Module-III:

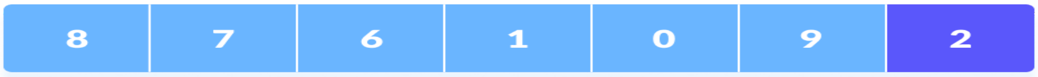
Sorting: quick sort, merge sort, and their algorithmic analysis. **Linked lists:** Single linked list, double linked list, circular linked list **Stacks:** Definition, operations: array implementation, linked list implementation. **Queues:** Definition, operations: array implementation, linked list implementation, and applications, Priority Queue. Double-Ended Queues.

Quicksort

Quicksort is an algorithm based on the divide and conquer approach in which the array is split into subarrays and these sub-arrays are recursively called to sort the elements.

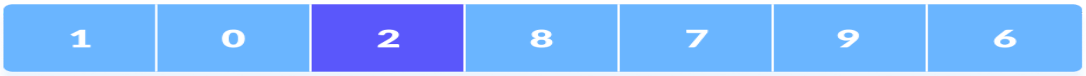
How QuickSort Works?

A pivot element is chosen from the array. You can choose any element from the array as the pivot element.
Here, we have taken the rightmost (ie. the last element) of the array as the pivot element.



Select a pivot element

The elements smaller than the pivot element are put on the left and the elements greater than the pivot element are put on the right.

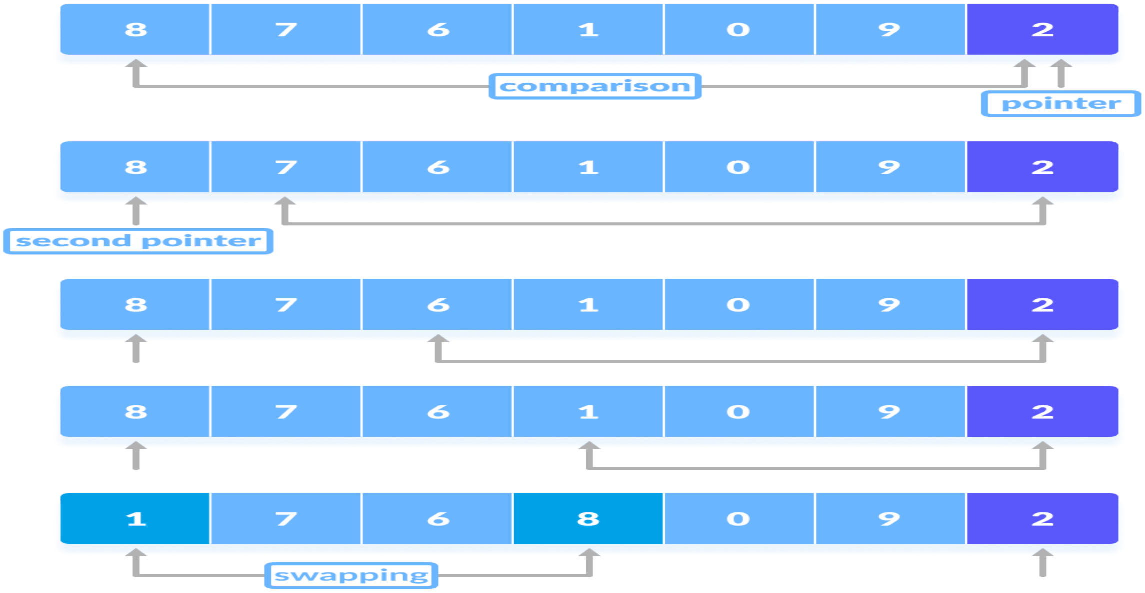


Put all the smaller elements on the left and greater on the right of the pivot element

The above arrangement is achieved by the following steps.

A pointer is fixed at the pivot element. The pivot element is compared with the elements beginning from the first index. If the element greater than the pivot element is reached, a second pointer is set for that element.

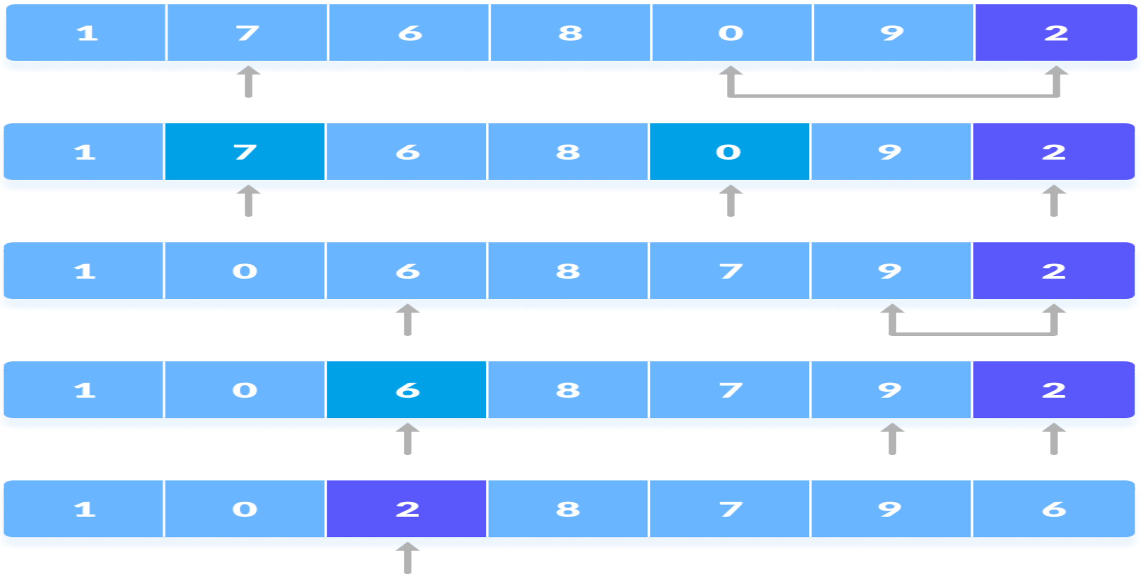
Now, the pivot element is compared with the other elements (a third pointer). If an element smaller than the pivot element is reached, the smaller element is swapped with the greater element found earlier.



Comparison of pivot element with other elements

The process goes on until the second last element is reached.

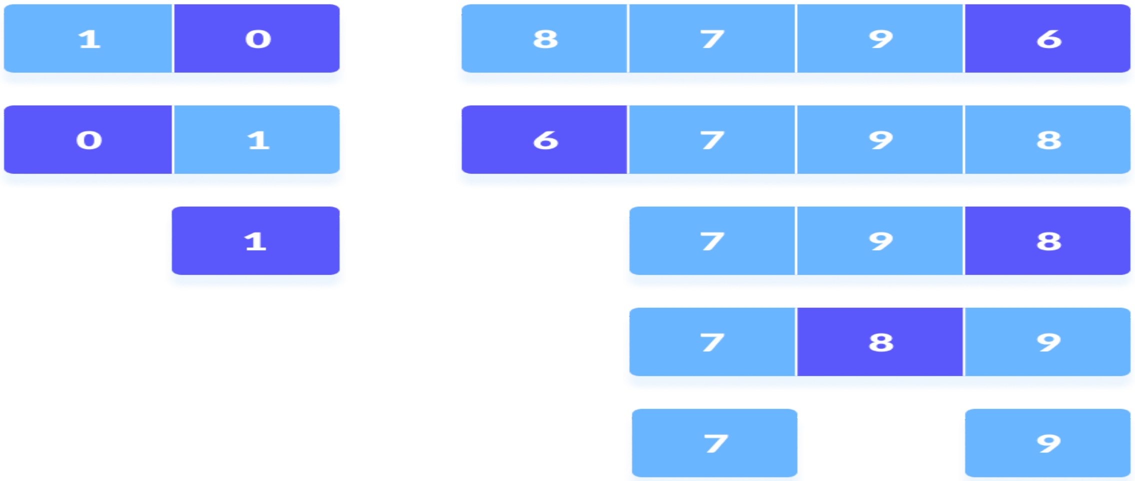
Finally, the pivot element is swapped with the second pointer.



Swap pivot element with the second pointer

Now the left and right subparts of this pivot element are taken for further processing in the steps below.

Pivot elements are again chosen for the left and the right sub-parts separately. Within these sub-parts, the pivot elements are placed in their right position. Then, step 2 is repeated.



Select pivot element in each half and put at the correct place using recursion

The sub-parts are again divided into smaller sub-parts until each subpart is formed of a single element.

At this point, the array is already sorted.

Quicksort uses recursion for sorting the sub-parts.

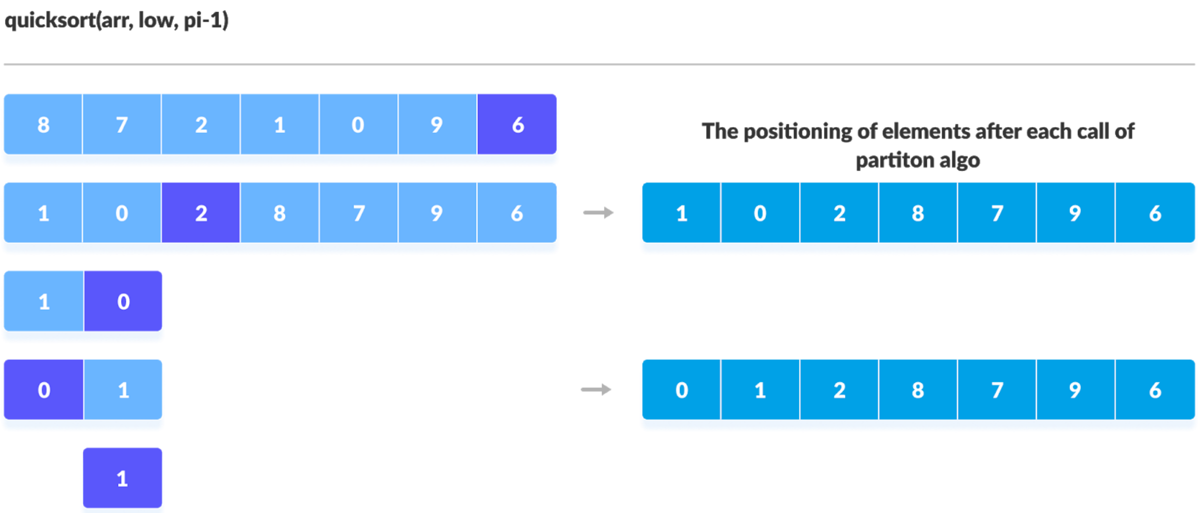
On the basis of the Divide and conquer approach, the quicksort algorithm can be explained as:

Divide: The array is divided into subparts taking pivot as the partitioning point. The elements smaller than the pivot are placed to the left of the pivot and the elements greater than the pivot are placed to the right.

