
print(s.isEmpty())
print("Push")
s.push(11)
s.push(12)
s.push(13)
print("Peek",s.peek())
print("Pop")
print(s.pop())
print(s.pop())
print("Size",s.size())
end = time.time()
print(f"Runtime of the program is {end - start}")
```

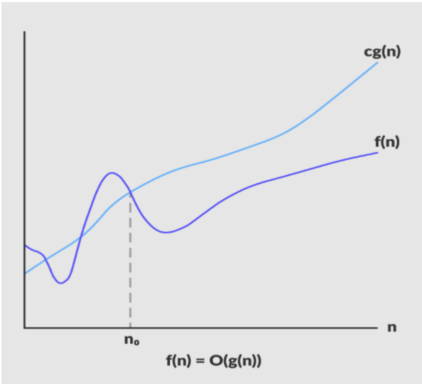
Output:

```
True
Push
11
12
13
Peek 13
Pop
13
12
Size 1
Runtime of the program is 0.015621423721313477
```

Time Complexity For open addressing:

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)
Insertion	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)

Time Complexity



- Worst Case Scenario is O(1) , as Deletion operation only removes the top element.
- Best Case Scenario is O(1).
- Average Case Scenario would be O(1) as only the top element is needed to be removed.

Space Complexity

- Space Complexity of Pop Operation is O(1) as no additional space is required for it.

2. Implement above solution using array and Compute space and time complexities and compare two solutions.

Algorithm

Step 1:[input operation & import time]

Import time

Start = time.time()

Step 2:[add values for the variable a]

A = [1,2,3,4]

Step 3:[output operation]

print(a)

a.insert(4,5)

print(a)

a.pop(0)

print(a)

l=len(a)

print(l)

end = time.time()

print(f"Runtime of the program is {end - start}")

Python Code

```
import time
start = time.time()
a = [1,2,3,4]
print(a)
a.insert(4,5)
print(a)
a.pop(0)
print(a)
l=len(a)
```

print(l)

end = time.time()

print(f"Runtime of the program is {end - start}")

Output:

[1, 2, 3, 4]

[1, 2, 3, 4, 5]

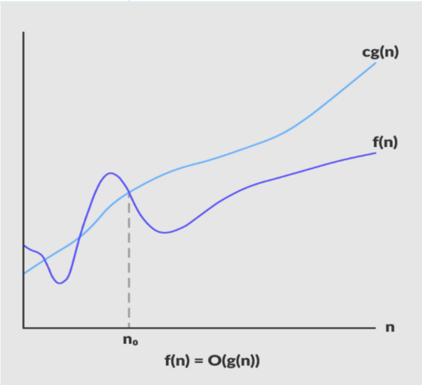
[2, 3, 4, 5]

4

Runtime of the program is 0.015436410903930664

Time Complexity For open addressing:

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)
Insertion	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)



Space complexity (for n push)

Time complexity of push()

Time complexity of pop()

Time complexity of length()

Time complexity of is_empty()

Week 3

1. **Implement Linear Search compute space and time complexities, plot graph using asymptomatic notations.**

Linear search algorithm

Step 1: [Import time]

```
Import time
start=time.time()
```

Step 2: [Define a function for linear search]

```
def linearsearch(a, key):
 n = len(a)
 for i in range(n):
     if a[i] == key:
         return i;
 return -1
```

Step 3: [Define an array]

```
a = [13,24,35,46,57,68,79]
print("the array elements are:",a)
```

Step 4: [Enter the element to be searched]

```
k = int(input("enter the key element to search:"))
```

Step 5: [Output operation]

```
i = linearsearch(a,k)
```

```
    if i == -1:
     print("Search UnSuccessful")
    else:
     print("Search Successful key found at location:",i+1)
    end=time.time()
    print(f"Runtime of the program is {end - start}")
```

Python Code

```
import time
start=time.time()
def linearsearch(a, key):
 n = len(a)
 for i in range(n):
     if a[i] == key:
         return i;
 return -1
a = [13,24,35,46,57,68,79]
print("the array elements are:",a)
k = int(input("enter the key element to search => "))
i = linearsearch(a,k)
if i == -1:
 print("Search UnSuccessful")
else:
 print("Search Successful key found at location:",i+1)
end=time.time()
```

print(f"Runtime of the program is {end - start}")

Output:

The array elements are : [13, 24, 35, 46, 57, 68, 79]

enter the key element to search => 46

Search Successful key found at location : 4

Runtime of the program is 2.468712568283081

The array elements are : [13, 24, 35, 46, 57, 68, 79]

enter the key element to search => 20

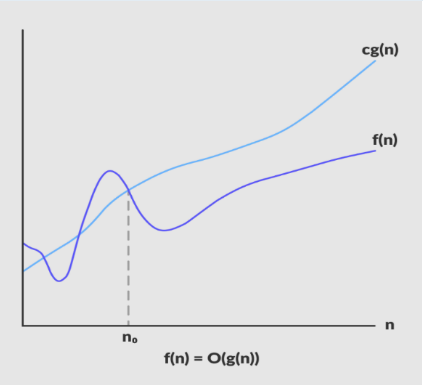
Search UnSuccessful

Runtime of the program is 2.468712568283081

The complexity of Linear Search Technique

Time Complexity For open addressing:

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)
Insertion	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)



Time Complexity: O(n)

Space Complexity: O(1)

2. Implement Bubble, Selection, insertion sorting algorithms compute space and time complexities, plot graph using asymptomatic notations.

Bubble sort Algorithm

Step 1: [Import time and define a function for Bubble sort]

```
import time
start=time.time()
def bubblesort(a):
    n = len(a)
    for i in range(n-2):
        for j in range(n-2-i):
            if a[j]>a[j+1]:
                temp = a[j]
                a[j] = a[j+1]
                a[j+1] = temp
```

Step 2: [Create array and do the operation]

```
alist = [34,46,43,27,57,41,45,21,70]
```

Step 3: [Output operation]

```
print("Before sorting:",alist)
bubblesort(alist)
end=time.time()
print(f"Runtime of the program is {end - start}")
```

Python Code

```
import time
start=time.time()
def bubblesort(a):
    n = len(a)
    for i in range(n-2):
        for j in range(n-2-i):
            if a[j]>a[j+1]:
                temp = a[j]
                a[j] = a[j+1]
                a[j+1] = temp
alist = [34,46,43,27,57,41,45,21,70]
print("Before sorting:",alist)
bubblesort(alist)
print("After sorting:",alist)
end=time.time()
print(f"Runtime of the program is {end - start}")
```

Output:

Before sorting: [34, 46, 43, 27, 57, 41, 45, 21, 70]

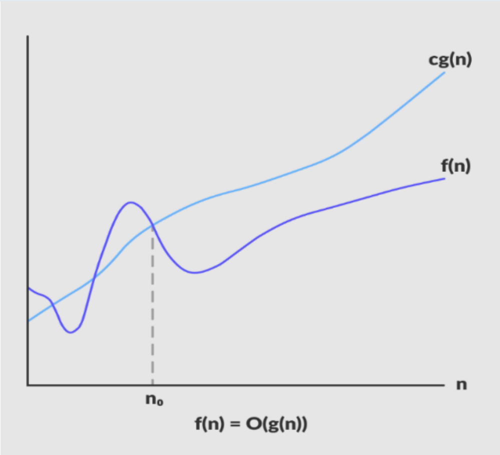
After sorting: [21, 27, 34, 41, 43, 45, 46, 57, 70]

Runtime of the program is 0.06250619888305664

Time Complexity For open addressing:

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Insertion	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)



The bubble sort algorithm is a reliable sorting algorithm. This algorithm has a worst-case time complexity of O(n2). The bubble sort has a space complexity of O(1).

Selection Sort Algorithm

Step 1: [Import time & create function of selection sort]

```
import time
start=time.time()
def selectionsort(a):
 n = len(a)
 for i in range(n-2):
     min = i
 for j in range(i+1,n-1):
     if a[j]<a[min]:
         temp = a[j]
```

