UNIT-I

1.ADT TYPES:

There are several types of ADTs (Abstract Data Types) in programming. Here are some examples and explanations:

1. Stack ADT: A stack is a linear data structure that operates on a LIFO (Last-In, First-Out) principle. The two main operations that can be performed on a stack are "push" (adding an element to the top of the stack) and "pop" (removing the top element from the stack). An example of a stack ADT is the call stack in computer programming, which keeps track of function calls.

2. Queue ADT: A queue is a linear data structure that operates on a FIFO (First-In, First-Out) principle. The two main operations that can be performed on a queue are "enqueue" (adding an element to the end of the queue) and "dequeue" (removing the first element from the queue). An example of a queue ADT is the message queue in operating systems, which stores messages to be sent between processes.

3. List ADT: A list is a data structure that stores an ordered sequence of elements. The main operations that can be performed on a list are "insert" (adding an element to a specific position in the list), "remove" (removing an element from a specific position in the list), and "get" (accessing an element at a specific position in the list). An example of a list ADT is an array or a linked list.

4. Tree ADT: A tree is a data structure that represents a hierarchical structure, with nodes representing elements and edges representing relationships between the elements. The main operations that can be performed on a tree are "insert" (adding a new node to the tree), "delete" (removing a node from the tree), and "traverse" (visiting all the nodes in the tree in a specific order). An example of a tree ADT is the DOM (Document Object Model) in web development, which represents the structure of an HTML document.

5. Map ADT: A map is a data structure that stores a collection of key-value pairs, where each key is associated with a value. The main operations that can be performed on a map are "put" (adding a key-value pair to the map), "get" (retrieving the value associated with a specific key), and "remove" (removing a key-value pair from the map). An example of a map ADT is the dictionary data structure in Python.

Note: Explain with diagram..

2. Date Abstract data type:

A date abstract data type (ADT) is a type of abstract data structure that represents a date. It typically includes components such as year, month, and day, and can provide operations such as comparing two dates, adding or subtracting a number of days, or formatting the date for display.

Here's an example implementation of a date ADT in Python:

```
class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    def __repr__(self):
        return f"{self.year}-{self.month}-{self.day}"

    def __eq__(self, other):
        return self.year == other.year and self.month == other.month and self.day == other.day

    def __lt__(self, other):
        if self.year < other.year:
            return True
        elif self.year == other.year and self.month < other.month:
            return True
        elif self.year == other.year and self.month == other.month and self.day < other.day:
            return True
        else:
            return False

    def add_days(self, days):
        # Implement logic to add days to the date
        pass

    def subtract_days(self, days):
        # Implement logic to subtract days from the date
        pass

    def format(self, format_string):
        # Implement logic to format the date for display
        pass
```

```
date1 = Date(2023, 3, 19)
date2 = Date(2023, 3, 20)
if date1 < date2:
   print("date1 is before date2")
else:
   print("date1 is after date2")
```

output:

date1 is before date2

3.Bags

Bags: A bag ADT is a type of collection that can store elements in an unordered manner. It is similar to a set, but allows for duplicate elements. The main operations that can be performed on a bag include adding elements, removing elements, and checking whether an element is present.

The Bag ADT provides the following basic operations:

add(item): Adds an item to the bag

remove(item): Removes one instance of an item from the bag

count(item): Returns the number of instances of an item in the bag

size(): Returns the number of items in the bag

Here is an example implementation of a Bag ADT in Python:

```
class Bag:
   def __init__(self):
      self.items = {}

   def add(self, item):
      if item in self.items:
         self.items[item] += 1
      else:
         self.items[item] = 1
```

```
    def remove(self, item):
        if item in self.items:
            if self.items[item] == 1:
                del self.items[item]
            else:
                self.items[item] -= 1

    def count(self, item):
        if item in self.items:
            return self.items[item]
        else:
            return 0

    def size(self):
        return sum(self.items.values())
```

In this implementation, the Bag class uses a Python dictionary to keep track of the items in the bag and their counts. The add method adds an item to the bag by either incrementing its count in the dictionary if it already exists, or creating a new entry with a count of 1 if it does not. The remove method removes one instance of an item from the bag by decrementing its count, and deleting it from the dictionary if the count reaches zero. The count method returns the count of a given item in the bag, or zero if it does not exist. The size method returns the total number of items in the bag by summing the values of the dictionary.

Here is an example usage of the Bag class:

```
my_bag = Bag()

my_bag.add("apple")
my_bag.add("banana")
my_bag.add("apple")
my_bag.add("orange")

print(my_bag.count("apple"))  # Output: 2
print(my_bag.count("banana"))  # Output: 1
```

```
print(my_bag.count("pear"))  # Output: 0

my_bag.remove("apple")
print(my_bag.count("apple"))  # Output: 1

print(my_bag.size())  # Output: 3
```

This example demonstrates how a Bag ADT can be used to keep track of a collection of items where duplicates are allowed, such as counting the frequency of words in a document or tracking the number of occurrences of each type of item in a shopping cart.

4.ITERATORS

Iterator in Python is an object that is used to iterate over iterable objects like lists, tuples, dicts, and sets. The iterator object is initialized using the iter() method. It uses the next() method for iteration.

__iter__(): The iter() method is called for the initialization of an iterator. This returns an iterator object
__next__(): The next method returns the next value for the iterable. When we use a for loop to traverse any iterable object, internally it uses the iter() method to get an iterator object, which further uses the next() method to iterate over. This method raises a StopIteration to signal the end of the iteration.

Python iter() Example

```
string = "GFG"
ch_iterator = iter(string)

print(next(ch_iterator))
print(next(ch_iterator))
print(next(ch_iterator))
```

Output :

```
G
F
G
```

This demonstrates how an iterator can be used to traverse through a container of data, such as a list, and access each element one at a time. Iterators are useful because they allow you to work with large data sets without having to load all the data into memory at once. Instead, you can retrieve each element as needed, making it more memory-efficient.

5.ARRAY STRUCTURE

In computer science, an array is a data structure that stores a fixed-size sequential collection of elements of the same type. Arrays are commonly used in programming languages to represent lists or vectors of elements. Each element in an array is identified by an index, which is a non-negative integer that represents the position of the element in the array.

Arrays are often used to store and manipulate large amounts of data in memory. They can be used to represent a variety of data types, including integers, floats, characters, and objects.

There are several types of arrays, including:

One-dimensional arrays: A one-dimensional array is a collection of elements arranged in a linear sequence. Each element is identified by its position in the sequence, which is called its index. One-dimensional arrays are also known as vectors or lists.

Two-dimensional arrays: A two-dimensional array is a collection of elements arranged in a two-dimensional grid. Each element is identified by its row and column indices.

Multi-dimensional arrays: A multi-dimensional array is a collection of elements arranged in a multi-dimensional grid. Each element is identified by its position in the grid, which is represented by a set of indices.

Arrays can be implemented using various programming languages. For example, in Python, arrays can be represented using the built-in list data type. Here is an example of a one-dimensional array in Python:
Eg:

```
my_array = [1, 2, 3, 4, 5]
```

In this example, my_array is a one-dimensional array that contains five integer values.

Here is an example of a two-dimensional array in Python:
my_matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
In this example, my_matrix is a two-dimensional array that contains three rows and three columns of integer values.

Arrays are an important data structure in computer science, and they are used in many applications, including numerical analysis, data processing, and image processing.

6.PYTHON LIST

In Python, a list is a built-in data structure that represents a collection of elements. Lists are similar to arrays in other programming languages, but they are more versatile and powerful.

Lists in Python are ordered and mutable, which means that you can access the elements of a list by their position (index) in the list, and you can add, remove, or modify elements in the list.

Here is an example of a list in Python:
my_list = [1, 2, 3, "four", 5.0]

In this example, my_list is a list that contains five elements, including integers, a string, and a float.

You can access the elements of a list using their index. The first element in the list has an index of 0, the second element has an index of 1, and so on.

Here is an example:
```
my_list = [1, 2, 3, "four", 5.0]
print(my_list[0])  # prints 1
print(my_list[3])  # prints "four"
```

You can add elements to a list using the append() method, remove elements using the remove() method, and modify elements using their index.

Here are some examples:

```
my_list = [1, 2, 3]
my_list.append(4)   # adds 4 to the end of the list
my_list.remove(2)   # removes the element with value 2 from the list
my_list[0] = 0      # modifies the first element of the list to 0
print(my_list)      # prints [0, 3, 4]
```

Lists can also be used in combination with loops and other control structures to perform complex operations on data. For example, you can use a for loop to iterate over the elements of a list and perform some operation on each element.

Here is an example:

```
my_list = [1, 2, 3]
for element in my_list:
    print(element * 2)   # prints 2, 4, 6
```

In addition to the built-in methods and operations, Python provides a rich set of libraries and tools for working with lists and other data structures. Some of the most popular libraries for data manipulation and analysis in Python include NumPy, Pandas, and Matplotlib.

7.TWO DIMENSIONAL ARRAY

In Python, a two-dimensional array is an array of arrays, where each element in the array is itself an array. This is also known as a matrix.

You can create a two-dimensional array in Python using nested lists.

Here is an example:

```
my_matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

In this example, my_matrix is a 3x3 matrix that contains integers. Each row in the matrix is itself a list, and the entire matrix is represented as a list of lists.

You can access the elements of a two-dimensional array using two indices. The first index refers to the row of the element, and the second index refers to the column of the element.

Here is an example:

```
my_matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(my_matrix[0][0])   # prints 1
print(my_matrix[1][2])   # prints 6
```

You can modify the elements of a two-dimensional array in the same way as a one-dimensional array, using their indices.

Here is an example:

```
my_matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
my_matrix[1][0] = 10
print(my_matrix)   # prints [[1, 2, 3], [10, 5, 6], [7, 8, 9]]
```

You can also use loops and other control structures to perform operations on the elements of a two-dimensional array. Here is an example that computes the sum of all the elements in a matrix:

```
my_matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
total = 0
for row in my_matrix:
    for element in row:
        total += element
print(total)   # prints 45
```

In addition to nested lists, there are other data structures and libraries in Python that can be used to represent and manipulate two-dimensional arrays. For example, the NumPy library provides a powerful multi-dimensional array object that is optimized for numerical operations.

8.MATRIX ABSTRACT DATA TYPE

A matrix is a rectangular array of numbers or symbols that are arranged in rows and columns. In computer science, a matrix can be represented as a two-dimensional array, where each element of the array represents a value in the matrix.

Operations that are included in the Matrix ADT:
Initialization, Access, Modification, Addition, Subtraction, Multiplication, Transposition, Determinant, Inverse

Python provides built-in support for matrices using nested lists. You can use nested lists to create and manipulate matrices in Python. You can also use libraries such as NumPy to perform more advanced matrix operations
In Python, you can represent a matrix using a two-dimensional list, where each element in the list is a list of numbers representing a row in the matrix. Here is an example:

matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
This represents a matrix with three rows and three columns, where the element in the first row and first column is 1, the element in the second row and third column is 6, and so on.

To access an element in the matrix, you can use the row and column indices. For example, to access the element in the second row and third column, you can use matrix[1][2].

To add two matrices, you need to add the corresponding elements in each matrix.

Here is an example:
matrix1 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
matrix2 = [[9, 8, 7], [6, 5, 4], [3, 2, 1]]
result = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]

```
# iterate through each element in the matrices and add them
for i in range(len(matrix1)):
    for j in range(len(matrix1[0])):
        result[i][j] = matrix1[i][j] + matrix2[i][j]

print(result)
```

This will output the result of adding the two matrices:
[[10, 10, 10], [10, 10, 10], [10, 10, 10]]

You can also perform other operations on matrices, such as multiplying them or transposing them. There are also libraries in Python, such as NumPy, that provide more advanced matrix operations and optimizations.

9.SETS AN MAP:

Sets and maps are two important abstract data types in data structures and algorithms.

A set is a collection of distinct elements, where the order of the elements does not matter. Sets are often used to perform operations such as union, intersection, and difference. In Python, sets can be represented using curly braces {} or the set() function.

Here are some common operations that are included in the Set ADT:
Initialization: This operation creates a new set and initializes it with default values.
Insertion: This operation adds a new element to the set.
Deletion: This operation removes an element from the set.
Membership test: This operation checks if an element is present in the set.
Union: This operation creates a new set that contains all the elements from both sets.
Intersection: This operation creates a new set that contains only the elements that are present in both sets.
Difference: This operation creates a new set that contains only the elements that are present in one set but not the other.

A map, also known as a dictionary, is a collection of key-value pairs. Maps are often used to store and retrieve data using a unique key. In Python, dictionaries can be represented using curly braces {} or the dict() function.

Here are some common operations that are included in the Map ADT:
Initialization: This operation creates a new map and initializes it with default values.
Insertion: This operation adds a new key-value pair to the map.
Deletion: This operation removes a key-value pair from the map.
Lookup: This operation retrieves the value associated with a given key.
Update: This operation updates the value associated with a given key.
Membership test: This operation checks if a key is present in the map.
Iteration: This operation allows you to iterate over the keys or values in the map.

Sets and maps are useful data structures for solving a wide range of problems in computer science, such as searching, sorting, and graph algorithms.

Implementation of sets:
In Python, sets can be implemented using curly braces {} or the built-in set() function. Here are some examples of how to create and manipulate sets in Python:

Creating a set:
```
set1 = {1, 2, 3, 4, 5}   # using curly braces
set2 = set([4, 5, 6, 7]) # using the set() function
```

Adding elements to a set:
```
set1.add(6)
```

Removing elements from a set:
```
set2.remove(6)
```

Checking if an element is present in a set:
```
if 5 in set1:
    print("5 is present in set1")
```

Performing set operations such as union, intersection, and difference:
```
set3 = set1.union(set2)         # union of set1 and set2
set4 = set1.intersection(set2)  # intersection of set1 and set2
set5 = set1.difference(set2)    # set difference of set1 and set2
```

Iterating over elements in a set:
```
for element in set1:
    print(element)
```

Sets are useful for solving problems such as removing duplicates from a list or checking if two lists have any common elements. They are also used in graph algorithms to represent nodes or edges.

Implementation of maps

In Python, maps can be implemented using dictionaries. Here are some examples of how to create and manipulate maps in Python:

Creating a map:

```
map1 = {'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}
```

Adding or updating values in a map:

```
map1['key4'] = 'value4'      # add a new key-value pair
map1['key2'] = 'new_value2'  # update an existing value
```

Removing a key-value pair from a map:

```
del map1['key3']
```

Checking if a key is present in a map:

```
if 'key2' in map1:
    print("key2 is present in map1")
```

Getting the value associated with a key:

```
value1 = map1['key1']
```

Iterating over keys, values, or key-value pairs in a map:

```
for key in map1:
    print(key)

for value in map1.values():
    print(value)

for key, value in map1.items():
    print(key, value)
```

Maps are useful for solving problems such as counting the frequency of elements in a list or storing metadata associated with objects. They are also used in graph algorithms to represent weighted edges or attributes of nodes

10.MULTI-DUNENSIONAL ARRAY:

The MultiArray abstract data type is a generalization of the one-dimensional array and the two-dimensional matrix to arrays with any number of dimensions. In other words, a multiarray can have any number of indices, with each index ranging over a certain range of values.

The multiarray data structure is particularly useful for representing tensors and other multidimensional data structures used in scientific computing, machine learning, and computer graphics.

Here's an example implementation of a multiarray in Python using nested lists:

```
class MultiArray:
    def __init__(self, shape):
        self.shape = shape
        self.data = self._create_array(shape)

    def _create_array(self, shape):
        if len(shape) == 1:
            return [0] * shape[0]
        else:
            return [self._create_array(shape[1:]) for i in range(shape[0])]

    def __getitem__(self, index):
        if len(index) == 1:
            return self.data[index[0]]
        else:
            return self.data[index[0]][index[1:]]