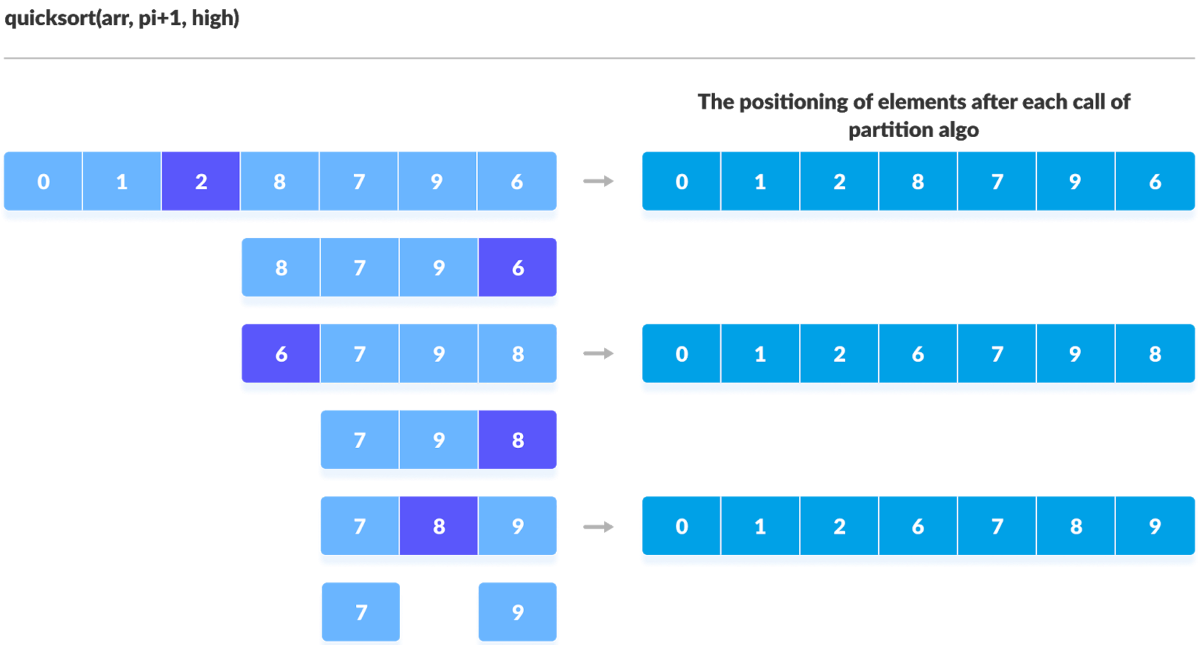
Conquer: The left and the right subparts are again partitioned using the by selecting pivot elements for them. This can be achieved by recursively passing the subparts into the algorithm.
Combine

This step does not play a significant role in quicksort. The array is already sorted at the end of the conquer step.

You can understand the working of quicksort with the help of the illustrations below.


quicksort(arr, low, pi-1)
The positioning of elements after each call of partiton algo
8 7 2 1 0 9 6
1 0 2 8 7 9 6
1 0 2 8 7 9 6
1 0
0 1
0 1 2 8 7 9 6
1

Sorting the elements on the left of pivot using recursion


quicksort(arr, pi+1, high)
The positioning of elements after each call of partition algo
0 1 2 8 7 9 6
0 1 2 8 7 9 6
8 7 9 6
6 7 9 8
0 1 2 6 7 9 8
7 9 8
7 8 9
0 1 2 6 7 8 9
7
9

Sorting the elements on the right of pivot using recursion

Quick Sort Algorithm

```
quickSort(array, leftmostIndex, rightmostIndex)
  if (leftmostIndex < rightmostIndex)
    pivotIndex <- partition(array,leftmostIndex, rightmostIndex)
    quickSort(array, leftmostIndex, pivotIndex)
    quickSort(array, pivotIndex + 1, rightmostIndex)

partition(array, leftmostIndex, rightmostIndex)
```

```
  set rightmostIndex as pivotIndex
  storeIndex <- leftmostIndex - 1
  for i <- leftmostIndex + 1 to rightmostIndex
  if element[i] < pivotElement
    swap element[i] and element[storeIndex]
    storeIndex++
  swap pivotElement and element[storeIndex+1]
return storeIndex + 1
```

Quicksort in Python

```python
# Function to partition the array on the basis of pivot element
def partition(array, low, high):

    # Select the pivot element
    pivot = array[high]
    i = low - 1

    # Put the elements smaller than pivot on the left and greater
    #than pivot on the right of pivot
    for j in range(low, high):
        if array[j] <= pivot:
            i = i + 1
            (array[i], array[j]) = (array[j], array[i])
    (array[i + 1], array[high]) = (array[high], array[i + 1])
    return i + 1

def quickSort(array, low, high):
    if low < high:
        # Select pivot position and put all the elements smaller
        # than pivot on left and greater than pivot on right
        pi = partition(array, low, high)

        # Sort the elements on the left of the pivot
        quickSort(array, low, pi - 1)

        # Sort the elements on the right of the pivot
        quickSort(array, pi + 1, high)

data = [8, 7, 2, 1, 0, 9, 6]
size = len(data)
quickSort(data, 0, size - 1)
print('Sorted Array in Ascending Order:')
print(data)
```

Quicksort Complexity
Time Complexities: Worst-Case Complexity [Big-O]: O(n2)

It occurs when the pivot element picked is either the greatest or the smallest element.
This condition leads to the case in which the pivot element lies at an extreme end of the sorted array. One sub-array is always empty and another sub-array contains n - 1 element. Thus, quicksort is called only on this sub-array.
However, the quick sort algorithm has better performance for scattered pivots.

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

It occurs when the pivot element is always the middle element or near to the middle element.

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

It occurs when the above conditions do not occur.

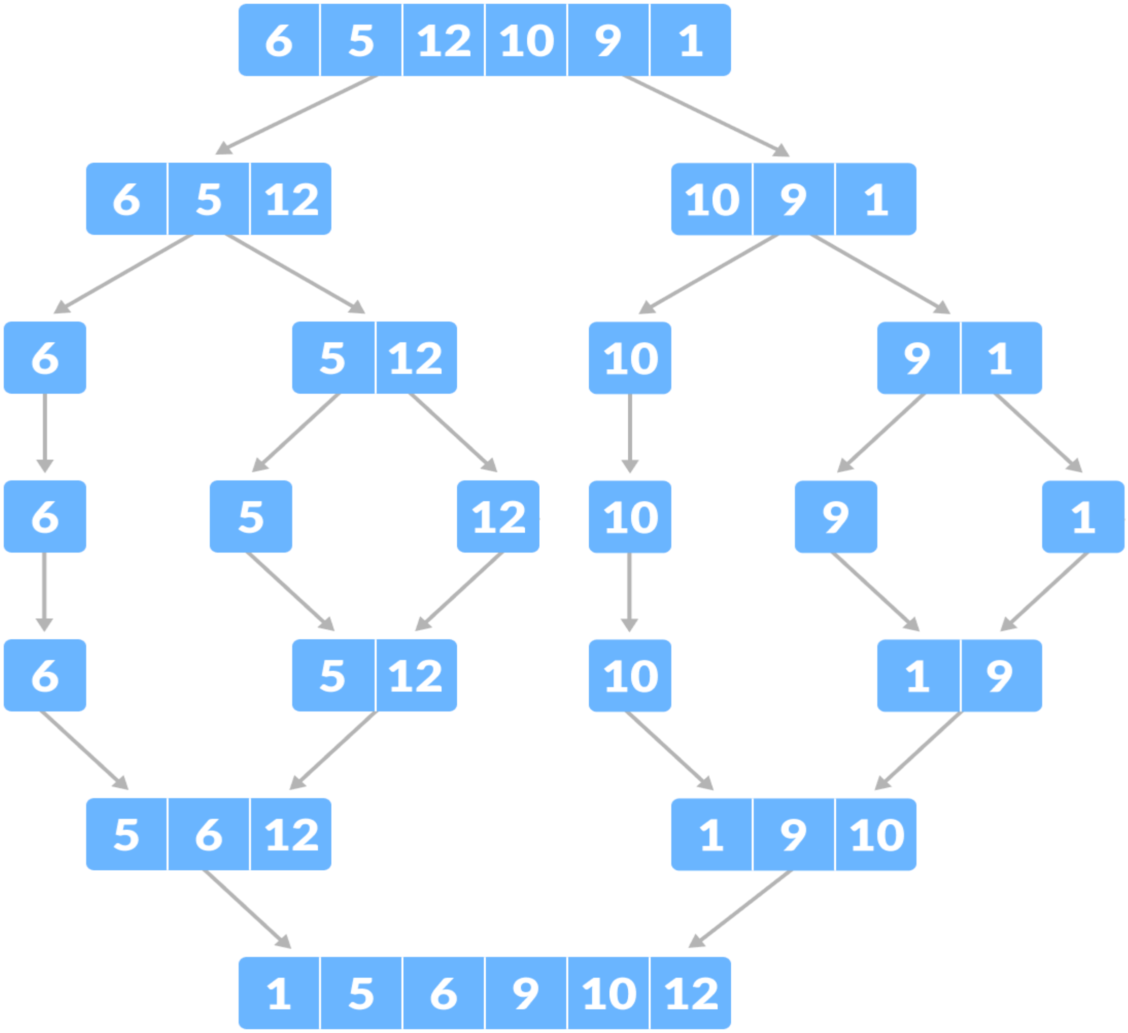
Space Complexity: The space complexity for quicksort is O(log n).

Quicksort Applications
Quicksort is implemented when

- the programming language is good for recursion
- time complexity matters
- space complexity matters

Merge Sort

Merge Sort is a kind of Divide and Conquer algorithm in computer programming. It is one of the most popular sorting algorithms and a great way to develop confidence in building recursive algorithms.



Merge Sort example

Divide and Conquer Strategy

Using the Divide and Conquer technique, we divide a problem into subproblems. When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem.

Suppose we had to sort an array A. A subproblem would be to sort a sub-section of this array starting at index p and ending at index r, denoted as A[p..r].

Divide

If q is the half-way point between p and r, then we can split the subarray A[p..r] into two arrays A[p..q] and A[q+1, r].

Conquer

In the conquer step, we try to sort both the subarrays A[p..q] and A[q+1, r]. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

Combine

When the conquer step reaches the base step and we get two sorted subarrays A[p..q] and A[q+1, r] for array A[p..r], we combine the results by creating a sorted array A[p..r] from two sorted subarrays A[p..q] and A[q+1, r].

The MergeSort Algorithm

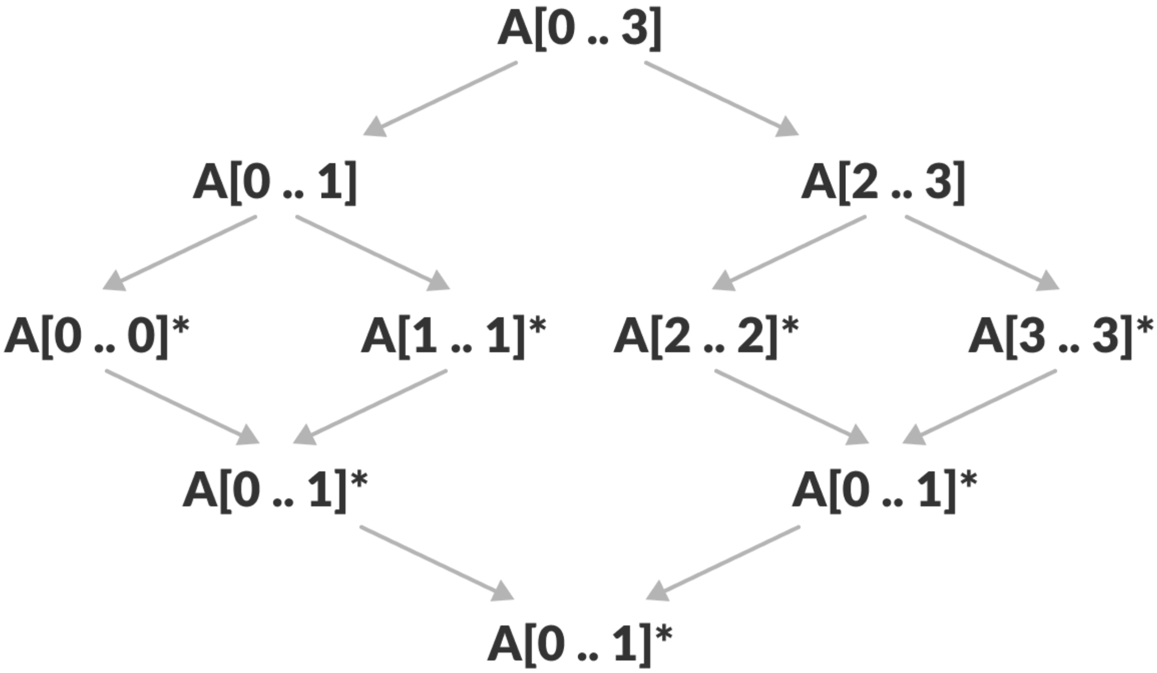
The MergeSort function repeatedly divides the array into two halves until we reach a stage where we try to perform MergeSort on a subarray of size 1 i.e. p == r.

