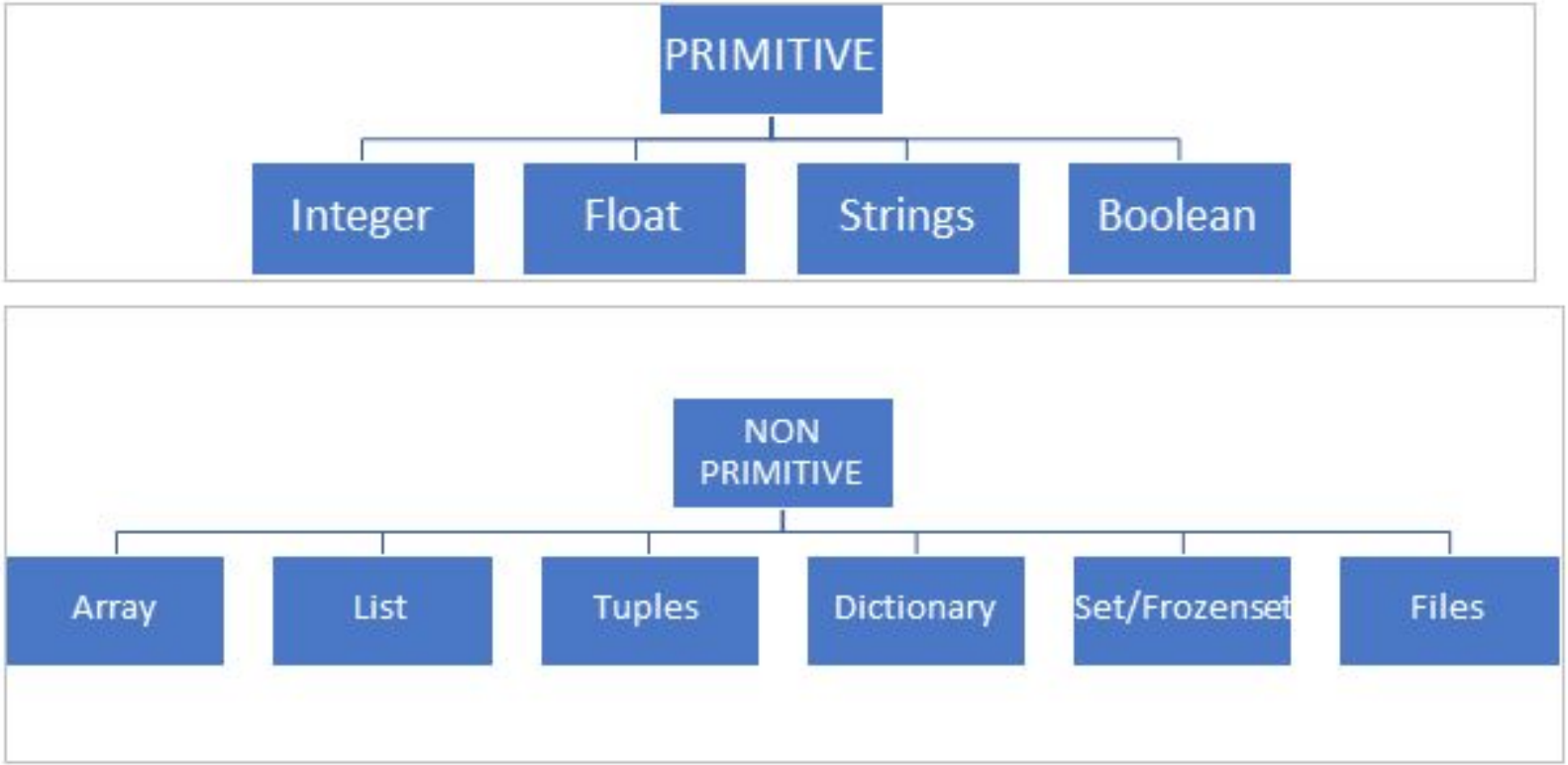
Data Structures and fundamentals of Algorithm

UNIT 1

Data Types


PRIMITIVE
Integer
Float
Strings
Boolean
NON PRIMITIVE
Array
List
Tuples
Dictionary
Set/Frozenset
Files

Data Structures

The data structure is a specific way of storing and organizing data in the computer's memory so that these data can be easily retrieved and efficiently used when needed later.

Built In	User Defined
1. List 2. Tuple 3. Dictionary 4. Set	1. Stack 2. Queue 3. Tree 4. Graph 5. Linked List 6. Hash map

****Only Tree and Graph are Non-Linear, other all are linear**

Abstract Data Types (ADT)

- An abstract data type (or ADT) is a programmer-defined data type that specifies a set of data values and a collection of well-defined operations that can be performed on those values
- Defined independent of their implementation details
- 2 types: Simple and Complex ADT
- ADT operations can be categorized as:
  - **Constructors**: Create and initialize new instances of the ADT.
  - **Accessors**: Return data contained in an instance without modifying it.
  - **Mutators**: Modify the contents of an ADT instance.
  - **Iterators**: Process individual data components sequentially.
- Eg. Date ADT, List, Stack, Queue

Abstractions

- An abstraction is a mechanism for separating the properties of an object and restricting the focus to those relevant in the current context.
- The user of the abstraction does not have to understand all of the details in order to utilize the object, but only those relevant to the current task or problem.
- Types of abstractions:
  - Procedural abstractions
  - Data abstractions

Abstractions

★ **Procedural abstractions:**

➔ Procedural abstraction is the use of a function or method knowing what it does but ignoring how it's accomplished.

➔ Consider the mathematical square root function which you have probably used at some point. You know the function will compute the square root of a given number, but do you know how the square root is computed?

★ **Data abstractions**

➔ The separation of the properties of a data type (its values and operations) from the implementation of that data type.

➔ You have used strings in Python many times. But do you know how they are implemented? That is, do you know how the data is structured internally or how he various operations are implemented?

Date ADT

- A date represents a single day in the proleptic Gregorian calendar.
- An example of simple ADT
- Operations:

1. Date(month, day, year): Creates a new Date instance initialized to the given Gregorian date which must be valid.
2. day(): Returns the Gregorian day number of this date.
3. month(): Returns the Gregorian month number of this date.
4. year(): Returns the Gregorian year of this date.
5. monthName(): Returns the Gregorian month name of this date.
6. dayOfWeek(): Returns the day of the week as a number between 0 and 6 with 0 representing Monday and 6 representing Sunday.

Date ADT

7. numDays(otherDate): Returns the number of days as a positive integer between this date and the otherDate.

8. isLeapYear(): Determines if this date falls in a leap year and returns the appropriate boolean value.

9. advanceBy(days): Advances the date by the given number of days.

Other examples of ADT

- Lists- sort(), append(), clear(), del() etc.
- Stack- push(), pop(), peek(), isEmpty()

Date ADT

```
from date import Date

def main():
    # Date before which a person must have been born to be 21 or older.
    bornBefore = Date(6, 1, 1988)

    # Extract birth dates from the user and determine if 21 or older.
    date = promptAndExtractDate()
    while date is not None :
        if date <= bornBefore :
            print( "Is at least 21 years of age: ", date )
        date = promptAndExtractDate()

def promptAndExtractDate():
    print( "Enter a birth date." )
    month = int( input("month (0 to quit): ") )
    if month == 0 :
        return None
    else :
        day = int( input("day: ") )
        year = int( input("year: ") )
        return Date( month, day, year )

# Call the main routine.
main()
```

Bags

- A bag is a simple container that can be used to store a collection of items.
- Allows duplicate values
- Unordered
- Cannot access individual items
- Operations that can be performed include:
  - Adding items
  - Removing items
  - Determine if item is in bag
  - Traverse over the collection of items

Bags

- A bag is a container that stores a collection in which duplicate values are allowed.
- The items, each of which is individually stored, have no particular order but they must be comparable.
- Bag(): Creates a bag that is initially empty.
- length (): Returns the number of items stored in the bag. Accessed using the len() function.

Bags

- contains (item): Determines if the given target item is stored in the bag and returns the appropriate boolean value. Accessed using the in operator.
- add(item): Adds the given item to the bag.
- remove(item): Removes and returns an occurrence of item from the bag. An exception is raised if the element is not in the bag.
- iterator (): Creates and returns an iterator that can be used to iterate over the collection of items.

Bags- Examples

```
class BagWithList(object):
  def __init__(self):
    self.collection = list()

  def add(self, item):
    self.collection.append(item)

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

  def is_empty(self):
    return len(self.collection) == 0

  def __iter__(self):
    return iter(self.collection)
```

Iterators

- Traversals are very common operations, especially on containers
- A traversal iterates over the entire collection, providing access to each individual element.
- Traversals can be used for a number of operations, including searching for a specific item or printing an entire collection
- Python’s container types—strings, tuples, lists, and dictionaries—can be traversed using the for loop construct.
- For our user-defined abstract data types, we can add methods that perform specific traversal operations when necessary

Iterators

- Python, provides a built-in iterator construct that can be used to perform traversals on user-defined ADTs.
- **An iterator is an object that provides a mechanism for performing generic traversals through a container without having to expose the underlying implementation.**
- Used with Python's for loop construct to provide a traversal mechanism for both built-in and user-defined containers.

Iterators

- It consists of the methods **__iter__() and __next__()**

CODE 1

```
mylist = ['b','a','n','a','n','a']
myit = iter(mylist)

print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
```

CODE 2

```
mylist = ['b','a','n','a','n','a']
myit = iter(mylist)

for x in myit:
    print(x)
```

Iterators- Create an Iterator

- To create an object/class as an iterator you have to implement the methods __iter__() and __next__() to your object.
- All classes have a function called __init__(), which allows you to do some initializing when the object is being created.
- The **__iter__()** method acts similar, you can do operations (initializing etc.), but must always return the iterator object itself.
- The **__next__()** method also allows you to do operations, and must return the next item in the sequence.

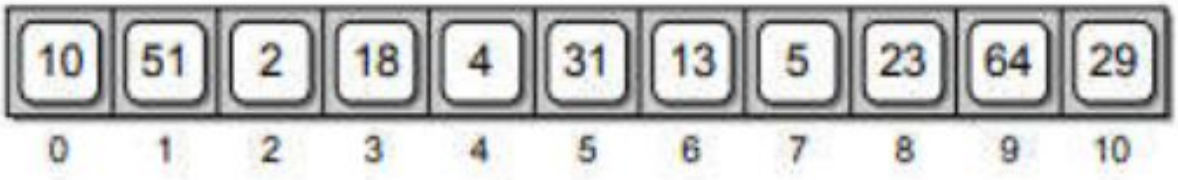
Iterators- Create an Iterator

Create an iterator that returns numbers, starting with 1, and each sequence will increase by one (returning 1,2,3,4,5 etc.)

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    if self.a <= 20:
      x = self.a
      self.a += 1
      return x
    else:
      raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)

for x in myiter:
  print(x)
```

ARRAYS

- An **array** is defined as a collection of elements of **similar data type**.
- Array elements are stored in contiguous memory.
- Array name represents its base address. The base address is the address of the first element of the array.
- Array elements are accessed by using an integer index



- Types of Array: Single Dimensional Array(1-D) & Multi-Dimensional Array (2-D OR 3-D)

Creating an Array

- An array cannot change size once it has been created.
- Array in Python can be created by importing an array module. array(data_type, value_list) is used to create an array with data type and value list specified in its arguments.

```
import array as arr
a = arr.array('i', [1, 2, 3])
print("The new created array is : ", end=" ")
for i in range(0, 3):
    print(a[i], end=" ")
print()
```

- Some of the data types are mentioned below which will help in creating an array of different data types.

Type code	C Type	Python Type	Minimum size in bytes
`'b'`	signed char	int	1
`'B'`	unsigned char	int	1
`'u'`	wchar_t	Unicode character	2
`'h'`	signed short	int	2
`'H'`	unsigned short	int	2
`'i'`	signed int	int	2
`'I'`	unsigned int	int	2
`'l'`	signed long	int	4
`'L'`	unsigned long	int	4
`'q'`	signed long long	int	8
`'Q'`	unsigned long long	int	8
`'f'`	float	float	4
`'d'`	double	float	8

Array ADT- Some Methods

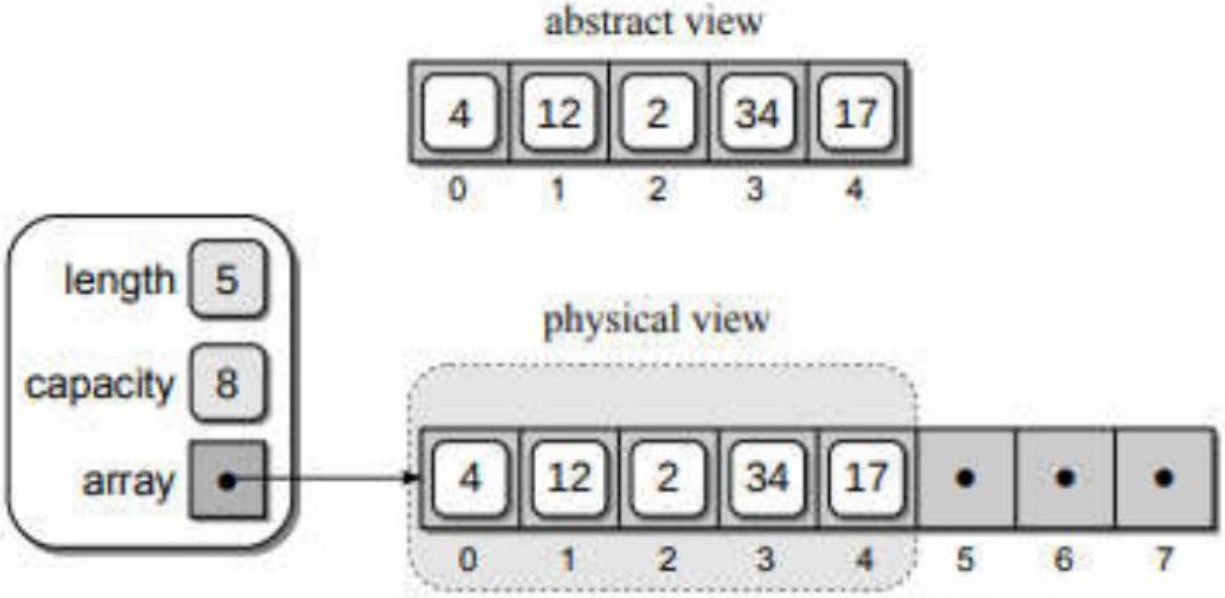
- **append(x):** Append a new item with value x to the end of the array.
- **count(x)** : Return the number of occurrences of x in the array.
- **extend(iterable)** : Append items from iterable to the end of the array. If iterable is another array, it must have exactly the same type code If iterable is not an array, it must be iterable and its elements must be the right type to be appended to the array.
- **insert(i, x)** : Insert a new item with value x in the array before position i. Negative values are treated as being relative to the end of the array.
- **pop([i]) :**Removes the item with the index i from the array and returns it. The optional argument defaults to -1, so that by default the last item is removed and returned.
- **reverse():** Reverse the order of the items in the array.

Python Lists

- An **list** is defined as a collection of elements of any data type.
- List items are ordered, changeable, and allow duplicate values.
- List items are indexed, the first item has index [0], the second item has index [1] etc.

Python Lists

- Suppose we create a list containing several values:
  pyList = [4, 12, 2, 34, 17]
- the list() constructor being called to create a list object and fill it with the given values.
- Following Figure illustrates the abstract and physical views of our sample list:



Python Lists

- In the physical view, the elements of the array structure used to store the actual contents of the list are enclosed inside the dashed gray box.
- The elements with null references shown outside the dashed gray box are the remaining elements of the underlying array structure that are still available for use.
- This notation will be used throughout the section to illustrate the contents of the list and the underlying array used to implement it.
- If there is room in the array, the item is stored in the next available slot of the array and the length field is incremented by one

Python Lists

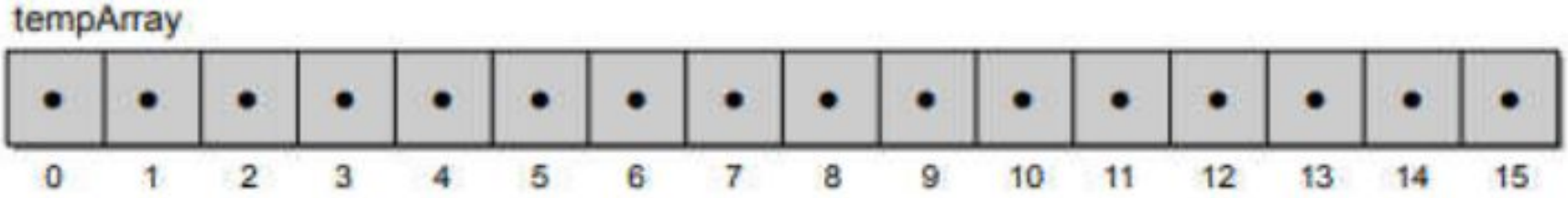
- For example, consider the following list operations:
  pyList.append(18)
  pyList.append(64)
  pyList.append(6)
- After the second statement is executed, the array becomes full

Python Lists

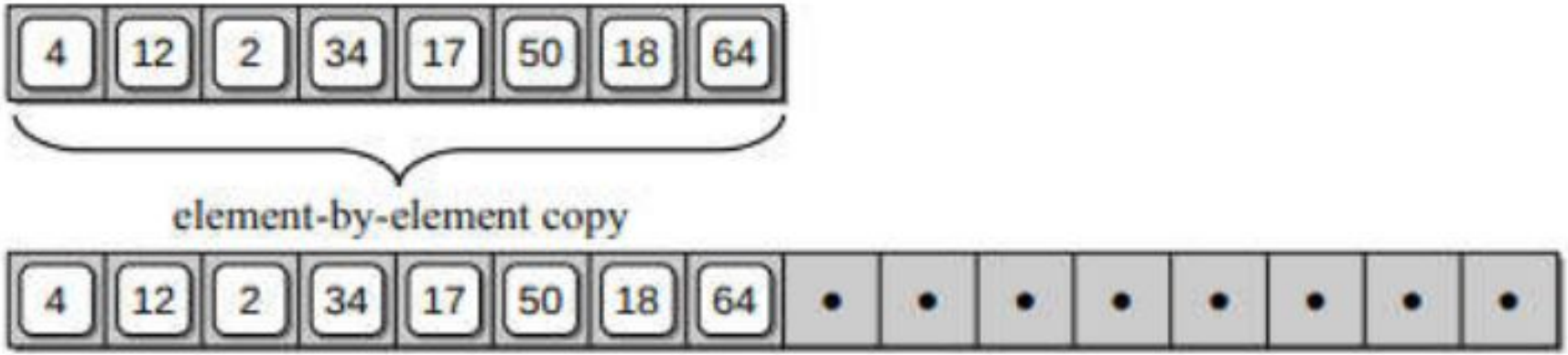
- By definition, a list can contain any number of items and never becomes full.
- Thus, when the third statement is executed, the array will have to be expanded to make room for value 6. (an array cannot change size once it has been created.)
- To allow for the expansion of the list, the following steps have to be performed:
  (1) a new array is created with additional capacity,
  (2) the items from the original array are copied to the new array,
  (3) the new larger array is set as the data structure for the list, and
  (4) the original smaller array is destroyed.

Python Lists

(1) A new array, double the size of the original, is created.

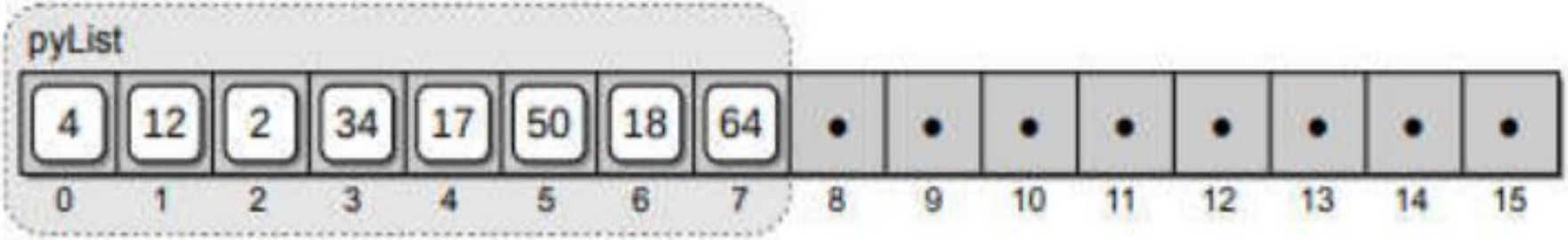


(2) The values from the original array are copied to the new larger array.

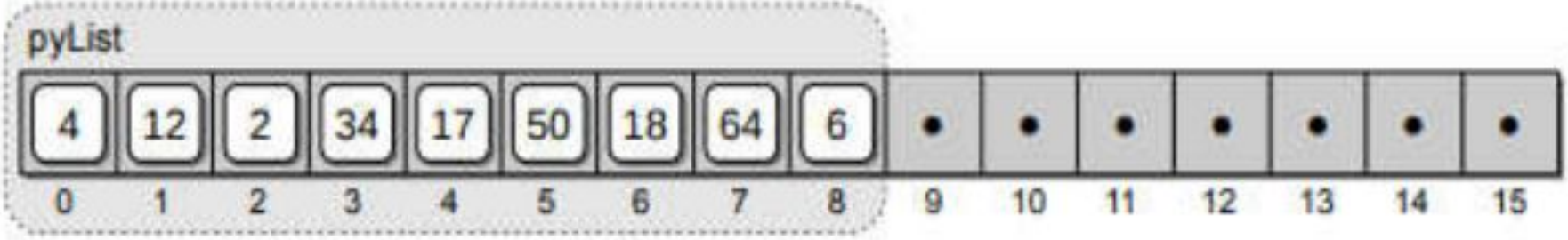


Python Lists

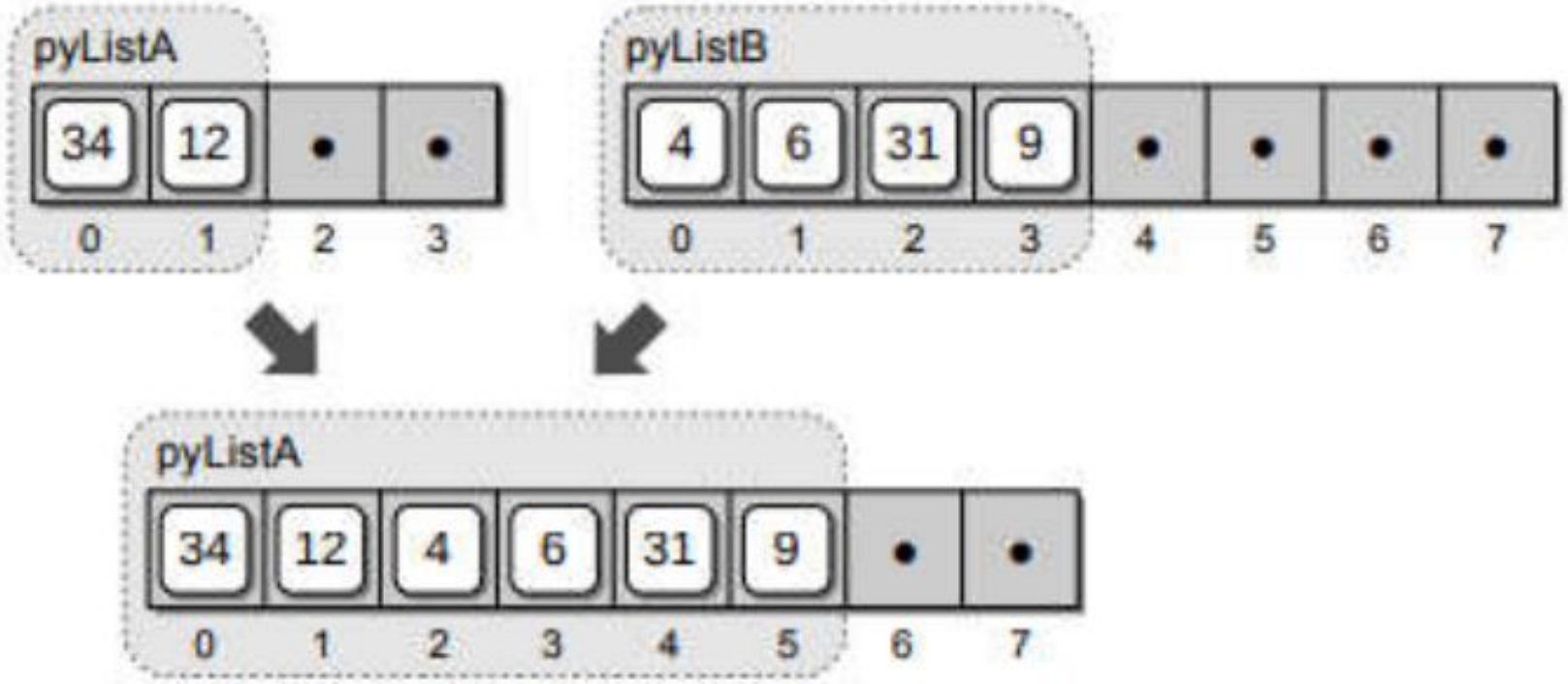
(3) The new array replaces the original in the list.



(4) Value 6 is appended to the end of the list.



Python Lists- Extending list



Python Lists- Inserting in a list

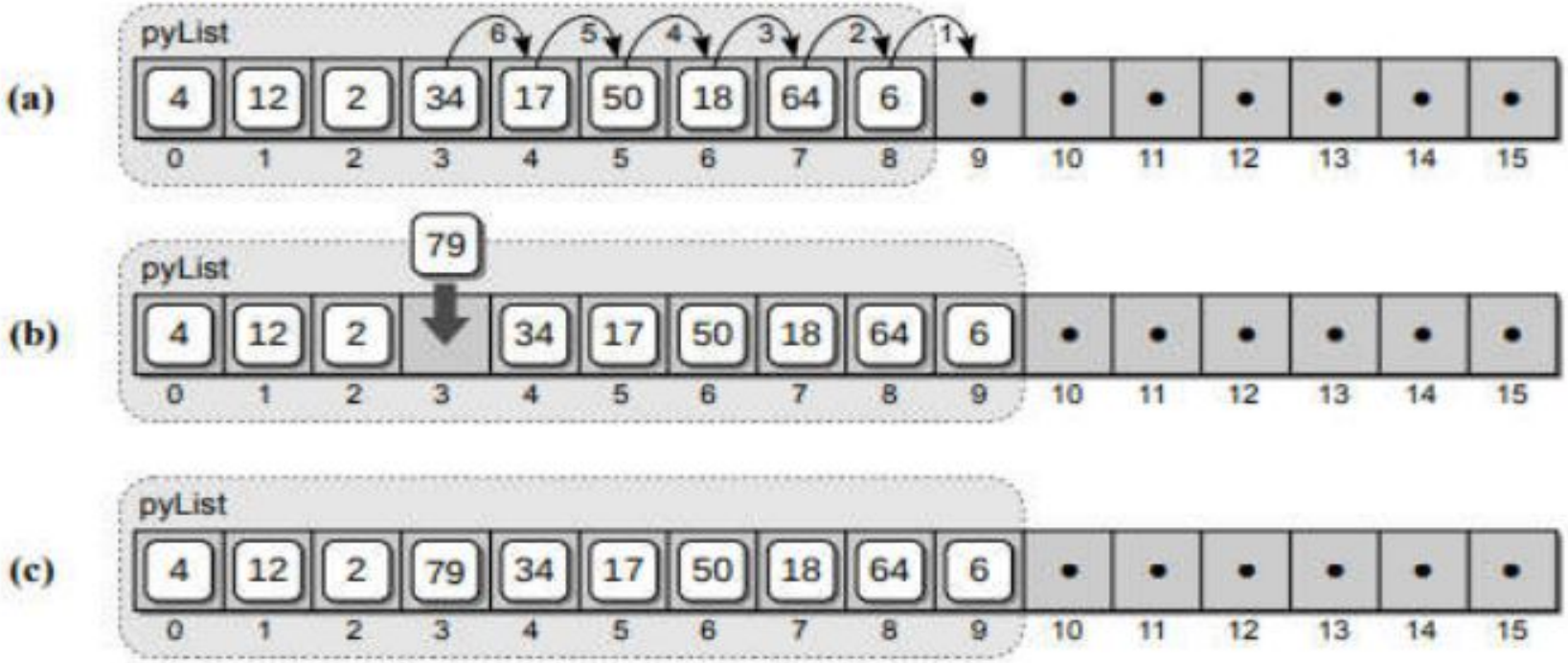


Figure: Inserting an item into a list: (a) the array elements are shifted to the right one at a time, traversing from right to left; (b) the new value is then inserted into the array at the given position; (c) the result after inserting the item.

Python Lists- Removing Items

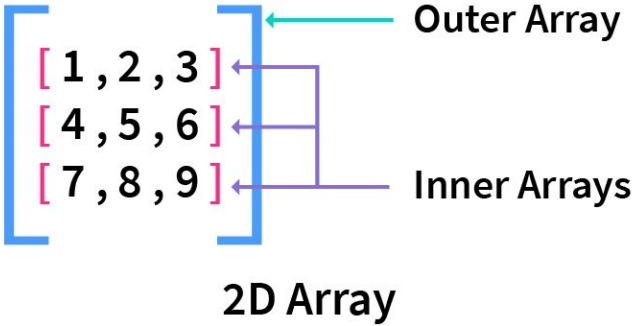
Figure: Removing an item from a list: (a) a copy of the item is saved; (b) the array elements are shifted to the left one at a time, traversing left to right; and (c) the size of the list is decremented by one.

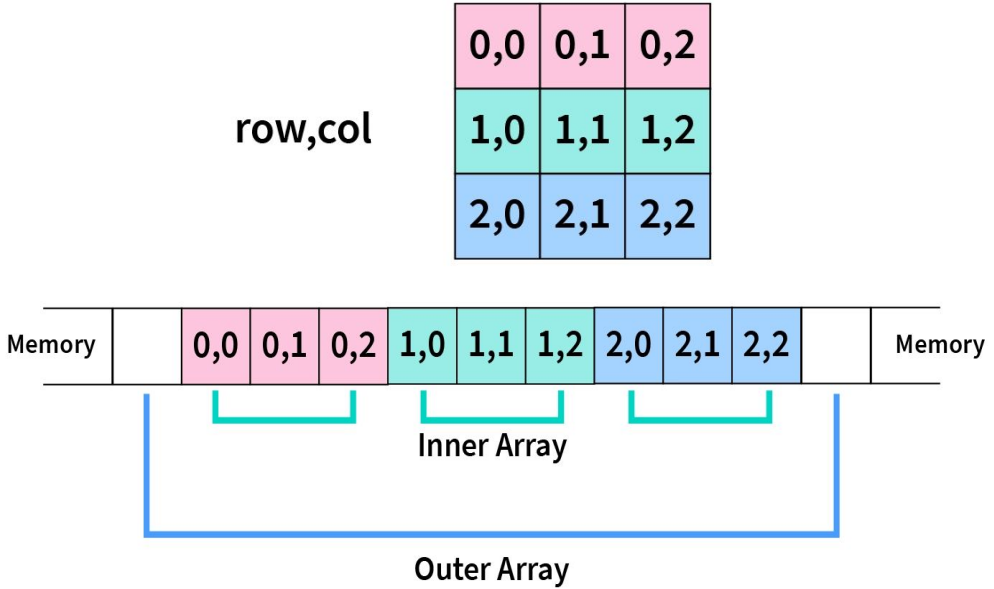
ARRAYS VS LISTS

LISTS	ARRAYS
It consists of elements that belong to the different data types.	It consists of elements that belong to the same data type.
It consumes a larger memory.	It consumes less memory than a list.
It favors a shorter sequence of data.	It favors a longer sequence of data.
The lists are the build-in data structure so we don't need to import it.	We need to import the array before work with the array.

TWO DIMENSIONAL ARRAYS

- A two-dimensional array consists of a collection of elements organized into rows and columns.
- Individual elements are referenced by specifying the specific row and column indices (r, c), both of which start at 0.
- Following: Figure shows an abstract view of both a one- and a two dimensional array





TWO DIMENSIONAL ARRAYS

- **Basic Operations**

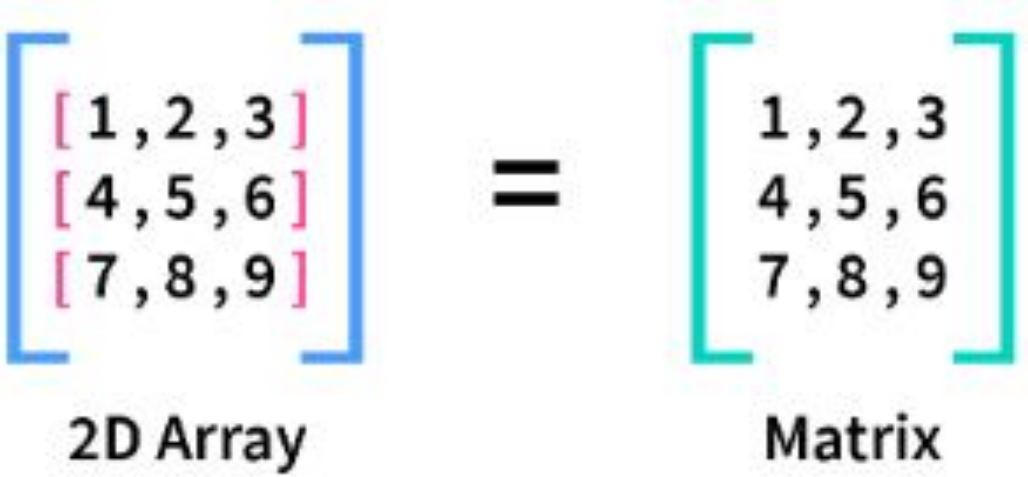
numRows()	The numRows() method can obtain the number of rows by checking the length of the main array, which contains an element for each row in the 2-D array.
numCols()	To determine the number of columns in the 2-D array, the numCols() method can simply check the length of any of the 1-D arrays used to store the individual rows.
clear()	The clear() method can set every element to the given value by calling the clear() method on each of the 1-D arrays used to store the individual rows.

TWO DIMENSIONAL ARRAYS

getitem(i1, i2):	Returns the value stored in the 2-D array element at the position indicated by the 2-tuple (i1, i2), both of which must be within the valid range. Accessed using the subscript operator: y = x[1,2].
setitem(i1, i2, value):	Modifies the contents of the 2-D array element indicated by the 2-tuple (i1, i2) with the new value. Both indices must be within the valid range. Accessed using the subscript operator: x[0,3] = y.

MATRIX ABSTRACT DATA TYPE

- A matrix is a collection of scalar(numeric, character) values arranged in rows and columns as a rectangular grid of a fixed size.
- The elements of the matrix can be accessed by specifying a given row and column index with indices starting at 0



MATRIX ABSTRACT DATA TYPE

Method	Description
Matrix(rows, ncols):	Creates a new matrix containing nrows and ncols with each element initialized to 0.
numRows():	Returns the number of rows in the matrix.
numCols():	Returns the number of columns in the matrix.
getitem (row, col):	Returns the value stored in the given matrix element. Both row and col must be within the valid range.
setitem (row, col, scalar):	Sets the matrix element at the given row and col to scalar. The element indices must be within the valid range.

MATRIX ABSTRACT DATA TYPE

scaleBy(scalar):	Multiplies each element of the matrix by the given scalar value. The matrix is modified by this operation.
transpose():	Returns a new matrix that is the transpose of this matrix.
add (rhsMatrix):	Creates and returns a new matrix that is the result of adding this matrix to the given rhsMatrix. The size of the two matrices must be the same.
subtract (rhsMatrix):	The same as the add() operation but subtracts the two matrices.
multiply (rhsMatrix):	Creates and returns a new matrix that is the result of multiplying this matrix to the given rhsMatrix. The two matrices must be of appropriate sizes as defined for matrix multiplication.

MATRIX ADT- Matrix Operations

- **Addition and Subtraction.**

1. Two m × n matrices can be added or subtracted to create a third m × n matrix.
2. When adding two m × n matrices, corresponding elements are summed as illustrated here.
3. Subtraction is performed in a similar fashion but the corresponding elements are subtracted instead of summed.

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix} + \begin{bmatrix} 6 & 7 \\ 8 & 9 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0+6 & 1+7 \\ 2+8 & 3+9 \\ 4+1 & 5+0 \end{bmatrix} = \begin{bmatrix} 6 & 8 \\ 10 & 12 \\ 5 & 5 \end{bmatrix}$$

MATRIX ADT- Matrix Operations

- **Scaling.**
  - A matrix can be uniformly scaled, which modifies each element of the matrix by the same scale factor.
  - A scale factor of less than 1 has the effect of reducing the value of each element whereas a scale factor greater than 1 increases the value of each element.
  - Scaling a matrix by a scale factor of 3 is illustrated here:

$$3\begin{bmatrix} 6 & 7 \\ 8 & 9 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 3*6 & 3*7 \\ 3*8 & 3*9 \\ 3*1 & 3*0 \end{bmatrix} = \begin{bmatrix} 18 & 21 \\ 24 & 27 \\ 3 & 0 \end{bmatrix}$$

MATRIX ADT- Matrix Operations

- **Transpose**
  - Another useful operation that can be applied to a matrix is the matrix transpose.
  - Given a m × n matrix, a transpose swaps the rows and columns to create a new matrix of size n × m as illustrated here:
  - $$\begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix}^T = \begin{bmatrix} 0 & 2 & 4 \\ 1 & 3 & 5 \end{bmatrix}$$

MATRIX ADT- Matrix Operations

- **Multiplication**

➔ Only defined for matrices where the number of columns in the matrix on the lefthand side is equal to the number of rows in the matrix on the righthand side

➔ Given a matrix of size m × n multiplied by a matrix of size n × p, the resulting matrix is of size m × p.

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{bmatrix} * \begin{bmatrix} 6 & 7 & 8 \\ 9 & 1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} (0*6+1*9) & (0*7+1*1) & (0*8+1*0) \\ (2*6+3*9) & (2*7+3*1) & (2*8+3*0) \\ (4*6+5*9) & (4*7+5*1) & (4*8+5*0) \end{bmatrix}$$

$$= \begin{bmatrix} 9 & 1 & 0 \\ 39 & 17 & 16 \\ 69 & 33 & 32 \end{bmatrix}$$

Sets

- A set is a container that stores a collection of unique values over a given comparable domain in which the stored values have no particular ordering.
- Unordered
- Mutable
- Defined using { } in python or set()

Method	Description
Set():	Creates a new set initialized to the empty set.
length ():	Returns the number of elements in the set, also known as the cardinality. Accessed using the len() function.
contains (element):	Determines if the given value is an element of the set and returns the appropriate boolean value. Accessed using the in operator.

Sets

Method	Description
add(element):	Modifies the set by adding the given value or element to the set if the element is not already a member. If the element is not unique, no action is taken and the operation is skipped.
remove(element):	Removes the given value from the set if the value is contained in the set and raises an exception otherwise.
equals (setB):	Determines if the set is equal to another set and returns a boolean value. For two sets, A and B, to be equal, both A and B must contain the same number of elements and all elements in A must also be elements in B. If both sets are empty, the sets are equal. Access with == or !=.
isSubsetOf(setB):	Determines if the set is a subset of another set and returns a boolean value. For set A to be a subset of B, all elements in A must also be elements in B.

Sets

Method	Description
union(setB):	Creates and returns a new set that is the union of this set and set B. The new set created from the union of two sets, A and B, contains all elements in A plus those elements in B that are not in A. Neither set A nor set B is modified by this operation.
intersect(setB):	Creates and returns a new set that is the intersection of this set and set B. The intersection of sets A and B contains only those elements that are in both A and B. Neither set A nor set B is modified by this operation.
difference(setB):	Creates and returns a new set that is the difference of this set and set B. The set difference, A–B, contains only those elements that are in A but not in B. Neither set A nor set B is modified by this operation.
iterator ():	Creates and returns an iterator that can be used to iterate over the collection of items.

Selecting a Data Structure for sets implementation

- To replicate the functionality of the set structure provided by Python, leaves the array, list, and dictionary containers for consideration in implementing the Set ADT
- Storage requirements for the **bag** and set are very similar with the difference being that a set cannot contain duplicates.
- **Dictionary** would seem to be the ideal choice since it can store unique items, but it would waste space in this case. As dictionary stores key/value pairs, which requires two data fields per entry.
- An **array** could be used to implement the set, but a set can contain any number of elements and by definition an array has a fixed size.
- To use the array structure, we would have to manage the expansion of the array when necessary in the same fashion as it's done for the list.

Selecting a Data Structure for sets implementation

- Since the list can grow as needed, it seems ideal for storing the elements of a set just as it was for the bag and it does provide for the complete functionality of the ADT.
- Since the list allows for duplicate values, however, we must make sure as part of the implementation that no duplicates are added to our set.

Maps

- A map is a container for storing a collection of data records in which each record is associated with a unique key.
- The key components must be comparable.

Method	Description
Map():	Creates a new empty map.
length ():	Returns the number of key/value pairs in the map.
contains (key):	Determines if the given key is in the map and returns True if the key is found and False otherwise.
add(key, value):	Adds a new key/value pair to the map if the key is not already in the map or replaces the data associated with the key if the key is in the map. Returns True if this is a new key and False if the data associated with the existing key is replaced.

remove(key):	Removes the key/value pair for the given key if it is in the map and raises an exception otherwise.
valueOf(key):	Returns the data record associated with the given key. The key must exist in the map or an exception is raised.
iterator ():	Creates and returns an iterator that can be used to iterate over the keys in the map.

Searching & Sorting algorithms

Searching

- Searching is the process of finding some particular element in the list.
- If the element is present in the list, then the process is called successful, and the process returns the location of that element; otherwise, the search is called unsuccessful.
- Two popular search methods are Linear Search and Binary Search.

Linear Search

- Linear search is also called a sequential search algorithm.
- It is the simplest searching algorithm.
- In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found.
- If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL.
- It is widely used to search an element from the unordered list, i.e., the list in which items are not sorted.
- The worst-case time complexity of linear search is O(n).

Linear Search- Algorithm

How Linear Search Works?

- The following steps are followed to search for an element $k = 1$ in the list below.

1. Start from the first element, compare k with each array element.

Linear Search

2. If x == k, element found.

Implementation of Linear Search in Python