    def __setitem__(self, index, value):
        if len(index) == 1:
            self.data[index[0]] = value
        else:
            self.data[index[0]][index[1:]] = value
```

In this implementation, the shape argument is a tuple representing the size of each dimension of the multiarray. The _create_array() method uses recursion to create nested lists of the appropriate shape to store the data.

The __getitem__() and __setitem__() methods allow us to access and modify elements of the multiarray using index tuples.

For example, we can create a 3-dimensional multiarray of shape (2, 3, 4) as follows:
m = MultiArray((2, 3, 4))

We can then set and get values as follows:
m[0, 1, 2] = 42
print(m[0, 1, 2]) # output: 42
This implementation can be extended to support other operations on multiarrays, such as reshaping, transposing, and matrix multiplication.

UNIT-2

ALGORITHM ANALYSIS

Algorithm is step by step process.

- Algorithm analysis involves studying an algorithm's efficiency and performance characteristics. This is done by evaluating the algorithm's time and space complexity, which refer to the amount of time and memory required to complete the task. (For example: - Going from one place to another, there is various way to travel. Like Bus, Train, Flight, Car, etc.)
- Time complexity is measured using big O notation, which describes the upper bound of the algorithm's time requirements, while space complexity is measured in terms of the upper bound of the algorithm's memory requirements. By analyzing these factors, we can determine the algorithm's efficiency and scalability for handling real-world problems with large data sets.

1. EXPERIMENTAL STUDIES:

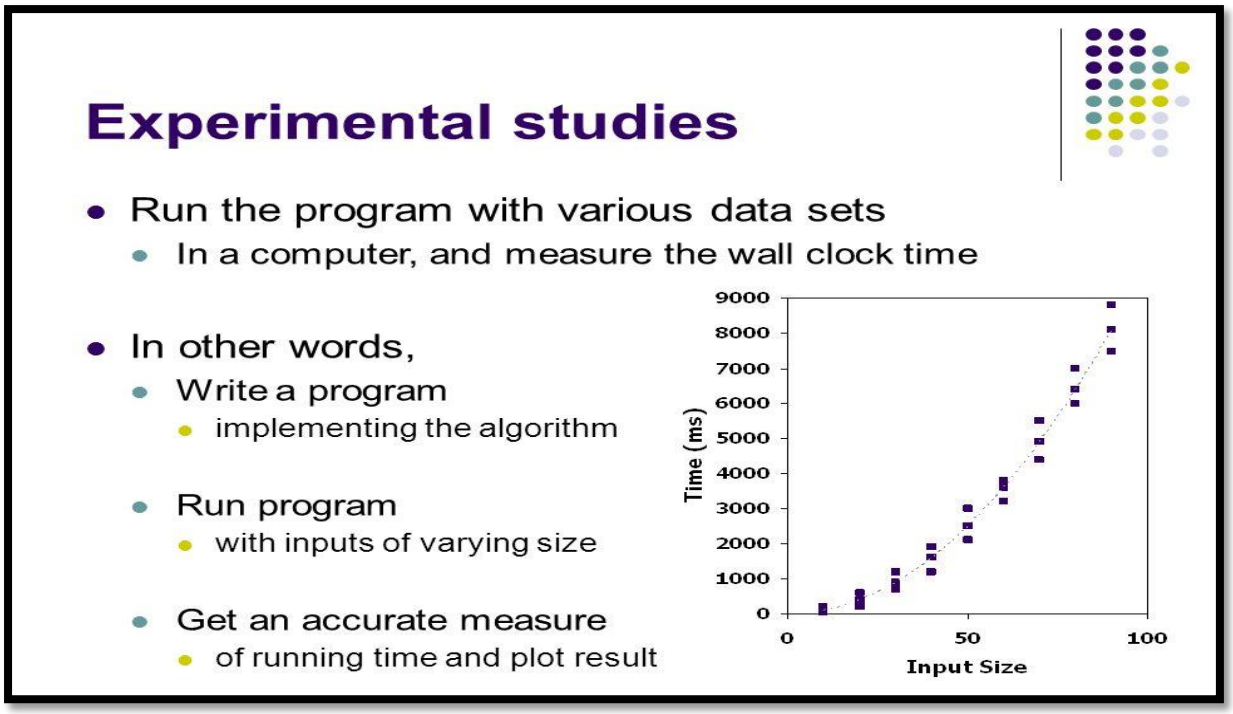
- Experimental studies are often used in algorithm analysis to evaluate the performance of algorithms in practice. These studies involve running the algorithm on real-world data sets and measuring its actual performance in terms of time and memory requirements.
- Example of an experimental study in algorithm analysis:
    - Compare the performance of two different sorting algorithms - quicksort and merge sort - in terms of their time complexity and memory usage for a large dataset.

Experimental studies are an important part of evaluating algorithms and help to inform decisions about which algorithm to use in different

scenarios. They can also help to identify areas where an algorithm can be improved or optimized to better handle real-world data sets.

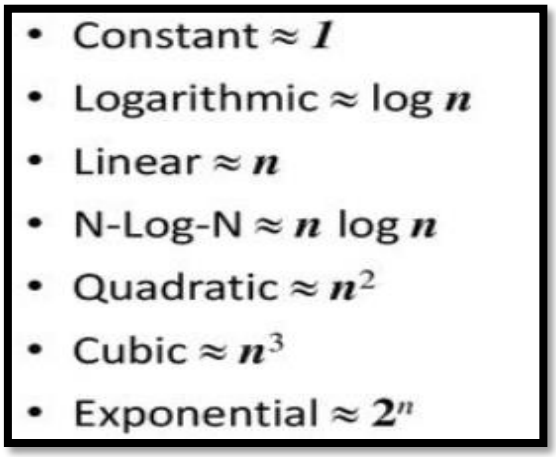
Limitations of Experiments:

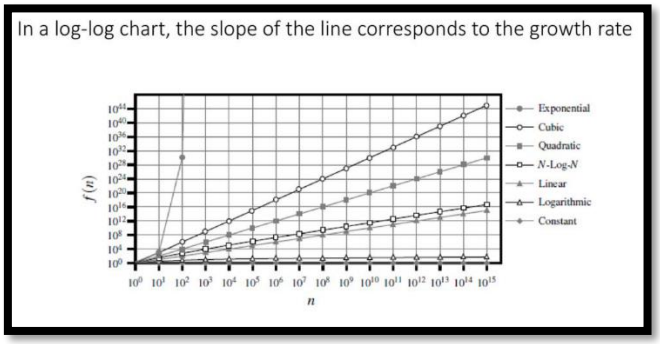
- It is necessary to implement the algorithm, which may be difficult.
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used.



2. SEVEN FUNCTIONS:

- When analyzing algorithms, there are several key functions or concepts that are commonly used to describe their behavior and performance:
    - The Constant Function.
    - The Logarithm Function.
    - The Linear Function.
    - The N-Log-N Function.
    - The Quadratic Function.
    - The Cubic Function and Other Polynomials.
    - The Exponential Function.





Constant function:

- The simplest function we can think of is the *constant function*. This is the function,
- As simple as it is, the constant function is useful in algorithm analysis, because it characterizes the number of steps needed to do a basic operation on a computer, like adding two numbers, assigning a value to some variable, or comparing two numbers.

$$f(n) = c$$

Logarithm function:

- An algorithm that has a logarithmic time complexity has a running time that increases slowly as the input size increases. For example, performing a binary search on a sorted array has a logarithmic time complexity.

$$f(n) = O(\log n)$$

Linear function:

- An algorithm analysis, a linear function is a function that has a time complexity of O(n), where n represents the size of the input data.

$$f(n) = n$$

N-log-n function:

- A running time that is directly proportional to the input size. For example, iterating over every element in an array has a linear time complexity.

$$f(n) = n \log n$$

Quadratic function:

- Quadratic time (O(n^2)): An algorithm that has a quadratic time complexity has a running time that increases rapidly as the input size increases. For example, performing a nested loop operation on an array has a quadratic time complexity.
    - $f(n) = n^2$

Cubic function and other polynomials:

- Continuing our discussion of functions that are powers of the input, we consider the *cubic function*,
    - $f(n) = n^3$
- **Polynomials:**
    - Most of the functions we have listed so far can each be viewed as being part of a larger class of functions, the *polynomials*. A *polynomial* function has the form,
        - $f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \cdots + a_d n^d$

Exponential function:

- Exponential time (O(2^n)): An algorithm that has an exponential time complexity has a running time that doubles with each additional input element. This type of algorithm can quickly become impractical for large data sets.
    - $f(n) = b^n$

3. ASYMPTOTIC ANALYSIS

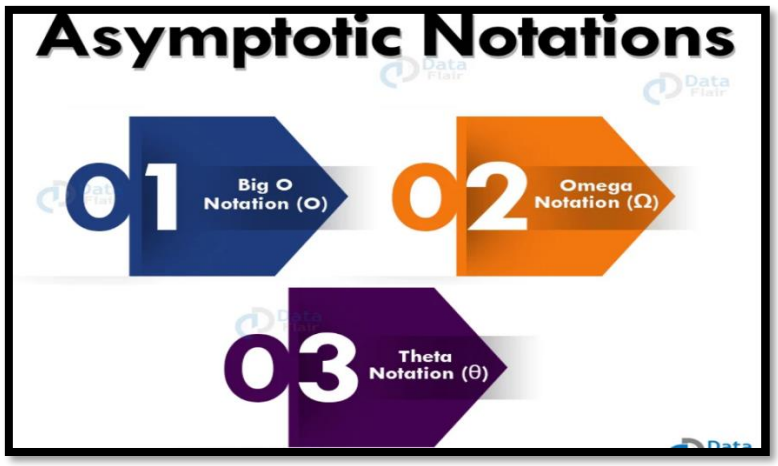
- Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst-case scenario of an algorithm.
- Usually, the time required by an algorithm falls under three types:

- Best Case − Minimum time required for program execution.
- Average Case − Average time required for program execution.
- Worst Case − Maximum time required for program execution.

- Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation, Ω Notation, θ Notation



- O Notation (less than):
  - $f(n) \leq cg(n)$ implies (f(n) ‘’< ‘’g(n))
- Ω Notation (greater than):
  - $f(n) \geq cg(n)$ implies (f(n) ‘’> ‘’g(n))
  - θ Notation (equal to):
  - $cg(n) \leq f(n) \leq cg(n)$ implies (f(n) ‘’=‘’ g(n))

4. RECURSION:

Recursion is a technique by which a function makes one or more calls to itself during execution, or by which a data structure relies upon smaller instances of the very same type of structure in its representation. There are many examples of recursion in art and nature. For example, fractal patterns are naturally recursive. A physical example of recursion used in art is in the Russian Matryoshka dolls. Each doll is either made of solid wood, or is hollow and contains another Matryoshka doll inside it.

1. The Factorial Function

A Recursive Implementation of the Factorial Function Recursion is not just a mathematical notation; we can use recursion to design a Python implementation of a factorial function

```
1 def factorial(n):
2     if n == 0:
3         return 1
4     else:
5         return n factorial(n−1)
```

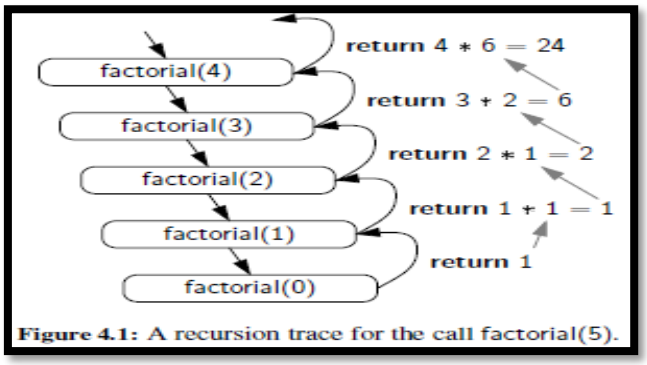


Figure 4.1: A recursion trace for the call factorial(5).

2. Drawing an English Ruler:

An ***English ruler*** has a recursive pattern that is a simple example of a fractal structure.

A Recursive Approach to Ruler Drawing

The English ruler pattern is a simple example of a ***fractal***, that is, a shape that has a self-recursive structure at various levels of magnification.

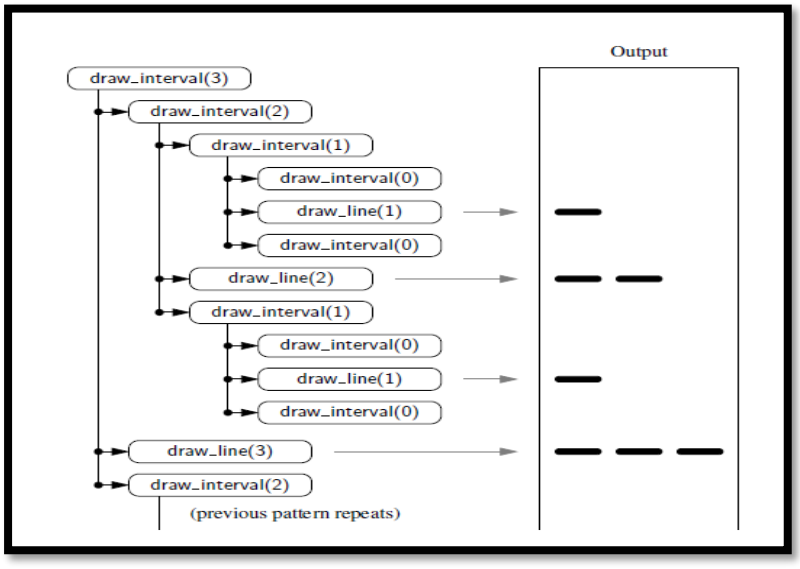
In general, an interval with a central tick length $L \geq 1$ is composed of:

- An interval with a central tick length $L-1$.
- A single tick of length L.
- An interval with a central tick length $L-1$.

The interesting work is done by the recursive draw interval function. This function draws the sequence of minor ticks within some interval, based upon the length of the interval's central tick. We rely on the intuition shown at the top of this page, and with a base case when $L = 0$ that draws nothing. For $L \geq 1$, the first and last steps are performed by recursively calling draw interval($L-1$). The middle step is performed by calling the function draw line(L).

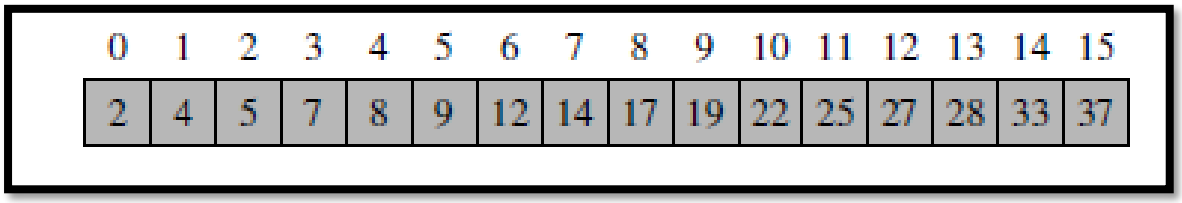
Illustrating Ruler Drawing Using a Recursion Trace

The execution of the recursive draw_interval function can be visualized using a recursion trace. The trace for draw interval is more complicated than in the factorial example, however, because each instance makes two recursive calls. To illustrate this, we will show the recursion trace in a form that is reminiscent of an outline for
a document.



3. Binary Search

In this section, we describe a classic recursive algorithm, ***binary search***, that is used to efficiently locate a target value within a sorted sequence of *n* elements. This is among the most important of computer algorithms, and it is the reason that we so often store data in sorted order



We then compare the target value to the median candidate, that is, the item data[mid] with index

mid = _(low+high)/2_

We consider three cases:

• If the target equals data[mid], then we have found the item we are looking for, and the search terminates successfully.

• If target < data[mid], then we recur on the first half of the sequence, that is,on the interval of indices from low to mid−1.

• If target > data[mid], then we recur on the second half of the sequence, that is, on the interval of indices from mid+1 to high.

An unsuccessful search occurs if low > high, as the interval [low,high] is empty.

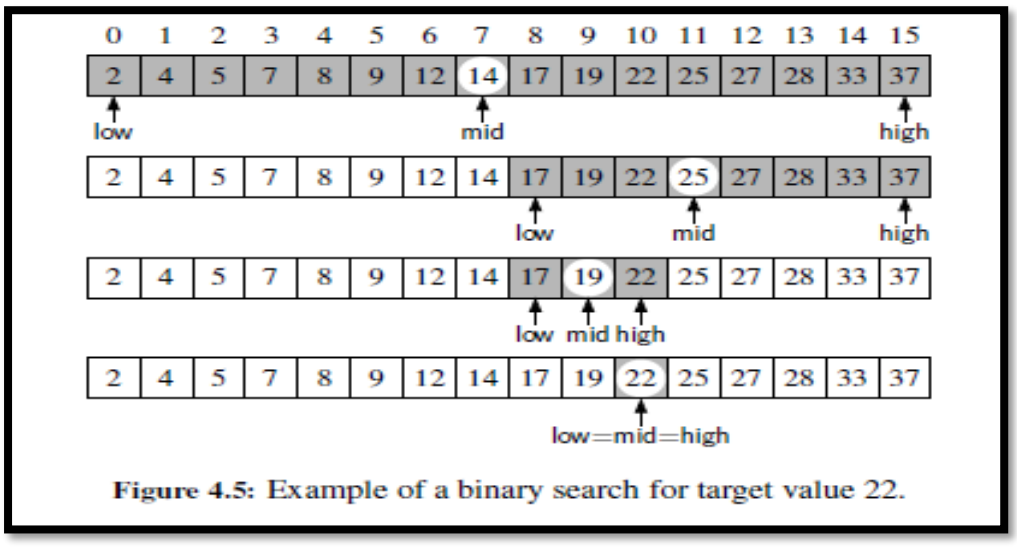


Figure 4.5: Example of a binary search for target value 22.

4.File Systems

The ***file system*** for a computer has a recursive structure in which directories can be nested arbitrarily deeply within other directories. Recursive algorithms are widely used to explore and manage these file systems.

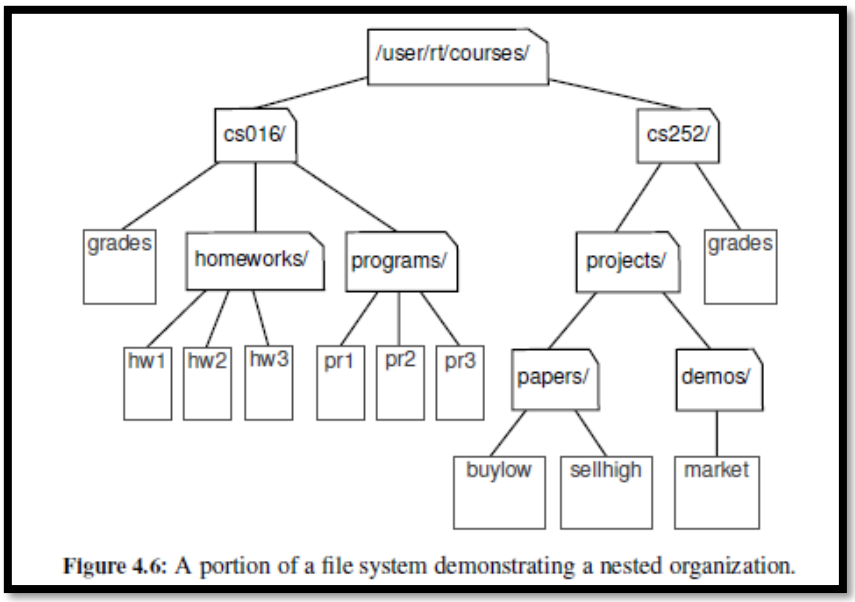


Figure 4.6: A portion of a file system demonstrating a nested organization.

Python's os Module:

To provide a Python implementation of a recursive algorithm for computing disk usage, we rely on Python's os module, which provides robust tools for interacting with the operating system during the execution of a program. This is an extensive library, but we will only need the following four functions:

- **os.path.getsize(path)**

Return the immediate disk usage (measured in bytes) for the file or directory that is identified by the string path (e.g., /user/rt/courses).

- **os.path.isdir(path)**

Return True if entry designated by string path is a directory; False otherwise.

- **os.listdir(path)**

Return a list of strings that are the names of all entries within a directory designated by string path. In our sample file system, if the parameter is /user/rt/courses, this returns the list [cs016, cs252].

- **os.path.join(path, filename)**

Compose the path string and filename string using an appropriate operating system separator between the two (e.g., the / character for a Unix/Linux system, and the \ character for Windows). Return the string that represents the full path to the file.