```
            a[j] = a[min]
            a[min] = temp
```

Step 2: [Define array & execute the operation]

```
    alist = [34,46,43,27,57,41,45,21,70]
    print("Before sorting:",alist)
    selectionsort(alist)
    print("After sorting:",alist)
    end=time.time()
    print(f"Runtime of the program is {end - start}")
```

Python Code

```
import time
start=time.time()
def selectionsort(a):
    n = len(a)
    for i in range(n-2):
        min = i
        for j in range(i+1,n-1):
            if a[j]<a[min]:
                temp = a[j]
                a[j] = a[min]
                a[min] = temp
alist = [34,46,43,27,57,41,45,21,70]
print("Before sorting:",alist)
selectionsort(alist)
print("After sorting:",alist)
```

end=time.time()

print(f"Runtime of the program is {end - start}")

Output:

Before sorting: [34, 46, 43, 27, 57, 41, 45, 21, 70]

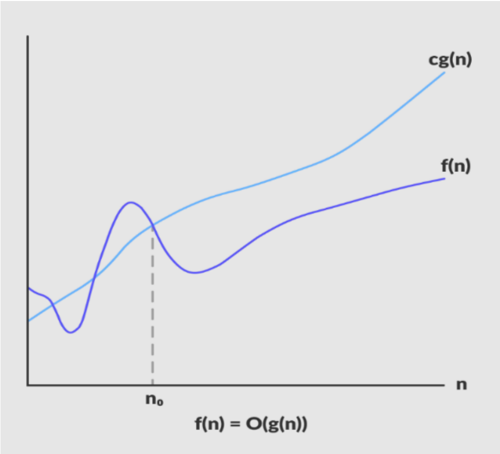
After sorting: [21, 27, 34, 41, 43, 45, 46, 57, 70]

Runtime of the program is 0.06238508224487305

Time Complexities:

Time Complexity For open addressing:

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)
Insertion	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)



The time complexity of the bubble sort is O(n2)
Worst Case Complexity: O(n2) : If we want to sort in ascending order and the array is in descending order then the worst case occurs.

Best Case Complexity: O(n2) : If the array is already sorted, then there is no need for sorting.
Average Case Complexity: O(n2) : It occurs when the elements of the array are in random order.

Space Complexity: Space complexity is O(1) because an extra variable (temp) is used for swapping.

Insertion sort Algorithm

Step 1: [Define insertion sort long with importing time]

```
import time
start=time.time()
def insertionsort(a):
 n = len(a)
 for i in range(1,n-1):
    k = a[i]
    j = i-1
    while j>=0 and a[j]>k:
        a[j+1] = a[j]
        j=j-1
        a[j+1] = k
```

Step 2: [Create an array]

```
alist = [34,46,43,27,57,41,45,21,70]
print("Before sorting:",alist)
```

Step 3: [Output operation]

```
insertionsort(alist)
print("After sorting:",alist)
```

```
        end=time.time()
        print(f"Runtime of the program is {end - start}")
```

Python code

```
import time
start=time.time()
def insertionsort(a):
 n = len(a)
 for i in range(1,n-1):
     k = a[i]
     j = i-1
     while j>=0 and a[j]>k:
         a[j+1] = a[j]
         j=j-1
         a[j+1] = k
alist = [34,46,43,27,57,41,45,21,70]
print("Before sorting:",alist)
insertionsort(alist)
print("After sorting:",alist)
end=time.time()
print(f"Runtime of the program is {end - start}")
```

Output:

Before sorting: [34, 46, 43, 27, 57, 41, 45, 21, 70]

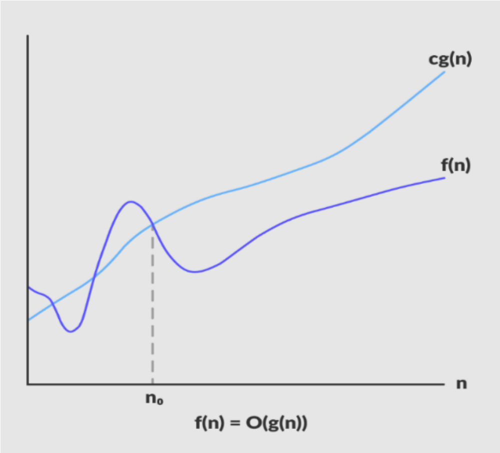
After sorting: [21, 27, 34, 41, 43, 45, 46, 57, 70]

Runtime of the program is 0.07800817489624023

Time Complexity For open addressing:

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)
Insertion	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)

Time Complexities:



The time complexity of the bubble sort is O(n2)

- Worst Case Complexity: O(n2) : If we want to sort in ascending order and the array is in descending order then the worst case occurs.
- Best Case Complexity: O(n) : If the array is already sorted, then there is no need for sorting.
- Average Case Complexity: O(n2) : It occurs when the elements of the array are in random order.
- Space Complexity: Space complexity is O(1) because an extra variable (temp) is used for swapping.

Week 4

1. Implement Binary Search using recursion Compute space and time complexities, plot graph using asymptomatic notations and compare two.

Algorithm:

Step 1: [Import Input Operation]

```
import time
start=time.time()
Input arr, low, high, x
```

Step 2: [Define Fuctions]

```
def binary_search(arr, low, high, x):
    if high >= low:
        mid = (high + low) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] > x:
            return binary_search(arr, low, mid - 1, x)
        else:
            return binary_search(arr, mid + 1, high, x)
    else:
        return -1
```

Step 3: [Output operation]

```
result = binary_search(arr, 0, len(arr)-1, x)
if result != -1:
    print("Element is present at index", str(result))
else:
    print("Element is not present in array")
```

```
end=time.time()
print(f"Runtime of the program is {end - start}")
```

Python code

```
import time
start=time.time()
def binary_search(arr, low, high, x):
    if high >= low:
        mid = (high + low) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] > x:
            return binary_search(arr, low, mid - 1, x)
        else:
            return binary_search(arr, mid + 1, high, x)
    else:
        return -1
arr = [ 2, 3, 4, 10, 40 ]
x = 10
result = binary_search(arr, 0, len(arr)-1, x)
```

```
if result != -1:
    print("Element is present at index", str(result))
else:
    print("Element is not present in array")
end=time.time()
print(f"Runtime of the program is {end - start}")
```

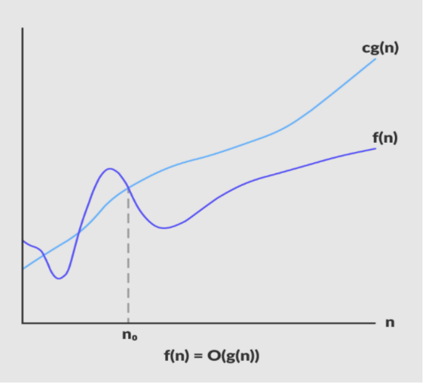
Output:

Element is present at index 3

Runtime of the program is 0.031137466430664062

Time Complexity For open addressing:

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)
Insertion	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)



The time complexity of the binary search algorithm is O(log n). The best-case time complexity would be O(1) when the central index would directly match the desired value. The worst-case scenario could be the values at either extremity of the list or values not in the list.

2. Implement Merge and quick sorting algorithms compute space and time complexities, plot graph using asymptomatic notations and compare all solutions

Merge sort

Algorithm

Step 1: [Import time]

```
import time
start=time.time()
```

Step 2: [Define Merage Sort]

```
def mergeSort(myList):
    if len(myList) > 1:
        mid = len(myList) // 2
        left = myList[:mid]
        right = myList[mid:]
```

Step 3: [Recursive operation on each half & Two iterators for traversing the two halves]

```
mergeSort(left)
        mergeSort(right)
        i = 0
        j = 0
```

Step 4: [Iterator for the main list]

```
k = 0
```

```
        while i < len(left) and j < len(right):
            if left[i] <= right[j]:
              myList[k] = left[i]
              i += 1
            else:
                myList[k] = right[j]
                j += 1
        # For all the remaining values
        while i < len(left):
            myList[k] = left[i]
            i += 1
            k += 1
        while j < len(right):
            myList[k]=right[j]
            j += 1
            k += 1
```

Step 5: [Output operation]