```
1  def linearSearch( theValues, target ) :
2    n = len( theValues )
3    for i in range( n ) :
4       # If the target is in the ith element, return True
5      if theValues[i] == target
6        return True
7
8    return False   # If not found, return False.
```

Linear Search- Time & Space complexity

Case	Time Complexity
Best Case	O(1)
Average Case	O(n)
Worst Case	O(n)

Space Complexity	O(1)

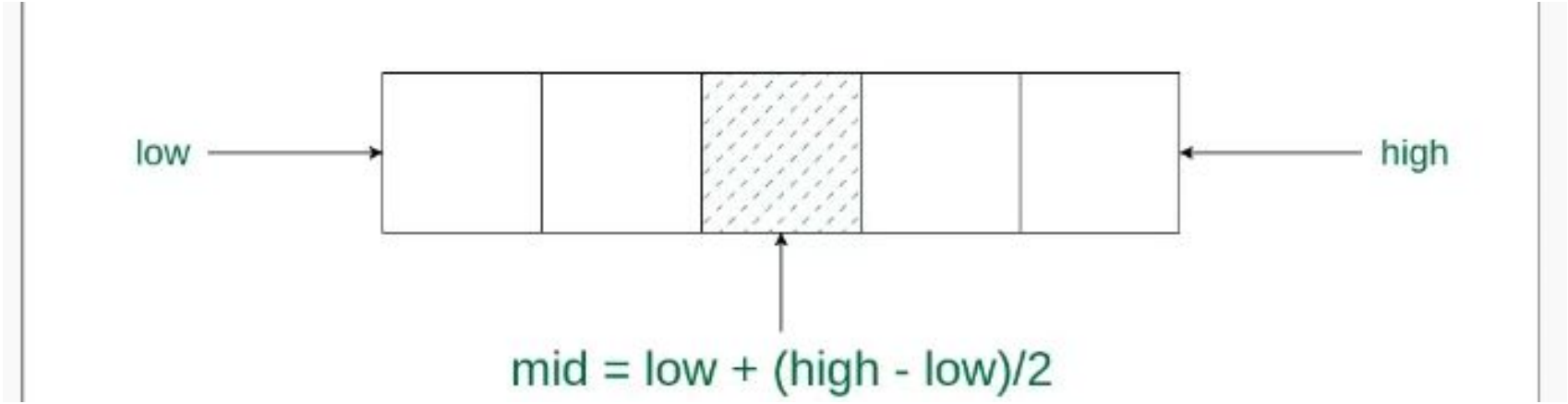
Binary Search

- Binary Search is a searching algorithm for finding an element's position in a sorted array.
- In this approach, the element is always searched in the middle of a portion of an array.
- If the match is found then, the location of the middle element is returned.
- Otherwise, we search into either of the halves depending upon the result produced through the match.
- Binary search can be implemented only on a sorted list of items.

Binary Search- Algorithm

In this algorithm,

- *Divide the search space into two halves by finding the middle index "mid".*



Binary Search- Algorithm

- *Compare the middle element of the search space with the key.*
- *If the key is found at middle element, the process is terminated.*
- *If the key is not found at middle element, choose which half will be used as the next search space.*
    - *If the key is smaller than the middle element, then the left side is used for next search.*
    - *If the key is larger than the middle element, then the right side is used for next search.*
- *This process is continued until the key is found or the total search space is exhausted.*

Implementation of Binary Search in Python

```
def binarySearch(mylist,target):
    start=0
    end=len(mylist)-1
    position=-1
    while start<=end:
        mid=(start + end)//2
        if target == mylist[mid]:
            position = mid
            break
        elif target >mylist[mid]:
            start = mid + 1
        else:
            end = mid - 1

    if position == -1:
        print("element not found")
    else:
        print("element found at ",position)
```

Binary Search- Time & Space complexity

Case	Time Complexity
Best Case	O(1)
Average Case	O(logn)
Worst Case	O(logn)

Space Complexity	O(1)

Sorting

- Sorting is the processing of arranging the data in ascending and descending order.
- There are several types of sorting in data structures namely – bubble sort, insertion sort, selection sort, merge sort, quick sort, radix sort etc.
- Sorting techniques are categorized into

★ **Internal Sorting** : Takes place in the main memory of a computer.

Example: - Bubble sort, Insertion sort, Quick sort, etc.

★ **External Sorting**: Takes place in the secondary memory of a computer,

Since the number of objects to be sorted is too large to fit in main memory.

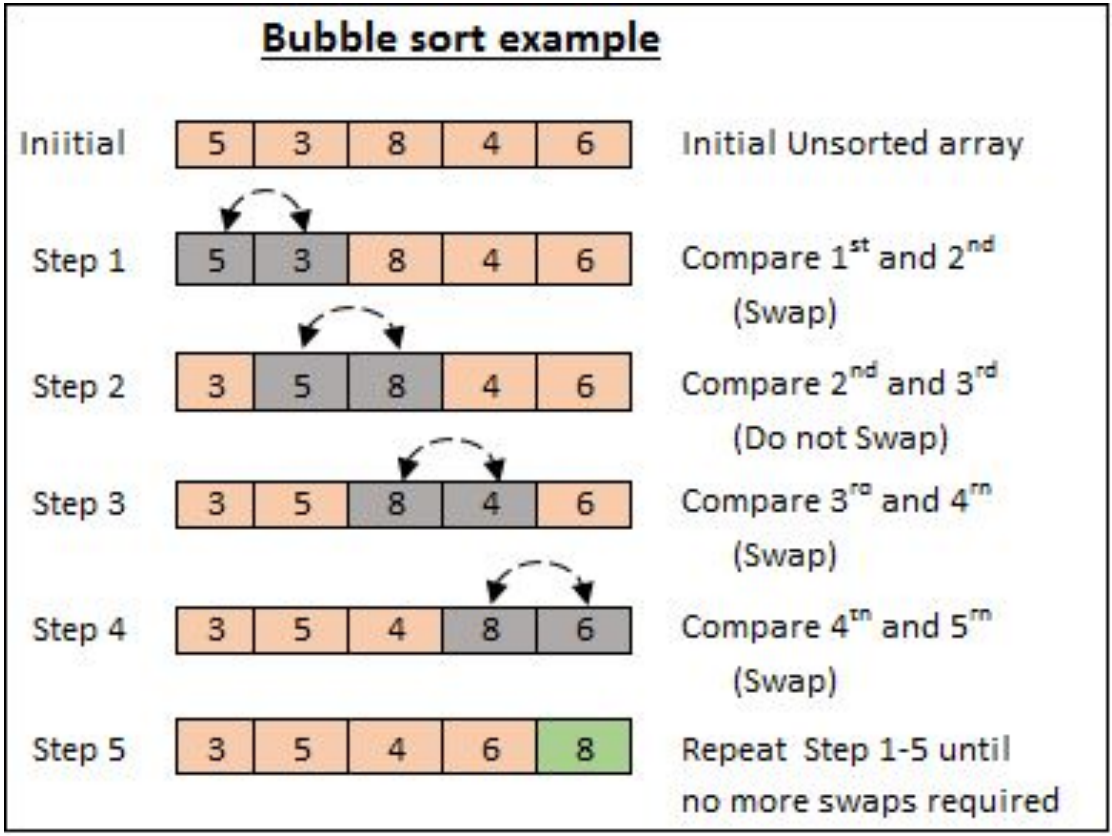
Example: - Merge Sort

Bubble Sort

- Bubble sort is a sorting algorithm that compares two adjacent elements and swaps them until they are not in the intended order.
- It is not suitable for large data sets.
- The average and worst-case complexity of Bubble sort is O(n2), where n is a number of items.
- Bubble short is majorly used where -
  - complexity does not matter
  - simple and shortcode is preferred

Bubble Sort Algorithm

- Traverse from left and compare adjacent elements and the higher one is placed at right side.
- In this way, the largest element is moved to the rightmost end at first.
- This process is then continued to find the second largest and place it and so on until the data is sorted.



Implementation of Bubble Sort in Python

```
def bubbleSort(array):
    for i in range(len(array)-1):
        flag=0
        for j in range(0, len(array) - i - 1 ):
            if array[j] > array[j + 1]:
                array[j],array[j+1] = array[j+1],array[j]
                flag=1

        if flag==0:
            break;
    return array
```

Bubble Sort - Time & Space complexity

Case	Time Complexity
Best Case	O(n)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

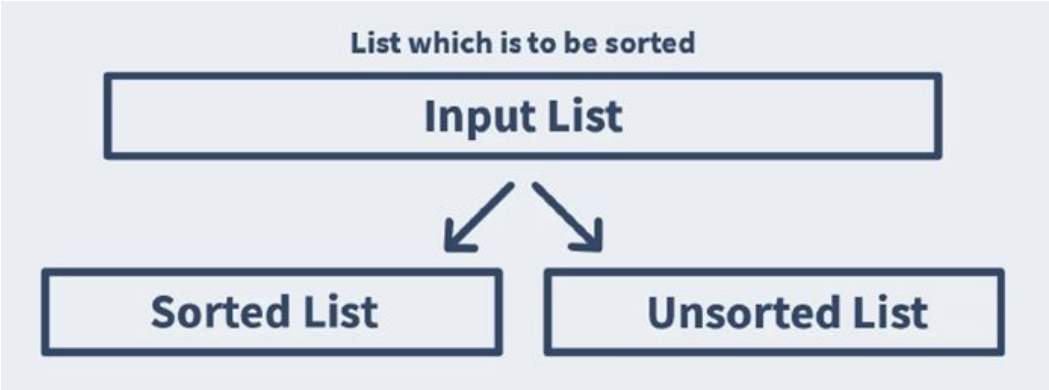
Space Complexity	O(1)
Stable	YES

- ***Stable sorting*** *maintains the position of two equals elements relative to one another.*
- ***Unstable sorting*** *does not maintain the position of two equals elements relative to one another.*

Selection Sort

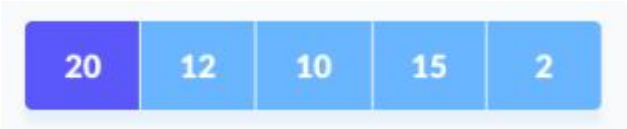
- Selection sort, also known as in-place comparison sort, is a simple sorting algorithm. It works on the idea of repeatedly finding the smallest element and placing it at its correct sorted position.
- Selection sort works by dividing the list into two sublists:

➔ Sorted sublist – that is built on the left end of the list from left to right.
➔ Unsorted sublist – that is the rest of the unsorted list, on the right end.



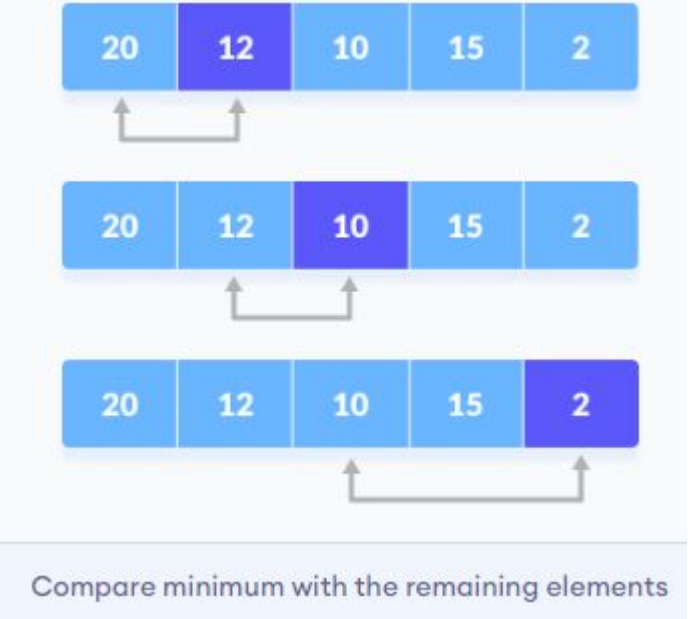
Working of Selection Sort

1. Set the first element as minimum



2. Compare minimum with the second element. If the second element is smaller than minimum, assign the second element as minimum.

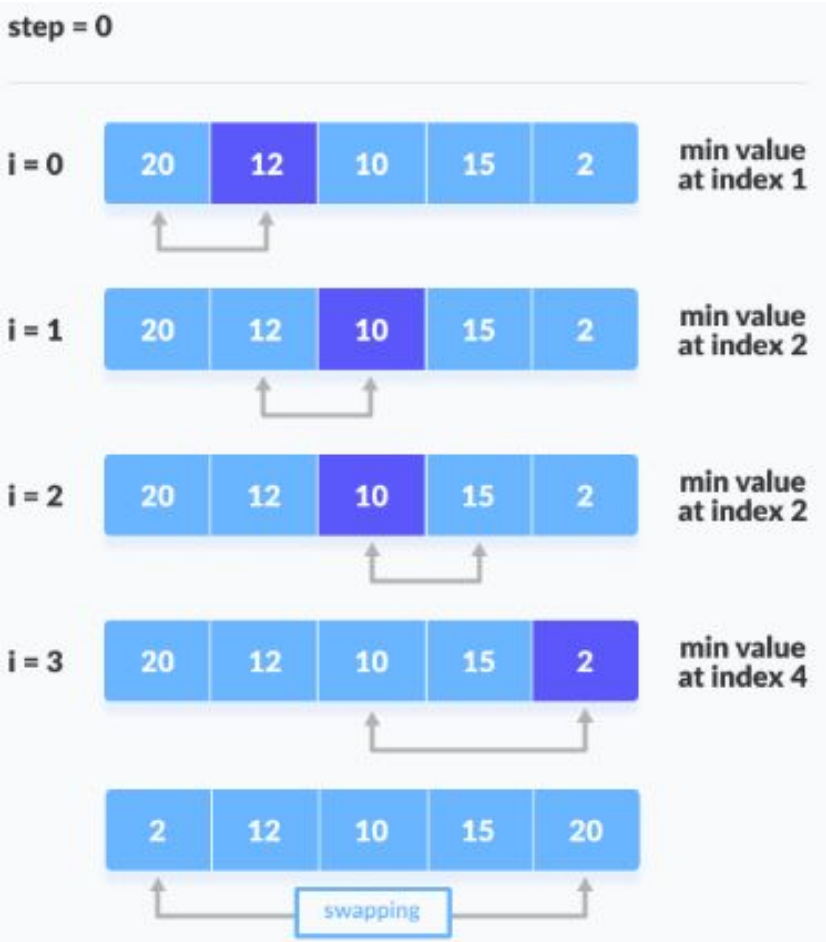
   Compare minimum with the third element. Again, if the third element is smaller, then assign minimum to the third element otherwise do nothing. The process goes on until the last element.



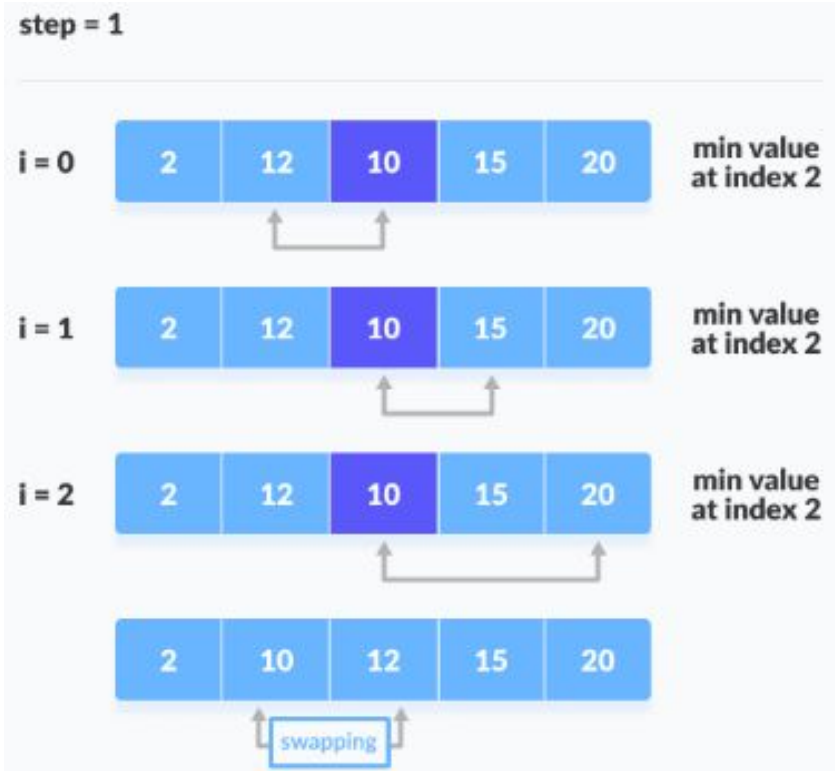
Working of Selection Sort

3. After each iteration, minimum is placed in the front of the unsorted list.

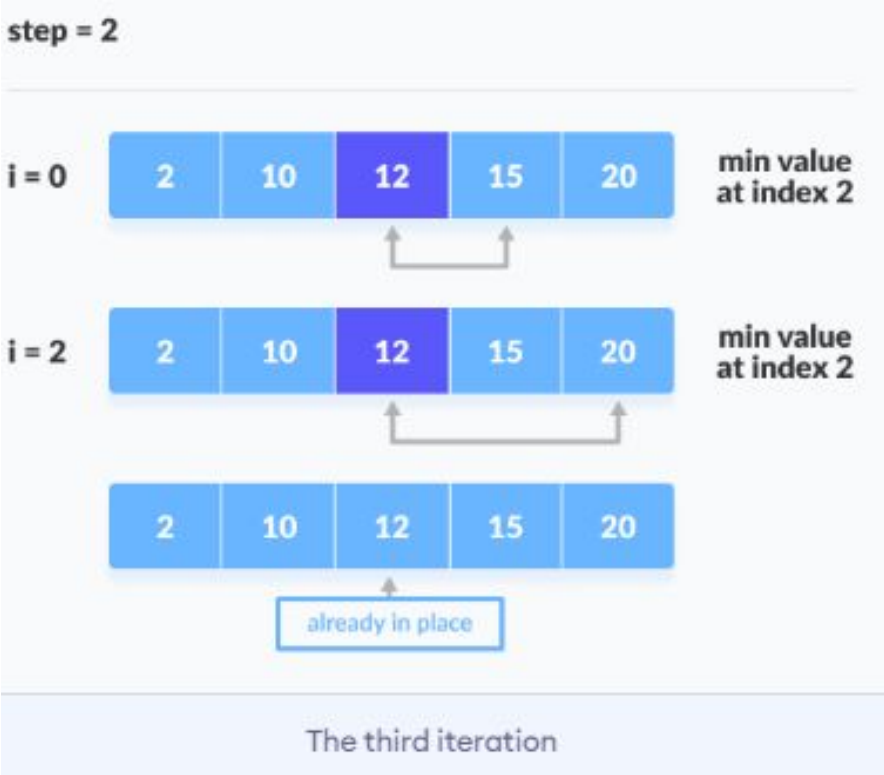
4. For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.



The first iteration



The second iteration



The third iteration

The fourth iteration

Implementation of Selection Sort