```
1  import os
2
3  def disk_usage(path):
4    """Return the number of bytes used by a file/folder and any descendents."""
5    total = os.path.getsize(path)                # account for direct usage
6    if os.path.isdir(path):                      # if this is a directory,
7      for filename in os.listdir(path):          # then for each child:
8        childpath = os.path.join(path, filename) # compose full path to child
9        total += disk_usage(childpath)           # add child's usage to total
10
11   print ('{0:<7}'.format(total), path)         # descriptive output (optional)
12   return total                                 # return the grand total
```

Code Fragment 4.5: A recursive function for reporting disk usage of a file system.

5. LINEAR RECURSION:

A linear recursion is a recursive function that calls itself exactly once in each recursive call. Here's an example of a Python implementation of a linear recursion for computing the nth Fibonacci number:

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

In this implementation, the function **fibonacci** takes a single argument **n**, which represents the index of the Fibonacci number to be computed. If **n** is less than 2, the function returns **n** directly. Otherwise, the function makes two recursive calls to itself with arguments **n-1** and **n-2**, and returns their sum as the result.

It's worth noting that linear recursions like this can be very inefficient for large values of **n**, since the function will make many duplicate recursive calls. To improve performance, you can use memoization to cache the results of previous calls and avoid redundant computations. Here's an example of a memoized version of the **fibonacci** function:

```
def fibonacci(n, memo={}):
    if n < 2:
        return n
    elif n in memo:
        return memo[n]
    else:
        memo[n] = fibonacci(n-1, memo) + fibonacci(n-2, memo)
        return memo[n]
```

In this version, we've added an optional second argument **memo**, which defaults to an empty dictionary. We use this dictionary to store the results of previous calls to **fibonacci**, so that if the function is called again with the same argument, we can simply return the cached result instead of recomputing it. This can greatly improve the performance of the function for large values of **n**.

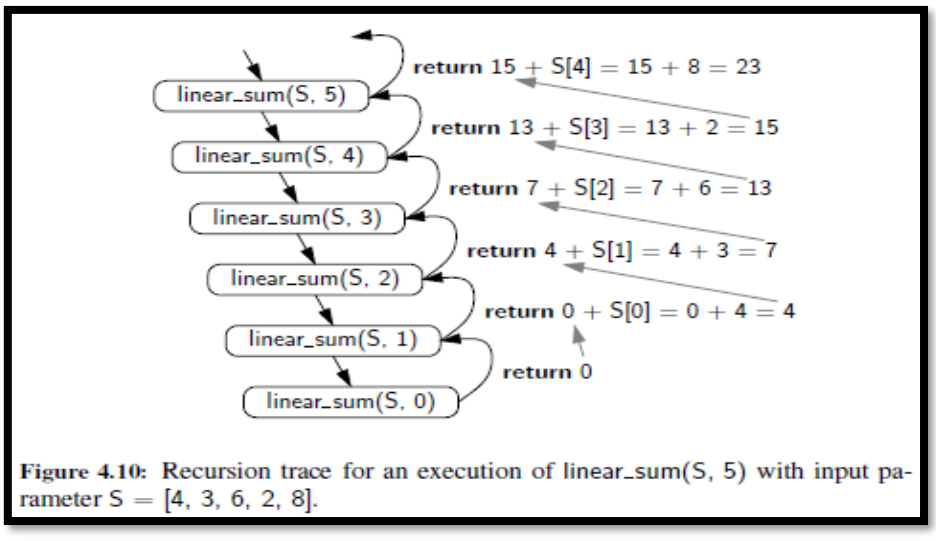


Figure 4.10: Recursion trace for an execution of linear_sum(S, 5) with input parameter S = [4, 3, 6, 2, 8].

6. BINARY RECURSION

When a function makes two recursive calls, we say that it uses ***binary recursion***.
We have already seen several examples of binary recursion, most notably when drawing the English ruler (Section 4.1.2), or in the bad fibonacci function of Section. As another application of binary recursion, let us revisit the problem of summing the n elements of a sequence, S, of numbers. Computing the sum of one or zero elements is trivial. With two or more elements, we can recursively compute the sum of the first half, and the sum of the second half, and add these sums together. Our implementation of such an algorithm, in Code Fragment, is initially invoked as binary sum(A, 0, len(A)).

```
1 def binary sum(S, start, stop):
2""" Return the sum of the numbers in implicit slice
S[start:stop]."""
3       if start >= stop: # zero elements in slice
4               return 0
5       elif start == stop−1: # one element in slice
6               return S[start]
7       else: # two or more elements in slice
8               mid = (start + stop) // 2
9               return binary sum(S, start, mid) + binary sum(S, mid,
                stop)
```

Code Fragment: Summing the elements of a sequence using binary recursion.

To analyze algorithm binary sum, we consider, for simplicity, the case where n is a power of two. Figure 4.13 shows the recursion trace of an execution of binary sum(0, 8). We label each box with the values of parameters start:stop for that call. The size of the range is divided in half at each recursive call, and so the depth of the recursion is 1+log2 n. Therefore, binary sum uses $O(\log n)$ amount of additional space, which is a big improvement over the $O(n)$ space used

by the linear sum function of Code Fragment 4.9. However, the running time of binary sum is $O(n)$, as there are $2n-1$ function calls, each requiring constant time.

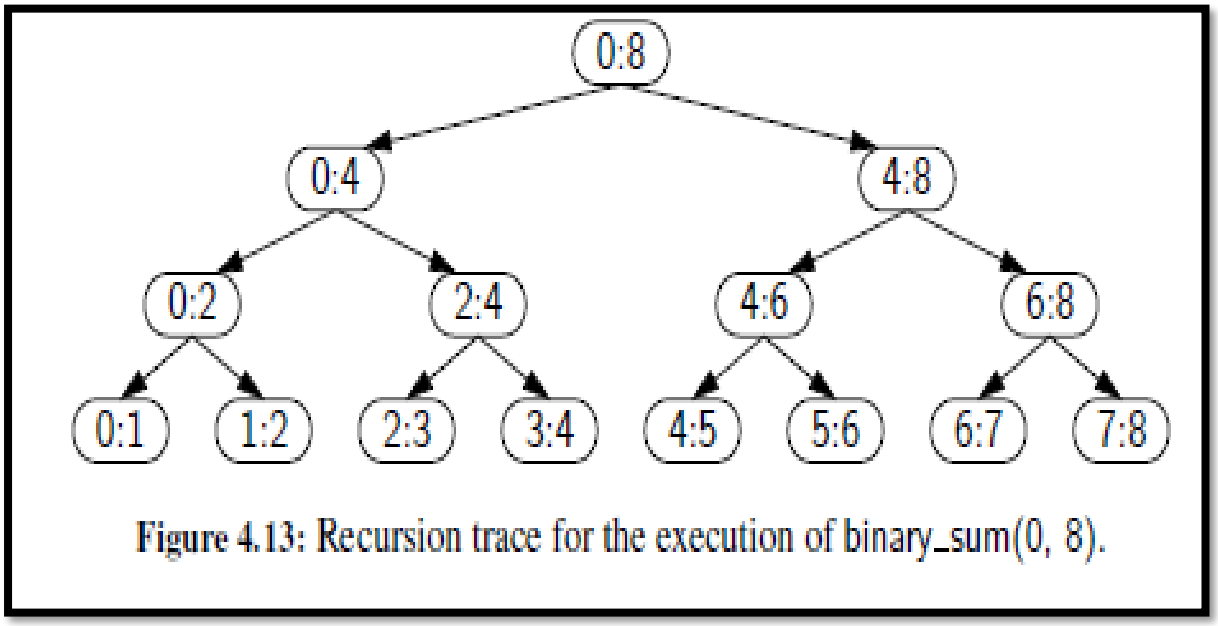


Figure 4.13: Recursion trace for the execution of binary_sum(0, 8).

7. MULTIPLE RECURSION:

Multiple recursion is a type of recursion in which a function makes more than one recursive call in each recursive step. This means that the function will recurse down multiple paths at the same time. Multiple recursion can be used to solve problems that can be expressed as recursive tree-like structures. However, it can be quite expensive for large inputs and may require additional optimization techniques to improve its performance.

One example of a function that uses multiple recursion is the Tower of Hanoi problem. In this problem, there are three pegs and a set of disks of varying sizes, initially placed in a stack on one peg. The objective is to move the entire stack to another peg while obeying the following rules:

1.Only one disk can be moved at a time.

2.A disk can only be moved if it is the uppermost disk on a peg.

3.A larger disk cannot be placed on top of a smaller disk.

Another common application of multiple recursion is when we want to enumerate various configurations in order to solve a combinatorial puzzle. For example,the following are all instances of what are known as *summation puzzles*:

pot + pan = bib
dog + cat = pig
boy+ girl = baby

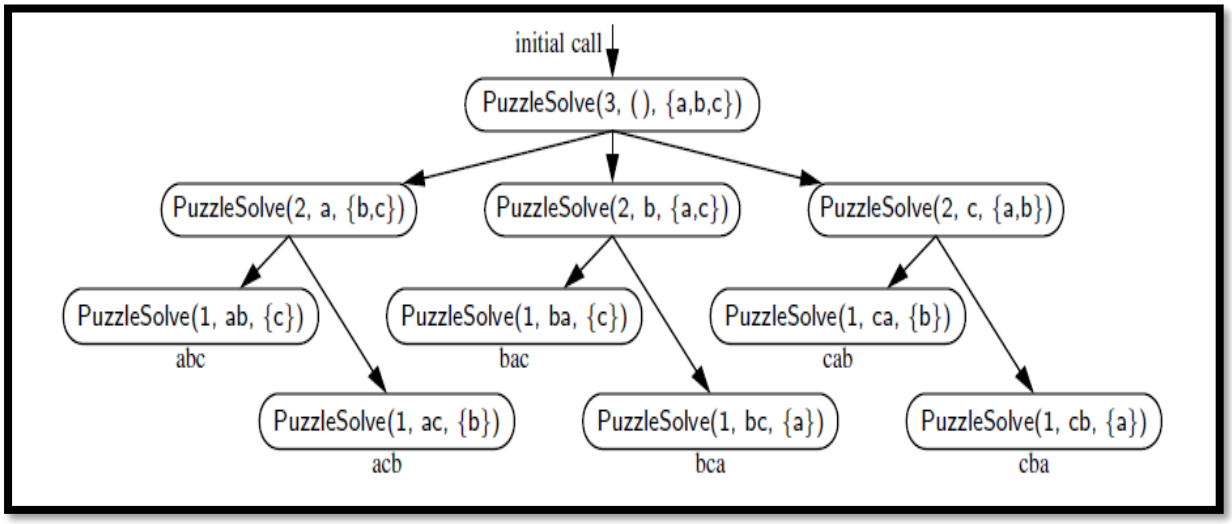
Here's a Python implementation of a function that uses multiple recursions to solve the Tower of Hanoi problem:

```
def hanoi(n, source, target, aux):
    if n > 0:
        hanoi(n-1, source, aux, target)
        print(f"Move disk {n} from {source} to {target}")
        hanoi(n-1, aux, target, source)
```

In this implementation, the **hanoi** function takes four arguments: **n**, the number of disks; **source**, the peg where the disks are initially placed; **target**, the peg where the disks should be moved; and **aux**, an auxiliary peg used for intermediate moves. The function uses multiple recursion to move the stack of disks recursively from the source peg to the target peg.

The advantage of using multiple recursion is that it allows the function to explore all possible paths of the recursive tree-like structure in parallel, potentially reducing the amount of time required to solve the problem. However, the disadvantage is that it can be computationally expensive for large inputs, since the function will make many recursive calls. To optimize multiple recursion, we can use memoization, dynamic programming, or other techniques to avoid redundant computations and reduce the number of recursive calls required.

In conclusion, multiple recursion is a powerful technique for solving problems that can be expressed as recursive tree-like structures. However, it can be quite expensive for large inputs and may require additional optimization techniques to improve its performance. The Tower of Hanoi problem is an example of a problem that can be solved using multiple recursion, and requires a deep understanding of the recursive structure of the problem to find an optimal solution.



UNIT-3

1.STACK:

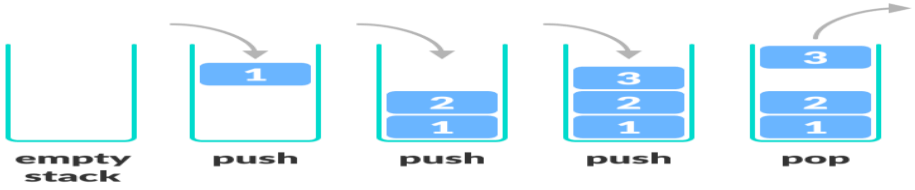
A stack is a linear data structure that follows the principle of **Last In First Out (LIFO)**. This means the last element inserted inside the stack is removed first.

You can think of the stack data structure as the pile of plates on top of another.

Here, you can:
Put a new plate on top
Remove the top plate
And, if you want the plate at the bottom, you must first remove all the plates on top. This is exactly how the stack data structure works.

LIFO Principle of Stack

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



In the above image, although item **3** was kept last, it was removed first. This is exactly how the **LIFO (Last In First Out) Principle** works.

We can implement a stack in any programming language like C, C++, Java, Python or C#, but the specification is pretty much the same.

Basic Operations of Stack

There are some basic operations that allow us to perform different actions on a stack.

Push: Add an element to the top of a stack

Pop: Remove an element from the top of a stack

IsEmpty: Check if the stack is empty

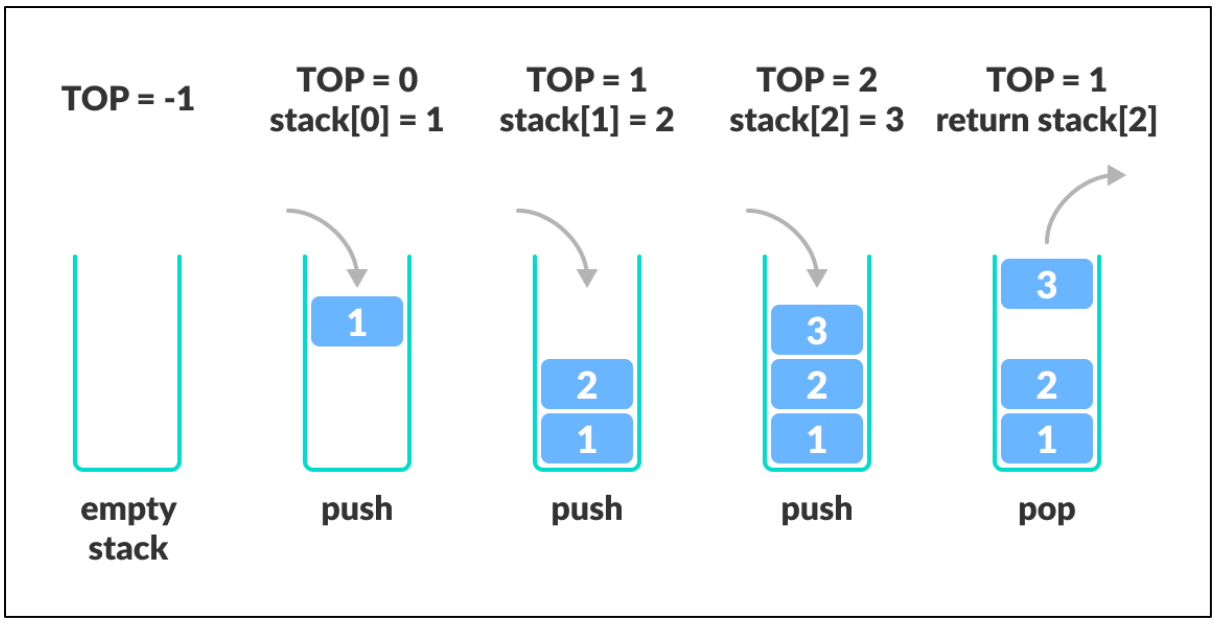
IsFull: Check if the stack is full

Peek: Get the value of the top element without removing it

Working of Stack Data Structure

The operations work as follows:

- ✓ A pointer called `TOP` is used to keep track of the top element in the stack.
- ✓ When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing `TOP == -1`.
- ✓ On pushing an element, we increase the value of `TOP` and place the new element in the position pointed to by `TOP`.
- ✓ On popping an element, we return the element pointed to by `TOP` and reduce its value.
- ✓ Before pushing, we check if the stack is already full
- ✓ Before popping, we check if the stack is already empty



Stack Implementations in Python

```
def create_stack():
    stack = []
    return stack
def check_empty(stack):
    return len(stack) == 0
def push(stack, item):
    stack.append(item)
    print("pushed item: " + item)
def pop(stack):
    if (check_empty(stack)):
        return "stack is empty"
    return stack.pop()

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

Stack Time Complexity

For the array-based implementation of a stack, the push and pop operations take constant time, i.e. O(1).

Applications of Stack Data Structure

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

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

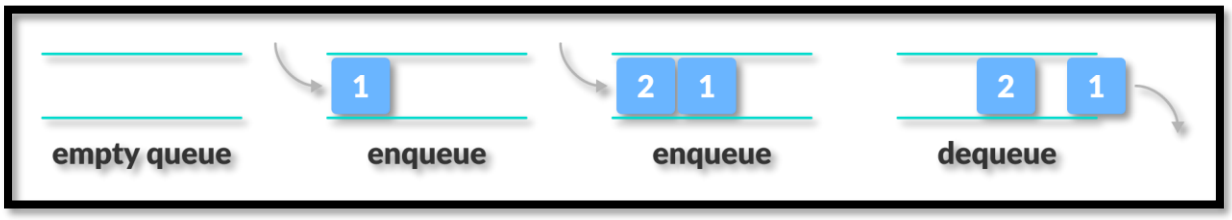
In compilers - Compilers use the stack to calculate the value of expressions like 2 + 4 / 5 * (7 - 9) by converting the expression to prefix or postfix form.

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

2.QUEUES:

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

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



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

In programming terms, putting items in the queue is called **enqueue**, and removing items from the queue is called **dequeue**.

We can implement the queue in any programming language like C, C++, Java, Python or C#, but the specification is pretty much the same.

