THE MONOLOGUE OF A FAANG INTERVIEWER

Sehtab Hossain

Sorting Algorithm

1. Selection Sort
2. Bubble Sort
3. Insertion Sort

Divide & Conquer

4. Merge Sort
5. Quick Sort
6. Iteration
7. Recursion
8. Dynamic Programming
9. Heap Sort
10. Binary Search
11. Tree
12. Stack
13. Queue
14. Deque
15. Linked List
16. Hash map/ Hash table
17. Heap
18. Graph
19. Bloom Filter
20. Trie

Selection Sort

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where **n** is the number of items.

How Selection Sort Works?

Consider the following depicted array as an example.

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

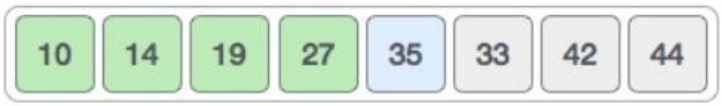
For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

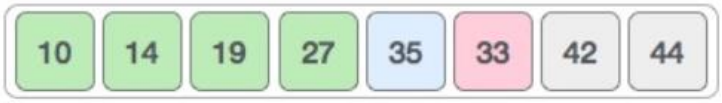
We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

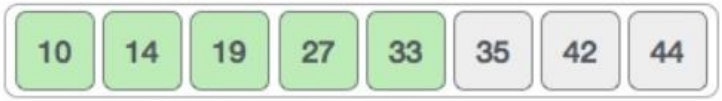
After two iterations, two least values are positioned at the beginning in a sorted manner.

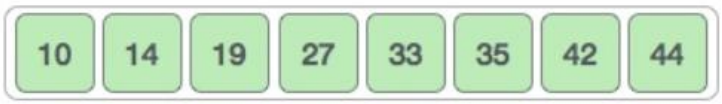
The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −









Now, let us learn some programming aspects of selection sort.

Algorithm

Step 1 – Set MIN to location 0
Step 2 – Search the minimum element in the list
Step 3 – Swap with value at location MIN
Step 4 – Increment MIN to point to next element
Step 5 – Repeat until list is sorted

Pseudocode

```
procedure selection sort
   list  : array of items
   n     : size of list

   for i = 1 to n - 1
   /* set current element as minimum*/
      min = i

      /* check the element to be minimum */

      for j = i+1 to n
         if list[j] < list[min] then
            min = j;
         end if
      end for

      /* swap the minimum element with the current element*/
      if indexMin != i  then
         swap list[min] and list[i]
      end if
   end for

end procedure
```

Selection Sort Complexity

Time Complexity	
Best	$O(n^2)$
Worst	$O(n^2)$
Average	$O(n^2)$
Space Complexity	O(1)
Stability	No

Cycle	Number of Comparison
1st	(n-1)
2nd	(n-2)
3rd	(n-3)
...	...
last	1

In [1]:

```
import timeit
start= timeit.default_timer()

def selectionSort(array, size):

    for ind in range(size):
        min_index = ind

        for j in range(ind + 1, size):
            # select the minimum element in every iteration
            if array[j] < array[min_index]:
                min_index = j
         # swapping the elements to sort the array
        (array[ind], array[min_index]) = (array[min_index], array[ind])

arr = [-2, 45, 0, 11, -9,88,-97,-202,747]
size = len(arr)
selectionSort(arr, size)
print('The array after sorting in Ascending Order by selection sort is:')
print(arr)
stop = timeit.default_timer()
execution_time = stop - start
print(f'Program Time: {execution_time} seconds')
```

```
The array after sorting in Ascending Order by selection sort is:
[-202, -97, -9, -2, 0, 11, 45, 88, 747]
Program Time: 0.00042508299975452246 seconds
```

```
import timeit
start= timeit.default_timer()

def selectionSort(array, size):

    for ind in range(size):
        min_index = ind

        for j in range(ind + 1, size):
            # select the minimum element in every iteration
            if array[j] < array[min_index]:
                min_index = j
         # swapping the elements to sort the array
        (array[ind], array[min_index]) = (array[min_index], array[ind])

arr = [-2, 45, 0, 11, -9,88,-97,-202,747]
size = len(arr)
selectionSort(arr, size)
print('The array after sorting in Ascending Order by selection sort is:')
print(arr)
stop = timeit.default_timer()
execution_time = stop - start
print(f'Program Time: {execution_time} seconds')
```

```
The array after sorting in Ascending Order by selection sort is:
[-202, -97, -9, -2, 0, 11, 45, 88, 747]
Program Time: 0.00042508299975452246 seconds
```

```
import timeit
start = timeit.default_timer()

def selectionsort(arr):
    for i in range(len(arr)):
        min = i
        for j in range(i+1, len(arr)):
            # select the min in every iteration
            if arr[j] < arr[min]:
                min = j
                # swapping the elements to sort the array
                (arr[i], arr[min]) = (arr[min], arr[i])
arr = [-2, 45, 0, 11, -9,88,-97,-202,747]
size = len(arr)
print("The array:")
print(arr)
selectionsort(arr)
print('The array after sorting in Ascending Order by selection sort is:')
print(arr)
stop = timeit.default_timer()
execution_time = stop - start
print(f'Program Time: {execution_time} seconds')
```

```
The array:
[-2, 45, 0, 11, -9, 88, -97, -202, 747]
The array after sorting in Ascending Order by selection sort is:
[-202, -97, -9, -2, 11, 0, 45, 88, 747]
Program Time: 0.0006713529983244371 seconds
```

2. Bubble Sort

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

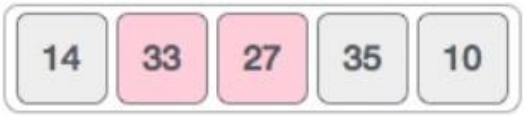
How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.

Bubble sort starts with very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.



The new array should look like this −

Next we compare 33 and 35. We find that both are in already sorted positions.

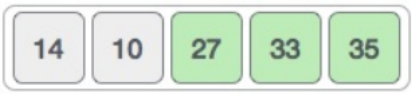
Then we move to the next two values, 35 and 10.

We know then that 10 is smaller 35. Hence they are not sorted.

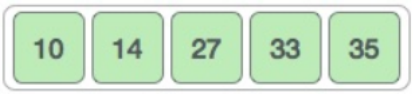
We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

Notice that after each iteration, at least one value moves at the end.



And when there's no swap required, bubble sorts learns that an array is completely sorted.



Now we should look into some practical aspects of bubble sort.

Algorithm

We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

```
begin BubbleSort(list)

   for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for

   return list

end BubbleSort
```

Pseudocode

We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.

To ease-out the issue, we use one flag variable **swapped** which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.

Pseudocode of BubbleSort algorithm can be written as follows −

```
procedure bubbleSort( list : array of items )

   loop = list.count;

   for i = 0 to loop-1 do:
      swapped = false

      for j = 0 to loop-1 do:

         /* compare the adjacent elements */
         if list[j] > list[j+1] then
            /* swap them */
            swap( list[j], list[j+1] )
            swapped = true
         end if

      end for

      /*if no number was swapped that means
      array is sorted now, break the loop.*/

      if(not swapped) then
         break
      end if

   end for

end procedure return list
```

The Time Complexity of the Bubble Sort Algorithm

- Bubble sort employs two loops: an inner loop and an outer loop.
- The inner loop performs O(n) comparisons deterministically.

Worst Case

- In the worst-case scenario, the outer loop runs O(n) times.
- As a result, the worst-case time complexity of bubble sort is O(n x n) = O(n x n) (n2).

Best Case

- In the best-case scenario, the array is already sorted, but just in case, bubble sort performs O(n) comparisons.
- As a result, the time complexity of bubble sort in the best-case scenario is O(n).

Average Case

- Bubble sort may require (n/2) passes and O(n) comparisons for each pass in the average case.
- As a result, the average case time complexity of bubble sort is O(n/2 x n) = O(n/2 x n) = O(n/2 x n) = O(n/2 x n) = O (n2).

The Space Complexity of the Bubble Sort Algorithm

- Bubble sort requires only a fixed amount of extra space for the flag, i, and size variables.
- As a result, the space complexity of bubble sort is O. (1).
- It is an in-place sorting algorithm, which modifies the original array's elements to sort the given array.

In this tutorial, you will see some of the benefits and drawbacks of the bubble sort.

```
import timeit
start = timeit.default_timer()
def bubblesort(customlist):
    # traverse through all the elements
    for i in range(len(customlist)-1):
        # range(n) works but outer loop will repeat 1 time more than needed.
        # Last i elements are already in place
        for j in range(len(customlist)-i-1):
            # traverse the array from 0 to n-i-1, swap if the element found is
greater than
            # the enxt element
            if customlist[j] > customlist[j+1]:
                customlist[j], customlist[j+1] = customlist[j+1], customlist[j]
    print(customlist)
clist = [2, 1, 7, 6 , 5, 3, 0, 9]
print("The array:")
print(clist)
bubblesort(clist)
print('The array after sorting in Ascending Order by selection sort is:')
print(clist)
stop = timeit.default_timer()
execution_time = stop - start
print(f'Program Time: {execution_time} seconds')
```

```
The array:
[2, 1, 7, 6, 5, 3, 0, 9]
[0, 1, 2, 3, 5, 6, 7, 9]
The array after sorting in Ascending Order by selection sort is:
[0, 1, 2, 3, 5, 6, 7, 9]
Program Time: 0.0005526450004254002 seconds
```

3. Insertion Sort

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where **n** is the number of items.

How Insertion Sort Works?

We take an unsorted array for our example.

Insertion sort compares the first two elements.

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion sort moves ahead and compares 33 with 27.

And finds that 33 is not in the correct position.

And finds that 33 is not in the correct position.

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

These values are not in a sorted order.

So we swap them.

However, swapping makes 27 and 10 unsorted.

Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

```
Step 1 − If it is the first element, it is already sorted. return 1;
Step 2 − Pick next element
Step 3 − Compare with all elements in the sorted sub-list
Step 4 − Shift all the elements in the sorted sub-list that is greater than the
value to be sorted
Step 5 − Insert the value
Step 6 − Repeat until list is sorted
```

Pseudocode

```
procedure insertionSort( A : array of items )
   int holePosition
   int valueToInsert

   for i = 1 to length(A) inclusive do:

      /* select value to be inserted */
      valueToInsert = A[i]
      holePosition = i

      /*locate hole position for the element to be inserted */

      while holePosition > 0 and A[holePosition-1] > valueToInsert do:
         A[holePosition] = A[holePosition-1]
         holePosition = holePosition -1
      end while
```

```
      /* insert the number at hole position */
      A[holePosition] = valueToInsert

   end for

end procedure
```

Complexity

- Worst case time complexity: `Θ(n^2)`
- Average case time complexity: `Θ(n^2)`
- Best case time complexity: `Θ(n)`
- Space complexity: `Θ(1)`

```
import timeit
start= timeit.default_timer()
def insertionsort(customlist):
    for i in range(1, len(customlist)):
        key = customlist[i]
        j = i-1
        while j >= 0 and key < customlist[j]:
            customlist[j+1] = customlist[j]
            j -= 1
        customlist[j+1] = key
    print(customlist)
clist = [2, 9, 3, 7, 1, 0, 7, 10 ]
print(insertionsort(clist))
stop = timeit.default_timer()
execution_time = stop - start
print(f'Program Time: {execution_time} seconds')
```

```
[0, 1, 2, 3, 7, 7, 9, 10]
None
Program Time: 0.0006874110003991518 seconds
```

Merge Sort (Divide & Conquer)

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

How Merge Sort Works?

To understand merge sort, we take an unsorted array as the following −

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

We further divide these arrays and we achieve atomic value which can no more be divided.

Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.

In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

After the final merging, the list should look like this −

Now we should learn some programming aspects of merge sorting.

Algorithm

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

```
Step 1 − if it is only one element in the list it is already sorted, return.
Step 2 − divide the list recursively into two halves until it can no more be
divided.
Step 3 − merge the smaller lists into new list in sorted order.
```

Pseudocode

We shall now see the pseudocodes for merge sort functions. As our algorithms point out two main functions − divide & merge.

Merge sort works with recursion and we shall see our implementation in the same way.

```
procedure mergesort( var a as array )
   if ( n == 1 ) return a

   var l1 as array = a[0] ... a[n/2]
   var l2 as array = a[n/2+1] ... a[n]

   l1 = mergesort( l1 )
   l2 = mergesort( l2 )

   return merge( l1, l2 )
end procedure

procedure merge( var a as array, var b as array )
```

```
  var c as array
  while ( a and b have elements )
     if ( a[0] > b[0] )
        add b[0] to the end of c
        remove b[0] from b
     else
        add a[0] to the end of c
        remove a[0] from a
     end if
  end while

  while ( a has elements )
     add a[0] to the end of c
     remove a[0] from a
  end while

  while ( b has elements )
     add b[0] to the end of c
     remove b[0] from b
  end while

  return c
```

```
end procedure
```

Complexity Analysis of Merge Sort

Merge Sort is quite fast, and has a time complexity of `O(n*log n)`. It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list.

In this section we will understand why the running time for merge sort is `O(n*log n)`.

As we have already learned in Binary Search that whenever we divide a number into half in every step, it can be represented using a logarithmic function, which is `log n` and the number of steps can be represented by `log n + 1`(at most)

Also, we perform a single step operation to find out the middle of any subarray, i.e. `O(1)`.

And to **merge** the subarrays, made by dividing the original array of `n` elements, a running time of `O(n)` will be required.

Hence the total time for `mergeSort` function will become `n(log n + 1)`, which gives us a time complexity of `O(n*log n)`.

Worst Case Time Complexity [Big-O]: **O(n*log n)**

Best Case Time Complexity [Big-omega]: **O(n*log n)**

Average Time Complexity [Big-theta]: **O(n*log n)**

Space Complexity: **O(n)**

- Time complexity of Merge Sort is `O(n*Log n)` in all the 3 cases (worst, average and best) as merge sort always **divides** the array in two halves and takes linear time to **merge** two halves.
- It requires **equal amount of additional space** as the unsorted array. Hence its not at all recommended for searching large unsorted arrays.

```
import timeit
start= timeit.default_timer()
# Python program for implementation of MergeSort
def mergeSort(arr):
        if len(arr) > 1:

                # Finding the mid of the array
                mid = len(arr)//2

                # Dividing the array elements
                L = arr[:mid]

                # into 2 halves
                R = arr[mid:]

                # Sorting the first half
                mergeSort(L)

                # Sorting the second half
                mergeSort(R)

                i = j = k = 0

                # Copy data to temp arrays L[] and R[]
                while i < len(L) and j < len(R):
                        if L[i] < R[j]:
                                arr[k] = L[i]
                                i += 1
                        else:
                                arr[k] = R[j]
                                j += 1
                        k += 1

                # Checking if any element was left
                while i < len(L):
                        arr[k] = L[i]
                        i += 1
                        k += 1

                while j < len(R):
                        arr[k] = R[j]
                        j += 1
                        k += 1

        for i in range(len(arr)):
                print(arr[i], end=" ")
        print()
```

```
# Code to print the list


def printList(arr):
        for i in range(len(arr)):
                print(arr[i], end=" ")
        print()


# Driver Code
if __name__ == '__main__':
        arr = [12, 11, 13, 5, 6, 7]
        print("Given array is", end="\n")
        printList(arr)
        mergeSort(arr)
        print("Sorted array is: ", end="\n")
        printList(arr)
stop = timeit.default_timer()
execution_time = stop - start
print(f'Program Time: {execution_time} seconds')
```

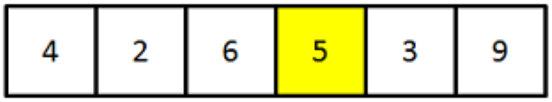
```
Given array is
12 11 13 5 6 7
Sorted array is:
5 6 7 11 12 13
Program Time: 0.0014851529999759805 seconds
```

Quick Sort (Divide & Conquer)

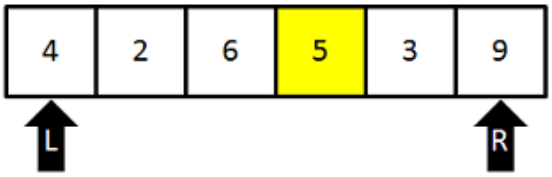
Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are $O(n^2)$, respectively.

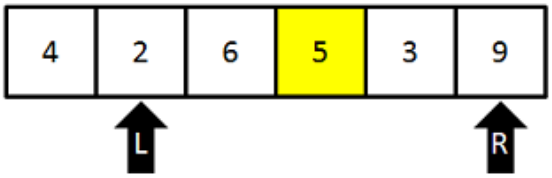
Step 1
Determine pivot



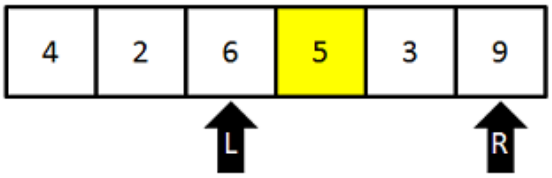
Step 2
Start pointers at left and right



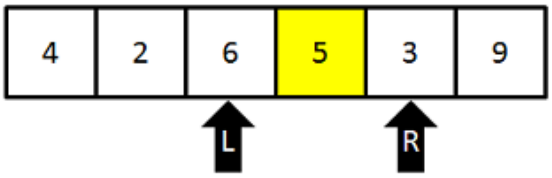
Step 3
Since 4 < 5, shift left pointer



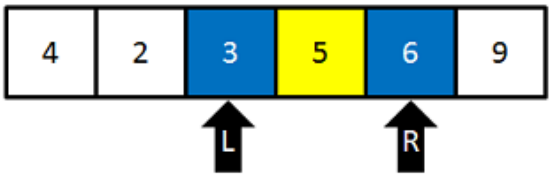
Step 4
Since 2 < 5, shift left pointer
Since 6 > 5, stop



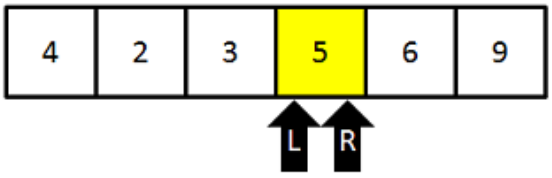
Step 5
Since 9 > 5, shift right pointer
Since 3 < 5, stop



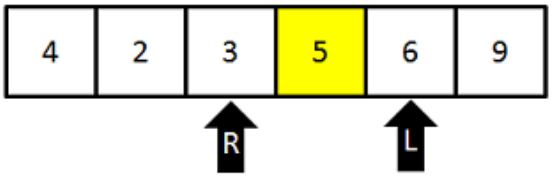
Step 6
Swap values at pointers

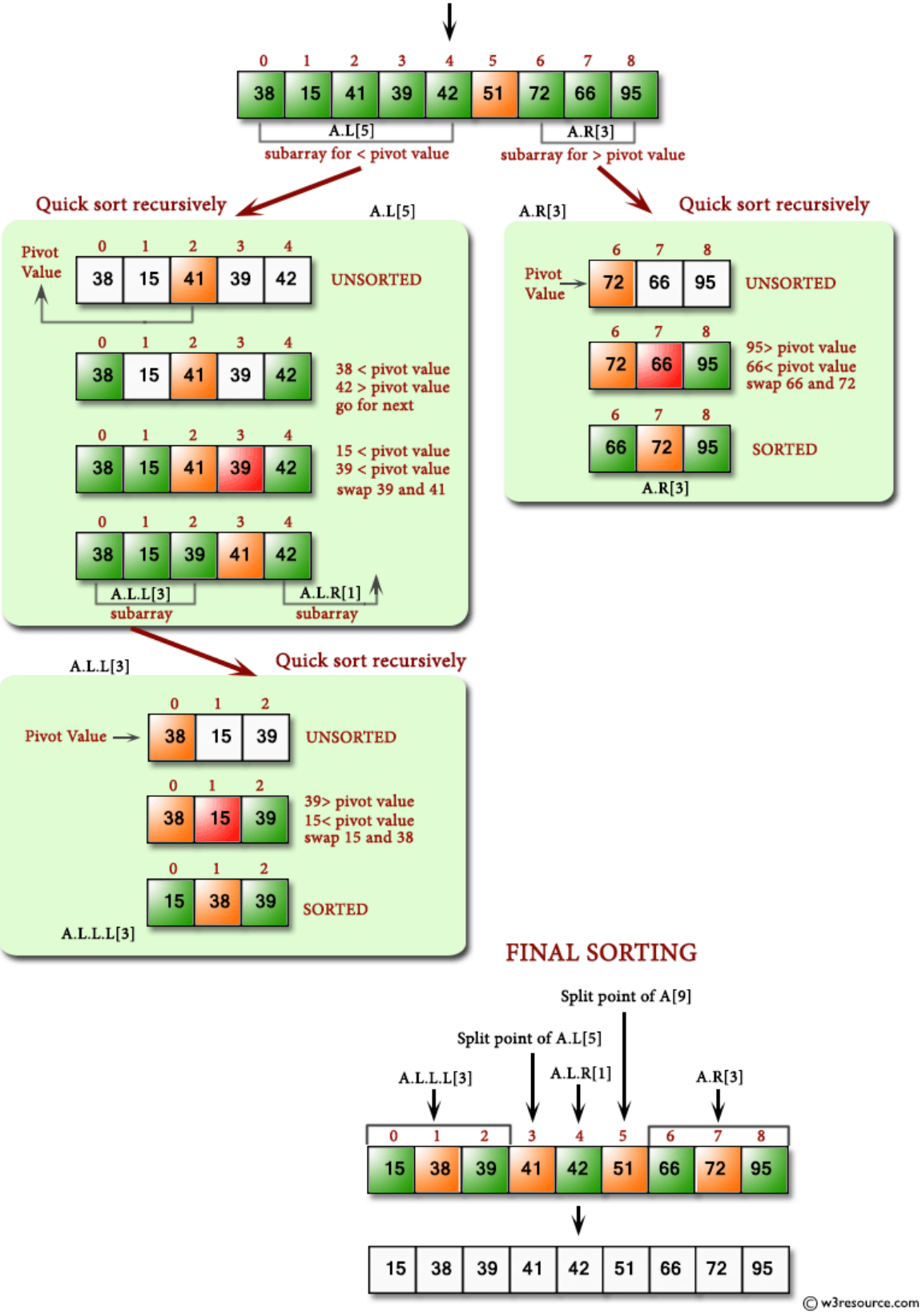


Step 7
Move pointers one more step



Step 8
Since 5 == 5,
move pointers one more step
Stop





The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.

Quick Sort Pivot Algorithm

Based on our understanding of partitioning in quick sort, we will now try to write an algorithm for it, which is as follows.

Step 1 – Choose the highest index value has pivot
Step 2 – Take two variables to point left and right of the list excluding pivot
Step 3 – left points to the low index
Step 4 – right points to the high
Step 5 – while value at left is less than pivot move right
Step 6 – while value at right is greater than pivot move left
Step 7 – if both step 5 and step 6 does not match swap left and right
Step 8 – if left ≥ right, the point where they met is new pivot

Quick Sort Pivot Pseudocode

The pseudocode for the above algorithm can be derived as –

```
function partitionFunc(left, right, pivot)
   leftPointer = left
   rightPointer = right - 1

   while True do
      while A[++leftPointer] < pivot do
         //do-nothing
      end while

      while rightPointer > 0 && A[--rightPointer] > pivot do
         //do-nothing
      end while

      if leftPointer >= rightPointer
         break
      else
         swap leftPointer,rightPointer
```

```
      end if

   end while

   swap leftPointer,right
   return leftPointer

end function
```

Quick Sort Algorithm

Using pivot algorithm recursively, we end up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quicksort as follows −

```
Step 1 − Make the right-most index value pivot
Step 2 − partition the array using pivot value
Step 3 − quicksort left partition recursively
Step 4 − quicksort right partition recursively
```

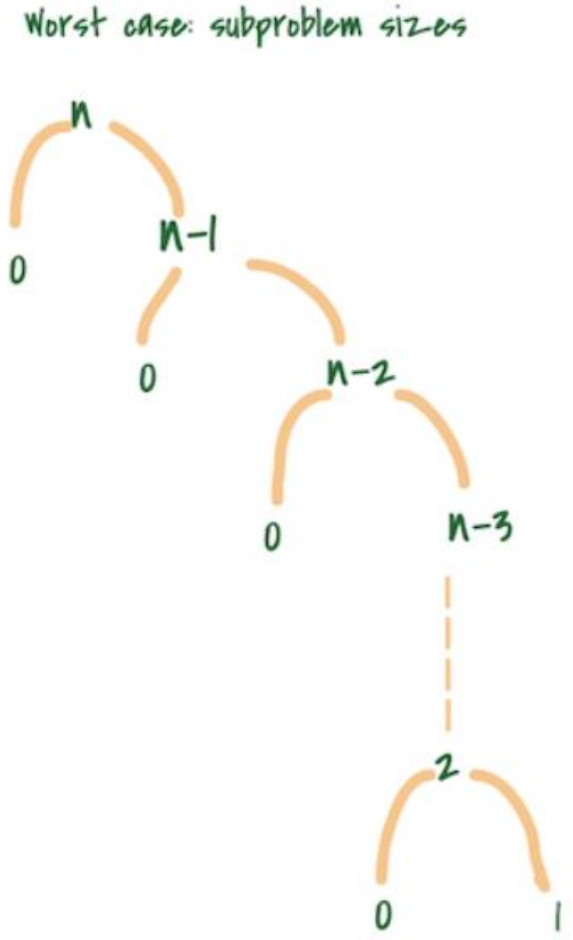
Quick Sort Pseudocode

To get more into it, let see the pseudocode for quick sort algorithm −

```
procedure quickSort(left, right)

   if right-left <= 0
      return
   else
      pivot = A[right]
      partition = partitionFunc(left, right, pivot)
      quickSort(left,partition-1)
      quickSort(partition+1,right)
   end if

end procedure
```

1. **Time complexity:**

The performance of quicksort is in order of `nlogn` for most of the cases. If the list of elements is already in sorted order or nearly sorted order then it takes n^2 comparisons to sort the array.



Yes, there are cases where Quicksort performs badly. If the list of elements is already in sorted order or nearly sorted then we might end up in an unbalanced sub-array, to an extent where there is no element on the left greater than the pivot, hence on the right.

Best case	Average case	Wrost case
O(nlogn)	O(nlogn)	$O(n^2)$

```
import timeit
start= timeit.default_timer()
def QuickSort(arr):

    elements = len(arr)

    #Base case
    if elements < 2:
        return arr

    current_position = 0 #Position of the partitioning element

    for i in range(1, elements): #Partitioning loop
         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



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))
stop = timeit.default_timer()
execution_time = stop - start
print(f'Program Time: {execution_time} seconds')
```

```
Original Array:  [4, 2, 7, 3, 1, 6]
Sorted Array:  [1, 2, 3, 4, 6, 7]
Program Time: 0.0006116779995863908 seconds
```

Iteration

Iteration is the repetition of a process in a computer program. Iterations of functions are common in computer programming, since they allow multiple blocks of data to be processed in sequence. This is typically done using a "while loop" or "for loop". These loops will repeat a process until a certain number or case is reached.

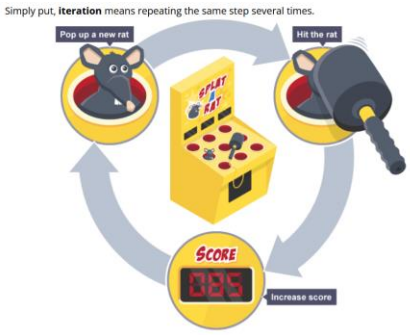


Figure 6 – Iteration: Repeating the same step several times
Source: okclipart.com

Iterative algorithms should obey three important principles:

1. An iterative process repeats (iterates) a certain sub-process.
2. Each iteration should change at least one value.
3. There should be some condition under which the iteration terminates. And the iteration should reach that state.

A very simple algorithm for eating breakfast cereal might consist of these steps (BBC Bitesize KS3 Subjects: *Iteration*):

> *(1) put cereal in bowl*
> *(2) add milk to cereal*
> *(3) spoon cereal and milk into mouth*
> *(3.1)* ***repeat*** *step 3* ***until*** *all cereal and milk is eaten*
> *(4) rinse bowl and spoon*

What we see in step 3.1 is the introduction of a **condition** which is a situation that is checked every time an iteration occurs. The condition is introduced with the words **repeat** and **until**. Without the condition, the algorithm would not know when to stop. The condition, in this case, will be to check if all the milk and cereals is eaten. If that condition is False (there is still milk and cereals in the bowl), then another iteration occurs. If the condition is True (there is no more milk and cereals in the bowl), then no more iterations occur.

The general form is:

```
repeat
    part of the algorithm
until condition
```

This means that the part of the algorithm between *repeat* and *until* is executed repeatedly until the condition specified after *until* holds true. The condition which comes after *until* is called a **terminating condition**.

The occurrence of iteration is called a loop.

Iteration allows algorithms to be simplified by stating that certain steps will repeat until told otherwise. This makes designing algorithms quicker and simpler because they don't need to include lots of unnecessary steps (BBC Bitesize KS3 Subjects: Iteration).

A more advanced use of iteration is the use of **nested loops**. A nested loop is a loop within a loop, an **inner loop** within the body of an **outer** one. How this works is that the first pass of the outer loop triggers the inner loop, which executes to completion. Then, the second pass of the outer loop triggers the inner loop again. This repeats until the outer loop finishes.

When one loop is nested within another, several iterations of the inner loop are performed for every single iteration of the outer loop.

Example: Suppose we have data in the form below, involving several ID strings. For each ID string, a variable number of readings have been recorded; the number of readings for each ID is shown in the howMany column:

ID	howMany	Readings
200	3	110 30 10
201	5	2 46 109 57 216
202	2	10 500

Our task is to read the data and display a summary chart showing the average of each reading per ID as follows**:**

ID	AVERAGE
200	50
201	86
202	255

This is how the algorithm could look like:

```
read first ID and howMany
    repeat
        display ID
            repeat
                read and sum up this ID's howMany readings
                calculate and display average for ID
            until end of howMany readings
            read next ID
    until end of ID
```

Looking at the algorithm we easily observe that the outer loop controls the number of lines and the inner loop controls the content of each line. So, 3 IDs leads to 3 rows and each row will display just one average calculated by adding all readings to get their sum and dividing by the number of readings to get the average.

Summary of most important algorithmic constructs

Sequence: A series of steps that are carried out sequentially in the order in which they are written. Each step is carried out only once

Selection: One of two or more alternatives should be chosen.

Iteration: Part of the algorithm should be capable for repetition, either a defined number of times or until a certain condition has been met.

Differences between iterative and recursive algorithms

Both iteration and recursion are based on a control structure: Iteration uses a repetition structure; recursion uses a selection structure. An **Iterative algorithm** will use looping statements such as for loop, while loop or do-while loop to repeat the same steps while a **Recursive algorithm,** a module (function) calls itself again and again till the *base condition*(stopping condition) is satisfied.

An Iterative algorithm will be faster than the Recursive algorithm because of overheads like calling functions and registering stacks repeatedly. Many times the recursive algorithms are not efficient as they take more space and time.

Recursive algorithms are mostly used to solve complicated problems when their application is easy and effective. For example Tower of Hannoi algorithm is made easy by recursion while iterations are widely used, efficient and popular.

Recursive vs Iterative Algorithms:

- **Approach:** In recursive approach, the function calls itself until the condition is met, whereas, in iterative approach, a function repeats until the condition fails.
- **Programming Construct Usage:** Recursive algorithm uses a branching structure, while iterative algorithm uses a looping construct.
- **Time & Space Effectiveness:** Recursive solutions are often less efficient in terms of time and space, when compared to iterative solutions.
- **Termination Test:** Iteration terminates when the loop-continuation condition fails; recursion terminates when a base case is recognized.
- **Infinite Call:** An infinite loop occurs with iteration if the loop-continuation test never becomes false; infinite recursion occurs if the recursion step does not reduce the problem in a manner that converges on the base case.

A practical problem: The birthday guy cuts the birthday cake and has to ensure that everyone in the room gets a slice.

Solution 1 – Iterative: The birthday guy uses a tray and goes around giving everyone a slice.

serveslice (while there are people to serve or slices left give slice)

Solution 2 – Recursive: Take a slice of cake from the tray and pass the tray to the next person who takes a slice from the tray and passes the tray to the next person, who takes a slice from the tray and passes the tray to the next person...

Note that the same function is being performed every time.

```
takeslice(tray)
If there are people to serve or slices left
    take slice and call takeslice(tray)
else return
```

Use of for-in (or for each) style: This style is used in python containing iterator of lists, dictionary, n dimensional-arrays etc. The iterator fetches each component and prints data while looping. The iterator is automatically incremented/decremented in this construct.

 # Accessing items using for-in loop

```
cars = ["Aston", "Audi", "McLaren"]
for x in cars:
        print x
```

Output:

```
Aston
Audi
McLaren
```

See this for more examples of different data types. **Indexing using Range function:** We can also use indexing using range() in Python.

 # Accessing items using indexes and for-in

```
cars = ["Aston", "Audi", "McLaren"]
```

```
for i in range(len(cars)):
        print cars[i]
```

Output:

```
Aston
Audi
McLaren
```

Enumerate: Enumerate is built-in python function that takes input as iterator, list etc and returns a tuple containing index and data at that index in the iterator sequence. For example, enumerate(cars), returns a iterator that will return (0, cars[0]), (1, cars[1]), (2, cars[2]), and so on.

```
# Accessing items using enumerate()

cars = ["Aston" , "Audi", "McLaren "]
for i, x in enumerate(cars):
        print (x)
```

Output :

```
Aston
Audi
McLaren
```

```
# Accessing items and indexes enumerate()

cars = ["Aston" , "Audi", "McLaren "]
for x in enumerate(cars):
        print (x[0], x[1])
```

Output :

```
(0, 'Aston')
(1, 'Audi')
(2, 'McLaren ')
```

We can also directly print returned value of enumerate() to see what it returns.

 # Printing return value of enumerate()

cars = ["Aston" , "Audi", "McLaren "]

print enumerate(cars)

Output :

```
[(0, 'Aston'), (1, 'Audi'), (2, 'McLaren ')]
```

Enumerate takes parameter start which is default set to zero. We can change this parameter to any value we like. In the below code we have used start as 1.