After that, the merge function comes into play and combines the sorted arrays into larger arrays until the whole array is merged.

```
MergeSort(A, p, r):
  if p > r
    return
  q = (p+r)/2
  mergeSort(A, p, q)
  mergeSort(A, q+1, r)
  merge(A, p, q, r)
```

To sort an entire array, we need to call MergeSort(A, 0, length(A)-1).

As shown in the image below, the merge sort algorithm recursively divides the array into halves until we reach the base case of an array with 1 element. After that, the merge function picks up the sorted sub-arrays and merges them to gradually sort the entire array.



Merge sort in action

The merge Step of Merge Sort

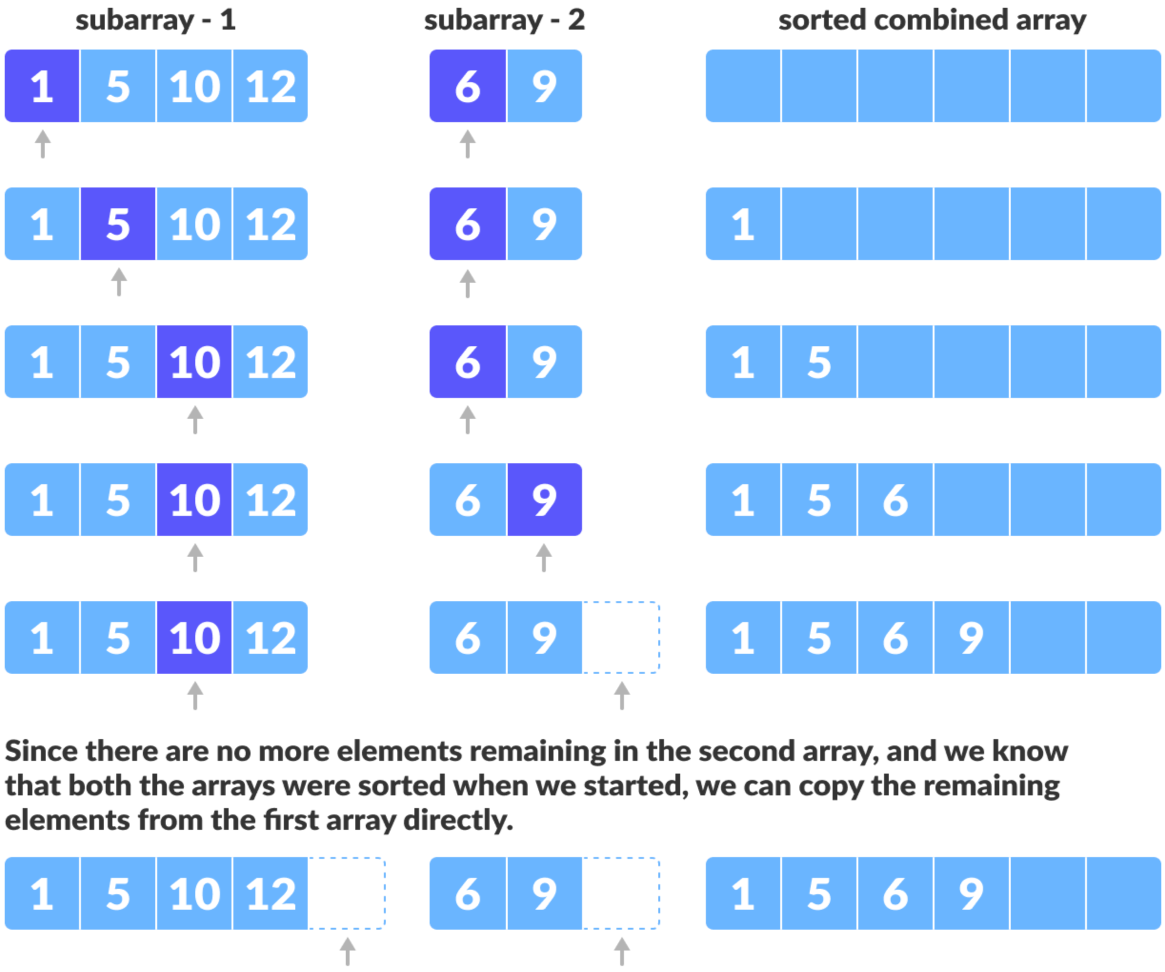
Every recursive algorithm is dependent on a base case and the ability to combine the results from base cases. Merge sort is no different. The most important part of the merge sort algorithm is, you guessed it, the merge step.

The merge step is the solution to the simple problem of merging two sorted lists(arrays) to build one large sorted list(array).

The algorithm maintains three-pointers, one for each of the two arrays and one for maintaining the current index of the final sorted array.

Have we reached the end of any of the arrays?

```
No:
    Compare current elements of both arrays
    Copy smaller element into a sorted array
    Move pointer of an element containing smaller element

Yes:
    Copy all remaining elements of a non-empty array
```



Since there are no more elements remaining in the second array, and we know that both the arrays were sorted when we started, we can copy the remaining elements from the first array directly.

Merge step

Writing the Code for Merge Algorithm

A noticeable difference between the merging step we described above and the one we use for merge sort is that we only perform the merge function on consecutive sub-arrays.

This is why we only need the array, the first position, the last index of the first subarray(we can calculate the first index of the second subarray), and the last index of the second subarray.

Our task is to merge two subarrays A[p..q] and A[q+1..r] to create a sorted array A[p..r]. So the inputs to the function are A, p, q and r

The merge function works as follows:

- Create copies of the subarrays L ← A[p..q] and M ← A[q+1..r].
- Create three-pointers i, j and k
- i maintains a current index of L, starting at 1
- j maintains a current index of M, starting at 1
- k maintains the current index of A[p..q], starting at p.

Until we reach the end of either L or M, pick the larger among the elements from L and M and place them in the correct position at A[p..q]

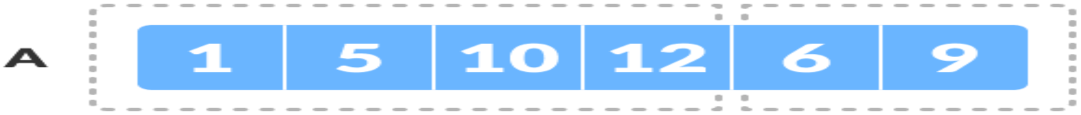
When we run out of elements in either L or M, pick up the remaining elements and put in A[p..q]

In code, this would look like:

```
// Merge two subarrays L and M into arr
void merge(int arr[], int p, int q, int r) {

    // Create L ← A[p..q] and M ← A[q+1..r]
    int n1 = q - p + 1;
    int n2 = r - q;

    int L[n1], M[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[p + i];
    for (int j = 0; j < n2; j++)
        M[j] = arr[q + 1 + j];

    // Maintain current index of sub-arrays and main array
    int i, j, k;
    i = 0;
    j = 0;
    k = p;

    // Until we reach either end of either L or M, pick larger among
    // elements L and M and place them in the correct position at A[p..r]
    while (i < n1 && j < n2) {
        if (L[i] <= M[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = M[j];
            j++;
        }
        k++;
    }

    // When we run out of elements in either L or M,
    // pick up the remaining elements and put in A[p..r]
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = M[j];
        j++;
        k++;
    }
}
```

Merge( ) Function Explained Step-By-Step

A lot is happening in this function, so let's take an example to see how this would work.
As usual, a picture speaks a thousand words.



Merging two consecutive subarrays of array

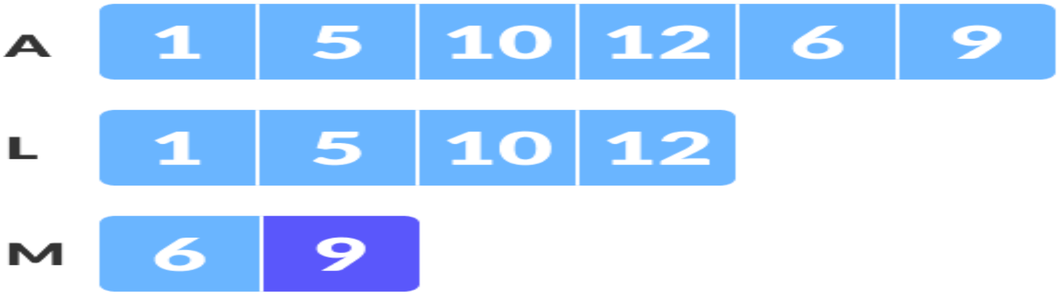
The array A[0..5] contains two sorted subarrays A[0..3] and A[4..5]. Let us see how the merge function will merge the two arrays.

```
void merge(int arr[], int p, int q, int r) {
// Here, p = 0, q = 4, r = 6 (size of array)
```

Step 1: Create duplicate copies of sub-arrays to be sorted

```
  // Create L ← A[p..q] and M ← A[q+1..r]
   int n1 = q - p + 1 = 3 - 0 + 1 = 4;
```

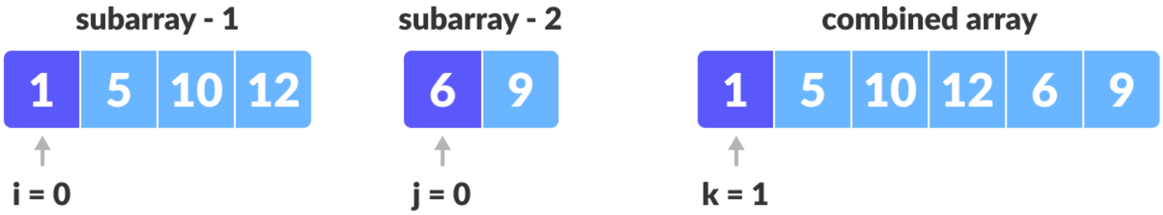
```
int n2 = r - q = 5 - 3 = 2;

int L[4], M[2];

for (int i = 0; i < 4; i++)
   L[i] = arr[p + i];
   // L[0,1,2,3] = A[0,1,2,3] = [1,5,10,12]

for (int j = 0; j < 2; j++)
   M[j] = arr[q + 1 + j];
   // M[0,1,2,3] = A[4,5] = [6,9]
```