```
        myList = [54,26,93,17,77,31,44,55,20]
        mergeSort(myList)
        print(myList)
        end=time.time()
        print(f"Runtime of the program is {end - start}")
```

Python Code

```
import time
start=time.time()
```

```
def mergeSort(myList):
    if len(myList) > 1:
        mid = len(myList) // 2
        left = myList[:mid]
        right = myList[mid:]
        mergeSort(left)
        mergeSort(right)
        i = 0
        j = 0
        k = 0
        while i < len(left) and j < len(right):
            if left[i] <= right[j]:
              myList[k] = left[i]
              i += 1
            else:
                myList[k] = right[j]
                j += 1
            k += 1
        while i < len(left):
            myList[k] = left[i]
            i += 1
            k += 1
        while j < len(right):
            myList[k]=right[j]
            j += 1
```

```
        k += 1
myList = [54,26,93,17,77,31,44,55,20]
mergeSort(myList)
print(myList)
end=time.time()
print(f"Runtime of the program is {end - start}")
```

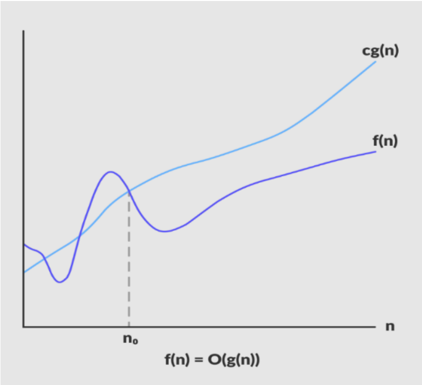
Output:

[17, 20, 26, 31, 44, 54, 55, 77, 93]

Runtime of the program is 0.015508413314819336

Time Complexity For open addressing:

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)
Insertion	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)



Best Case Time Complexity: O(n*log n)

Worst Case Time Complexity: O(n*log n)

Average Time Complexity: O(n*log n)

Quick Sort

Algorithm

Step 1: [Import time]

```
import time
start=time.time()
```

Step 2: [Define Quicksort & input an array]

```
def QuickSort(arr):
     elements = len(arr)
    #Base case
  if elements < 2:
    return arr
```

Step 3: [Position of the partitioning element & loop]

```
     current_position = 0
       for i in range(1, elements):
          if arr[i] <= arr[0]:
             current_position += 1
             temp = arr[i]
             arr[i] = arr[current_position]
             arr[current_position] = temp
  temp = arr[0]
  arr[0] = arr[current_position]
```

arr[current_position] = temp #Brings pivot to it's appropriate position

left = QuickSort(arr[0:current_position]) #Sorts the elements to the left of pivot

right = QuickSort(arr[current_position+1:elements]) #sorts the elements to the right of pivot

arr = left + [arr[current_position]] + right #Merging everything together

return arr

Step 4: [Output the program]

array_to_be_sorted = [4,2,7,3,1,6]

print("Original Array: ",array_to_be_sorted)

print("Sorted Array: ",QuickSort(array_to_be_sorted))

end=time.time()

print(f"Runtime of the program is {end - start}")

Python Code

```
import time
start=time.time()
def QuickSort(arr):
   elements = len(arr)
   if elements < 2:
      return arr
   current_position = 0
   for i in range(1, elements):
       if arr[i] <= arr[0]:
```

```
            current_position += 1
            temp = arr[i]
            arr[i] = arr[current_position]
            arr[current_position] = temp
    temp = arr[0]
    arr[0] = arr[current_position]
    arr[current_position] = temp
    left = QuickSort(arr[0:current_position])
    right = QuickSort(arr[current_position+1:elements])
    arr = left + [arr[current_position]] + right
    return arr
array_to_be_sorted = [4,2,7,3,1,6]
print("Original Array: ",array_to_be_sorted)
print("Sorted Array: ",QuickSort(array_to_be_sorted))
end=time.time()
print(f"Runtime of the program is {end - start}")
```

Output:

Original Array: [4, 2, 7, 3, 1, 6]

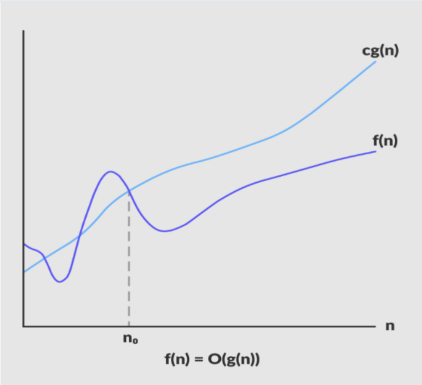
Sorted Array: [1, 2, 3, 4, 6, 7]

Runtime of the program is 0.046759843826293945

Time Complexity For open addressing:

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Insertion	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)



Best Case O(n*logn)

Average Case O(n*logn)

Worst Case O(n2)

3. Implement Fibonacci sequence with dynamic programming

Algorithm:

Step 1: [Defining Fibonacci function & import time]

```
import time
start = time.time()
def fibonacci(n):
    # Taking 1st two fibonacci numbers as 0 and 1
    f = [0, 1]
for i in range(2, n+1):
        f.append(f[i-1] + f[i-2])
    return f[n]
```

Step 2: [Output Operation]

```
print(fibonacci(9))
end = time.time()
print(f"Runtime of the program is {end - start}")
```

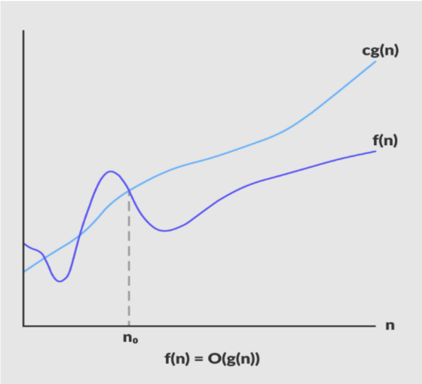
Python Code

```
import time
start = time.time()
def fibonacci(n):
    f = [0, 1]
    for i in range(2, n+1):
        f.append(f[i-1] + f[i-2])
    return f[n]
print(fibonacci(9))
end = time.time()
print(f"Runtime of the program is {end - start}")
```

Output:

34

Runtime of the program is 0.015629768371582031



Week 5

1. Implement Singly linked list (Traversing the Nodes, searching for a Node, Prepending Nodes, Removing Nodes)

Singly linked list

Algorithm

Step 1: [Create class node & import time]

```
import time
start = time.time()
class Node:
    def __init__(self,data):
        self.data = data
        self.ref = None
```

Step 2: [Create another class linkedlist]

```
class Linkedlist:
    def __init__(self):
        self.head = None
```

Step 3: [define functions to add, print, & delete inside class linked list & Searching given value]

```
    def print_LL(self):
        if self.head is None:
            print("linked list is empty!")
        else:
            n = self.head
            while n is not None:
                print(n.data)
```

```
            n = n.ref
    def add_begin(self,data):
        new_Node = Node(data)
        new_Node.ref = self.head
        self.head = new_Node
    def delete_by_value(self,x):
        if self.head is None:
            print("can't delete LL is empty !")
            return
        if x==self.head.data:
            self.head = self.head.ref
            return
        n = self.head
        while n.ref is not None:
            if x==n.ref.data:
                break
            n==n.ref
        if n.ref is None:
            print("Node is not present !")
        else:
            n.ref=n.ref.ref
def Search_value(self,x):
        if self.head is None:
            print("can't delete LL is empty !")
            return
```

```
            if x==self.head.data:
                self.head = self.head.ref
                print("Node is present at Beginning!")
                return
            n = self.head
            while n.ref is not None:
                if x==n.ref.data:
                    print("Node is present!")
                    break
                n==n.ref
```

Step 4: [Output operation]

```
        L1 = Linkedlist ()
        L1.add_begin (10)
        L1.add_begin (20)
        L1.add_begin (30)
        L1.delete_by_value(20)
        L1.print_LL()
        L1.Search_value (10)
        end = time.time()
        print(f"Runtime of the program is {end - start}")
```

Python Code