 # demonstrating the use of start in enumerate

cars = ["Aston" , "Audi", "McLaren "]

for x in enumerate(cars, start=1):

print (x[0], x[1])

Output :

```
(1, 'Aston')
(2, 'Audi')
(3, 'McLaren ')
```

enumerate() helps to embed solution for accessing each data item in the iterator and fetching index of each data item.

Looping extensions: i) Two iterators for a single looping construct: In this case, a list and dictionary are to be used for each iteration in a single looping block using enumerate function. Let us see example.

```
# Two separate lists
cars = ["Aston", "Audi", "McLaren"]
accessories = ["GPS kit", "Car repair-tool kit"]

# Single dictionary holds prices of cars and
# its accessories.
# First three items store prices of cars and
# next two items store prices of accessories.
prices = {1:"570000$", 2:"68000$", 3:"450000$",
                4:"8900$", 5:"4500$"}

# Printing prices of cars
for index, c in enumerate(cars, start=1):
        print "Car: %s Price: %s"%(c, prices[index])

# Printing prices of accessories
for index, a in enumerate(accessories,start=1):
        print ("Accessory: %s Price: %s"\
                %(a,prices[index+len(cars)]))
```

Output:

```
Car: Aston Price: 570000$
Car: Audi Price: 68000$
Car: McLaren Price: 450000$
Accessory: GPS kit Price: 8900$
Accessory: Car repair-tool kit Price: 4500$
```

ii) zip function (Both iterators to be used in single looping construct): This function is helpful to combine similar type iterators(list-list or dict- dict etc,) data items at ith position. It uses the shortest length of these input iterators. Other items of larger length iterators are skipped. In case of empty iterators, it returns No output.

For example, the use of zip for two lists (iterators) helped to combine a single car and its required accessory.

```
# Python program to demonstrate the working of zip

# Two separate lists
cars = ["Aston", "Audi", "McLaren"]
accessories = ["GPS", "Car Repair Kit",
                        "Dolby sound kit"]

# Combining lists and printing
for c, a in zip(cars, accessories):
        print "Car: %s, Accessory required: %s"\
                %(c, a)
```

Output:

```
Car: Aston, Accessory required: GPS
Car: Audi, Accessory required: Car Repair Kit
Car: McLaren, Accessory required: Dolby sound kit
```

The reverse of these iterators from zip function is known as unzipping using "*" operator. Use of enumerate function and zip function helps to achieve an effective extension of iteration logic in python and solves many more sub-problems of a huge task or problem.

```
# Python program to demonstrate unzip (reverse
# of zip)using * with zip function


# Unzip lists
l1,l2 = zip(*[('Aston', 'GPS'),
              ('Audi', 'Car Repair'),
              ('McLaren', 'Dolby sound kit')
             ])

# Printing unzipped lists
print(l1)
print(l2)
```

Output:

```
('Aston', 'Audi', 'McLaren')
('GPS', 'Car Repair', 'Dolby sound kit')
```

Recursion

What Is a Recursive Algorithm?

A recursive algorithm calls itself with smaller input values and returns the result for the current input by carrying out basic operations on the returned value for the smaller input. Generally, if a problem can be solved by applying solutions to smaller versions of the same problem, and the smaller versions shrink to readily solvable instances, then the problem can be solved using a recursive algorithm.

To build a recursive algorithm, you will break the given problem statement into two parts. The first one is the base case, and the second one is the recursive step.

- Base Case: It is nothing more than the simplest instance of a problem, consisting of a condition that terminates the recursive function. This base case evaluates the result when a given condition is met.
- Recursive Step: It computes the result by making recursive calls to the same function, but with the inputs decreased in size or complexity.

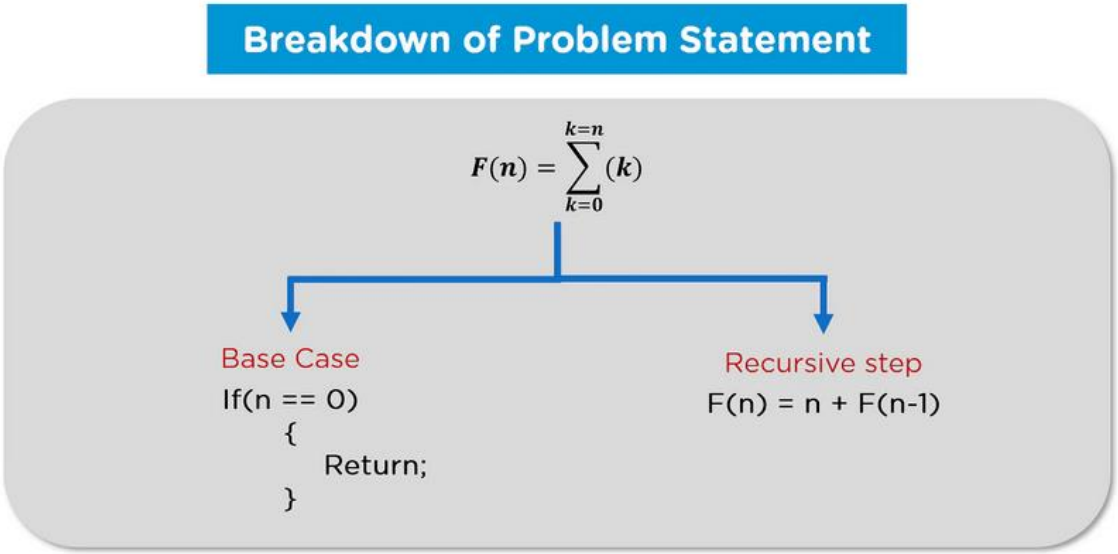
For example, consider this problem statement: Print sum of n natural numbers using recursion. This statement clarifies that we need to formulate a function that will calculate the summation of all natural numbers in the range 1 to n. Hence, mathematically you can represent the function as:

F(n) = 1 + 2 + 3 + 4 + ……..+ (n-2) + (n-1) + n

It can further be simplified as:

$$\mathbf{F(n)} = \sum_{k=1}^{k=n} \mathbf{(k)}$$

You can breakdown this function into two parts as follows:



Different Types of Recursion

There are four different types of recursive algorithms, you will look at them one by one.

- ## Direct Recursion

A function is called direct recursive if it calls itself in its function body repeatedly. To better understand this definition, look at the structure of a direct recursive program.

```
int fun(int z){

fun(z-1);  //Recursive call

}
```

In this program, you have a method named fun that calls itself again in its function body. Thus, you can say that it is direct recursive.

• Indirect Recursion

The recursion in which the function calls itself via another function is called indirect recursion. Now, look at the indirect recursive program structure.

```
int fun1(int z){      int fun2(int y){

    fun2(z-1);            fun1(y-2)

    }                     }
```

In this example, you can see that the function fun1 explicitly calls fun2, which is invoking fun1 again. Hence, you can say that this is an example of indirect recursion.

• Tailed Recursion

A recursive function is said to be tail-recursive if the recursive call is the last execution done by the function. Let's try to understand this definition with the help of an example.

```
int fun(int z)

    {

printf("%d",z);

  fun(z-1);

//Recursive call is last executed statement

    }
```

If you observe this program, you can see that the last line ADI will execute for method fun is a recursive call. And because of that, there is no need to remember any previous state of the program.

• Non-Tailed Recursion

A recursive function is said to be non-tail recursive if the recursion call is not the last thing done by the function. After returning back, there is something left to evaluate. Now, consider this example.

```
int fun(int z)

{

fun(z-1);

printf("%d",z);

//Recursive call is not the last executed statement

}
```

In this function, you can observe that there is another operation after the recursive call. Hence the ADI will have to memorize the previous state inside this method block. That is why this program can be considered non-tail recursive.

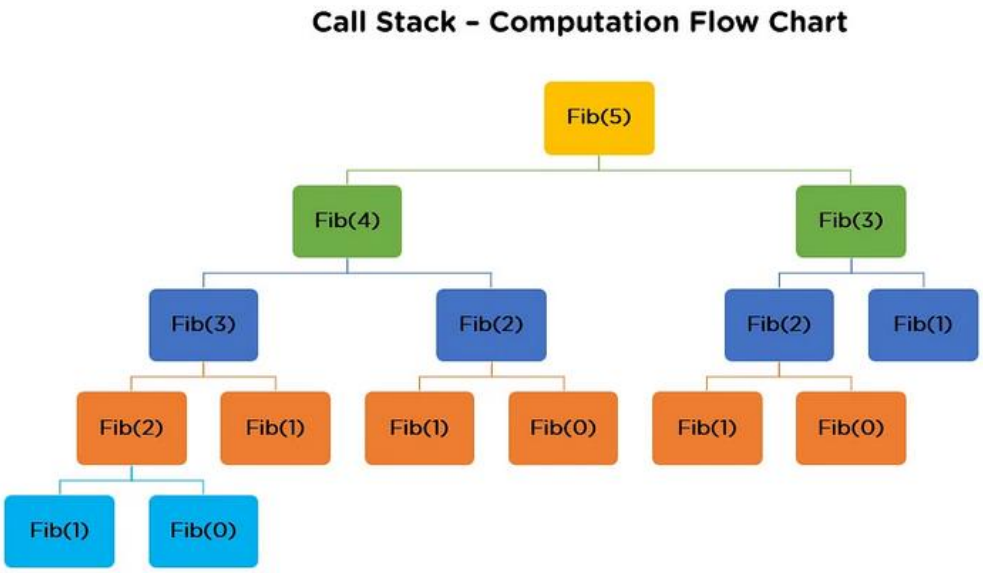
Memory Allocation of Recursive Method

Each recursive call generates a new copy of the function on stack memory. Once the procedure returns some data, the copy is deleted from storage. Each recursive call maintains a separate stack because all parameters and other variables defined inside functions are kept on the stack. The stack is deleted after the value from the relevant function is returned.

Recursion is quite complicated in terms of resolving and monitoring the values at each recursive call. As a result, you have to maintain the stack and track the values of the variables specified in it. To better understand the memory allocation of recursive functions, examine the following example.

```
//Fibonacci program recursive Function

int Fib(int num)

{

if ( num == 0 )

return 0;

else if ( num == 1 )

return 1;

else

return ( Fib(num-1) + Fib(num - 2) );

}

//Let's say we want to calculate Fibonacci number for n=5

Fib(5)
```

Now, have a look at this recursive Fibonacci code for n = 5. First, all stacks are preserved, each of which prints the matching value of n until n becomes zero. When the termination condition is achieved, the stacks are destroyed one at a time by returning 0 to their calling stack. To understand the call stack hierarchy, look at the figure below.




```
# recursion
def printFun(test):
        if (test < 1):
                return
        else:

                print(test, end=" ")
                printFun(test-1) # statement 2
                print(test, end=" ")
                return


# Driver Code
test = 3
printFun(test)
```

Problem 1: Write a program and recurrence relation to find the Fibonacci series of n where n>2 .

Mathematical Equation:

```
n if n == 0, n == 1;
fib(n) = fib(n-1) + fib(n-2) otherwise;
```

Recurrence Relation:

```
T(n) = T(n-1) + T(n-2) + O(1)
```

Recursive program:

```
Input: n = 5
Output:
Fibonacci series of 5 numbers is : 0 1 1 2 3
```

```
# Python code to implement Fibonacci series


# Function for fibonacci
def fib(n):


        # Stop condition
        if (n == 0):
                return 0


        # Stop condition
        if (n == 1 or n == 2):
                return 1


        # Recursion function
        else:
                return (fib(n - 1) + fib(n - 2))
```

```
# Driver Code
# Initialize variable n.
n = 5;
print("Fibonacci series of 5 numbers is :",end=" ")

# for loop to print the fibonacci series.
for i in range(0,n):
        print(fib(i),end=" ")
```

Output

```
Fibonacci series of 5 numbers is: 0 1 1 2 3
```

Time Complexity: $O(2^n)$

Auxiliary Space: $O(n)$

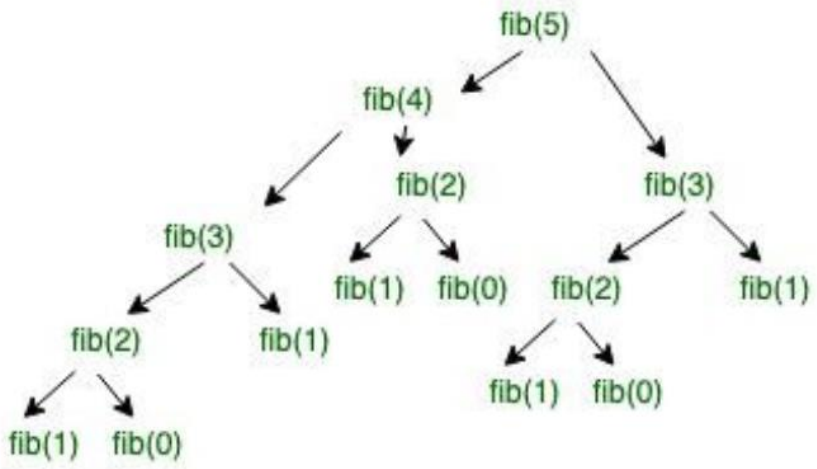
Here is the recursive tree for input 5 which shows a clear picture of how a big problem can be solved into smaller ones.
fib(n) is a Fibonacci function. The time complexity of the given program can depend on the function call.

*fib(n) -> level CBT (UB) -> 2^n-1 nodes -> 2^n function call -> 2^n*O(1) -> T(n) = O(2^n)*

For Best Case.

```
T(n) =   θ(2^n\2)
```

Working:



Problem 2: Write a program and recurrence relation to find the Factorial of n where n>2 .
Mathematical Equation:

```
1 if n == 0 or n == 1;
f(n) = n*f(n-1) if n> 1;
```

Recurrence Relation:

```
T(n) = 1 for n = 0
T(n) = 1 + T(n-1) for n > 0
```

Recursive Program:
Input: n = 5
Output:
factorial of 5 is: 120

```
# Python3 code to implement factorial

# Factorial function

def f(n):


        # Stop condition

        if (n == 0 or n == 1):

                return 1;


        # Recursive condition

        else:

                return n * f(n - 1);



# Driver code

if __name__=='__main__':


        n = 5;

        print("factorial of",n,"is:",f(n))
```

Output

```
factorial of 5 is: 120
```

Time complexity: $O(2^n)$.

Auxiliary Space: $O(n)$

Working:

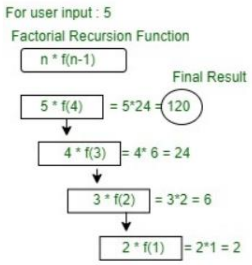


Diagram of factorial Recursion function for user input 5.

What are the disadvantages of recursive programming over iterative programming?

Note that both recursive and iterative programs have the same problem-solving powers, i.e., every recursive program can be written iteratively and vice versa is also true. The recursive program has greater space requirements than the iterative program as all functions will remain in the stack until the base case is reached. It also has greater time requirements because of function calls and returns overhead.

Moreover, due to the smaller length of code, the codes are difficult to understand and hence extra care has to be practiced while writing the code. The computer may run out of memory if the recursive calls are not properly checked.

What are the advantages of recursive programming over iterative programming?

Recursion provides a clean and simple way to write code. Some problems are inherently recursive like tree traversals, Tower of Hanoi, etc. For such problems, it is preferred to write recursive code. We can write such codes also iteratively with the help of a stack data structure. For example refer Inorder Tree Traversal without Recursion, Iterative Tower of Hanoi.

Summary of Recursion:

- There are two types of cases in recursion i.e. recursive case and a base case.
- The base case is used to terminate the recursive function when the case turns out to be true.
- Each recursive call makes a new copy of that method in the stack memory.
- Infinite recursion may lead to running out of stack memory.
- Examples of Recursive algorithms: Merge Sort, Quick Sort, Tower of Hanoi, Fibonacci Series, Factorial Problem, etc.

Dynamic Programming

In this tutorial, you will learn what dynamic programming is. Also, you will find the comparison between dynamic programming and greedy algorithms to solve problems.

Dynamic Programming is a technique in computer programming that helps to efficiently solve a class of problems that have overlapping subproblems and optimal substructure property.

If any problem can be divided into subproblems, which in turn are divided into smaller subproblems, and if there are overlapping among these subproblems, then the solutions to these subproblems can be saved for future reference. In this way, efficiency of the CPU can be enhanced. This method of solving a solution is referred to as dynamic programming.

Such problems involve repeatedly calculating the value of the same subproblems to find the optimum solution.

Dynamic Programming Example

Let's find the fibonacci sequence upto 5th term. A fibonacci series is the sequence of numbers in which each number is the sum of the two preceding ones. For example, `0,1,1, 2, 3`. Here, each number is the sum of the two preceding numbers.

Algorithm

```
Let n be the number of terms.

1. If n <= 1, return 1.
2. Else, return the sum of two preceding numbers.
```

We are calculating the fibonacci sequence up to the 5th term.

1. The first term is 0.
2. The second term is 1.
3. The third term is sum of 0 (from step 1) and 1(from step 2), which is 1.
4. The fourth term is the sum of the third term (from step 3) and second term (from step 2) i.e. `1 + 1 = 2`.

5. The fifth term is the sum of the fourth term (from step 4) and third term (from step 3) i.e. `2 + 1 = 3`.

Hence, we have the sequence `0,1,1, 2, 3`. Here, we have used the results of the previous steps as shown below. This is called a **dynamic programming approach**.

```
F(0) = 0
F(1) = 1
F(2) = F(1) + F(0)
F(3) = F(2) + F(1)
F(4) = F(3) + F(2)
```

How Dynamic Programming Works

Dynamic programming works by storing the result of subproblems so that when their solutions are required, they are at hand and we do not need to recalculate them.

This technique of storing the value of subproblems is called memoization. By saving the values in the array, we save time for computations of sub-problems we have already come across.

```
var m = map(0 → 0, 1 → 1)
function fib(n)
    if key n is not in map m
        m[n] = fib(n − 1) + fib(n − 2)
    return m[n]
```

Dynamic programming by memoization is a top-down approach to dynamic programming. By reversing the direction in which the algorithm works i.e. by starting from the base case and working towards the solution, we can also implement dynamic programming in a bottom-up manner.

```
function fib(n)
    if n = 0
        return 0
    else
        var prevFib = 0, currFib = 1
        repeat n - 1 times
            var newFib = prevFib + currFib
            prevFib = currFib
            currFib  = newFib
    return currentFib
```

Recursion vs Dynamic Programming

Dynamic programming is mostly applied to recursive algorithms. This is not a coincidence, most optimization problems require recursion and dynamic programming is used for optimization.

But not all problems that use recursion can use Dynamic Programming. Unless there is a presence of overlapping subproblems like in the fibonacci sequence problem, a recursion can only reach the solution using a divide and conquer approach.

That is the reason why a recursive algorithm like Merge Sort cannot use Dynamic Programming, because the subproblems are not overlapping in any way.

Greedy Algorithms vs Dynamic Programming

Greedy Algorithms are similar to dynamic programming in the sense that they are both tools for optimization.

However, greedy algorithms look for locally optimum solutions or in other words, a greedy choice, in the hopes of finding a global optimum. Hence greedy algorithms can make a guess that looks optimum at the time but becomes costly down the line and do not guarantee a globally optimum.

Dynamic programming, on the other hand, finds the optimal solution to subproblems and then makes an informed choice to combine the results of those subproblems to find the most optimum solution.

What is the Principle of Optimality?

The dynamic programming algorithm obtains the solution using the principle of optimality. The principle of optimality states that " in an optimal sequence of decisions or choices, each subsequence must also be optimal". When it is not possible to apply the principle of optimality it is almost impossible to obtain the solution using the dynamic programming approach.

The principle of optimality: "If k is a node on the shortest path from i to j, then the part of the path from i to k, and the part from k to i, must also be optimal."

How to Solve Dynamic Programming Problems?

After knowing what dynamic programming is, it is important to learn the recipe for solving dynamic programming problems. When it comes to finding the solution to the problem using dynamic programming, below are the few steps you should consider to follow:

1) Recognize the DP (dynamic programming) problem

Identifying that the given problem statement can be solved by a dynamic programming algorithm is the most crucial step. You can solve this difficulty by asking yourself whether you can divide the given problem statements into smaller parts as a function and find its solution or not.

2) Identify problem variables

After deciding that the problem can be solved using dynamic programming, the next thing you have to do is to find the recursive structure between the subproblems of the original problem. Here, you have to consider the changing parameters of the problem. This changing parameter can be anything like the array position or the speed of the problem-solving. Also, it is important to identify the number of subproblems of the original problem.

3) Express the recurrence relation

Many developers rush through the coding part of problem-solving and forget to define the recurrence relation. Expressing the recurrence relation clearly before coding the problem will strengthen your problem understanding and make the process efficient.

4) Identify the base case

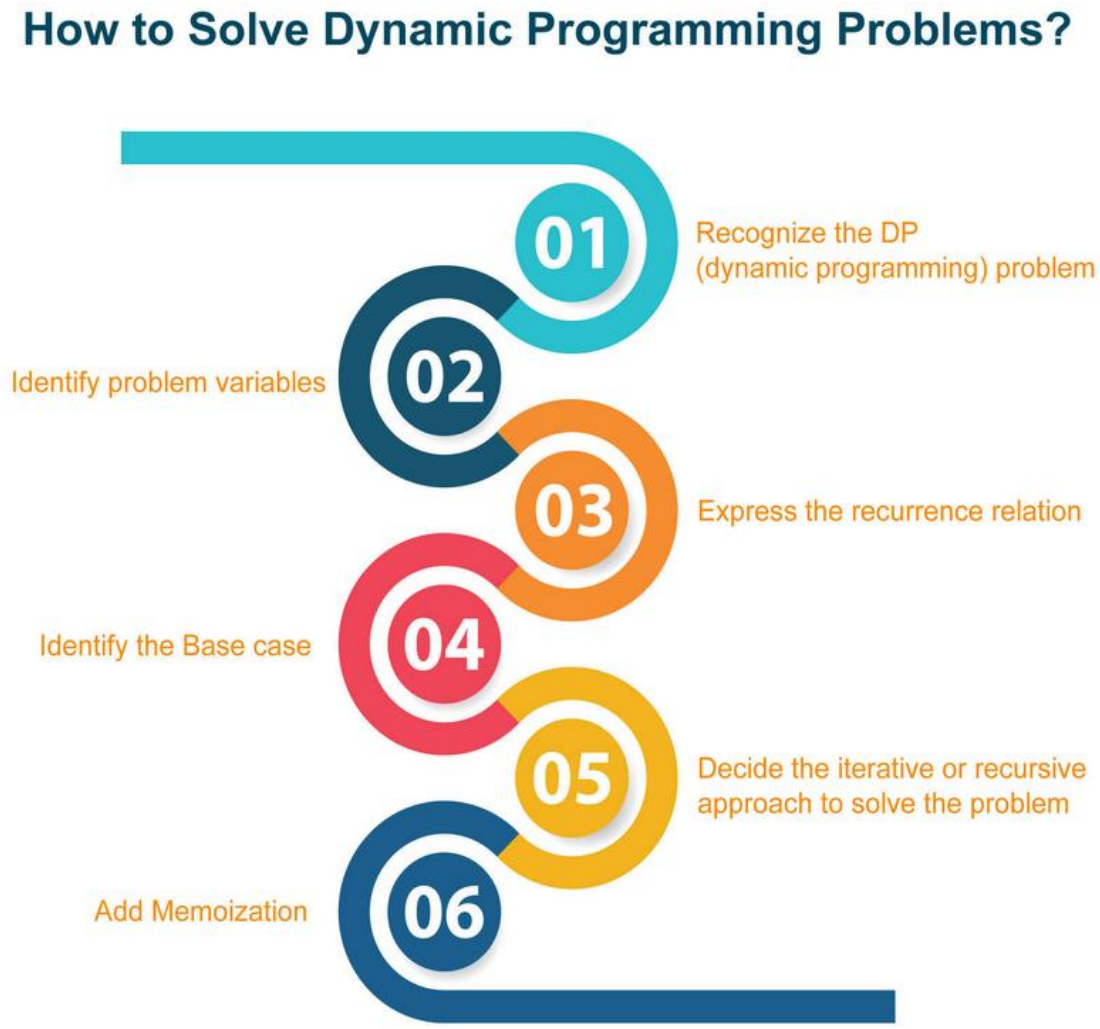
The case is a part of the subproblem which is independent of other subproblems. In order to define the base case of your subproblem, you have to identify the point at which your problem cannot be simplified further.

5) Decide the iterative or recursive approach to solve the problem

Dynamic programming problems can be solved using an iterative or recursive approach. By the discussion till now, you must assume that the recursive process is better. But it is important to know that all the above points we discussed are completely independent of the problem-solving approach. Whether you choose the iterative method or the recursive method, it is important for you to decide the recurrence relation and the base case of the problem.

6) Add memoization

Memoization is the process of storing the result of the subproblem and calling them again when a similar subproblem is to be solved. This will reduce the time complexity of the problem. If we do not use the memorization, similar subproblems are repeatedly solved which can lead to exponential time complexities.



Characteristics and Elements of Dynamic Programming

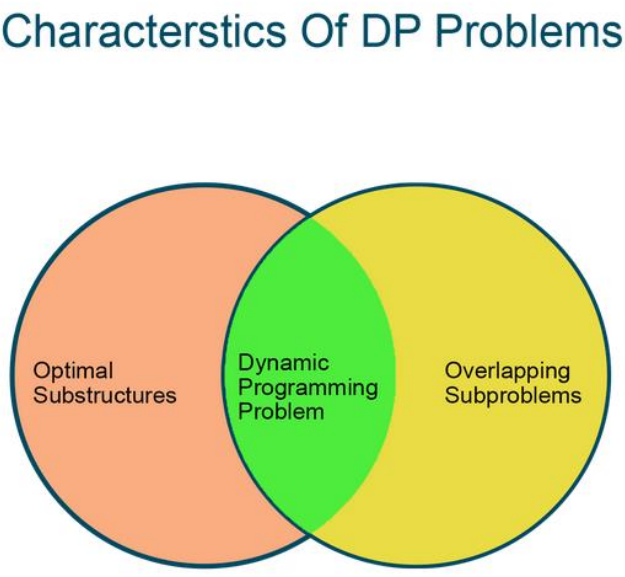
Before applying the dynamic programming approach to a problem statement, it is important to know that when to apply a dynamic programming algorithm. Let us understand 2 characteristics of the dynamic problem which explain the working while solving the problem statement.

1) Optimal substructure

The problem gives the optimal substructure if the optimal solution contains optimal sub solutions in it. We can recursively define the optimal solution if the problem has an optimal substructure. Also, there is no base to define a recursive algorithm if the problem doesn't have an optimal solution.

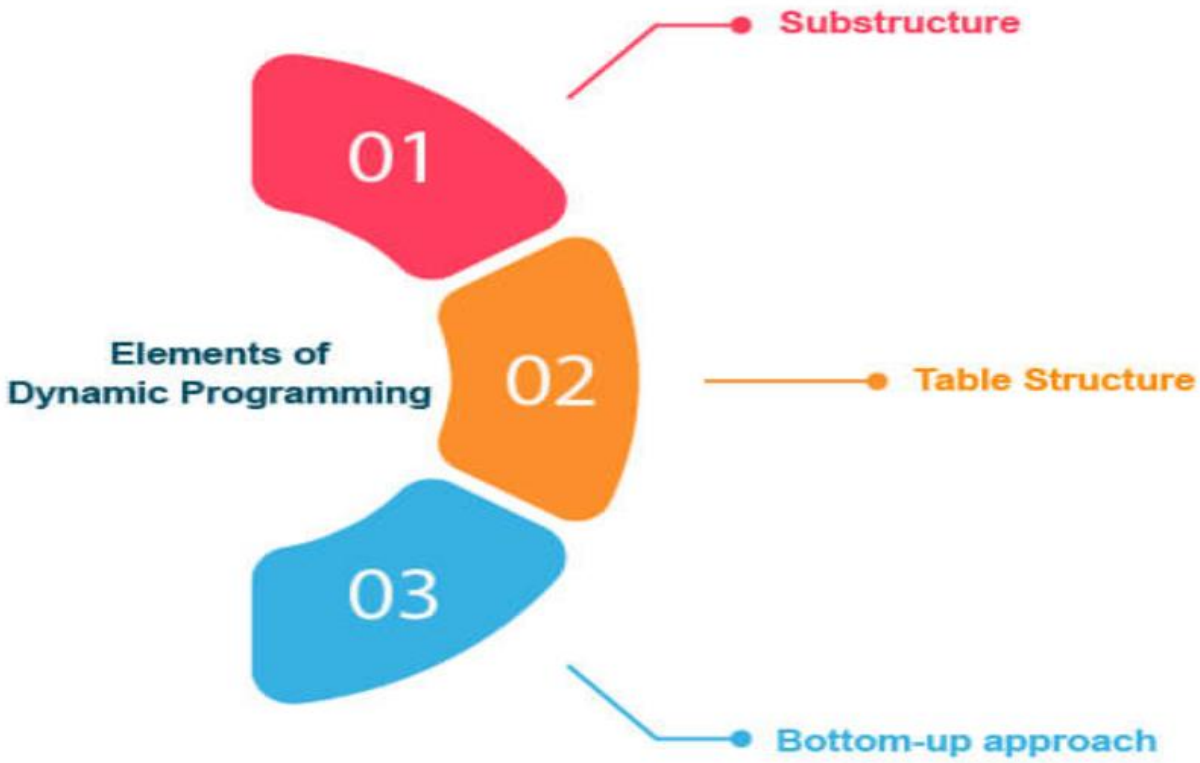
2) Overlapping subproblems

The problem is called overlapping subproblems if the recursive algorithm repeatedly visits the same sub-problems. If any problem has overlapping subproblems then we can improve the repetitive implementation of sub-problems by computing it only once. If the problem doesn't have overlapping subproblems, then it is not feasible to use a dynamic programming algorithm to solve the problem.



As you have understood, dynamic programming is the algorithm that divides the problem into various sub-problems to find out the optimal solution. While solving the problem statement using the dynamic programming approach, the problem is divided into 3 elements in total to get the final result. These elements are:

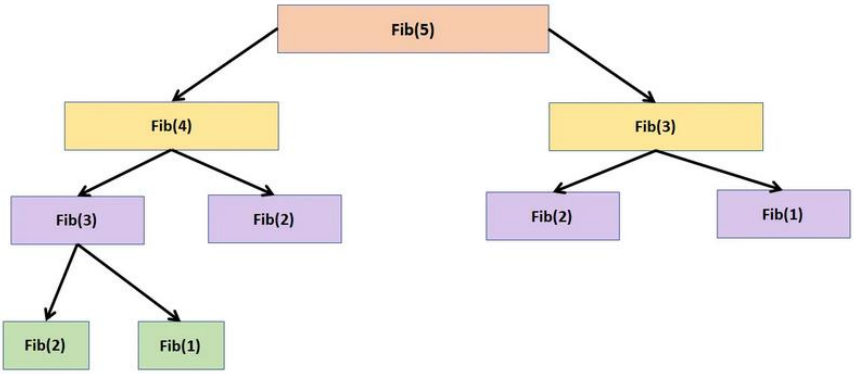
1. **Substructure:** Sub-Structuring is the process of dividing the given problem statement into smaller sub-problems. Here we manage to identify the solution of the original problem in terms of the solution of sub-problems.
2. **Table Structure:** It is necessary to store the solution of the sub-problem into a table after solving it. This is important because as we know, dynamic programming reuse the solutions of subproblems many times so that we don't have to repeatedly solve the same problem, again and again,
3. **Bottom-up approach:** The process of combining the solutions of subproblems to achieve the final result using the table. The process starts with solving the smallest subproblem and later combining their solution with the subproblems of increasing size until you get the final solution of the original problem.



Implementation of Fibonacci Series using Memoization

The Fibonacci Series is a series of numbers in which each number is the sum of the preceding two numbers. By definition, the first two numbers are 0 and 1.

Example: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55,…….



Fibonacci Series Algorithm

Fibonacci Series can be implemented using Memoization using the following steps:

- Declare the function and take the number whose Fibonacci Series is to be printed and a dictionary `memo` as parameters.
- If `n` equals 1, return 0.
- If `n` equals 2, return 1.
- If the current element is memo, add it to the memo by recursivel calling the function for the previous to values and adding them.

```python
 # Function to implement Fibonacci Series
def fibMemo(n, memo):
    if n == 1:
        return 0
    if n == 2:
        return 1
    if not n in memo:
        memo[n] = fibMemo(n-1, memo) + fibMemo(n-2, memo)
    return memo[n]

tempDict = {}
fibMemo(6, tempDict)

#Printing the elements of the Fibonacci Series
print("0")
print("1")
for element in tempDict.values():
    print(element)
```

Output

```
0
1
1
2
3
5
```

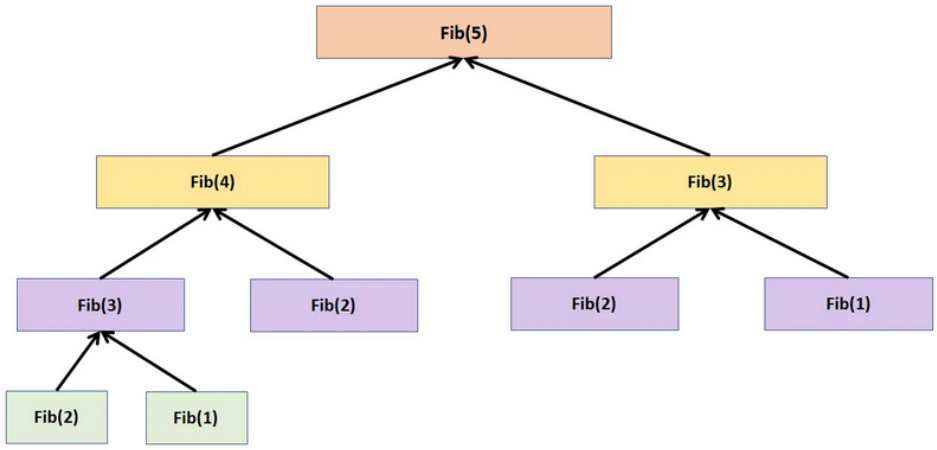
Bottom Up Approach with Tabulation

Tabulation is the opposite of the top-down approach and does not involve recursion. In this approach, we solve the problem "bottom-up". This means that the subproblems are solved first and are then combined to form the solution to the original problem.