Create copies of subarrays for merging

Step 2: Maintain a current index of sub-arrays and the main array

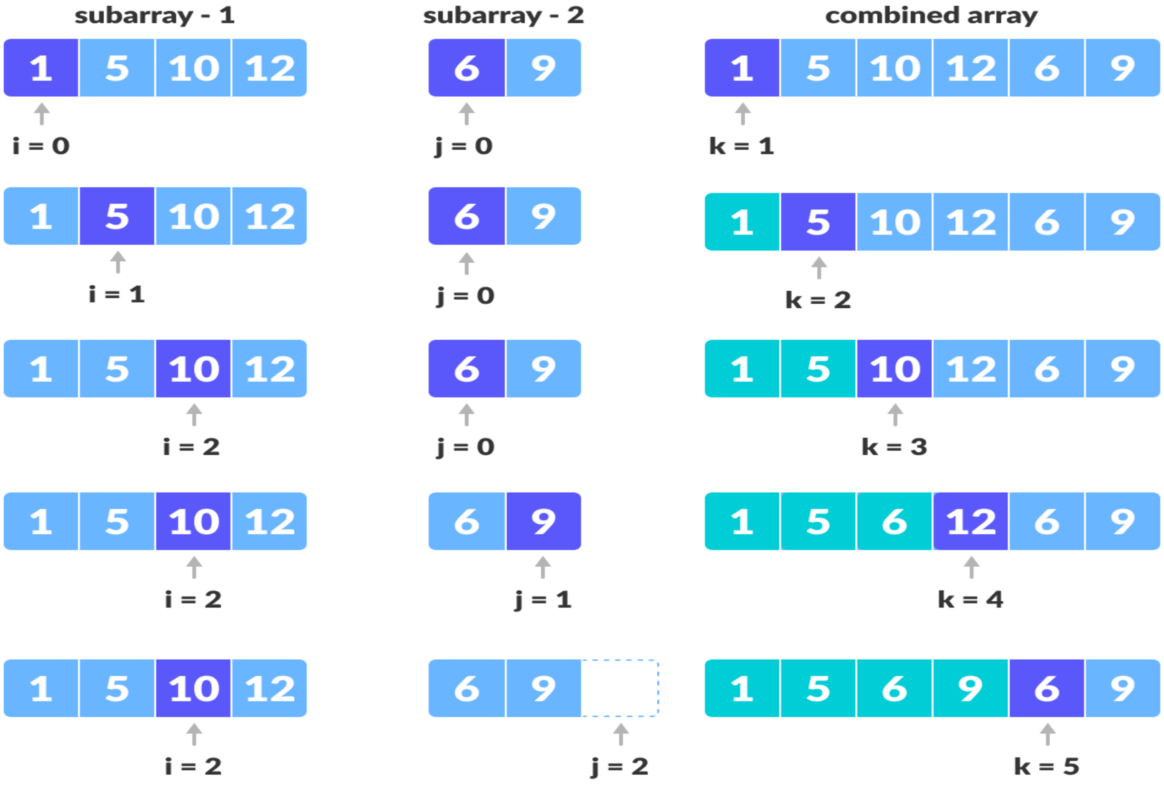
```
int i, j, k;
i = 0;
j = 0;
k = p;
```



Maintain indices of copies of sub array and main array

Step 3: Until we reach the end of either L or M, pick larger among elements L and *M and place them in the correct position at A[p..r]*

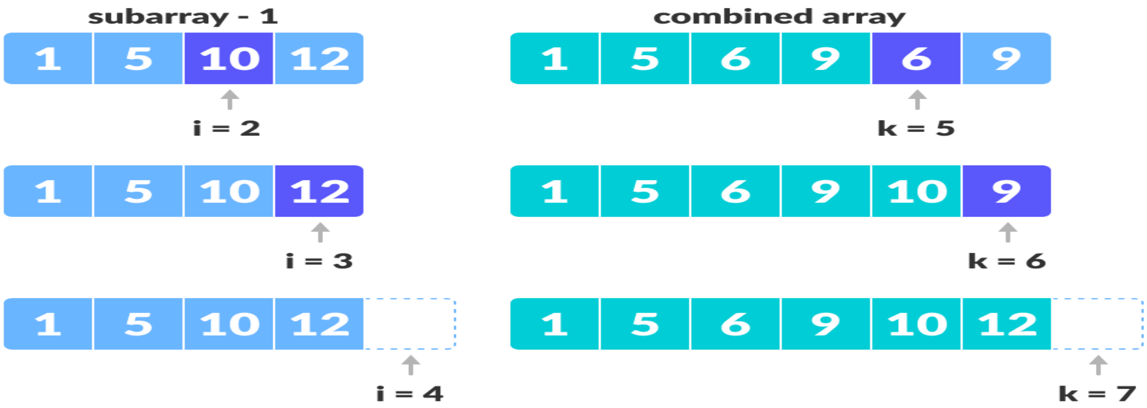
```
  while (i < n1 && j < n2) {
     if (L[i] <= M[j]) {
        arr[k] = L[i]; i++;
     }
     else {
        arr[k] = M[j];
        j++;
     }
     k++;
 }
```



Comparing individual elements of sorted subarrays until we reach end of one

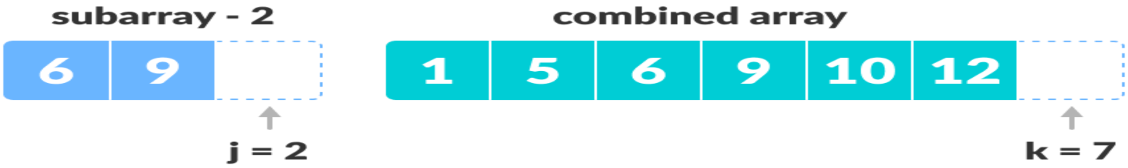
Step 4: When we run out of elements in either L or M, pick up the remaining elements and put in A[p..r]

```
  // We exited the earlier loop because j < n2 doesn't hold
   while (i < n1)
   {
       arr[k] = L[i];
       i++;
       k++;
   }
```



Copy the remaining elements from the first array to the main subarray

```
// We exited the earlier loop because i < n1 doesn't hold
   while (j < n2)
   {
      arr[k] = M[j];
      j++;
      k++;
   }
}
```



Copy remaining elements of the second array to the main subarray

This step would have been needed if the size of M was greater than L.
At the end of the merge function, the subarray A[p..r] is sorted.

```
# MergeSort in Python
def mergeSort(array):
   if len(array) > 1:

      #  r is the point where the array is divided into two subarrays
      r = len(array)//2
      L = array[:r]
      M = array[r:]
```

```
        # Sort the two halves
        mergeSort(L)
        mergeSort(M)

        i = j = k = 0

        # Until we reach either end of either L or M, pick larger among
        # elements L and M and place them in the correct position at A[p..r]
        while i < len(L) and j < len(M):
            if L[i] < M[j]:
                array[k] = L[i]
                i += 1
            else:
                array[k] = M[j]
                j += 1
            k += 1

        # When we run out of elements in either L or M,
        # pick up the remaining elements and put in A[p..r]
        while i < len(L):
            array[k] = L[i]
            i += 1
            k += 1

        while j < len(M):
            array[k] = M[j]
            j += 1
            k += 1


# Print the array
def printList(array):
    for i in range(len(array)):
        print(array[i], end=" ")
    print()

# Driver program
if __name__ == '__main__':
    array = [6, 5, 12, 10, 9, 1]

    mergeSort(array)
    print("Sorted array is: ")
    printList(array)
```

Merge Sort Complexity
Time Complexity
Best Case Complexity: O(n*log n)
Worst Case Complexity: O(n*log n)
Average Case Complexity: O(n*log n)

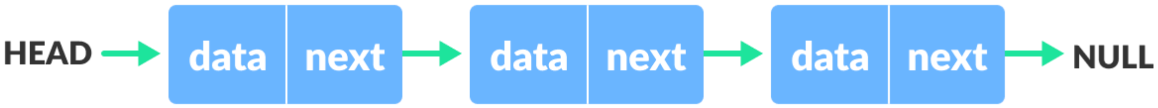
Space Complexity
The space complexity of merge sort is O(n).

Merge Sort Applications

- Inversion count problem
- External sorting
- E-commerce applications

Linked List

A linked list data structure includes a series of connected nodes. Here, each node store the data and the address of the next node. For example,



Linked in Data Structure

You have to start somewhere, so we give the address of the first node a special name called HEAD.

Also, the last node in the linked list can be identified because its next portion points to NULL.

You might have played the game Treasure Hunt, where each clue includes information about the next clue. That is how the linked list operates.

Representation of Linked List

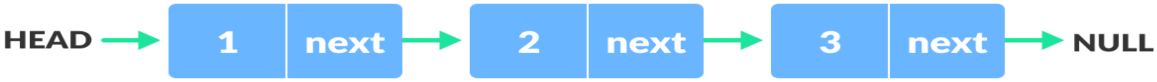
Let's see how each node of the LinkedList is represented. Each node consists:

A data item

An address of another node

```
class Node:
    # Creating a node
    def __init__(self, item):
        self.item = item
        self.next = None
```

In just a few steps, we have created a simple linked list with three nodes.



Linked List Representation

The power of LinkedList comes from the ability to break the chain and rejoin it. E.g. if you wanted to put an element 4 between 1 and 2, the steps would be:

Create a new struct node and allocate memory to it.

Add its data value as 4

Point its next pointer to the struct node containing 2 as the data value Change the next pointer of "1" to the node we just created. Doing something similar in an array would have required shifting the positions of all the subsequent elements.

Linked List Utility

Lists are one of the most popular and efficient data structures, with implementation in Python.

Apart from that, linked lists are a great way to learn how pointers work. By practicing how to manipulate linked lists, you can prepare yourself to learn more advanced data structures like graphs and trees.

```
# Linked list implementation in Python
class Node:
    # Creating a node
    def __init__(self, item):
        self.item = item
        self.next = None

class LinkedList:

    def __init__(self):
        self.head = None

if __name__ == '__main__':

    linked_list = LinkedList()

    # Assign item values
```

```
linked_list.head = Node(1)
second = Node(2)
third = Node(3)

# Connect nodes
linked_list.head.next = second
second.next = third

# Print the linked list item
while linked_list.head != None:
   print(linked_list.head.item, end=" ")
   linked_list.head = linked_list.head.next
```

Linked List Complexity

Time Complexity

	Worst case	Average Case
Search	O(n)	O(n)
Insert	O(1)	O(1)
Deletion	O(1)	O(1)

Space Complexity: O(n)

Linked List Applications

- Dynamic memory allocation
- Implemented in stack and queue
- In undoing functionality of the software
- Hash tables, Graphs
- Linked List Operations: Traverse, Insert and Delete

Traversing a Linked List

Displaying the contents of a linked list is very simple. We keep moving the temp node to the next one and display its contents.

When the temp is NULL, we know that we have reached the end of the linked list so we get out of the while loop.

```
def printList(self):
    temp_node = self.head
    while (temp_node):
        print(str(temp_node.item) + " ", end="")
        temp_node = temp_node.next
```

Insertion Elements to a Linked List

You can add elements to either the beginning, middle or end of the linked list.