```
import time
start = time.time()
class Node:
```

```
    def __init__(self,data):
        self.data = data
        self.ref = None
class Linkedlist:
    def __init__(self):
        self.head = None
    def print_LL(self):
        if self.head is None:
            print("linked list is empty!")
        else:
            n = self.head
            while n is not None:
                print(n.data)
                n = n.ref
    def add_begin(self,data):
        new_Node = Node(data)
        new_Node.ref = self.head
        self.head = new_Node
    def delete_by_value(self,x):
        if self.head is None:
            print("can't delete LL is empty !")
            return
        if x==self.head.data:
            self.head = self.head.ref
            return
```

```
        n = self.head
        while n.ref is not None:
            if x==n.ref.data:
                break
            n==n.ref
        if n.ref is None:
            print("Node is not present !")
        else:
            n.ref=n.ref.ref
    def Search_value(self,x):
        if self.head is None:
            print("can't delete LL is empty !")
            return
        if x==self.head.data:
            self.head = self.head.ref
            print("Node is present at Beginning!")
            return
        n = self.head
        while n.ref is not None:
            if x==n.ref.data:
                print("Node is present!")
                break
            n==n.ref
L1 = Linkedlist ()
L1.add_begin (10)
```

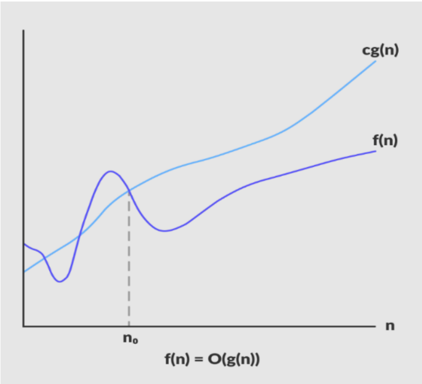
```
L1.add_begin (20)
L1.add_begin (30)
L1.delete_by_value(20)
L1.print_LL()
L1.Search_value (10)
end = time.time()
print(f"Runtime of the program is {end - start}")
```

Output:

30
10
Node is present!
Runtime of the program is 0.046753883361816406

Time Complexity For open addressing:

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)
Insertion	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)



The summary of Time Complexity of operations in a Linked List is:

SINGLY LINKED LIST OPERATION	**REAL TIME COMPLEXITY**	**ASSUMED TIME COMPLEXITY**
Access i-th element	O(√N * N)	O(N)
Traverse all elements	O(√N * N)	O(N)
Insert element E at current point	O(1)	O(1)
Delete current element	O(1)	O(1)
Insert element E at front	O(1)	O(1)
Insert element E at end	O(√N * N)	O(N)

The Space Complexity of the above Linked List operations is O(1).

Week 6

1. Implement linked list Iterators.

Algorithm

Step 1: [Create class Find odd & import time]

```
import time

start = time.time()

class FindOdd:
```

Step 2: [Define functions inside class]

```
def __init__(self,end):
    self.__start = 1
    self.__end = end
  def __iter__(self):
    return OddsIterator(self.__end)
```

Step 3: [Define another class Odds Iterator]

```
class OddsIterator:
  def __init__(self,finish):
    self.__current = 0
    self.__step = 1
    self.__end = finish
  def __next__(self):
```

```
        x = None
        if self.__current > self.__end:
            raise StopIteration
        else:
            self.__current += self.__step
            if (self.__current - self.__step + 1) % 2 != 0:
                x = self.__current - self.__step + 1
        if x != None:
            return x
```

Step 4: [Output Operation]

```
    odds = FindOdd(2)
    print(list(odds))
    end = time.time()
    print(f"Runtime of the program is {end - start}")
```

Python code

```
import time
start = time.time()
class FindOdd:
    def __init__(self,end):
        self.__start = 1
        self.__end = end
    def __iter__(self):
        return OddsIterator(self.__end)
class OddsIterator:
        def __init__(self,finish):
            self.__current = 0
            self.__step = 1
             self.__end = finish
    def __next__(self):
        x = None
        if self.__current > self.__end:
            raise StopIteration
        else:
            self.__current += self.__step
```

```
            if (self.__current - self.__step + 1) % 2 != 0:
                x = self.__current - self.__step + 1
        if x != None:
            return x
odds = FindOdd(2)
print(list(odds))
end = time.time()
```

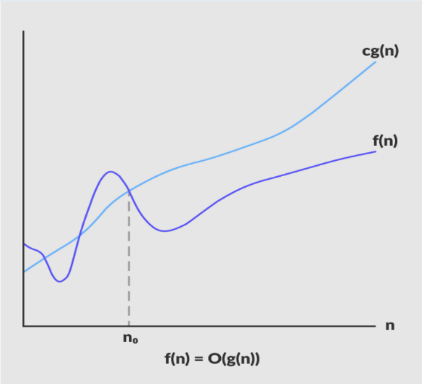
print(f"Runtime of the program is {end - start}")

Output:
[1, None, 3]

Runtime of the program is 0.015508651733398438

Time Complexity For open addressing:

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)
Insertion	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)



Time Complexity
As we are only traversing the list to the overall time complexity will be O(n).
Space complexity
Since no extra space is used the space complexity will be O(1).

Week7

1. Implement DLL

Algorithm

Step 1: [Create class & Initialize the node & import time]

```
import time
start = time.time()
class Node:
    def __init__(self, data):
        self.item = data
        self.next = None
        self.prev = None
```

Step 2: [Create another class for doubly linked list]

```
class doublyLinkedList:
    def __init__(self):
        self.start_node = None
```

Step 3: [Add Insert to Empty list function inside the class]

```
    def InsertToEmptyList(self, data):
        if self.start_node is None:
            new_node = Node(data)
            self.start_node = new_node
        else:
            print("The list is empty")
```

Step 4: [Add Insert to End function inside the class]

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

```
        if self.start_node is None:
            new_node = Node(data)
            self.start_node = new_node
            return
        n = self.start_node
        while n.next is not None:
            n = n.next
        new_node = Node(data)
        n.next = new_node
        new_node.prev = n
```

Step 5: [Define function to delete the elements from the end]

```
    def DeleteAtStart(self):
        if self.start_node is None:
            print("The Linked list is empty, no element to delete")
            return
        if self.start_node.next is None:
            self.start_node = None
            return
        self.start_node = self.start_node.next
        self.start_prev = None;
      def delete_at_end(self):
        if self.start_node is None:
            print("The Linked list is empty, no element to delete")
            return
        if self.start_node.next is None:
```

```
            self.start_node = None
            return
        n = self.start_node
        while n.next is not None:
            n = n.next
        n.prev.next = None
```

Step 6: [Create function to display & traversing]

```
    def Display(self):
        if self.start_node is None:
            print("The list is empty")
            return
        else:
            n = self.start_node
            while n is not None:
                print("Element is: ", n.item)
                n = n.next
        print("\n")
```

Step 7: [Function for searching element in list]

```
    def search(self,x):
        if self.start_node is None:
            print("The list is empty")
            return
        else:
            n = self.start_node
            while n is not None:
```

```
            if x==n.item:
             print("item present")
             break
             n = n.next
             print("\n")
```

Step 8: [Create elements to display output]

```
    NewDoublyLinkedList = doublyLinkedList()
    NewDoublyLinkedList.InsertToEmptyList(10)
    NewDoublyLinkedList.InsertToEnd(20)
    NewDoublyLinkedList.InsertToEnd(30)
    NewDoublyLinkedList.InsertToEnd(40)
    NewDoublyLinkedList.InsertToEnd(50)
    NewDoublyLinkedList.InsertToEnd(60)
```

Step 9: [Output Operation]

```
    NewDoublyLinkedList.Display()
    NewDoublyLinkedList.DeleteAtStart()
    NewDoublyLinkedList.DeleteAtStart()
    NewDoublyLinkedList.Display()
    NewDoublyLinkedList.search(30)
    end = time.time()
    print(f"Runtime of the program is {end - start}")
```

Python Code