This is achieved by filling up a table. Based on the results of the table, the solution to the original problem is computed.

Implementation of Fibonacci Series using Tabulation

Example: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55,…….



Fibonacci Series Algorithm

Fibonacci Series can be implemented using Tabulation using the following steps:

- Declare the function and take the number whose Fibonacci Series is to be printed.
- Initialize the list and input the values 0 and 1 in it.
- Iterate over the range of 2 to `n+1`.
- Append the list with the sum of the previous two values of the list.
- Return the list as output.

```
 # Function to implement Fibonacci Series
def fibTab(n):
    tb = [0, 1]
    for i in range(2, n+1):
      tb.append(tb[i-1] + tb[i-2])
    return tb

print(fibTab(6))

#Output
[0, 1, 1, 2, 3, 5, 8]
```

Top-Down Approach vs Bottom-Up Approach

	Top-Down	Bottom-Up
Easiness	**Easy to solve as it is an extension of Divide and Conquer**	**Difficult to come up with a solution**
Runtime	**Slow**	**Fast**
Space Efficiency	**Unnecessary use of stack space**	**Stack is not used**
When to use	**Need a quick solution**	**Need an efficient solution**

Number Factor

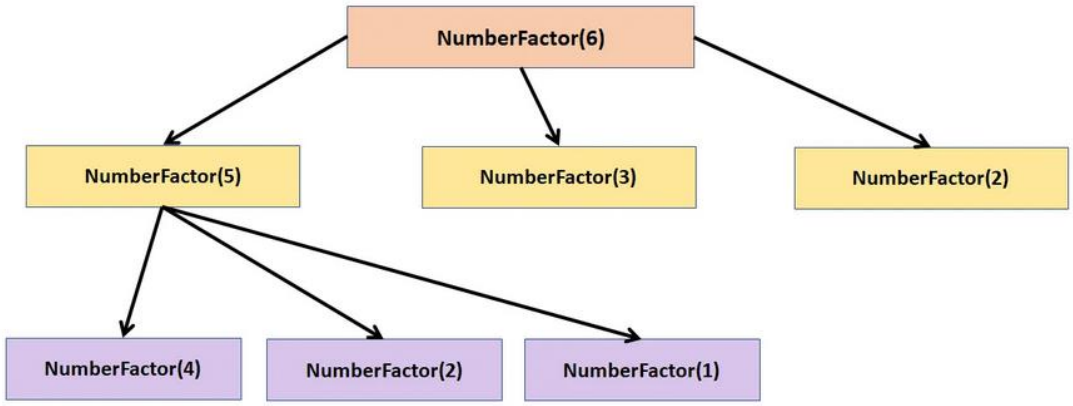
We are given a number N. Find the number of ways of expressing N as a sum 1, 3, and 4.

Example:

– N = 5

– Number of ways = 6

– Explanation: There are 6 ways to express N. {4,1}, {1,4}, {1,3,1}, {3,1,1}, {1,1,3}, {1,1,1,1,1}



Implementation of Number Factor using Top Down Approach

Number Factor problem can be implemented using the Top-Down Approach in the following manner:

- Declare the function and take the number whose Number Factor is to be printed and a temporary dictionary as parameters.
- If the number is 0, 1, or 2, return 1.
- Else, if the number is 3, return 2.
- Else, recursively call the function for three sub-parts `N-1`, `N-3`, `N-4`.
- Assign the sum of the three sub-parts to the dictionary.
- Return the last element of the dictionary as output.

```
 #Function to solve Number Factor Problem
def numberFactor(n, tempDict):
    if n in (0,1,2):
        return 1
    elif n == 3:
        return 2
    else:
        subPart1 = numberFactor(n-1, tempDict)
        subPart2 = numberFactor(n-3, tempDict)
        subPart3 = numberFactor(n-4, tempDict)
        tempDict[n] = subPart1+subPart2+subPart3
        return tempDict[n]

print(numberFactor(5, {}))

#Output
6
```

Implementation of Number Factor using Bottom Up Approach

Number Factor problem can be implemented using the Bottom-Up Approach in the following manner:

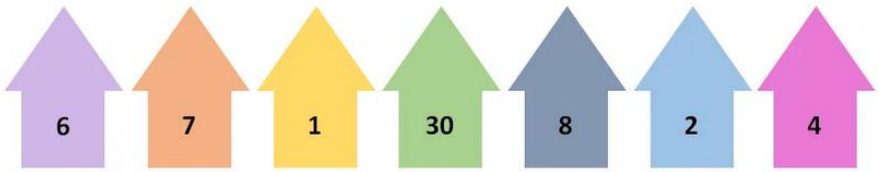
- Declare the function and take the number whose Number Factor is to be printed as parameter.
- Initialize an array with the values 1, 1, 1, 2.
- Iterate over the range from 4 to `n+1`.
- Append the array with the sum of the values at indices – `i-1`, `i-3`, `i-4`.
- Return the last element of the array.

```
 #Function to solve Number Factor Problem
def numberFactor(n):
   tempArr = [1,1,1,2]
   for i in range(4, n+1)
       tempArr.append(tempArr[i-1]+tempArr[i-3]+tempArr[i-4])
       return tempArr[n]

print(numberFactor(5))

#Output
6
```

House Robber

We are given N number of houses along a street, each having some amount of money. A thief can not steal from adjacent houses. Find the maximum amount that can be stolen.

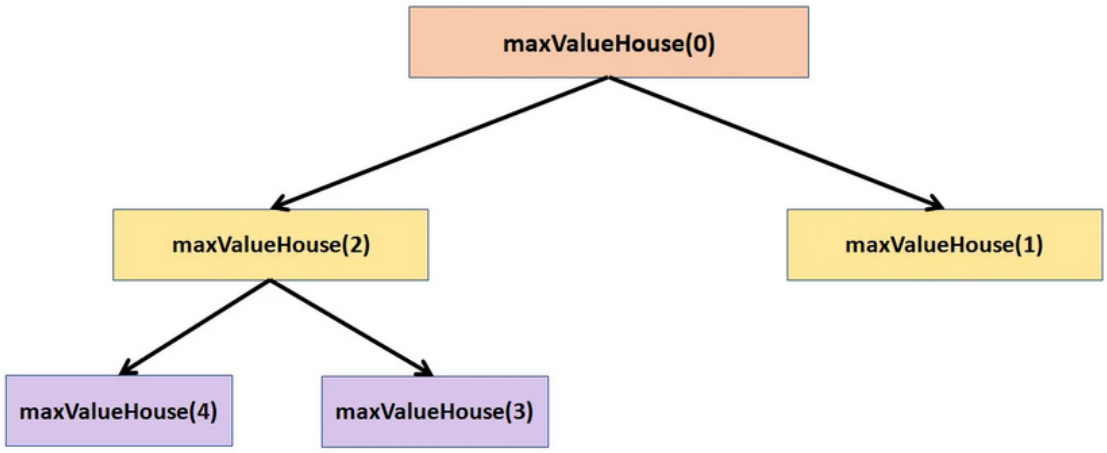
Example:



Answer:

– Maximum amount = 41

– Houses Stolen = 7, 30, 4



Implementation of House Robber using Top Down Approach

House Robber problem can be implemented using the Top-Down Approach in the following manner:

- Declare the function and take the houses, current index, and a temporary dictionary as parameters.
- If the `currentIndex` is greater than the length of the list of houses, return 0.
- Then, we create two lists – `stealFirstHouse` and `skipFirstHouse`.
- We recursively call the function for each variable by incrementing the index.
- Finally, the max value between the two sets is assigned to the current index of the dictionary.
- Return the value at the current index of the dictionary.

```
 #Function to solve House Robber Problem
def houseRobber(houses, currentIndex, tempDict):
    if currentIndex >= len(houses):
        return 0
    else:
        stealFirstHouse = houses[currentIndex] + houseRobber(houses, currentIndex + 2, tempDict)
        skipFirstHouse = houseRobber(houses, currentIndex+1)
        tempDict[currentIndex] = max(stealFirstHouse, skipFirstHouse, tempDict)
        return tempDict[currentIndex]

houses = [6,7,1,30,8,2,4]
print(houseRobber(houses, 0, {}))

#Output
41
```

Implementation of House Robber using Bottom Up Approach

House Robber problem can be implemented using the Bottom-Up Approach in the following manner:

- Declare the function and take the houses and current index as parameters.
- Initialize a temporary array with all the elements set to zero.
- Iterate over the list of houses by decrementing the indices.
- In the current index of the array, store the maximum of the following two values – `houses[i] + tempArr[i+2]`, `tempArr[i+1]`.
- Return the first element of the array as output.

```
 #Function to solve House Robber Problem
def houseRobber(houses, currentIndex):
    tempArr = [0] * (len(houses)+2)
    for i in range(len(houses)-1, -1, -1):
        tempArr[i] = max(houses[i] + tempArr[i+2], tempArr[i+1])
    return tempArr[0]

houses = [6,7,1,30,8,2,4]
print(houseRobber(houses, 0))

#Output
41
```

Convert String

We are given two strings S1 and S2. Convert S2 to S1 using delete, insert or replace operations. Find the minimum number of edit operations

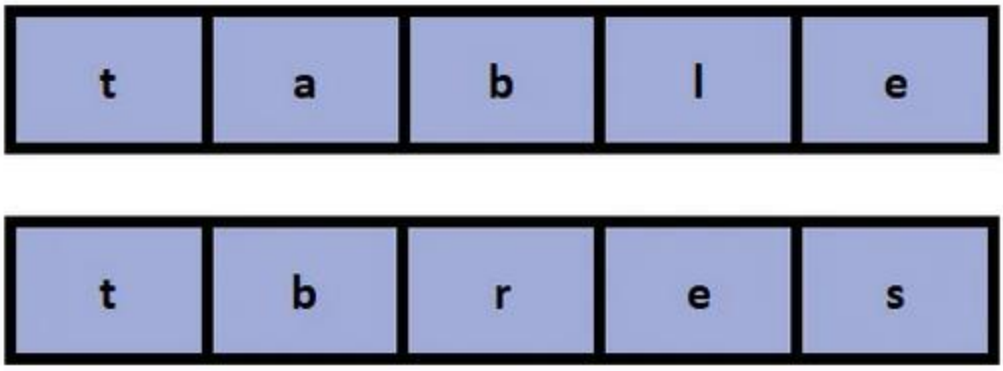
Example:

– S1 = “table”

– S2 = “tbres”

– Output = 3

– Explanation: Insert “a” to second position, replace “r” with “l”, and delete “s”.



Implementation of String Convert using Top Down Approach

String Convert problem can be implemented using the Top-Down Approach using the following steps:

- Declare the function and take the strings `s1` and `s2`, along with their starting indices and a temporary dictionary as parameters.
- If `s1` or `s2` are empty strings, return the differnece between the empty string and the other string as output.
- If they have the same length, recursively invoke the function by incementing the index.
- Declare `dictKey` as the sum of the two indices in string format.
- Else, recursively invoke the function for each seperate operation by incrementing the value of the indices.
- Assign the minimum of the three values to the dictionary at `dictKey`.

```
 #Function to solve String Convert Problem
def findMinOperation(s1, s2, index1, index2, tempDict):
    if index1 == len(s1):
        return len(s2)-index2
    if index2 == len(s2):
        return len(s1)-index1
    if s1[index1] == s2[index2]:
        return findMinOperation(s1, s2, index1+1, index2+1)
    else:
        dictKey = str(index1) + str(index2)
        if dictKey not in tempDict:
            deleteOp = 1 + findMinOperation(s1, s2, index1, index2+1)
            insertOp = 1 + findMinOperation(s1, s2, index1+1, index2)
            replaceOp = 1 + findMinOperation(s1, s2, index1+1, index2+1)
            tempDict[dictKey] = min (deleteOp, insertOp, replaceOp)
        return tempDict[dictKey]

print(findMinOperation("table", "tbres", 0, 0))

#Output
3
```

Implementation of String Convert using Bottom Up Approach

String Convert problem can be implemented using the Bottom-Up Approach using the following steps:

- Declare the function and take the strings `s1` and `s2`, and a temporary dictionary as parameters.
- Iterate over the first string `s1` and create key-value pairs using the indices of the string. Store it in the temporary dictionary.
- Iterate over the second string `s2` and create key-value pairs using the indices of the string. Store it in the temporary dictionary.
- Create a nested loop and iterate over the two given strings.
- Compare the values of the associated elements of the strings. For each comparison and operation count, and create new key-value pairs for each.
- Finally, chose the dictionary key with the minimum value and return it as output.

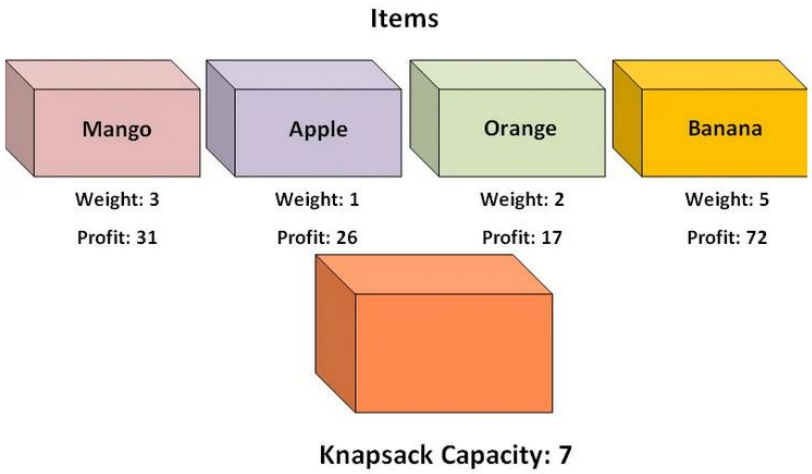
```python
 #Function to solve String Convert Problem
def findMinOperation(s1, s2, tempDict):
    for i1 in range(len(s1)+1):
        dictKey = str(i1)+'0'
        tempDict[dictKey] = i1
    for i2 in range(len(s2)+1):
        dictKey = '0'+str(i2)
        tempDict[dictKey] = i2

    for i1 in range(1, len(s1)+1):
        for i2 in range(1, len(s2)+1):
            if s1[i1-1] == s2[i2-1]:
                dictKey = str(i1)+str(i2)
                dictKey1 = str(i1-1)+str(i2-1)
                tempDict[dictKey] = tempDict[dictKey1]
            else:
                dictKey = str(i1)+str(i2)
                dictKeyD = str(i1-1)+str(i2)
                dictKeyI = str(i1)+str(i2-1)
                dictKeyR = str(i1-1)+str(i2-1)
                tempDict[dictKey] = 1 + min(tempDict[dictKeyD],
                                            min(tempDict[dictKeyI], tempDict[dictKeyR]))
    dictKey = str(len(s1))+str(len(s2))
    return tempDict[dictKey]

print(findMinOperation("table", "tbres", {}))

#Output
3
```

.

Zero One Knapsack Problem

We are given the weights and profits of N items. Find the maximum profit if we are given the knapsack capacity C. Items cannot be broken and must be added fully.

Example:



Answer:

Max profit is achieved by the following combination:

```
Apple(W: 1, P: 26) + Banana(W: 5, P: 72) = W: 6, P: 98
```

Implementation of Zero One Knapsack Problem using Top Down Approach

For the implementation of the Zero Knapsack Problem, we first create the `Item` where each object holds two variables – `weight` and `profit`.

Zero One Knapsack Problem can be implemented using the Top-Down Approach in the following manner:

- Declare the function and take the items, capacity, a temporary dictionary, and current index as parameters.
- If `capacity` is less than 0, `currentIndex` is less than 0, or `currentIndex` is greater than length of the list, return 0.
- Else, if the total weight is less than the capacity, create two profits for checking adajacent items. This is achieved by recursively calling the function with different parameters.
- Assign the maximum out of these two profitsto the current element of the dictionary.
- Return that value of the dictionary as output.

```python
 #Function to solve Zero One Knapsack Problem
class Item:
    def __init__(self, profit, weight):
        self.profit = profit
        self.weight = weight

def zeroKnapsack(items, capacity, currentIndex, tempDict):
    dictKey = str(currentIndex) + str(capacity)
    if capacity <= 0 or currentIndex < 0 or currentIndex >= len(items):
        return 0
    elif dictKey in tempDict:
        return tempDict[currentIndex]
    elif items[currentIndex].weight <= capacity:
        profit1 = items[currentIndex].profit + zeroKnapsack(
            items, capacity-items[currentIndex].weight, currentIndex+1, tempDict)
        profit2 = zeroKnapsack(items, capacity, currentIndex+1, tempDict)
        tempDict[dictKey] = max(profit1, profit2)
        return tempDict[dictKey]
    else:
        return 0

mango = Item(31, 3)
apple = Item(26, 1)
orange = Item(17, 2)
banana = Item(72, 5)

items = [mango, apple, orange, banana]
```

```python
print(zeroKnapsack(items, 7, 0, {}))

#Output
98
```

Implementation of Zero One Knapsack Problem using Bottom Up Approach

For the implementation of the Zero Knapsack Problem, we first create the `Item` where each object holds two variables – `weight` and `profit`.

Zero One Knapsack Problem can be implemented using the Bottom-Up Approach in the following manner:

- Declare the function and take the profts, weights of the items, and the knapsack capacity as parameters.
- If the capacity is less than 0, length of profits equals 0, or the length of the profits and weights are not same, return 0.
- Create variables – `numberOfRows` and `dp`. Iterate over number of rows and capacity, initializing the values of `dp` to 0.
- Create two variables for storing profits. Store the values for each row and column of the matrix `dp` as long as its weight does not exceed capacity.
- Repeat the process and return the max of the two variables as output.

```
 #Function to solve Zero One Knapsack Problem
class Item:
    def __init__(self, profit, weight):
        self.profit = profit
        self.weight = weight

def zoKnapsackBU(profits, weights, capacity):
    if capacity <= 0 or len(profits) == 0 or len(weights) != len(profits):
        return 0
    numberOfRows = len(profits) + 1
    dp = [[None for i in range(capacity+2)] for j in range(numberOfRows)]
    for i in range(numberOfRows):
        dp[i][0] = 0
    for i in range(capacity+1):
        dp[numberOfRows-1][i] = 0
    for row in range(numberOfRows-2, -1, -1):
        for column in range(1, capacity+1):
            profit1 = 0
            profit2 = 0
            if weights[row] <= column:
                profit1 = profits[row] + dp[row + 1][column - weights[row]]
            profit2 = dp[row + 1][column]
            dp[row][column] = max(profit1, profit2)
    return dp[0][capacity]

profits = [31, 26, 17, 72]
weights = [3, 1, 2, 5]
```

```
print(zoKnapsackBU(profits, weights, 7))

#Output
98
```

Heap Sort

What is Heap Sort

> ***Heap sort*** *is a comparison-based sorting technique based on Binary Heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.*

- Heap sort is an in-place algorithm.
- Its typical implementation is not stable, but can be made stable (See this)
- Typically 2-3 times slower than well-implemented QuickSort. The reason for slowness is a lack of locality of reference.

Advantages of heapsort:

- **Efficiency –** The time required to perform Heap sort increases logarithmically while other algorithms may grow exponentially slower as the number of items to sort increases. This sorting algorithm is very efficient.
- **Memory Usage –** Memory usage is minimal because apart from what is necessary to hold the initial list of items to be sorted, it needs no additional memory space to work
- **Simplicity –** It is simpler to understand than other equally efficient sorting algorithms because it does not use advanced computer science concepts such as recursion

Applications of HeapSort:

- Heapsort is mainly used in hybrid algorithms like the IntroSort.
- Sort a nearly sorted (or K sorted) array
- k largest(or smallest) elements in an array

The heap sort algorithm has limited uses because Quicksort and Mergesort are better in practice. Nevertheless, the Heap data structure itself is enormously used. See Applications of Heap Data Structure

What is meant by Heapify?

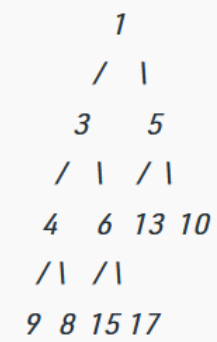
Heapify is the process of creating a heap data structure from a binary tree represented using an array. It is used to create Min-Heap or Max-heap. Start from the first index of the non-leaf node whose index is given by n/2 – 1. Heapify uses recursion

Algorithm for Heapify:

```
heapify(array)
 Root = array[0]

  Largest = largest( array[0] , array [2 * 0 + 1]/ array[2 * 0 + 2])
if(Root != Largest)
 Swap(Root, Largest)
```

How does Heapify work?

Array = {1, 3, 5, 4, 6, 13, 10, 9, 8, 15, 17}
Corresponding Complete Binary Tree is:

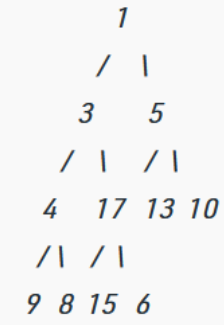


The task to build a Max-Heap from above array.

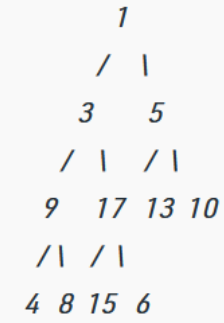
Total Nodes = 11.
Last Non-leaf node index = (11/2) – 1 = 4.
Therefore, last non-leaf node = 6.

To build the heap, heapify only the nodes: [1, 3, 5, 4, 6] in reverse order.

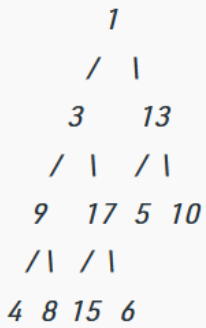
Heapify 6: *Swap 6 and 17.*



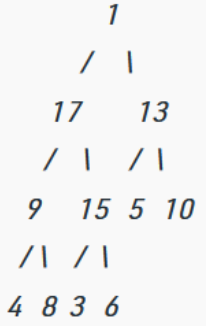
Heapify 4: *Swap 4 and 9.*



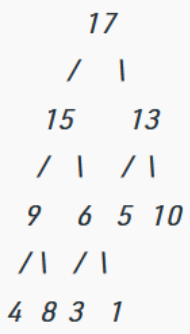
Heapify 5: *Swap 13 and 5.*



Heapify 3: *First Swap 3 and 17, again swap 3 and 15.*



***Heapify 1**: First Swap 1 and 17, again swap 1 and 15, finally swap 1 and 6.*



Heap Sort Algorithm

To solve the problem follow the below idea:

> *First convert the array into heap data structure using heapify, than one by one delete the root node of the Max-heap and replace it with the last node in the heap and then heapify the root of the heap. Repeat this process until size of heap is greater than 1.*

Follow the given steps to solve the problem:

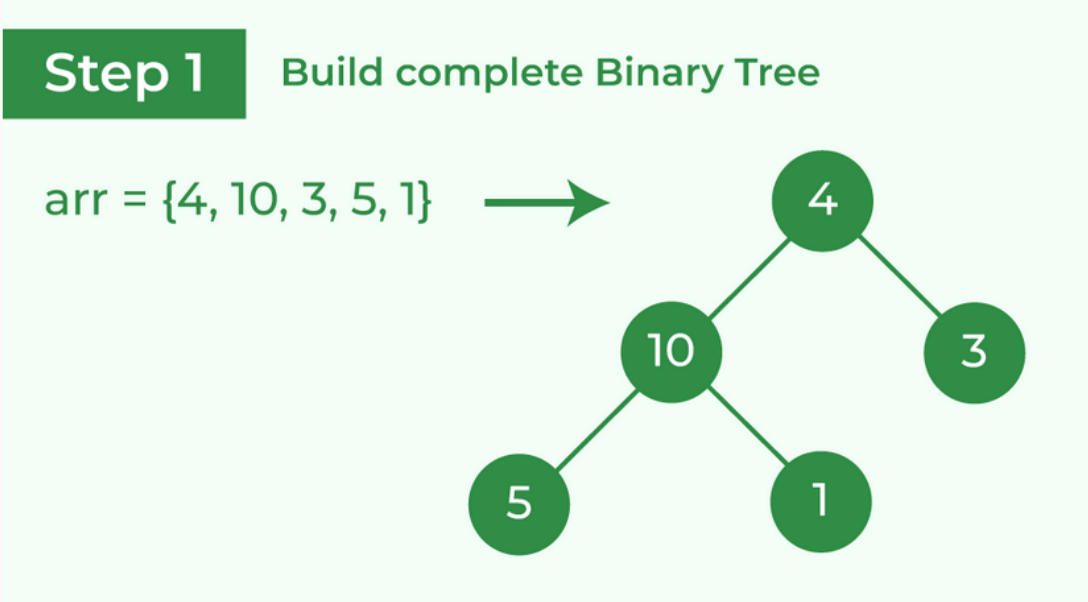
- Build a max heap from the input data.
- At this point, the maximum element is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of the heap by 1. Finally, heapify the root of the tree.
- Repeat step 2 while the size of the heap is greater than 1.

Note: The heapify procedure can only be applied to a node if its children nodes are heapified. So the heapification must be performed in the bottom-up order.

Detailed Working of Heap Sort

To understand heap sort more clearly, let's take an unsorted array and try to sort it using heap sort.
Consider the array: arr[] = {4, 10, 3, 5, 1}.

Build Complete Binary Tree: *Build a complete binary tree from the array.*



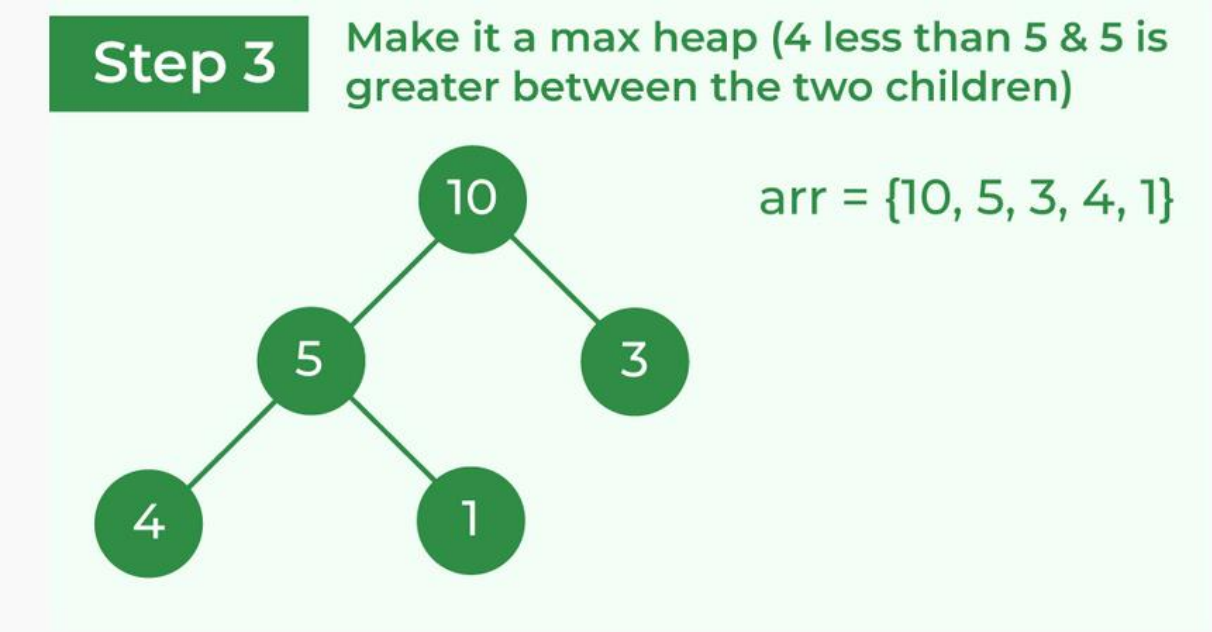
Build complete binary tree from the array

Transform into max heap: *After that, the task is to construct a tree from that unsorted array and try to convert it into max heap.*

- *To transform a heap into a max-heap, the parent node should always be greater than or equal to the child nodes*
  - *Here, in this example, as the parent node **4** is smaller than the child node **10,** thus, swap them to build a max-heap.*

Transform it into a max heap image widget

- *Now, as seen, **4** as a parent is smaller than the child **5**, thus swap both of these again and the resulted heap and array should be like this:*



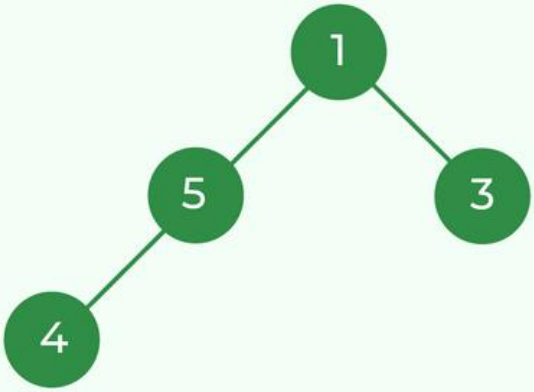
Make the tree a max heap

Perform heap sort: *Remove the maximum element in each step (i.e., move it to the end position and remove that) and then consider the remaining elements and transform it into a max heap.*

- *Delete the root element* ***(10)*** *from the max heap. In order to delete this node, try to swap it with the last node, i.e.* ***(1).*** *After removing the root element, again heapify it to convert it into max heap.*
  - *Resulted heap and array should look like this:*

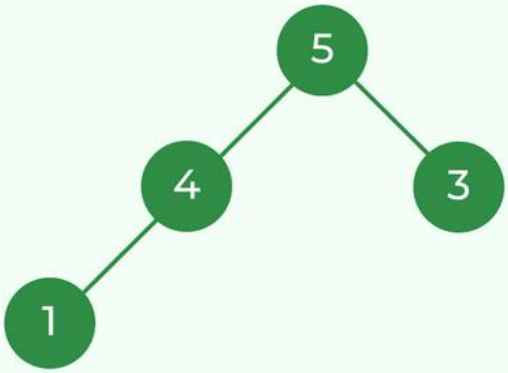
Step 4 Remove the max(10) & heapify

→ Remove the max (i.e., move it to the end)



arr = {1, 5, 3, 4, 10}

→ Heapify



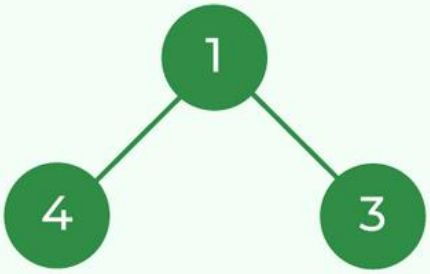
arr = {5, 4, 3, 1, 10}

Remove 10 and perform heapify

- *Repeat the above steps and it will look like the following:*

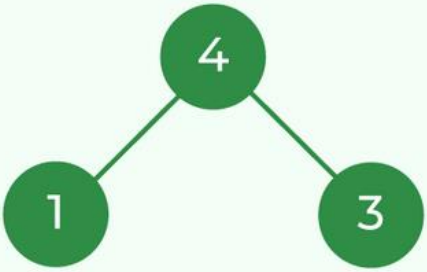
Step 5 Remove the current max(5) & heapify

→ Remove the max (i.e., move it to the end)



arr = {1, 4, 3, 5, 10}

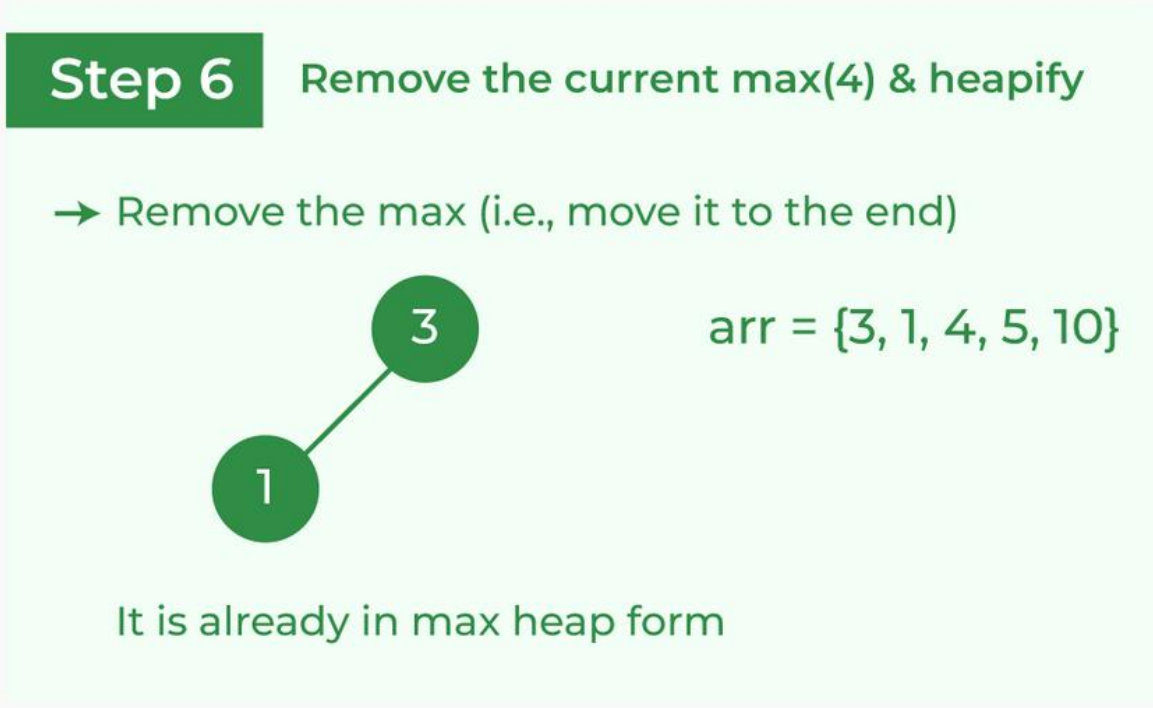
→ Heapify



arr = {4, 1, 3, 5, 10}

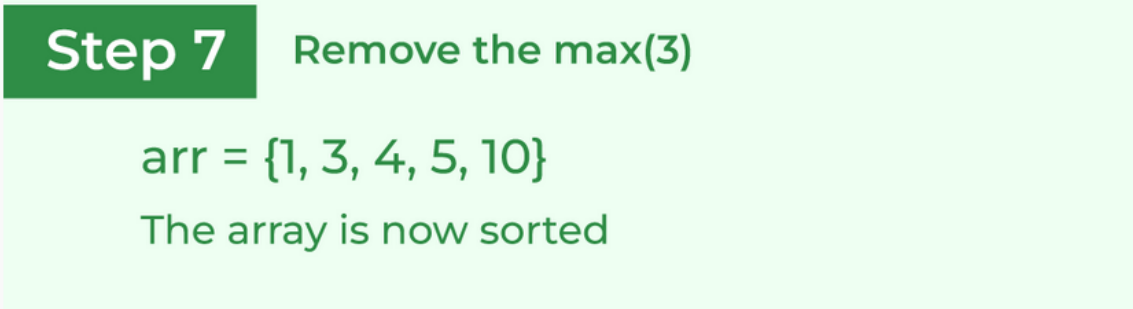
Remove 5 and perform heapify

- *Now remove the root (i.e. 3) again and perform heapify.*



Remove 4 and perform heapify

- *Now when the root is removed once again it is sorted. and the sorted array will be like* ***arr[] = {1, 3, 4, 5, 10}***.