Basic Operations of Queue

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

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

Working of Queue

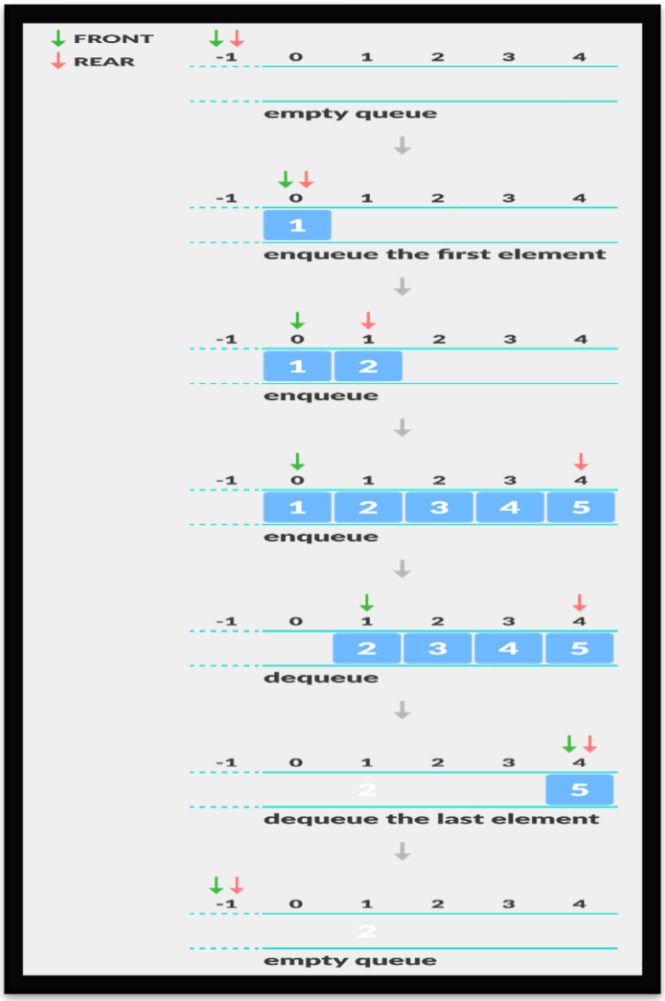
Queue operations work as follows:

- ✓ two pointers `FRONT` and `REAR`
- ✓ `FRONT` track the first element of the queue
- ✓ `REAR` track the last element of the queue
- ✓ initially, set value of `FRONT` and `REAR` to -1

Enqueue Operation

- ✓ check if the queue is full
- ✓ for the first element, set the value of `FRONT` to 0
- ✓ increase the `REAR` index by 1

✓ add the new element in the position pointed to by REAR



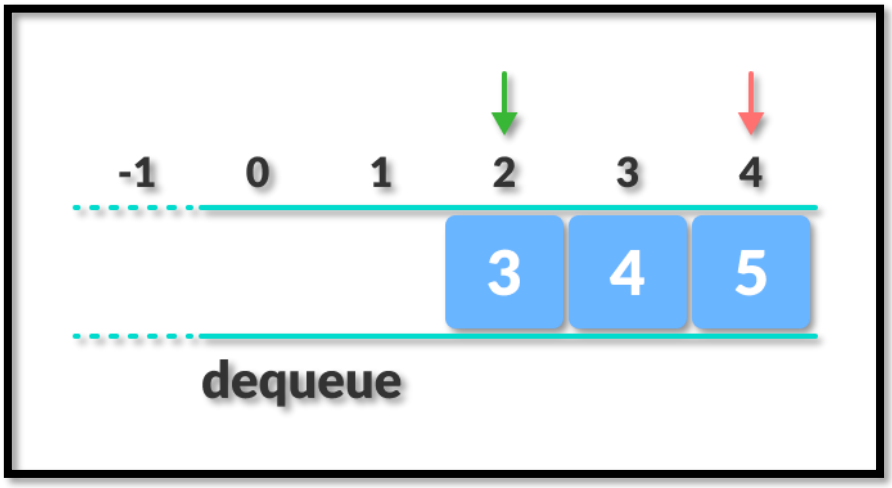
Queue Implementations in Python

```
# Queue implementation in Python

class Queue:

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

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

    # Remove an element
```

```
    def dequeue(self):
        if len(self.queue) < 1:
            return None
        return self.queue.pop(0)
    # Display  the queue
    def display(self):
        print(self.queue)

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

q = Queue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
q.enqueue(4)
q.enqueue(5)
q.display()
q.dequeue()
print("After removing an element")
q.display()
```

Limitations of Queue

As you can see in the image below, after a bit of enqueuing and dequeuing, the size of the queue has been reduced.



And we can only add indexes 0 and 1 only when the queue is reset (when all the elements have been dequeued).

After REAR reaches the last index, if we can store extra elements in the empty spaces (0 and 1), we can make use of the empty spaces. This is implemented by a modified queue called the circular queue.

Complexity Analysis

The complexity of enqueue and dequeue operations in a queue using an array is O(1). If you use pop(N) in python code, then the complexity might be O(n) depending on the position of the item to be popped.

Applications of Queue

- ✓ CPU scheduling, Disk Scheduling
- ✓ When data is transferred asynchronously between two processes.The queue is used for synchronization. For example: IO Buffers, pipes, file IO, etc
- ✓ Handling of interrupts in real-time systems.
- ✓ Call Center phone systems use Queues to hold people calling them in order.

3.ENDED QUEUES LINKED:

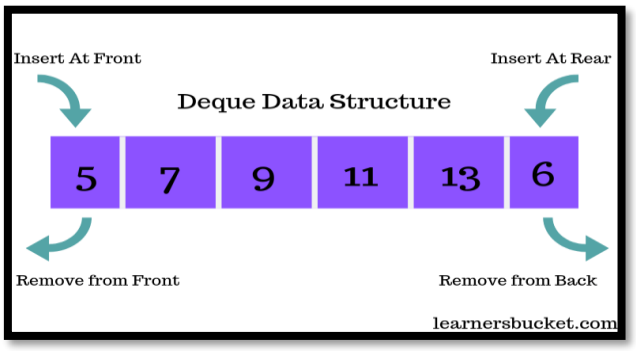
A double-ended queue (deque) can be implemented using a linked list data structure. In a linked list implementation, each node of the linked list will contain a value and two pointers: one pointing to the next node in the list and another pointing to the previous node in the list. This allows for efficient insertion and deletion at both the front and the back of the deque.

A deque supports the following operations:

- ✓ **add_front(value)**: Insert a value at the front of the deque.
- ✓ **add_back(value)**: Insert a value at the back of the deque.
- ✓ **remove_front()**: Remove and return the value at the front of the deque.
- ✓ **remove_back()**: Remove and return the value at the back of the deque.
- ✓ **is_empty()**: Return **True** if the deque is empty, **False** otherwise.
- ✓ **size()**: Return the number of elements in the deque.

The time complexity of each operation depends on the implementation. For example, a deque implemented using an array has constant time complexity for adding or removing elements from the front or back of the deque, but linear time complexity for accessing or removing elements from the middle of the deque. On the other hand, a deque implemented using a linked list has constant time complexity for adding or removing elements from any position in the deque, but requires more memory overhead than an array-based implementation

Here is an example implementation of a deque



In this implementation, the **Node** class represents a node in the linked list with a value and two pointers. The **Deque** class represents the deque itself and has methods to add or remove elements at either end of the deque. The **is_empty** method checks if the deque is empty, the **add_front** and **add_back** methods add elements to the front and back of the deque, respectively, and the **remove_front** and **remove_back** methods remove elements from the front and back of the deque, respectively.

Note that in this implementation, adding or removing elements from the front or back of the deque takes constant time O(1), since it only involves updating a few pointers. However, accessing or removing elements from the middle of the deque takes linear time O(n), since it requires traversing the linked list from one end to the middle.

4.SINGLE LINKED LIST:

A singly linked list is a data structure that consists of a sequence of nodes, where each node stores a value and a pointer to the next node in the list. The first node in the list is called the head node, and the last node is called the tail node.

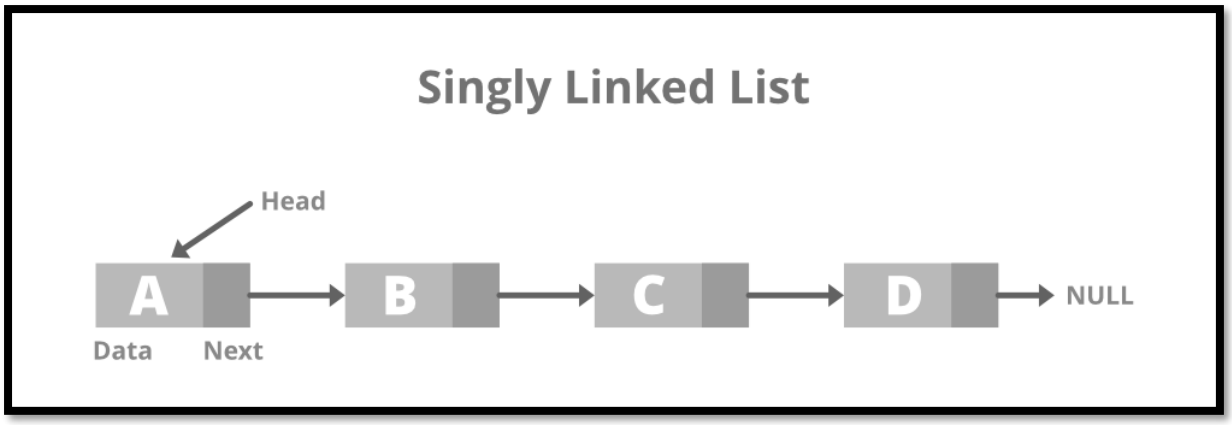
To traverse a singly linked list, we start at the head node and follow the next pointers until we reach the end of the list (i.e., the tail node).

The main advantage of a singly linked list over an array is that inserting or deleting an element from the list requires only O(1) time, regardless of the size of the list. This is because we only need to update the next pointer of the preceding node or the node being deleted. In contrast, inserting or deleting an element from an array requires shifting all subsequent elements, which takes O(n) time in the worst case.

However, a singly linked list has some limitations. For example, it does not support constant-time access to elements by index, since we need to traverse the list to find the desired element. Moreover, it cannot be easily traversed in reverse order, since each node only has a pointer to the next node.

Here are the key operations that can be performed on a singly linked list:

- ✓ insert_front(value): Create a new node with the given value and insert it at the front of the list (i.e., make it the new head node).
- ✓ insert_back(value): Create a new node with the given value and insert it at the back of the list (i.e., make it the new tail node).
- ✓ insert_after(node, value): Create a new node with the given value and insert it after the given node.
- ✓ delete_front(): Remove and return the head node of the list.
- ✓ delete_back(): Remove and return the tail node of the list.
- ✓ delete_after(node): Remove the node that comes after the given node.
- ✓ search(value): Traverse the list and return the first node that contains the given value (or None if the value is not found).
- ✓ size(): Traverse the list and count the number of nodes in it.
- ✓ is_empty(): Return True if the list is empty, False otherwise.



5. CIRCULARLY LINKED LIST:

A circularly linked list is a variation of a singly linked list in which the last node points back to the first node instead of to None. This means that the list forms a loop or circle, hence the name.

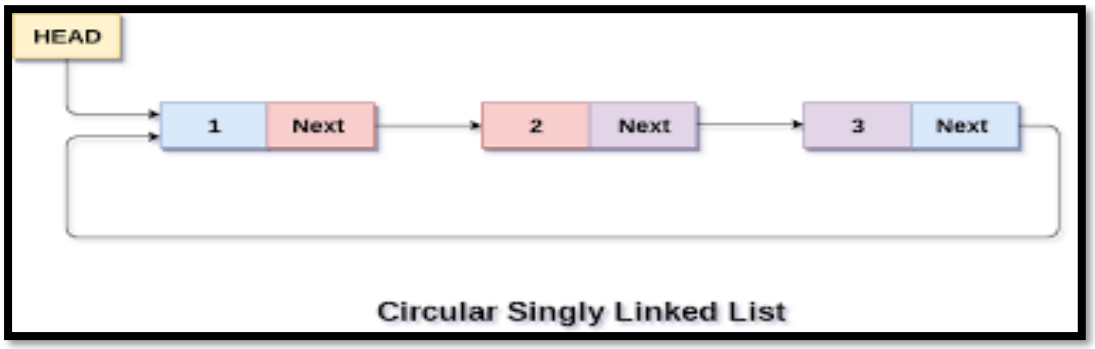
To traverse a circularly linked list, we can start at any node and follow the next pointers until we return to the starting node. Since the last node points back to the first node, we can also traverse the list starting at the tail node and moving backwards using the previous pointers.

The key advantage of a circularly linked list over a singly linked list is that it provides constant-time access to the last node of the list. In a singly linked list, we need to traverse the entire list to reach the last node, which takes linear time.

Here are the key operations that can be performed on a circularly linked list:

- ✓ insert_front(value): Create a new node with the given value and insert it at the front of the list (i.e., make it the new head node).
- ✓ insert_back(value): Create a new node with the given value and insert it at the back of the list (i.e., make it the new tail node).
- ✓ insert_after(node, value): Create a new node with the given value and insert it after the given node.
- ✓ delete_front(): Remove and return the head node of the list.

- ✓ delete_back(): Remove and return the tail node of the list.
- ✓ delete_after(node): Remove the node that comes after the given node.
- ✓ search(value): Traverse the list and return the first node that contains the given value (or None if the value is not found).
- ✓ size(): Traverse the list and count the number of nodes in it.
- ✓ is_empty(): Return True if the list is empty, False otherwise.

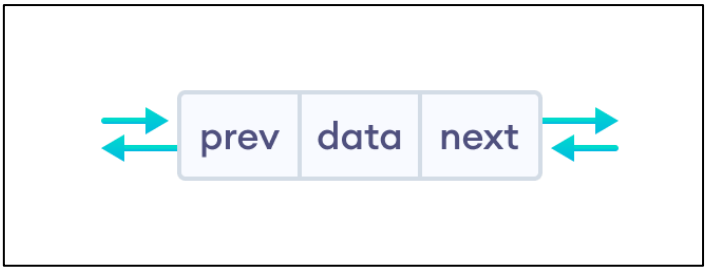


In summary, a circularly linked list is a variation of a singly linked list that provides constant-time access to the last node and forms a loop or circle. It can be used in situations where we need to repeatedly traverse the list from both ends or where we want to represent a circular data structure such as a queue or a ring buffer.

6.DLL:

A doubly linked list is a type of linked list in which each node consists of 3 components:

- `*prev` - address of the previous node
- `data` - data item
- `*next` - address of next node



Representation of Doubly Linked List

Let's see how we can represent a doubly linked list on an algorithm/code. Suppose we have a doubly linked list:

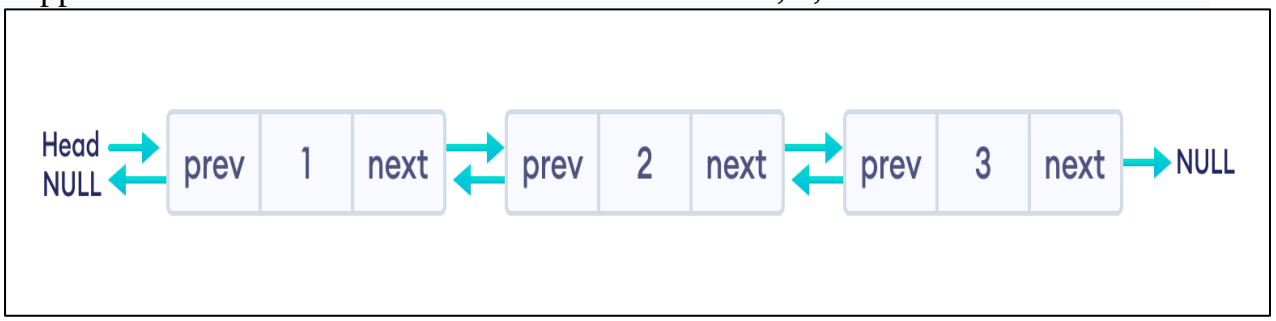
Insertion on a Doubly Linked List

Pushing a node to a doubly-linked list is similar to pushing a node to a linked list, but extra work is required to handle the pointer to the previous node.

We can insert elements at 3 different positions of a doubly-linked list:

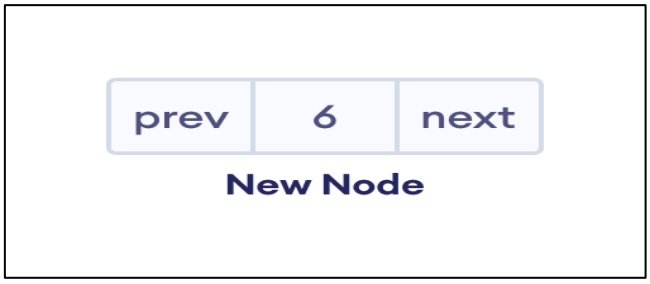
1. Insertion at the beginning
2. Insertion in-between nodes
3. Insertion at the End

Suppose we have a double-linked list with elements **1**, **2**, and **3**.