Add to the beginning

Allocate memory for the new node

Store data

Change next of new node to point to head

Change head to point to the recently created node

```
def insertAtBeginning(self, data):
    new_node = Node(data)
    new_node.next = self.head
    self.head = new_node
```

Add to the End

Allocate memory for a new node

Store data

Traverse to the last node

Change next of the last node to the recently created node

```
def insertAtEnd(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        return
    last = self.head
    while (last.next):
        last = last.next
    last.next = new_node
```

Add to the Middle

Allocate memory and store data for new node

Traverse to node just before the required position of new node

Change the next pointers to include new node in between

```
def insertAfter(self, node, data):
        if node is None:
            print("The given previous node must in LinkedList.")
            return
        new_node = Node(data)
        new_node.next = node.next
        node.next = new_node
```

Removing an Item from a Linked List

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

```
def deleteNode(self, position):
        if self.head == None:
            return
        temp_node = self.head
        if position == 0:
            self.head = temp_node.next
            temp_node = None
            return
        # Find the key to be deleted
        for i in range(position - 1):
            temp_node = temp_node.next
            if temp_node is None:
                break
        # If the key is not present
        if temp_node is None:
            return
        if temp_node.next is None:
            return
        next = temp_node.next.next
        temp_node.next = None
        temp_node.next = next
```

Implementing Linked List Operations

# Linked list operations in Python

```
# Create a node
class Node:
    def __init__(self, item):
        self.item = item
        self.next = None
```

```
class LinkedList:

    def __init__(self):
        self.head = None

    # Insert at the beginning
    def insertAtBeginning(self, data):
        new_node = Node(data)

        new_node.next = self.head
        self.head = new_node

    # Insert after a node
    def insertAfter(self, node, data):

        if node is None:
            print("The given previous node must inLinkedList.")
            return

        new_node = Node(data)
        new_node.next = node.next
        node.next = new_node

    # Insert at the end
    def insertAtEnd(self, data):
        new_node = Node(data)

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

        last = self.head
        while (last.next):
            last = last.next

        last.next = new_node

    # Deleting a node
    def deleteNode(self, position):

        if self.head == None:
            return
        temp_node = self.head

        if position == 0:
            self.head = temp_node.next
            temp_node = None
            return

        # Find the key to be deleted
        for i in range(position - 1):
            temp_node = temp_node.next
            if temp_node is None:
                break

        # If the key is not present
        if temp_node is None:
            return
        if temp_node.next is None:
            return
        next = temp_node.next.next
        temp_node.next = None
        temp_node.next = next

    def printList(self):
        temp_node = self.head
        while (temp_node):
            print(str(temp_node.item) + " ", end="")
            temp_node = temp_node.next

if __name__ == '__main__':

    llist = LinkedList()
    llist.insertAtEnd(1)
    llist.insertAtBeginning(2)
    llist.insertAtBeginning(3)
    llist.insertAtEnd(4)
    llist.insertAfter(llist.head.next, 5)
    print('Linked list:')
    llist.printList()

    print("\nAfter deleting an element:")
    llist.deleteNode(3)
    llist.printList()
```

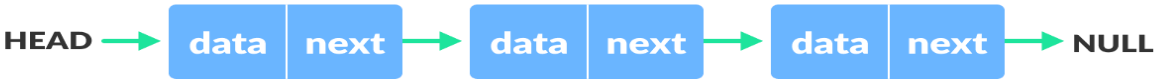
Types of Linked List

There are three common types of Linked List.

- Singly Linked List
- Doubly Linked List
- Circular Linked List

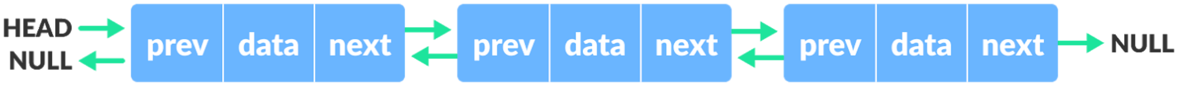
Singly Linked List

It is the most common. Each node has data and a pointer to the next node.



Doubly Linked List

We add a pointer to the previous node in a doubly-linked list. Thus, we can go in either direction: forward or backward.



```
# Node of a doubly linked list
class Node:
    def __init__(self, next=None, prev=None, data=None):
        self.next = next # reference to next node in DLL
        self.prev = prev # reference to previous node in DLL
        self.data = data
```

The following are advantages/disadvantages of a doubly-linked list over a singly linked list.

Advantages over a singly linked list

1) A DLL can be traversed in both forward and backward directions.
2) The delete operation in DLL is more efficient if a pointer to the node to be deleted is given.
3) We can quickly insert a new node before a given node.

In singly linked list, to delete a node, a pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using the previous pointer.

Disadvantages over a singly linked list

1) Every node of DLL requires extra space for a previous pointer. It is possible to implement DLL with a single pointer though (See this and this).

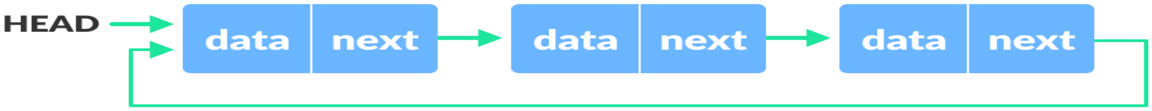
2) All operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with next pointers. For example, in the following functions for insertions at different positions, we need 1 or 2 extra steps to set the previous pointer.

Example:

```
class _Node:
    __slots__ = '_element', '_next', '_prev'

    def __init__(self, element, next, prev):
        self._element = element
        self._next = next
        self._prev = prev

class DoublyLinkedList:
    def __init__(self):
        self._head = None
        self._tail = None
        self._size = 0

    def __len__(self):
        return self._size

    def isempty(self):
        return self._size == 0

    def addlast(self, e):
        newest = _Node(e, None, None)
        if self.isempty():
            self._head = newest
            self._tail = newest
        else:
            self._tail._next = newest
            newest._prev = self._tail
            self._tail = newest
        self._size += 1

    def display(self):
        p = self._head
        while p:
            print(p._element,end=' --> ')
            p = p._next
        print()

    def displayrev(self):
        p = self._tail
        while p:
            print(p._element,end=' <-- ')
            p = p._prev
        print()


L = DoublyLinkedList()
L.addlast(7)
L.addlast(4)
L.addlast(12)
L.addlast(8)
L.addlast(3)
L.display()
print('Size:',len(L))
L.displayrev()
```

Circular Linked List

A circular linked list is a variation of a linked list in which the last element is linked to the first element. This forms a circular loop.



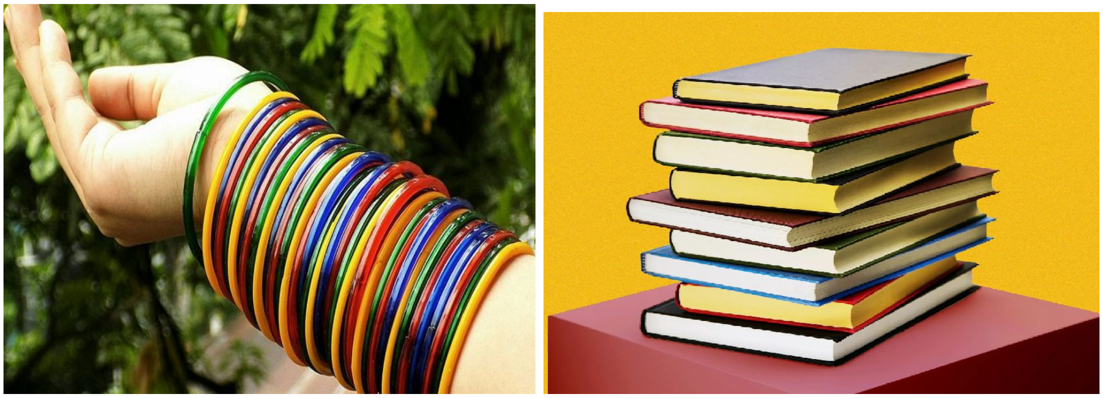
A circular linked list can be either singly linked or doubly linked.for a singly linked list, the next pointer of the last item points to the first item In the doubly linked list, the prev pointer of the first item points to the last item as well.

Example:

```
class _Node:
    __slots__ = '_element', '_next'

    def __init__(self, element, next):
        self._element = element
        self._next = next

class CircularLinkedList:
    def __init__(self):
        self._head = None
        self._tail = None
        self._size = 0

    def __len__(self):
        return self._size

    def isempty(self):
        return self._size == 0

    def addlast(self, e):
        newest = _Node(e, None)
        if self.isempty():
            newest._next = newest
            self._head = newest
        else:
            newest._next = self._tail._next
            self._tail._next = newest
        self._tail = newest
        self._size += 1

def addfirst(self,e):
        newest = _Node(e, None)
        if self.isempty():
            newest._next = newest
            self._head = newest
            self._tail = newest
        else:
            self._tail._next = newest
            newest._next = self._head
            self._head = newest
        self._size += 1

    def addany(self, e, position):
        newest = _Node(e, None)
        p = self._head
        i = 1
        while i < position - 1:
            p = p._next
            i = i + 1
        newest._next = p._next
        p._next = newest
        self._size += 1

    def display(self):
        p = self._head
        i = 0
        while i < len(self):
            print(p._element,end='-->')
            p = p._next
            i += 1
        print()

    def search(self,key):
        p = self._head
        index = 0
        while index < len(self):
            if p._element == key:
                return index
            p = p._next
            index = index + 1
        return -1

C = CircularLinkedList()
C.addlast(7)
C.addlast(4)
C.addlast(12)
C.display()
print('Size:',len(C))
C.addlast(8)
C.addlast(3)
C.addfirst(20)
C.addfirst(15)
C.addany(3, 30)
C.display()
print('Size:',len(C))
```

Stack

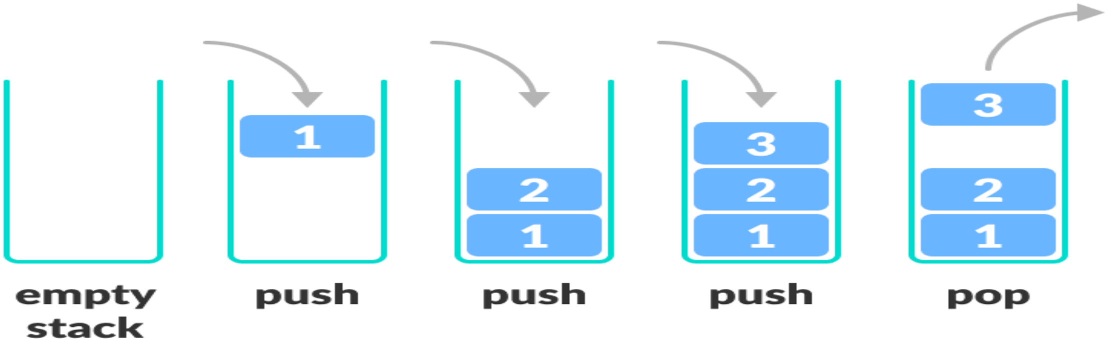
A stack is a useful data structure in programming. A stack is also called Last In First Out - the last item that was placed is the first item to go out.



Example of Stack

LIFO Principle of Stack

In programming terms, putting an item on top of the stack is called "push" and removing an item is called "pop".



Stack Push and Pop Operations

In the above image, although item 2 was kept last, it was removed first - so it follows the Last In First Out(LIFO) principle.

Basic Operations of Stack

A stack is an object or more specifically an abstract data structure(ADT) that allows the following operations:

Push: Add an element to the top of a stack
Pop: Remove an element from the top of a stack
IsEmpty: Check if the stack is empty
IsFull: Check if the stack is full
Peek: Get the value of the top element without removing it

Working of Stack Data Structure

The operations work as follows:

A pointer called TOP is used to keep track of the top element in the stack.

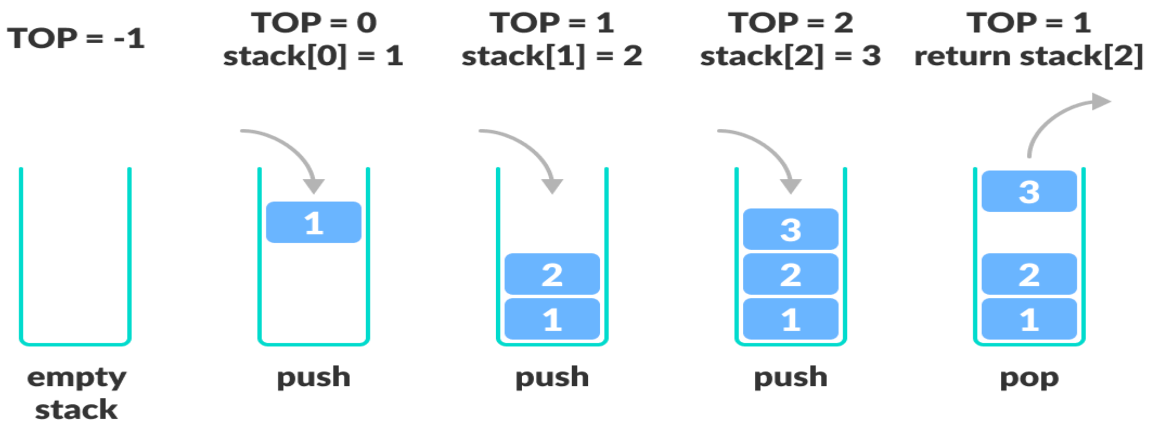
When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing TOP == -1.