The sorted array

```
# Python program for implementation of heap Sort


# To heapify subtree rooted at index i.

# n is size of heap



def heapify(arr, N, i):

        largest = i # Initialize largest as root

        l = 2 * i + 1         # left = 2*i + 1

        r = 2 * i + 2         # right = 2*i + 2


        # See if left child of root exists and is

        # greater than root

        if l < N and arr[largest] < arr[l]:

                largest = l


        # See if right child of root exists and is

        # greater than root

        if r < N and arr[largest] < arr[r]:

                largest = r


        # Change root, if needed

        if largest != i:

                arr[i], arr[largest] = arr[largest], arr[i] # swap


                # Heapify the root.

                heapify(arr, N, largest)


# The main function to sort an array of given size
```

```
def heapSort(arr):
	N = len(arr)

	# Build a maxheap.
	for i in range(N//2 - 1, -1, -1):
		heapify(arr, N, i)

	# One by one extract elements
	for i in range(N-1, 0, -1):
		arr[i], arr[0] = arr[0], arr[i] # swap
		heapify(arr, i, 0)



# Driver's code
if __name__ == '__main__':
	arr = [12, 11, 13, 5, 6, 7]

	# Function call
	heapSort(arr)
	N = len(arr)

	print("Sorted array is")
	for i in range(N):
		print("%d" % arr[i], end=" ")
```

Output

```
Sorted array is
5 6 7 11 12 13
```

Time complexity of Heap Data Structure

Because we make use of a binary tree, the bottom of the heap contains the maximum number of nodes. As we go up a level, the number of nodes decreases by half. Considering there are 'n' number of nodes, then the number of nodes starting from the bottom-most level would be-

- n/2
- n/4 (at the next level)
- n/8
- and so on

Complexity of inserting a new node

Therefore, when we insert a new value in the heap when making the heap, the max number of steps we would need to take comes out to be O(log(n)). As we use binary trees, we know that the max height of such a structure is always O(log(n)). When we insert a new value in the heap, we will swap it with a value greater than it, to maintain the max-heap property. The number of such swaps would be O(log(n)). Therefore, the insertion of a new value when building a max-heap would be O(log(n)).

Complexity of removing the max valued node from heap

Likewise, when we remove the max valued node from the heap, to add to the end of the list, the max number of steps required would also be O(log(n)). Since we swap the max valued node till it comes down to the bottom-most level, the max number of steps we'd need to take is the same as when inserting a new node, which is O(log(n)).

Therefore, the total time complexity of the `max_heapify` function turns out to be O(log(n)).

Complexity of creating a heap

The time complexity of converting a list into a heap using the `create_heap` function is not O(log(n)). This is because when we create a heap, not all nodes will move down O(log(n)) times. It's only the root node that'll do so. The nodes at the bottom-most level (given by n/2) won't move down at all. The nodes at the second last level (n/4) would move down 1 time, as there is only one level below remaining to move down. The nodes at the third last level would move down 2 times, and so on. So if we multiply the number of moves we take for all nodes, mathematically, it would turn out like a geometric series, as explained below-

(n/2 * 0) + (n/4 * 1) + (n/8 * 2) + (n/16 * 3) + ...h

Here h represents the height of the max-heap structure.

The summation of this series, upon calculation, gives a value of n/2 in the end. Therefore, the time complexity of `create_heap` turns out to be O(n).

Total time complexity

In the final function of `heapsort`, we make use of `create_heap`, which runs once to create a heap and has a runtime of O(n). Then using a for-loop, we call the `max_heapify` for each node, to maintain the max-heap property whenever we remove or insert a node in the heap. Since there are 'n' number of nodes, therefore, the total runtime of the algorithm turns out to be O(n(log(n)), and we use the `max-heapify` function for each node.
Mathematically, we see that-

- The first remove of a node takes log(n) time
- The second remove takes log(n-1) time
- The third remove takes log(n-2) time
- and so on till the last node, which will take log(1) time

So summing up all the terms, we get-

log(n) + log(n-1) + log(n-2) + ...log(1)
as log(x) + log(y) = log(x * y), we get
$=\log(n*(n-1)*(n-2)*\ldots*2*1)$
=log(n!)
Upon further simplification (using Stirling's approximation), log(n!) turns out to be
$=n*\log(n)-n+O(\log(n))$

Taking into account the highest ordered term, the total runtime turns out to be O(n(log(n)).

Worst Case Time Complexity of Heap Sort

The worst case for heap sort might happen when all elements in the list are distinct. Therefore, we would need to call `max-heapify` every time we remove an element. In such a case, considering there are 'n' number of nodes-

- The number of swaps to remove every element would be log(n), as that is the max height of the heap
- Considering we do this for every node, the total number of moves would be n * (log(n)).

Therefore, the runtime in the worst case will be O(n(log(n)).

Best Case Time Complexity of Heap Sort

The best case for heapsort would happen when all elements in the list to be sorted are identical. In such a case, for 'n' number of nodes-

- Removing each node from the heap would take only a constant runtime, O(1). There would be no need to bring any node down or bring max valued node up, as all items are identical.
- Since we do this for every node, the total number of moves would be n * O(1).

Therefore, the runtime in the best case would be O(n).

Average Case Time Complexity of Heap Sort

In terms of total complexity, we already know that we can create a heap in O(n) time and do insertion/removal of nodes in O(log(n)) time. In terms of average time, we need to take into account all possible inputs, distinct elements or otherwise. If the total number of nodes is 'n', in such a case, the `max-heapify` function would need to perform:

- log(n)/2 comparisons in the first iteration (since we only compare two values at a time to build max-heap)
- log(n-1)/2 in the second iteration
- log(n-2)/2 in the third iteration
- and so on

So mathematically, the total sum would turn out to be-

(log(n))/2 + (log(n-1))/2 + (log(n-2))/2 + (log(n-3))/2 + ...
Upon approximation, the final result would be
=1/2(log(n!))
=1/2(n∗log(n)−n+O(log(n)))

Considering the highest ordered term, the average runtime of `max-heapify` would then be O(n(log(n)).

Since we call this function for all nodes in the final `heapsort` function, the runtime would be (n * O(n(log(n))). Calculating the average, upon dividing by n, we'd get a final average runtime of O(n(log(n))

Space Complexity of Heap Sort

Since heapsort is an in-place designed sorting algorithm, the space requirement is constant and therefore, O(1). This is because, in case of any input-

- We arrange all the list items in place using a heap structure
- We put the removed item at the end of the same list after removing the max node from the max-heap.

Therefore, we don't use any extra space when implementing this algorithm. This gives the algorithm a space complexity of O(1).

Conclusion

As a summary, heapsort has:

- Worst case time complexity of O(n(log(n)) [all elements in the list are distinct]
- Best case time complexity of O(n) [all elements are same]
- Average case time complexity of O(n(log(n))
- Space complexity of O(1)

With this article at OpenGenus, you must have the complete idea of Time & Space Complexity of Heap Sort.

Binary Search

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

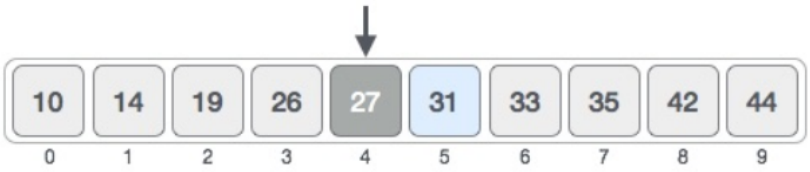
How Binary Search Works?

For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

First, we shall determine half of the array by using this formula −

```
mid = low + (high - low) / 2
```

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

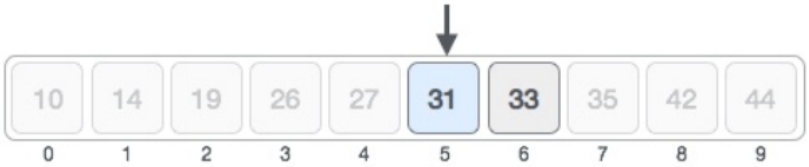
We change our low to mid + 1 and find the new mid value again.

```
low = mid + 1
mid = low + (high - low) / 2
```

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Pseudocode

The pseudocode of binary search algorithms should look like this −

```
Procedure binary_search
   A ← sorted array
   n ← size of array
   x ← value to be searched

   Set lowerBound = 1
   Set upperBound = n

   while x not found
      if upperBound < lowerBound
         EXIT: x does not exists.

      set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

      if A[midPoint] < x
         set lowerBound = midPoint + 1

      if A[midPoint] > x
         set upperBound = midPoint - 1
```

```
      if A[midPoint] = x
         EXIT: x found at location midPoint
   end while

end procedure
```

Best case complexity of Binary Search

The best case complexity of Binary Search occurs when the first comparison is correct (the target value is in the middle of the input array).

This means that regardless of the size of the array, we'll always get the result in constant time. Therefore, the best case time complexity is O(1) - constant time.

Worst case complexity of Binary Search

The worst case complexity of Binary Search occurs when the target value is at the beginning or end of the array.

See the image below: if we have an array 32 elements long and our target is 32, then the array will be divided five times until we find 32. So, the Big O complexity of binary search is O(log(n)) – logarithmic time complexity: log(32) = 5.

Average case complexity of Binary Search

The average case is also of O(log(n)).

Space complexity of Binary Search

Binary Search requires three pointers to elements (start, middle and end), regardless of the size of the array. Therefore the space complexity of Binary Search is O(1) – constant space.

Performance summary table

Time complexity (best)	O(1)
Time complexity (average)	O(log(n))
Time complexity (worst)	O(log(n))
Space complexity	O(1)

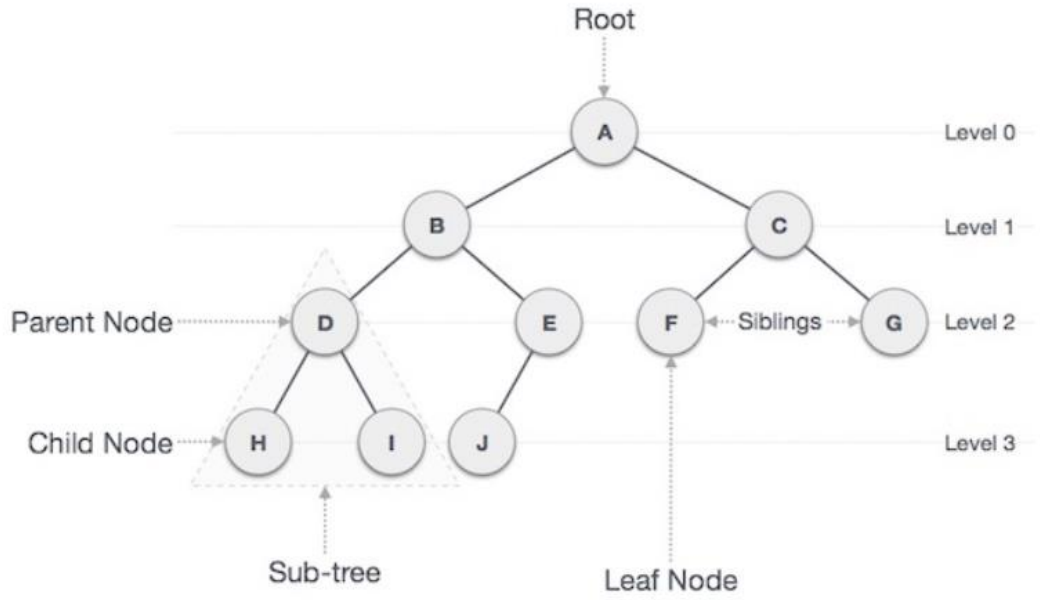
```
import math
import timeit
start= timeit.default_timer()
def binarysearch(array,value):
    start = 0
    end = len(array) - 1
    middle  = math.floor((start+end)/2)

    while not(array[middle] == value):
        if value < array[middle]:
            end = middle - 1
        else:
            start = middle + 1
        middle = math.floor((start+end)/2)
        #print(start, middle, end)
    if array[middle] == value:
        return middle
    else:
        return  -1

c = [8,9,12,15,17,19,20,21,28]
print(c)
print(binarysearch(c,20))
#print(binarysearch(c,30))
stop = timeit.default_timer()
execution_time = stop - start
print(f'Program Time: {execution_time} seconds')
```

```
[8, 9, 12, 15, 17, 19, 20, 21, 28]
6
Program Time: 0.0008139960000335122 seconds
```

Tree

Tree represents the nodes connected by edges. We will discuss binary tree or binary search tree specifically.

Binary Tree is a special datastructure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.



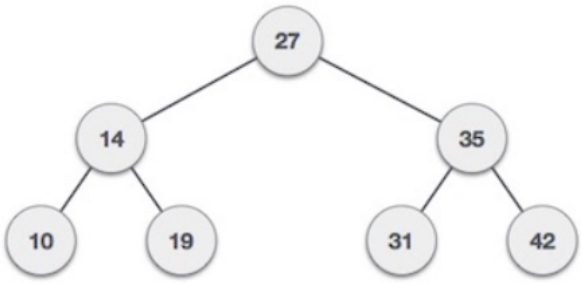
Important Terms

Following are the important terms with respect to tree.

- **Path** − Path refers to the sequence of nodes along the edges of a tree.
- **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** − Any node except the root node has one edge upward to a node called parent.
- **Child** − The node below a given node connected by its edge downward is called its child node.
- **Leaf** − The node which does not have any child node is called the leaf node.
- **Subtree** − Subtree represents the descendants of a node.
- **Visiting** − Visiting refers to checking the value of a node when control is on the node.
- **Traversing** − Traversing means passing through nodes in a specific order.
- **Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.

Binary Search Tree Representation

Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.



We're going to implement tree using node object and connecting them through references.

Tree Node

The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
```

In a tree, all nodes share common construct.

BST Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following −

- **Insert** − Inserts an element in a tree/create a tree.
- **Search** − Searches an element in a tree.
- **Preorder Traversal** − Traverses a tree in a pre-order manner.
- **Inorder Traversal** − Traverses a tree in an in-order manner.
- **Postorder Traversal** − Traverses a tree in a post-order manner.

We shall learn creating (inserting into) a tree structure and searching a data item in a tree in this chapter. We shall learn about tree traversing methods in the coming chapter.

Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm

```
If root is NULL
   then create root node
return

If root exists then
   compare the data with node.data

   while until insertion position is located

      If data is greater than node.data
         goto right subtree
      else
         goto left subtree

   endwhile

   insert data

end If
```

Implementation

The implementation of insert function should look like this −

```
void insert(int data) {
   struct node *tempNode = (struct node*) malloc(sizeof(struct node));
   struct node *current;
   struct node *parent;

   tempNode->data = data;
   tempNode->leftChild = NULL;
   tempNode->rightChild = NULL;

   //if tree is empty, create root node
   if(root == NULL) {
      root = tempNode;
   } else {
      current = root;
      parent  = NULL;
```

```
      while(1) {
         parent = current;

         //go to left of the tree
         if(data < parent->data) {
            current = current->leftChild;

            //insert to the left
            if(current == NULL) {
               parent->leftChild = tempNode;
               return;
            }
         }

         //go to right of the tree
         else {
            current = current->rightChild;

            //insert to the right
            if(current == NULL) {
               parent->rightChild = tempNode;
               return;
```

```
            }
         }
      }
   }
}
```

Search Operation

Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm

```
If root.data is equal to search.data
   return root
else
   while data not found

      If data is greater than node.data
         goto right subtree
      else
         goto left subtree

      If data found
         return node
   endwhile

   return data not found

end if
```

The implementation of this algorithm should look like this.

```
struct node* search(int data) {
   struct node *current = root;
   printf("Visiting elements: ");

   while(current->data != data) {
      if(current != NULL)
      printf("%d ",current->data);

      //go to left tree

      if(current->data > data) {
         current = current->leftChild;
      }
      //else go to right tree
      else {
         current = current->rightChild;
      }

      //not found
      if(current == NULL) {
         return NULL;
      }
   }
```

```
    return current;
}
```

Implementation:

C++ Java Python C# Javascript

```
# A utility function to search a given key in BST
def search(root,key):

    # Base Cases: root is null or key is present at root
    if root is None or root.val == key:
        return root

    # Key is greater than root's key
    if root.val < key:
        return search(root.right,key)

    # Key is smaller than root's key
    return search(root.left,key)

# This code is contributed by Bhavya Jain
```

 # Python program to demonstrate

 # insert operation in binary search tree

 # A utility class that represents

 # an individual node in a BST

class Node:

def __init__(self, key):

self.left = None

self.right = None

self.val = key

 # A utility function to insert

```
# a new node with the given key


def insert(root, key):
        if root is None:
                return Node(key)
        else:
                if root.val == key:
                        return root
                elif root.val < key:
                        root.right = insert(root.right, key)
                else:
                        root.left = insert(root.left, key)
        return root

# A utility function to do inorder tree traversal


def inorder(root):
        if root:
                inorder(root.left)
                print(root.val)
                inorder(root.right)


# Driver program to test the above functions
# Let us create the following BST
# 50
```

**# / **

 # 30 70

**# / \ / **

 # 20 40 60 80

r = Node(50)

r = insert(r, 30)

r = insert(r, 20)

r = insert(r, 40)

r = insert(r, 70)

r = insert(r, 60)

r = insert(r, 80)

 # Print inoder traversal of the BST

inorder(r)

Search for a value in a B-tree

Searching for a value in a tree involves comparing the incoming value with the value exiting nodes. Here also we traverse the nodes from left to right and then finally with the parent. If the searched for value does not match any of the exiting value, then we return not found message, or else the found message is returned.

```
class Node:
   def __init__(self, data):
      self.left = None
      self.right = None
      self.data = data
# Insert method to create nodes
   def insert(self, data):
      if self.data:
         if data < self.data:
            if self.left is None:
               self.left = Node(data)
            else:
               self.left.insert(data)
            else data > self.data:
               if self.right is None:
                  self.right = Node(data)
```

```
               else:
                  self.right.insert(data)
         else:
            self.data = data
# findval method to compare the value with nodes
   def findval(self, lkpval):
      if lkpval < self.data:
         if self.left is None:
            return str(lkpval)+" Not Found"
         return self.left.findval(lkpval)
       else if lkpval > self.data:
            if self.right is None:
               return str(lkpval)+" Not Found"
            return self.right.findval(lkpval)
        else:
            print(str(self.data) + ' is found')
# Print the tree
   def PrintTree(self):
      if self.left:
         self.left.PrintTree()
      print( self.data),
      if self.right:
         self.right.PrintTree()
root = Node(12)
root.insert(6)
root.insert(14)
root.insert(3)
print(root.findval(7))
print(root.findval(14))
```

Output

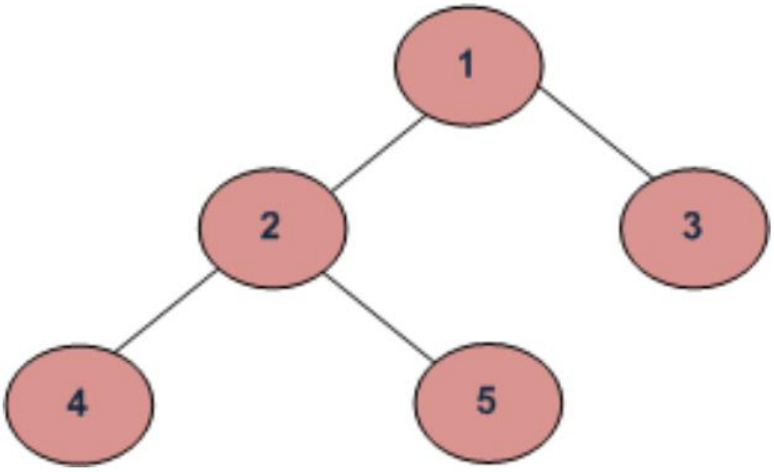
When the above code is executed, it produces the following result −

```
7 Not Found
14 is found
```

Tree Traversals (Inorder, Preorder and Postorder)

Difficulty Level : Easy • Last Updated : 16 Jul, 2022

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



Depth First Traversals:
(a) Inorder (Left, Root, Right) : 4 2 5 1 3
(b) Preorder (Root, Left, Right) : 1 2 4 5 3
(c) Postorder (Left, Right, Root) : 4 5 2 3 1
Breadth-First or Level Order Traversal: 1 2 3 4 5
Please see this post for Breadth-First Traversal.

Inorder Traversal (Practice):

```
Algorithm Inorder(tree)
   1. Traverse the left subtree, i.e., call Inorder(left-subtree)
   2. Visit the root.
   3. Traverse the right subtree, i.e., call Inorder(right-subtree)
```

Uses of Inorder
In the case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal s reversed can be used.
Example: In order traversal for the above-given figure is 4 2 5 1 3.

Preorder Traversal (Practice):

```
Algorithm Preorder(tree)
   1. Visit the root.
   2. Traverse the left subtree, i.e., call Preorder(left-subtree)
   3. Traverse the right subtree, i.e., call Preorder(right-subtree)
```

Uses of Preorder

Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on an expression tree. Please see http://en.wikipedia.org/wiki/Polish_notation to know why prefix expressions are useful.

Example: Preorder traversal for the above-given figure is 1 2 4 5 3.

Postorder Traversal (Practice):

```
Algorithm Postorder(tree)
   1. Traverse the left subtree, i.e., call Postorder(left-subtree)
   2. Traverse the right subtree, i.e., call Postorder(right-subtree)
   3. Visit the root.
```

Uses of Postorder

Postorder traversal is used to delete the tree. Please see the question for the deletion of a tree for details. Postorder traversal is also useful to get the postfix expression of an expression tree. Please see http://en.wikipedia.org/wiki/Reverse_Polish_notation for the usage of postfix expression.

Example: Postorder traversal for the above-given figure is 4 5 2 3 1.

 # Python program to for tree traversals

 # A class that represents an individual node in a

 # Binary Tree

class Node:

    def __init__(self, key):

```
        self.left = None
        self.right = None
        self.val = key


# A function to do inorder tree traversal
def printInorder(root):

    if root:

        # First recur on left child
        printInorder(root.left)

        # then print the data of node
        print(root.val),

        # now recur on right child
        printInorder(root.right)


# A function to do postorder tree traversal
def printPostorder(root):

    if root:

        # First recur on left child
        printPostorder(root.left)
```

```
        # the recur on right child
        printPostorder(root.right)

        # now print the data of node
        print(root.val),


# A function to do preorder tree traversal
def printPreorder(root):

    if root:

        # First print the data of node
        print(root.val),

        # Then recur on left child
        printPreorder(root.left)

        # Finally recur on right child
        printPreorder(root.right)


# Driver code
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
```

print "Preorder traversal of binary tree is"

printPreorder(root)

print "\nInorder traversal of binary tree is"

printInorder(root)

print "\nPostorder traversal of binary tree is"

printPostorder(root)

Output:

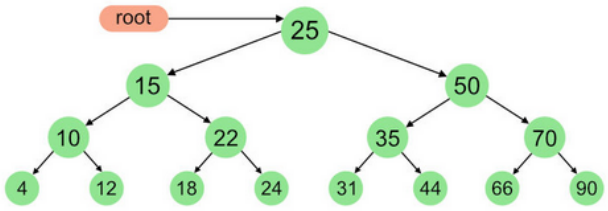
```
Preorder traversal of binary tree is
1 2 4 5 3
Inorder traversal of binary tree is
4 2 5 1 3
Postorder traversal of binary tree is
4 5 2 3 1
```

One more example:

InOrder(root) visits nodes in the following order:
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:
25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



Time Complexity: O(n)

Let us see different corner cases.

Complexity function T(n) – for all problems where tree traversal is involved – can be defined as:

T(n) = T(k) + T(n – k – 1) + c

Where k is the number of nodes on one side of the root and n-k-1 on the other side.

Let's do an analysis of boundary conditions

Case 1: Skewed tree (One of the subtrees is empty and another subtree is non-empty)

k is 0 in this case.

T(n) = T(0) + T(n-1) + c

T(n) = 2T(0) + T(n-2) + 2c

T(n) = 3T(0) + T(n-3) + 3c

T(n) = 4T(0) + T(n-4) + 4c

.....

.....

T(n) = (n-1)T(0) + T(1) + (n-1)c

T(n) = nT(0) + (n)c

Value of T(0) will be some constant say d. (traversing an empty tree will take some constants time)

T(n) = n(c+d)

T(n) = Θ(n) (Theta of n)

Case 2: Both left and right subtrees have an equal number of nodes.

T(n) = 2T(|_n/2_|) + c

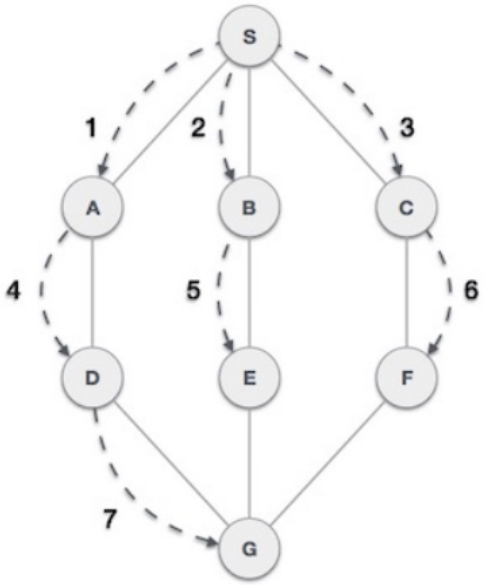
This recursive function is in the standard form (T(n) = aT(n/b) + (-)(n)) for master method http://en.wikipedia.org/wiki/Master_theorem. If we solve it by master method we get (-)(n)

Auxiliary Space: If we don't consider the size of the stack for function calls then O(1) otherwise O(h) where h is the height of the tree.

The height of the skewed tree is n (no. of elements) so the worst space complexity is O(n) and height is (Log n) for the balanced tree so the best space complexity is O(Log n).

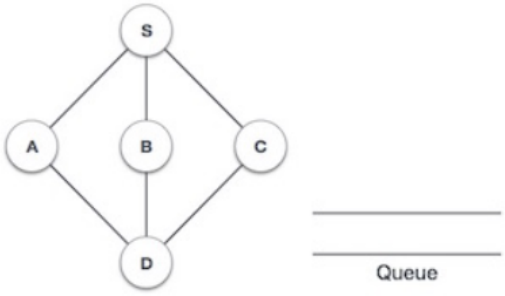
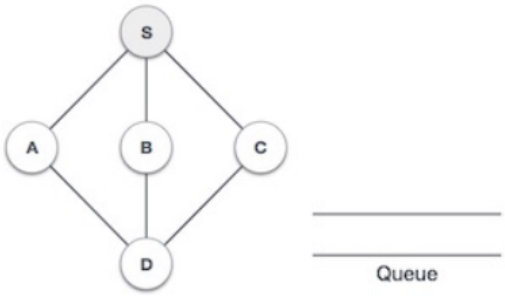
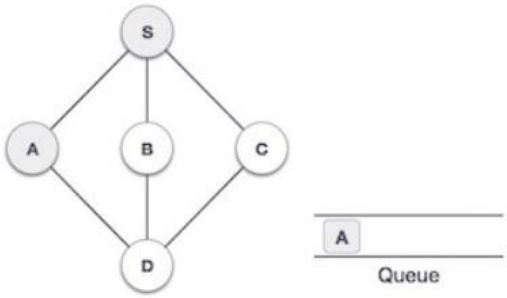
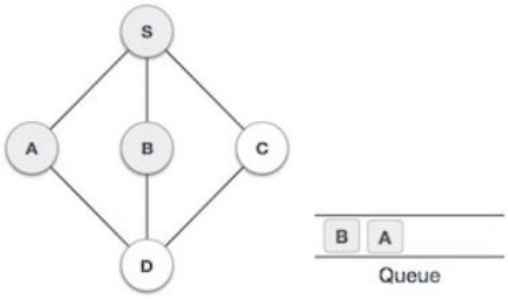
Breadth First Search

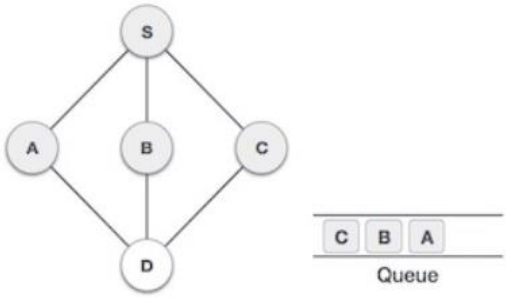
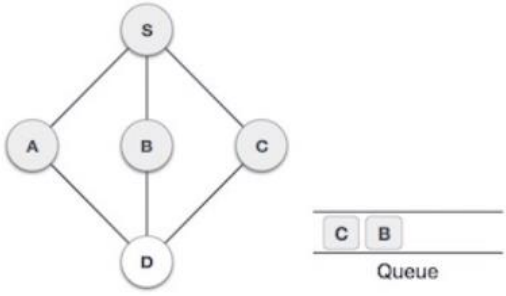
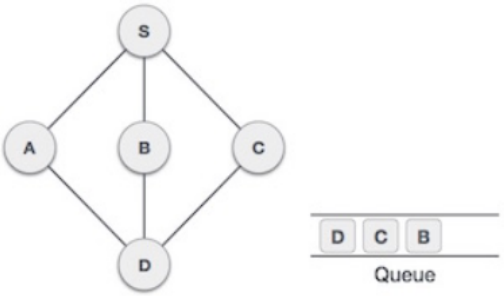
Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



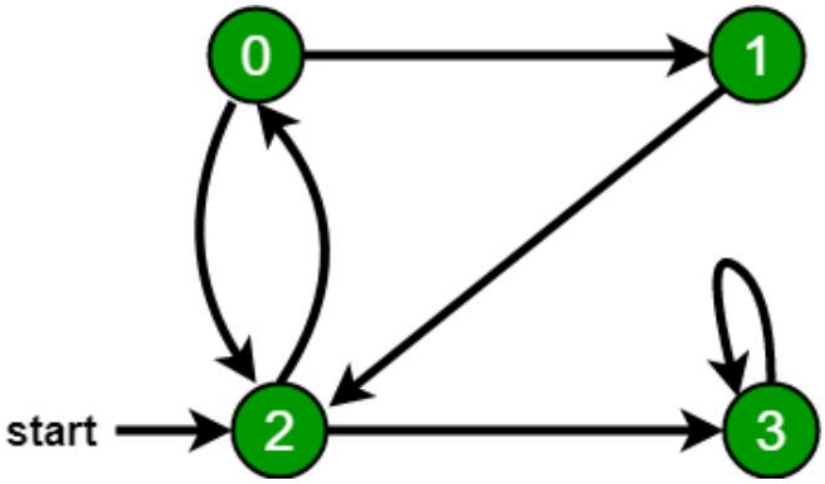
As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** − If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** − Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1		Initialize the queue.
2		We start from visiting **S** (starting node), and mark it as visited.
3		We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.
4		Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.

5		Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.
6		Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.
7		From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.



There can be multiple BFS traversals for a graph. Different BFS traversals for the above graph :

2, 3, 0, 1

2, 0, 3, 1

```
# Python3 Program to print BFS traversal
# from a given source vertex. BFS(int s)
# traverses vertices reachable from s.
from collections import defaultdict

# This class represents a directed graph
# using adjacency list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
```

```
        self.graph[u].append(v)

# Function to print a BFS of graph
def BFS(self, s):

        # Mark all the vertices as not visited
        visited = [False] * (max(self.graph) + 1)

        # Create a queue for BFS
        queue = []

        # Mark the source node as
        # visited and enqueue it
        queue.append(s)
        visited[s] = True

        while queue:

                # Dequeue a vertex from
                # queue and print it
                s = queue.pop(0)
                print (s, end = " ")

                # Get all adjacent vertices of the
                # dequeued vertex s. If a adjacent
                # has not been visited, then mark it
                # visited and enqueue it
                for i in self.graph[s]:
```

```
                    if visited[i] == False:
                        queue.append(i)
                        visited[i] = True

# Driver code

# Create a graph given in
# the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print ("Following is Breadth First Traversal"
                    " (starting from vertex 2)")
g.BFS(2)
```

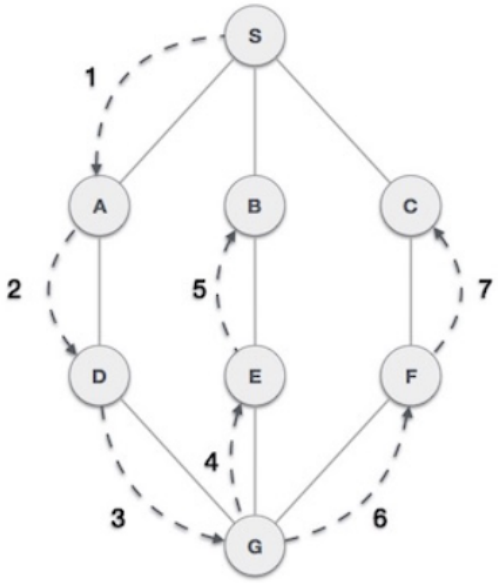
Output