```
import time
start = time.time()
```

```
class Node:
    def __init__(self, data):
        self.item = data
        self.next = None
        self.prev = None
class doublyLinkedList:
    def __init__(self):
        self.start_node = None
    def InsertToEmptyList(self, data):
        if self.start_node is None:
            new_node = Node(data)
            self.start_node = new_node
        else:
            print("The list is empty")
    def InsertToEnd(self, data):
        if self.start_node is None:
            new_node = Node(data)
            self.start_node = new_node
            return
        n = self.start_node
        while n.next is not None:
            n = n.next
        new_node = Node(data)
        n.next = new_node
        new_node.prev = n
```

```
def DeleteAtStart(self):
    if self.start_node is None:
        print("The Linked list is empty, no element to delete")
        return
    if self.start_node.next is None:
        self.start_node = None
        return
    self.start_node = self.start_node.next
    self.start_prev = None;
def delete_at_end(self):
    if self.start_node is None:
        print("The Linked list is empty, no element to delete")
        return
    if self.start_node.next is None:
        self.start_node = None
        return
    n = self.start_node
    while n.next is not None:
        n = n.next
    n.prev.next = None
def Display(self):
    if self.start_node is None:
        print("The list is empty")
        return
    else:
```

```
            n = self.start_node
            while n is not None:
                print("Element is: ", n.item)
                n = n.next
        print("\n")
    def search(self,x):
        if self.start_node is None:
            print("The list is empty")
            return
        else:
            n = self.start_node
            while n is not None:
                if x==n.item:
                 print("item present")
                 break
                 n = n.next
                 print("\n")
NewDoublyLinkedList = doublyLinkedList()
NewDoublyLinkedList.InsertToEmptyList(10)
NewDoublyLinkedList.InsertToEnd(20)
NewDoublyLinkedList.InsertToEnd(30)
NewDoublyLinkedList.InsertToEnd(40)
NewDoublyLinkedList.InsertToEnd(50)
NewDoublyLinkedList.InsertToEnd(60)
NewDoublyLinkedList.Display()
```

```
NewDoublyLinkedList.DeleteAtStart()
NewDoublyLinkedList.DeleteAtStart()
NewDoublyLinkedList.Display()
NewDoublyLinkedList.search(30)
end = time.time()
print(f"Runtime of the program is {end - start}")
```

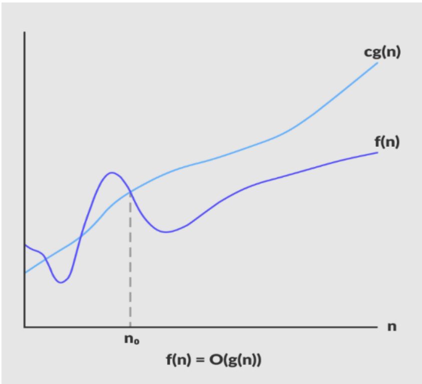
Output:

```
Element is:  10
Element is:  20
Element is:  30
Element is:  40
Element is:  50
Element is:  60


Element is:  30
Element is:  40
Element is:  50
Element is:  60


item present
Runtime of the program is 0.3124046325683594
```

Time Complexity For open addressing:

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)
Insertion	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)

Time and Space Complexity



The time complexity for searching a given element in the linked list is **O(N)** as we have to loop over all the nodes and check for the required one. The space complexity is **O(1)** as no additional memory is required to traverse through a doubly linked list and perform a search.

2. Implement CDLL

Step 1: [Create class node]

```
import time
start = time.time()
class Node:
  def __init__(self, my_data):
    self.data = my_data
```

```
        self.next = None
```

Step 2: [Create another class circular linked list]

```
class circularLinkedList:
  def __init__(self):
    self.head = None
  def add_data(self, my_data):
    ptr_1 = Node(my_data)
    temp = self.head
    ptr_1.next = self.head
    if self.head is not None:
      while(temp.next != self.head):
        temp = temp.next
      temp.next = ptr_1
    else:
      ptr_1.next = ptr_1
    self.head = ptr_1
```

Step 3: [Define output function]

```
  def print_it(self):
    temp = self.head
    if self.head is not None:
      while(True):
        print("%d" %(temp.data)),
        temp = temp.next
        if (temp == self.head):
          break
```

Step 4: [Output operation]

```
my_list = circularLinkedList()
print("Elements are added to the list ")
my_list.add_data (56)
my_list.add_data (78)
my_list.add_data (12)
print("The data is : ")
my_list.print_it()
end = time.time()
print(f"Runtime of the program is {end - start}")
```

Python code

```
import time
start = time.time()
class Node:
  def __init__(self, my_data):
    self.data = my_data
    self.next = None
class circularLinkedList:
  def __init__(self):
    self.head = None
  def add_data(self, my_data):
    ptr_1 = Node(my_data)
    temp = self.head
    ptr_1.next = self.head
```

```
        if self.head is not None:
            while(temp.next != self.head):
                temp = temp.next
            temp.next = ptr_1
        else:
            ptr_1.next = ptr_1
        self.head = ptr_1
    def print_it(self):
        temp = self.head
        if self.head is not None:
            while(True):
                print("%d" %(temp.data)),
                temp = temp.next
                if (temp == self.head):
                    break
my_list = circularLinkedList()
print("Elements are added to the list ")
my_list.add_data (56)
my_list.add_data (78)
my_list.add_data (12)
print("The data is : ")
my_list.print_it()
end = time.time()
print(f"Runtime of the program is {end - start}")
```

Output:

Elements are added to the list

The data is :

12

78

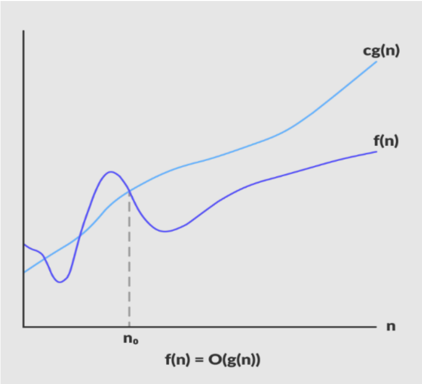
56

Runtime of the program is 0.06238698959350586

Time Complexity For open addressing:

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)
Insertion	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)

Time and Space Complexity



The time complexity for searching a given element in the linked list is **O(N)** as we have to loop over all the nodes and check for the required one. The space complexity is **O(1)** as no additional memory is required to traverse through a circular singly linked list and perform a search.

Week 8

1. Implement Stack Data

Step 1: [Create stack of list & import time]

```
import time
start = time.time()
stack = []
```

Step 2: [Function to push element in the stack]

```
stack.append('a')
stack.append('b')
stack.append('c')
print('Initial stack')
print(stack)
```

Step 3: [Function to pop element from stack]

```
print(stack.pop())
print(stack.pop())
print(stack.pop())
```

Step 4: [Output operation]

```
print('\nElements popped from stack:')
print('\nStack after elements are popped:')
print(stack)
end = time.time()
print(f"Runtime of the program is {end - start}")
```

Python Code

```
import time
start = time.time()
stack = []
stack.append('a')
stack.append('b')
stack.append('c')
print('Initial stack')
print(stack)
print('\nElements popped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())
print('\nStack after elements are popped:')
print(stack)
end = time.time()
print(f"Runtime of the program is {end - start}")
```

Output:

```
Initial stack
['a', 'b', 'c']

Elements popped from stack:
c
b
```

a

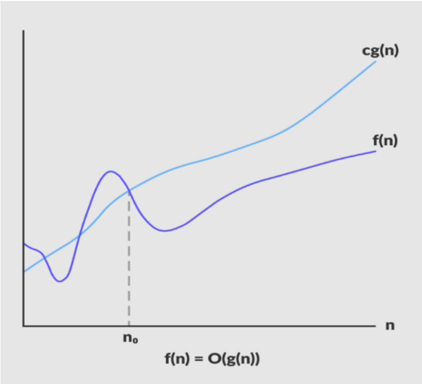
Stack after elements are popped:

[]

Runtime of the program is 0.10926008224487305

Time Complexity For open addressing:

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)
Insertion	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)



The time complexity of creating a Stack using a list is O(1) as it takes a constant amount of time to initialize a list. The space complexity is O(1) as well since no additional memory is required.

2. Implement Bracket matching using stack

Step 1: [Define function to balance Brackets & import time]