On pushing an element, we increase the value of TOP and place the new element in the position pointed to by TOP.

On popping an element, we return the element pointed to by TOP and reduce its value.

Before pushing, we check if the stack is already full

Before popping, we check if the stack is already empty



Stack implementation is using arrays

```
# Creating a stack
def create_stack():
    stack = []
    return stack

# Creating an empty stack
def check_empty(stack):
    return len(stack) == 0

# Adding items into the stack
def push(stack, item):
    stack.append(item)
    print("pushed item: " + item)
```

```
# Removing an element from the stack
def pop(stack):
    if (check_empty(stack)):
        return "stack is empty"

    return stack.pop()

stack = create_stack()
push(stack, str(1))
push(stack, str(2))
push(stack, str(3))
push(stack, str(4))
print("popped item: " + pop(stack))
print("stack after popping an element: " + str(stack))
```

Stack Time Complexity

For the array-based implementation of a stack, the push and pop operations take constant time i.e. O(1) because there is only a movement of the pointer in both the cases.

Applications of Stack Data Structure

Although stack is a simple data structure to implement, it is very powerful. The most common uses of a stack are:

To reverse a word - Put all the letters in a stack and pop them out. Because of the LIFO order of stack, you will get the letters in reverse order.

In compilers - Compilers use the stack to calculate the value of expressions like 2 + 4 / 5 * (7 - 9) by converting the expression to prefix or postfix form.

In browsers - The back button in a browser saves all the URLs you have visited previously in a stack. Each time you visit a new page, it is added on top of the stack. When you press the back button, the current URL is removed from the stack and the previous URL is accessed.

Implementation using 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.

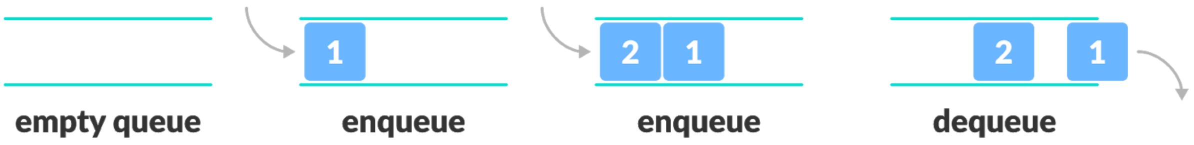
peek() – Return the top item in the stack. If the stack is empty, raise an exception.
push(value) – Push a value into the head of the stack.
pop() – Remove and return a value in the head of the stack. If the stack is empty, raise an exception.

```
 # 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}")
```

Queue

A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.

Queue follows the First In First Out(FIFO) rule - the item that goes in first is the item that comes out first too.



FIFO Representation of Queue

In the above image, since 1 was kept in the queue before 2, it was the first to be removed from the queue as well. It follows the FIFO rule.

Example

In programming terms, putting an item in the queue is called an "enqueue" and removing an item from the queue is called "dequeue".

Basic Operations of Queue

A queue is an object or more specifically an abstract data structure(ADT) that allows the following operations:

Enqueue: Add an element to the end of the queue
Dequeue: Remove an element from the front of the queue
IsEmpty: Check if the queue is empty
IsFull: Check if the queue is full
Peek: Get the value of the front of the queue without removing it

Working of Queue

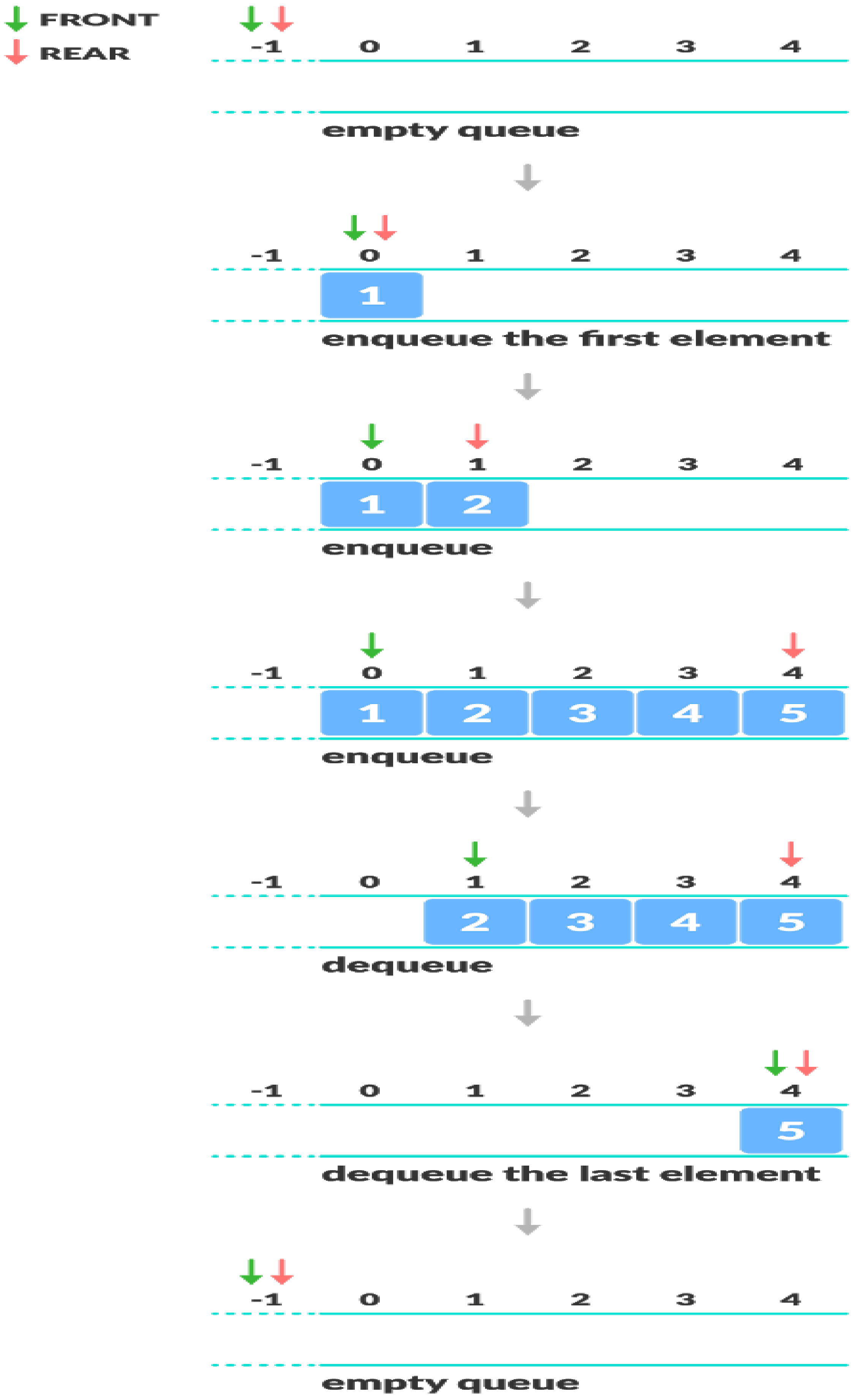
Queue operations work as follows: Two pointers FRONT and REAR
FRONT track the first element of the queue
REAR track the last elements of the queue
initially, set value of FRONT and REAR to -1

Enqueue Operation

check if the queue is full
for the first element, set value of FRONT to 0
increase the REAR index by 1
add the new element in the position pointed to by REAR

Dequeue Operation

check if the queue is empty
return the value pointed by FRONT
increase the FRONT index by 1
for the last element, reset the values of FRONT and REAR to -1


FRONT
REAR
-1 0 1 2 3 4
empty queue
-1 0 1 2 3 4
1
enqueue the first element
-1 0 1 2 3 4
1 2
enqueue
-1 0 1 2 3 4
1 2 3 4 5
enqueue
-1 0 1 2 3 4
2 3 4 5
dequeue
-1 0 1 2 3 4
5
dequeue the last element
-1 0 1 2 3 4
empty queue

Enqueue and Dequeue Operations

queue implementation is using arrays

# 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()
```

Queue – Linked List Implementation

In a Queue data structure, we maintain two pointers, front and rear. The front points the first item of the queue and the rear points to the last item.

enQueue() This operation adds a new node after rear and moves rear to the next node.

deQueue() This operation removes the front node and moves front to the next node.

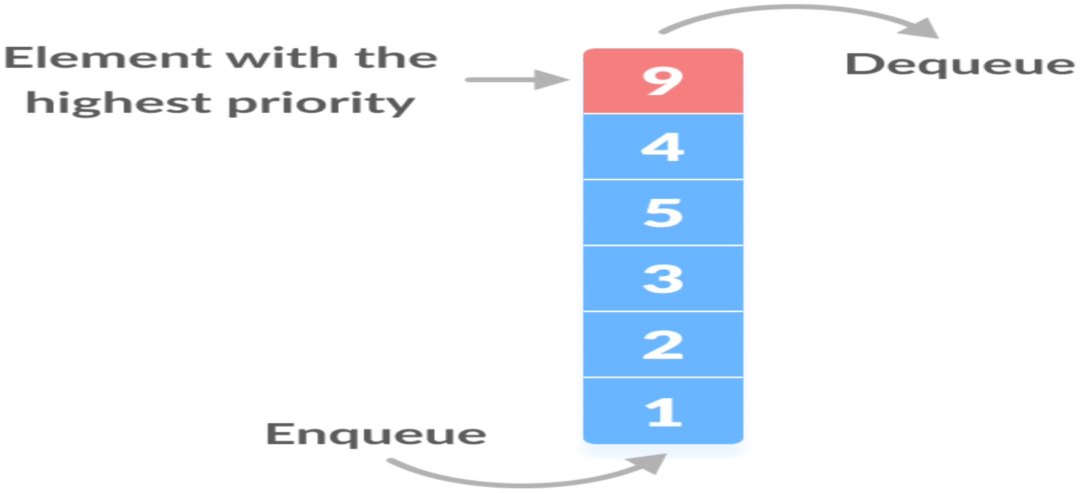
```
class Node:

    def __init__(self, data):
        self.data = data
        self.next = None

# A class to represent a queue

# The queue, front stores the front node
# of LL and rear stores the last node of LL
class Queue:

    def __init__(self):
        self.front = self.rear = None

    def isEmpty(self):
        return self.front == None

    # Method to add an item to the queue
    def EnQueue(self, item):
        temp = Node(item)

        if self.rear == None:
            self.front = self.rear = temp
            return
        self.rear.next = temp
        self.rear = temp