```
Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1
```

Time Complexity: O(V+E), where V is the number of nodes and E is the number of edges.
Auxiliary Space: O(V)

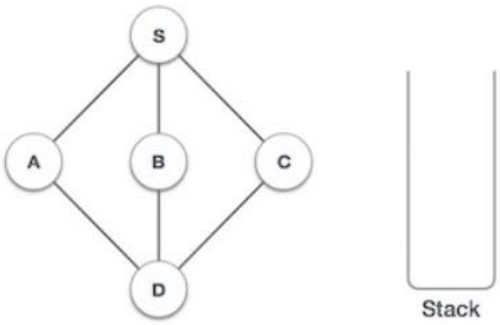
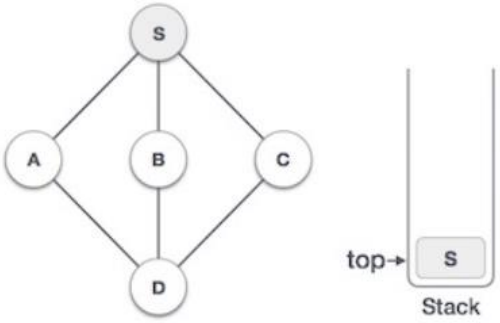
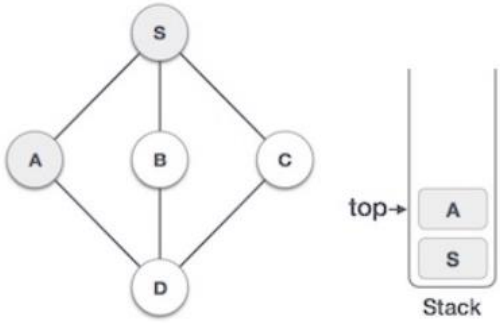
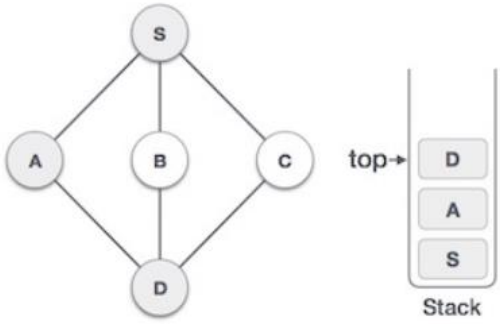
Depth First Search

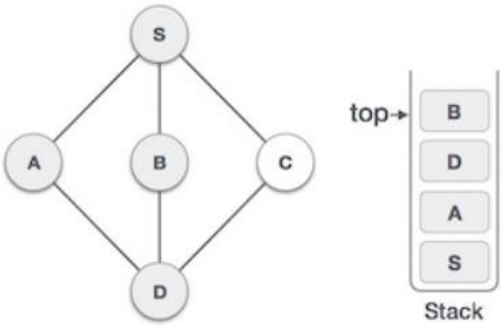
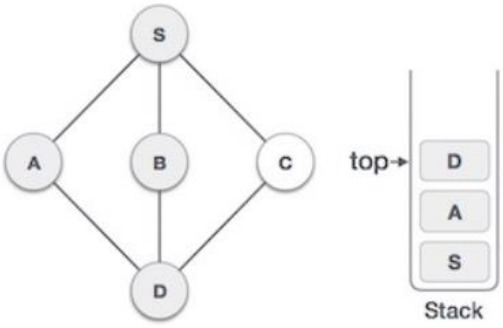
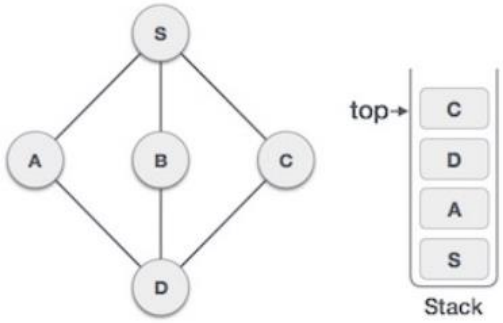
Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

- **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** − Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1		Initialize the stack.
2		Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3		Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.
4		Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.

5		We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.
6		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.
7		Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

 # Python3 program to print DFS traversal

 # from a given graph

from collections import defaultdict

 # This class represents a directed graph using

 # adjacency list representation

class Graph:

```
# Constructor
def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

# function to add an edge to graph
def addEdge(self, u, v):
        self.graph[u].append(v)

# A function used by DFS
def DFSUtil(self, v, visited):

        # Mark the current node as visited
        # and print it
        visited.add(v)
        print(v, end=' ')

        # Recur for all the vertices
        # adjacent to this vertex
        for neighbour in self.graph[v]:
                if neighbour not in visited:
                        self.DFSUtil(neighbour, visited)

# The function to do DFS traversal. It uses
# recursive DFSUtil()
def DFS(self, v):
```

```
        # Create a set to store visited vertices
        visited = set()

        # Call the recursive helper function
        # to print DFS traversal
        self.DFSUtil(v, visited)

# Driver code


# Create a graph given
# in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print("Following is DFS from (starting from vertex 2)")
g.DFS(2)
```

Output:

```
Following is Depth First Traversal (starting from vertex 2)
2 0 1 3
```

Complexity Analysis:

- **Time complexity:** O(V + E), where V is the number of vertices and E is the number of edges in the graph.
- **Space Complexity:** O(V), since an extra visited array of size V is required.

Handling A Disconnected Graph:

- **Solution:**
  This will happen by handling a corner case.
  The above code traverses only the vertices reachable from a given source vertex. All the vertices may not be reachable from a given vertex, as in a Disconnected graph. To do a complete DFS traversal of such graphs, run DFS from all unvisited nodes after a DFS.
  The recursive function remains the same.
- **Algorithm:**
  1. Create a recursive function that takes the index of the node and a visited array.
  2. Mark the current node as visited and print the node.
  3. Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.
  4. Run a loop from 0 to the number of vertices and check if the node is unvisited in the previous DFS, call the recursive function with the current node.

Implementation

'''Python program to print DFS traversal for complete graph'''

from collections import defaultdict

 # this class represents a directed graph using adjacency list representation

class Graph:

    # Constructor

    def __init__(self):

```
        # default dictionary to store graph
        self.graph = defaultdict(list)

    # Function to add an edge to graph
    def addEdge(self, u, v):
        self.graph[u].append(v)
    # A function used by DFS

    def DFSUtil(self, v, visited):
        # Mark the current node as visited and print it
        visited.add(v)
        print(v,end=" ")

        # recur for all the vertices adjacent to this vertex
        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil(neighbour, visited)
        # The function to do DFS traversal. It uses recursive DFSUtil

    def DFS(self):
        # create a set to store all visited vertices
        visited = set()
        # call the recursive helper function to print DFS traversal starting from all
        # vertices one by one
        for vertex in self.graph:
            if vertex not in visited:
                self.DFSUtil(vertex, visited)
# Driver code
```

 # create a graph given in the above diagram

print("Following is Depth First Traversal \n")

g = Graph()

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 2)

g.addEdge(2, 0)

g.addEdge(2, 3)

g.addEdge(3, 3)

g.DFS()

 # Improved by Dheeraj Kumar

Output:

```
Following is Depth First Traversal
0 1 2 3 9
```

Complexity Analysis:

- **Time complexity:** O(V + E), where V is the number of vertices and E is the number of edges in the graph.
- **Space Complexity:** O(V), since an extra visited array of size V is required.

Stack

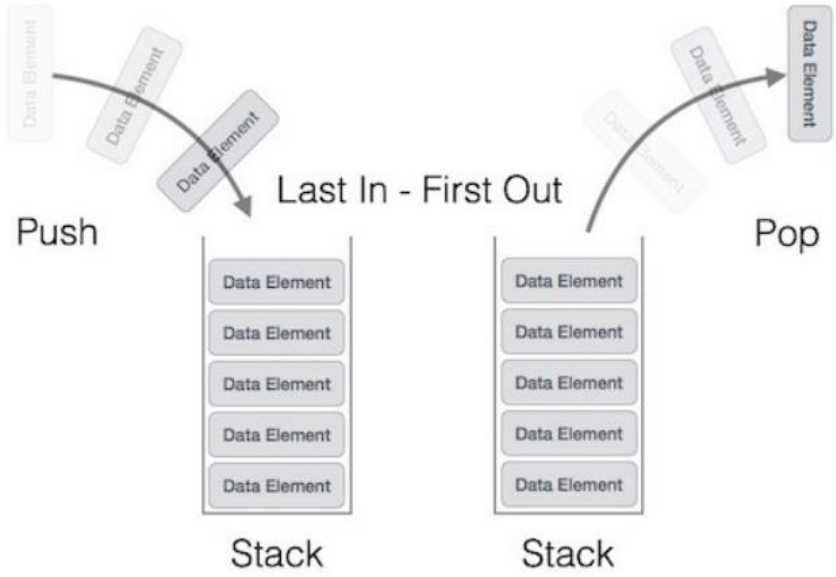
A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- **peek()** − get the top data element of the stack, without removing it.
- **isFull()** − check if stack is full.
- **isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions −

peek()

Algorithm of peek() function −

```
begin procedure peek
return stack[top]
end procedure
```

Implementation of peek() function in C programming language −

Example

```
int peek() {
   return stack[top];
}
```

isfull()

Algorithm of isfull() function −

```
begin procedure isfull

   if top equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

Implementation of isfull() function in C programming language −

Example

```
bool isfull() {
   if(top == MAXSIZE)
      return true;
   else
      return false;
}
```

isempty()

Algorithm of isempty() function −

```
begin procedure isempty

   if top less than 1
      return true
   else
      return false
   endif

end procedure
```

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code −

Example

```
bool isempty() {
   if(top == -1)
      return true;
   else
      return false;
}
```

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

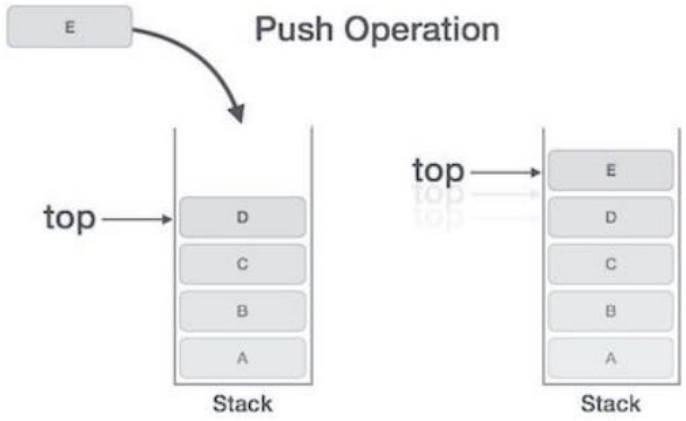
- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows −

```
begin procedure push: stack, data

   if stack is full
      return null
   endif

   top ← top + 1
   stack[top] ← data

end procedure
```

Implementation of this algorithm in C, is very easy. See the following code −



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Example

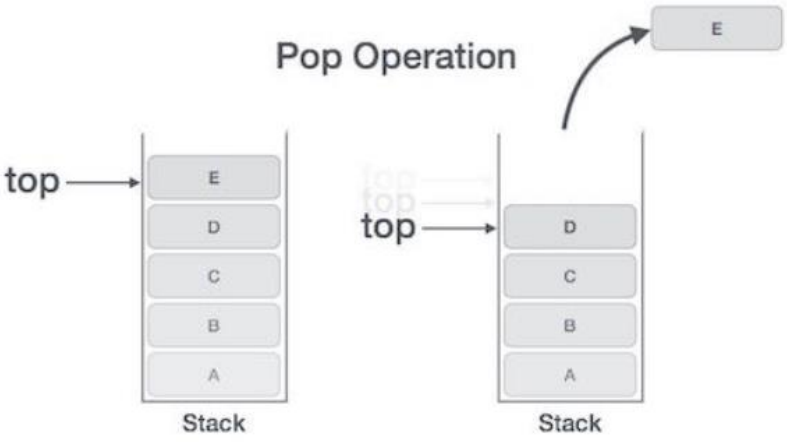
```
void push(int data) {
   if(!isFull()) {
      top = top + 1;
      stack[top] = data;
   } else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.



Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows −

```
begin procedure pop: stack

   if stack is empty
      return null
   endif

   data ← stack[top]
   top ← top - 1
   return data

end procedure
```

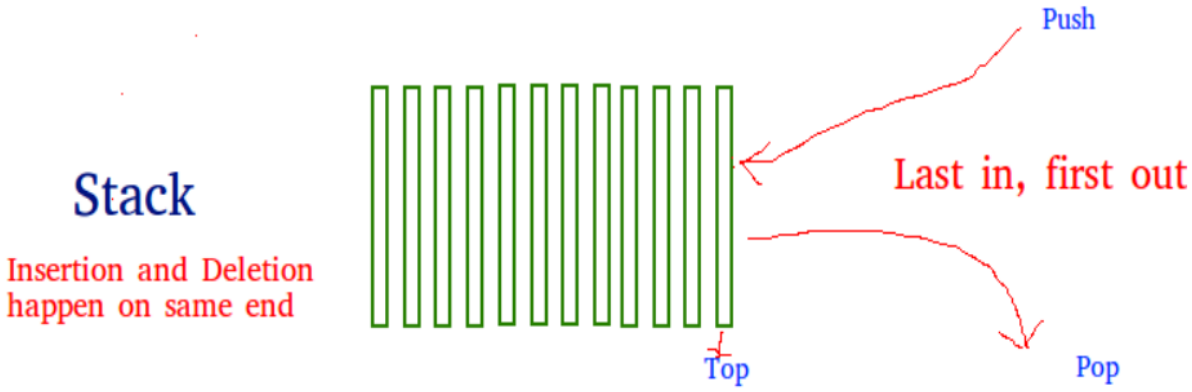
Implementation of this algorithm in C, is as follows −

Example

```
int pop(int data) {

   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}
```

For a complete stack program in C programming language, please click here .

A **stack** is a linear data structure that stores items in a **Last-In/First-Out (LIFO)** or First-In/Last-Out (FILO) manner. In stack, a new element is added at one end and an element is removed from that end only. The insert and delete operations are often called push and pop.



The functions associated with stack are:

- **empty()** – Returns whether the stack is empty – Time Complexity: O(1)
- **size()** – Returns the size of the stack – Time Complexity: O(1)
- **top() / peek()** – Returns a reference to the topmost element of the stack – Time Complexity: O(1)
- **push(a)** – Inserts the element 'a' at the top of the stack – Time Complexity: O(1)
- **pop()** – Deletes the topmost element of the stack – Time Complexity: O(1)

Implementation:

There are various ways from which a stack can be implemented in Python. This article covers the implementation of a stack using data structures and modules from the Python library.
Stack in Python can be implemented using the following ways:

- list
- Collections.deque
- queue.LifoQueue

Implementation using list:

Python's built-in data structure list can be used as a stack. Instead of push(), append() is used to add elements to the top of the stack while pop() removes the element in LIFO order.
Unfortunately, the list has a few shortcomings. The biggest issue is that it can run into speed issues as it grows. The items in the list are stored next to each other in memory, if the stack grows bigger than the block of memory that currently holds it, then Python needs to do some memory allocations. This can lead to some append() calls taking much longer than other ones.

```
# Python program to
# demonstrate stack implementation
# using list

stack = []

# append() function to push
# element in the stack
stack.append('a')
stack.append('b')
stack.append('c')

print('Initial stack')
print(stack)

# pop() function to pop
# element from stack in
# LIFO order
print('\nElements popped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())

print('\nStack after elements are popped:')
print(stack)

# uncommenting print(stack.pop())
# will cause an IndexError
```

 # as the stack is now empty

Output

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

Elements popped from stack:
c
b
a

Stack after elements are popped:
[]
```

Implementation using collections.deque:

Python stack can be implemented using the deque class from the collections module. Deque is preferred over the list in the cases where we need quicker append and pop operations from both the ends of the container, as deque provides an O(1) time complexity for append and pop operations as compared to list which provides O(n) time complexity.

 # Python program to

 # demonstrate stack implementation

 # using collections.deque

from collections import deque

stack = deque()

 # append() function to push

 # element in the stack

stack.append('a')

stack.append('b')

stack.append('c')

```
print('Initial stack:')
print(stack)

# pop() function to pop
# element from stack in
# LIFO order
print('\nElements popped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())

print('\nStack after elements are popped:')
print(stack)

# uncommenting print(stack.pop())
# will cause an IndexError
# as the stack is now empty
```

Output

```
Initial stack:
deque(['a', 'b', 'c'])

Elements popped from stack:
c
b
a

Stack after elements are popped:
deque([])
```

Implementation using queue module

Queue module also has a LIFO Queue, which is basically a Stack. Data is inserted into Queue using the put() function and get() takes data out from the Queue.

There are various functions available in this module:

- **maxsize** – Number of items allowed in the queue.
- **empty()** – Return True if the queue is empty, False otherwise.
- **full()** – Return True if there are *maxsize* items in the queue. If the queue was initialized with maxsize=0 (the default), then full() never returns True.
- **get()** – Remove and return an item from the queue. If the queue is empty, wait until an item is available.
- **get_nowait()** – Return an item if one is immediately available, else raise QueueEmpty.
- **put(item)** – Put an item into the queue. If the queue is full, wait until a free slot is available before adding the item.
- **put_nowait(item)** – Put an item into the queue without blocking. If no free slot is immediately available, raise QueueFull.
- **qsize()** – Return the number of items in the queue.

```
# Python program to
# demonstrate stack implementation
# using queue module

from queue import LifoQueue

# Initializing a stack
stack = LifoQueue(maxsize=3)

# qsize() show the number of elements
# in the stack
print(stack.qsize())

# put() function to push
# element in the stack
```

```
stack.put('a')
stack.put('b')
stack.put('c')

print("Full: ", stack.full())
print("Size: ", stack.qsize())

# get() function to pop
# element from stack in
# LIFO order
print('\nElements popped from the stack')
print(stack.get())
print(stack.get())
print(stack.get())

print("\nEmpty: ", stack.empty())
```

Output

```
0
Full:  True
Size:  3

Elements popped from the stack
c
b
a

Empty:  True
```

Implementation using a singly linked list:

The linked list has two methods addHead(item) and removeHead() that run in constant time. These two methods are suitable to implement a stack.

- **getSize()** – Get the number of items in the stack.
- **isEmpty()** – Return True if the stack is empty, False otherwise.
- **peek()** – Return the top item in the stack. If the stack is empty, raise an exception.
- **push(value)** – Push a value into the head of the stack.
- **pop()** – Remove and return a value in the head of the stack. If the stack is empty, raise an exception.

Below is the implementation of the above approach:

```
# Python program to demonstrate
# stack implementation using a linked list.
# node class

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


class Stack:

    # Initializing a stack.
    # Use a dummy node, which is
    # easier for handling edge cases.
    def __init__(self):
        self.head = Node("head")
        self.size = 0
```

```
# String representation of the stack
def __str__(self):
        cur = self.head.next
        out = ""
        while cur:
                out += str(cur.value) + "->"
                cur = cur.next
        return out[:-3]

# Get the current size of the stack
def getSize(self):
        return self.size

# Check if the stack is empty
def isEmpty(self):
        return self.size == 0

# Get the top item of the stack
def peek(self):

        # Sanitary check to see if we
        # are peeking an empty stack.
        if self.isEmpty():
                raise Exception("Peeking from an empty stack")
        return self.head.next.value

# Push a value into the stack.
def push(self, value):
```

```
            node = Node(value)
            node.next = self.head.next
            self.head.next = node
            self.size += 1

    # Remove a value from the stack and return.
    def pop(self):
            if self.isEmpty():
                    raise Exception("Popping from an empty stack")
            remove = self.head.next
            self.head.next = self.head.next.next
            self.size -= 1
            return remove.value


# Driver Code
if __name__ == "__main__":
    stack = Stack()
    for i in range(1, 11):
            stack.push(i)
    print(f"Stack: {stack}")

    for _ in range(1, 6):
            remove = stack.pop()
            print(f"Pop: {remove}")
    print(f"Stack: {stack}")
```

Output

```
Stack: 10->9->8->7->6->5->4->3->2->
Pop: 10
Pop: 9
Pop: 8
Pop: 7
Pop: 6
Stack: 5->4->3->2->
```

Queue

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −

As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

- **peek()** − Gets the element at the front of the queue without removing it.
- **isfull()** − Checks if the queue is full.
- **isempty()** − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

Let's first learn about supportive functions of a queue −

peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows −

Algorithm

```
begin procedure peek
return queue[front]
end procedure
```

Implementation of peek() function in C programming language −

Example

```
int peek() {
   return queue[front];
}
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function −

Algorithm

```
begin procedure isfull

   if rear equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

Implementation of isfull() function in C programming language −

Example

```
bool isfull() {
   if(rear == MAXSIZE - 1)
      return true;
   else
      return false;
}
```

isempty()

Algorithm of isempty() function −

Algorithm

```
begin procedure isempty

   if front is less than MIN  OR front is greater than rear
      return true
   else
      return false
   endif

end procedure
```

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code −

Example

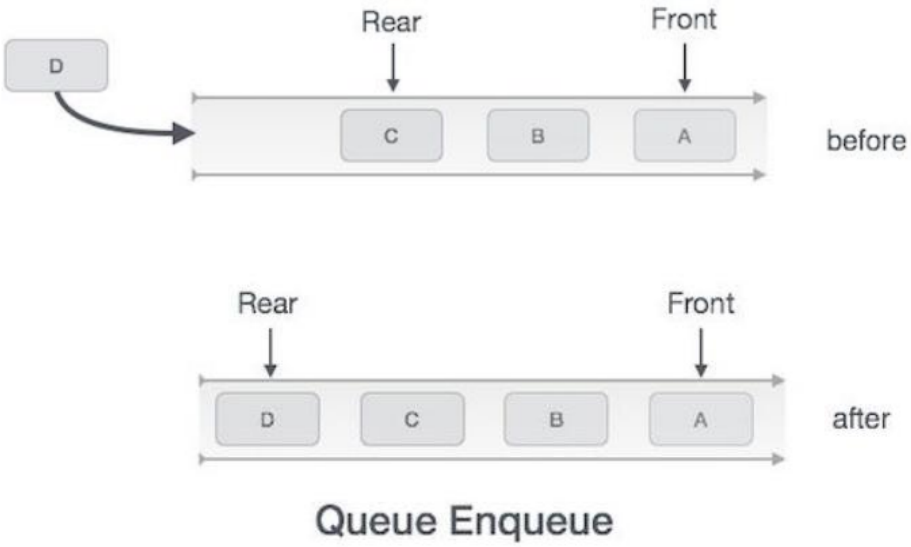
```
bool isempty() {
   if(front < 0 || front > rear)
      return true;
   else
      return false;
}
```

Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − return success.



Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue operation

```
procedure enqueue(data)

   if queue is full
      return overflow
   endif

   rear ← rear + 1
   queue[rear] ← data
   return true

end procedure
```

Implementation of enqueue() in C programming language –

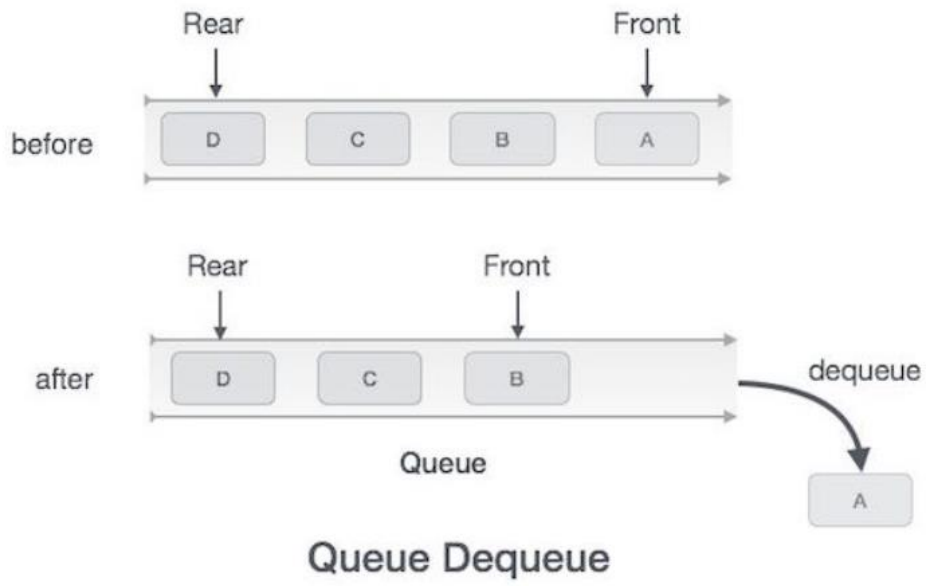
Example

```
int enqueue(int data)
   if(isfull())
      return 0;

   rear = rear + 1;
   queue[rear] = data;

   return 1;
end procedure
```

Dequeue Operation

Accessing data from the queue is a process of two tasks – access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation –

- **Step 1** – Check if the queue is empty.
- **Step 2** – If the queue is empty, produce underflow error and exit.
- **Step 3** – If the queue is not empty, access the data where **front** is pointing.
- **Step 4** – Increment **front** pointer to point to the next available data element.
- **Step 5** – Return success.



Queue Dequeue

Algorithm for dequeue operation

```
procedure dequeue

   if queue is empty
      return underflow
   end if

   data = queue[front]
   front ← front + 1
   return true

end procedure
```

Implementation of dequeue() in C programming language –

Example

```
int dequeue() {
   if(isempty())
      return 0;

   int data = queue[front];
   front = front + 1;

   return data;
}
```

```
# Queue implementation in Python

class Queue:

    def __init__(self):
        self.queue = []

    # Add an element
    def enqueue(self, item):
        self.queue.append(item)

    # Remove an element
    def dequeue(self):
        if len(self.queue) < 1:
            return None
        return self.queue.pop(0)

    # Display  the queue
    def display(self):
        print(self.queue)

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


q = Queue()
q.enqueue(1)
q.enqueue(2)
```

```
q.enqueue(3)
q.enqueue(4)
q.enqueue(5)

q.display()

q.dequeue()

print("After removing an element")
q.display()
```

Implementation using list

List is a Python's built-in data structure that can be used as a queue. Instead of enqueue() and dequeue(), append() and pop() function is used. However, lists are quite slow for this purpose because inserting or deleting an element at the beginning requires shifting all of the other elements by one, requiring O(n) time.

```
# Python program to
# demonstrate queue implementation
# using list

# Initializing a queue
queue = []

# Adding elements to the queue
queue.append('a')
queue.append('b')
queue.append('c')
```

```
print("Initial queue")
print(queue)

# Removing elements from the queue
print("\nElements dequeued from queue")
print(queue.pop(0))
print(queue.pop(0))
print(queue.pop(0))

print("\nQueue after removing elements")
print(queue)

# Uncommenting print(queue.pop(0))
# will raise and IndexError
# as the queue is now empty
```

Output:

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

Elements dequeued from queue
a
b
c

Queue after removing elements
[]
```

Implementation using collections.deque

Queue in Python can be implemented using deque class from the collections module. Deque is preferred over list in the cases where we need quicker append and pop operations from both the ends of container, as deque provides an O(1) time complexity for append and pop operations as compared to list which provides O(n) time complexity. Instead of enqueue and deque, append() and popleft() functions are used.

```
# Python program to
# demonstrate queue implementation
# using collections.dequeue


from collections import deque

# Initializing a queue
q = deque()

# Adding elements to a queue
q.append('a')
q.append('b')
q.append('c')

print("Initial queue")
print(q)

# Removing elements from a queue
print("\nElements dequeued from the queue")
print(q.popleft())
print(q.popleft())
print(q.popleft())
```

print("\nQueue after removing elements")

print(q)

 # Uncommenting q.popleft()

 # will raise an IndexError

 # as queue is now empty

Output:

```
Initial queue
deque(['a', 'b', 'c'])

Elements dequeued from the queue
a
b
c

Queue after removing elements
deque([])
```

Implementation using queue.Queue

Queue is built-in module of Python which is used to implement a queue. queue.Queue(maxsize) initializes a variable to a maximum size of maxsize. A maxsize of zero '0' means a infinite queue. This Queue follows FIFO rule.

There are various functions available in this module:

- **maxsize** – Number of items allowed in the queue.
- **empty()** – Return True if the queue is empty, False otherwise.
- **full()** – Return True if there are maxsize items in the queue. If the queue was initialized with maxsize=0 (the default), then full() never returns True.
- **get()** – Remove and return an item from the queue. If queue is empty, wait until an item is available.
- **get_nowait()** – Return an item if one is immediately available, else raise QueueEmpty.
- **put(item)** – Put an item into the queue. If the queue is full, wait until a free slot is available before adding the item.
- **put_nowait(item)** – Put an item into the queue without blocking. If no free slot is immediately available, raise QueueFull.
- **qsize()** – Return the number of items in the queue.

```
# Python program to
# demonstrate implementation of
# queue using queue module


from queue import Queue

# Initializing a queue
q = Queue(maxsize = 3)

# qsize() give the maxsize
# of the Queue
print(q.qsize())
```

```
# Adding of element to queue
q.put('a')
q.put('b')
q.put('c')

# Return Boolean for Full
# Queue
print("\nFull: ", q.full())

# Removing element from queue
print("\nElements dequeued from the queue")
print(q.get())
print(q.get())
print(q.get())

# Return Boolean for Empty
# Queue
print("\nEmpty: ", q.empty())

q.put(1)
print("\nEmpty: ", q.empty())
print("Full: ", q.full())

# This would result into Infinite
# Loop as the Queue is empty.
# print(q.get())
```

Output:

```
0

Full:  True

Elements dequeued from the queue
a
b
c

Empty:  True

Empty:  False
Full:  False
```

Double Ended (Deque)

A double-ended queue is an abstract data type similar to an simple queue, it allows you to insert and delete from both sides means items can be added or deleted from the front or rear end.

Algorithm for Insertion at rear end

```
Step -1: [Check for overflow]
        if(rear==MAX)
                Print("Queue is Overflow”);
        return;
Step-2: [Insert element]
        else
                rear=rear+1;
                q[rear]=no;
                [Set rear and front pointer]
                if rear=0
                        rear=1;
                if front=0
                        front=1;
Step-3: return
```

Implementation of Insertion at rear end

```
void add_item_rear()
{
        int num;
        printf("\n Enter Item to insert : ");
        scanf("%d",&num);
        if(rear==MAX)
        {
                printf("\n Queue is Overflow");
                return;
        }
        else
```

```
        {
                rear++;
                q[rear]=num;
                if(rear==0)
                        rear=1;
                if(front==0)
                        front=1;
        }
}
```

Algorithm for Insertion at font end

```
Step-1 : [Check for the front position]
        if(front<=1)
                Print ("Cannot add item at front end");
        return;
Step-2  : [Insert at front]
        else
                front=front-1;
                q[front]=no;
Step-3  : Return
```

Implementation of Insertion at font end

```
void add_item_front()
{
        int num;
        printf("\n Enter item to insert:");
        scanf("%d",&num);
        if(front<=1)
        {
                printf("\n Cannot add item at front end");
                return;
        }
        else
```

```
        {
                front--;
                q[front]=num;
        }
}
```

Algorithm for Deletion from front end

```
Step-1 [ Check for front pointer]
        if front=0
                print(" Queue is Underflow”);
        return;
Step-2 [Perform deletion]
        else
                no=q[front];
                print(“Deleted element is”,no);
                [Set front and rear pointer]
        if front=rear
                front=0;
                rear=0;
        else
                front=front+1;
Step-3 : Return
```

Implementation of Deletion from front end

```
void delete_item_front()
{
        int num;
        if(front==0)
        {
                printf("\n Queue is Underflow\n");
                return;
        }
        else
        {
                num=q[front];
                printf("\n Deleted item is %d\n",num);
                if(front==rear)
                {
                        front=0;
                        rear=0;
                }
                else
                {
                        front++;
                }
        }
}
```

Algorithm for Deletion from rear end

```
Step-1 : [Check for the rear pointer]
        if rear=0
                print(“Cannot delete value at rear end”);
        return;
Step-2: [ perform deletion]
        else
                no=q[rear];
                [Check for the front and rear pointer]
        if front= rear
                front=0;
                rear=0;
        else
                rear=rear-1;
                print(“Deleted element is”,no);
Step-3 : Return
```

Implementation of Deletion from rear end

```
void delete_item_rear()
{
        int num;
        if(rear==0)
        {
                printf("\n Cannot delete item at rear end\n");
                return;
        }
        else
        {
                num=q[rear];
                if(front==rear)
                {
                        front=0;
                        rear=0;
                }
```

```
                else
                {
                        rear--;
                        printf("\n Deleted item is %d\n",num);
                }
        }
}
```

Deque (Doubly Ended Queue) in Python is implemented using the module "**collections**". Deque is preferred over a list in the cases where we need quicker append and pop operations from both the ends of the container, as deque provides an **O(1)** time complexity for append and pop operations as compared to list which provides O(n) time complexity.

 # Python code to demonstrate deque

```
from collections import deque

# Declaring deque
queue = deque(['name','age','DOB'])

print(queue)
```

Output:

```
deque(['name', 'age', 'DOB'])
```

Let's see various Operations on deque :

- **append()**:- This function is used to **insert** the value in its argument to the **right end** of the deque.
- **appendleft()**:- This function is used to **insert** the value in its argument to the **left end** of the deque.
- **pop()**:- This function is used to **delete** an argument from the **right end** of the deque.
- **popleft()**:- This function is used to **delete** an argument from the **left end** of the deque.