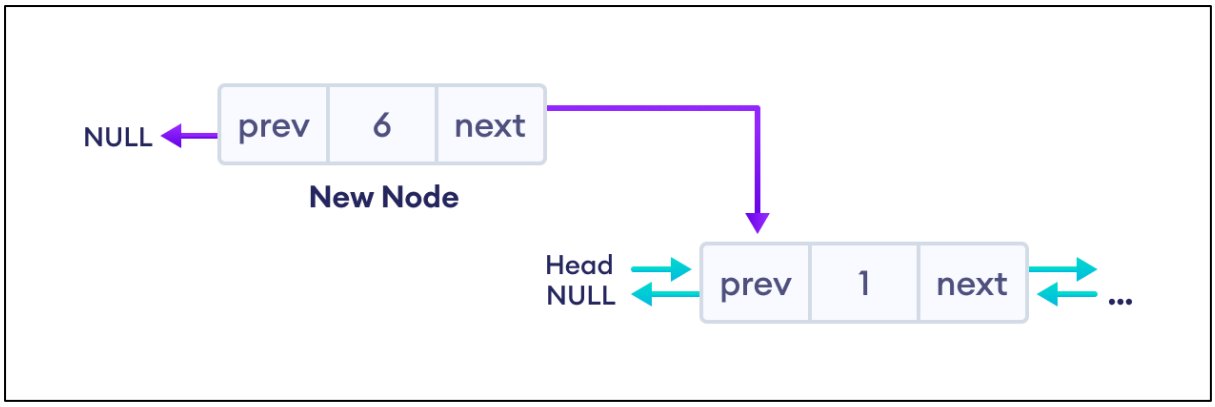


1. Create a new node

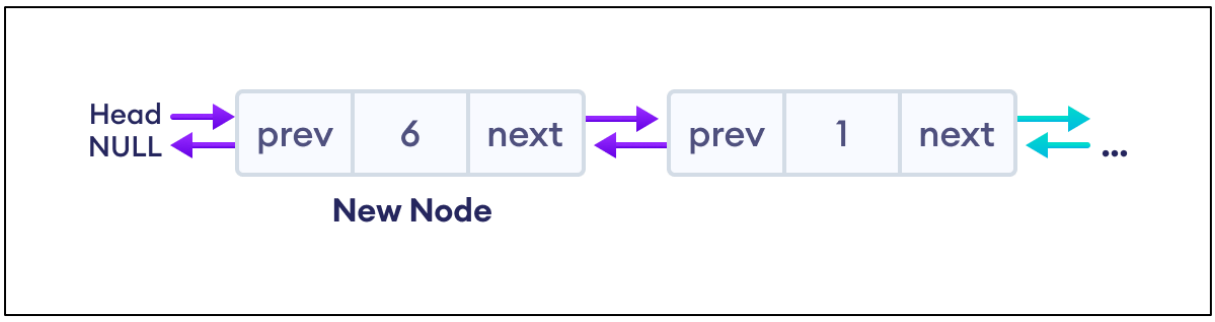
1. Insertion at the Beginning



2. Set prev and next pointers of new node



3. Make new node as head node



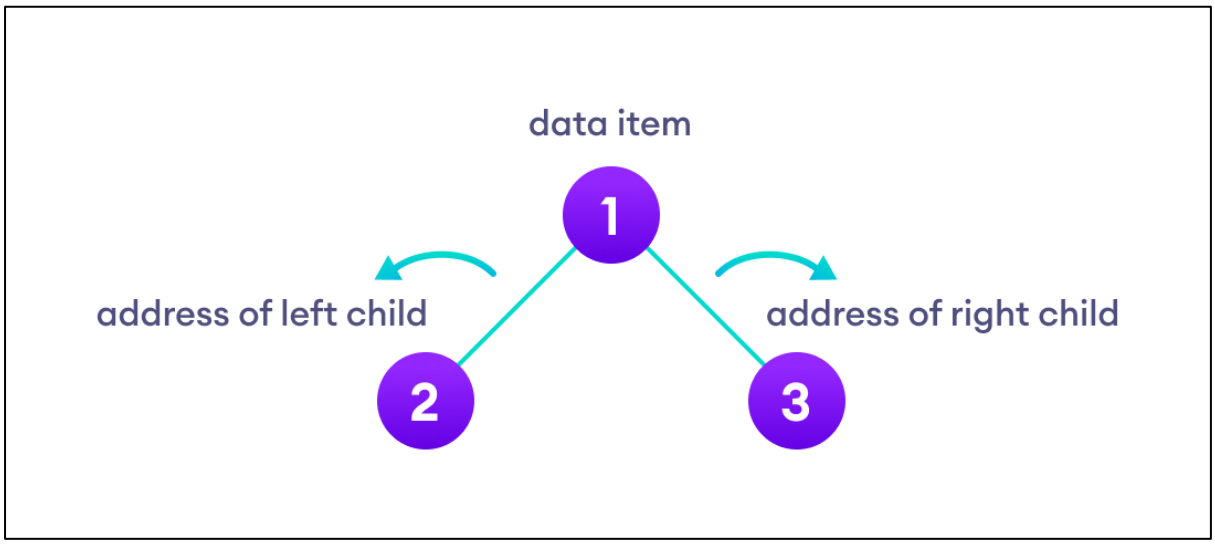
REFERE NOTE AND EXAM PAPER

6. BINARY TREE:

In this tutorial, you will learn about binary tree and its different types. Also, you will find working examples of binary tree in C, C++, Java and Python.

A binary tree is a tree data structure in which each parent node can have at most two children. Each node of a binary tree consists of three items:

- ✓ data item
- ✓ address of left child
- ✓ address of right child

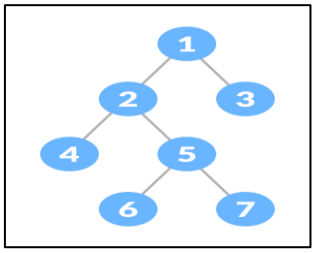


Binary Tree

Types of Binary Tree

1. Full Binary Tree

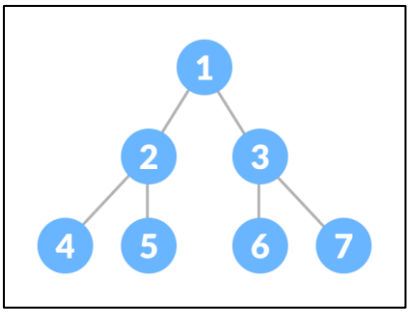
A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.



Full Binary Tree

2. Perfect Binary Tree

A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.

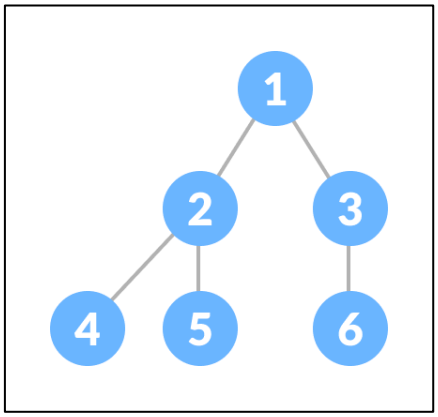


Perfect Binary Tree

3. Complete Binary Tree

A complete binary tree is just like a full binary tree, but with two major differences

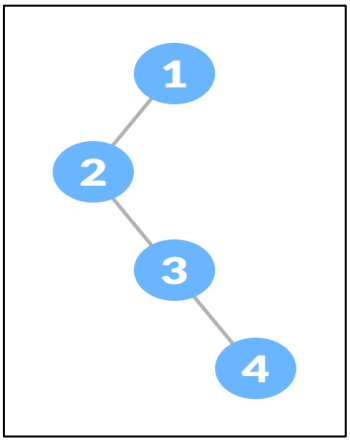
- ✓ Every level must be completely filled
- ✓ All the leaf elements must lean towards the left.
- ✓ The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.



Complete Binary Tree

4. Degenerate or Pathological Tree

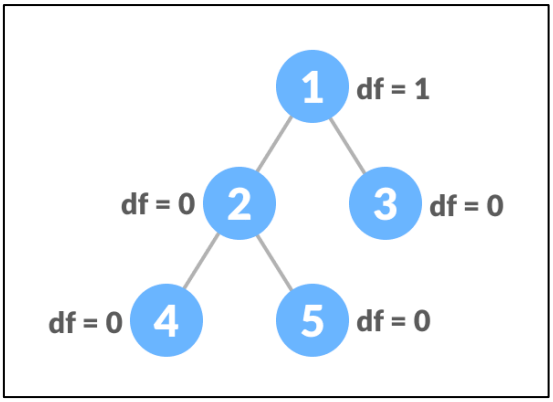
A degenerate or pathological tree is the tree having a single child either left or right.



Degenerate Binary Tree

5. Balanced Binary Tree

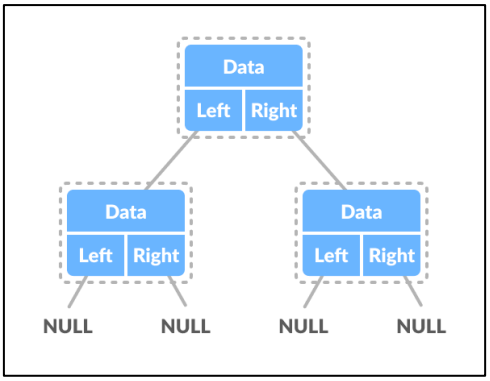
It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1.



Balanced Binary Tree

Binary Tree Representation

A node of a binary tree is represented by a structure containing a data part and two pointers to other structures of the same type.



Binary Tree Representation

Python, Examples

```
# Binary Tree in Python
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

    # Traverse preorder
    def traversePreOrder(self):
        print(self.val, end=' ')
        if self.left:
            self.left.traversePreOrder()
        if self.right:
            self.right.traversePreOrder()

    # Traverse inorder
    def traverseInOrder(self):
        if self.left:
```

```
            self.left.traverseInOrder()
        print(self.val, end=' ')
        if self.right:
            self.right.traverseInOrder()

    # Traverse postorder
    def traversePostOrder(self):
        if self.left:
            self.left.traversePostOrder()
        if self.right:
            self.right.traversePostOrder()
        print(self.val, end=' ')


root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
print("Pre order Traversal: ", end="")
root.traversePreOrder()
print("\nIn order Traversal: ", end="")
root.traverseInOrder()
print("\nPost order Traversal: ", end="")
root.traversePostOrder()
```

Binary Tree Applications

- ✓ For easy and quick access to data
- ✓ In router algorithms
- ✓ To implement heap data structure
- ✓ Syntax tree

7.TREE TRAVERSAL ALGORITHMS:

There are three main traversal algorithms:

Pre-order traversal: In this algorithm, the root node is visited first, followed by the left subtree, and then the right subtree. This is a recursive algorithm, where the base case is when there is no more node to visit.

Here's an example implementation in Python:

```
def pre_order(node):
    if node is None:
        return
    print(node.value)
    pre_order(node.left)
    pre_order(node.right)
```

In-order traversal: In this algorithm, the left subtree is visited first, followed by the root node, and then the right subtree. This is also a recursive algorithm.

Here's an example implementation in Python:

```
def in_order(node):
    if node is None:
        return
```

```
    in_order(node.left)
    print(node.value)
    in_order(node.right)
```

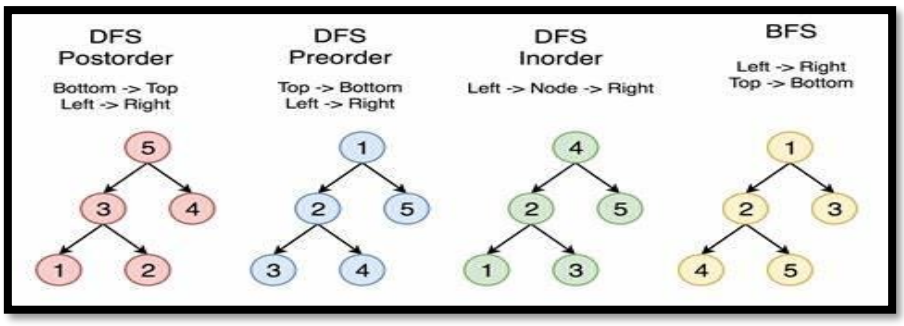
Post-order traversal: In this algorithm, the left subtree is visited first, followed by the right subtree, and then the root node. This is also a recursive algorithm.

Here's an example implementation in Python:

```
def post_order(node):
    if node is None:
        return
    post_order(node.left)
    post_order(node.right)
    print(node.value)
```

In addition to these basic traversal algorithms, there are also variants of these algorithms, such as level-order traversal, which visits each level of the tree from top to bottom before moving on to the next level. Level-order traversal is usually implemented using a queue data structure.

It's worth noting that the choice of traversal algorithm can have an impact on the efficiency of certain operations performed on the tree. For example, in-order traversal of a binary search tree will produce the nodes in sorted order, which can be useful for certain applications.



UNIT-I V

1.Priority Queues:

A priority queue is a **special type of queue** in which each element is associated with a **priority value**. And, elements are served on the basis of their priority. That is, higher priority elements are served first.

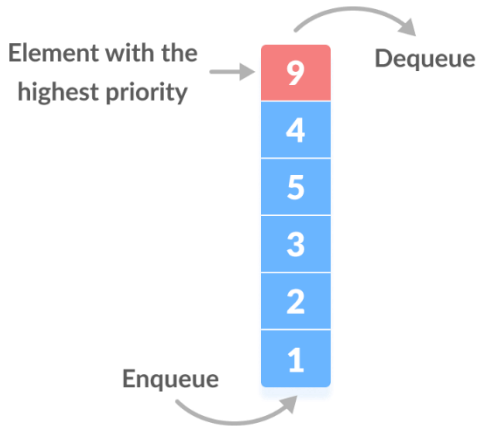
However, if elements with the same priority occur, they are served according to their order in the queue.

Assigning Priority Value

Generally, the value of the element itself is considered for assigning the priority. For example,

The element with the highest value is considered the highest priority element. However, in other cases, we can assume the element with the lowest value as the highest priority element.

We can also set priorities according to our needs.



Removing Highest Priority Element

Difference between Priority Queue and Normal Queue

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

Implementation of Priority Queue

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

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

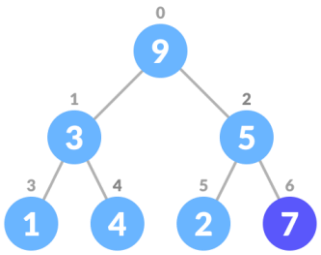
Operations	peek	insert	delete
Linked List	O(1)	O(n)	O(1)
Binary Heap	O(1)	O(log n)	O(log n)
Binary Search Tree	O(1)	O(log n)	O(log n)

Priority Queue Operations

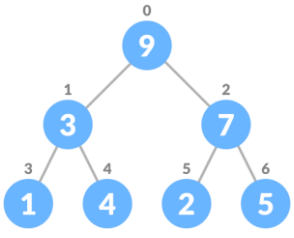
Basic operations of a priority queue are inserting, removing, and peeking elements.

1. Inserting an Element into the Priority Queue

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



Insert the new element at the end of the tree. Insert an element at the end of the queue

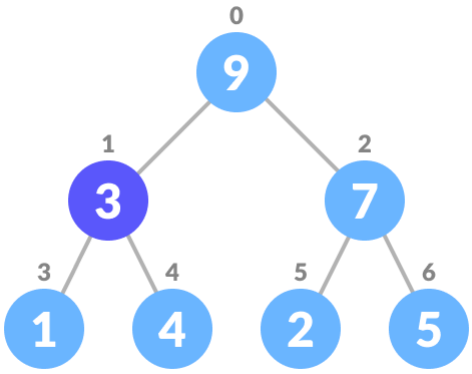


Heapify after insertion

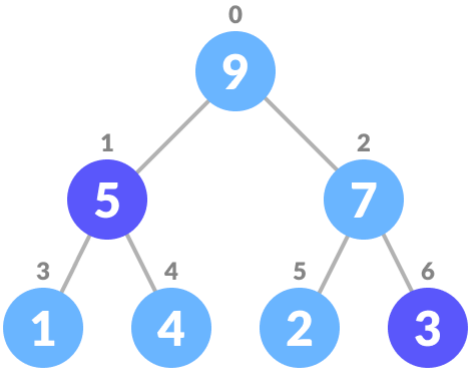
2. Deleting an Element from the Priority Queue

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

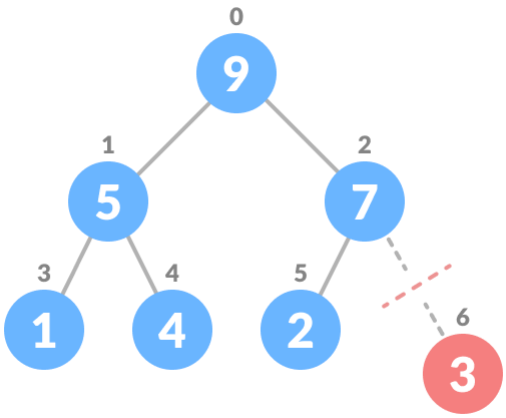
Select the element to be deleted.



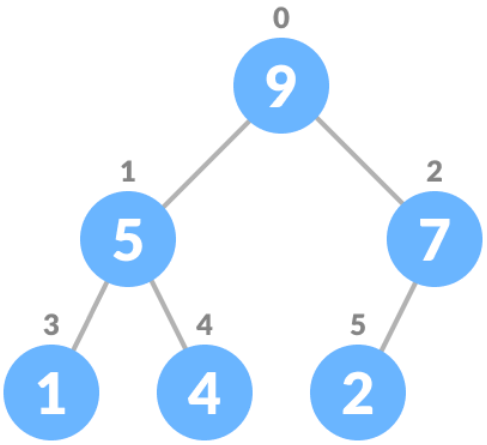
Swap it with the last element.



Remove the last element.



Heapify the tree.



Peeking from the Priority Queue (Find max/min)

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

For both Max heap and Min Heap

```
return rootNode
```

Priority Queue Implementations in Python

```
import heapq
class PriorityQueue:
    def __init__(self):
        self._heap = []
        self._index = 0

    def push(self, item, priority):
        heapq.heappush(self._heap, (-priority, self._index, item))
        self._index += 1

    def pop(self):
        return heapq.heappop(self._heap)[-1]
```

```
    def is_empty(self):
        return len(self._heap) == 0
```

In this implementation, we use a list _heap to store the items in the priority queue. Each item is represented as a tuple containing the priority (negated so that the highest priority comes first), an index to ensure stable sorting, and the actual item. We use the heappush and heappop functions from the heapq module to insert and remove items from the heap. The is_empty method simply checks if the _heap list is empty.

Here's an example usage of this priority queue:

```
pq = PriorityQueue()
pq.push('task1', 3)
pq.push('task2', 1)
pq.push('task3', 2)

while not pq.is_empty():
    print(pq.pop())  # prints 'task2', 'task3', 'task1'
```

In this example, we create a PriorityQueue instance pq and add three tasks with different priorities using the push method. We then use a loop to pop items from the priority queue using the pop method and print them in order of decreasing priority.