```
ll= [23,19,34,10,12,2]
def select_sort(l):
    for i in range(len(l)-1):
        min = i
        for j in range(i+1,len(l)):
            if l[j] < l[min]:
                min = j
        if min != i:
            l[i],l[min] = l[min],l[i]
    return l

print(select_sort(ll))
```

Selection Sort - Time & Space complexity

Case	Time Complexity
Best Case	$O(n^2)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

Space Complexity	O(1)
Stable	YES

- ***Stable sorting*** *maintains the position of two equals elements relative to one another.*
- ***Unstable sorting*** *does not maintain the position of two equals elements relative to one another.*

Insertion Sort

- Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.
- Insertion sort works similarly as we sort cards in our hand in a card game.
- We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put in their right place.

Insertion Sort- Algorithm

Step 1 - If the element is the first element, assume that it is already sorted. Return 1.

Step2 - Pick the next element, and store it separately in a **key.**

Step3 - Now, compare the **key** with all elements in the sorted array.

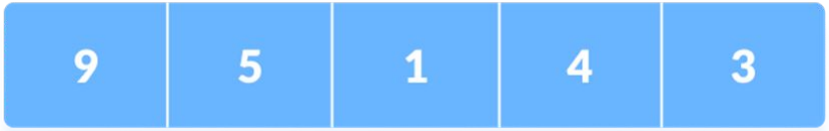
Step 4 - If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

Step 5 - Insert the value.

Step 6 - Repeat until the array is sorted.

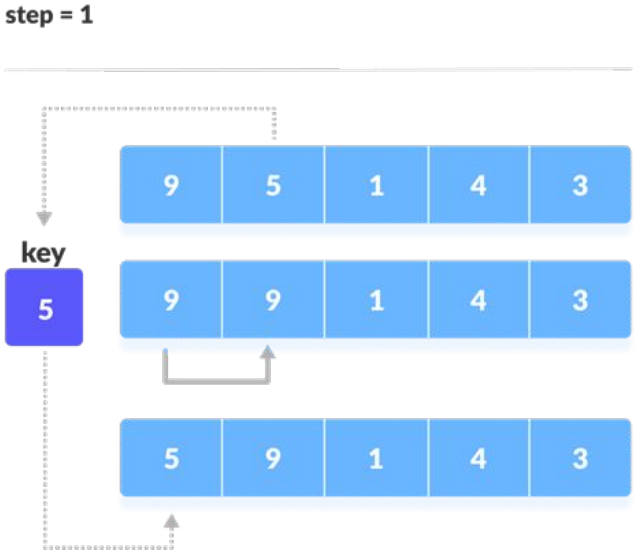
Working of Insertion Sort

1. Suppose we need to sort the following array



2. The first element in the array is assumed to be sorted. Take the second element and store it separately in key.

   Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element.

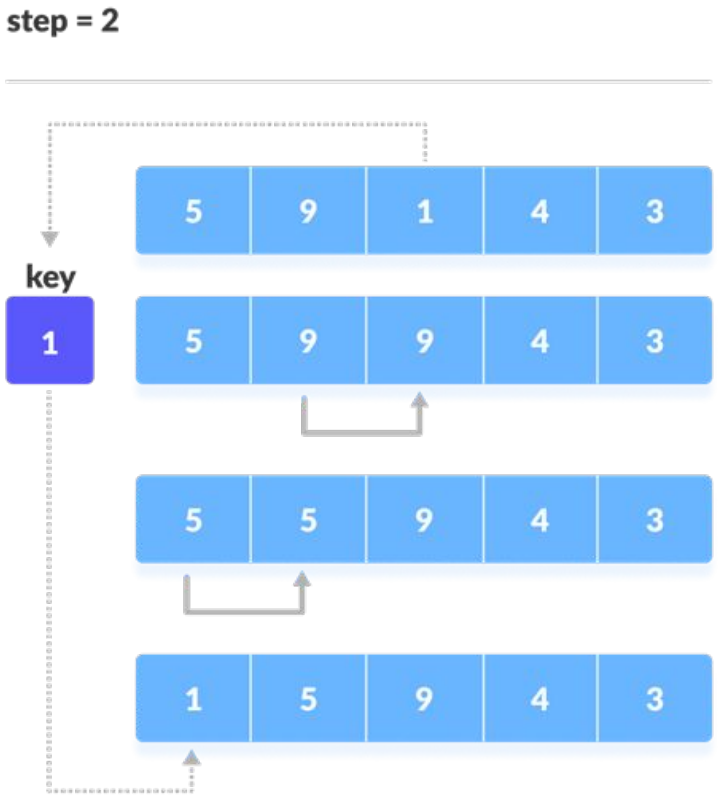


Working of Selection Sort

3. Now, the first two elements are sorted.

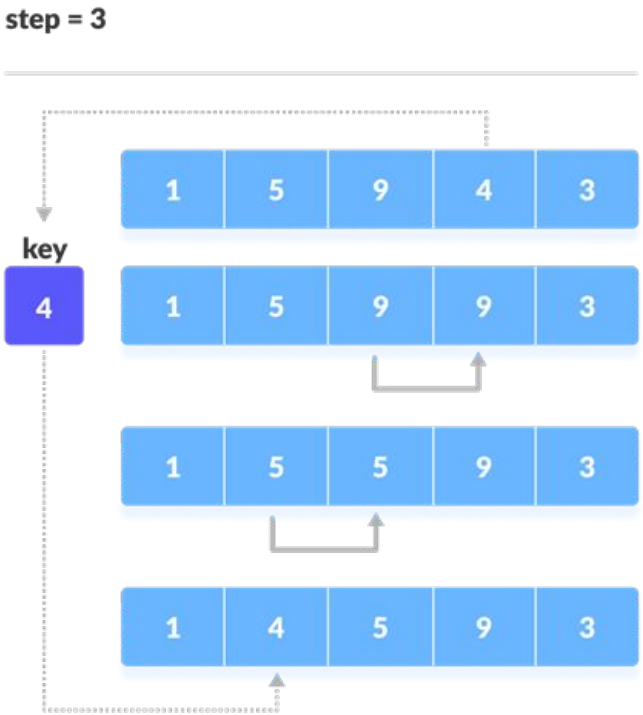
Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it.

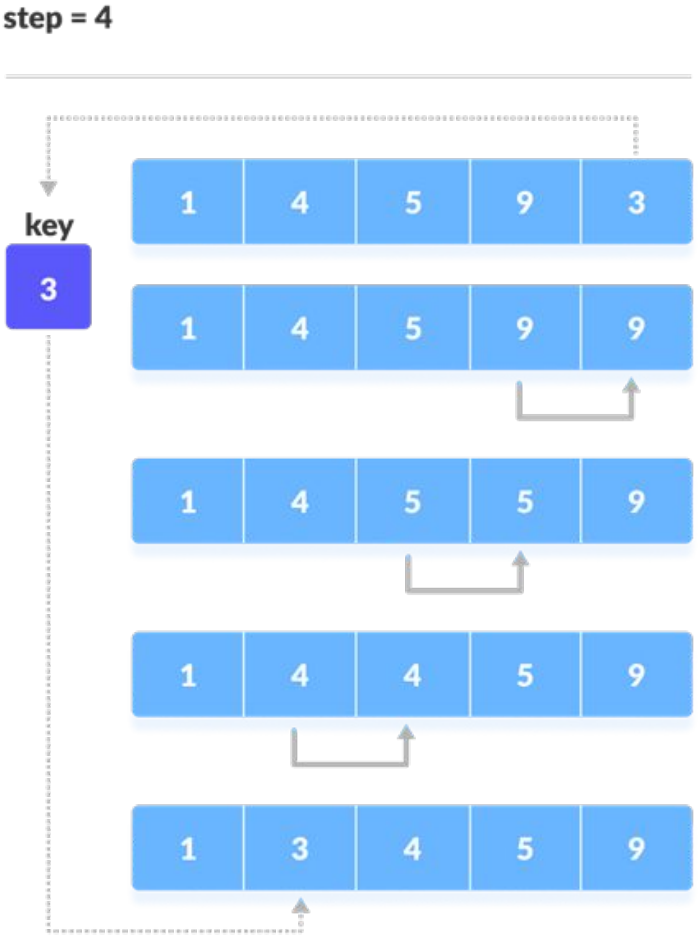
If there is no element smaller than it, then place it at the beginning of the array.



Working of Selection Sort

4. Similarly, place every unsorted element at its correct position





Implementation of Insertion Sort

```
def insertionSort(array):

    for step in range(1, len(array)):
        key = array[step]
        j = step - 1

        while j >= 0 and key < array[j]:
            array[j + 1] = array[j]
            j = j - 1
        array[j + 1] = key


data = [9, 5, 1, 4, 3]
insertionSort(data)
print('Sorted Array in Ascending Order:')
print(data)
```

Insertion Sort - Time & Space complexity

Case	Time Complexity
Best Case	O(n)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

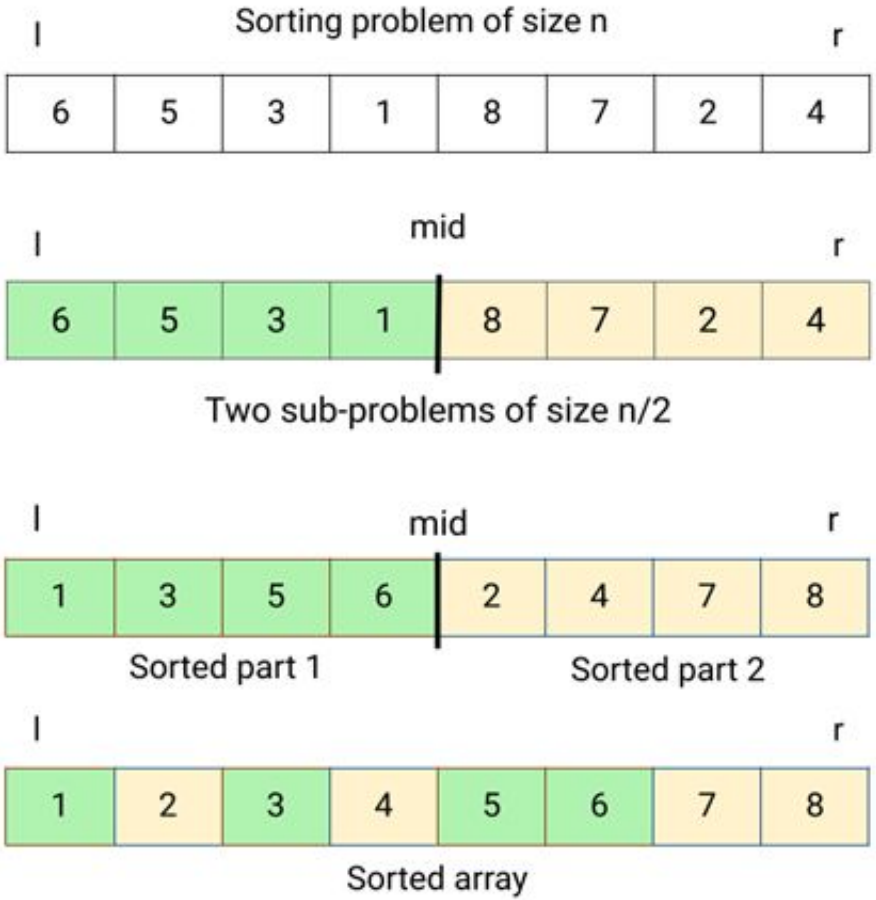
Space Complexity	O(1)
Stable	YES

- ***Stable sorting*** *maintains the position of two equals elements relative to one another.*
- ***Unstable sorting*** *does not maintain the position of two equals elements relative to one another.*

Merge Sort

- Merge Sort is one of the most popular sorting algorithms that is based on the principle of **Divide and Conquer Algorithm**.
- Here, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are combined to form the final solution.
- 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.

Merge sort visualization



Divide part : Dividing the problem into two equal sub-problems

Conquer part: Solving both the sub-problems recursively

Combine Part : Merging both the sorted part into a larger sorted array

Implementation of Merge Sort

```
def mergeSort(array):
    if len(array) > 1:
        r = (len(array)-1)//2
        L = array[:r+1]
        M = array[r+1:]
        mergeSort(L)
        mergeSort(M)

        i = j = k = 0

        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

        while i < len(L):
            array[k] = L[i]
            i += 1
            k += 1

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

Merge Sort - Time & Space complexity

Case	Time complexity
Best	O(nlogn)
Average	O(nlogn)
Worst	O(nlogn)

Space Complexity	O(n)
Stable	Yes

- ***Stable sorting*** *maintains the position of two equals elements relative to one another.*
- ***Unstable sorting*** *does not maintain the position of two equals elements relative to one another.*

Quick Sort

Quicksort is a sorting algorithm based on the **divide and conquer approach** where :

1. An array is divided into subarrays by selecting a pivot element (element selected from the array).
2. While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.
3. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
4. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

Working of Quick Sort

To sort an array, you will follow the steps below:

1. You will make any index value in the array as a pivot.
2. Then you will partition the array according to the pivot.
3. Then you will recursively quicksort the left partition
4. After that, you will recursively quicksort the correct partition.

Working of Quick Sort

Let's have a closer look at the partition bit of this algorithm:

1. You will pick any pivot, let's say the **lowest** index value.
2. You will take two variables to point left and right of the list, excluding pivot.
3. The left will point to the lower index, and the right will point to the higher index.
4. Now you will move all elements which are greater than pivot to the right.
5. Then you will move all elements smaller than the pivot to the left partition.

Implementation of Quick Sort

```
def partition(lst, lb, ub):
    pivot = lst[lb]
    start = lb
    end = ub
    while True:
        while start <= end and lst[end] > pivot:
            end = end - 1
        while start <= end and lst[start] <= pivot:
            start = start + 1
        if start <= end:
            lst[start], lst[end] = lst[end], lst[start]
        else:
            break
    lst[lb], lst[end] = lst[end], lst[lb]
    return end

def quick_sort(lst, lb, ub):
    if lb < ub:
        p = partition(lst, lb, ub)
        quick_sort(lst, lb, p-1)
        quick_sort(lst, p+1, ub)
```

Quick Sort - Time & Space complexity

Case	Time complexity
Best	O(nlogn)
Average	O(nlogn)
Worst	$O(n^2)$

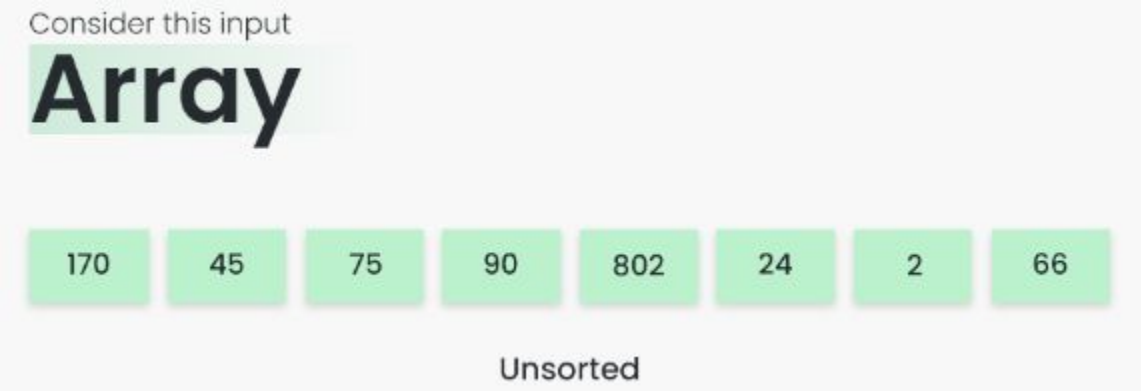
Space Complexity	O(n)
Stable	Yes

- ***Stable sorting*** *maintains the position of two equals elements relative to one another.*
- ***Unstable sorting*** *does not maintain the position of two equals elements relative to one another.*

Radix Sort

1. Radix Sort is a linear sorting algorithm.
2. Radix Sort's time complexity of O(nk), where n is the size of the array and k is the number of digits in the largest number.
3. It is not an in-place sorting algorithm because it requires extra space.
4. Radix Sort is a stable sort because it maintains the relative order of elements with equal values.
5. Because it is based on digits or letters, radix sort is less flexible than other sorts. If the type of data changes, the Radix sort must be rewritten.

Working of Radix Sort



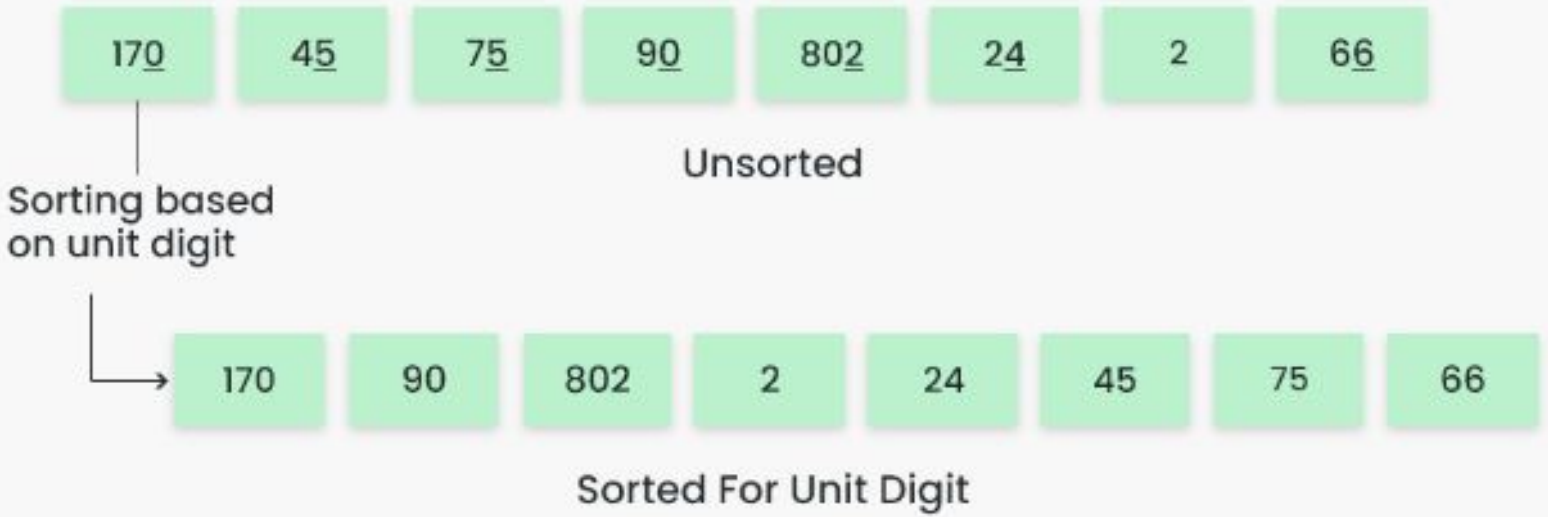
Step 1: *Find the largest element in the array, which is 802. It has three digits, so we will iterate three times, once for each significant place.*

Step 2: *Sort the elements based on the unit place digits (X=0). We use a stable sorting technique, such as counting sort, to sort the digits at each significant place.*

Working of Radix Sort

Sorting based on the unit place:

- *Perform counting sort on the array based on the unit place digits.*
- *The sorted array based on the unit place is [170, 90, 802, 2, 24, 45, 75, 66].*

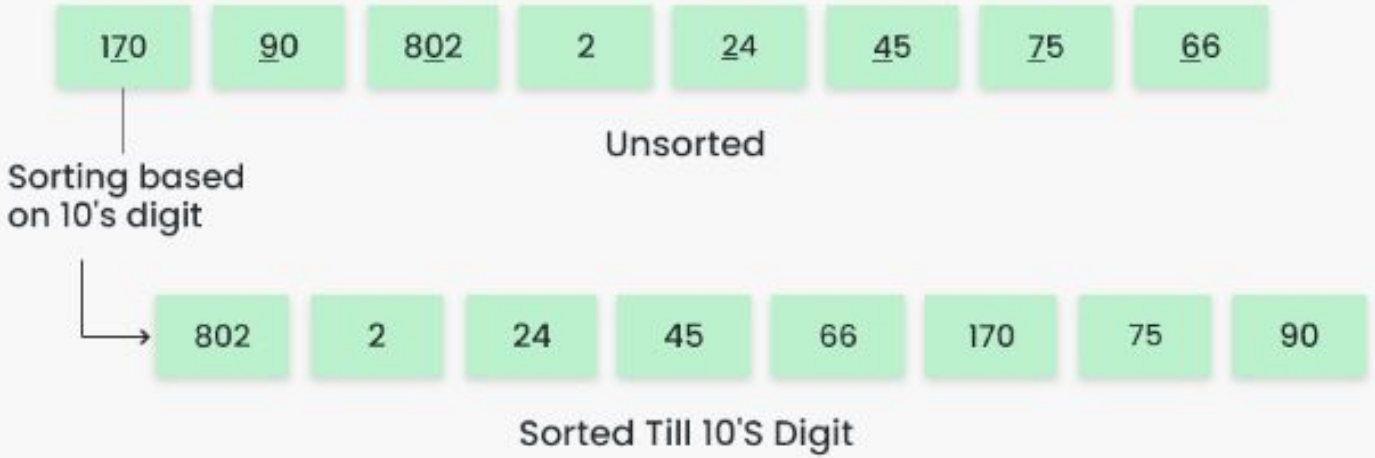


Working of Radix Sort

Step 3: *Sort the elements based on the tens place digits.*

Sorting based on the tens place:

- *Perform counting sort on the array based on the tens place digits.*
- *The sorted array based on the tens place is [802, 2, 24, 45, 66, 170, 75, 90].*

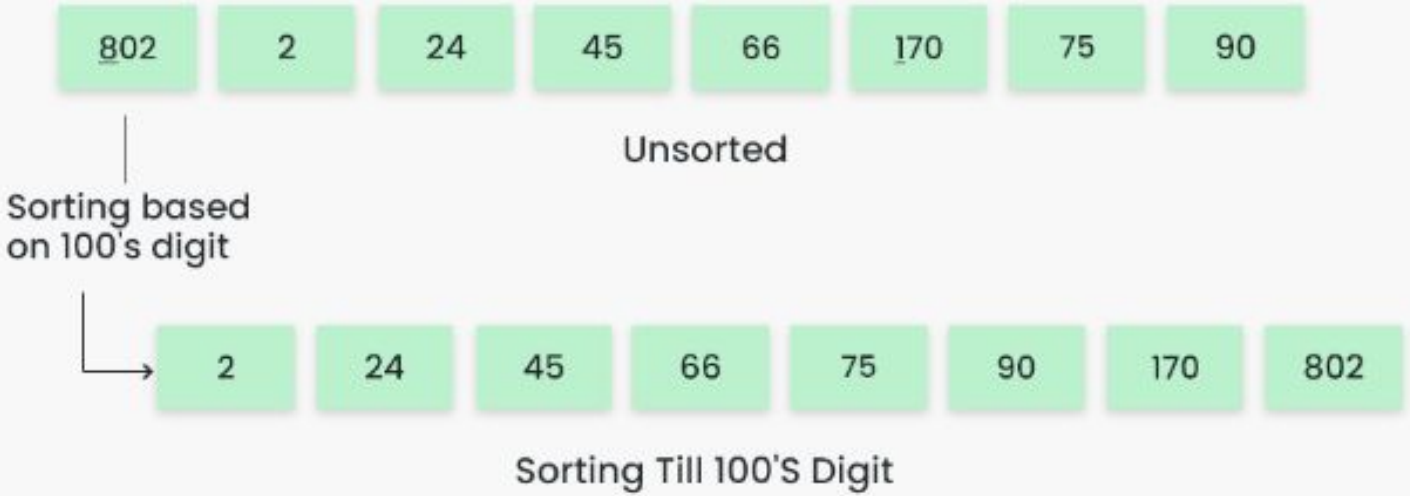


Working of Radix Sort

Step 4: *Sort the elements based on the hundreds place digits.*

Sorting based on the hundreds place:

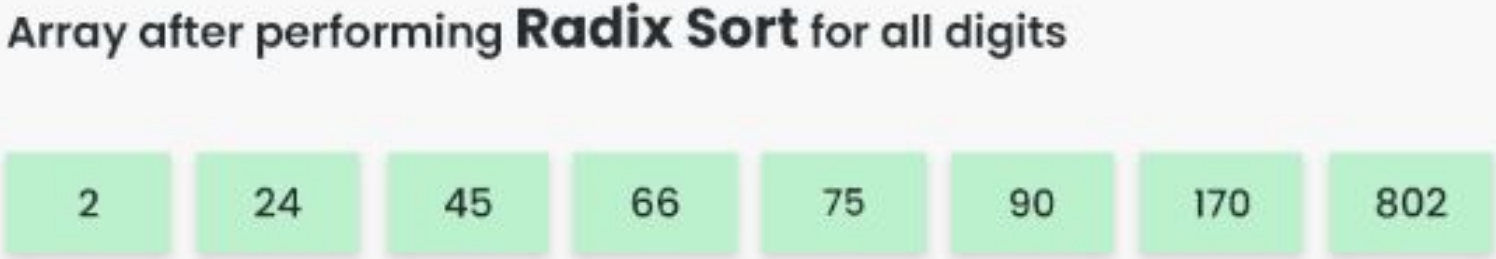
- *Perform counting sort on the array based on the hundreds place digits.*
- *The sorted array based on the hundreds place is [2, 24, 45, 66, 75, 90, 170, 802].*



Working of Radix Sort

Step 5: *The array is now sorted in ascending order.*

The final sorted array using radix sort is [2, 24, 45, 66, 75, 90, 170, 802].



Implementation of Radix Sort

```
def radixsort(A, size):
    high=max(A)
    pos=1
    b=[0] * size
    while(high//pos)>0:
        count = [0] * 10
        # add frequency
        for i in range(0,size):
            index=(A[i]//pos)%10
            count[index]+=1
        #get positions
        for j in range(1,len(count)):
            count[j]= count[j]+count[j-1]
        #get index to add in A
        #for loop to go from last index 1st index(0)
        for k in range(size-1,-1,-1):
            index = (A[k]//pos)%10
            count[index] -=1
            b[count[index]] = A[k]
        #copy elements of list b into original array
        for m in range(len(b)):
            A[m] = b[m]
        pos*=10
```

Quick Sort - Time & Space complexity

Case	Time complexity
Best	O(nk)
Average	O(nk)
Worst	O(nk)

Space Complexity	O(n+k)
Stable	Yes

- ***Stable sorting*** *maintains the position of two equals elements relative to one another.*
- ***Unstable sorting*** *does not maintain the position of two equals elements relative to one another.*

Sorting Algorithm	Time Complexity	Space complexity
Bubble sort	$O(n^2)$	O(1)
Selection sort	$O(n^2)$	O(1)
Insertion sort	$O(n^2)$	O(1)
Merge sort	O(nlogn)	O(n)
Quick sort	$O(n^2)$	O(n)
Radix sort	O(nk)	O(n+k)

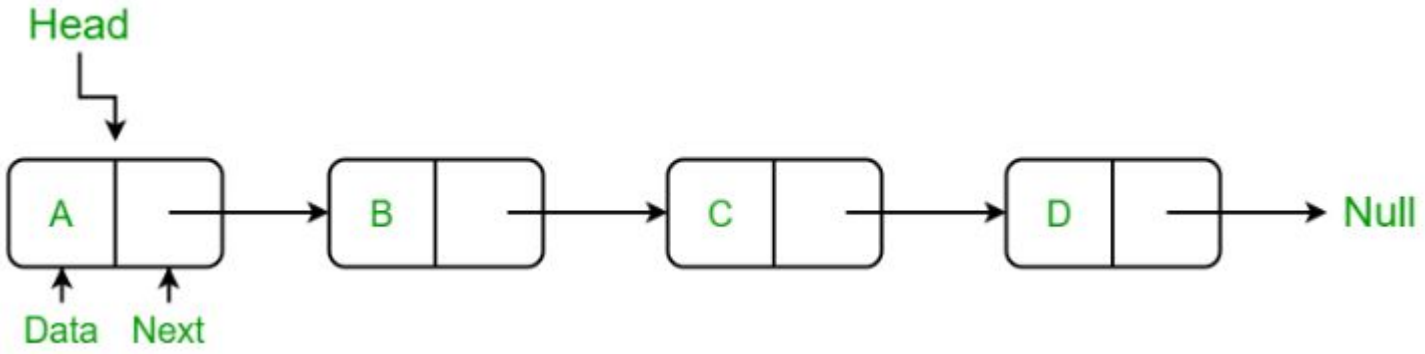
Time Complexity	Space Complexity
Calculates the time required	Estimates the space (memory) required
Time is counted for all statement	Memory space is counted for all variables, inputs and outputs.
The size of the input data is the primary determinant	Primary determinant is the auxiliary variable size
Deals with the computational time with the change in the size of the input	Deals with how much (extra) space would be required with a change in the input size.

Linked Structures

Unit 2

Intro to Linked Lists

- A Linked List is a linear data structure, in which the elements (data) are not stored at contiguous (Sequential) memory location.
- The elements in a linked list are linked using pointer (address).
- Linked list is a very commonly used data structure which consists of group of nodes.
- Each node has some information. A node contains 2 fields, 1st field is used to store data and 2nd field is used to store address of next node.



Applications of linked list in real world

- **Image viewer** – Previous and next images are linked, hence can be accessed by next and previous button.
- **Previous and next page in web browser** – We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list.
- **Music Player** – Songs in music player are linked to previous and next song. You can play songs either from starting or ending of the list.

Advantages of Linked List

- **Dynamic data structure**: A linked list is a dynamic arrangement so it can grow and shrink at runtime by allocating and deallocating memory. So there is no need to give the initial size of the linked list.
- **No memory wastage:** In the Linked list, efficient memory utilization can be achieved since the size of the linked list increase or decrease at run time so there is no memory wastage and there is no need to pre-allocate the memory.
- **Implementation**: Linear data structures like stack and queues are often easily implemented using a linked list.
- **Insertion and Deletion Operations**: Insertion and deletion operations are quite easier in the linked list. There is no need to shift elements after the insertion or deletion of an element only the address present in the next pointer needs to be updated.