```
# Python code to demonstrate working of
# append(), appendleft(), pop(), and popleft()

# importing "collections" for deque operations
import collections

# initializing deque
de = collections.deque([1,2,3])

# using append() to insert element at right end
# inserts 4 at the end of deque
```

```
de.append(4)

# printing modified deque
print ("The deque after appending at right is : ")
print (de)

# using appendleft() to insert element at left end
# inserts 6 at the beginning of deque
de.appendleft(6)

# printing modified deque
print ("The deque after appending at left is : ")
print (de)

# using pop() to delete element from right end
# deletes 4 from the right end of deque
de.pop()

# printing modified deque
print ("The deque after deleting from right is : ")
print (de)

# using popleft() to delete element from left end
# deletes 6 from the left end of deque
de.popleft()

# printing modified deque
print ("The deque after deleting from left is : ")
```

print (de)

Output:

```
The deque after appending at right is :
deque([1, 2, 3, 4])
The deque after appending at left is :
deque([6, 1, 2, 3, 4])
The deque after deleting from right is :
deque([6, 1, 2, 3])
The deque after deleting from left is :
deque([1, 2, 3])
```

- **index(ele, beg, end)**:- This function **returns the first index of the value** mentioned in arguments, **starting searching from beg till end** index.
- **insert(i, a)** :- This function **inserts the value** mentioned in arguments(a) **at index(i)** specified in arguments.
- **remove()**:- This function **removes the first occurrence** of value mentioned in arguments.
- **count()**:- This function **counts the number of occurrences** of value mentioned in arguments.

 # Python code to demonstrate working of

 # insert(), index(), remove(), count()

 # importing "collections" for deque operations

import collections

 # initializing deque

de = collections.deque([1, 2, 3, 3, 4, 2, 4])

 # using index() to print the first occurrence of 4

print ("The number 4 first occurs at a position : ")

print (de.index(4,2,5))

 # using insert() to insert the value 3 at 5th position

de.insert(4,3)

 # printing modified deque

print ("The deque after inserting 3 at 5th position is : ")

print (de)

 # using count() to count the occurrences of 3

print ("The count of 3 in deque is : ")

print (de.count(3))

 # using remove() to remove the first occurrence of 3

de.remove(3)

 # printing modified deque

print ("The deque after deleting first occurrence of 3 is : ")

print (de)

Output:

```
The number 4 first occurs at a position :
4
The deque after inserting 3 at 5th position is :
deque([1, 2, 3, 3, 3, 4, 2, 4])
The count of 3 in deque is :
3
The deque after deleting first occurrence of 3 is :
deque([1, 2, 3, 3, 4, 2, 4])
```

- **extend(iterable)**:- This function is used to **add multiple values at the right end** of the deque. The argument passed is iterable.
- **extendleft(iterable)**:- This function is used to **add multiple values at the left end** of the deque. The argument passed is iterable. **Order is reversed** as a result of left appends.
- **reverse()**:- This function is used to **reverse the order** of deque elements.
- **rotate()**:- This function **rotates the deque** by the number specified in arguments. **If the number specified is negative, rotation occurs to the left. Else rotation is to right.**

```
# Python code to demonstrate working of
# extend(), extendleft(), rotate(), reverse()

# importing "collections" for deque operations
import collections

# initializing deque
de = collections.deque([1, 2, 3,])

# using extend() to add numbers to right end
# adds 4,5,6 to right end
de.extend([4,5,6])

# printing modified deque
print ("The deque after extending deque at end is : ")
print (de)

# using extendleft() to add numbers to left end
# adds 7,8,9 to left end
de.extendleft([7,8,9])

# printing modified deque
```

```
print ("The deque after extending deque at beginning is : ")
print (de)

# using rotate() to rotate the deque
# rotates by 3 to left
de.rotate(-3)

# printing modified deque
print ("The deque after rotating deque is : ")
print (de)

# using reverse() to reverse the deque
de.reverse()

# printing modified deque
print ("The deque after reversing deque is : ")
print (de)
```

Output :

```
The deque after extending deque at end is :
deque([1, 2, 3, 4, 5, 6])
The deque after extending deque at beginning is :
deque([9, 8, 7, 1, 2, 3, 4, 5, 6])
The deque after rotating deque is :
deque([1, 2, 3, 4, 5, 6, 9, 8, 7])
The deque after reversing deque is :
deque([7, 8, 9, 6, 5, 4, 3, 2, 1])
```

Linked List

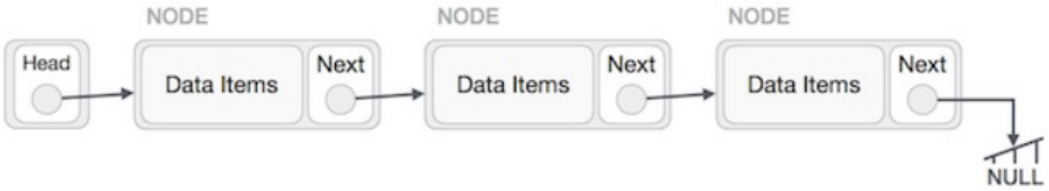
A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** − Each link of a linked list can store a data called an element.
- **Next** − Each link of a linked list contains a link to the next link called Next.
- **LinkedList** − A Linked List contains the connection link to the first link called First.

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.



As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

Types of Linked List

Following are the various types of linked list.

- **Simple Linked List** − Item navigation is forward only.
- **Doubly Linked List** − Items can be navigated forward and backward.
- **Circular Linked List** − Last item contains link of the first element as next and the first element has a link to the last element as previous.

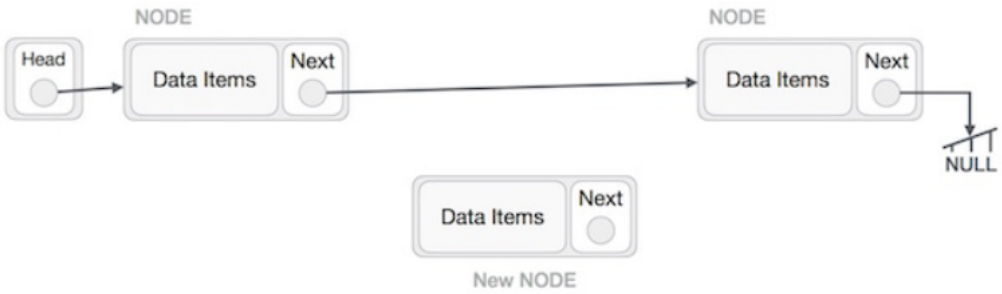
Basic Operations

Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Display** − Displays the complete list.
- **Search** − Searches an element using the given key.
- **Delete** − Deletes an element using the given key.

Insertion Operation

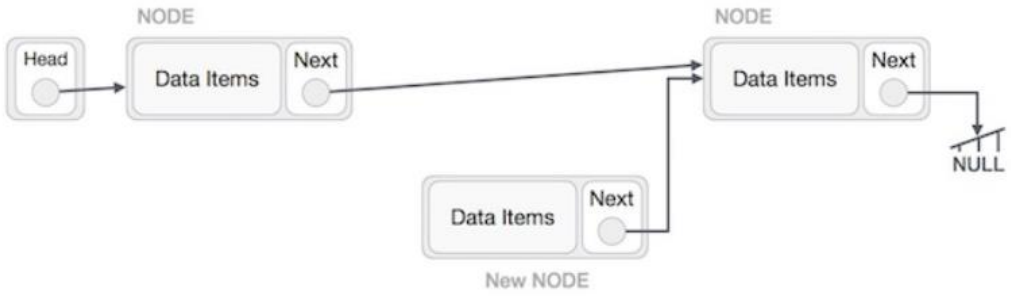
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode) and **C** (RightNode). Then point B.next to C −

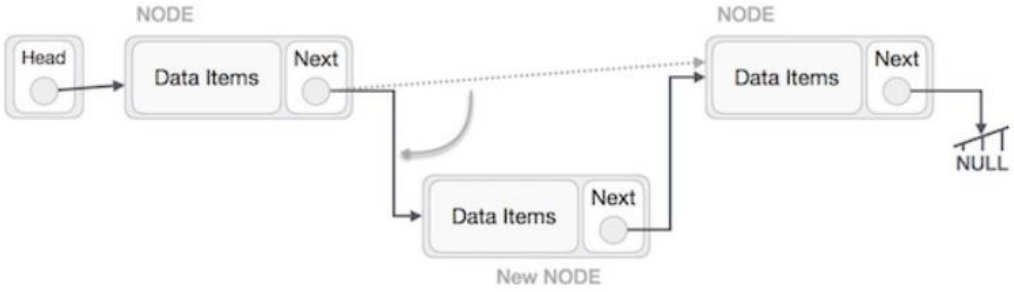
```
NewNode.next -> RightNode;
```

It should look like this −



Now, the next node at the left should point to the new node.

```
LeftNode.next -> NewNode;
```

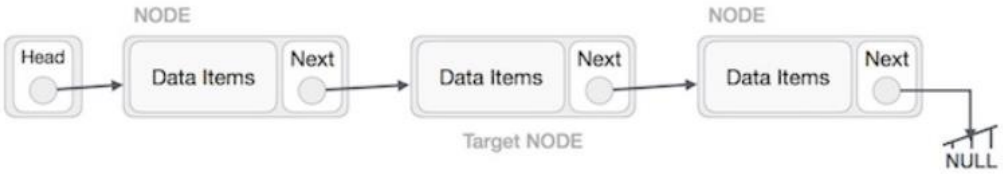


This will put the new node in the middle of the two. The new list should look like this −

Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.

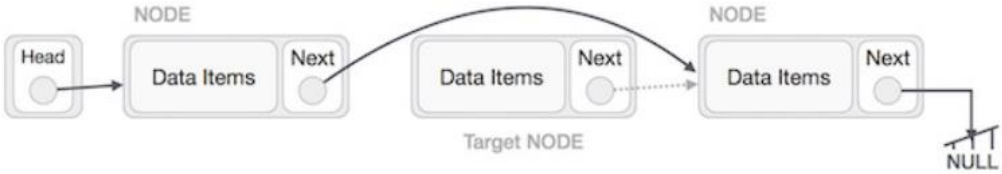


The left (previous) node of the target node now should point to the next node of the target node −

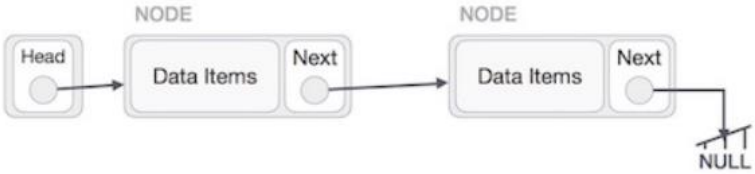
```
LeftNode.next -> TargetNode.next;
```

This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

```
TargetNode.next -> NULL;
```

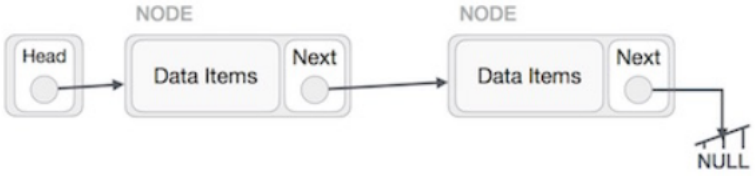


We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

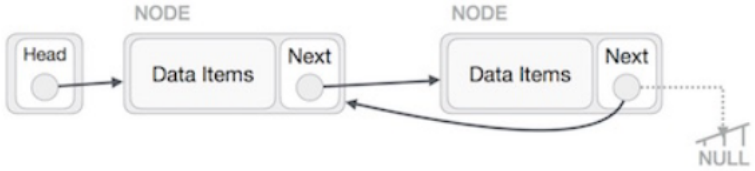


Reverse Operation

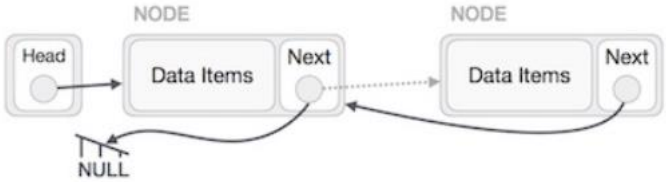
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



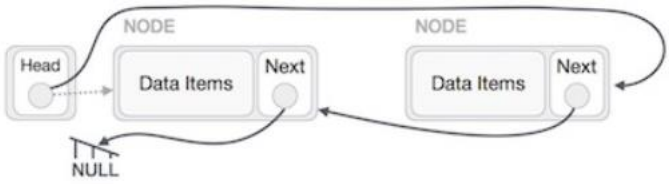
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node −



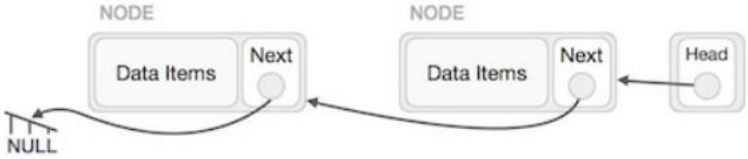
We have to make sure that the last node is not the last node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.



We'll make the head node point to the new first node by using the temp node.



Representation:

A linked list is represented by a pointer to the first node of the linked list. The first node is called the head. If the linked list is empty, then the value of the head points to NULL.

Each node in a list consists of at least two parts:

1. A Data Item (we can store integer, strings or any type of data).
2. Pointer (Or Reference) to the next node (connects one node to another) or An address of another node

In C, we can represent a node using structures. Below is an example of a linked list node with integer data.
In Java or C#, LinkedList can be represented as a class and a Node as a separate class. The LinkedList class contains a reference of Node class type.

```
# Node class
class Node:

    # Function to initialize the node object
    def __init__(self, data):
        self.data = data # Assign data
        self.next = None # Initialize
                                    # next as null

# Linked List class
class LinkedList:

    # Function to initialize the Linked
    # List object
    def __init__(self):
        self.head = None
```

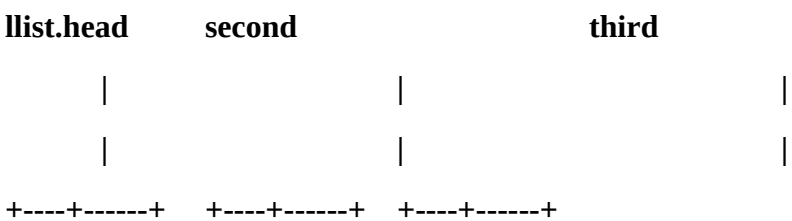
```
# A simple Python program to introduce a linked list

# Node class
class Node:

    # Function to initialise the node object
    def __init__(self, data):
        self.data = data # Assign data
        self.next = None # Initialize next as null


# Linked List class contains a Node object
class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None


# Code execution starts here
if __name__=='__main__':

    # Start with the empty list
    llist = LinkedList()

    llist.head = Node(1)
    second = Node(2)
    third = Node(3)
```

'''

Three nodes have been created.

We have references to these three blocks as head,

second and third



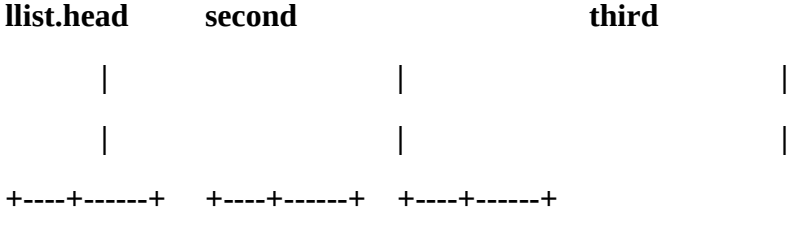
'''

llist.head.next = second; # Link first node with second

'''

Now next of first Node refers to second. So they

both are linked.



'''

```
second.next = third; # Link second node with the third node

'''
Now next of second Node refers to third. So all three
nodes are linked.

llist.head        second                        third
        |                    |                            |
        |                    |                            |
+----+------+    +----+------+    +----+------+
| 1 | o-------->| 2 | o-------->| 3 | null |
+----+------+    +----+------+    +----+------+
'''
```

Linked List Traversal

In the previous program, we have created a simple linked list with three nodes. Let us traverse the created list and print the data of each node. For traversal, let us write a general-purpose function printList() that prints any given list.

```
# A simple Python program for traversal of a linked list

# Node class
class Node:

    # Function to initialise the node object
    def __init__(self, data):
        self.data = data # Assign data
        self.next = None # Initialize next as null
```

```
# Linked List class contains a Node object
class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # This function prints contents of linked list
    # starting from head
    def printList(self):
        temp = self.head
        while (temp):
            print (temp.data)
            temp = temp.next


# Code execution starts here
if __name__=='__main__':

    # Start with the empty list
    llist = LinkedList()

    llist.head = Node(1)
    second = Node(2)
    third = Node(3)

    llist.head.next = second; # Link first node with second
    second.next = third; # Link second node with the third node
```

llist.printList()

Output:

```
1  2  3
```

Linked List Complexity:

Time Complexity

Time Complexity	Worst Case	Average Case
Search	O(n)	O(n)
Insert	O(1)	O(1)
Deletion	O(1)	O(1)

Auxiliary Space: O(n)

Important Links :

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

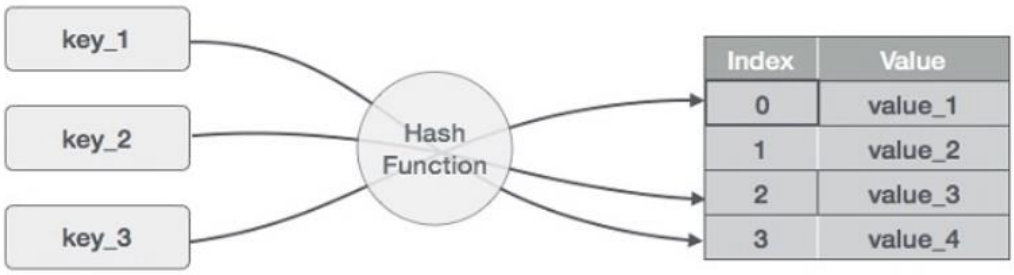
Hash map:

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

Hashing

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.



- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)
- (17,11)
- (13,78)
- (37,98)

Sr.No.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

Linear Probing

As we can see, it may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.

Sr.No.	Key	Hash	Array Index	After Linear Probing, Array Index
1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2
3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12
6	14	14 % 20 = 14	14	14
7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13
9	37	37 % 20 = 17	17	18

Basic Operations

Following are the basic primary operations of a hash table.

- **Search** − Searches an element in a hash table.
- **Insert** − inserts an element in a hash table.
- **delete** − Deletes an element from a hash table.

DataItem

Define a data item having some data and key, based on which the search is to be conducted in a hash table.

```
struct DataItem {
int data;
int key;
};
```

Hash Method

Define a hashing method to compute the hash code of the key of the data item.

```
int hashCode(int key){
   return key % SIZE;
}
```

Search Operation

Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code.

Example

```
struct DataItem *search(int key) {
   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
   while(hashArray[hashIndex] != NULL) {

      if(hashArray[hashIndex]->key == key)
         return hashArray[hashIndex];

      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }

   return NULL;
}
```

Insert Operation

Whenever an element is to be inserted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code.

Example

```
void insert(int key,int data) {
   struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem));
   item->data = data;
   item->key = key;

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty or deleted cell
   while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1) {
      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }

   hashArray[hashIndex] = item;
}
```

Delete Operation

Whenever an element is to be deleted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing to get the element ahead if an element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.

Example

```
struct DataItem* delete(struct DataItem* item) {
   int key = item->key;

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
   while(hashArray[hashIndex] !=NULL) {

      if(hashArray[hashIndex]->key == key) {
         struct DataItem* temp = hashArray[hashIndex];

         //assign a dummy item at deleted position
         hashArray[hashIndex] = dummyItem;
         return temp;
      }

      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }

   return NULL;
}
```

What is a Hash table or a Hashmap in Python?

In computer science, a Hash table or a Hashmap is a type of data structure that maps keys to its value pairs (implement abstract array data types). It basically makes use of a function that computes an index value that in turn holds the elements to be searched, inserted, removed, etc. This makes it easy and fast to access data. In general, hash tables store key-value pairs and the key is generated using a hash function.

Hash tables or has maps in Python are implemented through the built-in dictionary data type. The keys of a dictionary in Python are generated by a hashing function. The elements of a dictionary are not ordered and they can be changed.

An example of a dictionary can be a mapping of employee names and their employee IDs or the names of students along with their student IDs.

Moving ahead, let's see the difference between the hash table and hashmap in Python.

Hash Table vs hashmap: Difference between Hash Table and Hashmap in Python

Hash Table	Hashmap
Synchronized	Non-Synchronized
Fast	Slow
Allows one null key and more than one null values	Does not allows null keys or values

Hash maps are indexed data structures. A hash map makes use of a hash function to compute an index with a key into an array of buckets or slots. Its value is mapped to the bucket with the corresponding index. The key is unique and immutable. Think of a hash map as a cabinet having drawers with labels for the things stored in them. For example, storing user information- consider email as the key, and we can map values corresponding to that user such as the first name, last name etc to a bucket.

Hash function is the core of implementing a hash map. It takes in the key and translates it to the index of a bucket in the bucket list. Ideal hashing should produce a different index for each key. However, collisions can occur. When hashing gives an existing index, we can simply use a bucket for multiple values by appending a list or by rehashing.

In Python, dictionaries are examples of hash maps. We'll see the implementation of hash map from scratch in order to learn how to build and customize such data structures for optimizing search.

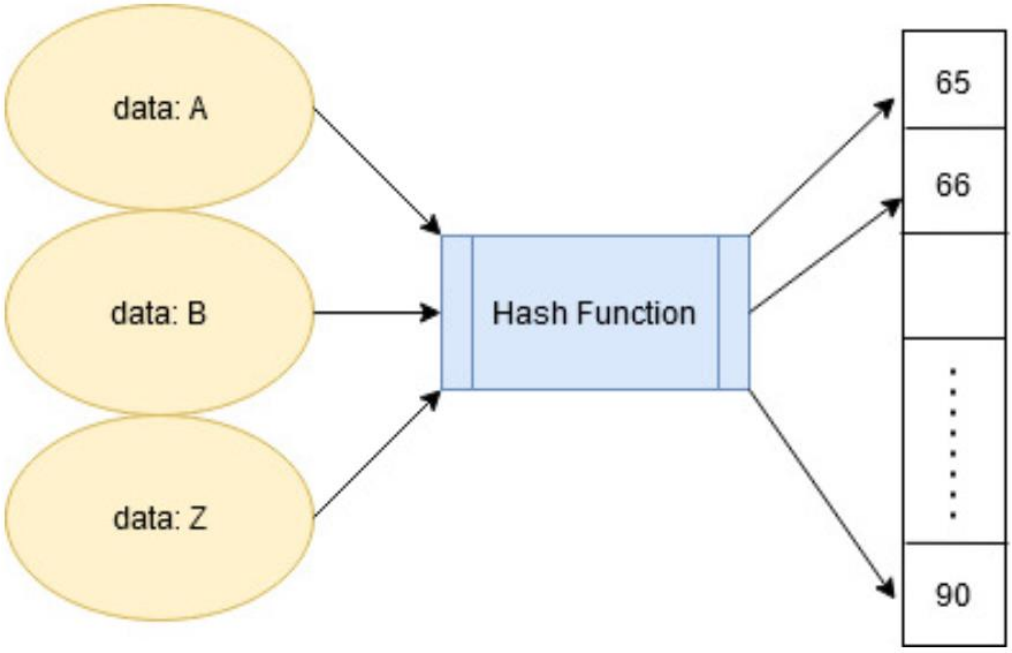
The hash map design will include the following functions:

- **set_val(key, value):** Inserts a key-value pair into the hash map. If the value already exists in the hash map, update the value.
- **get_val(key):** Returns the value to which the specified key is mapped, or "No record found" if this map contains no mapping for the key.
- **delete_val(key):** Removes the mapping for the specific key if the hash map contains the mapping for the key.

> *Hashing is the process of transforming data and mapping it to a range of values which can be efficiently looked up.*

In this article, we have explored the idea of collision in hashing and explored different collision resolution techniques such as:

- Open Hashing (Separate chaining)
- Closed Hashing (Open Addressing)
  - Liner Probing
  - Quadratic probing
  - Double hashing



- *Hash table*: a data structure where the data is stored based upon its hashed key which is obtained using a hashing function.
- *Hash function*: a function which for a given data, outputs a value mapped to a fixed range. A hash table leverages the hash function to efficiently map data such that it can be retrieved and updated quickly. Simply put, assume S = {s1, s2, s3, ..., sn} to be a set of objects that we wish to store into a map of size N, so we use a hash function H, such that for all s belonging to S; H(s) -> x, where x is guaranteed to lie in the range [1,N]
- *Perfect Hash function*: a hash function that maps each item into a unique slot (no collisions).

Hash Collisions: As per the Pigeonhole principle if the set of objects we intend to store within our hash table is larger than the size of our hash table we are bound to have two or more different objects having the same hash value; a hash collision. Even if the size of the hash table is large enough to accommodate all the objects finding a hash function which generates a unique hash for each object in the hash table is a difficult task. Collisions are bound to occur (unless we find a perfect hash function, which in most of the cases is hard to find) but can be significantly reduced with the help of various collision resolution techniques.

Following are the collision resolution techniques used:

- Open Hashing (Separate chaining)
- Closed Hashing (Open Addressing)
  - Liner Probing
  - Quadratic probing
  - Double hashing

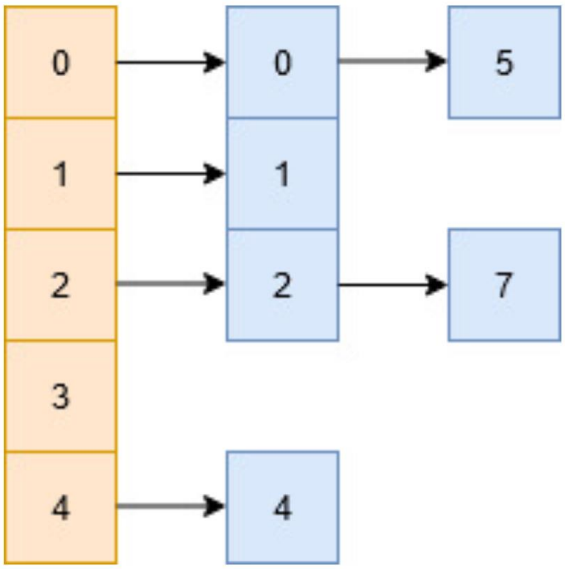
1. Open Hashing (Separate chaining)

Collisions are resolved using a list of elements to store objects with the same key together.

- Suppose you wish to store a set of numbers = {0,1,2,4,5,7} into a hash table of size 5.
- Now, assume that we have a hash function H, such that `H(x) = x%5`
- So, if we were to map the given data with the given hash function we'll get the corresponding values

```
H(0)-> 0%5 = 0
H(1)-> 1%5 = 1
H(2)-> 2%5 = 2
H(4)-> 4%5 = 4
H(5)-> 5%5 = 0
H(7)-> 7%5 = 2
```

- Clearly 0 and 5, as well as 2 and 7 will have the same hash value, and in this case we'll simply append the colliding values to a list being pointed by their hash keys.



Obviously in practice the table size can be significantly large and the hash function can be even more complex, also the data being hashed would be more complex and non-primitive, but the idea remains the same.
This is an easy way to implement hashing but it has its own demerits.

- The lookups/inserts/updates can become linear [O(N)] instead of constant time [O(1)] if the hash function has too many collisions.
- It doesn't account for any empty slots which can be leveraged for more efficient storage and lookups.
- Ideally we require a good hash function to guarantee even distribution of the values.
- Say, for a **load factor**
  `λ=number of objects stored in table/size of the table (can be >1)`
  a good hash function would guarantee that the maximum length of list associated with each key is close to the load factor.

Note that the order in which the data is stored in the lists (or any other data structures) is based upon the implementation requirements. Some general ways include insertion order, frequency of access etc.

2. Closed Hashing (Open Addressing)

This collision resolution technique requires a hash table with fixed and known size. During insertion, if a collision is encountered, alternative cells are tried until an empty bucket is found. These techniques require the size of the hash table to be supposedly larger than the number of objects to be stored (something with a load factor < 1 is ideal).
There are various methods to find these empty buckets:

```
a. Liner Probing
b. Quadratic probing
c. Double hashing
```

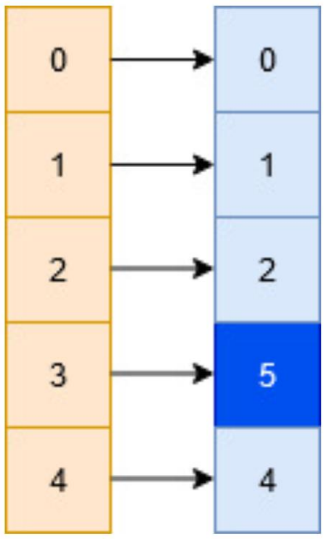
a. Linear Probing

The idea of linear probing is simple, we take a fixed sized hash table and every time we face a hash collision we linearly traverse the table in a cyclic manner to find the next empty slot.

- Assume a scenario where we intend to store the following set of numbers = {0,1,2,4,5,7} into a hash table of size 5 with the help of the following hash function H, such that `H(x) = x%5`.
- So, if we were to map the given data with the given hash function we'll get the corresponding values

```
H(0)-> 0%5 = 0
H(1)-> 1%5 = 1
H(2)-> 2%5 = 2
H(4)-> 4%5 = 4
H(5)-> 5%5 = 0
```

- in this case we see a collision of two terms (0 & 5). In this situation we move linearly down the table to find the first empty slot. Note that this linear traversal is cyclic in nature, i.e. in the event we exhaust the last element during the search we start again from the beginning until the initial key is reached.



- In this case our hash function can be considered as this:
  `H(x, i) = (H(x) + i)%N`
  where N is the size of the table and i represents the linearly increasing variable which starts from 1 (until empty bucket is found).

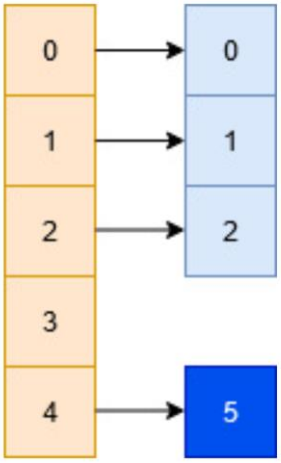
Despite being easy to compute, implement and deliver best cache performance, this suffers from the problem of clustering (many consecutive elements get grouped together, which eventually reduces the efficiency of finding elements or empty buckets).

b. Quadratic Probing

This method lies in the middle of great cache performance and the problem of clustering. The general idea remains the same, the only difference is that we look at the Q(i) increment at each iteration when looking for an empty bucket, where Q(i) is some quadratic expression of i. A simple expression of Q would be Q(i) = i^2, in which case the hash function looks something like this:

`H(x, i) = (H(x) + i^2)%N`

- In general, `H(x, i) = (H(x) + ((c1\*i^2 + c2\*i + c3)))%N`, for some choice of constants c1, c2, and c3
- Despite resolving the problem of clustering significantly it may be the case that in some situations this technique does not find any available bucket, unlike linear probing which always finds an empty bucket.
- Luckily, we can get good results from quadratic probing with the right combination of probing function and hash table size which will guarantee that we will visit as many slots in the table as possible. In particular, if the hash table's size is a prime number and the probing function is `H(x, i) = i^2`, then at least 50% of the slots in the table will be visited. Thus, if the table is less than half full, we can be certain that a free slot will eventually be found.
- Alternatively, if the hash table size is a power of two and the probing function is `H(x, i) = (i^2 + i)/2`, then every slot in the table will be visited by the probing function.
- Assume a scenario where we intend to store the following set of numbers = {0,1,2,5} into a hash table of size 5 with the help of the following hash function H, such that `H(x, i) = (x%5 + i^2)%5`.



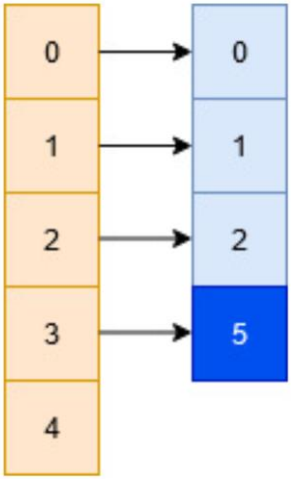
Clearly 5 and 0 will face a collision, in which case we'll do the following:

```
- we look at 5%5 = 0 (collision)
- we look at (5%5 + 1^2)%5 = 1 (collision)
- we look at (5%5 + 2^2)%5 = 4 (empty -> place element here)
```

c. Double Hashing

This method is based upon the idea that in the event of a collision we use an another hashing function with the key value as an input to find where in the open addressing scheme the data should actually be placed at.

- In this case we use two hashing functions, such that the final hashing function looks like:
  `H(x, i) = (H1(x) + i*H2(x))%N`
- Typically for `H1(x) = x%N` a good H2 is `H2(x) = P - (x%P)`, where P is a prime number smaller than N.
- A good H2 is a function which never evaluates to zero and ensures that all the cells of a table are effectively traversed.
- Assume a scenario where we intend to store the following set of numbers = {0,1,2,5} into a hash table of size 5 with the help of the following hash function H, such that
  `H(x, i) = (H1(x) + i*H2(x))%5`
  `H1(x) = x%5` and `H2(x) = P - (x%P)`, where P = 3
  (3 is a prime smaller than 5)



Clearly 5 and 0 will face a collision, in which case we'll do the following:

```
- we look at 5%5 = 0 (collision)
- we look at (5%5 + 1*(3 - (5%3)))%5 = 1 (collision)
- we look at (5%5 + 2*(3 - (5%3)))%5 = 2 (collision)
- we look at (5%5 + 3*(3 - (5%3)))%5 = 3 (empty -> place element here)
```

```
class HashTable:

    # Create empty bucket list of given size
    def __init__(self, size):
        self.size = size
        self.hash_table = self.create_buckets()

    def create_buckets(self):
        return [[] for _ in range(self.size)]

    # Insert values into hash map
    def set_val(self, key, val):

        # Get the index from the key
        # using hash function
        hashed_key = hash(key) % self.size

        # Get the bucket corresponding to index
        bucket = self.hash_table[hashed_key]

        found_key = False
        for index, record in enumerate(bucket):
            record_key, record_val = record

            # check if the bucket has same key as
            # the key to be inserted
            if record_key == key:
                found_key = True
```

```
                    break

            # If the bucket has same key as the key to be inserted,
            # Update the key value
            # Otherwise append the new key-value pair to the bucket
            if found_key:
                    bucket[index] = (key, val)
            else:
                    bucket.append((key, val))

    # Return searched value with specific key
    def get_val(self, key):

            # Get the index from the key using
            # hash function
            hashed_key = hash(key) % self.size

            # Get the bucket corresponding to index
            bucket = self.hash_table[hashed_key]

            found_key = False
            for index, record in enumerate(bucket):
                    record_key, record_val = record

                    # check if the bucket has same key as
                    # the key being searched
                    if record_key == key:
                            found_key = True
```

```
                    break

        # If the bucket has same key as the key being searched,
        # Return the value found
        # Otherwise indicate there was no record found
        if found_key:
            return record_val
        else:
            return "No record found"

# Remove a value with specific key
def delete_val(self, key):

        # Get the index from the key using
        # hash function
        hashed_key = hash(key) % self.size

        # Get the bucket corresponding to index
        bucket = self.hash_table[hashed_key]

        found_key = False
        for index, record in enumerate(bucket):
            record_key, record_val = record

            # check if the bucket has same key as
            # the key to be deleted
            if record_key == key:
                found_key = True
```

```
                        break
                if found_key:
                        bucket.pop(index)
                return

        # To print the items of hash map
        def __str__(self):
                return "".join(str(item) for item in self.hash_table)


hash_table = HashTable(50)

# insert some values
hash_table.set_val('gfg@example.com', 'some value')
print(hash_table)
print()

hash_table.set_val('portal@example.com', 'some other value')
print(hash_table)
print()

# search/access a record with key
print(hash_table.get_val('portal@example.com'))
print()

# delete or remove a value
hash_table.delete_val('portal@example.com')
print(hash_table)
```

Output:

```
[][][][][][][][][][][][][][][][][][][][][][][('gfg@example.com', 'some value')][][][][][]
[][][][][][][][][][][][][][][][][][][][][][]

[][][][][][][][][][][][][][][][][][][][][][][('gfg@example.com', 'some value')][][][][][]
[('portal@example.com', 'some other value')][][][][][][][][][][][][][][][][][][][][][]

some other value

[][][][][][][][][][][][][][][][][][][][][][][('gfg@example.com', 'some value')][][][][][]
[][][][][][][][][][][][][][][][][][][][][][]
```

Time Complexity:

Memory index access takes constant time and hashing takes constant time. Hence, the search complexity of a hash map is also constant time, that is, O(1).

Heap

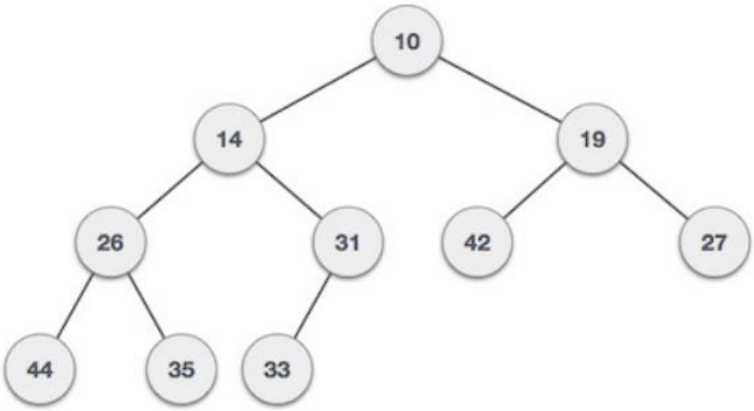
Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly. If **α** has child node **β** then −

key(α) ≥ key(β)

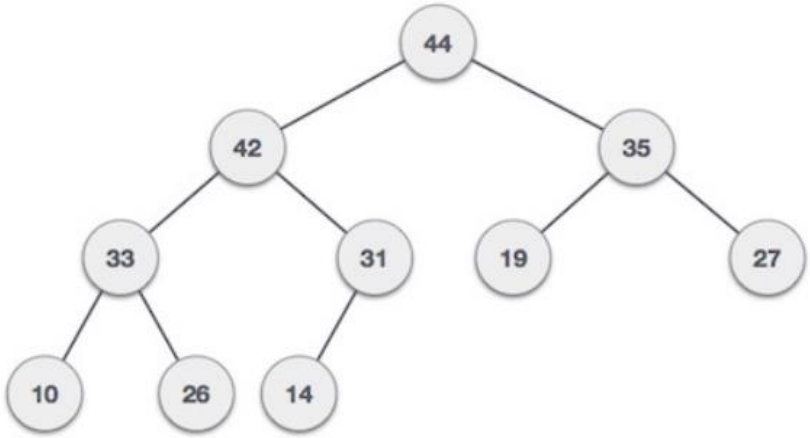
As the value of parent is greater than that of child, this property generates **Max Heap**. Based on this criteria, a heap can be of two types −

```
For Input → 35 33 42 10 14 19 27 44 26 31
```

Min-Heap − Where the value of the root node is less than or equal to either of its children.



Max-Heap − Where the value of the root node is greater than or equal to either of its children.



Both trees are constructed using the same input and order of arrival.

Max Heap Construction Algorithm

We shall use the same example to demonstrate how a Max Heap is created. The procedure to create Min Heap is similar but we go for min values instead of max values.

We are going to derive an algorithm for max heap by inserting one element at a time. At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.