Priority Queue Applications

Dijkstra's algorithm

for implementing stack

for load balancing and interrupt handling in an operating system

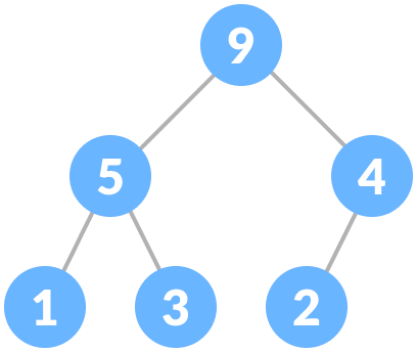
for data compression in Huffman code

2.HEAP

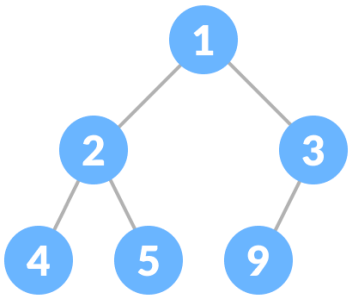
Heap data structure is a complete binary tree that satisfies the heap property, where any given node is

always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called max heap property.

always smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called min heap property.



Max-heap



Min-heap

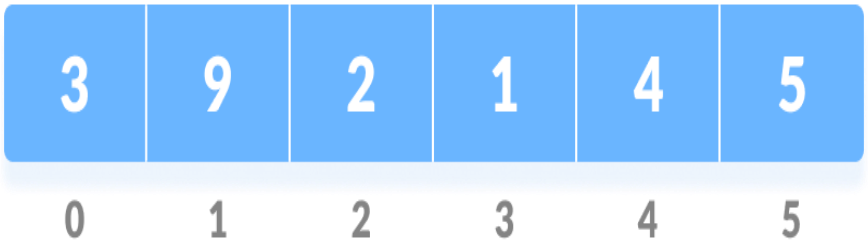
This type of data structure is also called a **binary heap**.

Heap Operations

Some of the important operations performed on a heap are described below along with their algorithms.

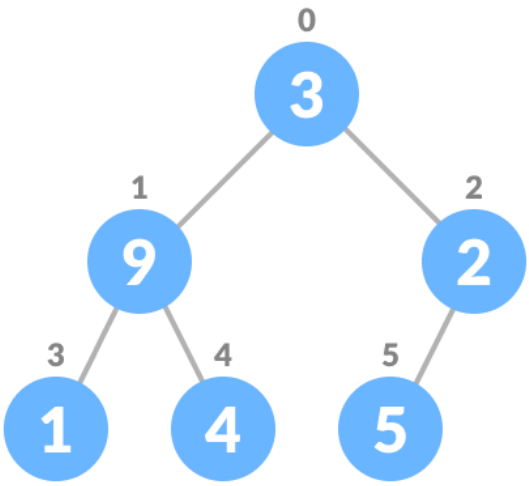
Heapify is the process of creating a heap data structure from a binary tree. It is used to create a Min-Heap or a Max-Heap.

Let the input array be



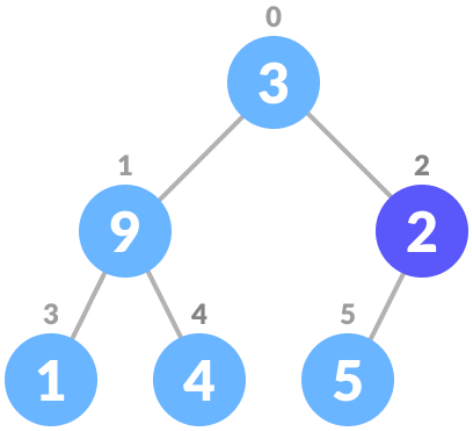
Initial Array

1.Create a complete binary tree from the array



Complete binary tree

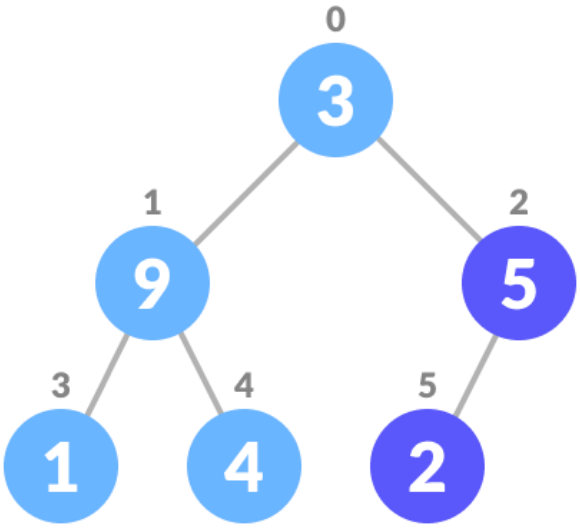
2.Start from the first index of non-leaf node whose index is given by n/2 - 1.



Start from the first on leaf node

3.Set current element i as largest.

4.The index of left child is given by 2i + 1 and the right child is given by 2i + 2.
5.If leftChild is greater than currentElement (i.e. element at ith index), set leftChildIndex as largest.
6.If rightChild is greater than element in largest, set rightChildIndex as largest.



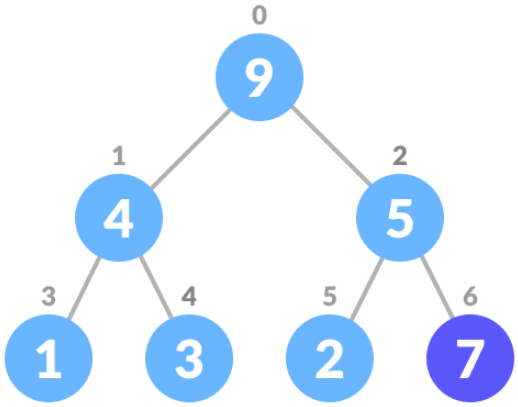
7.Swap largest with currentElement

Swap if necessary

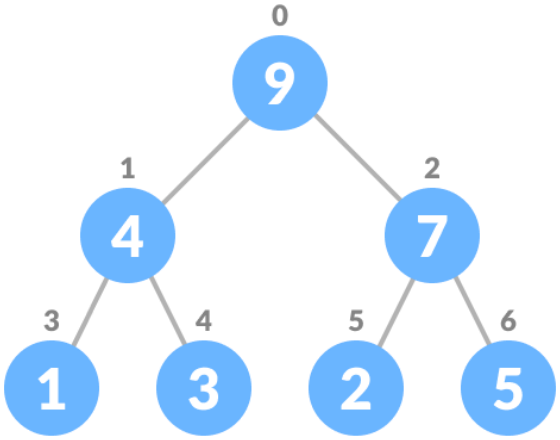
Repeat steps 3-7 until the subtrees are also heapified

- Insert Element into Heap

Insert the new element at the end of the tree. Insert at the end



Heapify the tree.

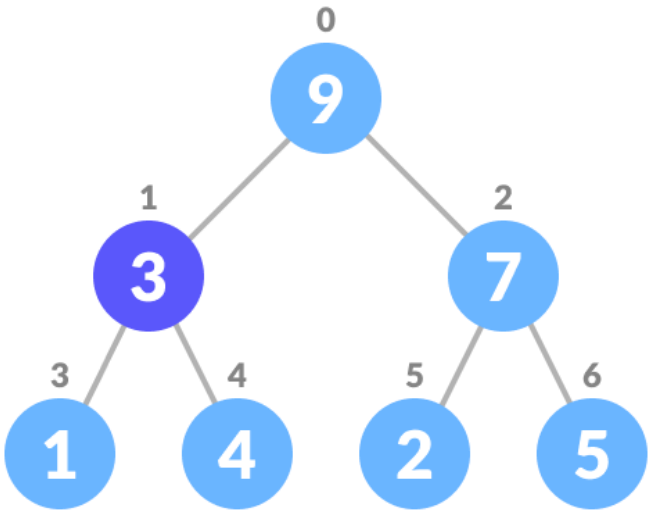


Heapify the array

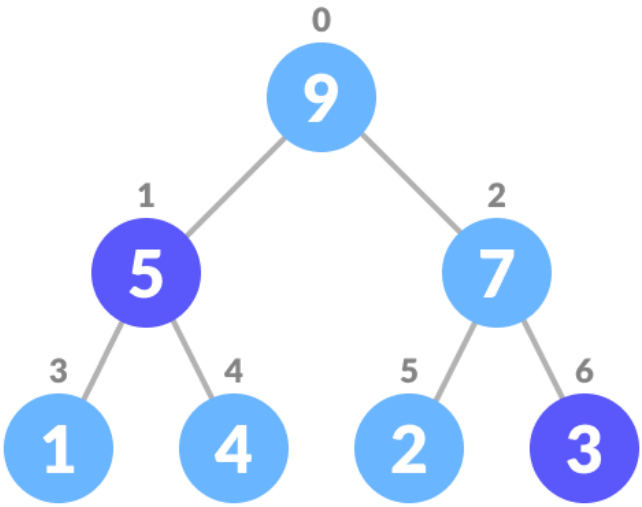
For Min Heap, the above algorithm is modified so that `parentNode` is always smaller than `newNode`.

➢ Delete Element from Heap

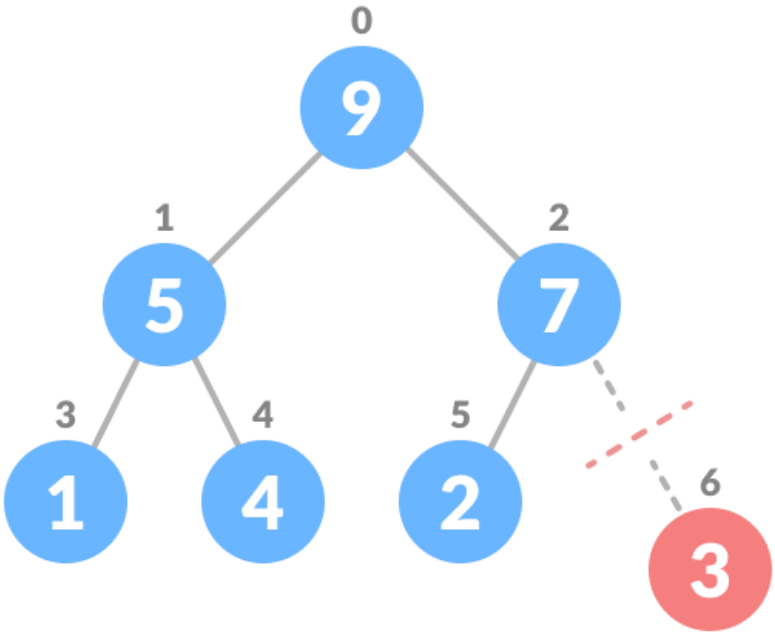
Select the element to be deleted.



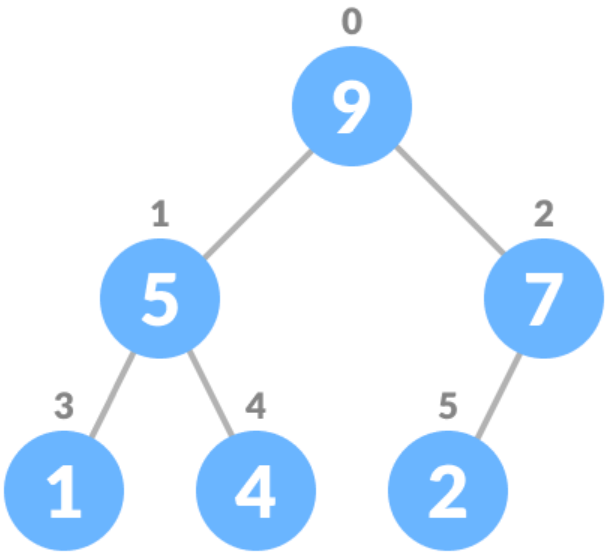
Swap it with the last element.



➢ Remove the last element.



Heapify the tree.



Heapify the array

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

Peek (Find max/min)

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

For both Max heap and Min Heap

return rootNode

Heap Data Structure Applications

Heap is used while implementing a priority queue.

Dijkstra's Algorithm

Heap Sort

3.HAS TABLE

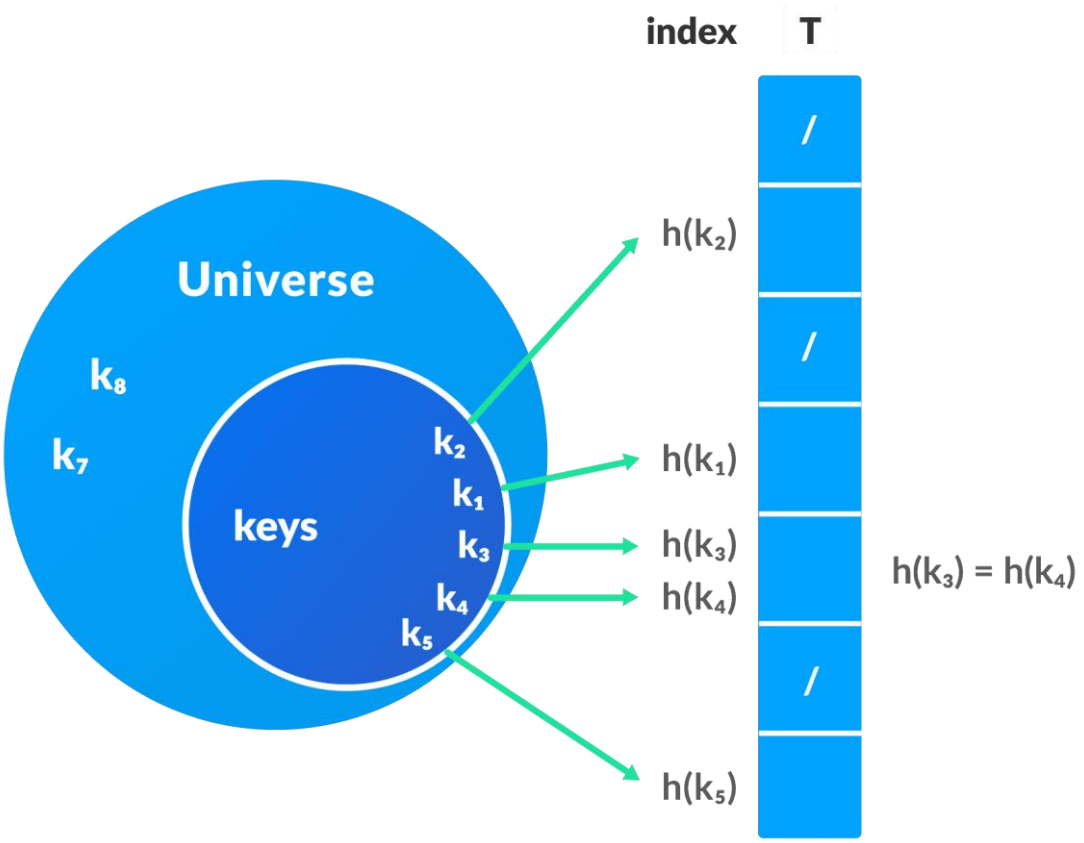
The Hash table data structure stores elements in key-value pairs where

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

Key and Value in Hash table

Hashing (Hash Function)

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**.
Let k be a key and h(x) be a hash function.
Here, h(k) will give us a new index to store the element linked with k.



Hash table Representation

Hash Collision

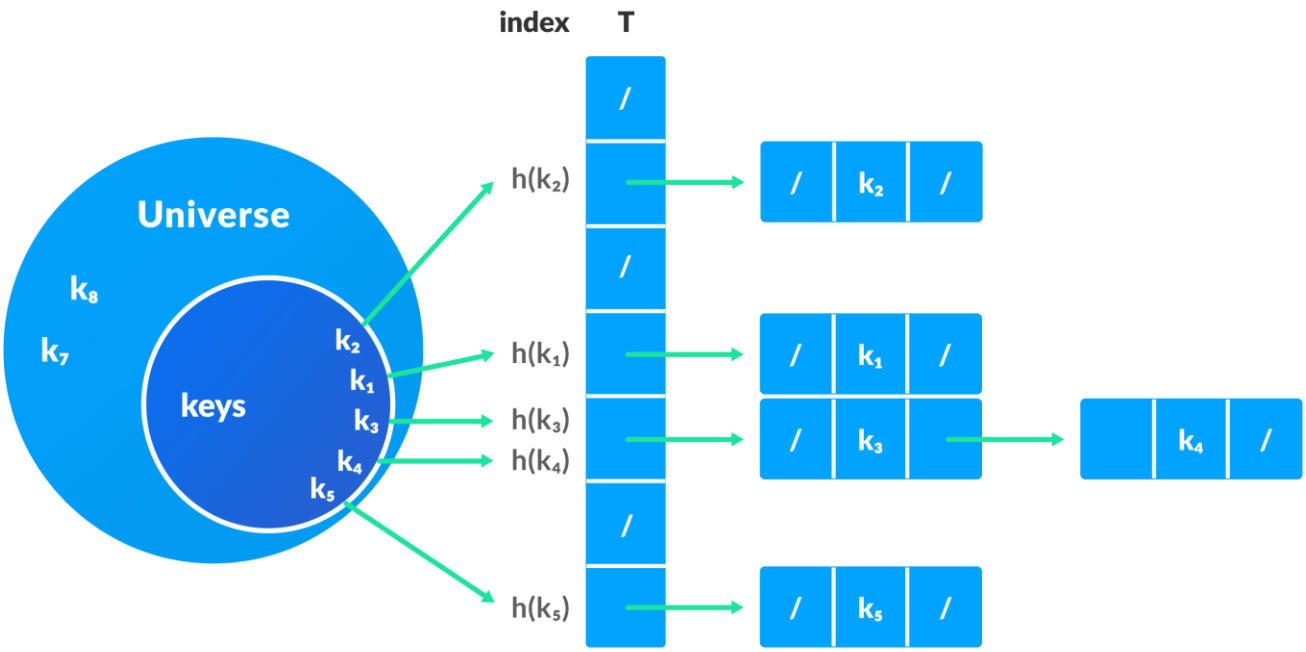
When the hash function generates the same index for multiple keys, there will be a conflict (what value to be stored in that index). This is called a **hash collision.** We can resolve the hash collision using one of the following techniques.