```
import time
```

```
start = time.time()
def areBracketsBalanced(expr):
    stack = []
```

Step 2: [Push the element & Traverse the expression in function]

```
    for char in expr:
        if char in ["(", "{", "["]:
            stack.append(char)
        else:
            if not stack:
                return False
            current_char = stack.pop()
            if current_char == '(':
                if char != ")":
                    return False
            if current_char == '{':
                if char != "}":
                    return False
            if current_char == '[':
                if char != "]":
                    return False
```

Step 3: [Condition to check the stack]

```
    if stack:
        return False
    return True
```

Step 4: [Source code to set the special variable]

```
if __name__ == "__main__":
    expr = "{()}[]"
    if areBracketsBalanced(expr):
        print("Balanced")
    else:
        print("Not Balanced")
end = time.time()
print(f"Runtime of the program is {end - start}")
```

Python Code

```
import time
start = time.time()
def areBracketsBalanced(expr):
    stack = []
    for char in expr:
        if char in ["(", "{", "["]:
            stack.append(char)
        else:
            if not stack:
                return False
            current_char = stack.pop()
            if current_char == '(':
                if char != ")":
                    return False
            if current_char == '{':
```

```
                if char != "}":
                    return False
            if current_char == '[':
                if char != "]":
                    return False
    if stack:
        return False
    return True
if __name__ == "__main__":
    expr = "{()}[]"
    if areBracketsBalanced(expr):
        print("Balanced")
    else:
        print("Not Balanced")
end = time.time()
print(f"Runtime of the program is {end - start}")
```

Output:

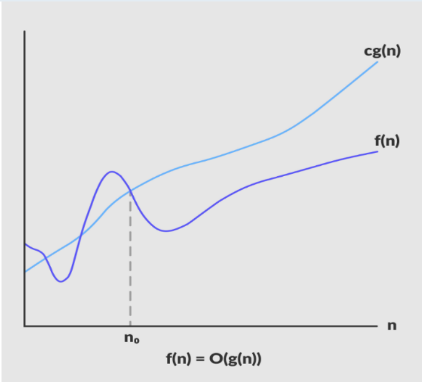
Balanced

Runtime of the program is 0.031132936477661133

Time Complexity For open addressing:

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)
Insertion	O(1)	O(1)	O(n)

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)



O(1) space | O(N^2) time complexity

Week 9

1. Program to demonstrate recursive operations (Factorial / Fibonacci)

Step 1: [Create function for recursive operations & import time]

```
import time
start = time.time()
def recur_fibo(n):
  if n <= 1:
     return n
  else:
     return(recur_fibo(n-1) + recur_fibo(n-2))
nterms = 10
```

Step 2: [Program to check if the number of terms is valid & output operation]

```
    if nterms <= 0:
      print("Plese enter a positive integer")
    else:
      print("Fibonacci sequence:")
      for i in range(nterms):
          print(recur_fibo(i))
    end = time.time()
    print(f"Runtime of the program is {end - start}")
```

Python Code

```
import time
start = time.time()
def recur_fibo(n):
  if n <= 1:
      return n
  else:
      return(recur_fibo(n-1) + recur_fibo(n-2))
nterms = 10
if nterms <= 0:
  print("Plese enter a positive integer")
else:
  print("Fibonacci sequence:")
  for i in range(nterms):
      print(recur_fibo(i))
end = time.time()
```

```
print(f"Runtime of the program is {end - start}")
```

Output:

Fibonacci Sequence:

0

1

1

2

3

5

8

13

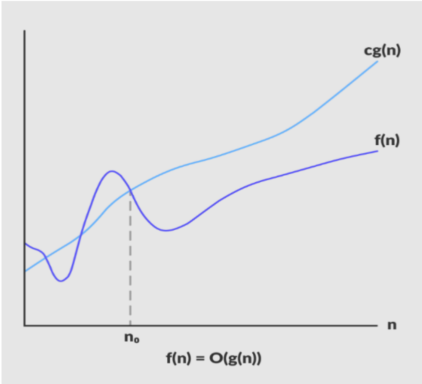
21

34

Runtime of the program is 0.10938119888305664

Time Complexity For open addressing:

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)
Insertion	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)



Space complexity is O(1) & time complexity O(n)

2. Implement Solution for Towers of Hanoi

Step 1: [Define function Towers of Hanoi & import time]

```
import time
start = time.time()
def TowerOfHanoi(n , source, destination, auxiliary):
   if n==1:
      print ("Move disk 1 from source",source,"to
destination",destination)
      return
```

Step 2: [Execution program]

```
TowerOfHanoi(n-1, source, auxiliary, destination)
   print ("Move disk",n,"from source",source,"to
destination",destination)
   TowerOfHanoi(n-1, auxiliary, destination, source)
end = time.time()
print(f"Runtime of the program is {end - start}")
```

Python Code

```
import time
start = time.time()
def TowerOfHanoi(n , source, destination, auxiliary):
    if n==1:
        print ("Move disk 1 from source",source,"to destination",destination)
        return
    TowerOfHanoi(n-1, source, auxiliary, destination)
    print ("Move disk",n,"from source",source,"to destination",destination)
    TowerOfHanoi(n-1, auxiliary, destination, source)
n = 3
TowerOfHanoi(n,'A','B','C')
end = time.time()
print(f"Runtime of the program is {end - start}")
```

Output:

Move disk 1 from source A to destination B

Move disk 2 from source A to destination C

Move disk 1 from source B to destination C

Move disk 3 from source A to destination B

Move disk 1 from source C to destination A

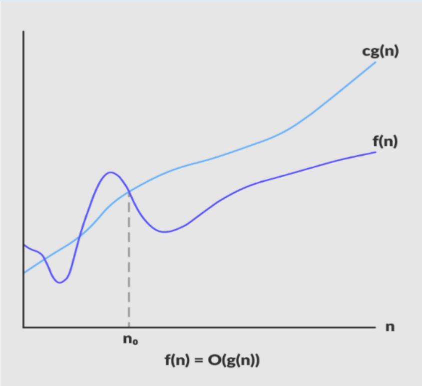
Move disk 2 from source C to destination B

Move disk 1 from source A to destination B

Runtime of the program is 0.29689645767211914

Time Complexity For open addressing:

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)
Insertion	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)



Space complexity is O(n). Here time complexity is exponential but space complexity is linear

Week 10

1. **Implement Queue**

Algorithm

Step 1: [Create class of Queue & import time]

```
import time
start = time.time()
class Queue:
```

Step 2: [Define functions of basic operations class queue]

```
    def __init__(self):
        self.items = []
    def isEmpty(self):
        return self.items == []
```

def enqueue(self,item):

self.items.append(item)

print(item)

def dequeue(self):

return self.items.pop(0)

def front(self):

return self.items[len(self.items)-1]

def size(self):

return len(self.items)

Step 3: [Output Operation]

print(q.isEmpty(),"- because queue is empty")

print("Enquee")

q.enqueue(15)

q.enqueue(16)

q.enqueue(17)

print("Front",q.front())

print("Dequee")

print(q.dequeue())

print(q.dequeue())

print("Size",q.size())

end = time.time()

print(f"Runtime of the program is {end - start}")

Python Code

import time