```
Step 1 − Create a new node at the end of heap.
Step 2 − Assign new value to the node.
Step 3 − Compare the value of this child node with its parent.
Step 4 − If value of parent is less than child, then swap them.
Step 5 − Repeat step 3 & 4 until Heap property holds.
```

Note − In Min Heap construction algorithm, we expect the value of the parent node to be less than that of the child node.

Let's understand Max Heap construction by an animated illustration. We consider the same input sample that we used earlier.

Max Heap Deletion Algorithm

Let us derive an algorithm to delete from max heap. Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.

```
Step 1 − Remove root node.
Step 2 − Move the last element of last level to root.
Step 3 − Compare the value of this child node with its parent.
Step 4 − If value of parent is less than child, then swap them.
Step 5 − Repeat step 3 & 4 until Heap property holds.
```

How is Min Heap represented ?

A Min Heap is a Complete Binary Tree. A Min heap is typically represented as an array. The root element will be at **Arr[0]**. For any ith node, i.e., **Arr[i]**:

- **Arr[(i -1) / 2]** returns its parent node.
- **Arr[(2 * i) + 1]** returns its left child node.
- **Arr[(2 * i) + 2]** returns its right child node.

Operations on Min Heap :

1. **getMin()**: It returns the root element of Min Heap. Time Complexity of this operation is **O(1)**.
2. **extractMin()**: Removes the minimum element from MinHeap. Time Complexity of this Operation is **O(Log n)** as this operation needs to maintain the heap property (by calling heapify()) after removing root.
3. **insert()**: Inserting a new key takes **O(Log n)** time. We add a new key at the end of the tree. If new key is larger than its parent, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.

```
# Python3 implementation of Min Heap

import sys

class MinHeap:

    def __init__(self, maxsize):
        self.maxsize = maxsize
        self.size = 0
        self.Heap = [0]*(self.maxsize + 1)
        self.Heap[0] = -1 * sys.maxsize
        self.FRONT = 1

    # Function to return the position of
    # parent for the node currently
    # at pos
```

```
def parent(self, pos):
        return pos//2

# Function to return the position of
# the left child for the node currently
# at pos
def leftChild(self, pos):
        return 2 * pos

# Function to return the position of
# the right child for the node currently
# at pos
def rightChild(self, pos):
        return (2 * pos) + 1

# Function that returns true if the passed
# node is a leaf node
def isLeaf(self, pos):
        return pos*2 > self.size

# Function to swap two nodes of the heap
def swap(self, fpos, spos):
        self.Heap[fpos], self.Heap[spos] = self.Heap[spos], self.Heap[fpos]

# Function to heapify the node at pos
def minHeapify(self, pos):

        # If the node is a non-leaf node and greater
```

```
            # than any of its child
            if not self.isLeaf(pos):
                if (self.Heap[pos] > self.Heap[self.leftChild(pos)] or
                self.Heap[pos] > self.Heap[self.rightChild(pos)]):

                    # Swap with the left child and heapify
                    # the left child
                    if self.Heap[self.leftChild(pos)] <
self.Heap[self.rightChild(pos)]:
                        self.swap(pos, self.leftChild(pos))
                        self.minHeapify(self.leftChild(pos))

                    # Swap with the right child and heapify
                    # the right child
                    else:
                        self.swap(pos, self.rightChild(pos))
                        self.minHeapify(self.rightChild(pos))

        # Function to insert a node into the heap
        def insert(self, element):
            if self.size >= self.maxsize :
                return
            self.size+= 1
            self.Heap[self.size] = element

            current = self.size

            while self.Heap[current] < self.Heap[self.parent(current)]:
                self.swap(current, self.parent(current))
```

```
                current = self.parent(current)

    # Function to print the contents of the heap
    def Print(self):
        for i in range(1, (self.size//2)+1):
            print(" PARENT : "+ str(self.Heap[i])+" LEFT CHILD : "+
                                    str(self.Heap[2 * i])+" RIGHT
CHILD : "+
                                    str(self.Heap[2 * i + 1]))

    # Function to build the min heap using
    # the minHeapify function
    def minHeap(self):

        for pos in range(self.size//2, 0, -1):
            self.minHeapify(pos)

    # Function to remove and return the minimum
    # element from the heap
    def remove(self):

        popped = self.Heap[self.FRONT]
        self.Heap[self.FRONT] = self.Heap[self.size]
        self.size-= 1
        self.minHeapify(self.FRONT)
        return popped

# Driver Code
if __name__ == "__main__":
```

```
print('The minHeap is ')
minHeap = MinHeap(15)
minHeap.insert(5)
minHeap.insert(3)
minHeap.insert(17)
minHeap.insert(10)
minHeap.insert(84)
minHeap.insert(19)
minHeap.insert(6)
minHeap.insert(22)
minHeap.insert(9)
minHeap.minHeap()

minHeap.Print()
print("The Min val is " + str(minHeap.remove()))
```

Output :

```
The Min Heap is
 PARENT : 3 LEFT CHILD : 5 RIGHT CHILD :6
 PARENT : 5 LEFT CHILD : 9 RIGHT CHILD :84
 PARENT : 6 LEFT CHILD : 19 RIGHT CHILD :17
 PARENT : 9 LEFT CHILD : 22 RIGHT CHILD :10
The Min val is 3
```

Using Library functions :

We use heapq class to implement Heaps in Python. By default Min Heap is implemented by this class.

```
# Python3 program to demonstrate working of heapq
```

```
from heapq import heapify, heappush, heappop

# Creating empty heap
heap = []
heapify(heap)

# Adding items to the heap using heappush function
heappush(heap, 10)
heappush(heap, 30)
heappush(heap, 20)
heappush(heap, 400)

# printing the value of minimum element
print("Head value of heap : "+str(heap[0]))

# printing the elements of the heap
print("The heap elements : ")
for i in heap:
    print(i, end = ' ')
print("\n")

element = heappop(heap)

# printing the elements of the heap
print("The heap elements : ")
for i in heap:
    print(i, end = ' ')
```

Output :

```
Head value of heap : 10
The heap elements :
10 30 20 400

The heap elements :
20 30 400
```

How is Max Heap represented?

A max Heap is a Complete Binary Tree. A max heap is typically represented as an array. The root element will be at Arr[0]. Below table shows indexes of other nodes for the ith node, i.e., Arr[i]:

- Arr[(i-1)/2] Returns the parent node.
- Arr[(2*i)+1] Returns the left child node.
- Arr[(2*i)+2] Returns the right child node.

Operations on Max Heap:

1. **getMax()**: It returns the root element of Max Heap. Time Complexity of this operation is **O(1)**.
2. **extractMax()**: Removes the maximum element from MaxHeap. Time Complexity of this Operation is **O(log n)** as this operation needs to maintain the heap property (by calling heapify()) after removing the root.
3. **insert()**: Inserting a new key takes **O(log n)** time. We add a new key at the end of the tree. If the new key is smaller than its parent, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.

Note: In the below implementation, we do indexing from index 1 to simplify the implementation.

```
# Python3 implementation of Max Heap

import sys


class MaxHeap:


    def __init__(self, maxsize):
```

```
        self.maxsize = maxsize
        self.size = 0
        self.Heap = [0] * (self.maxsize + 1)
        self.Heap[0] = sys.maxsize
        self.FRONT = 1

# Function to return the position of
# parent for the node currently
# at pos
def parent(self, pos):

        return pos // 2

# Function to return the position of
# the left child for the node currently
# at pos
def leftChild(self, pos):

        return 2 * pos

# Function to return the position of
# the right child for the node currently
# at pos
def rightChild(self, pos):

        return (2 * pos) + 1
```

```
# Function that returns true if the passed
# node is a leaf node
def isLeaf(self, pos):

        if pos >= (self.size//2) and pos <= self.size:
                return True
        return False

# Function to swap two nodes of the heap
def swap(self, fpos, spos):

        self.Heap[fpos], self.Heap[spos] = (self.Heap[spos],

self.Heap[fpos])

# Function to heapify the node at pos
def maxHeapify(self, pos):

        # If the node is a non-leaf node and smaller
        # than any of its child
        if not self.isLeaf(pos):
                if (self.Heap[pos] < self.Heap[self.leftChild(pos)] or
                        self.Heap[pos] < self.Heap[self.rightChild(pos)]):

                        # Swap with the left child and heapify
                        # the left child
                        if (self.Heap[self.leftChild(pos)] >
                                self.Heap[self.rightChild(pos)]):
                                self.swap(pos, self.leftChild(pos))
```

```
                    self.maxHeapify(self.leftChild(pos))

                # Swap with the right child and heapify
                # the right child
                else:
                    self.swap(pos, self.rightChild(pos))
                    self.maxHeapify(self.rightChild(pos))

# Function to insert a node into the heap
def insert(self, element):

    if self.size >= self.maxsize:
        return
    self.size += 1
    self.Heap[self.size] = element

    current = self.size

    while (self.Heap[current] >
        self.Heap[self.parent(current)]):
        self.swap(current, self.parent(current))
        current = self.parent(current)

# Function to print the contents of the heap
def Print(self):

    for i in range(1, (self.size // 2) + 1):
        print("PARENT : " + str(self.Heap[i]) +
```

```
                        "LEFT CHILD : " + str(self.Heap[2 * i]) +
                        "RIGHT CHILD : " + str(self.Heap[2 * i + 1]))

    # Function to remove and return the maximum
    # element from the heap
    def extractMax(self):

        popped = self.Heap[self.FRONT]
        self.Heap[self.FRONT] = self.Heap[self.size]
        self.size -= 1
        self.maxHeapify(self.FRONT)

        return popped

# Driver Code
if __name__ == "__main__":

    print('The maxHeap is ')

    maxHeap = MaxHeap(15)
    maxHeap.insert(5)
    maxHeap.insert(3)
    maxHeap.insert(17)
    maxHeap.insert(10)
    maxHeap.insert(84)
    maxHeap.insert(19)
    maxHeap.insert(6)
    maxHeap.insert(22)
```

```
maxHeap.insert(9)

maxHeap.Print()

print("The Max val is " + str(maxHeap.extractMax()))
```

Output

```
The maxHeap is
PARENT : 84LEFT CHILD : 22RIGHT CHILD : 19
PARENT : 22LEFT CHILD : 17RIGHT CHILD : 10
PARENT : 19LEFT CHILD : 5RIGHT CHILD : 6
PARENT : 17LEFT CHILD : 3RIGHT CHILD : 9
The Max val is 84
```

Using Library functions:

We use heapq class to implement Heap in Python. By default Min Heap is implemented by this class. But we multiply each value by -1 so that we can use it as MaxHeap.

```
# Python3 program to demonstrate working of heapq

from heapq import heappop, heappush, heapify

# Creating empty heap
heap = []
heapify(heap)

# Adding items to the heap using heappush
# function by multiplying them with -1
heappush(heap, -1 * 10)
heappush(heap, -1 * 30)
heappush(heap, -1 * 20)
```

```
heappush(heap, -1 * 400)

# printing the value of maximum element
print("Head value of heap : " + str(-1 * heap[0]))

# printing the elements of the heap
print("The heap elements : ")
for i in heap:
    print((-1*i), end=" ")
print("\n")

element = heappop(heap)

# printing the elements of the heap
print("The heap elements : ")
for i in heap:
    print(-1 * i, end = ' ')
```

Output

```
Head value of heap : 400
The heap elements :
400 30 20 10

The heap elements :
30 10 20
```

Graph

A graph is an abstract notation used to represent the connection between pairs of objects. A graph consists of −

- **Vertices** − Interconnected objects in a graph are called vertices. Vertices are also known as **nodes**.
- **Edges** − Edges are the links that connect the vertices.

There are two types of graphs −

- **Directed graph** − In a directed graph, edges have direction, i.e., edges go from one vertex to another.
- **Undirected graph** − In an undirected graph, edges have no direction.

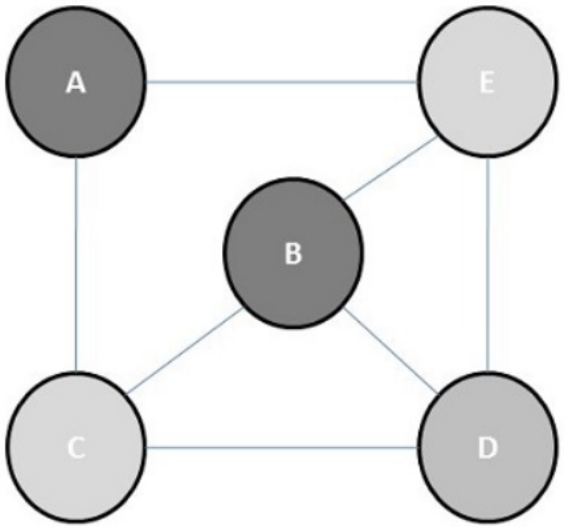
Graph Coloring

Graph coloring is a method to assign colors to the vertices of a graph so that no two adjacent vertices have the same color. Some graph coloring problems are −

- **Vertex coloring** − A way of coloring the vertices of a graph so that no two adjacent vertices share the same color.
- **Edge Coloring** − It is the method of assigning a color to each edge so that no two adjacent edges have the same color.
- **Face coloring** − It assigns a color to each face or region of a planar graph so that no two faces that share a common boundary have the same color.

Chromatic Number

Chromatic number is the minimum number of colors required to color a graph. For example, the chromatic number of the following graph is 3.



The concept of graph coloring is applied in preparing timetables, mobile radio frequency assignment, Suduku, register allocation, and coloring of maps.

Steps for graph coloring

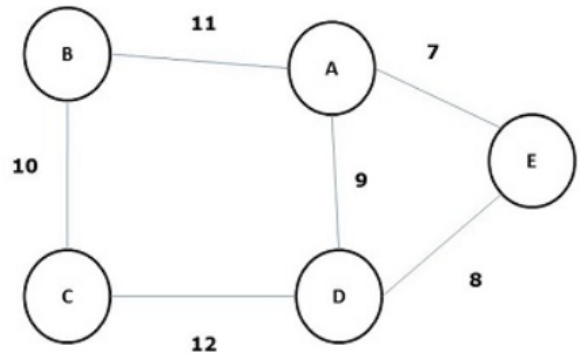
- Set the initial value of each processor in the n-dimensional array to 1.
- Now to assign a particular color to a vertex, determine whether that color is already assigned to the adjacent vertices or not.
- If a processor detects same color in the adjacent vertices, it sets its value in the array to 0.
- After making n^2 comparisons, if any element of the array is 1, then it is a valid coloring.

Pseudocode for graph coloring

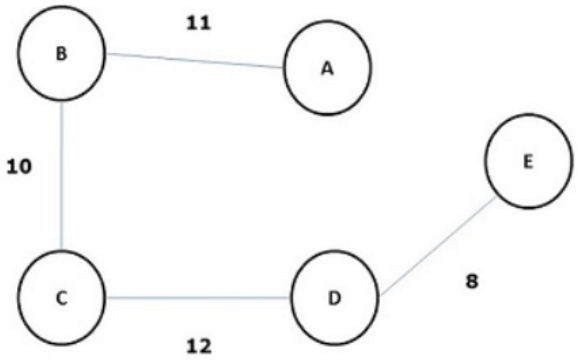
```
begin

   create the processors P(i0,i1,...in-1) where 0_iv < m, 0 _ v < n
   status[i0,..in-1] = 1

   for j varies from 0 to n-1 do
      begin

         for k varies from 0 to n-1 do
         begin
            if aj,k=1 and ij=ikthen
            status[i0,..in-1] =0
         end

      end
      ok = ΣStatus

   if ok > 0, then display valid coloring exists
   else
      display invalid coloring

end
```

Minimal Spanning Tree

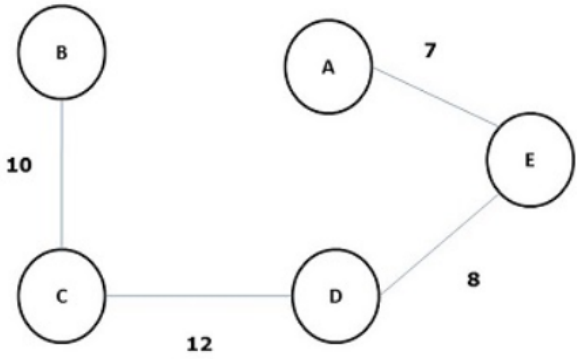
A spanning tree whose sum of weight (or length) of all its edges is less than all other possible spanning tree of graph G is known as a **minimal spanning tree** or **minimum cost spanning** tree. The following figure shows a weighted connected graph.



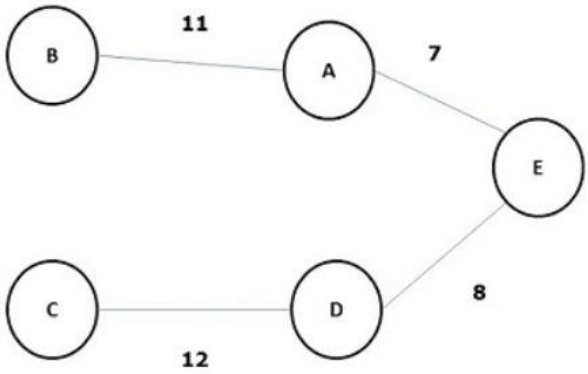
Some possible spanning trees of the above graph are shown below −



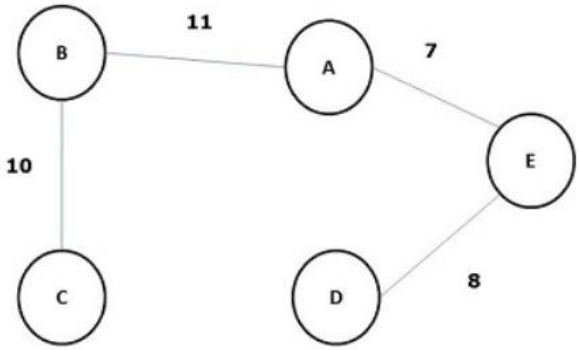
Total weight =11+10+12+8=41

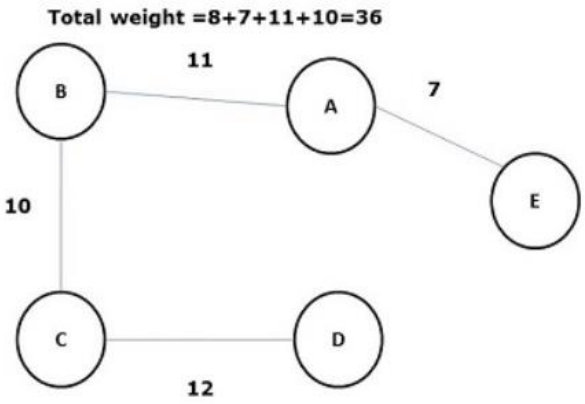


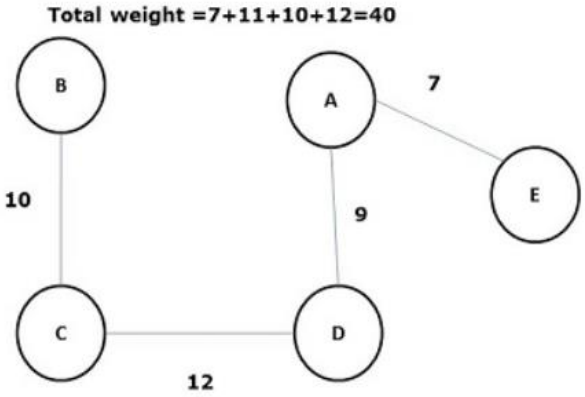
Total weight =10+12+8+7=37

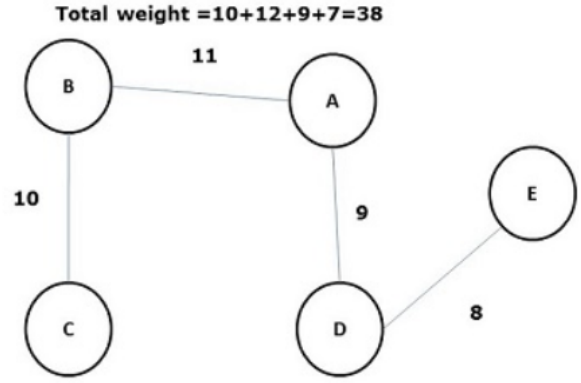


Total weight =12+8+7+11=38









Total weight =10+11+9+8=38

Among all the above spanning trees, figure (d) is the minimum spanning tree. The concept of minimum cost spanning tree is applied in travelling salesman problem, designing electronic circuits, Designing efficient networks, and designing efficient routing algorithms.

To implement the minimum cost-spanning tree, the following two methods are used −

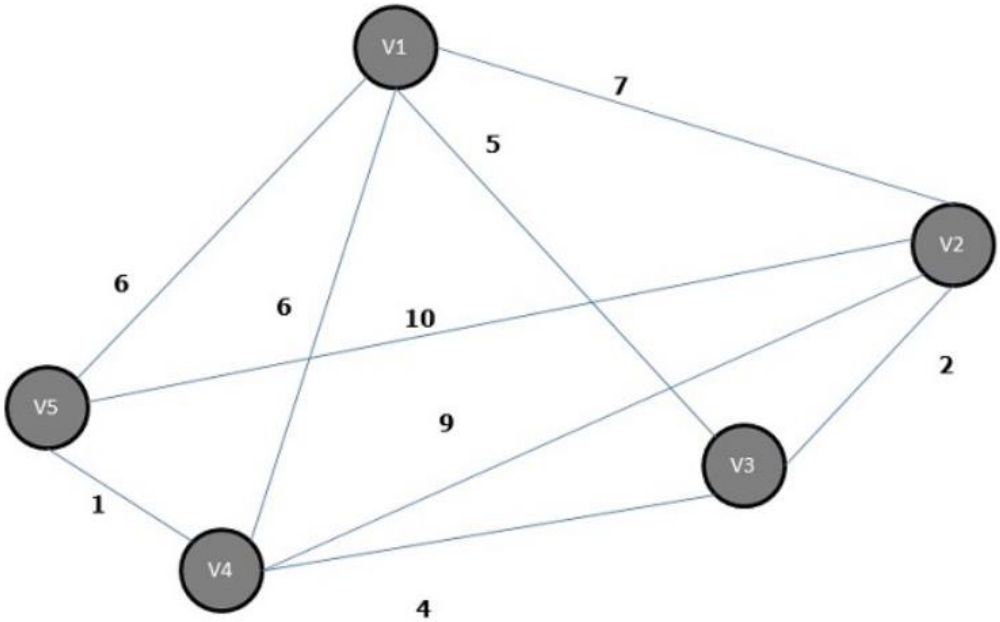
- Prim's Algorithm
- Kruskal's Algorithm

Prim's Algorithm

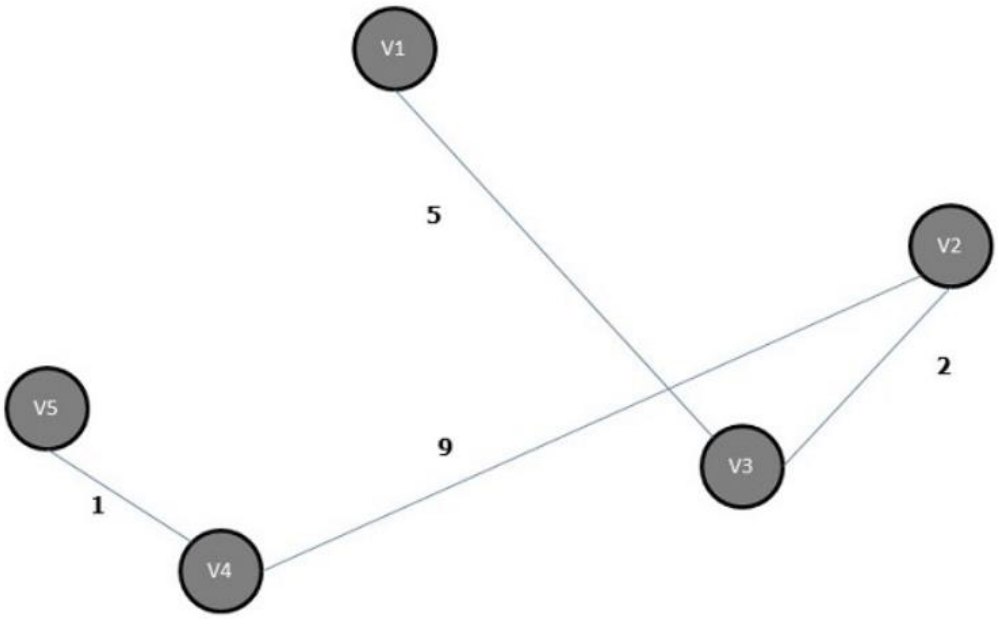
Prim's algorithm is a greedy algorithm, which helps us find the minimum spanning tree for a weighted undirected graph. It selects a vertex first and finds an edge with the lowest weight incident on that vertex.

Steps of Prim's Algorithm

- Select any vertex, say v_1 of Graph G.
- Select an edge, say e_1 of G such that $e_1 = v_1 v_2$ and $v_1 \neq v_2$ and e_1 has minimum weight among the edges incident on v_1 in graph G.
- Now, following step 2, select the minimum weighted edge incident on v_2.
- Continue this till n–1 edges have been chosen. Here **n** is the number of vertices.



The minimum spanning tree is −

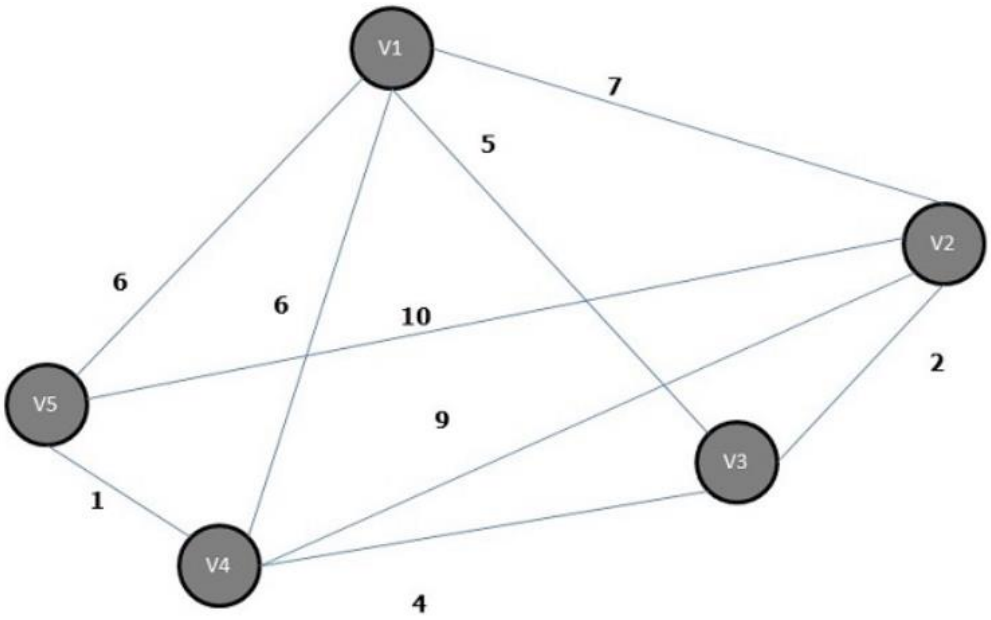


Kruskal's Algorithm

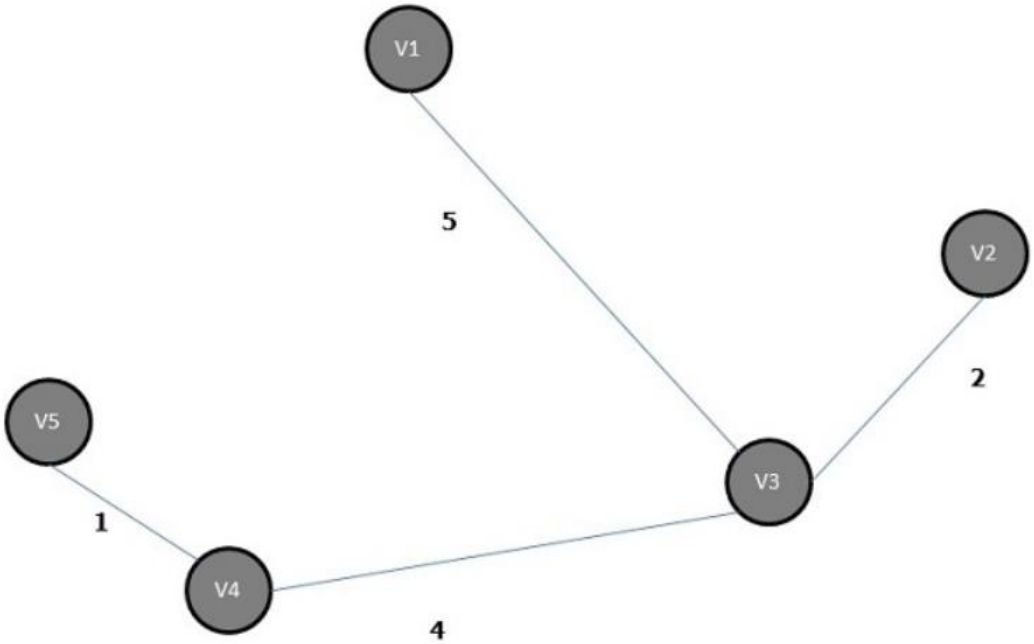
Kruskal's algorithm is a greedy algorithm, which helps us find the minimum spanning tree for a connected weighted graph, adding increasing cost arcs at each step. It is a minimum-spanning-tree algorithm that finds an edge of the least possible weight that connects any two trees in the forest.

Steps of Kruskal's Algorithm

- Select an edge of minimum weight; say e_1 of Graph G and e_1 is not a loop.
- Select the next minimum weighted edge connected to e_1.
- Continue this till n–1 edges have been chosen. Here **n** is the number of vertices.



The minimum spanning tree of the above graph is −



Shortest Path Algorithm

Shortest Path algorithm is a method of finding the least cost path from the source node(S) to the destination node (D). Here, we will discuss Moore's algorithm, also known as Breadth First Search Algorithm.

Moore's algorithm

- Label the source vertex, S and label it **i** and set **i=0**.
- Find all unlabeled vertices adjacent to the vertex labeled **i**. If no vertices are connected to the vertex, S, then vertex, D, is not connected to S. If there are vertices connected to S, label them **i+1**.
- If D is labeled, then go to step 4, else go to step 2 to increase i=i+1.
- Stop after the length of the shortest path is found.

Basic Greedy Coloring Algorithm:

1. Color first vertex with first color.
2. Do following for remaining V-1 vertices.
*..... **a)** Consider the currently picked vertex and color it with the lowest numbered color that has not been used on any previously colored vertices adjacent to it. If all previously used colors appear on vertices adjacent to v, assign a new color to it.*

```
# Python3 program to implement greedy
# algorithm for graph coloring

def addEdge(adj, v, w):

    adj[v].append(w)

    # Note: the graph is undirected
    adj[w].append(v)
    return adj
```

```
# Assigns colors (starting from 0) to all
# vertices and prints the assignment of colors
def greedyColoring(adj, V):

    result = [-1] * V

    # Assign the first color to first vertex
    result[0] = 0;



    # A temporary array to store the available colors.
    # True value of available[cr] would mean that the
    # color cr is assigned to one of its adjacent vertices
    available = [False] * V

    # Assign colors to remaining V-1 vertices
    for u in range(1, V):

        # Process all adjacent vertices and
        # flag their colors as unavailable
        for i in adj[u]:
            if (result[i] != -1):
                available[result[i]] = True

        # Find the first available color
        cr = 0
        while cr < V:
```

```
                if (available[cr] == False):
                    break

                cr += 1

            # Assign the found color
            result[u] = cr

            # Reset the values back to false
            # for the next iteration
            for i in adj[u]:
                if (result[i] != -1):
                    available[result[i]] = False

        # Print the result
        for u in range(V):
            print("Vertex", u, " ---> Color", result[u])

# Driver Code
if __name__ == '__main__':

        g1 = [[] for i in range(5)]
        g1 = addEdge(g1, 0, 1)
        g1 = addEdge(g1, 0, 2)
        g1 = addEdge(g1, 1, 2)
        g1 = addEdge(g1, 1, 3)
        g1 = addEdge(g1, 2, 3)
        g1 = addEdge(g1, 3, 4)
```

```
print("Coloring of graph 1 ")
greedyColoring(g1, 5)

g2 = [[] for i in range(5)]
g2 = addEdge(g2, 0, 1)
g2 = addEdge(g2, 0, 2)
g2 = addEdge(g2, 1, 2)
g2 = addEdge(g2, 1, 4)
g2 = addEdge(g2, 2, 4)
g2 = addEdge(g2, 4, 3)
print("\nColoring of graph 2")
greedyColoring(g2, 5)

# This code is contributed by mohit kumar 29
```

Output:

```
Coloring of graph 1
Vertex 0 --->  Color 0
Vertex 1 --->  Color 1
Vertex 2 --->  Color 2
Vertex 3 --->  Color 0
Vertex 4 --->  Color 1

Coloring of graph 2
Vertex 0 --->  Color 0
Vertex 1 --->  Color 1
Vertex 2 --->  Color 2
Vertex 3 --->  Color 0
Vertex 4 --->  Color 3
```

Time Complexity: O(V^2 + E) in worst case.