Disadvantages of Linked List

- **Memory usage**: More memory is required in the linked list as compared to an array. Because in a linked list, a pointer is also required to store the address of the next element and it requires extra memory for itself.
- **Traversal**: In a Linked list traversal is more time-consuming as compared to an array. Direct access to an element is not possible in a linked list as in an array by index. For example, for accessing a node at position n, one has to traverse all the nodes before it.
- **Random Access:** Random access is not possible in a linked list due to its dynamic memory allocation.

ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

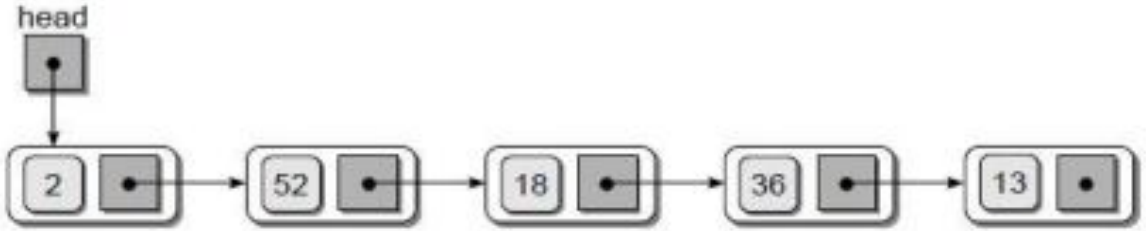
Types of Linked List

There are mainly three types of linked lists:

1. Singly Linked List/One Way Linked List
2. Doubly Linked List/Two Way Linked List
3. Circular Linked List

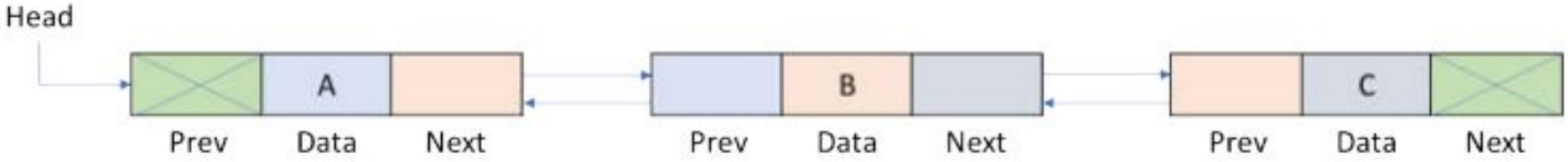
1. Singly Linked List/One Way Linked List

- In Singly linked list each node has 2 fields one for data and other is for address of next node
- Here, we can only traverse in one direction due to the linking of every node to its next node.
- Head node will always store the address of first node.
- Link/Next field of last node will always be NULL because it won't point to any other node.
- The pictorial representation of a singly linked list consisting of some items is shown in the figure



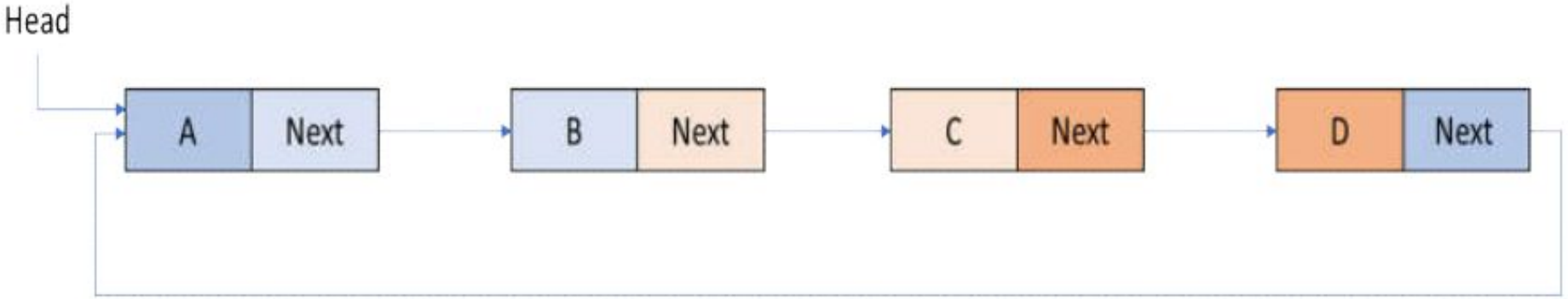
2. Doubly Linked List/Two Way Linked List

- In doubly linked list each node has 3 fields one for data and other two for address of previous and next node.
- Here, we can traverse in both directions.
- Head node will always store the address of first node.
- Previous field of first node and Next field of last node will always be NULL in doubly linked list.



3. Circular Linked List

- A circular linked list is that in which the last node contains the pointer/address to the first node of the list.



Linked List Implementation

Suppose we have a basic class containing a single data field

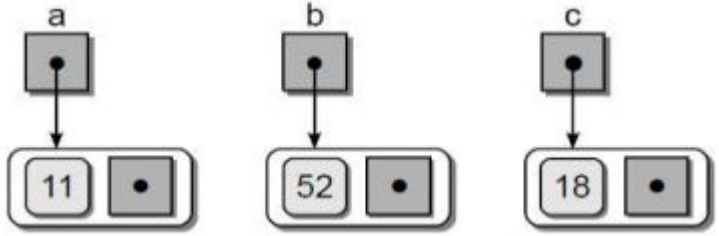
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

We can create several instances of this class, each storing data of our choosing. In the following example, we create three instances, each storing an integer value:

a = ListNode(11)

b = ListNode(52)

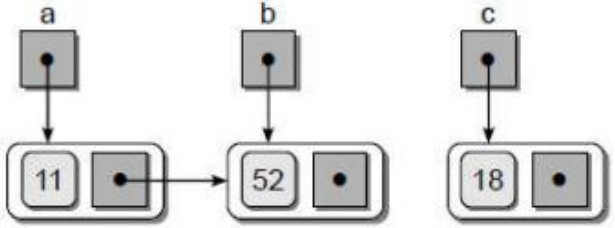
c = ListNode(18)



Linked List Implementation

Since the next field can contain a reference to any type of object, we can assign to it a reference to one of the other ListNode objects. For example, suppose we assign b to the next eld of object a:

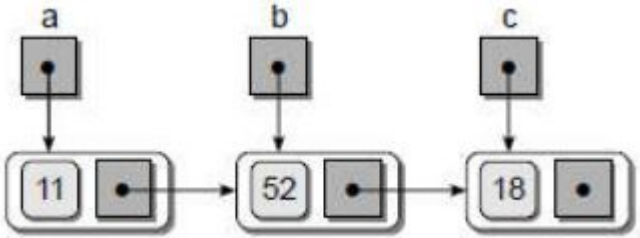
a.next = b



which output in object a being linked to object b, as shown

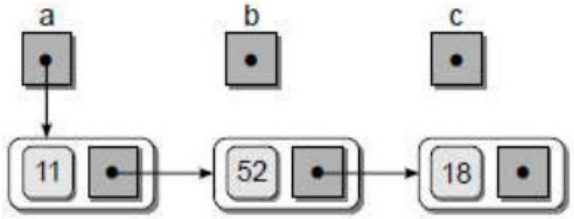
And at the end, link object b to object c

b.next = c



Linked List Implementation

We can remove the two external references b and c by assigning None to each, as shown here:



The result is a linked list structure. The two objects previously pointed to by b and c are still accessible via a. For example, suppose we wanted to print the values of the three objects. We can access the other two objects through the next field of the first object:

```
print(a.data )

print(a.next.data )

print(a.next.next.data )
```

Singly Linked List Python Code

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

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

    def insertatBeg(self, data):
        new_node = Node(data)
        if  self.head == None:
            self.head = new_node
        else:
            new_node.next = self.head
            self.head = new_node
```

```
def insertatEnd(self, data):
    new_node = Node(data)
    if  self.head == None:
        self.head = new_node
    else:
        curr_node = self.head
        while curr_node.next:
            curr_node = curr_node.next
        curr_node.next = new_node
```

Singly Linked List Python Code

```
def insertAtIndex(self, data, index):
    new_node = Node(data)
    curr_node = self.head
    position = 0
    if position == index:
        self.insertAtBegin(data)
    else:
        while(curr_node != None and position+1 != index):
            position = position+1
            curr_node = curr_node.next

        if curr_node != None:
            new_node.next = curr_node.next
            curr_node.next = new_node
        else:
            print("Index not present")
```

Singly Linked List Python Code

```
def removeBeg(self):
    if self.head != None:
        self.head = self.head.next
    else:
        print("No nodes to delete")
```

```
def rem_end(self):
    if self.head != None:
        curr_node= self.head
        if curr_node.next == None:
            self.head =None
        else:
            while curr_node.next.next:
                curr_node =curr_node.next
            curr_node.next = None
    else:
        print("nonodes to print")
```

```
def removeatIndex(self, index):
    if self.head != None:
        curr_node =self.head
        pos = 0
        if pos == index:
            self.removeBeg()
        else:
            while curr_node != None and pos+1 != index:
                pos = pos + 1
                curr_node = curr_node.next
            if curr_node != None:
                curr_node.next = curr_node.next.next
            else:
                print("Index not present")
    else:
        print("No nodes to delete")
```

Singly Linked List Python Code

```python
def printll(self):
    curr_node = self.head
    while curr_node:
        print(curr_node.data)
        curr_node =curr_node.next

def sizeofLL(self):
    size = 0
    if self.head:
        curr_node = self.head
        while curr_node:
            size += 1
            curr_node = curr_node.next
    print(f"Size of Linked List is {size}")
```

```python
def searchlist(self,data):
    if self.head:
        pos=0
        curr_node = self.head
        while curr_node:
            if curr_node.data == data:
                print(f"Data found at index {pos}")
                break
            else:
                pos+=1
                curr_node = curr_node.next
        else:
            print("Data not found")
    else:
        print("No nodes to search")
```

STACK

- Stack is a linear data structure and it is an ordered collection of items.
- All the items of stack are inserted and removed from the same end and that end is known as TOP of the stack.
- Random access of items are not possible in stack.
- Every time an element is added, it goes on the top of the stack and the only element that can be removed is the element that is at the top of the stack.
- Stack follow a particular order in which the operations (insertion/deletion of elements) are performed. The order is LIFO(Last In First Out) or FILO(First In Last Out) which means the item which is inserted first will be removed out at last.

Stack Operations

1. **Stack():** Creates a new (empty) stack.
2. **isEmpty()**: Returns a Boolean value if the stack is empty.
3. **length ()**: Returns the number of elements in the stack.
4. **pop():** Removes and returns the top element of the stack, if the stack is not empty. Items cannot be removed from an empty stack. The next item on the stack becomes the new top item.

   Stack under- flow:

   When elements are being deleted, there is a possibility of stack being empty. When stack is empty, it is not possible to delete any item. Trying to delete an element from an empty stack results in stack underflow.

Stack Operations

5. peek (): Use as a reference to the item on top of a non-empty stack without removing it. Peeking, which cannot be done on an empty stack, does not modify the stack contents.

6. push (element): Adds the given element to the top of the stack.

Stack overflow :

When elements are being inserted, there is a possibility of stack being full. Once the stack is full, it is not possible to insert any element. Trying to insert an element, even when the stack is full results in overflow of stack.

Implementing Stacks- Python Lists

```
def push(stk,item):
    stk.append(item)
    top = len(stk)-1

def pop(stk):
    if isEmpty(stk):
        print("Underflow")
    else:
        item=stk.pop()
        if len(stk)==0:
            top = None
        else:
            top= len(stk)
            print("Popped item is ",item)
```

```
def isEmpty(stk):
    if stk == []:
        return True
    else:
        return False

def display(stk):
    if isEmpty(stk):
        print("Stack is Empty")
    else:
        top = len(stk)-1
        print("Elements in stack are")
        for i in range(top,-1,-1):
            print(stk[i])
```

Implementing Stacks- Linked Lists

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


class stack:
    def __init__(self):
        self.top = None

    def push(self, data):
        new_node = StackNode(data)
        if  self.top == None:
            self.top = new_node
        else:
            new_node.next = self.top
            self.top = new_node
```

```
def pop(self):
    if self.top != None:
        print(f"Popped element {self.top.data}")
        self.top = self.top.next
    else:
        print("Stack Underflow")

def printstack(self):
    curr_node = self.top
    while curr_node:
        print(curr_node.data)
        curr_node =curr_node.next
def peek(self):
    print(f"Top of stack is {self.top.data}")

def isEmpty(self):
    if self.top is None:
        print("Stack is Empty")
    else:
        print("Stack is not Empty")
```

Notations

The way to write arithmetic expression is known as a notation. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are:

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

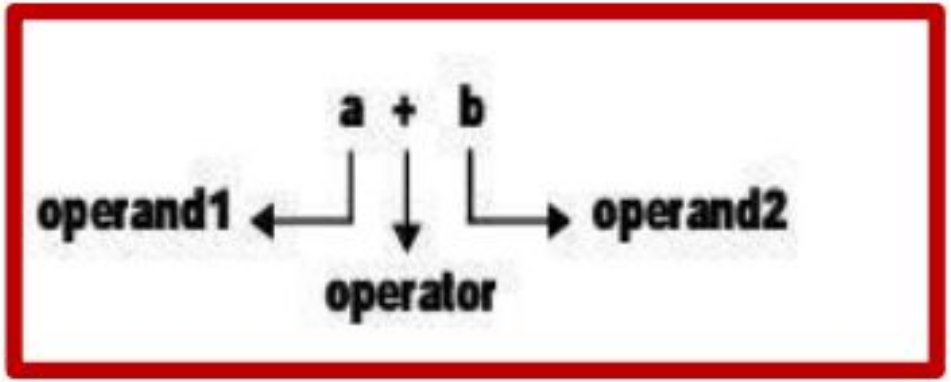
Notations

- **Infix Notation:**

1. In an expression, if an operator is in between two operands, the expression is called an infix expression.
2. The infix expression can be parenthesized or un-parenthesized. For example,

➔ a + b is an un-parenthesized infix expression

➔ (a + b) is a parenthesized infix expression

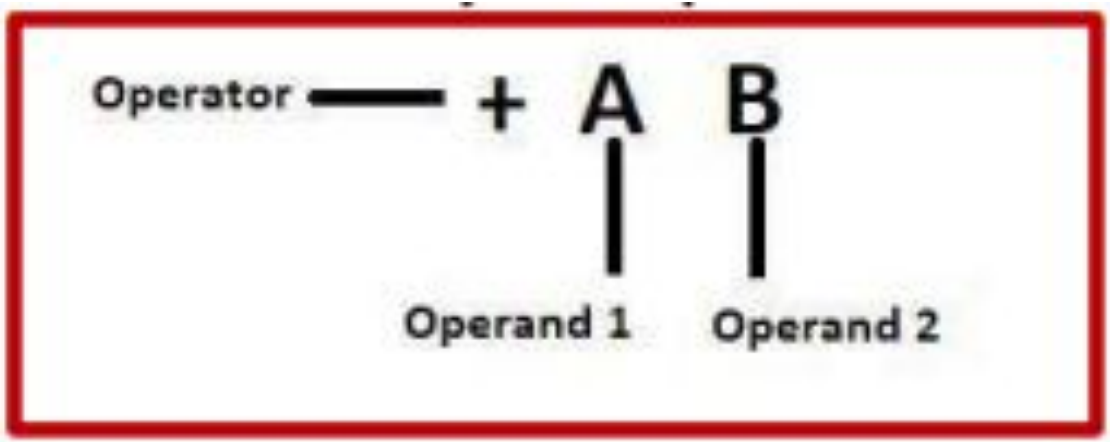


Notations

- **Prefix (Polish) Notation**

In an expression, If an operator comes before the operands, the expression is called a prefix expression.

- + A B its an prefix expression
- - 20 10 it's an prefix expression



Notations

- **Postfix (Reverse-Polish) Notation**

In an expression, If an operator comes after the operands, the expression is called postfix expression.

- A B + its an postfix expression
- 100 10 / its an postfix expression

Precedence and Associativity of operator

Precedence of operator:

- The precedence of operators determines which operator is executed first if there is more than one operator in an expression.
- Precedence means priority. Each and every operator has some priority and according to the priority only expressions are executed.

Associativity of Operator:

Operators Associativity is used when two operators of same precedence appear in an expression.

Associativity can be either Left to Right or Right to Left.

Precedence and Associativity of operator

Operator	Precedence/Priority	Associativity
( ), [ ]	1 (Highest)	L -> R
^(exponent/power)	2	L -> R
*	3	L -> R
/		L -> R
%		L -> R
+	4	L -> R
-		L -> R
=	5	R-> L

Stack Applications- Conversion of expressions (Infix to postfix)

1. Scan the Infix Expression from left to right and repeat Step 2 to 5 for each element of infix expression until the Stack is empty.
2. If incoming symbol is an operand , add it to the postfix expression.
3. If incoming symbol is a left parenthesis , push it onto Stack.
4. If incoming symbol is an operator ,then:
    - If scanned operator (incoming operator) precedence is higher than the operator in the stack then push the incoming operator in the stack.
    - If scanned operator (incoming operator) precedence is lower or same than the operator in the stack then pop the operator of the stack and again compare the incoming operator with stack operator.

Stack Applications- Conversion of expressions (Infix to postfix)

5. If a right parenthesis is encountered ,then:

- Repeatedly pop from Stack and add to postfix expression each operator (on the top of Stack) until a left parenthesis is encountered.
- Remove the left Parenthesis.

EXAMPLE 1: CONVERT THE BELOW INFIX TO POSTFIX

INFIX: 4 * 3 + 2 - 5

SOLUTION:

POSTFIX: 4 3 * 2 + 5 -

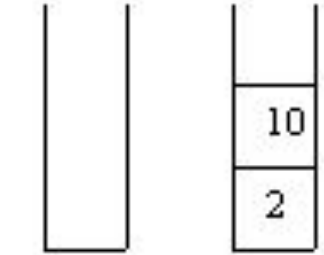
Input/scan	Stack	Postfix expression
4	empty	4
*	*	4
3	*	4 3
+	+	4 3 *
2	+	4 3 * 2
-	-	4 3 * 2 +
5	-	4 3 * 2 + 5
	Stack is empty	4 3 * 2 + 5-

Stack Applications- Evaluation of postfix expressions

- Create a stack to store operands (or values).
- Scan the given expression from left to right and do the following for every scanned element.
  - If the element is a number, push it into the stack
  - If the element is an operator, pop operands for the operator from the stack. Evaluate the operator and push the result back to the stack
- When the expression is ended, the number in the stack is the final answer.

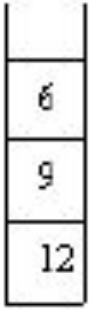
2 10 + 9 6 - /

push 2
push 10



pop 10
pop 2
push 2 + 10 = 12

push 9
push 6



pop 6
pop 9
push 9 - 6 = 3

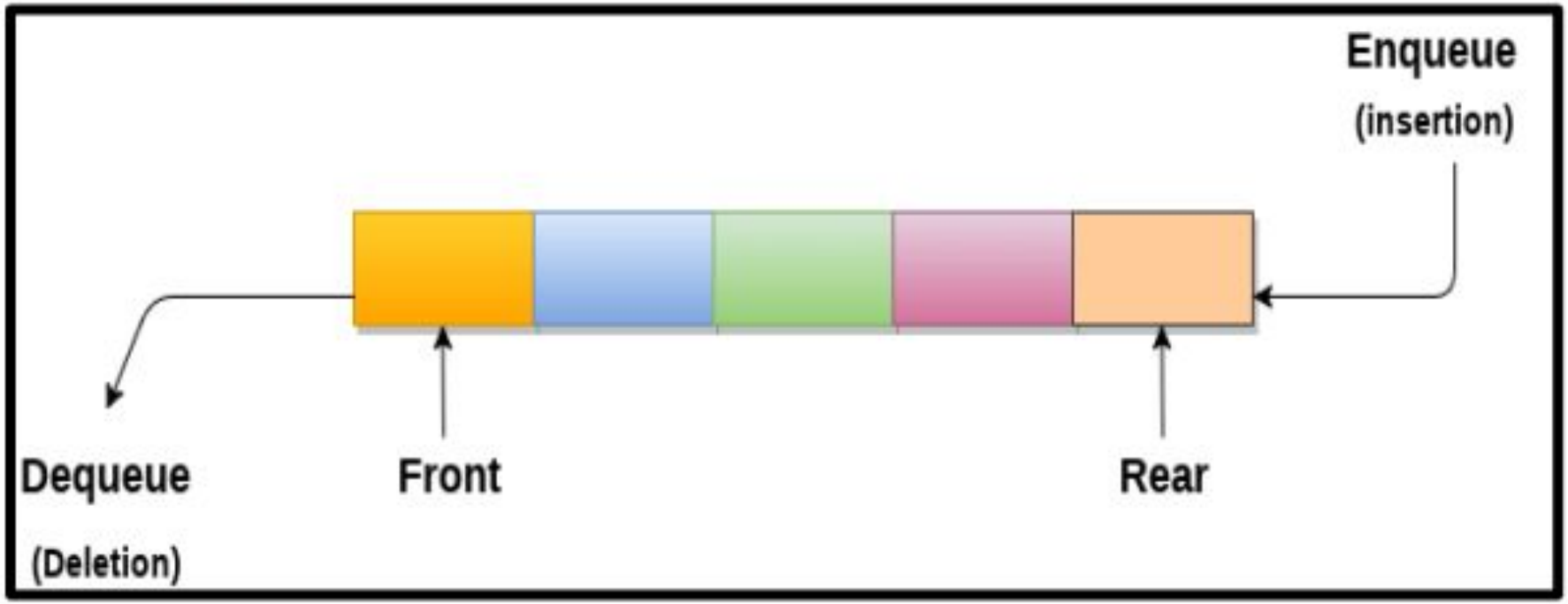
pop 3
pop 12
push 12 / 3 = 4

pop answer: 4

QUEUE

1. A Queue is a linear structure where elements are inserted from one end and elements are deleted from the other end.
2. The end from where the elements are inserted is called rear end and the end from where elements are deleted is called front end.
3. Queue follow principle of FIFO i.e. First In First Out, which means that element inserted first will be removed first.
4. In programming terms, putting an item in the queue is called an "enqueue" and removing an item from the queue is called "dequeue".
5. Queue has 2 ends “FRONT” & “REAR”.
6. “FRONT” end is used for element deletion and “REAR” end is used for element insertion.
7. For example. At railway station people stand in a queue (line) to get tickets and the person who will be standing first in queue will get the tickets first and also moved out first from the queue.

Pictorial Representation:



Applications of Queue

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.

2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same

rate between two processes) for eg. pipes, file IO, sockets.

3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.

4. Queues are used to maintain the playlist in media players in order to add and remove the songs from the playlist.

5. Queues are used in operating systems for handling interrupts.

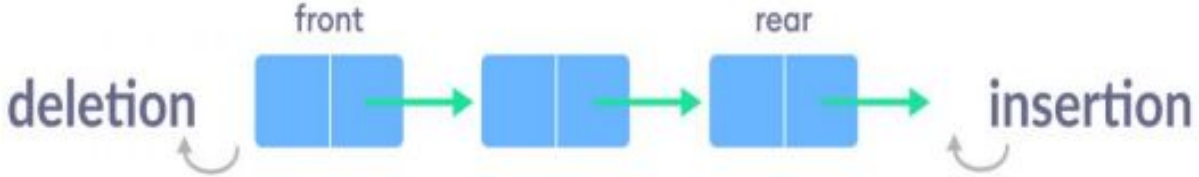
Basic Operations of Queue

- 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

Types of Queue

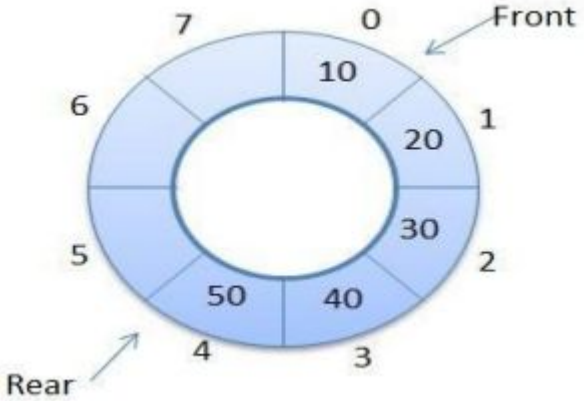
- **Linear Queue**

In a simple queue, insertion takes place at the rear end and removal occurs at the front end. It strictly follows FIFO rules.



- **Circular Queue**

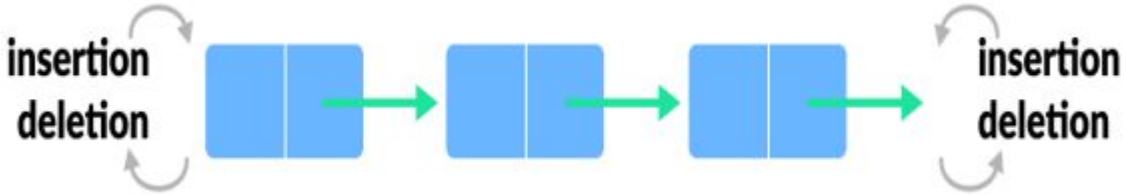
Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called 'Ring Buffer'



Types of Queue

- **Double Ended Queue**

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).



Implementation of Queue- Python List

```
def enqueue(q,item):
    q.append(item)

def dequeue(q):
    if isEmpty(q):
        print("Queue is empty")
    else:
        print(f"Popped item is {q.pop(0)}")

def display(q):
    for item in q:
        print(item,end=" ")

def isEmpty(q):
    if not q:
        return True
    else:
        return False
```

Implementation of Queue- Linked List

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

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

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

    def EnQueue(self, item):
        temp = Node(item)

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

```
def DeQueue(self):
    if self.isEmpty():
        return
    temp = self.front
    self.front = temp.next

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

def display(self):
    if self.front:
        curitem= self.front
        while curitem:
            print(curitem.data)
            curitem = curitem.next
```

PRIORITY QUEUE — ABSTRACT DATA TYPE

- Priority Queue is an Abstract Data Type (ADT) that holds a collection of elements, it is similar to a normal Queue, the difference is that the elements will be dequeued following a priority order.
- Priority Queue is an extension of queue with following properties:

1. Every item has a priority associated with it.
2. An element with high priority is dequeued before an element with low priority.
3. If two elements have the same priority, they are served according to their order in the queue.

PRIORITY QUEUE — ABSTRACT DATA TYPE

- A real-life example of a priority queue would be a hospital queue where the patient with the most critical situation would be the first in the queue. In this case, the priority order is the situation of each patient.
- Atypical priority queue supports following operations.

1. **insert (item, priority)**: Inserts an item with given priority.
2. **getHighestPriority ():** Returns the highest priority item.
3. **deleteHighestPriority ()**: Removes the highest priority item.

Applications of Priority Queue:

1. CPU Scheduling
2. Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc
3. All queue applications where priority is involved.

Bounded Priority Queue

- Bounded Queues are queues which are bounded by capacity that means we need to provide the max size of the queue at the time of creation
- A Bounded Priority Queue implements a priority queue with an upper bound on the number of elements.
- If the queue is not full, added elements are always added.
- If the queue is full and the added element is greater than the smallest element in the queue, the smallest element is removed and the new element is added.
- If the queue is full and the added element is not greater than the smallest element in the queue, the new element is not added.

Unbounded Priority Queues

- Unbounded Queues are queues which are NOT bounded by capacity that means we should not provide the size of the queue. For example, LinkedList
- An unbounded priority queue based on a priority heap.
- The elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.

RECURSION

UNIT 3

Recursive Functions

- Recursion is a process for solving problems by subdividing a larger problem into smaller cases of problem itself and then solving the smaller parts.
- Recursion is a powerful programming and problem solving tool.

Recursive Function

- In programming terms a recursive function can be defined as a routine that calls itself directly or indirectly.

Properties of Recursion

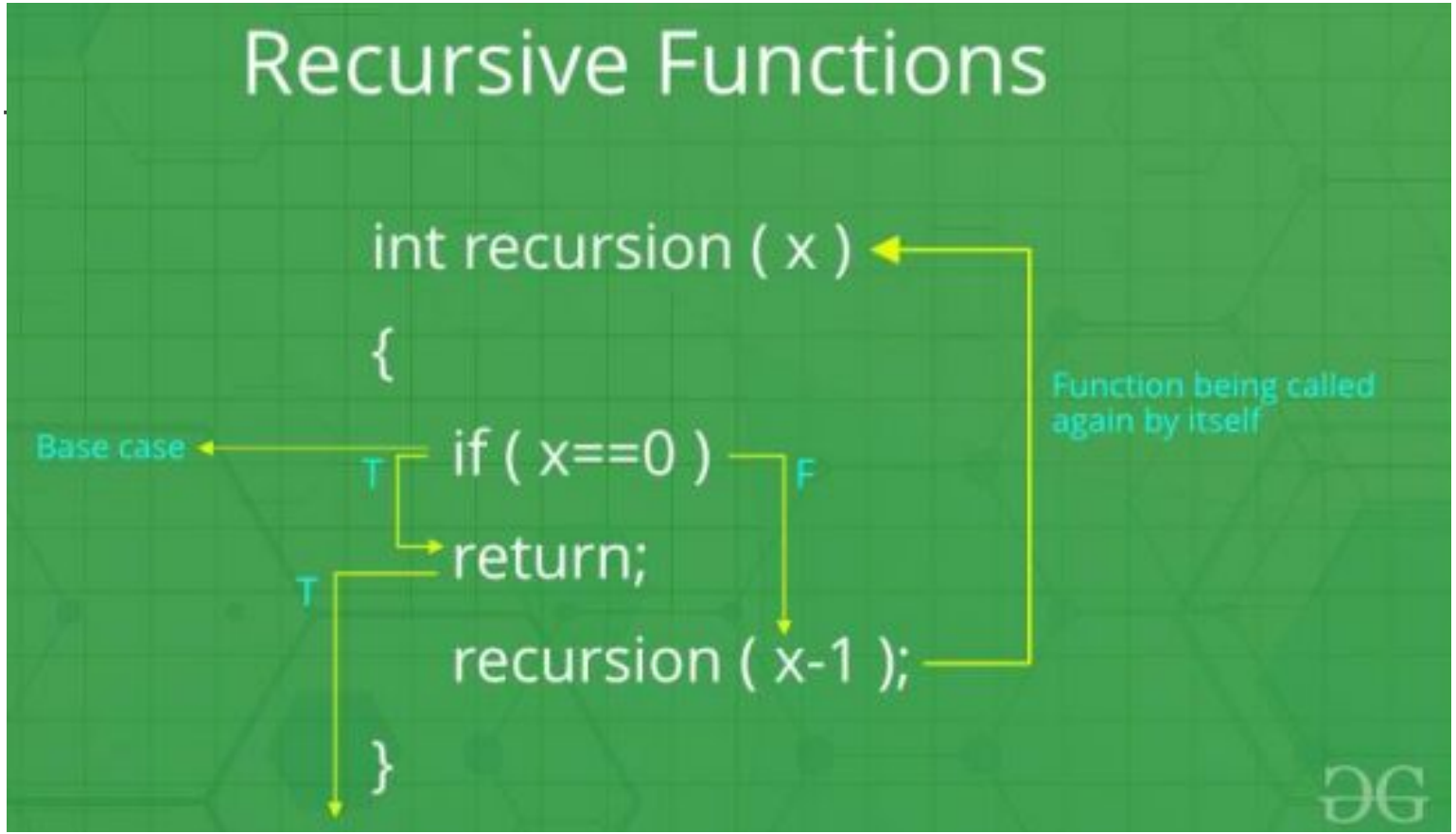
All recursive solution must satisfy three rules or properties:

1) A recursive solution must contain a base case.
2) A recursive solution must contain a recursive case.
3) A recursive solution must make progress toward the base case.