```
start = time.time()
class Queue:
    def __init__(self):
        self.items = []
    def isEmpty(self):
        return self.items == []
    def enqueue(self,item):
        self.items.append(item)
        print(item)
    def dequeue(self):
        return self.items.pop(0)
    def front(self):
        return self.items[len(self.items)-1]
    def size(self):
        return len(self.items)
q=Queue()
print(q.isEmpty(),"- because queue is empty")
print("Enquee")
q.enqueue(15)
q.enqueue(16)
q.enqueue(17)
print("Front",q.front())
print("Dequee")
print(q.dequeue())
print(q.dequeue())
print("Size",q.size())
end = time.time()
print(f"Runtime of the program is {end - start}")
```

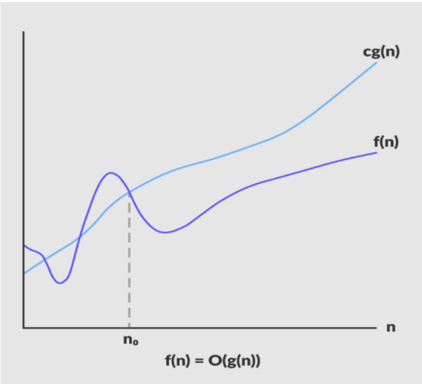
Output:

```
True - because queue is empty
Enquee
15
16
17
Front 17
```

Dequee
15
16
Size 1
Runtime of the program is 0.140514135360717771
Time Complexity For open addressing:

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)
Insertion	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)

Time and Space Complexity



The time complexity of creating a Queue using a list is **O(1)** as it takes a constant amount of time to initialize a list. The space complexity is **O(1)** as well since no additional memory is required.

2. **Implement priority queue**

Algorithm

Step 1: [Create class Priority queue & import time]

```
import time

start = time.time()

class PriorityQueue(object):
    def __init__(self):
```

```
            self.queue = []
        def __str__(self):
            return ' '.join([str(i) for i in self.queue])
        def isEmpty(self):
            return len(self.queue) == 0
        def insert(self, data):
            self.queue.append(data)
        def delete(self):
            try:
                max_val = 0
```
Step 2: [Checking the Condition]
```
            for i in range(len(self.queue)):
                if self.queue[i] > self.queue[max_val]:
                    max_val = i
            item = self.queue[max_val]
            del self.queue[max_val]
            return item
        except IndexError:
            print()
            exit()
```
Step 3: [Inserting in Queue]
```
    if __name__ == '__main__':
        myQueue = PriorityQueue()
        myQueue.insert(12)
        myQueue.insert(1)
        myQueue.insert(14)
        myQueue.insert(7)
```
Step 4: [Output Operation]
```
        print(myQueue)
        while not myQueue.isEmpty():
            print(myQueue.delete())
    end = time.time()

    print(f"Runtime of the program is {end - start}")
```

Python Code

```
import time
```

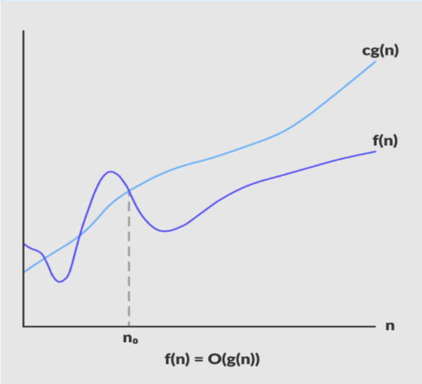
```
start = time.time()

class PriorityQueue(object):
    def __init__(self):
        self.queue = []
    def __str__(self):
        return ' '.join([str(i) for i in self.queue])
    def isEmpty(self):
        return len(self.queue) == 0
    def insert(self, data):
        self.queue.append(data)
    def delete(self):
        try:
            max_val = 0
            for i in range(len(self.queue)):
                if self.queue[i] > self.queue[max_val]:
                    max_val = i
            item = self.queue[max_val]
            del self.queue[max_val]
            return item
        except IndexError:
            print()
            exit()
if __name__ == '__main__':
    myQueue = PriorityQueue()
    myQueue.insert(12)
    myQueue.insert(1)
    myQueue.insert(14)
    myQueue.insert(7)
    print(myQueue)
    while not myQueue.isEmpty():
        print(myQueue.delete())
end = time.time()

print(f"Runtime of the program is {end - start}")
```

Output:

12 1 14 7

14
12
7
1
Runtime of the program is 0.07807803153991699
Time Complexity For open addressing:

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)
Insertion	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)



Time complexity: By Using heap data structure to implement Priority Queue

Insert Operation: 0(log(n))

Delete Operation: 0(log(n))

Week 11

1. Implement Binary search tree and its operations using list.

Algorithm

Step 1: [Creating a Binary tree by defining class initially & import time]

```
import time
start = time.time()
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
```

Step 2: [Inorder traversal function inside node]

```
def inorder(root):
    if root is not None:
        inorder(root.left)
        print(str(root.key) + "->", end=' ')
        inorder(root.right)
```

Step 3: [Inserting a Node]

```
def insert(node, key):
    if node is None:
        return Node(key)
    if key < node.key:
        node.left = insert(node.left, key)
    else:
        node.right = insert(node.right, key)
    return node
```

Step 4: [Find the inorder Successor]

```
def minValueNode(node):
    current = node
    while(current.left is not None):
        current = current.left
    return current
```

Step 5: [Deleting a node]

```
def deleteNode(root, key):
    if root is None:
        return root
    if key < root.key:
        root.left = deleteNode(root.left, key)
    elif(key > root.key):
        root.right = deleteNode(root.right, key)
    else:
        if root.left is None:
            temp = root.right
            root = None
            return temp
        elif root.right is None:
            temp = root.left
            root = None
            return temp
        temp = minValueNode(root.right)
        root.key = temp.key
```

Step 6: [Delete the inorder successor]

```
            root.right = deleteNode(root.right, temp.key)
        return root
```

Step 7: [Search the given node]

```
    def searchNode(root, key):
        if root.left is None:
            print("The element has not found.")
            return
        elif root.key == key:
            print("The element has been found.")
        elif key <root.key:
            if root.left.key == key:
                print("The element has been found.")
            else:
                searchNode(root.left,key)
        else:
            if root.right.key == key:
                print(key," element has been found.")
            else:
                searchNode(root.right,key)
```

Step 8: [Output operation]

```
    root = None
    root = insert(root, 8)
    root = insert(root, 3)
    root = insert(root, 1)
    root = insert(root, 6)
```

```
    root = insert(root, 7)
    root = insert(root, 10)
    root = insert(root, 14)
    root = insert(root, 4)
    print("Inorder traversal: ", end=' ')
    inorder(root)
    print("\nDelete 10")
    root = deleteNode(root, 10)
    print("Inorder traversal: ", end=' ')
    inorder(root)
    print("\n")
    searchNode(root,14)
    end = time.time()
    print(f"Runtime of the program is {end - start}")
```

Python Code

```
import time
start = time.time()
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
def inorder(root):
    if root is not None:
```

```
        inorder(root.left)
        print(str(root.key) + "->", end=' ')
        inorder(root.right)
def insert(node, key):
    if node is None:
        return Node(key)
    if key < node.key:
        node.left = insert(node.left, key)
    else:
        node.right = insert(node.right, key)
    return node
def minValueNode(node):
    current = node
    while(current.left is not None):
        current = current.left
    return current
def deleteNode(root, key):
    if root is None:
        return root
    if key < root.key:
        root.left = deleteNode(root.left, key)
    elif(key > root.key):
        root.right = deleteNode(root.right, key)
    else:
        if root.left is None:
```

```
            temp = root.right
            root = None
            return temp
        elif root.right is None:
            temp = root.left
            root = None
            return temp
        temp = minValueNode(root.right)
        root.key = temp.key
        root.right = deleteNode(root.right, temp.key)
    return root
def searchNode(root, key):
        if root.left is None:
            print("The element has not found.")
            return
        elif root.key == key:
            print("The element has been found.")
        elif key <root.key:
            if root.left.key == key:
                print("The element has been found.")
            else:
                searchNode(root.left,key)
        else:
            if root.right.key == key:
                print(key," element has been found.")
```