```
# A Python program for Prim's Minimum Spanning Tree (MST) algorithm.
# The program is for adjacency matrix representation of the graph

import sys # Library for INT_MAX

class Graph():

    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    # A utility function to print the constructed MST stored in parent[]
    def printMST(self, parent):
        print ("Edge \tWeight")
        for i in range(1, self.V):
            print (parent[i], "-", i, "\t", self.graph[i][parent[i]])

    # A utility function to find the vertex with
    # minimum distance value, from the set of vertices
    # not yet included in shortest path tree
    def minKey(self, key, mstSet):

        # Initialize min value
        min = sys.maxsize

        for v in range(self.V):
            if key[v] < min and mstSet[v] == False:
```

```
                min = key[v]
                min_index = v

        return min_index

# Function to construct and print MST for a graph
# represented using adjacency matrix representation
def primMST(self):

        # Key values used to pick minimum weight edge in cut
        key = [sys.maxsize] * self.V
        parent = [None] * self.V # Array to store constructed MST
        # Make key 0 so that this vertex is picked as first vertex
        key[0] = 0
        mstSet = [False] * self.V

        parent[0] = -1 # First node is always the root of

        for cout in range(self.V):

                # Pick the minimum distance vertex from
                # the set of vertices not yet processed.
                # u is always equal to src in first iteration
                u = self.minKey(key, mstSet)

                # Put the minimum distance vertex in
                # the shortest path tree
                mstSet[u] = True
```

```
                # Update dist value of the adjacent vertices
                # of the picked vertex only if the current
                # distance is greater than new distance and
                # the vertex in not in the shortest path tree
                for v in range(self.V):

                        # graph[u][v] is non zero only for adjacent vertices of m
                        # mstSet[v] is false for vertices not yet included in MST
                        # Update the key only if graph[u][v] is smaller than key[v]
                        if self.graph[u][v] > 0 and mstSet[v] == False and key[v] >
self.graph[u][v]:
                                key[v] = self.graph[u][v]
                                parent[v] = u

        self.printMST(parent)

g = Graph(5)
g.graph = [ [0, 2, 0, 6, 0],
            [2, 0, 3, 8, 5],
            [0, 3, 0, 0, 7],
            [6, 8, 0, 0, 9],
            [0, 5, 7, 9, 0]]

g.primMST();

# Contributed by Divyanshu Mehta
```

Output:

```
Edge   Weight
0 - 1    2
1 - 2    3
0 - 3    6
1 - 4    5
```

The Time Complexity of the above program is O(V^2). If the input graph is represented using adjacency list, then the time complexity of Prim's algorithm can be reduced to O(E log V) with the help of a binary heap. In this implementation, we are always considering the spanning tree to start from the root of the graph, and this is the basic difference between Kruskal's Minimum Spanning Tree and Prim's Minimum Spanning tree.

```
# Python program for Kruskal's algorithm to find
# Minimum Spanning Tree of a given connected,
# undirected and weighted graph

from collections import defaultdict

# Class to represent a graph


class Graph:

    def __init__(self, vertices):
        self.V = vertices # No. of vertices
        self.graph = [] # default dictionary
        # to store graph

    # function to add an edge to graph
    def addEdge(self, u, v, w):
```

```
        self.graph.append([u, v, w])

    # A utility function to find set of an element i
    # (uses path compression technique)
    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    # A function that does union of two sets of x and y
    # (uses union by rank)
    def union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)

        # Attach smaller rank tree under root of
        # high rank tree (Union by Rank)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot

        # If ranks are same, then make one as root
        # and increment its rank by one
        else:
            parent[yroot] = xroot
            rank[xroot] += 1
```

```
# The main function to construct MST using Kruskal's
        # algorithm
def KruskalMST(self):

        result = [] # This will store the resultant MST

        # An index variable, used for sorted edges
        i = 0

        # An index variable, used for result[]
        e = 0

        # Step 1: Sort all the edges in
        # non-decreasing order of their
        # weight. If we are not allowed to change the
        # given graph, we can create a copy of graph
        self.graph = sorted(self.graph,
                                            key=lambda item: item[2])

        parent = []
        rank = []

        # Create V subsets with single elements
        for node in range(self.V):
                parent.append(node)
                rank.append(0)

        # Number of edges to be taken is equal to V-1
```

```
        while e < self.V - 1:

            # Step 2: Pick the smallest edge and increment
            # the index for next iteration
            u, v, w = self.graph[i]
            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent, v)

            # If including this edge doesn't
            # cause cycle, include it in result
            # and increment the indexof result
            # for next edge
            if x != y:
                e = e + 1
                result.append([u, v, w])
                self.union(parent, rank, x, y)
            # Else discard the edge

        minimumCost = 0
        print ("Edges in the constructed MST")
        for u, v, weight in result:
            minimumCost += weight
            print("%d -- %d == %d" % (u, v, weight))
        print("Minimum Spanning Tree" , minimumCost)

# Driver code
g = Graph(4)
```

g.addEdge(0, 1, 10)

g.addEdge(0, 2, 6)

g.addEdge(0, 3, 5)

g.addEdge(1, 3, 15)

g.addEdge(2, 3, 4)

 # Function call

g.KruskalMST()

 # This code is contributed by Neelam Yadav

Output

```
Following are the edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Minimum Cost Spanning Tree: 19
```

Time Complexity: O(ElogE) or O(ElogV). Sorting of edges takes O(ELogE) time. After sorting, we iterate through all edges and apply the find-union algorithm. The find and union operations can take at most O(LogV) time. So overall complexity is O(ELogE + ELogV) time. The value of E can be at most $O(V^2)$, so O(LogV) is O(LogE) the same. Therefore, the overall time complexity is O(ElogE) or O(ElogV)

Shortest Path Algorithm

 # Python program for Dijkstra's single

 # source shortest path algorithm. The program is

 # for adjacency matrix representation of the graph

class Graph():

    def __init__(self, vertices):

```
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                                for row in range(vertices)]

def printSolution(self, dist):
        print("Vertex \t Distance from Source")
        for node in range(self.V):
                print(node, "\t\t", dist[node])

# A utility function to find the vertex with
# minimum distance value, from the set of vertices
# not yet included in shortest path tree
def minDistance(self, dist, sptSet):

        # Initialize minimum distance for next node
        min = 1e7

        # Search not nearest vertex not in the
        # shortest path tree
        for v in range(self.V):
                if dist[v] < min and sptSet[v] == False:
                        min = dist[v]
                        min_index = v

        return min_index

# Function that implements Dijkstra's single source
# shortest path algorithm for a graph represented
```

```
# using adjacency matrix representation
def dijkstra(self, src):

        dist = [1e7] * self.V
        dist[src] = 0
        sptSet = [False] * self.V

        for cout in range(self.V):

                # Pick the minimum distance vertex from
                # the set of vertices not yet processed.
                # u is always equal to src in first iteration
                u = self.minDistance(dist, sptSet)

                # Put the minimum distance vertex in the
                # shortest path tree
                sptSet[u] = True

                # Update dist value of the adjacent vertices
                # of the picked vertex only if the current
                # distance is greater than new distance and
                # the vertex in not in the shortest path tree
                for v in range(self.V):
                        if (self.graph[u][v] > 0 and
                        sptSet[v] == False and
                        dist[v] > dist[u] + self.graph[u][v]):
                                dist[v] = dist[u] + self.graph[u][v]
```

```
            self.printSolution(dist)

# Driver program
g = Graph(9)
g.graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0],
           [4, 0, 8, 0, 0, 0, 0, 11, 0],
           [0, 8, 0, 7, 0, 4, 0, 0, 2],
           [0, 0, 7, 0, 9, 14, 0, 0, 0],
           [0, 0, 0, 9, 0, 10, 0, 0, 0],
           [0, 0, 4, 14, 10, 0, 2, 0, 0],
           [0, 0, 0, 0, 0, 2, 0, 1, 6],
           [8, 11, 0, 0, 0, 0, 1, 0, 7],
           [0, 0, 2, 0, 0, 0, 6, 7, 0]
           ]

g.dijkstra(0)

# This code is contributed by Divyanshu Mehta
```

Output

```
Vertex      Distance from Source
0           0
1           4
2           12
3           19
4           21
5           11
6           9
7           8
8           14
```

Bloom Filter

Suppose you are creating an account on Geekbook, you want to enter a cool username, you entered it and got a message, "Username is already taken". You added your birth date along username, still no luck. Now you have added your university roll number also, still got "Username is already taken". It's really frustrating, isn't it?
But have you ever thought about how quickly Geekbook checks availability of username by searching millions of username registered with it. There are many ways to do this job –

- **Linear search** : Bad idea!
- **Binary Search** : Store all username alphabetically and compare entered username with middle one in list, If it matched, then username is taken otherwise figure out, whether entered username will come before or after middle one and if it will come after, neglect all the usernames before middle one(inclusive). Now search after middle one and repeat this process until you got a match or search end with no match. This technique is better and promising but still it requires multiple steps.
  But, there must be something better!!

Bloom Filter is a data structure that can do this job.
For understanding bloom filters, you must know what is hashing. A hash function takes input and outputs a unique identifier of fixed length which is used for identification of input.

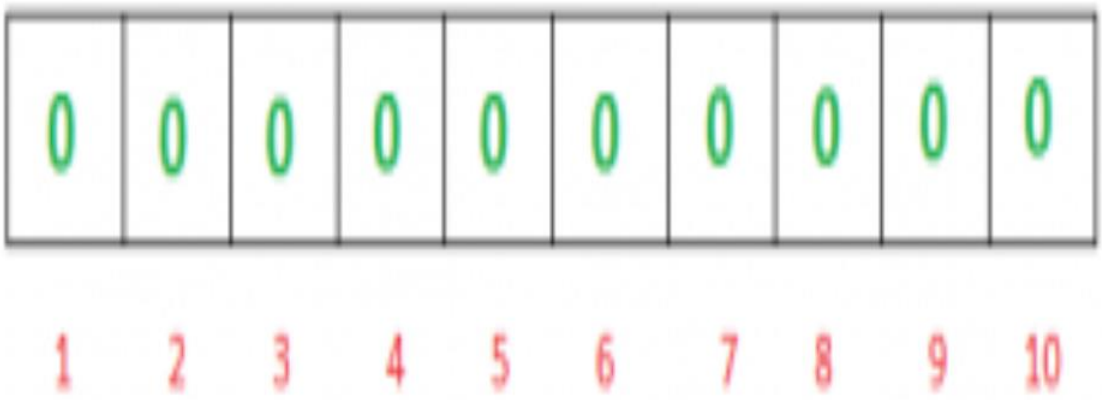
What is Bloom Filter?

A Bloom filter is a **space-efficient probabilistic** data structure that is used to test whether an element is a member of a set. For example, checking availability of username is set membership problem, where the set is the list of all registered username. The price we pay for efficiency is that it is probabilistic in nature that means, there might be some False Positive results. **False positive means**, it might tell that given username is already taken but actually it's not.
Interesting Properties of Bloom Filters

- Unlike a standard hash table, a Bloom filter of a fixed size can represent a set with an arbitrarily large number of elements.
- Adding an element never fails. However, the false positive rate increases steadily as elements are added until all bits in the filter are set to 1, at which point all queries yield a positive result.
- Bloom filters never generate **false negative** result, i.e., telling you that a username doesn't exist when it actually exists.
- Deleting elements from filter is not possible because, if we delete a single element by clearing bits at indices generated by k hash functions, it might cause deletion of few other elements. Example – if we delete "geeks" (in given example below) by clearing bit at 1, 4 and 7, we might end up deleting "nerd" also Because bit at index 4 becomes 0 and bloom filter claims that "nerd" is not present.

Working of Bloom Filter

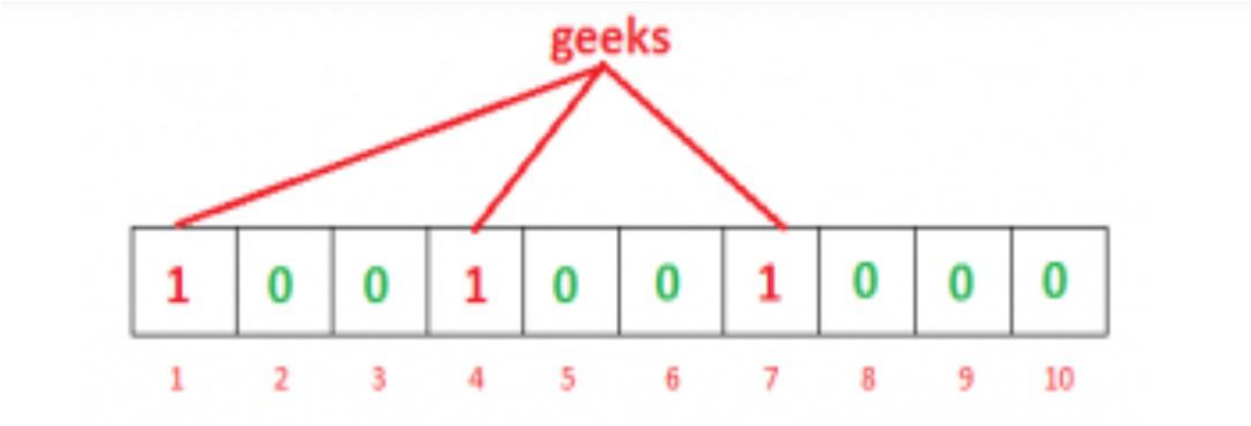
A empty bloom filter is a **bit array** of **m** bits, all set to zero, like this –



We need **k** number of **hash functions** to calculate the hashes for a given input. When we want to add an item in the filter, the bits at k indices h1(x), h2(x), ... hk(x) are set, where indices are calculated using hash functions.
Example – Suppose we want to enter "geeks" in the filter, we are using 3 hash functions and a bit array of length 10, all set to 0 initially. First we'll calculate the hashes as follows:

```
h1("geeks") % 10 = 1
h2("geeks") % 10 = 4
h3("geeks") % 10 = 7
```

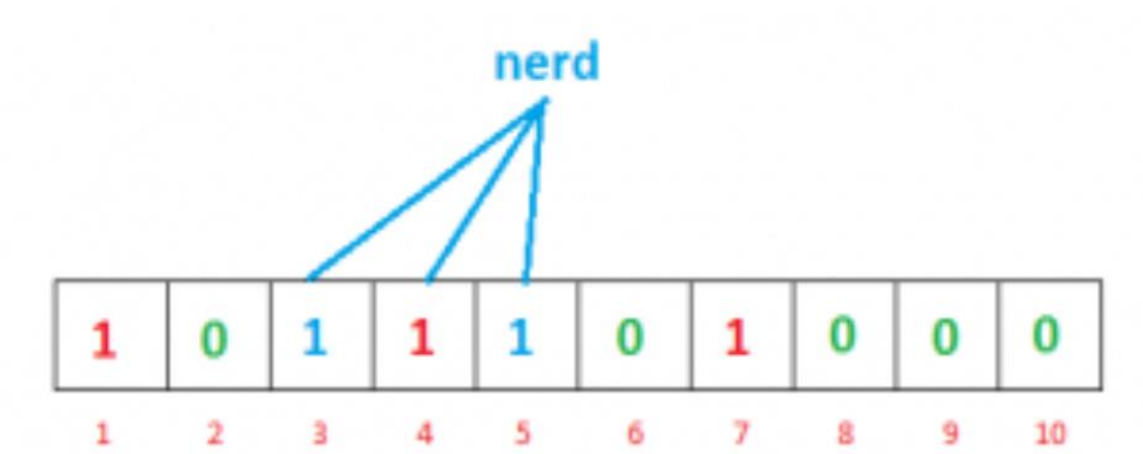
Note: These outputs are random for explanation only.
Now we will set the bits at indices 1, 4 and 7 to 1



Again we want to enter "nerd", similarly, we'll calculate hashes

```
h1("nerd") % 10 = 3
h2("nerd") % 10 = 5
h3("nerd") % 10 = 4
```

Set the bits at indices 3, 5 and 4 to 1



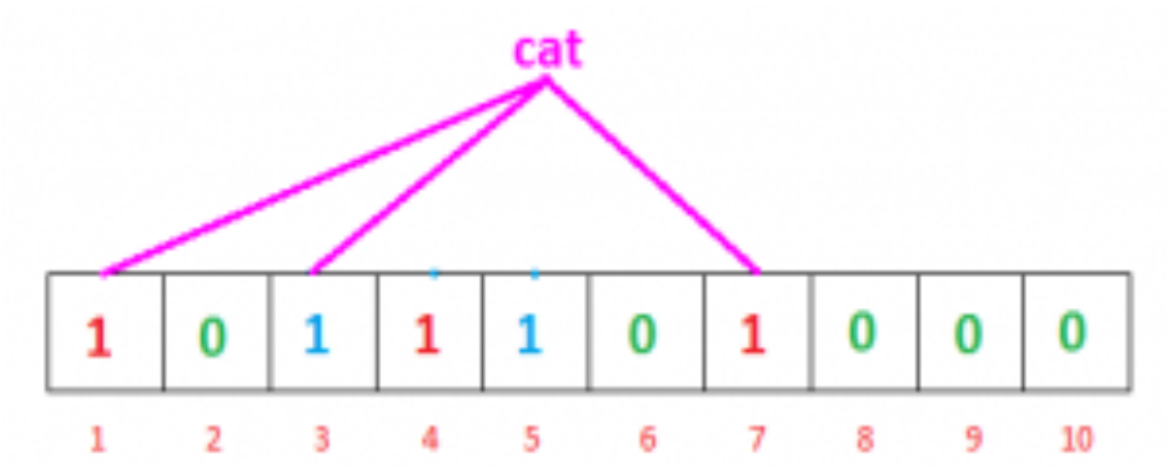
Now if we want to check "geeks" is present in filter or not. We'll do the same process but this time in reverse order. We calculate respective hashes using h1, h2 and h3 and check if all these indices are set to 1 in the bit array. If all the bits are set then we can say that "geeks" is **probably present**. If any of the bit at these indices are 0 then "geeks" is **definitely not present**.

False Positive in Bloom Filters

The question is why we said **"probably present"**, why this uncertainty. Let's understand this with an example. Suppose we want to check whether "cat" is present or not. We'll calculate hashes using h1, h2 and h3

```
h1("cat") % 10 = 1
h2("cat") % 10 = 3
h3("cat") % 10 = 7
```

If we check the bit array, bits at these indices are set to 1 but we know that "cat" was never added to the filter. Bit at index 1 and 7 was set when we added "geeks" and bit 3 was set we added "nerd".



So, because bits at calculated indices are already set by some other item, bloom filter erroneously claims that "cat" is present and generating a false positive result. Depending on the application, it could be huge downside or relatively okay.
We can control the probability of getting a false positive by controlling the size of the Bloom filter. More space means fewer false positives. If we want to decrease probability of false positive result, we have to use more number of hash functions and larger bit array. This would add latency in addition to the item and checking membership.

Operations that a Bloom Filter supports

- insert(x) : To insert an element in the Bloom Filter.
- lookup(x) : to check whether an element is already present in Bloom Filter with a positive false probability.

NOTE : We cannot delete an element in Bloom Filter.
Probability of False positivity: Let **m** be the size of bit array, k be the number of hash functions and **n** be the number of expected elements to be inserted in the filter, then the probability of false positive **p** can be calculated as:

$$P = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^{k}$$

Size of Bit Array: If expected number of elements **n** is known and desired false positive probability is **p** then the size of bit array **m** can be calculated as :

$$m = -\frac{n \ln P}{(ln2)^2}$$

Optimum number of hash functions: The number of hash functions **k** must be a positive integer. If **m** is size of bit array and **n** is number of elements to be inserted, then k can be calculated as :

$k = \frac{m}{n} ln2$

Space Efficiency

If we want to store large list of items in a set for purpose of set membership, we can store it in hashmap, tries or simple array or linked list. All these methods require storing item itself, which is not very memory efficient. For example, if we want to store "geeks" in hashmap we have to store actual string " geeks" as a key value pair {some_key : "geeks"}.
Bloom filters do not store the data item at all. As we have seen they use bit array which allow hash collision. Without hash collision, it would not be compact.

Choice of Hash Function

The hash function used in bloom filters should be independent and uniformly distributed. They should be fast as possible. Fast simple non cryptographic hashes which are independent enough include murmur, FNV series of hash functions and Jenkins hashes.
Generating hash is major operation in bloom filters. Cryptographic hash functions provide stability and guarantee but are expensive in calculation. With increase in number of hash functions k, bloom filter become slow. All though non-cryptographic hash functions do not provide guarantee but provide major performance improvement.

Basic implementation of Bloom Filter class in Python3. Save it as **bloomfilter.py**

```
# Python 3 program to build Bloom Filter
# Install mmh3 and bitarray 3rd party module first
# pip install mmh3
# pip install bitarray
import math
import mmh3
from bitarray import bitarray


class BloomFilter(object):

        '''
```

```
Class for Bloom filter, using murmur3 hash function
'''

def __init__(self, items_count, fp_prob):
        '''
        items_count : int
                Number of items expected to be stored in bloom filter
        fp_prob : float
                False Positive probability in decimal
        '''
        # False possible probability in decimal
        self.fp_prob = fp_prob

        # Size of bit array to use
        self.size = self.get_size(items_count, fp_prob)

        # number of hash functions to use
        self.hash_count = self.get_hash_count(self.size, items_count)

        # Bit array of given size
        self.bit_array = bitarray(self.size)

        # initialize all bits as 0
        self.bit_array.setall(0)

def add(self, item):
        '''
        Add an item in the filter
```

```
        '''
        digests = []
        for i in range(self.hash_count):

                # create digest for given item.
                # i work as seed to mmh3.hash() function
                # With different seed, digest created is different
                digest = mmh3.hash(item, i) % self.size
                digests.append(digest)

                # set the bit True in bit_array
                self.bit_array[digest] = True

def check(self, item):
        '''
        Check for existence of an item in filter
        '''
        for i in range(self.hash_count):
                digest = mmh3.hash(item, i) % self.size
                if self.bit_array[digest] == False:

                        # if any of bit is False then,its not present
                        # in filter
                        # else there is probability that it exist
                        return False
        return True

@classmethod
```

```
        def get_size(self, n, p):
                '''
                Return the size of bit array(m) to used using
                following formula
                m = -(n * lg(p)) / (lg(2)^2)
                n : int
                        number of items expected to be stored in filter
                p : float
                        False Positive probability in decimal
                '''
                m = -(n * math.log(p))/(math.log(2)**2)
                return int(m)

        @classmethod
        def get_hash_count(self, m, n):
                '''
                Return the hash function(k) to be used using
                following formula
                k = (m/n) * lg(2)

                m : int
                        size of bit array
                n : int
                        number of items expected to be stored in filter
                '''
                k = (m/n) * math.log(2)
                return int(k)
from bloomfilter import BloomFilter
```

```
from random import shuffle

n = 20 #no of items to add
p = 0.05 #false positive probability

bloomf = BloomFilter(n,p)
print("Size of bit array:{}".format(bloomf.size))
print("False positive Probability:{}".format(bloomf.fp_prob))
print("Number of hash functions:{}".format(bloomf.hash_count))

# words to be added
word_present = ['abound','abounds','abundance','abundant','accessible',
                'bloom','blossom','bolster','bonny','bonus','bonuses',
                'coherent','cohesive','colorful','comely','comfort',
                'gems','generosity','generous','generously','genial']

# word not added
word_absent = ['bluff','cheater','hate','war','humanity',
               'racism','hurt','nuke','gloomy','facebook',
               'geeksforgeeks','twitter']

for item in word_present:
        bloomf.add(item)

shuffle(word_present)
shuffle(word_absent)

test_words = word_present[:10] + word_absent
```

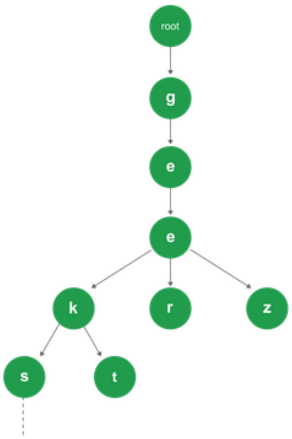
```
shuffle(test_words)
for word in test_words:
    if bloomf.check(word):
        if word in word_absent:
            print("'{}' is a false positive!".format(word))
        else:
            print("'{}' is probably present!".format(word))
    else:
        print("'{}' is definitely not present!".format(word))
```

Output

```
Size of bit array:124
False positive Probability:0.05
Number of hash functions:4
'war' is definitely not present!
'gloomy' is definitely not present!
'humanity' is definitely not present!
'abundant' is probably present!
'bloom' is probably present!
'coherent' is probably present!
'cohesive' is probably present!
'bluff' is definitely not present!
'bolster' is probably present!
'hate' is definitely not present!
'racism' is definitely not present!
'bonus' is probably present!
'abounds' is probably present!
'genial' is probably present!
'geeksforgeeks' is definitely not present!
'nuke' is definitely not present!
'hurt' is definitely not present!
```

```
'twitter' is a false positive!
'cheater' is definitely not present!
'generosity' is probably present!
'facebook' is definitely not present!
'abundance' is probably present!
```

Trie is an efficient information retrieval data structure. Using Trie, search complexities can be brought to optimal limit (key length). If we store keys in a binary search tree, a well balanced BST will need time proportional to **M * log N**, where M is the maximum string length and N is the number of keys in the tree. Using Trie, we can search the key in O(M) time. However, the penalty is on Trie storage requirements (Please refer to Applications of Trie for more details)

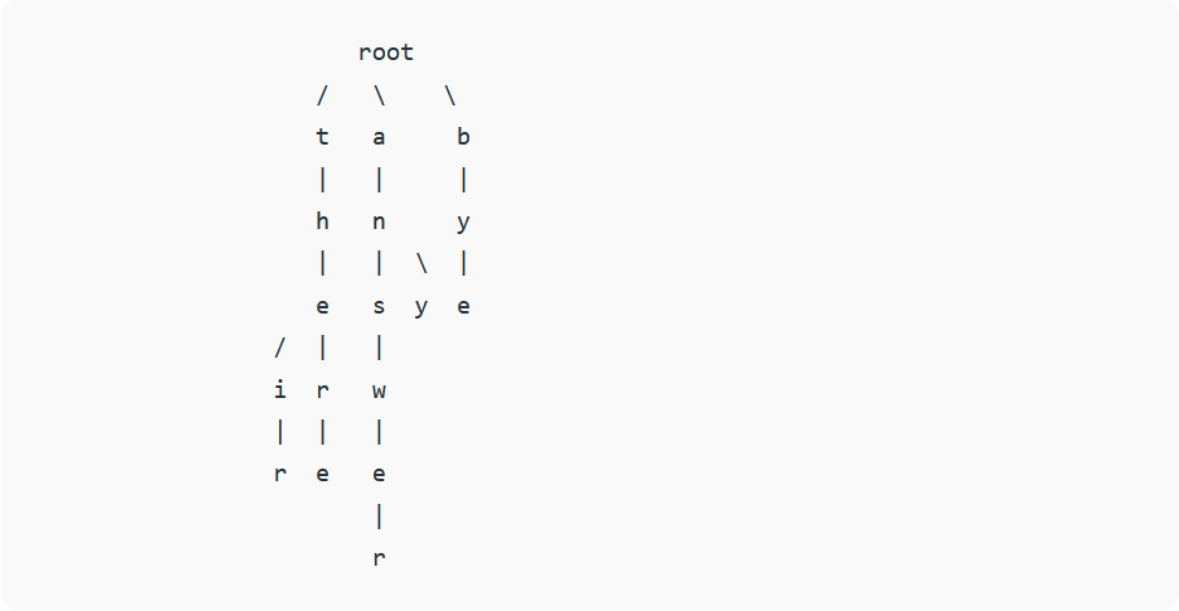


Every node of Trie consists of multiple branches. Each branch represents a possible character of keys. We need to mark the last node of every key as the end of the word node. A Trie node field *isEndOfWord* is used to distinguish the node as the end of the word node. A simple structure to represent nodes of the English alphabet can be as follows,

```
// Trie node
struct TrieNode
{
   struct TrieNode *children[ALPHABET_SIZE];
   // isEndOfWord is true if the node
   // represents end of a word
   bool isEndOfWord;
};
```

Inserting a key into Trie is a simple approach. Every character of the input key is inserted as an individual Trie node. Note that the *children* is an array of pointers (or references) to next level trie nodes. The key character acts as an index to the array *children*. If the input key is new or an extension of the existing key, we need to construct non-existing nodes of the key, and mark the end of the word for the last node. If the input key is a prefix of the existing key in Trie, we simply mark the last node of the key as the end of a word. The key length determines Trie depth.

Searching for a key is similar to an insert operation, however, we only compare the characters and move down. The search can terminate due to the end of a string or lack of key in the trie. In the former case, if the *isEndofWord* field of the last node is true, then the key exists in the trie. In the second case, the search terminates without examining all the characters of the key, since the key is not present in the trie.
The following picture explains the construction of trie using keys given in the example below,



In the picture, every character is of type *trie_node_t*. For example, the *root* is of type trie_node_t, and it's children *a*, *b* and *t* are filled, all other nodes of root will be NULL. Similarly, "a" at the next level is having only one child ("n"), all other children are NULL. The leaf nodes are in blue.

Insert and search costs **O(key_length)**, however, the memory requirements of Trie is **O(ALPHABET_SIZE * key_length * N)** where N is the number of keys in Trie. There are efficient representations of trie nodes (e.g. compressed trie, ternary search tree, etc.) to minimize the memory requirements of the trie.

 # Python program for insert and search

 # operation in a Trie

class TrieNode:

 # Trie node class

```
    def __init__(self):
        self.children = [None]*26

        # isEndOfWord is True if node represent the end of the word
        self.isEndOfWord = False

class Trie:

    # Trie data structure class
    def __init__(self):
        self.root = self.getNode()

    def getNode(self):

        # Returns new trie node (initialized to NULLs)
        return TrieNode()

    def _charToIndex(self,ch):

        # private helper function
        # Converts key current character into index
        # use only 'a' through 'z' and lower case

        return ord(ch)-ord('a')


    def insert(self,key):
```

```
        # If not present, inserts key into trie
        # If the key is prefix of trie node,
        # just marks leaf node
        pCrawl = self.root
        length = len(key)
        for level in range(length):
                index = self._charToIndex(key[level])

                # if current character is not present
                if not pCrawl.children[index]:
                        pCrawl.children[index] = self.getNode()
                pCrawl = pCrawl.children[index]

        # mark last node as leaf
        pCrawl.isEndOfWord = True

def search(self, key):

        # Search key in the trie
        # Returns true if key presents
        # in trie, else false
        pCrawl = self.root
        length = len(key)
        for level in range(length):
                index = self._charToIndex(key[level])
                if not pCrawl.children[index]:
                        return False
                pCrawl = pCrawl.children[index]
```

```
                return pCrawl.isEndOfWord

# driver function
def main():

        # Input keys (use only 'a' through 'z' and lower case)
        keys = ["the","a","there","anaswe","any",
                        "by","their"]
        output = ["Not present in trie",
                        "Present in trie"]

        # Trie object
        t = Trie()

        # Construct trie
        for key in keys:
                t.insert(key)

        # Search for different keys
        print("{} ---- {}".format("the",output[t.search("the")]))
        print("{} ---- {}".format("these",output[t.search("these")]))
        print("{} ---- {}".format("their",output[t.search("their")]))
        print("{} ---- {}".format("thaw",output[t.search("thaw")]))

if __name__ == '__main__':
        main()
```

 # This code is contributed by Atul Kumar

Output :

```
the --- Present in trie
these --- Not present in trie
their --- Present in trie
thaw --- Not present in trie
```