Note:

Base case: It is a termination point for a recursive function. Every recursive program must have base case to make sure that the function will terminate. Missing base case results in unexpected behaviour.

Recursive Function



Recursive Function

- In programming terms a recursive function can be defined as a routine that calls itself directly or indirectly.

Properties of Recursion

All recursive solution must satisfy three rules or properties:

1) A recursive solution must contain a base case.
2) A recursive solution must contain a recursive case.
3) A recursive solution must make progress toward the base case.

Note:

Base case: It is a termination point for a recursive function. Every recursive program must have base case to make sure that the function will terminate. Missing base case results in unexpected behaviour.

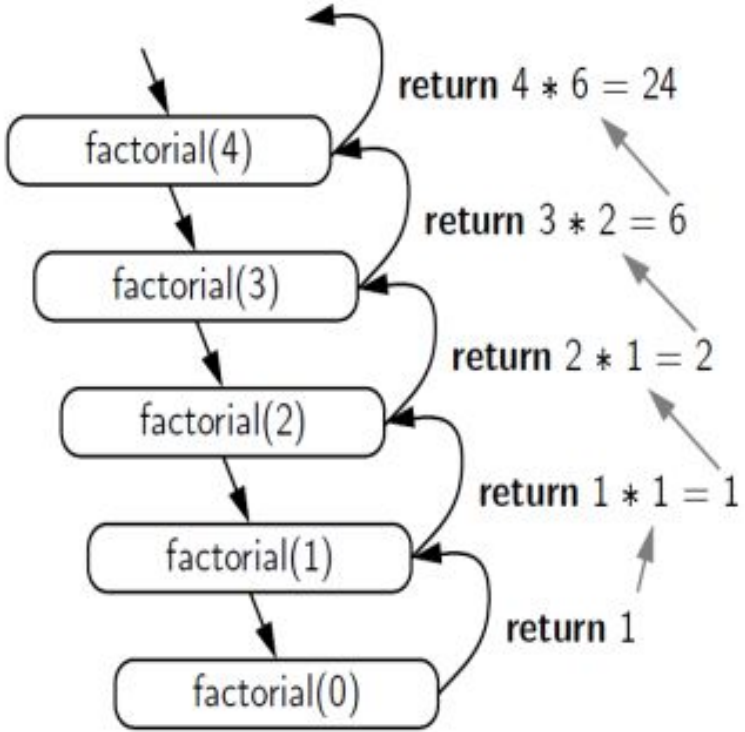
Recursive Function Examples- Factorial and Fibonacci

```
def factorial(n):
    if n == 1:
        return n
    else:
        return n*factorial(n-1)

print(factorial(5))
```

```
def fibo(n):
    if n <= 1:
        return n
    else:
        return(fibo(n-1) + fibo(n-2))

n=10
for i in range(n):
    print(fibo(i))
```

Recursive call Trees

- When evaluating a recursive function, we typically use a recursive call tree such as for factorial function shown in figure.



Application Of Recursion

- Tree and graph traversal
- Sorting algorithms
- Divide-and-conquer algorithms
- Tower of Hanoi
- Fibonacci Numbers

BINARY TREE

Tree

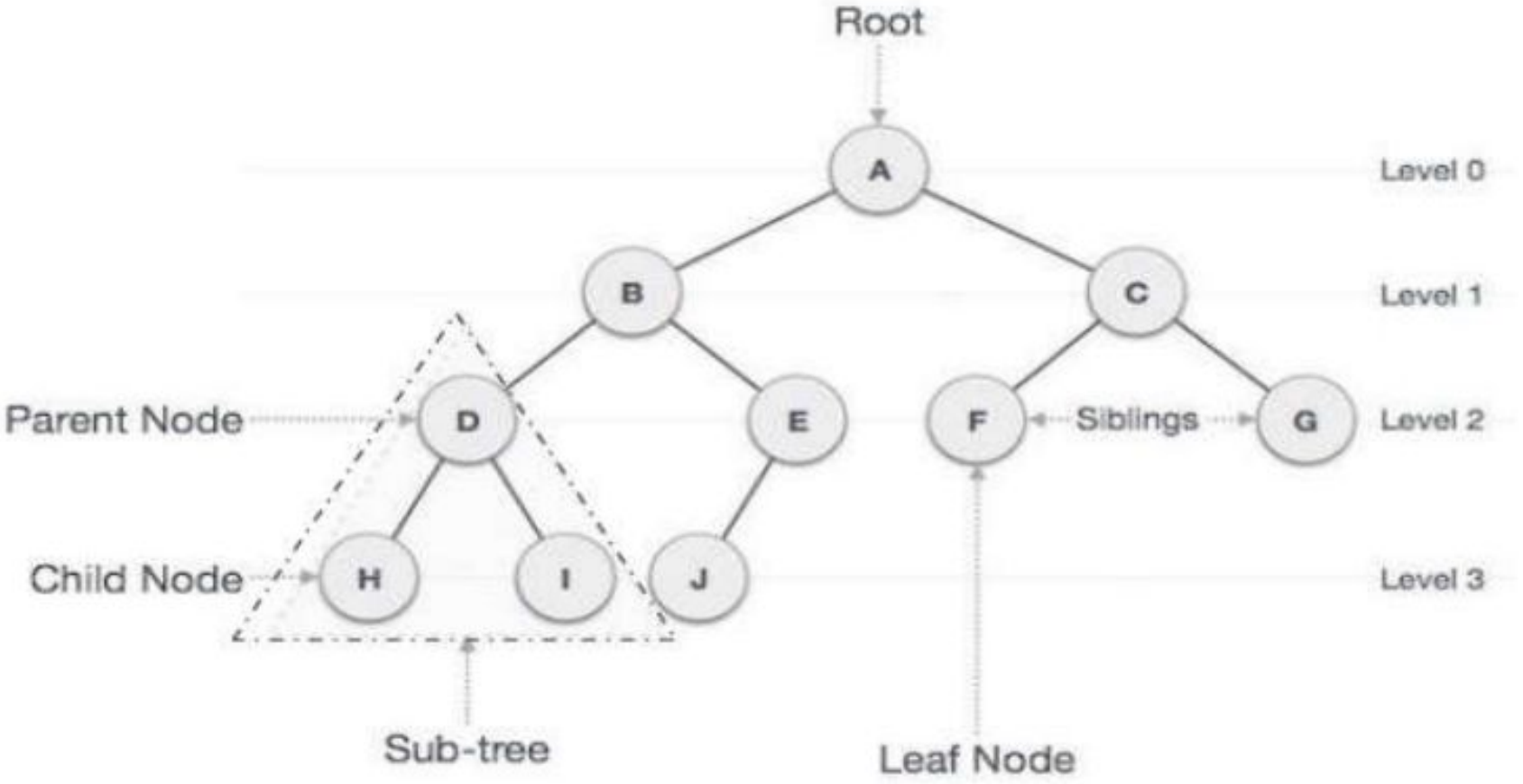
- Tree is a data structure in which nodes represent values and are connected via edges.
- It is non- linear data structure that is arranged in a hierarchical fashion.
- In the tree abstract data type, proper arrangements of elements are not important.
- A tree has the following properties:

  1. The tree has one node called root. The tree originates from this, and hence it does not have any parent.

  2. Each node has one parent only but can have multiple children.

  3. Each node is connected to its children via edge.

Tree


Root
A
B
C
D
E
F
G
H
I
J
Level 0
Level 1
Level 2
Level 3
Parent Node
Siblings
Child Node
Sub-tree
Leaf Node

Tree

Terminology	Description	Example From Diagram
Root	Root is a special node in a tree. The entire tree originates from it. It does not have a parent.	Node A
Parent Node	Parent node is an immediate predecessor of a node.	B is parent of D & E
Child Node	All immediate successors of a node are its children.	D & E are children of B
Leaf	Node which does not have any child is called as leaf	H, I, J, F and G are leaf nodes

Tree

Terminology	Description	Example From Diagram
Edge	Edge is a connection between one node to another. It is a line between two nodes or a node and a leaf.	Line between A & B is edge
Siblings	Nodes with the same parent are called Siblings.	D & E are siblings
Path / Traversing	Path is a number of successive edges from source node to destination node.	A – B – E – J is path from node A to E

Tree

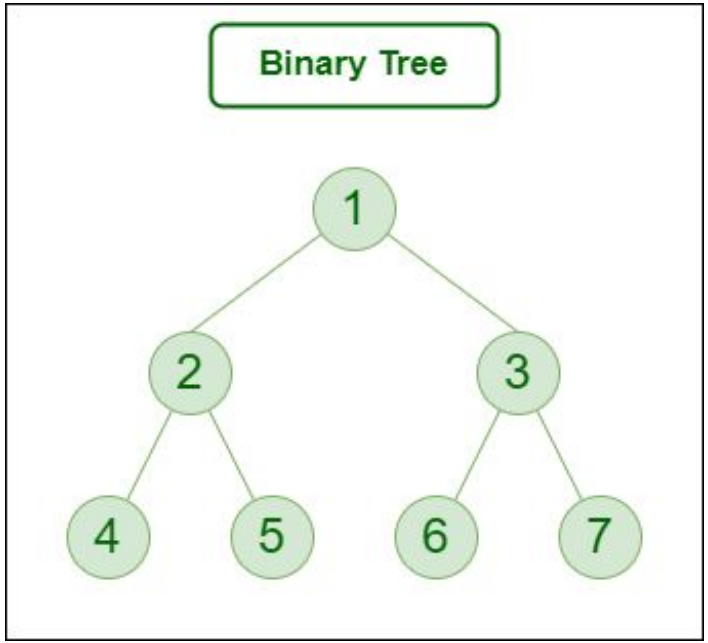
Terminology	Description	Example From Diagram
Height of Node	Height of a node represents the number of edges on the longest path between that node and a leaf.	A, B, C, D & E can have height. Height of A is no. of edges between A and H, as that is the longest path, which is 3. Height of C is 1
Levels of node	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	Level of H, I & J is 3. Level of D, E, F & G is 2

Tree

Terminology	Description	Example From Diagram
Degree of Node	Degree of a node represents the number of children of a node.	Degree of D is 2 and of E is 1
Sub tree	Descendants of a node represent subtree.	Nodes D, H, I represent one subtree.

BINARY TREE

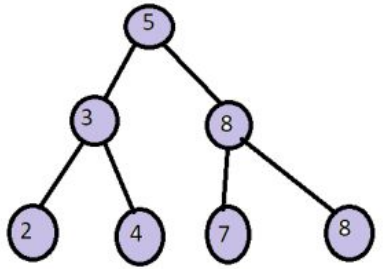
- Binary tree is an ordered tree if every node has zero, one or two child nodes.
- Each node should have at most two children in binary tree.
- Binary trees start with root node.
- Each child node in binary tree represented as left child or right child.
- The subtree rooted at left child is called as left subtree.
- The subtree rooted at right child is called as right subtree.

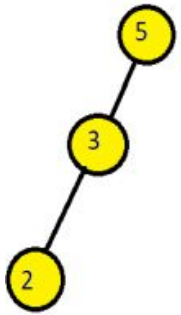


Properties of Binary Tree:

- If binary tree has height h, maximum number of nodes will be when levels are completely full. Total number of nodes will be $2^{(h+1)}-1$
- If the binary tree has height h, minimum number of nodes is h+1(in case of left skewed and right skewed tree.)
- The maximum number of nodes at level 'l' of a binary tree is 2^l
- In a non-empty binary tree, if n is the total number of nodes and e is the total number of edges,then e = n-1

$2^{(2+1)}-1 = 7$ nodes





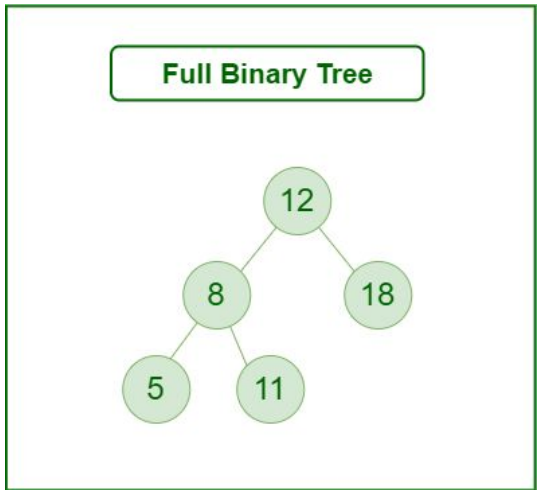
Binary tree shown in figure with height h has 3 nodes.

TYPES OF BINARY TREE

There are various types of binary trees, and each of these binary tree types has unique characteristics

- **Full Binary Tree:**

1. A Binary Tree is a full binary tree if every node has 0 or 2 children
2. It is also known as a proper binary tree.

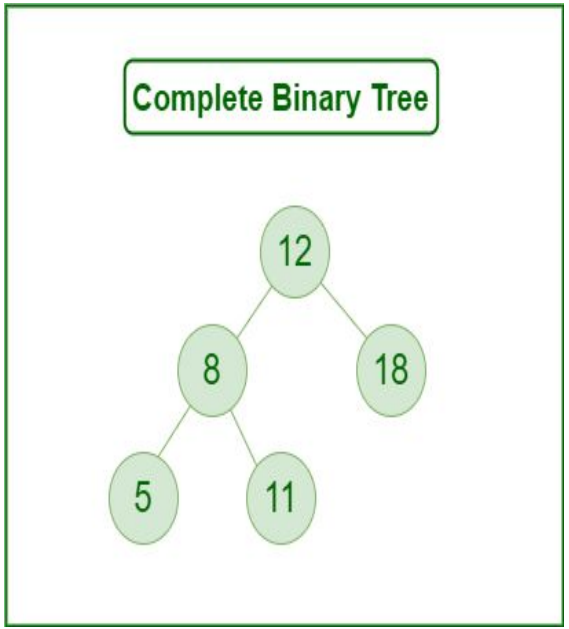


TYPES OF BINARY TREE

- **Complete Binary Tree**

1. A Binary Tree is a Complete Binary Tree i all the levels are completely filled except possibly the last level and the last level has all keys as left as possible.
2. A complete binary tree is just like a ful binary tree, but with two major differences:

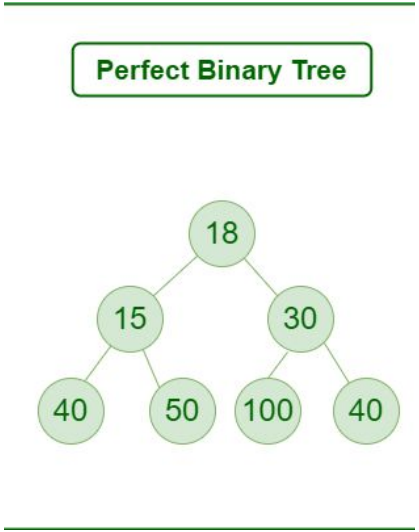
- Every level except the last level must be completely filled.
- All the leaf elements must lean towards the left.
- The last leaf element might not have a right sibling i.e. a complete binary tree doesn’t have to be a full binary tree.



TYPES OF BINARY TREE

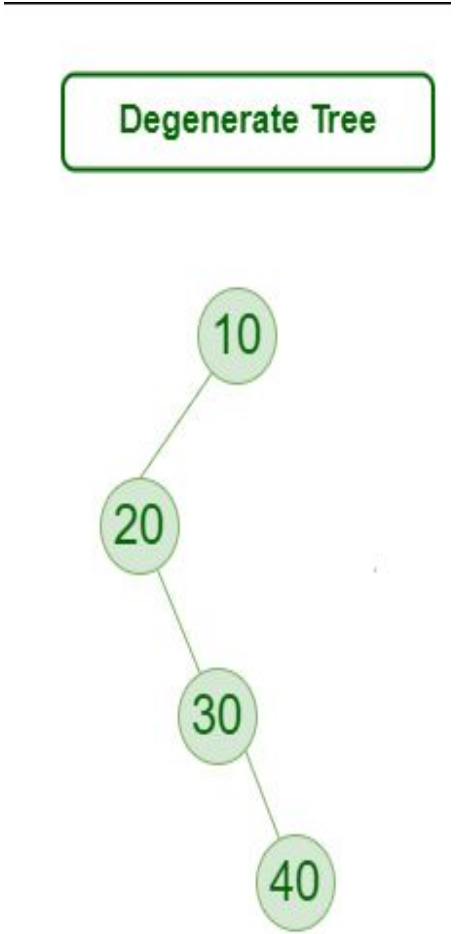
- **Perfect Binary Tree**

A Binary tree is a Perfect Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level.



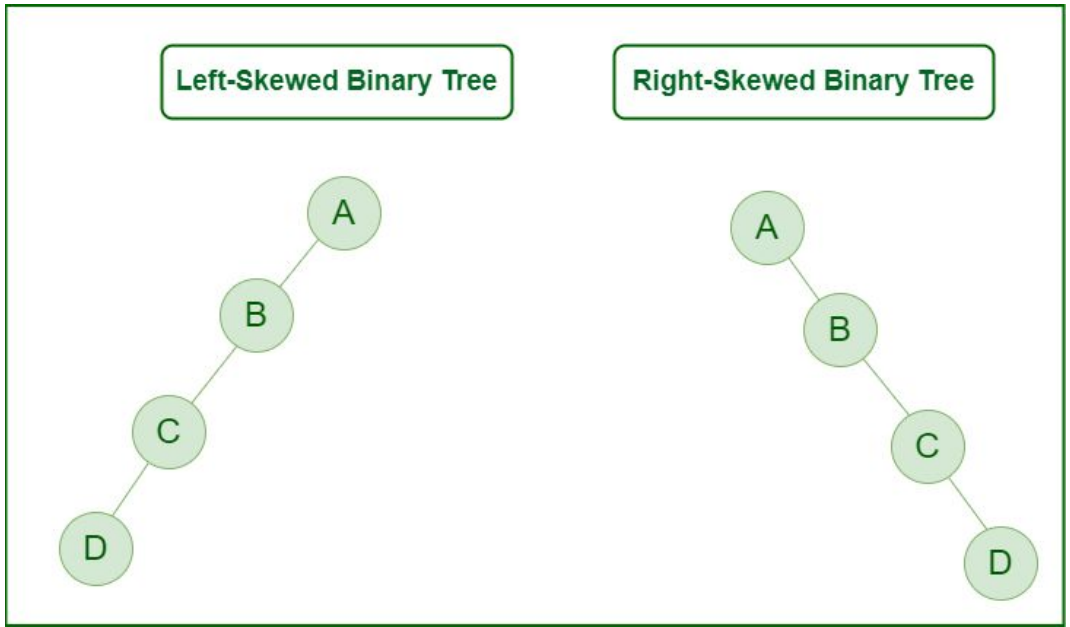
- **Degenerate Binary Tree**

1. A Tree where every internal node has one child.
2. Such trees are performance-wise same as linked list.
3. A degenerate or pathological tree is a tree having a single child either left or right.

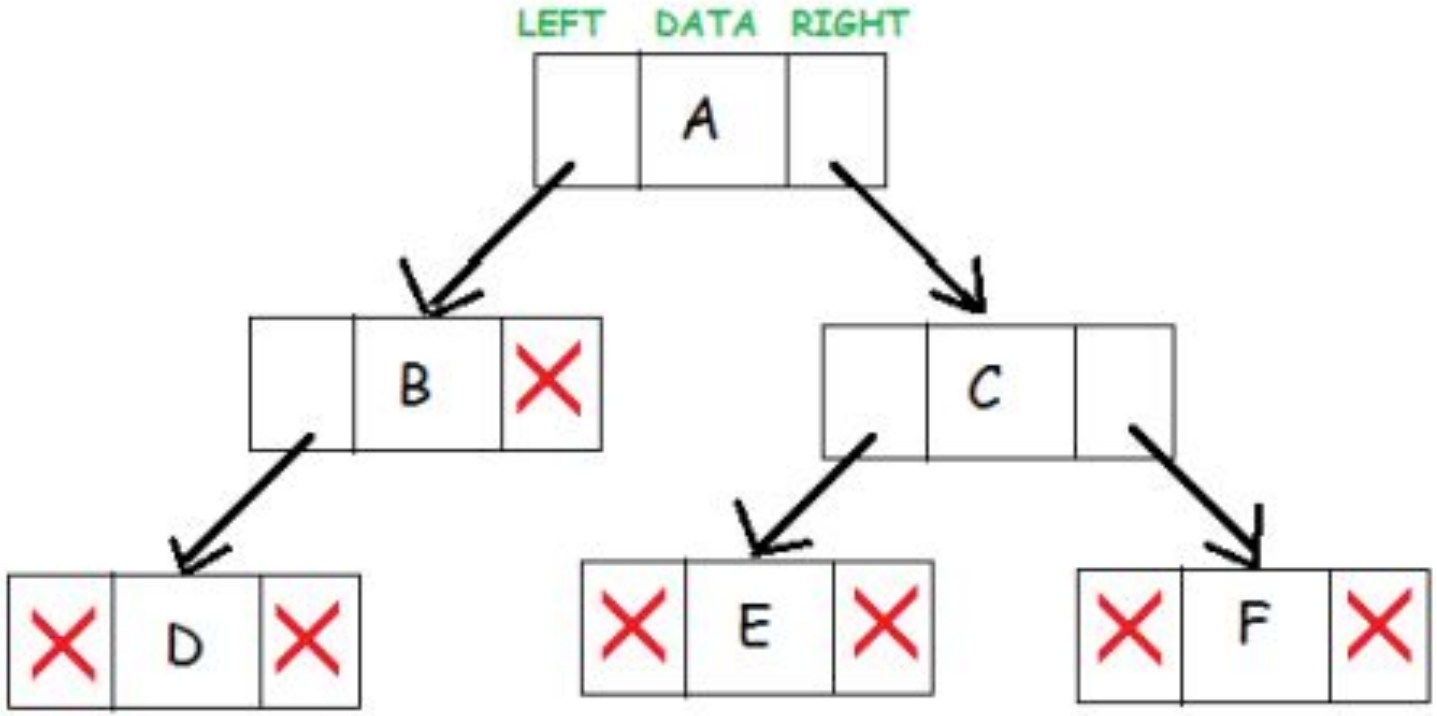


- **Skewed Binary Tree**

1. A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes.
2. Thus, there are two types of skewed binary tree: left-skewed binary tree and right-skewed binary tree.



Implementation of binary tree



Implementation of binary tree

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

root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
```

Tree Traversing Types

- Tree traversal means traversing or visiting each node of a tree.
- Linear data structures like Stack, Queue, and linked list have only one way for traversing, whereas the tree has various ways to traverse or visit each node.
- The following are the three different ways of traversal:

1. Inorder traversal(LMR)
2. Preorder traversal(MLR)
3. Postorder traversal(LRM)

1. **Inorder traversal(LMR)**

An inorder traversal is a traversal technique that follows the policy, i.e., Left Root Right.

2. **Preorder traversal(MLR)**

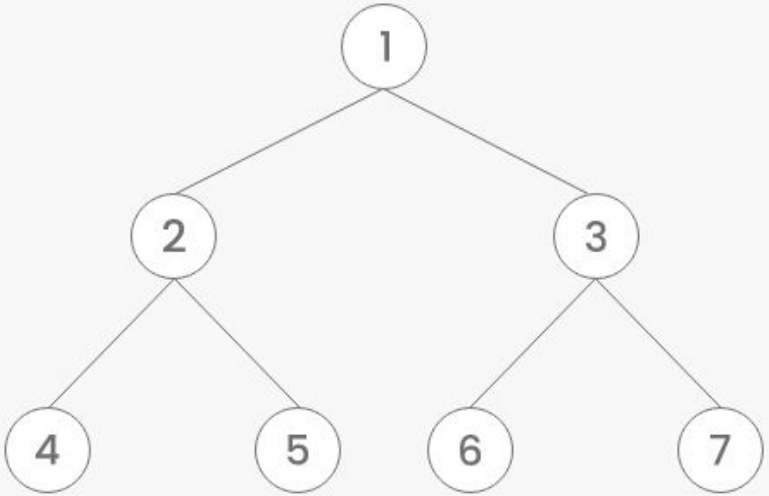
A preorder traversal is a traversal technique that follows the policy, i.e., Root Left Right.

3. **Postorder traversal(LRM)**

A Postorder traversal is a traversal technique that follows the policy, i.e., Left Right Root.

Tree Traversing Types

Tree Traversal Techniques



Inorder Traversal

4	2	5	1	6	3	7

Preorder Traversal

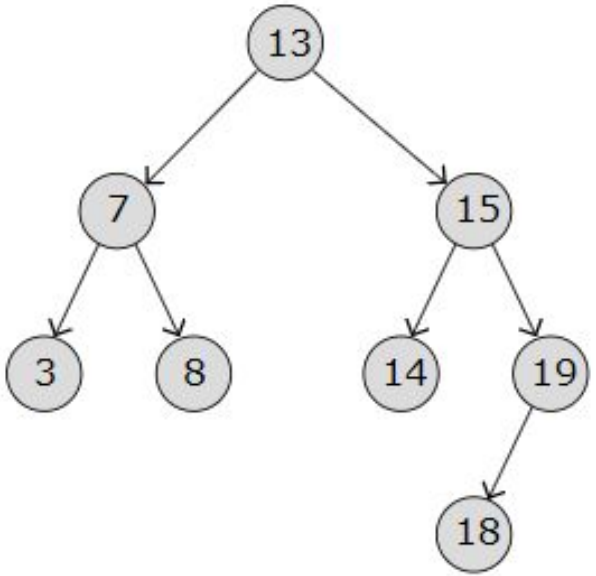
1	2	4	5	3	6	7

Postorder Traversal

4	5	2	6	7	3	1

Binary Search Tree

- A Binary Search Tree is a Binary Tree where every parent node's left child has a lower value, and every node's right child has a higher value.
- A clear advantage with Binary Search Trees is that operations like search, delete, and insert are fast and done without having to shift values in memory.



Implement Binary Search Tree

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

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

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

Implement Binary Search Tree

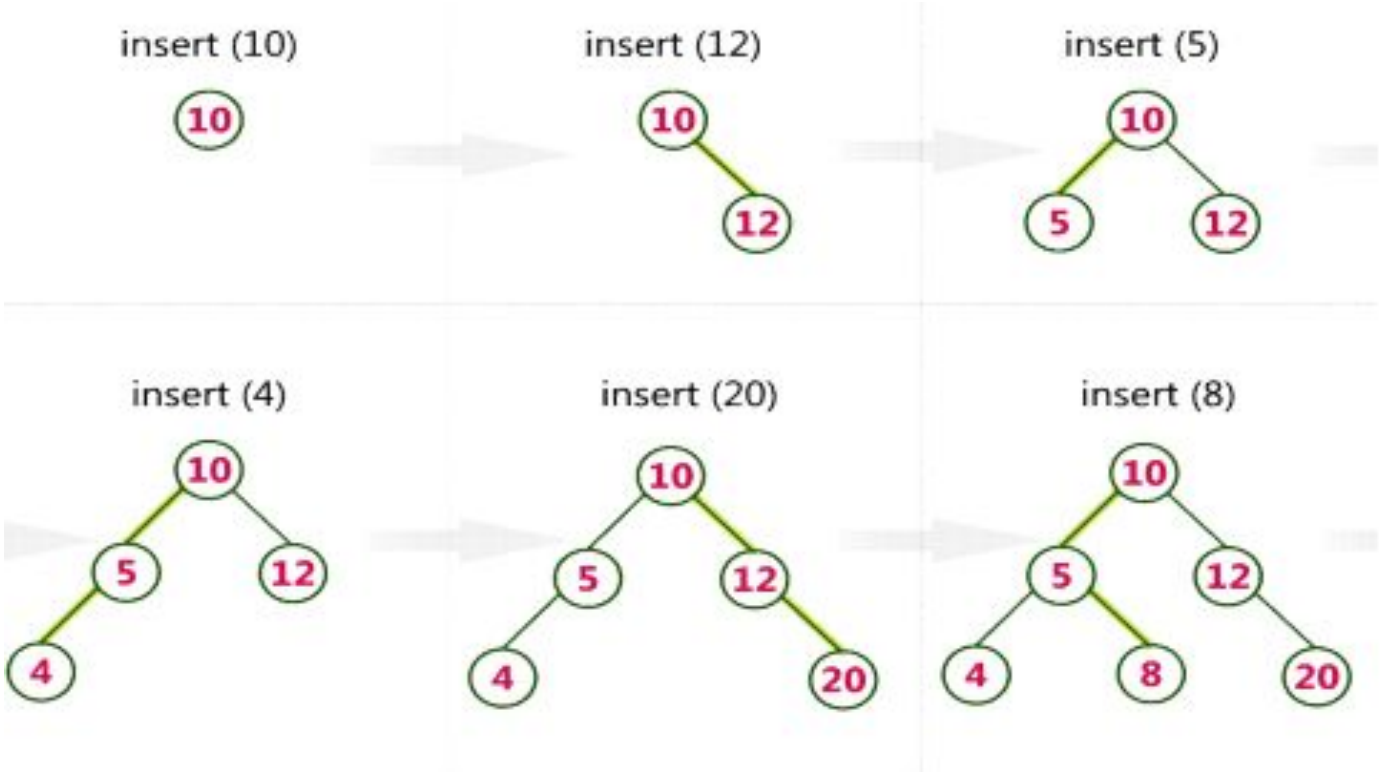
```
def search(node, target):
    if node is None:
        return None
    elif node.data == target:
        return node
    elif target < node.data:
        return search(node.left, target)
    else:
        return search(node.right, target)
```

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