- Collision resolution by chaining
- Open Addressing: Linear/Quadratic Probing and Double Hashing

1. Collision resolution by chaining

In chaining, if a hash function produces the same index for multiple elements, these elements are stored in the same index by using a doubly-linked list.

If `j` is the slot for multiple elements, it contains a pointer to the head of the list of elements. If no element is present, `j` contains `NIL`.



Collision Resolution using chaining

2. Open Addressing

Unlike chaining, open addressing doesn't store multiple elements into the same slot. Here, each slot is either filled with a single key or left NIL.
Different techniques used in open addressing are:

i. Linear Probing

In linear probing, collision is resolved by checking the next slot.The problem with linear probing is that a cluster of adjacent slots is filled. When inserting a new element, the entire cluster must be traversed. This adds to the time required to perform operations on the hash table.

ii. Quadratic Probing

It works similar to linear probing but the spacing between the slots is increased (greater than one)

iii. Double hashing

If a collision occurs after applying a hash function h(k), then another hash function is calculated for finding the next slot.

Good Hash Functions

A good hash function may not prevent the collisions completely however it can reduce the number of collisions.

Here, we will look into different methods to find a good hash function

1. Division Method

2. Multiplication Method

3. Universal Hashing

Applications of Hash Table

Hash tables are implemented where

- constant time lookup and insertion is required
- cryptographic applications
- indexing data is required

EXAMPLE OF HASTABLE

```
hashTable = {}
def insertData(key, data):
    hashTable[key] = data
def removeData(key):
    del hashTable[key]
insertData(123, "apple")
insertData(432, "mango")
insertData(213, "banana")
insertData(654, "guava")
print(hashTable)
removeData(123)
print(hashTable)
```

OUTPUT:
{123: 'apple', 432: 'mango', 213: 'banana', 654: 'guava'}
{432: 'mango', 213: 'banana', 654: 'guava'}

In this program, we first create an empty dictionary hashTable. Then, we add four key-value pairs to the dictionary using the insertData function. After that, we print the contents of the dictionary using the print statement, which displays all the key-value pairs in the dictionary. Next, we call the removeData function to delete the key 123 from the dictionary. Finally, we print the contents of the dictionary again using the print statement, which shows that the key 123 has been successfully removed from the dictionary.

4.SKIP LIST

A skip list is a probabilistic data structure. The skip list is used to store a sorted list of elements or data with a linked list. It allows the process of the elements or data to view efficiently. In one single step, it skips several elements of the entire list, which is why it is known as a skip list.

The skip list is an extended version of the linked list. It allows the user to search, remove, and insert the element very quickly. It consists of a base list that includes a set of elements which maintains the link hierarchy of the subsequent elements.

Skip list structure

It is built in two layers: The lowest layer and Top layer.

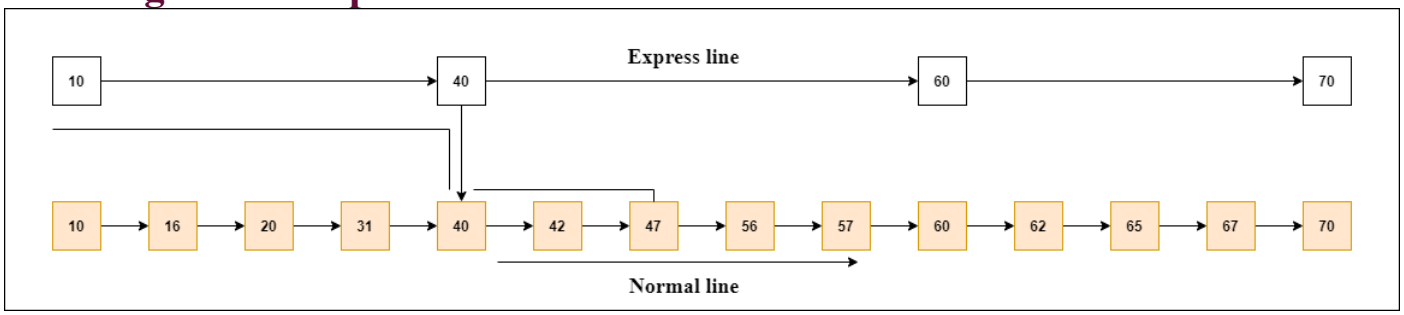
The lowest layer of the skip list is a common sorted linked list, and the top layers of the skip list are like an "express line" where the elements are skipped.

Complexity table of the Skip list

S. No	Complexity	Average case	Worst case
1.	Access complexity	O(logn)	O(n)
2.	Search complexity	O(logn)	O(n)
3.	Delete complexity	O(logn)	O(n)
4.	Insert complexity	O(logn)	O(n)

5.	Space complexity	-	O(nlogn)

Working of the Skip list



Skip List Basic Operations

There are the following types of operations in the skip list.

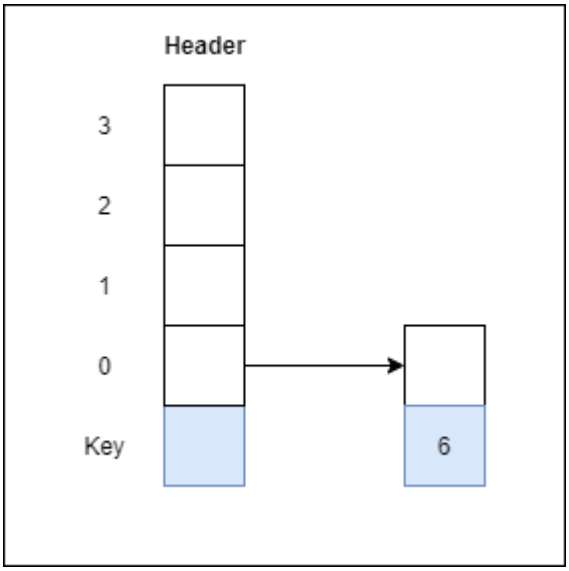
- **Insertion operation:** It is used to add a new node to a particular location in a specific situation.
- **Deletion operation:** It is used to delete a node in a specific situation.
- **Search Operation:** The search operation is used to search a particular node in a skip list.

Example 1: Create a skip list, we want to insert these following keys in the empty skip list.

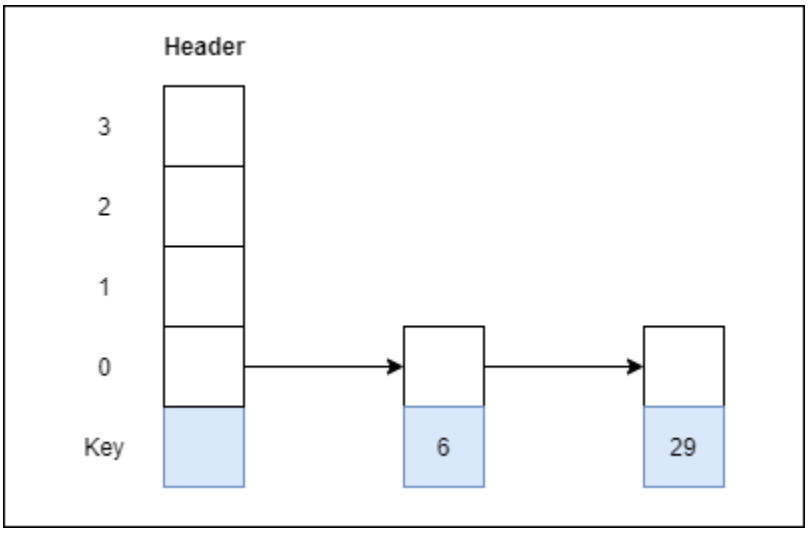
1. 6 with level 1.
2. 29 with level 1.
3. 22 with level 4.
4. 9 with level 3.
5. 17 with level 1.
6. 4 with level 2.

Ans:

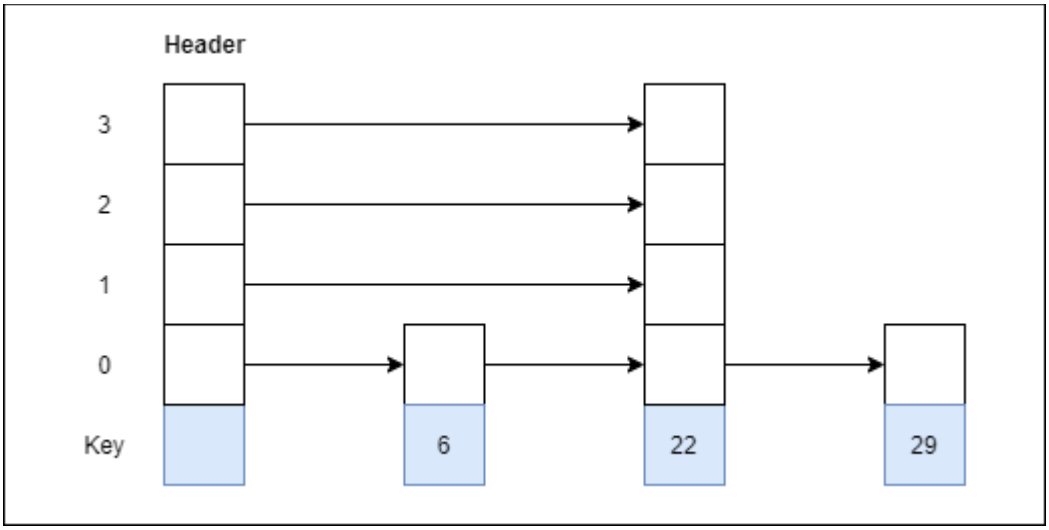
Step 1: Insert 6 with level 1



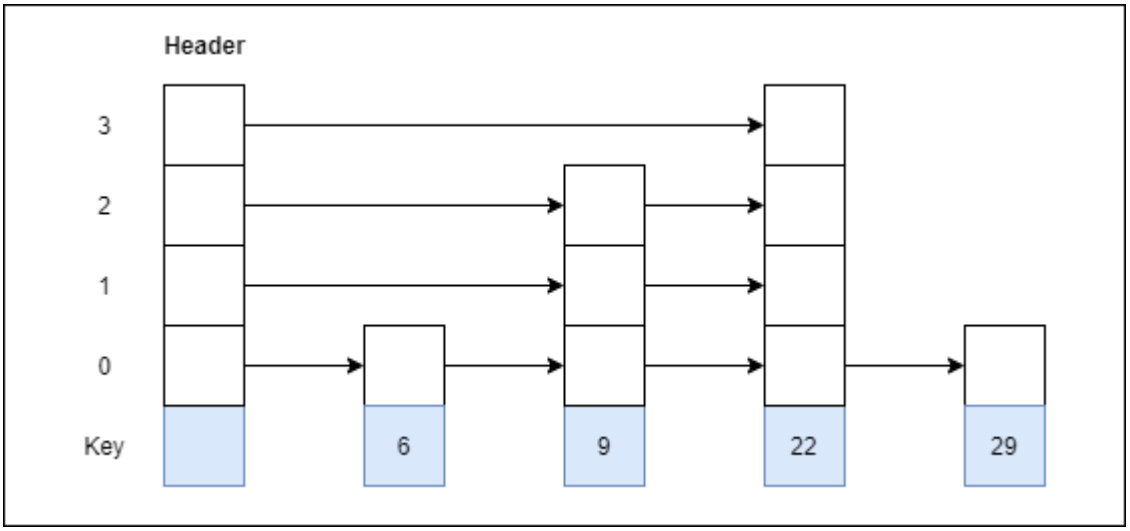
Step 2: Insert 29 with level 1



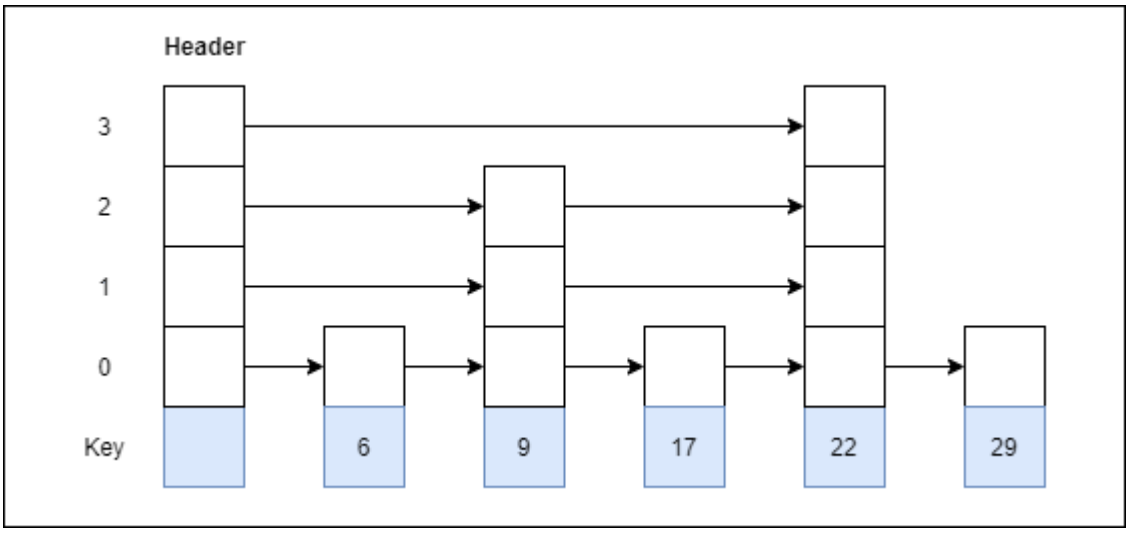
Step 3: Insert 22 with level 4



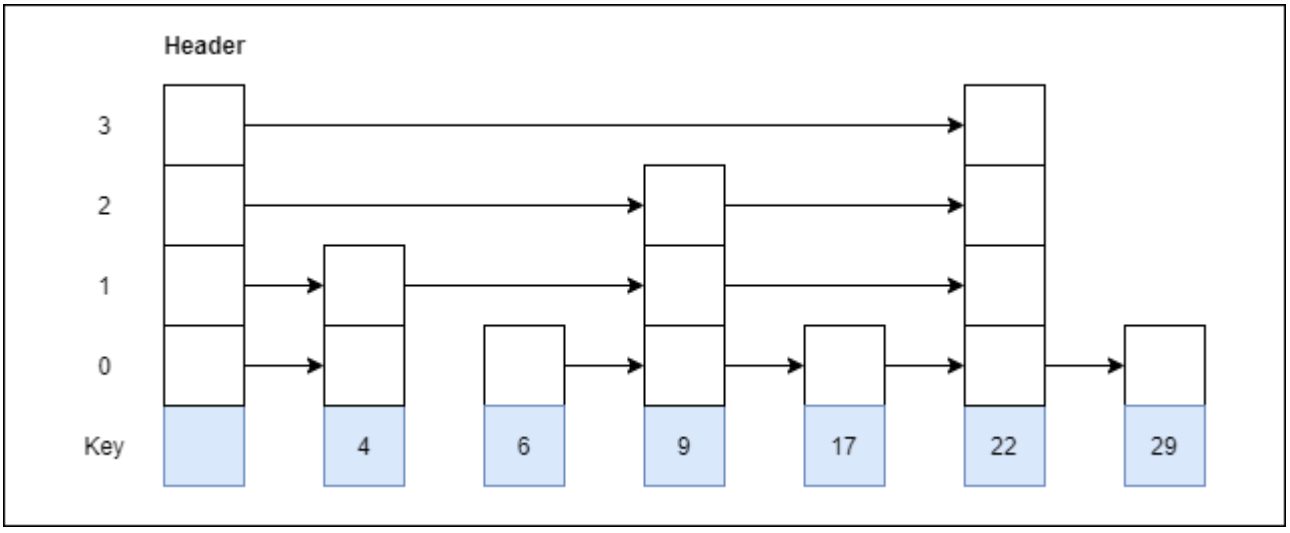
Step 4: Insert 9 with level 3



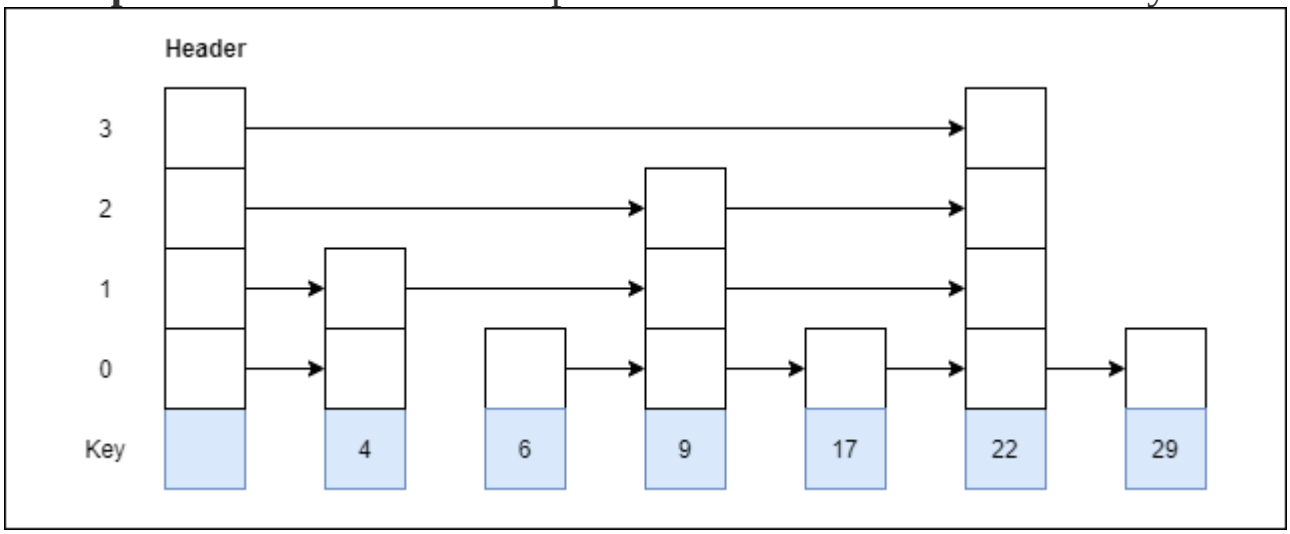
Step 5: Insert 17 with level 1



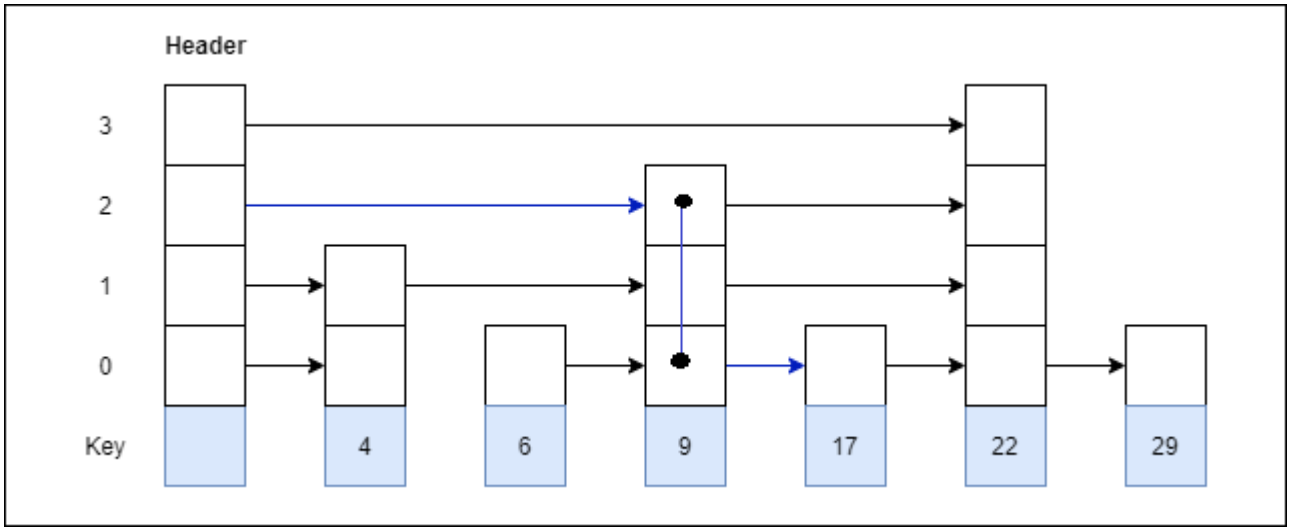
Step 6: Insert 4 with level 2



- **Example 2:** Consider this example where we want to search for key 17.