```
        else:
            searchNode(root.right,key)
root = None
root = insert(root, 8)
root = insert(root, 3)
root = insert(root, 1)
root = insert(root, 6)
root = insert(root, 7)
root = insert(root, 10)
root = insert(root, 14)
root = insert(root, 4)
print("Inorder traversal: ", end=' ')
inorder(root)
print("\nDelete 10")
root = deleteNode(root, 10)
print("Inorder traversal: ", end=' ')
inorder(root)
print("\n")
searchNode(root,14)
end = time.time()
print(f"Runtime of the program is {end - start}")
```

Output:

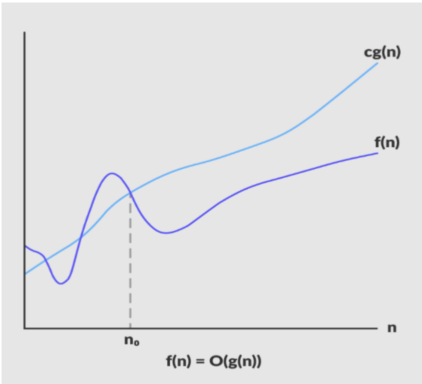
```
Inorder traversal:  1-> 3-> 4-> 6-> 7-> 8-> 10-> 14->
Delete 10
Inorder traversal:  1-> 3-> 4-> 6-> 7-> 8-> 14->
```

14 element has been found.
Runtime of the program is 0.29677271842956543

Binary Search Tree Complexities
Time Complexity

Operation	**Best Case Complexity**	**Average Case Complexity**	**Worst Case Complexity**
Search	O(log n)	O(log n)	O(n)
Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)

Here, n is the number of nodes in the tree.



Space Complexity: The space complexity for all the operations is O(n).

Week 12
1. Implementation of BFS.

Algorithm

Step 1: [Import queue & import time]

import time

start = time.time()

from queue import Queue

graph = {'A':['B','D','E','F'],'D':['A'],'B':['A','F','C'],'F':['B','A'],'C':['B'],'E':['A']}

print("Given Graph is : ")

print(graph)

Step 2: [Define BFS function]

```
def BFS(input_graph,source):
    Q = Queue()
    visited_vertices = list()
    Q.put(source)
    visited_vertices.append(source)
```

Step 3: [Checking the condition]

```
    while not Q.empty():
        vertex = Q.get()
        print(vertex,end= " ")
        for u in input_graph[vertex]:
            if u not in (visited_vertices):
                Q.put(u)
                visited_vertices.append(u)
```

Step 4: [Output operation]

print("BFS traversal of graph with source A is:")

BFS(graph,"A")

end = time.time()

```
    print(f"Runtime of the program is {end - start}")
```

Python Code

```
import time
start = time.time()
from queue import Queue
graph =
{'A':['B','D','E','F'],'D':['A'],'B':['A','F','C'],'F':['B','A'],'C':['B'],'E':['A']}
print("Given Graph is : ")
print(graph)
def BFS(input_graph,source):
    Q = Queue()
    visited_vertices = list()
    Q.put(source)
    visited_vertices.append(source)
    while not Q.empty():
        vertex = Q.get()
        print(vertex,end= " ")
        for u in input_graph[vertex]:
            if u not in (visited_vertices):
                Q.put(u)
                visited_vertices.append(u)
print("BFS traversal of graph with source A is:")
BFS(graph,"A")
end = time.time()
```

print(f"Runtime of the program is {end - start}")

Output:

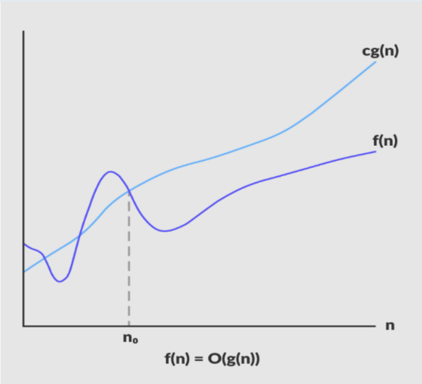
Given Graph is :

{'A': ['B', 'D', 'E', 'F'], 'D': ['A'], 'B': ['A', 'F', 'C'], 'F': ['B', 'A'], 'C': ['B'], 'E': ['A']}

BFS traversal of graph with source A is:
A B D E F C
Runtime of the program is 0.1092679500579834

Time Complexity For open addressing:

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)
Insertion	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)



Time complexity is O(|V|) , where |V| is the number of nodes. You need to traverse all nodes. Space complexity is O(|V|) as well - since at worst case you need to hold all vertices in the queue.23-Mar-2012

2. Implementations of DFS.

Algorithm

Step 1: [Define graph & import time]

```
import time
start = time.time()
graph =
{'A':['B','D','E','F'],'D':['A'],'B':['A','F','C'],'F':['B','A'],'C':['B'],'E':[
'A']}
print("Given Graph is : ")
print(graph)
```

Step 2: [Create DFS_traversal function]

```
def dfs_traversal(input_graph,source):
    stack = list()
    visited_list = list()
    stack.append(source)
    visited_list.append(source)
    while stack:
        vertex = stack.pop()
        print(vertex,end =" ")
        for u in input_graph[vertex]:
            if u not in visited_list:
                stack.append(u)
                visited_list.append(u)
```

Step 3: [Output operation]

```
print("DFS traversal of graph with source A is:")
dfs_traversal(graph,"A")
end = time.time()
print(f"Runtime of the program is {end - start}")
```

Python Code

```
import time
start = time.time()
graph =
{'A':['B','D','E','F'],'D':['A'],'B':['A','F','C'],'F':['B','A'],'C':['B'],'E':['A']}
```

```
print("Given Graph is : ")
print(graph)
def dfs_traversal(input_graph,source):
    stack = list()
    visited_list = list()
    stack.append(source)
    visited_list.append(source)
    while stack:
        vertex = stack.pop()
        print(vertex,end =" ")
        for u in input_graph[vertex]:
            if u not in visited_list:
                stack.append(u)
                visited_list.append(u)
print("DFS traversal of graph with source A is:")
dfs_traversal(graph,"A")
end = time.time()

print(f"Runtime of the program is {end - start}")
```

Output:

Given Graph is :
{'A': ['B', 'D', 'E', 'F'], 'D': ['A'], 'B': ['A', 'F', 'C'], 'F': ['B', 'A'], 'C': ['B'], 'E': ['A']}
DFS traversal of graph with source A is:

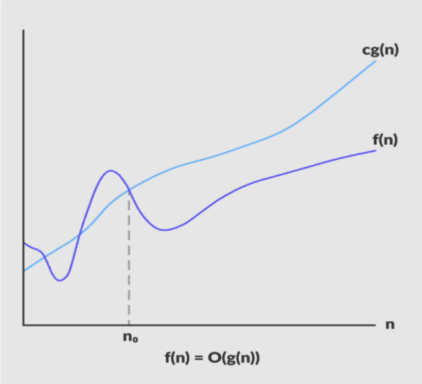
A F E D B C

Runtime of the program is 0.1561589241027832

Time Complexity For open addressing:

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)
Insertion	O(1)	O(1)	O(n)

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)



Time complexity: O(N) N is the number of nodes. · Space complexity: O(H) H is the height of the tree.

Week 13
1. Implement Hash functions

Step 1: [Define variables of different datatypes & import time]

```
import time

start = time.time()

int_val = 4
str_val = 'GeeksforGeeks'
flt_val = 24.56
```

Step 2: [Print the variables]

```
print("The integer hash value is : " + str(hash(int_val)))
print("The string hash value is : " + str(hash(str_val)))
print("The float hash value is : " + str(hash(flt_val)))
end = time.time()

print(f"Runtime of the program is {end - start}")
```

Python Code

```
import time

start = time.time()

int_val = 4
str_val = 'GeeksforGeeks'
flt_val = 24.56
print("The integer hash value is : " + str(hash(int_val)))
print("The string hash value is : " + str(hash(str_val)))
print("The float hash value is : " + str(hash(flt_val)))
end = time.time()

print(f"Runtime of the program is {end - start}")
```

Output:
The integer hash value is : 4
The string hash value is : -112777785087808093
The float hash value is : 1291272085159665688
Runtime of the program is 0.04675722122192383

Time Complexity For open addressing:

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)
Insertion	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)