```
def delete(node, data):
    if not node:
        return None

    if data < node.data:
        node.left = delete(node.left, data)
    elif data > node.data:
        node.right = delete(node.right, data)
    else:
        # Node with only one child or no child
        if not node.left:
            temp = node.right
            node = None
            return temp
        elif not node.right:
            temp = node.left
            node = None
            return temp

        # Node with two children, get the in-order successor
        node.data = minValueNode(node.right).data
        node.right = delete(node.right, node.data)

    return node
```

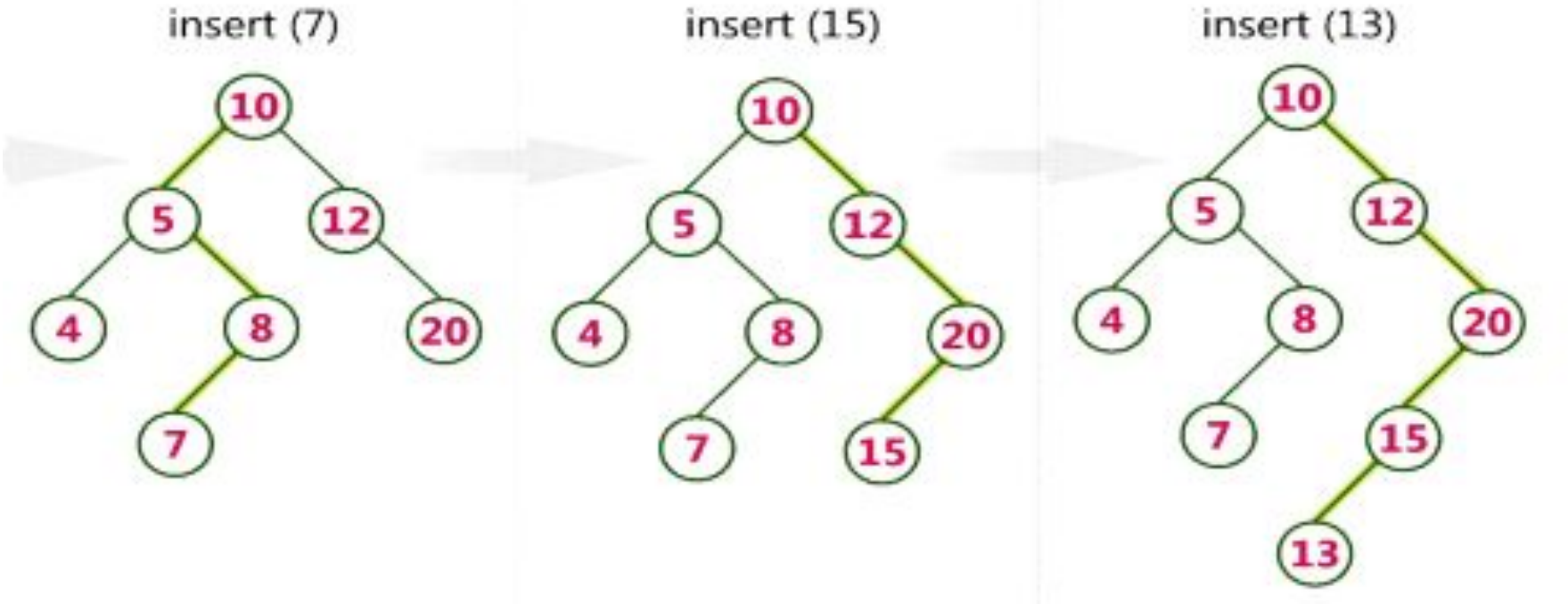
Construct Binary Search Tree from an Array

Construct a Binary Search Tree by inserting the following sequence of numbers: **10,12,5,4,20,8,7,15,13**



Construct Binary Search Tree from an Array

Construct a Binary Search Tree by inserting the following sequence of numbers: **10,12,5,4,20,8,7,15,13**



HEAPS AND HEAPSORT

- In the more commonly-used heap type, there are at most 2 children of a node and it's known as a Binary heap
- A Binary Heap is a Binary Tree with following properties.

1. It’s a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.
2. A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to MinHeap.

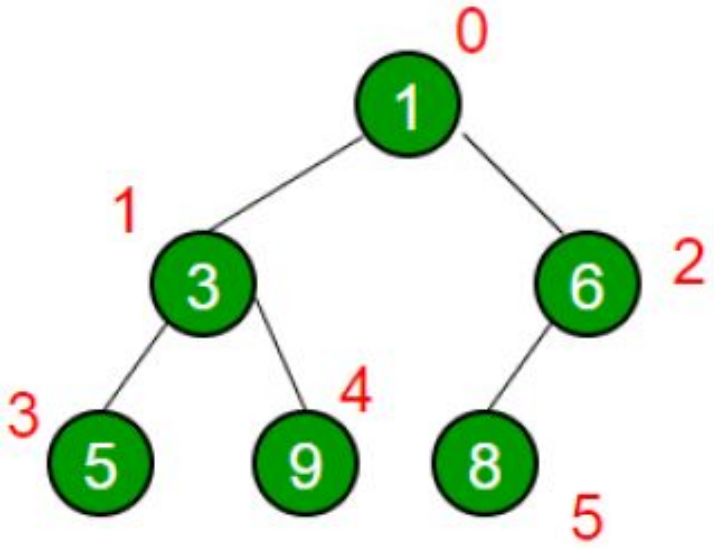
How is Binary Heap represented?

- A binary 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]:

1. Arr[(i-1)/2] : Returns the parent node
2. Arr[(2*i)+1] : Returns the left child node
3. Arr[(2*i)+2] : Returns the right child node

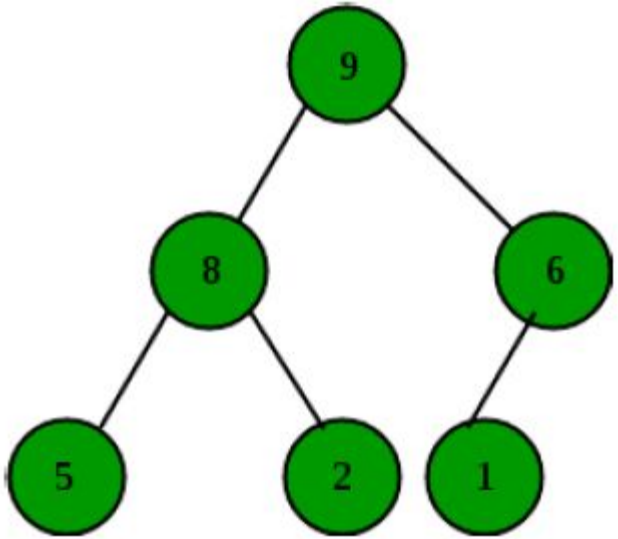
How is Binary Heap represented?

- The traversal method use to achieve Array representation is Level Order



Max Heap

- A max-heap is a complete binary tree in which the value in each internal node is greater than or equal to the values in the children of that node.
- Mapping the elements of a heap into an array is trivial: if a node is stored an index k, then its left child is stored at index 2k+1 and its right child at index 2k+2.

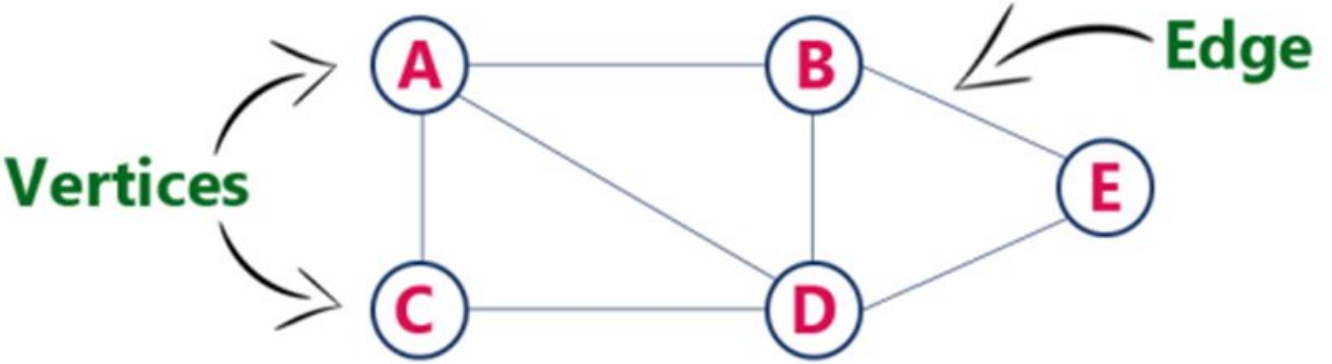


GRAPHS

- A Graph is a non-linear data structure consisting of vertices and edges.
- The graph is denoted by G(E, V).
- Example: graph G can be defined as G = (V , E) Where

  V = {A,B,C,D,E}

  E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}. This is a graph with 5 vertices and 6 edges.



Components of a Graph

1. **Vertex** : An individual data element of a graph is called as Vertex. Vertex is also known as node. In above example graph, A, B, C, D & E are known as vertices.
2. **Edge** : An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (starting Vertex, ending Vertex). In above graph, the link between vertices A and B is represented as (A,B). Edges are three types:

   i.**Undirected Edge** - An undirected edge is a bidirectional edge. If there is an undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).

ii.**Directed Edge** - A directed edge is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).

iii.**Weighted Edge** - A weighted edge is an edge with cost on it.

3.Outgoing Edge :A directed edge is said to be outgoing edge on its origin vertex.

4. Incoming Edge: A directed edge is said to be incoming edge on its destination vertex.

5.Degree: Total number of edges connected to a vertex is said to be degree of that vertex.

6. Indegree:Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

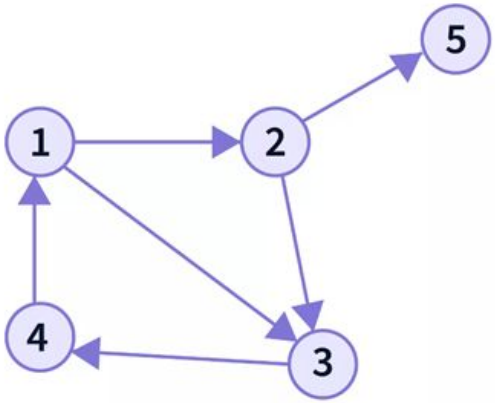
7.**Outdegree**: Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Types of a Graph

Graphs are classified based on the characteristics of their edges. There are two types of graphs:

1. Directed Graphs/ Digraphs

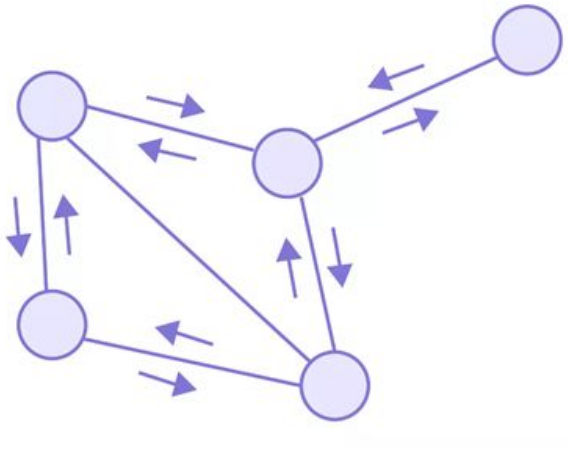
- The edges have directions from one node towards the other node.
- Can only traverse from one node to another if the edge have a direction pointing to that node.
- The maximum number of edges in an directed graph is n(n-1).



Types of a Graph

2. Undirected Graphs/ Digraphs

- Have edges that do not have a direction.
- The graph can be traversed in either direction. So it is bidirectional
- The maximum number of edges in an undirected graph is n(n-1)/2.



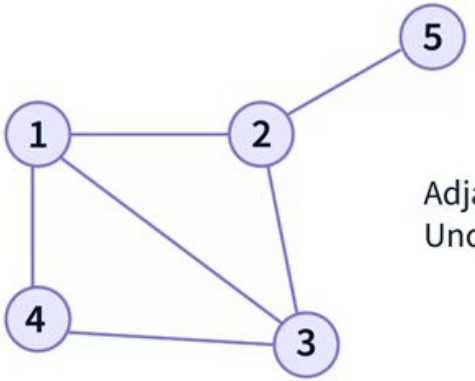
Graph Representation

- In graph data structure, a graph representation is a technique to store graph into the memory of computer. We can represent a graph in many ways.
- The following two are the most commonly used representations of a graph.

  1. Adjacency Matrix
  2. Adjacency List

1. Adjacency Matrix

- An Adjacency Matrix is a 2D array of size V x V where V is the number of nodes in a graph. It is used to represent a "finite graph", with 0's and 1's. Since, it's size is V x V, it is a square matrix.
- The elements of the matrix indicates whether pairs of vertices are adjacent or not in the graph i.e. is there any edge connecting a pair of nodes in the graph.
- Adjacency matrix of an undirected graph is symmetric. Hence, the above graph is symmetric.
- If the graph is weighted, then we usually call the matrix as the cost matrix.



Adjacency Matrix for
Undirected Graph

Nodes →

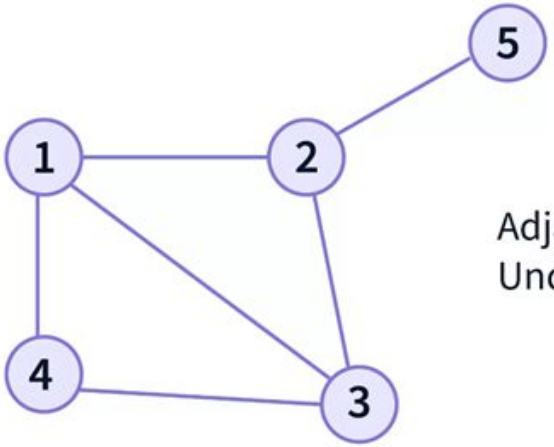
	1	2	3	4	5
1	0	1	1	1	0
2	1	0	1	0	1
3	1	1	0	1	0
4	1	0	1	0	0
5	0	1	0	0	0

Nodes ↑

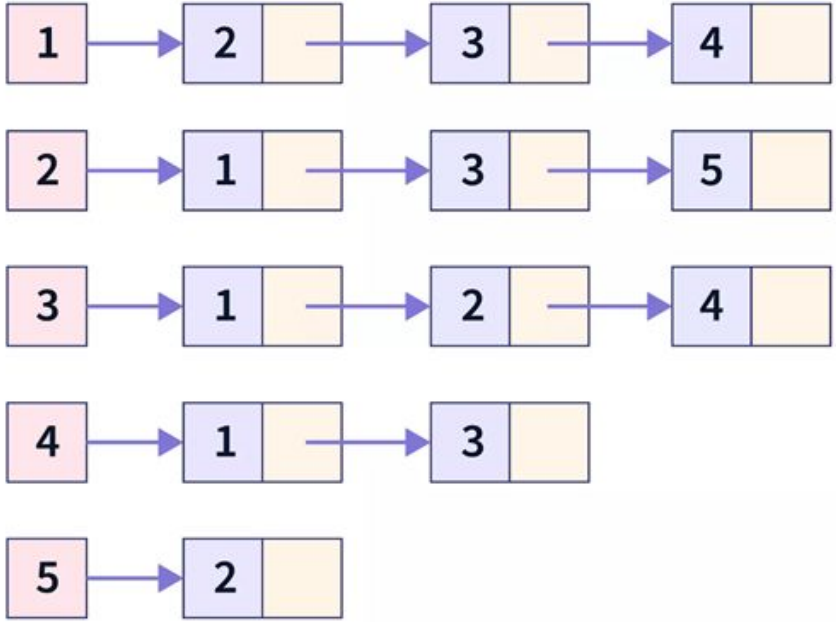
- In the above graph, there is an edge between node 1 & node 2, so in the matrix, we have A[1][2] = 1 and A[2][1] = 1.
- If there is no edge between 2 nodes, then that cell in the matrix will contain '0'. For example, there is no edge from node 1 to node 5, so, in the matrix, A[1][5] = 0 and A[5][1] = 0.

2. Adjacency List

- An adjacency list represents a graph as an array of linked lists.
- The index of the array represents a node.
- Each element in the linked list represents the nodes that are connected to that node by an edge.
- Adjacency List also follows the same rule in case of directed graph, where the nodes will only be linked to the nodes to whom they have a directed edge(or, to the nodes their outgoing edges are pointing to).



Adjacency List for
Undirected Graph



- The graph in our example is undirected and we have represented it using the Adjacency List. Let us look into some important points through this graph:

1. Here, 1, 2, 3, 4, 5 are the nodes(vertices) and each of them forms an array of linked list with all of its adjacent nodes(vertices).

2. In the graph, the node 1 has 3 adjacent nodes namely -- node 2, node 3, node 4. So, in the list, the node is linked with 2, 3 & 4.

3. Node 4 has only 2 adjacent nodes, node 1 & node 3, so it is linked to the nodes 1 and 3 only, in the array of linked list.

4. The last node in the linked list will point to null.

Graph Traversal

1. The process of visiting or updating each vertex in a graph is known as graph traversal.
2. The sequence in which they visit the vertices is used to classify such traversals. Graph traversal is a subset of tree traversal.
3. There are two techniques to implement a graph traversal algorithm:

- Breadth-first search
- Depth-first search

Breadth First Search (BFS)

Breadth First Search (BFS) is a fundamental graph traversal algorithm. It involves visiting all the connected nodes of a graph in a level-by-level manner.

Let's discuss the algorithm for the BFS:

1. **Initialization:** Enqueue the starting node into a queue and mark it as visited.
2. **Exploration:** While the queue is not empty:
    - Dequeue a node from the queue and visit it (e.g., print its value).
    - For each unvisited neighbor of the dequeued node:
        - Enqueue the neighbor into the queue.
        - Mark the neighbor as visited.
3. **Termination:** Repeat step 2 until the queue is empty.

Breadth First Search (BFS)----> Queue data structure

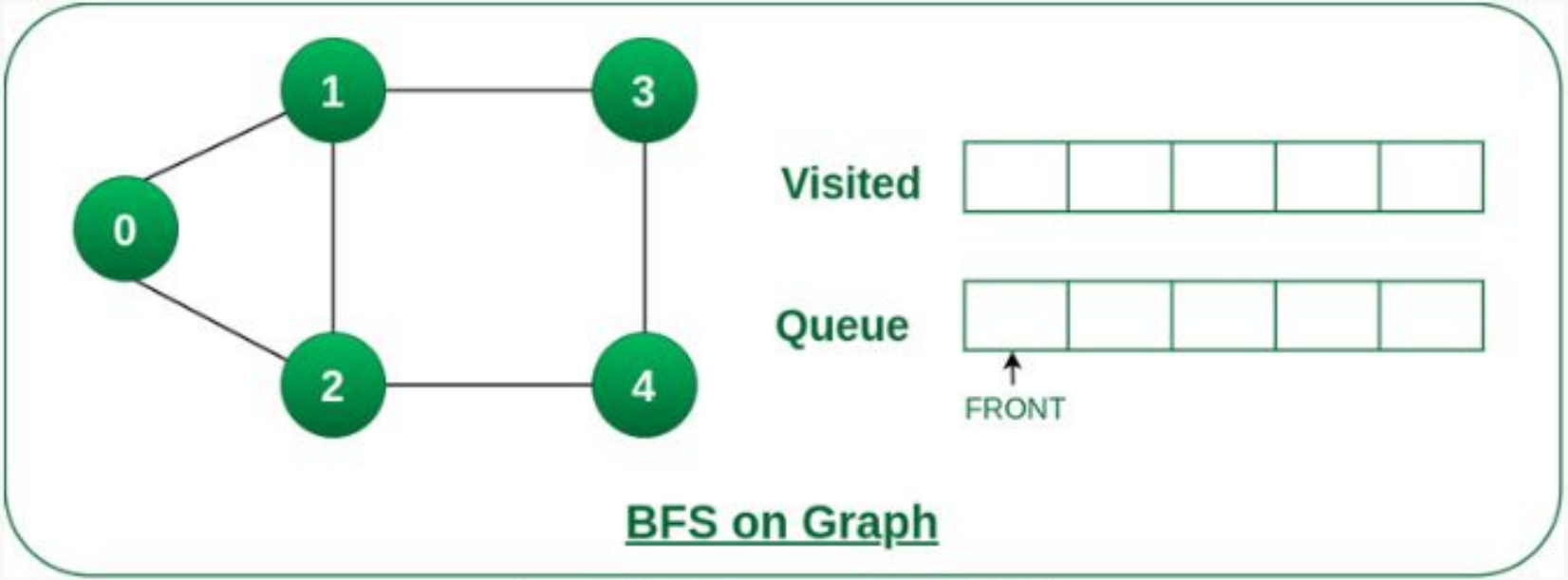
Breadth First Search (BFS) is a fundamental graph traversal algorithm. It involves visiting all the connected nodes of a graph in a level-by-level manner.

Let's discuss the algorithm for the BFS:

1. **Initialization:** Enqueue the starting node into a queue and mark it as visited.
2. **Exploration:** While the queue is not empty:
    - Dequeue a node from the queue and visit it (e.g., print its value).
    - For each unvisited neighbor of the dequeued node:
        - Enqueue the neighbor into the queue.
        - Mark the neighbor as visited.
3. **Termination:** Repeat step 2 until the queue is empty.

How Does the BFS Algorithm Work?

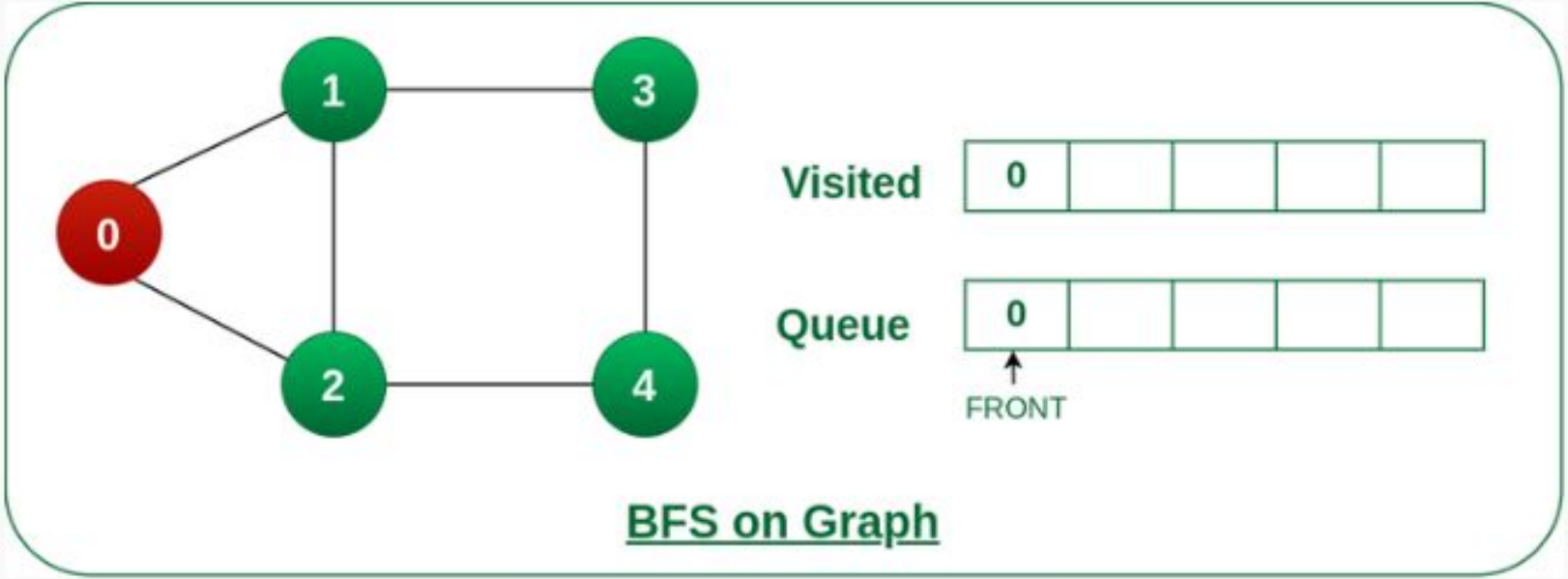
Step1: *Initially queue and visited arrays are empty.*



Queue and visited arrays are empty initially.

How Does the BFS Algorithm Work?

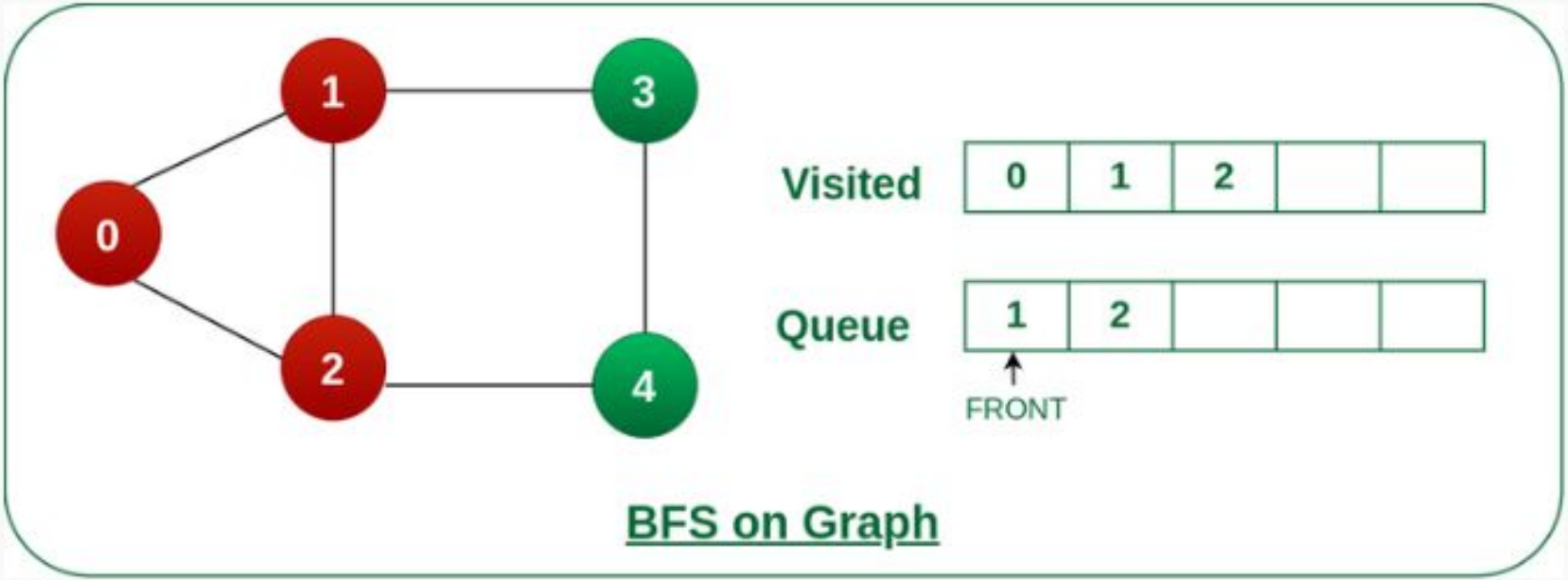
Step2: *Push node 0 into queue and mark it visited.*



Push node 0 into queue and mark it visited.

How Does the BFS Algorithm Work?

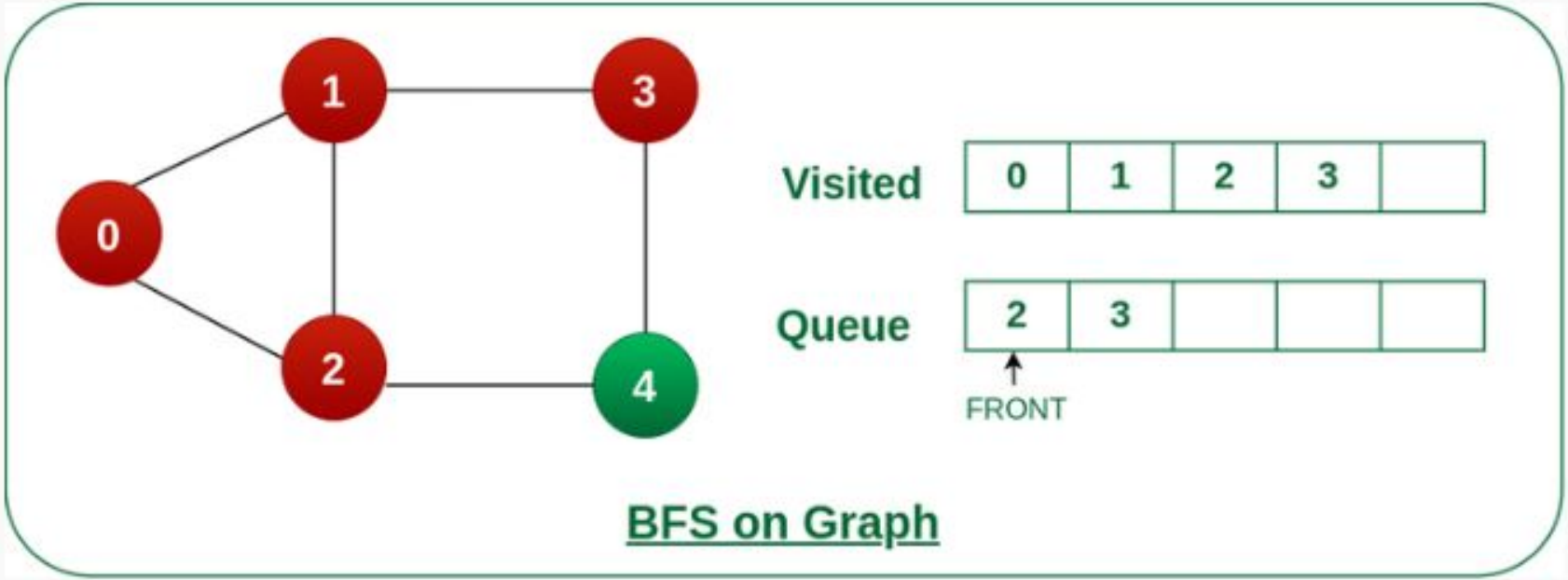
Step 3: *Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.*



Remove node 0 from the front of queue and visited the unvisited neighbours and push into queue.

How Does the BFS Algorithm Work?

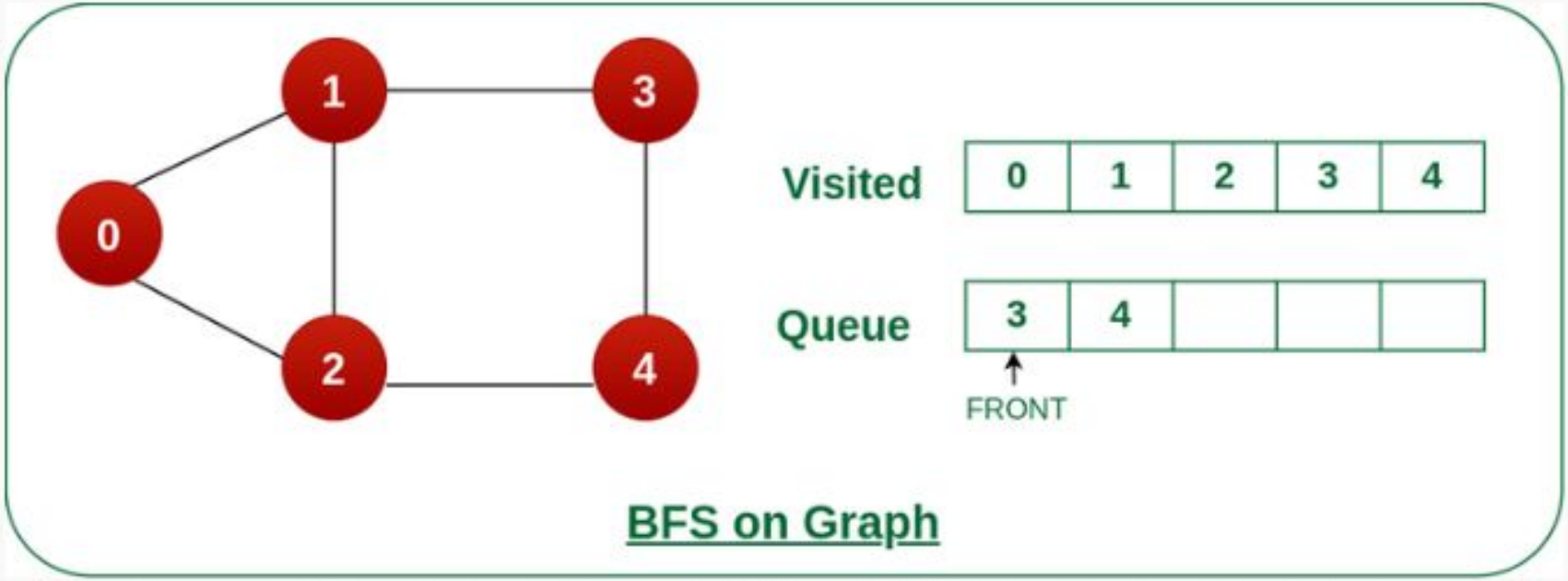
Step 4: *Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.*



Remove node 1 from the front of queue and visited the unvisited neighbours and push

How Does the BFS Algorithm Work?

Step 5: *Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.*

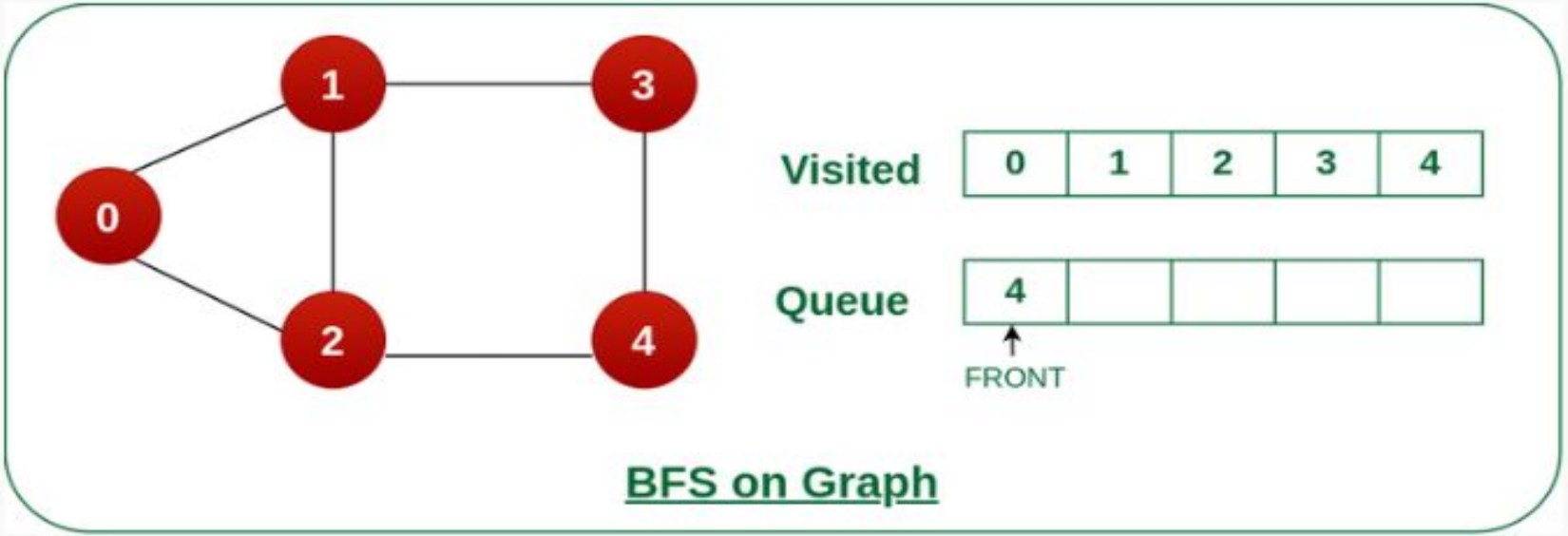


Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.

How Does the BFS Algorithm Work?

Step 6: *Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.*

As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue.

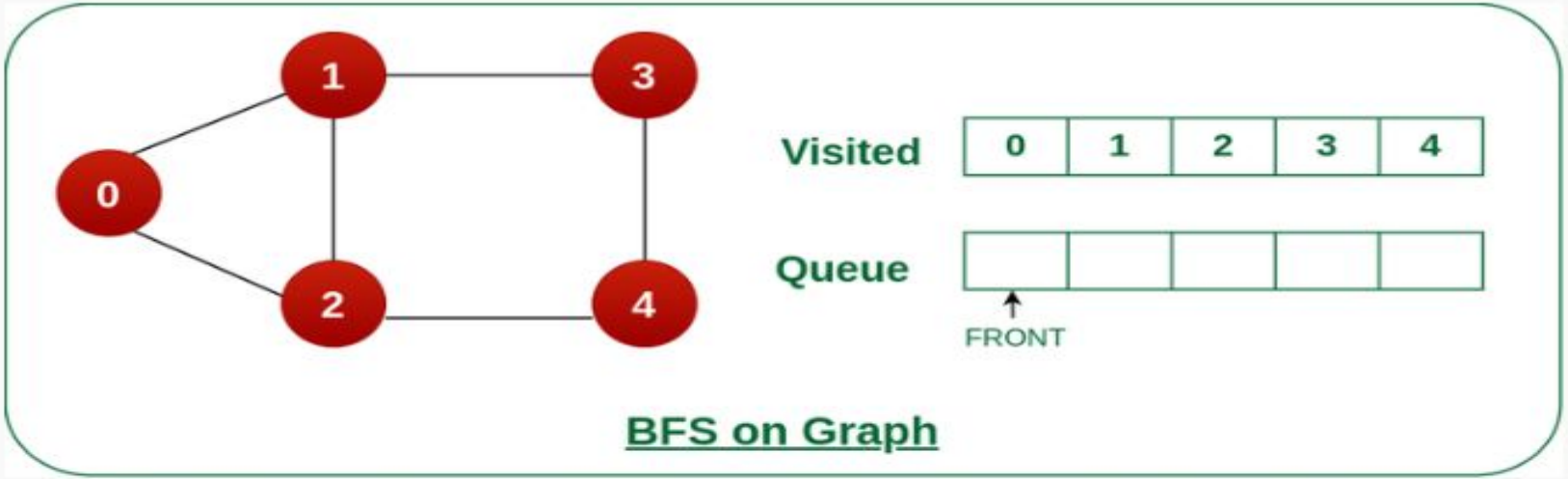


Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.

How Does the BFS Algorithm Work?

Steps 7: *Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.*

As we can see that every neighbours of node 4 are visited, so move to the next node that is in the front of the queue.



Remove node 4 from the front of queue and visit the unvisited neighbours and push them into queue.

Now, Queue becomes empty, So, terminate these process of iteration.

Depth First Search (DFS) ——---> Stack data structure

- Depth-first search is an algorithm for traversing or searching tree or graph data structures.
- The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.
- DFS is implemented using stack data structure

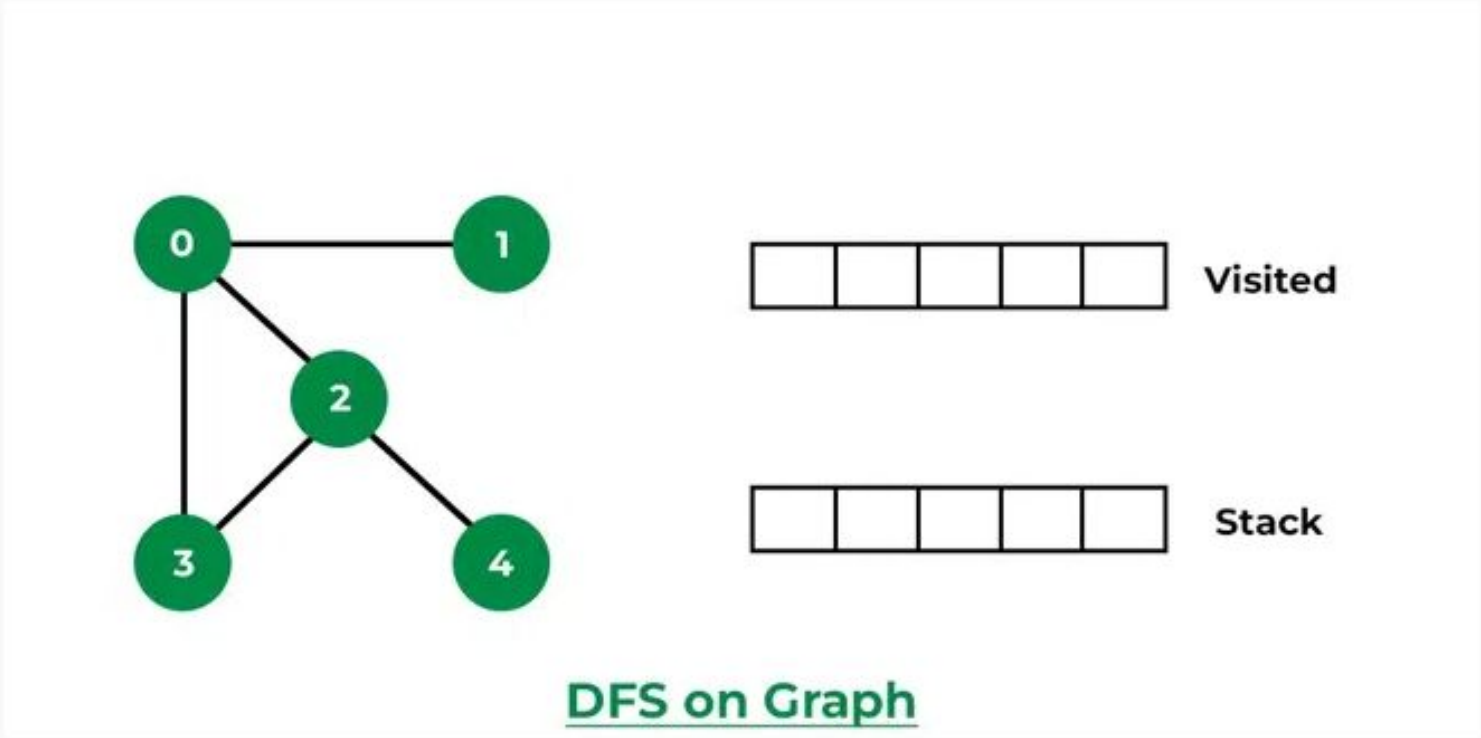
Depth First Search (DFS)

The step by step process to implement the DFS traversal is given as follows -

1. First, create a stack with the total number of vertices in the graph.
2. Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
3. After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
4. Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
5. If no vertex is left, go back and pop a vertex from the stack.
6. Repeat steps 2, 3, and 4 until the stack is empty.

How Does the DFS Algorithm Work?

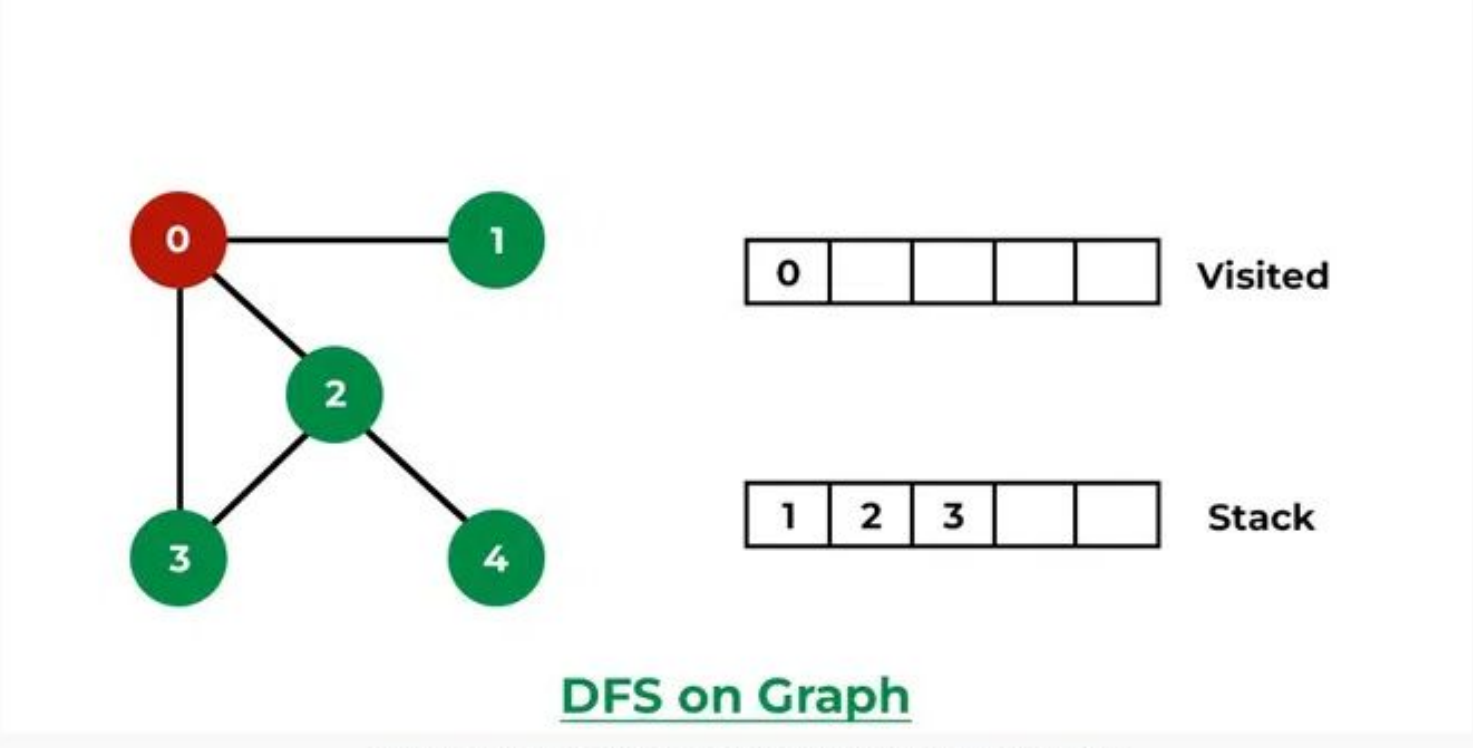
Step1: *Initially stack and visited arrays are empty.*



Stack and visited arrays are empty initially.

How Does the DFS Algorithm Work?

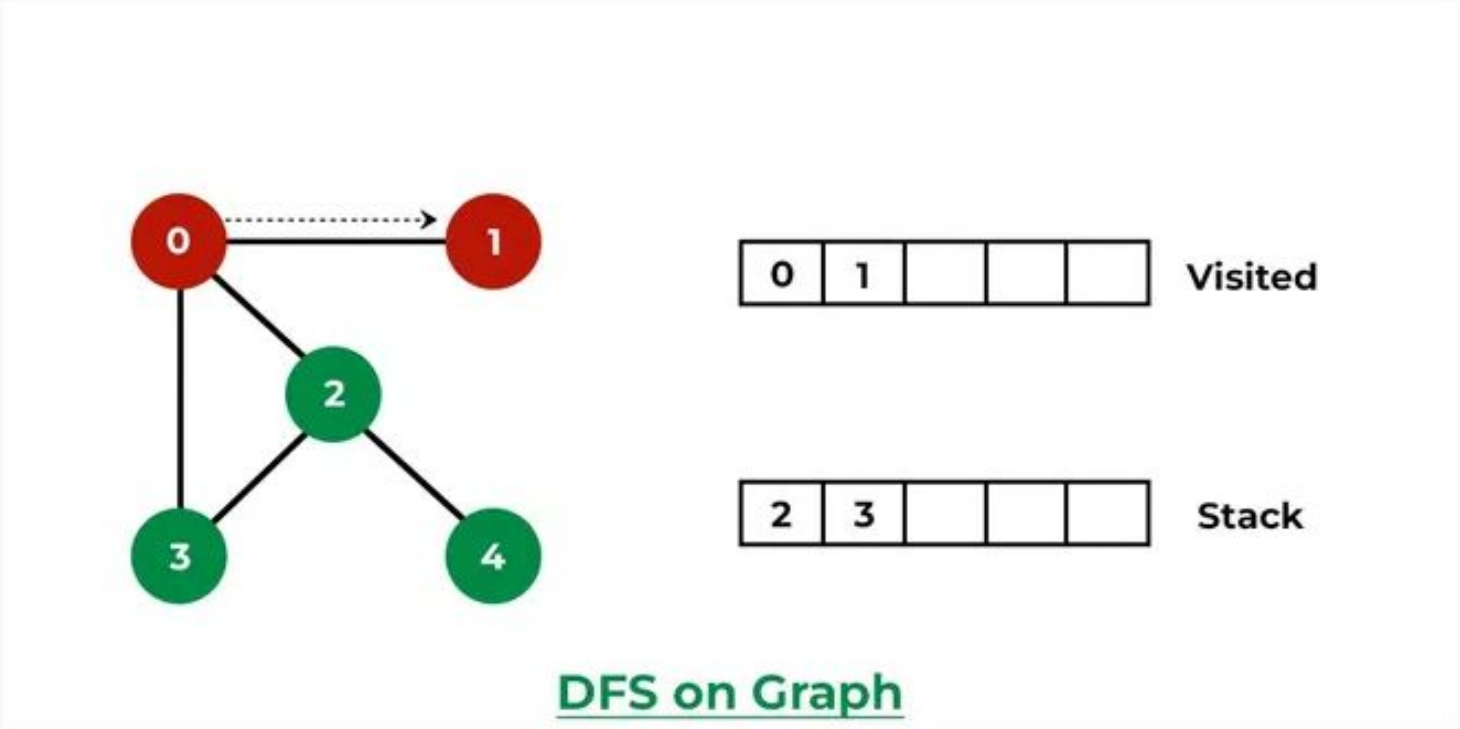
Step 2: *Visit 0 and put its adjacent nodes which are not visited yet into the stack.*



Visit node 0 and put its adjacent nodes (1, 2, 3) into the stack

How Does the DFS Algorithm Work?

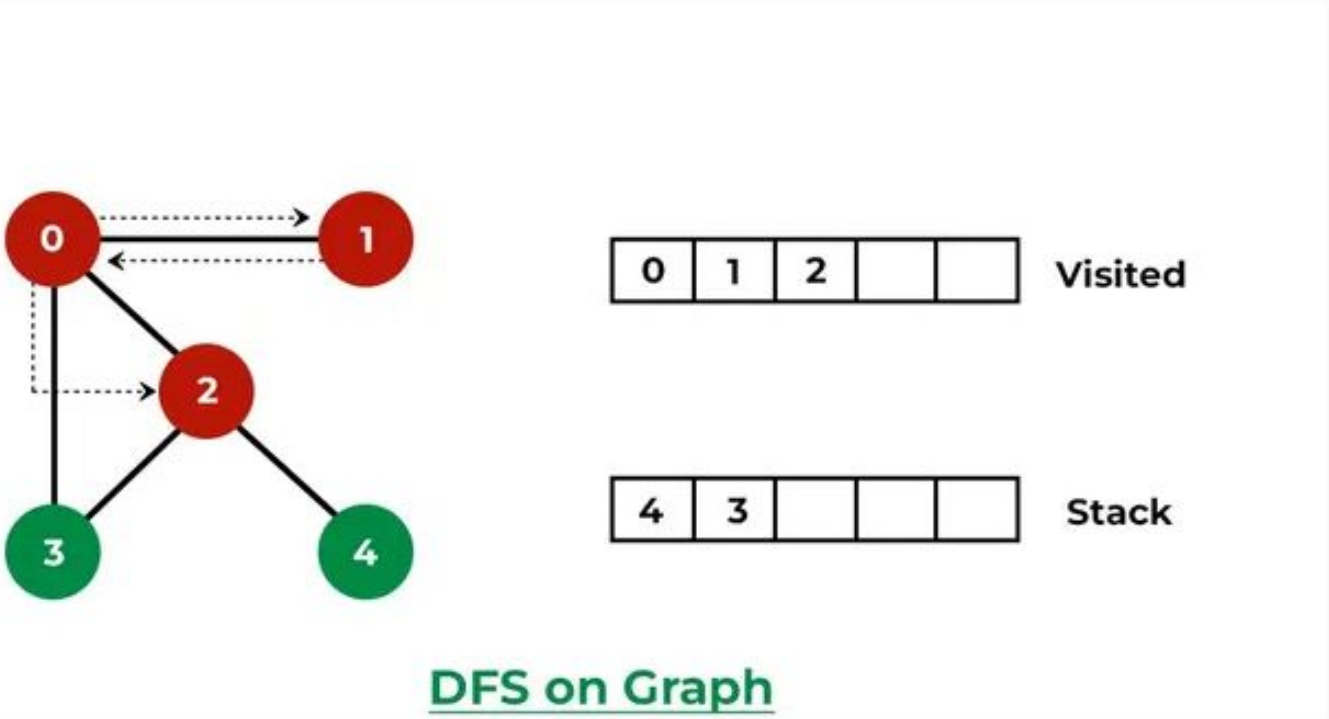
Step 3: *Now, Node 1 at the top of the stack, so visit node 1 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.*



Visit node 1

How Does the DFS Algorithm Work?

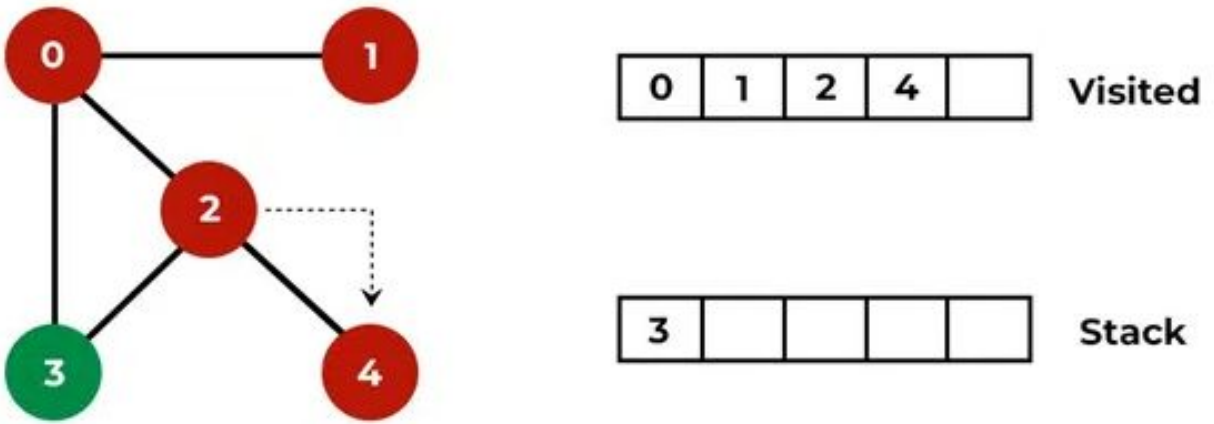
Step 4: *Now, Node 2 at the top of the stack, so visit node 2 and pop it from the stack and put all of its adjacent nodes which are not visited (i.e, 3, 4) in the stack.*



Visit node 2 and put its unvisited adjacent nodes (3, 4) into the stack

How Does the DFS Algorithm Work?

Step 5: *Now, Node 4 at the top of the stack, so visit node 4 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.*

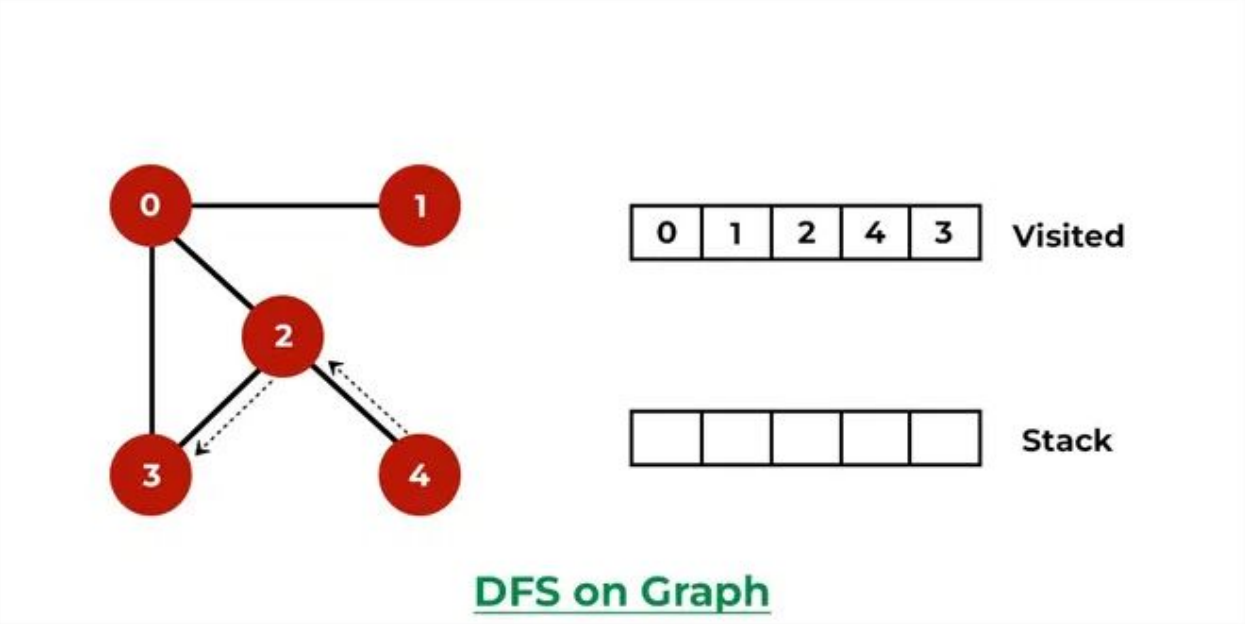


DFS on Graph

Visit node 4

How Does the DFS Algorithm Work?

Step 6: *Now, Node 3 at the top of the stack, so visit node 3 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.*



Visit node 3

Now, Stack becomes empty, which means we have visited all the nodes and our DFS traversal ends.

Time & Space Complexities

Algorithm	**Time Complexity**	**Space complexity**
Breadth First Search	O(V + E)	O(V)
Depth First Search	O(V + E)	O(V)

Applications of Graphs

1. GPS systems and Google Maps use graphs to find the shortest path from one destination to another.
2. The Google Search algorithm uses graphs to determine the relevance of search results.
3. World Wide Web is the biggest graph. All the links and hyperlinks are the nodes and their interconnection is the edges. This is why we can open one webpage from the other.
4. Social Networks like facebook, twitter, etc. use graphs to represent connections between users.
5. The nodes we represent in our graphs can be considered as the buildings, people, group, landmarks or anything in general , whereas the edges are the paths connecting them.

Hashing

Hash Table

1. The Hash table data structure stores elements in key-value pairs where

- Key- unique integer that is used for indexing the values
- Value - data that are associated with keys.

2. Access of data becomes very fast if we know the index of the desired data.
3. Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data.

Hashing (Hash Function)

1. In a hash table, a new index is processed using the keys. And, the element corresponding to that key is stored in the index. This process is called hashing.
2. The idea of hashing is to distribute entries (key/value pairs) uniformly across an array. By using that key you can access the element in O(1) time.
3. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.

Hashing (Hash Function)

- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)

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

Collision Resolution Techniques

- The hash function is used to find the index of the array.
- The hash value is used to create an index for the key in the hash table.
- The hash function may return the same hash value for two or more keys.
- When two or more keys have the same hash value, a collision happens. To handle this collision, we use collision resolution techniques.
- There are two types of collision resolution techniques.

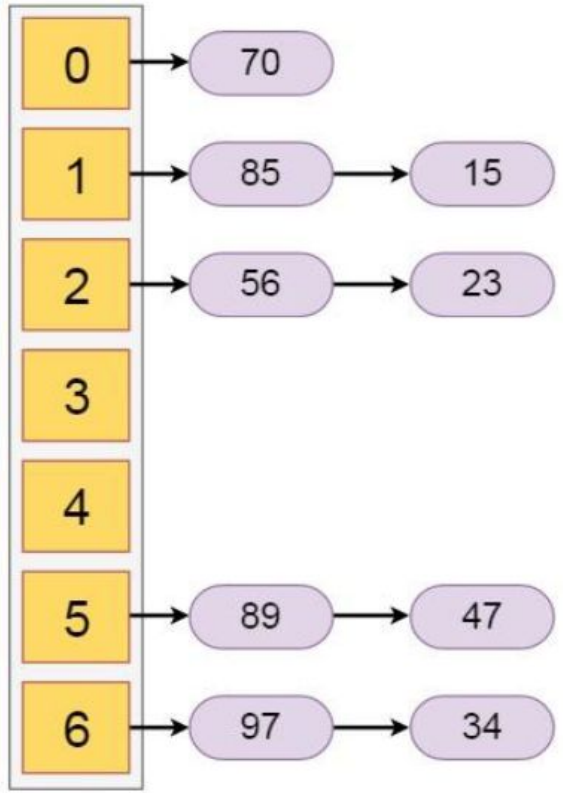
1. Separate chaining (open hashing)
2. Open addressing (closed hashing)

Collision Resolution Techniques

1. **Separate chaining (open hashing)**
   - This method involves making a linked list out of the slot where the collision happened, then adding the new key to the list.
   - Separate chaining is the term used to describe how this connected list of slots resembles a chain.
   - It is more frequently utilized when we are unsure of the number of keys to add or remove.
   - Time complexity
1. Its worst-case complexity for searching is o(n).
2. Its worst-case complexity for deletion is o(n).

Example: If we have some elements like {15, 47, 23, 34, 85, 97, 65, 89, 70}. And our hash function is h(x) = x mod 7.

x	h(x) = x mod 7
15	1
47	5
23	2
34	6
85	1
97	6
56	2
89	5
70	0



Chaining

Collision Resolution Techniques

2.Open addressing (closed hashing)

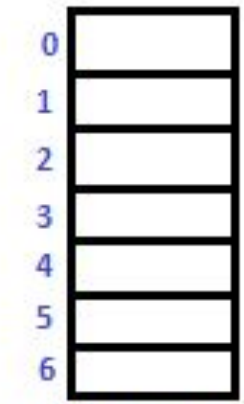
- No key is kept anywhere else besides the hash table.
- As a result, the hash table’s size is never equal to or less than the number of keys. Additionally known as closed hashing.
- The following techniques are used in open addressing:
- Linear probing
- Quadratic probing
- Double hashing

Linear Probing:

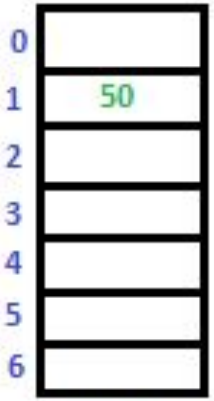
- In linear probing, the hash table is searched sequentially that starts from the original location of the hash.
- If in case the location that we get is already occupied, then we check for the next location.
- Let hash(x) be the slot index computed using a hash function and S be the table size.

1. If slot hash(x) % S is full, then we try (hash(x) + 1) % S
2. If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S
3. If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S

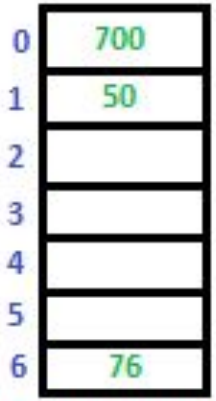
If collision occurs then the next free location in linear probing is calculated as (u+i) where u is the location(index) where collision occurred and i runs from 0 to (size of hashtable -1)



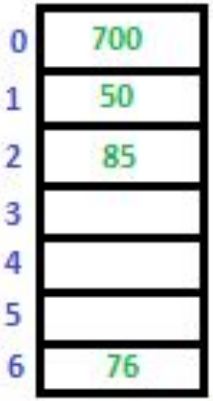
Initial Empty Table



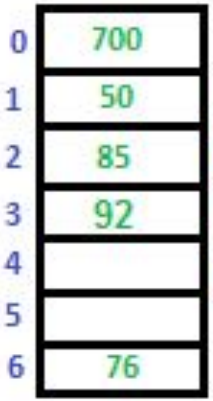
Insert 50



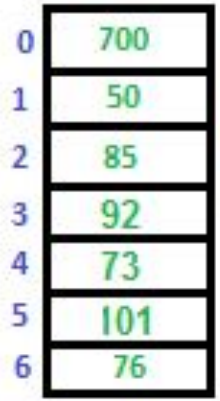
Insert 700 and 76



Insert 85: Collision Occurs, insert 85 at next free slot.



Insert 92, collision occurs as 50 is there at index 1. Insert at next free slot



Insert 73 and 101

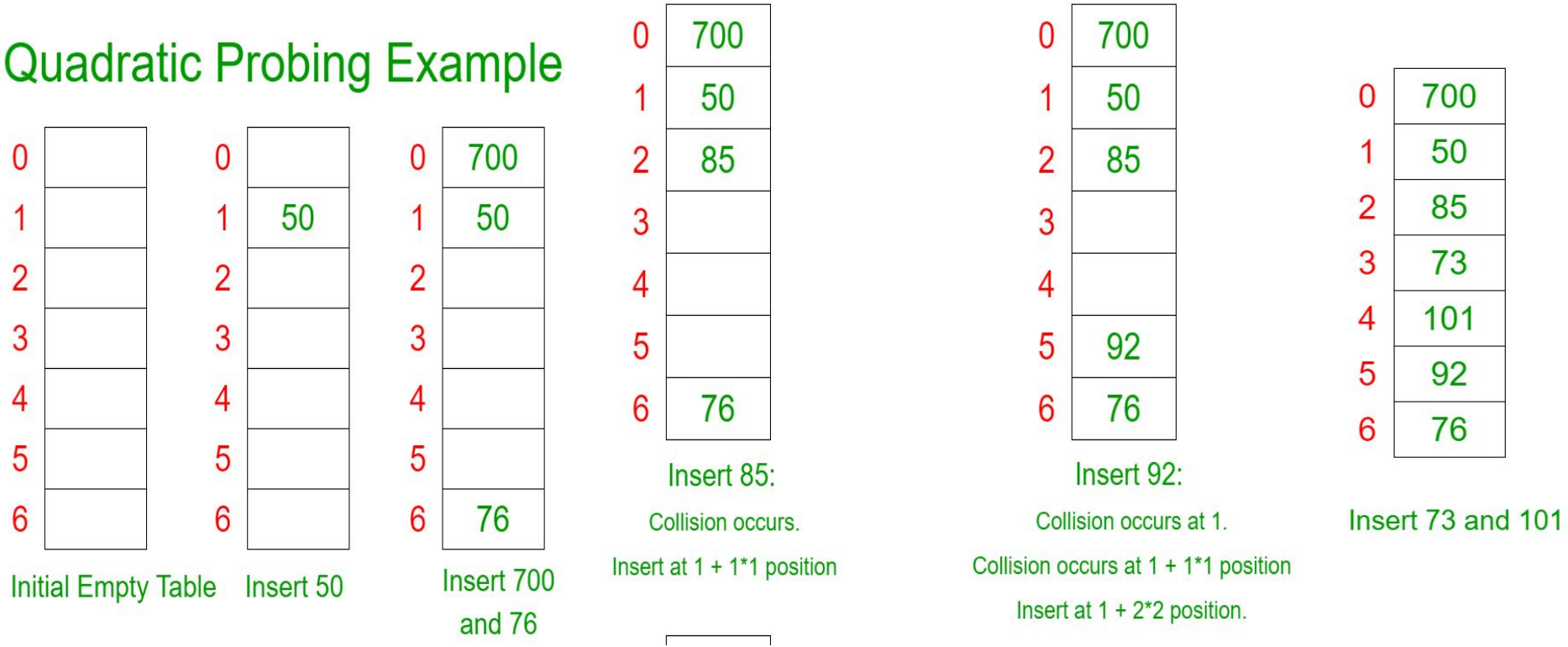
Quadratic Probing:

1. Quadratic probing is an open-addressing scheme where we look for the free location using $(u + i^2)$ where u is the location(index) where collision occurred and i runs from 0 to (size of hashtable -1) if the given hash value x collides in the hash table.

- If the slot hash(x) % S is full, then we try (hash(x) + 1*1) % S.
- If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S.
- If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S.
- This process is repeated for all the values of i until an empty slot is found.

Quadratic Probing:

2. For example: Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101



Double Hashing/Rehashing:

1. In double hashing, we make use of two hash functions. The first hash function is h1(k), this function takes in our key and gives out a location on the hash-table. If the new location is empty, we can easily place our key in there without ever using the secondary hash function.
2. In case of a collision, we need to use secondary hash-function h2(k) in combination with the first hash-function h1(k) to find a new location on the hash-table.
3. The combined hash-function used is of the form newloc=u+v*i Where u= h1(k) hash value, v=h2(k) hash value and i=0 to (size of hash table -1)
4. The second hash function is used only when collision occurs

Algorithm Analysis

Unit 3

What is Algorithm?

- A finite set of instruction that specifies a sequence of operation is to be carried out in order to solve a specific problem or class of problems is called an Algorithm.
- In computer programming terms, an algorithm is a set of well-defined instructions to solve a particular problem. It takes a set of input and produces a desired output. For example,

  An algorithm to add two numbers:

  1. Take two number inputs
  2. Add numbers using the + operator
  3. Display the result

Qualities of Good Algorithms

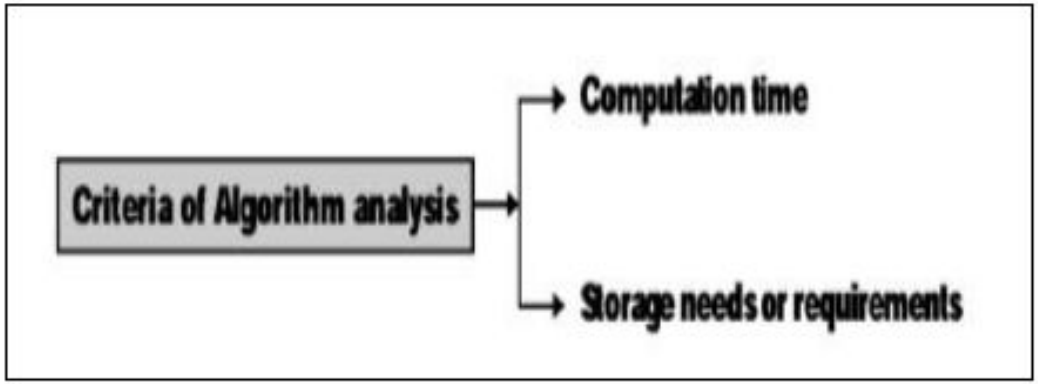
- Input and output should be defined precisely.
- Each step in the algorithm should be clear and unambiguous.
- Algorithms should be most effective among many different ways to solve a problem.
- An algorithm shouldn't include computer code. Instead, the algorithm should be written in such a way that it can be used in different programming languages.

- Reasons for analyzing algorithms:

1. To predict the resources that the algorithm require
   - Computational Time(CPU consumption).
   - Memory Space(RAM consumption).
   - Communication bandwidth consumption.
2. To predict the running time of an algorithm
   - Total number of primitive operations executed.

Algorithm Analysis:

Analysis of algorithms is the determination of the amount of time and space resources required to execute it.



The criteria of analysis of an algorithm are collectively called as its complexity. The computation time is called as 'Time Complexity' and the storage needs or requirements are called as 'Space Complexity'.

Time Complexity:

- The total amount of computer time needed to complete the execution of an algorithm (indirectly program) is called as time complexity.
- The total time required for the algorithm to complete its execution includes compilation time and run time. The compilation time of an algorithm is constant that depends on the compiler. So, the time complexity of an algorithm is calculated on the basis of the run time.
- For the sake of simplicity unit of run time (i.e. 1) is considered for all the executable statements of the algorithm. So, the unit time considered for the calculation of time complexity includes arithmetic operation, assignment operation, condition operation, etc. Following list shows the example of operations and units of run time.

Operation	Unit of Run Time	Remark
Num = 10	1	1 for assignment
Sum = Sum + 10	2	1 for addition and 1 for assignment
SI = P*T*R/100	4	2 for multiplication, 1 for division and 1 for assignment
Num != 0	1	1 for condition
Return sum	1	1 for return statement

- The number of operations involved in each of the statements is calculated. If more operations are there then they are added. The frequency of the statement is calculated. Then these two components are multiplied to get the total time of the statements.
- Like this, the total time for each statement of the algorithm is calculated. The grand total of these total times of every statement is the time complexity of the algorithm. All these items can be tabulated to find the time complexity for every algorithm.

Example : Find the time complexity of the following algorithm :