    # Method to remove an item from queue
    def DeQueue(self):
        if self.isEmpty():
            return
        temp = self.front
        self.front = temp.next

        if(self.front == None):
            self.rear = None
```

```
# Driver Code
if __name__== '__main__':
    q = Queue()
    q.EnQueue(10)
    q.EnQueue(20)
    q.DeQueue()
    q.DeQueue()
    q.EnQueue(30)
    q.EnQueue(40)
    q.EnQueue(50)
    q.DeQueue()
    print("Queue Front " + str(q.front.data))
    print("Queue Rear " + str(q.rear.data))
```

Priority Queue

A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue.

Generally, the value of the element itself is considered for assigning the priority. For example, The element with the highest value is considered as the highest priority element. However, in other cases, we can assume the element with the lowest value as the highest priority element. In other cases, we can set priorities according to our needs.



Removing Highest Priority Element

Difference between Priority Queue and Normal Queue

In a queue, the first-in-first-out rule is implemented whereas, in a priority queue, the values are removed on the basis of priority. The element with the highest priority is removed first.

Implementation of Priority Queue

Priority queue can be implemented using an array, a linked list, a heap data structure, or a binary search tree. Among these data structures, heap data structure provides an efficient implementation of priority queues.

Hence, we will be using the heap data structure to implement the priority queue in this tutorial. A max-heap is implemented in the following operations. If you want to learn more about it, please visit max-heap and mean-heap.

A comparative analysis of different implementations of priority queue is given below.

Operations	peek	insert	delete
Linked List	O(1)	O(n)	O(1)
Binary Heap	O(1)	O(log n)	O(log n)
Binary Search Tree	O(1)	O(log n)	O(log n)

Priority Queue Operations

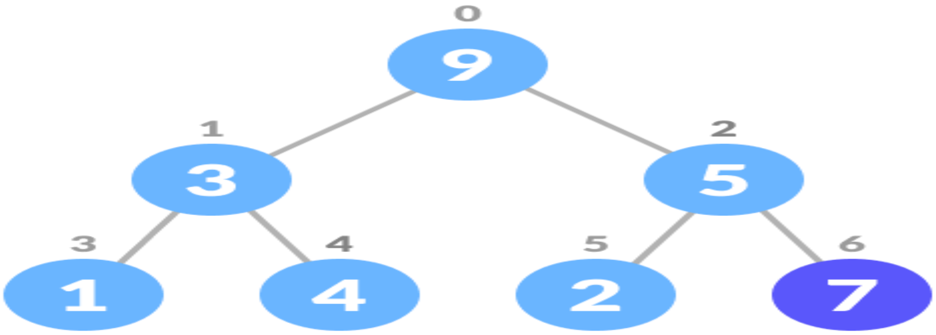
Basic operations of a priority queue are inserting, removing, and peeking elements.

Before studying the priority queue, please refer to the heap data structure for a better understanding of binary heap as it is used to implement the priority queue in this article.

1. Inserting an Element into the Priority Queue

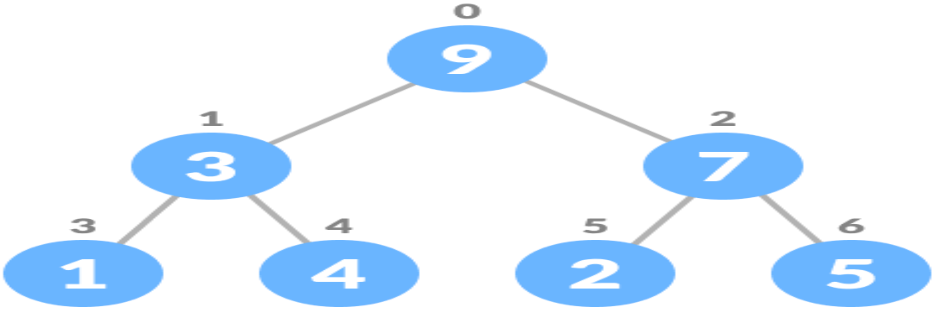
Inserting an element into a priority queue (max-heap) is done by the following steps.

Insert the new element at the end of the tree.



Insert an element at the end of the queue

Heapify the tree.



Heapify after insertion

Algorithm for insertion of an element into priority queue (max-heap)

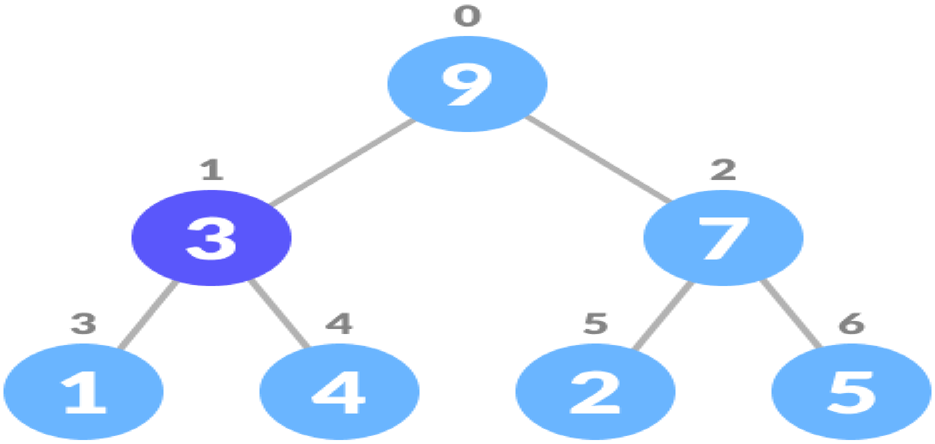
```
If there is no node,
  create a newNode.
else (a node is already present)
  insert the newNode at the end (last node from left to right.)

heapify the array
```

For Min Heap, the above algorithm is modified so that parentNode is always smaller than newNode.

2. Deleting an Element from the Priority Queue

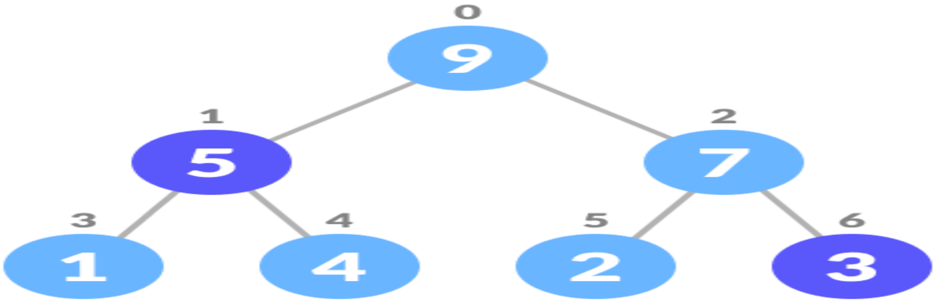
Deleting an element from a priority queue (max-heap) is done as follows:

Select the element to be deleted.



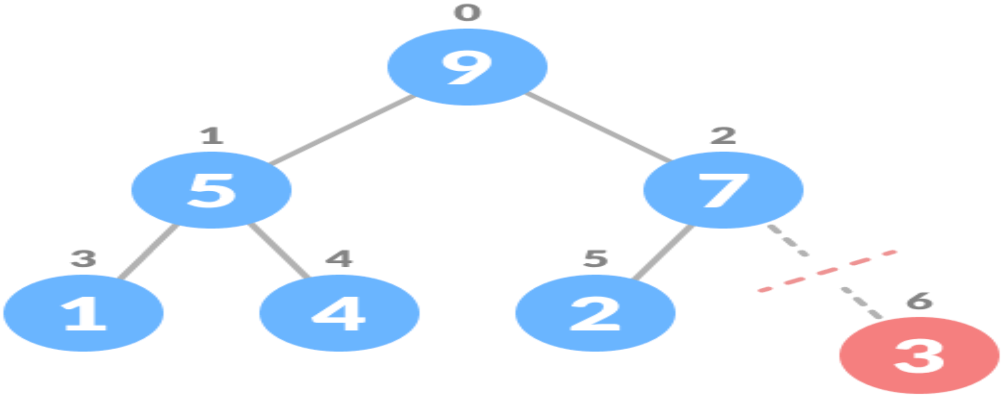
Select the element to be deleted

Swap it with the last element.

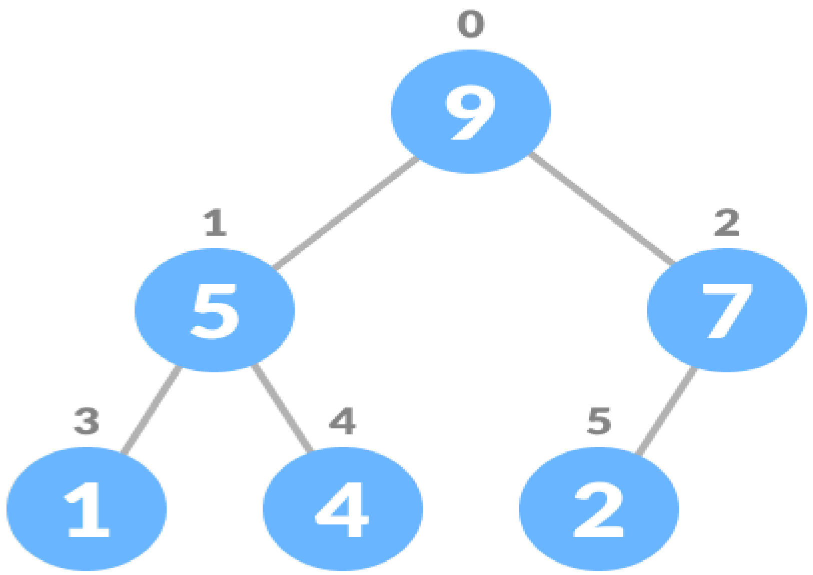


Swap with the last leaf node element

Remove the last element.



Remove the last element leaf



Heapify the tree.

Heapify the priority queue

Algorithm for deletion of an element in the priority queue (max-heap)

```
If nodeToBeDeleted is the leafNode
  remove the node
Else swap nodeToBeDeleted with the lastLeafNode
  remove noteToBeDeleted

heapify the array
```

For Min Heap, the above algorithm is modified so that the both childNodes are smaller than currentNode.

3. Peeking from the Priority Queue (Find max/min)

Peek operation returns the maximum element from Max Heap or minimum element from Min Heap without deleting the node.