```
Sum(Arr, N)
/* Arr is an array of size N */
{
S = 0
I = 1
while (I<=N)
{
S = S + Arr[I]
I = I + 1
}
return S
}
```

Cost: No. of operations involved in each statements Return and print statements have cost 1.

Frequency: Total time a statement is executing

Statement	cost	Frequency	Total Time	Remarks
Sum(Arr, N)	0	0	0	Heading of algorithm, not an executable statement.
{	0	0	0	Block begins, not an executable statement.
S = 0	1	1	1	1 unit for assignment statement.
I = 1	1	1	1	1 unit for assignment statement.
while (I<=N)	1	N+1	N+1	Condition, i<=n is tested for N+ 1 time. When N is positive & the testing begins for I = 1. The condition is true when I< = N. when I is equal to N+ 1 the condition is false.
S = S + Arr[I]	2	N	2N	Addition & assignment. It is in the loop and executed for N times.
I = I + 1	2	N	2N	Addition & assignment. It is in the loop and executed for N times.
return S	1	1	1	Return statement. Executed once because it is outside the loop.
}	0	0	0	Sum of Total Time of each statement.
Grand Total Time			**5N+4**	

- So, the time complexity of the above algorithm is 5N+4.
- Here the exact time is not fixed and it changes based on the value of 'n'. If we increase the 'n' value then time required also increases linearly.
- If the amount of time required by an algorithm is increased with the input value then that time complexity is said to be "Linear Time Complexity".
- The Linear time complexity can also be written as O(n) -> order of n.
- It means that the running time increases with the increase in input value.
  5N+4 -> O(n)

Example 2: Find the time complexity of below algorithm.

```
int sum(int a,int b)
{
   int c=0
   c=a+b
   return c
}
```

Statement	cost	Frequency	Total Time
int c=0	1	1	1
c=a+b	2	1	2
return c	1	1	1
Grand Total Time			4 Unit

- So, the complexity of above algorithm is 4 unit, here the exact time is fixed i.e. 4 and it will not change if any changes made in input value a and b
- such time complexity is also called as "constant time complexity" and it is represented by O(1) ->Order of 1.

Space Complexity:

The amount of memory used by a program to execute it is represented by its space complexity.

Calculating the Space Complexity:

For calculating the space complexity, we need to know the value of memory used by different type of datatype variables, which generally varies for different operating systems, but the method for calculating the space complexity remains the same.

Example:

```
{
   int z = a + b + c;
   return (z);
}
```

In the above expression, variables a, b, c and z are all integer types, hence they will take up 4 bytes each, so total memory requirement will be 4 (4) = 16 bytes , This space requirement is fixed for the above example, hence it is called **Constant Space Complexity**.

Asymptotic Notation:

Asymptotic Notations are the expressions that are used to represent the complexity of an algorithm.

There are three types of analysis that we perform on a particular algorithm.

1. **Best Case**: In which we analyze the performance of an algorithm for the input, for which the algorithm takes less time or space.
2. **Worst Case**: In which we analyses the performance of an algorithm for the input, for which the algorithm takes long time or space.
3. **Average Case**: In which we analyze the performance of an algorithm for the input, for which the algorithm takes time or space that lies between best and worst case.

Types of Asymptotic Notation:

1. Big-O Notation (O) – Big O notation specifically describes worst case scenario.
2. Omega Notation (Ω) – Omega (Ω) notation specifically describes best case scenario.
3. Theta Notation (θ) – This notation represents the average complexity of an algorithm.

1. Big-O Notation (O):

Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity. That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity. Big - Oh Notation can be defined as follows...

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) <= C g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as O(g(n)).

Example:

Time complexity of an algorithm is 4n+4, let's say it is represented by a function f(n)=4n+4. It can be represented as O(n).

F(n)=4n+4

g(n)=n (always consider highest order of n of f(n) as g(n))

Now we have prove that,

f(n) <= c.g(n) for all n>=1 and c>0

It means we have to find value for c and n. we can suppose value of n as any positive constant greater than 0. And the value of constant c is assumed as coefficient of n plus 1 (only for big omega) in this case coefficient of n is 4 so value of c is 5.

Assume c=5,n=1: f(n)=4n+4=8 and c.g(n)= 5*1=5

8 <=5 False

For n=2: f(n)=4n+4=12 and c.g(n)=5*2=10

10<=12 False

For n=3: f(n)=4n+4=17 and c*g(n)=5*3=15

17<=15 False

For n=4: f(n)=4n+4=20 and c*g(n)=5*4=20

20<=20 True

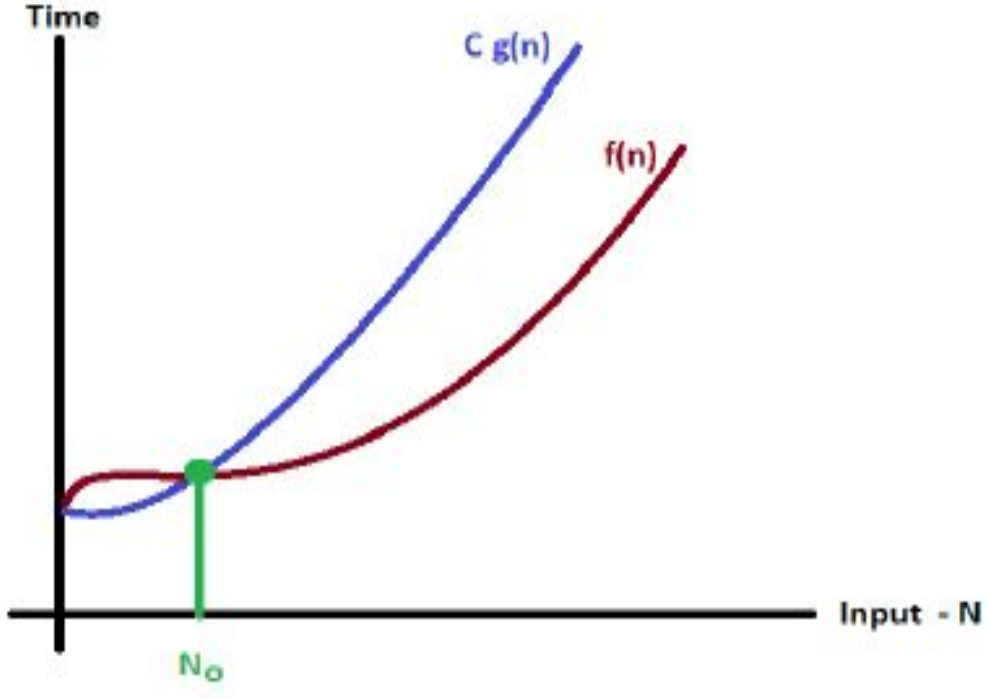
Above condition is proved hence we can say that, f(n)<=c*g(n)

i.e. 4n+4 =O(g(n))

4n+4= O(n) ...(value of g(n) is n)

F(n)=O(n) for all c=5 and n>=4

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C.g(n) is greater than f(n) which indicates the algorithm's upper bound.

2. Big Omega Notation (Ω):

Big - Omega notation is used to define the lower bound of an algorithm in terms of Time Complexity. That means Big-Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big-Omega notation describes the best case of an algorithm time complexity.

Big - Omega Notation can be defined as follows...

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) >= C g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as Ω(g(n)).

Example:

Time complexity of an algorithm is $4n^2+4n+2$, let's say it is represented by a function $f(n)= 4n^2+4n+2$. It can be represented as $\Omega(n^2)$.

Proof:

$F(n)= 4n^2+4n+2$

$g(n)=n^2$ (always consider highest order of n of f(n) as g(n))

Now we have prove that,

$f(n) >= c.g(n)$ for all $n>=1$ and $c>0$

It means we have to find value for c and n. we can suppose value of n as any positive constant greater than 0. And the value of constant c is assumed as coefficient of n (only for big omega) in this case coefficient of n is 4 so value of c is 4.

Assume $c=4, n=1$: $f(n)= 4n^2+4n+2=10$ and $c*g(n)=4*1=4$

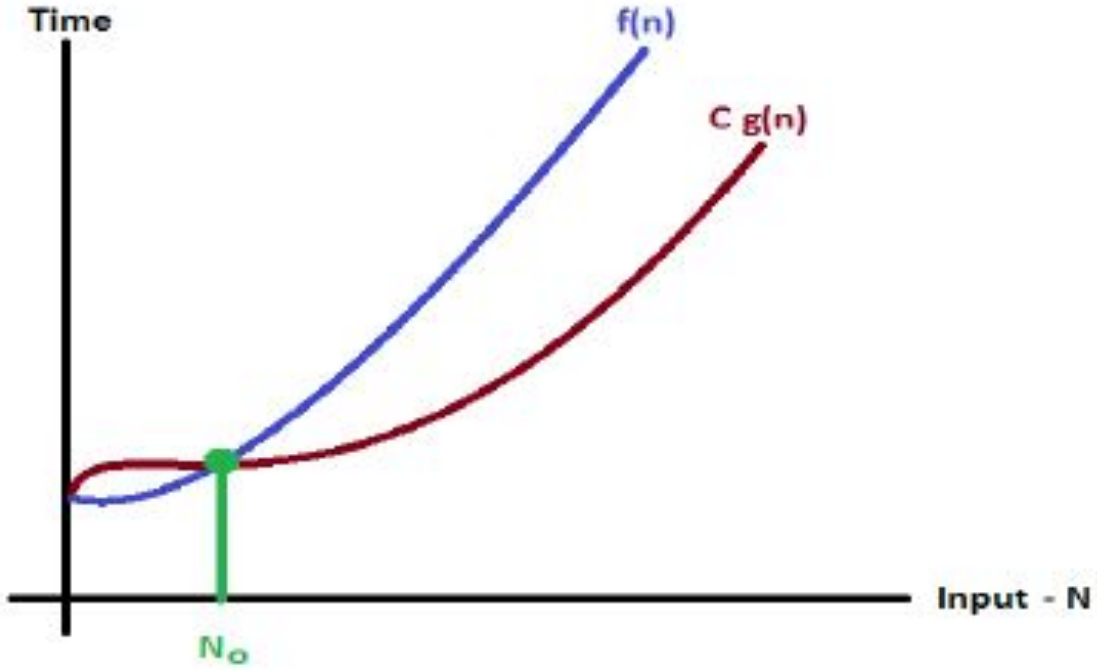
$10 >=4$ True

Above condition is proved hence we can say that, $f(n)>=c.g(n)$

i.e. $4n^2+4n+2= \Omega(g(n))$.

$4n^2+4n+2= \Omega(n^2)$....(value of g(n) is n^2)
$F(n)= \Omega(n^2)$ for all $c=4$ and $n>=1$

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C g(n) is less than f(n) which indicates the algorithm's lower bound.

4. Big - Theta notation:

It is used to define the average bound of an algorithm in terms of Time Complexity.

That means Big - Theta notation always indicates the average time required by an algorithm for all input values. That means Big - Theta notation describes the average case of an algorithm time complexity.

Big - Theta Notation can be defined as follows...

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If $C_1\ g(n) <= f(n) <= C_2\ g(n)$ for all $n >= n_0$, $C_1 > 0$, $C_2 > 0$ and $n_0 >= 1$. Then we can represent f(n) as $\Theta(g(n))$.

Example:

Time complexity of an algorithm is $4n^2+4n+2$, let's say it is represented by a function $f(n)= 4n^2+4n+2$. It can be represented as $\Theta(n^2)$.

Proof:

$F(n)= 4n^2+4n+2$

$g(n)=n^2$ (always consider highest order of n of f(n) as g(n))

Now we have prove that,

$$C1*g(n) <= f(n) <= c2*g(n) \text{ for all } n>=1 \text{ and } c>0$$

It means we have to find value for c1,c2 and n. we can suppose value of n as any positive constant greater than 0. And the value of constant c1(lower bound constant) is assumed as coefficient of n, in this case coefficient of n is 4 so value of c1 is 4 and the value of constant c2(upper bound constant) is assumed as coefficient of n plus 1 in this case coefficient of n is 4 so value of c2 is 5.

Assume c1=4,c2=5,n=1:

$c1.g(n)=4, f(n)= 4n^2+4n+2=10, c2.g(n)=5$

$4<=10<=5$ False

For n=2:

c1.g(n)=16,f(n)= $4n^2+4n+2=26$, c2.g(n)=20

16<=26<=20 False

For n=3:

c1.g(n)=36,f(n)= $4n^2+4n+2=50$, c2.g(n)=45

36<=50<=45 False

For n=4:

c1.g(n)=64,f(n)= $4n^2+4n+2=82$, c2.g(n)=80

64<=82<=80 False

For n=5:

c1.g(n)=100,f(n)= $4n^2+4n+2=122$, c2.g(n)=125

100<=122<=125 True

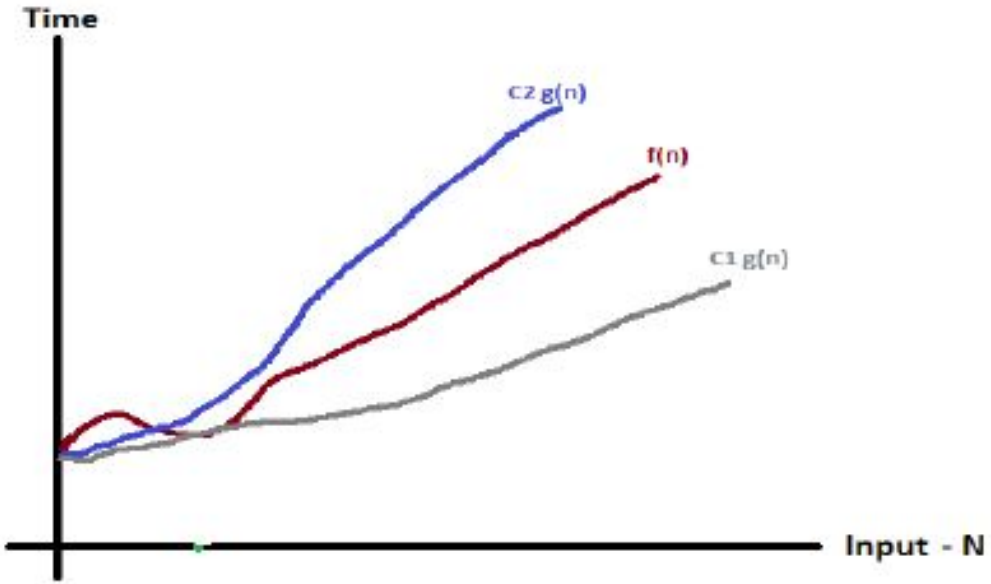
Above condition is proved hence we can say that, C1.g(n) <= f(n) <=c2.g(n)

i.e. $4n^2+4n+2= \Theta(g(n))$.

$4n^2+4n+2= \Theta(n^2)$....(value of g(n) is n^2)

$F(n)= \Theta(n^2)$ for all c1=4 ,c2=5 and n>=5

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C_1 g(n) is less than f(n) and C_2 g(n) is greater than f(n) which indicates the algorithm's average bound.

Selecting constant values:

Eg 1: $f(n)=4n+2$

Lower Bound	Upper Bound	Tight Bound
C=4 (first term)	C=5 (first term+1)	4 (first term)

- Value of C in Big Oh(upper Bound) is 5
- Value of C in Big Omega (Lower Bound) is 4
- In big theta 2 constants are used c1(lower bound) and c2(upper bound),so c1=4 and c2=5

Eg 2: $f(n)=5n^2+4n+2$

Lower Bound	Upper Bound	Tight Bound
C=5 (first term coefficient)	C=6 (first term coefficient +1)	5 (first term)

- Value of C in Big Oh(upper Bound) is 6
- Value of C in Big Omega (Lower Bound) is 5
- In big theta 2 constants are used c1(lower bound) and c2(upper bound),so c1=5 and c2=6

another example:

```
def method1():
    l = []
    for n in xrange(10000):
        l = l + [n]
```

- In the above code, 4*n bytes of space is required for the array L[] elements.
- 4 bytes for n

Hence the total memory requirement will be (4n + 4), which is increasing linearly with the increase in the input value n, hence it is called as Linear Space Complexity.

Similarly, we can have quadratic and other complex space complexity as well, as the complexity of an algorithm increases.
But we should always focus on writing algorithm code in such a way that we keep the space complexity **minimum**.

MASTER THEOREM FOR DIVIDE & CONQUER

Refer problems done in class

- The Master Theorem is a tool used to solve recurrence relations that arise in the analysis of divide-and-conquer algorithms.
- Master Theorem is used to determine running time of algorithms (divide and conquer algorithms) in terms of asymptotic notations.
- The Master Theorem provides a systematic way of solving recurrence relations of the form:

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n),$$

Here n/b is the size of the sub problem, a is the number of subproblems, f(n)the time to create the subproblems and combine their results in the above procedure.

- The Master Theorem provides a systematic way of solving recurrence relations of the form:

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n),$$

Where a ≥ 1, b > 1 and f(n)must be always positive

CASE 1: If $f(n) < n^{\log_b a}$ **, then** $T(n) = O(n^{\log_b a})$

CASE 2: If $f(n) = n^{\log_b a}$ **, then** $T(n) = O(n^{\log_b a} \cdot \log n)$

CASE 3: If $f(n) > n^{\log_b a}$ **, then** $T(n) = O(f(n))$

MASTER THEOREM FOR SUBTRACT & CONQUER

Refer problems done in class

- Master theorem is used to determine the Big – O upper bound on functions which possess recurrence, i.e which can be broken into sub problems.
- Used to directly calculate the time complexity function of 'decreasing' recurrence relations of the form

$$T(n) = aT(n-b) + n^k$$

Here n-b is the size of the sub problem, a is the number of subproblems, f(n)the time to create the subproblems and combine their results in the above procedure.

- The Master Theorem provides a systematic way of solving recurrence relations of the form:

$$T(n) = aT(n-b) + n^k$$

Where a ≥ 1, b > 1 and f(n)is the time to create the subproblems and combine their results in the above procedure.

CASE 1: If a<1, then $T(n) = O(n^d)$

CASE 2: If a=1, then $T(n) = O(n^{d+1})$

CASE 3: If a>1, then $T(n) = O(n^d \cdot a^{n/b})$