```
For both Max heap and Min Heap
return rootNode
```

4. Extract-Max/Min from the Priority Queue

Extract-Max returns the node with maximum value after removing it from a Max Heap whereas Extract-Min returns the node with minimum value after removing it from Min Heap.

Priority Queue Implementations

```
# Priority Queue implementation in Python


# Function to heapify the tree
def heapify(arr, n, i):
    # Find the largest among root, left child and right child
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2

    if l < n and arr[i] < arr[l]:
        largest = l

    if r < n and arr[largest] < arr[r]:
        largest = r

    # Swap and continue heapifying if root is not largest
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

# Function to insert an element into the tree
def insert(array, newNum):
    size = len(array)
    if size == 0:
        array.append(newNum)
    else:
        array.append(newNum)
        for i in range((size // 2) - 1, -1, -1):
            heapify(array, size, i)

# Function to delete an element from the tree
def deleteNode(array, num):
    size = len(array)
    i = 0
    for i in range(0, size):
        if num == array[i]:
            break

    array[i], array[size - 1] = array[size - 1], array[i]

    array.remove(size - 1)
    for i in range((len(array) // 2) - 1, -1, -1):
        heapify(array, len(array), i)

arr = []

insert(arr, 3)
insert(arr, 4)
insert(arr, 9)
insert(arr, 5)
insert(arr, 2)

print ("Max-Heap array: " + str(arr))

deleteNode(arr, 4)
print("After deleting an element: " + str(arr))
```

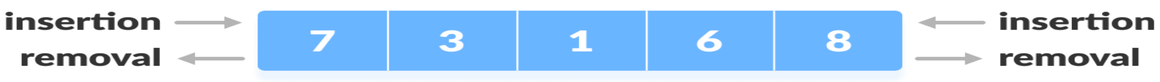
Priority Queue Applications

Some of the applications of a priority queue are:

- Dijkstra's algorithm
- for implementing stack
- for load balancing and interrupt handling in an operating system
- for data compression in Huffman code

Deque

Deque or Double Ended Queue is a type of queue in which insertion and removal of elements can be performed from either from the front or rear. Thus, it does not follow FIFO rule (First In First Out).



Representation of Deque

Types of Deque

Input Restricted Deque

In this deque, input is restricted at a single end but allows deletion at both the ends.

Output Restricted Deque

In this deque, output is restricted at a single end but allows insertion at both the ends.

Operations on a Deque

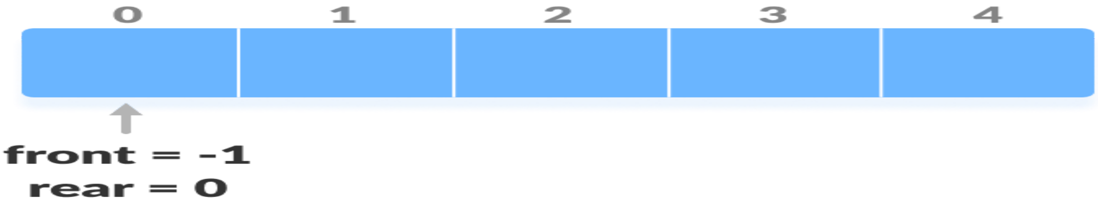
Below is the circular array implementation of deque. In a circular array, if the array is full, we start from the beginning.

But in a linear array implementation, if the array is full, no more elements can be inserted. In each of the operations below, if the array is full, "overflow message" is thrown.

Before performing the following operations, these steps are followed.

Take an array (deque) of size n.

Set two pointers at the first position and set front = -1 and rear = 0.

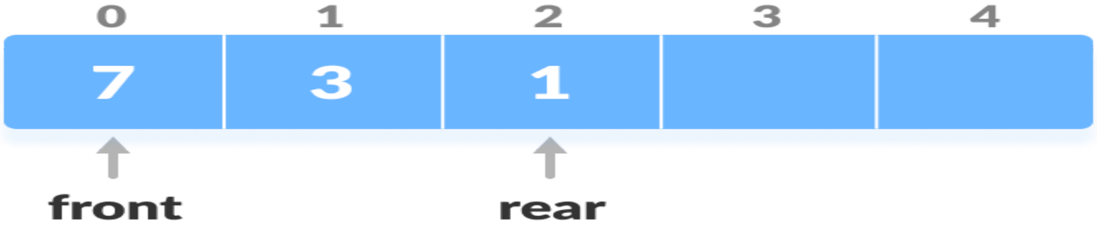


Initialize an array and pointers for deque

1. Insert at the Front

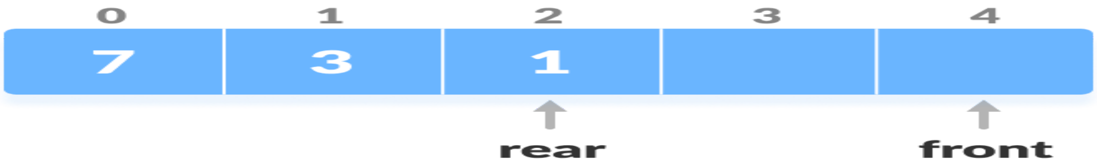
This operation adds an element at the front.

Check the position of front.



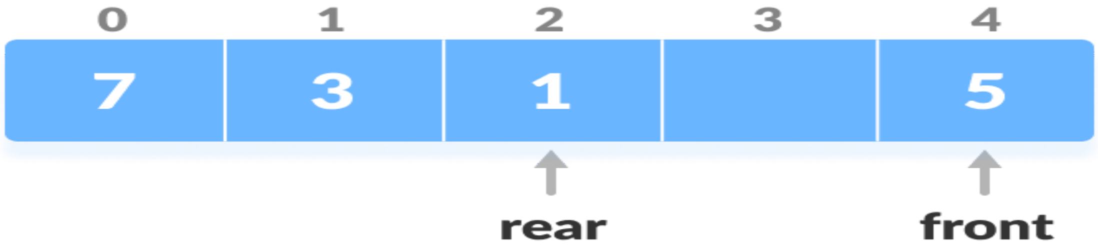
Check the position of front

If front < 1, reinitialize front = n-1 (last index).



Shift front to the end

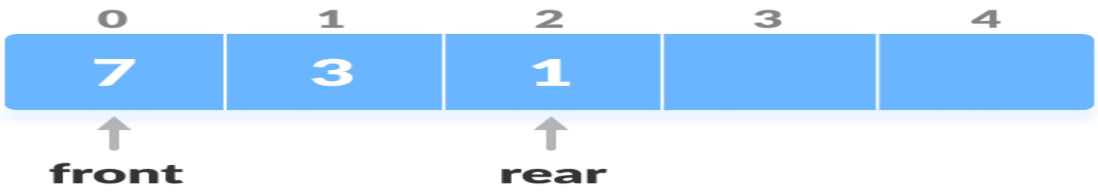
Else, decrease front by 1.
Add the new key 5 into array[front].



Insert the element at Front

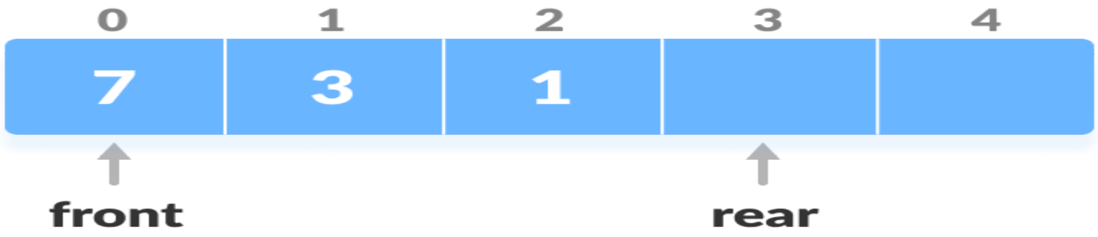
2. Insert at the Rear

This operation adds an element to the rear.
Check if the array is full.



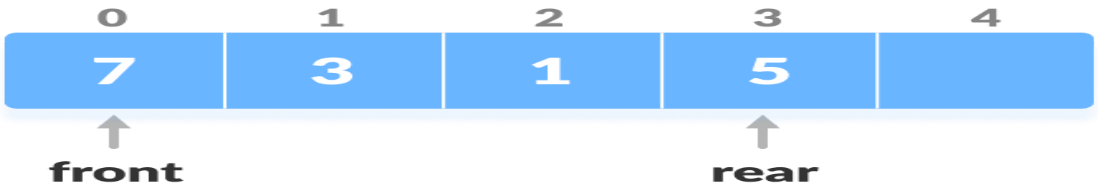
Check if deque is full

If the deque is full, reinitialize rear = 0.
Else, increase rear by 1.



Increase the rear

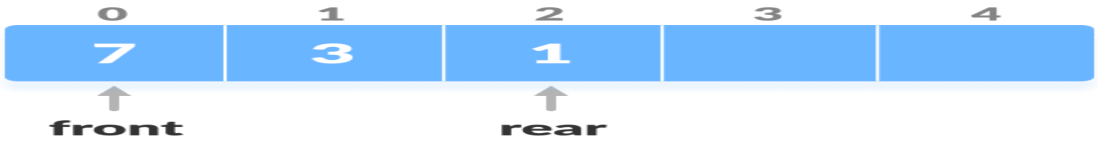
Add the new key 5 into array[rear].



Insert the element at rear

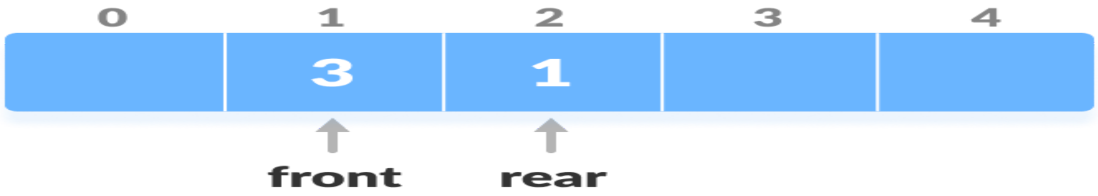
3. Delete from the Front

The operation deletes an element from the front.
Check if the deque is empty.



Check if deque is empty

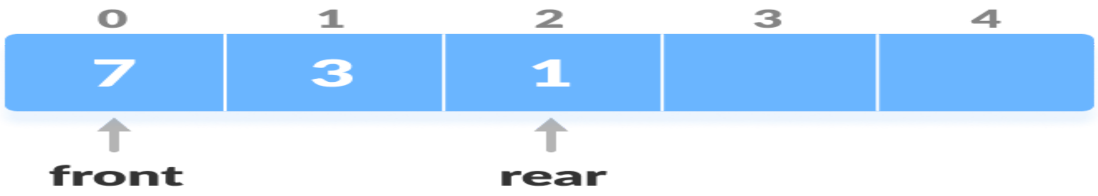
If the deque is empty (i.e. front = -1), deletion cannot be performed (underflow condition).
If the deque has only one element (i.e. front = rear), set front = -1 and rear = -1.
Else if front is at the end (i.e. front = n - 1), set go to the front front = 0.
Else, front = front + 1.



Increase the front

4. Delete from the Rear

This operation deletes an element from the rear.
Check if the deque is empty.



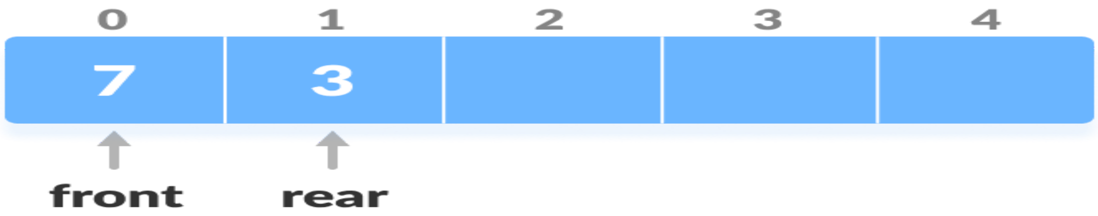
Check if deque is empty

If the deque is empty (i.e. front = -1), deletion cannot be performed (underflow condition).

If the deque has only one element (i.e. front = rear), set front = -1 and rear = -1, else follow the steps below.

If rear is at the front (i.e. rear = 0), set go to the front rear = n - 1.

Else, rear = rear - 1.



Decrease the rear

5. Check Empty

This operation checks if the deque is empty. If front = -1, the deque is empty.

6. Check Full

This operation checks if the deque is full. If front = 0 and rear = n - 1 OR front = rear + 1, the deque is full.

Deque Implementation

```
class Deque:
  def __init__(self):
    self.items = []

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

  def addRear(self, item):
    self.items.append(item)

  def addFront(self, item):
    self.items.insert(0, item)

  def removeFront(self):
    return self.items.pop()

  def removeRear(self):
    return self.items.pop(0)
    def size(self):
        return len(self.items)

d = Deque()
print(d.isEmpty())
d.addRear(8)
d.addRear(5)
d.addFront(7)
d.addFront(10)
print(d.size())
print(d.isEmpty())
d.addRear(11)
print(d.removeRear())
print(d.removeFront())
d.addFront(55)
d.addRear(45)
print(d.items)
```

Time Complexity

The time complexity of all the above operations is constant i.e. O(1).

Applications of Deque Data Structure

- In undo operations on software.
- To store history in browsers.
- For implementing both stacks and queues.