- **Ans:**



Advantages of the Skip list

1. If you want to insert a new node in the skip list, then it will insert the node very fast because there are no rotations in the skip list.
2. The skip list is simple to implement as compared to the hash table and the binary search tree.
3. It is very simple to find a node in the list because it stores the nodes in sorted form.
4. The skip list algorithm can be modified very easily in a more specific structure, such as indexable skip lists, trees, or priority queues.
5. The skip list is a robust and reliable list.

Disadvantages of the Skip list

1. It requires more memory than the balanced tree.

2. Reverse searching is not allowed.
3. The skip list searches the node much slower than the linked list.

Applications of the Skip list

- Database indexing
- Cache implementation
- Web server performance optimization
- Priority queues

UNIT-5

1. AVL TREE:

AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.

AVL tree got its name after its inventor Georgy Adelson-Velsky and Landis.

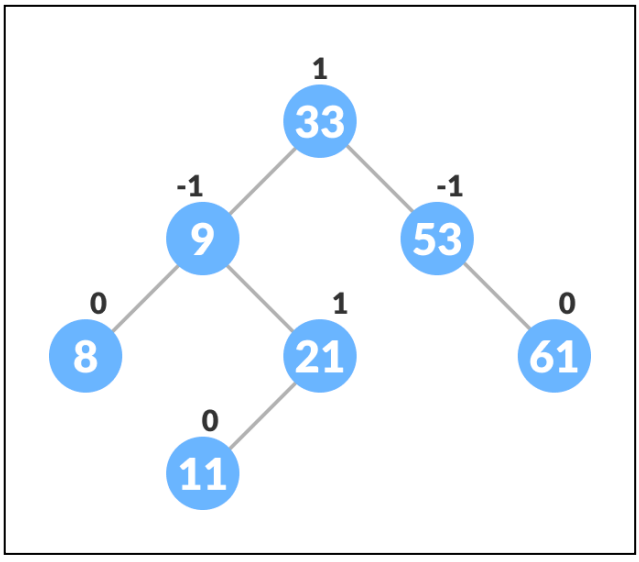
Balance Factor

Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.

Balance Factor = (Height of Left Subtree - Height of Right Subtree) or (Height of Right Subtree - Height of Left Subtree)

The self-balancing property of an avl tree is maintained by the balance factor. The value of balance factor should always be -1, 0 or +1.

An example of a balanced avl tree is:

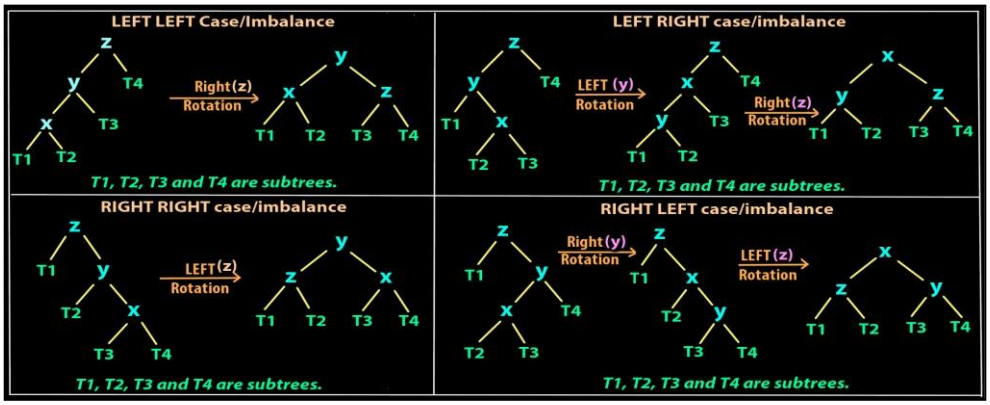


Avl tree

AVL trees support the following operations:

- ✓ Search: Given a value, the search operation traverses the tree to find the node with the matching value. The time complexity of this operation is O(log n) in the worst case.
- ✓ Insertion: Given a value, the insertion operation adds a new node with the value to the tree while maintaining the AVL tree's balancing property. The time complexity of this operation is O(log n) in the worst case.
- ✓ Deletion: Given a value, the deletion operation removes the node with the matching value from the tree while maintaining the AVL tree's balancing property. The time complexity of this operation is O(log n) in the worst case.
- ✓ Traversal: There are three types of traversal algorithms for AVL trees: in-order, pre-order, and post-order. These algorithms visit each node in the tree in a specific order. The time complexity of these algorithms is O(n), where n is the number of nodes in the tree.
- ✓ Balancing: AVL trees require balancing operations to maintain their balancing property. The four types of rotations mentioned earlier are used to balance the tree when a node violates the balancing property. The time complexity of these operations is O(1).

AVL trees are useful in applications that require fast search, insertion, and deletion operations with a high degree of reliability and predictability. They are particularly useful in databases, compilers, and other software systems that rely heavily on efficient data structures.



AVL Tree Pros:

1. Fast search, insertion, and deletion with O(log n) worst-case time complexity.
2. Maintains balancing property for logarithmic tree height and faster access than non-balanced binary search trees.
3. Useful for reliable and predictable applications like databases and compilers.
4. Well-defined rules for balancing, making them easier to implement and maintain compared to other self-balancing trees.

AVL Tree Cons:

1. Balancing operations can be expensive and have a worst-case time complexity of O(log n).
2. Requires additional memory to store the balance factor for each node, increasing overall memory usage.
3. Insertions and deletions can be slower due to rebalancing operations to maintain balancing property.
4. Can be more complex to implement and maintain compared to non-balanced trees due to balancing requirements.

APPLICATIONS:

AVL trees are commonly used for fast and reliable search, insertion, and deletion operations on large amounts of data. Specifically, they are used in databases, compilers, text editors, network routing protocols, and file systems. The self-balancing property of AVL trees ensures efficient operations and makes them useful in any application requiring these operations.

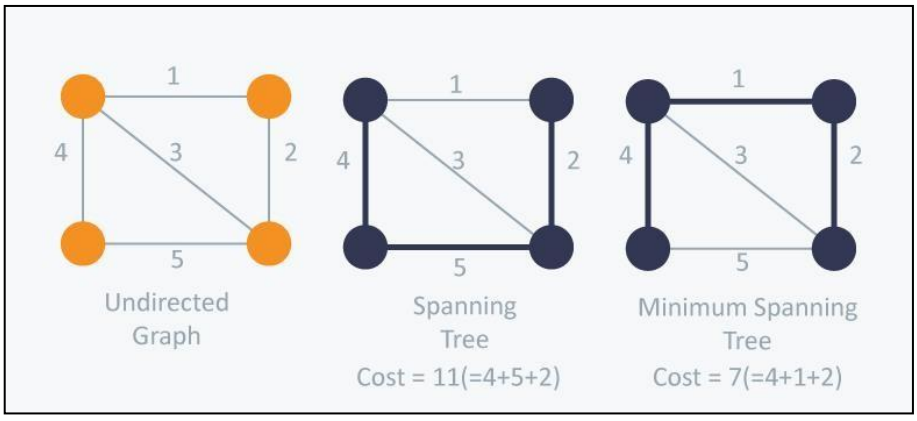
2.MINIMUM SPANNING TREE:

A minimum spanning tree (MST) is a subset of the edges of a weighted, connected, and undirected graph that connects all the vertices with the minimum possible total edge weight. The MST is a fundamental problem in graph theory with various applications in real-world scenarios such as network design, transportation planning, and image segmentation.

There are several algorithms to find the MST of a graph, including:

- ✓ Kruskal's algorithm: This algorithm starts with an empty set of edges and iteratively adds the minimum weight edge that does not create a cycle until all vertices are connected.
- ✓ Prim's algorithm: This algorithm starts with a single vertex and iteratively adds the minimum weight edge that connects a vertex in the tree to a vertex outside the tree until all vertices are connected.
- ✓ Boruvka's algorithm: This algorithm starts with a forest of single-vertex trees and iteratively adds the minimum weight edge that connects a tree to another tree until all vertices are connected.

Finding the MST has several practical applications, such as minimizing the cost of connecting a network of cities with roads or finding the optimal layout of a power grid.



EX FOR MST:

```python
class Djset:
    def __init__(self,size):
        self.parent=list(range(size))

    def find(self,i):
        if self.parent[i]!=i:
            self.parent[i]=self.find(self.parent[i])
        return self.parent[i]

    def union(self,i,j):
        self.parent[self.find(i)]=self.find(j)


class kMST:
    def __init__(self,edges,n_v):
        self.edges=edges
        self.n_v=n_v

    def find_mst(self):
        res=[]
        dj_set=Djset(self.n_v)
        for u,v,weight in sorted(self.edges,key=lambda e:e[2]):
            if dj_set.find(u)!=dj_set.find(v):
                res.append((u,v,weight))
                dj_set.union(u,v)
        return res

if __name__=="__main__":
    edges=[(0, 1, 8), (0, 2, 5), (1, 2, 9), (1, 3, 11), (2, 3, 15), (2, 4, 10), (3, 4, 7)]
    kur=kMST(edges,n_v=5)
    mst=kur.find_mst()
    for u,v,weight in mst:
        print(f"Edges : {u} - {v} weight :{weight}")

```

APPLICATION:

Minimum Spanning Trees (MSTs) are used in various fields, including network design, transportation planning, circuit design, image segmentation, and clustering. MSTs can help minimize the total cost, length, or distance traveled while connecting a set of nodes. For example, MSTs are used to design communication networks, electronic circuits, and transportation networks. In image processing and data mining, MSTs are used to segment images and cluster data points based on their proximity in the tree. Overall, MSTs are a versatile tool for solving optimization problems involving connected nodes.

Applications of MST

- Telephone wiring (use as little wire as possible)
- Electronic circuit board design
- Cancer imaging (MSTs describe arrangements of nuclei in skin cells)
- Biomedical image analysis (detect actin fibers in cell images)
- Astronomy (find clusters of quasars and Seyfert galaxies)
- Finding road networks in satellite and aerial imagery

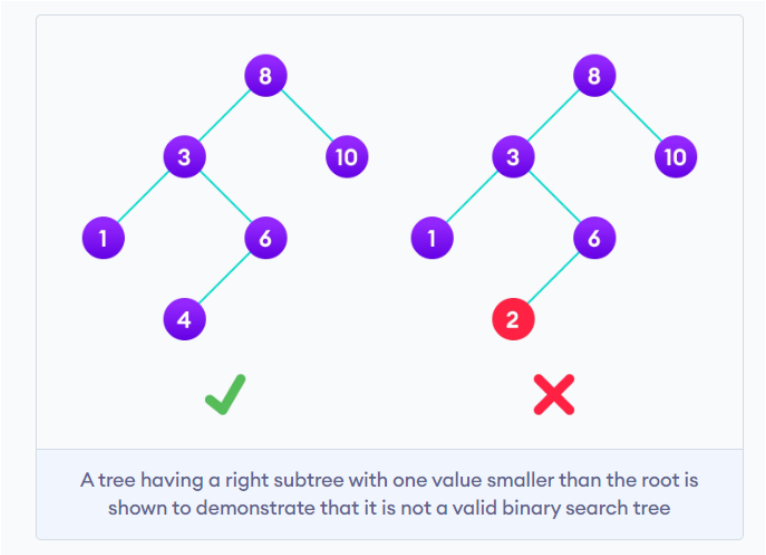
3. BINARY SEARCH TREES

Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.

- It is called a binary tree because each tree node has a maximum of two children.
- It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.

The properties that separate a binary search tree from a regular binary tree is

1. All nodes of left subtree are less than the root node
2. All nodes of right subtree are more than the root node
3. Both subtrees of each node are also BSTs i.e. they have the above two properties



A tree having a right subtree with one value smaller than the root is shown to demonstrate that it is not a valid binary search tree

The binary tree on the right isn't a binary search tree because the right subtree of the node "3" contains a value smaller than it.

There are two basic operations that you can perform on a binary search tree:

Search Operation

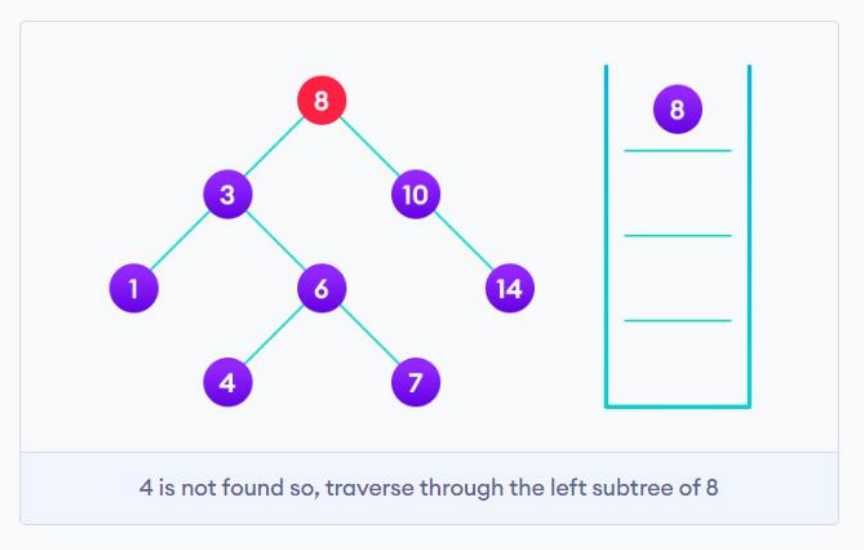
The algorithm depends on the property of BST that if each left subtree has values below root and each right subtree has values above the root.

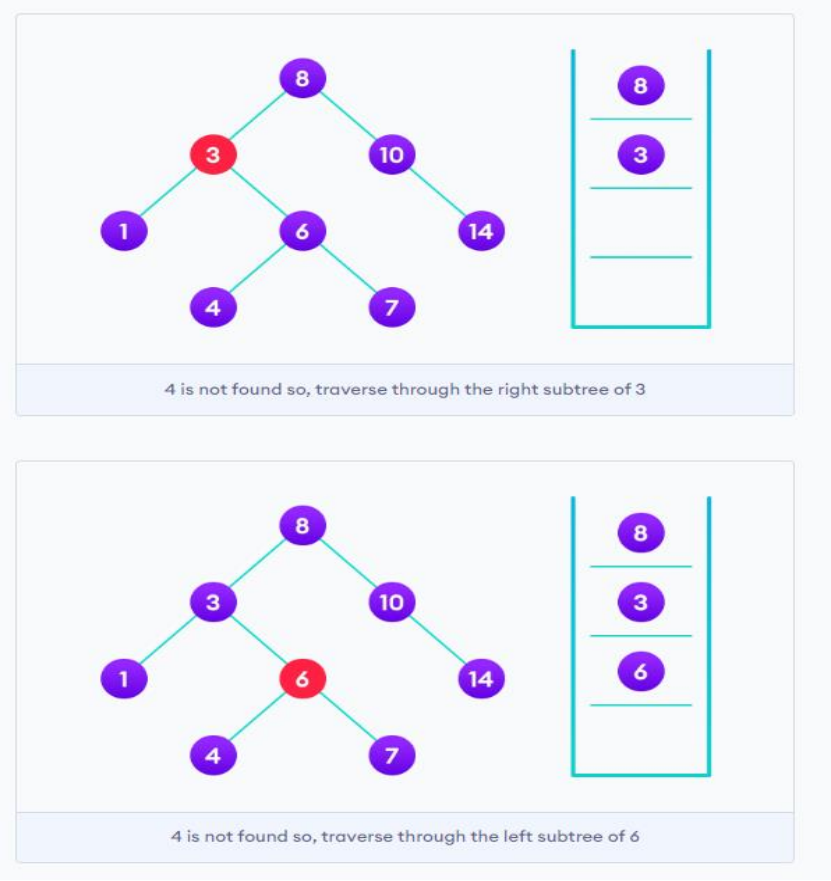
If the value is below the root, we can say for sure that the value is not in the right subtree; we need to only search in the left subtree and if the value is above the root, we can say for sure that the value is not in the left subtree; we need to only search in the right subtree.

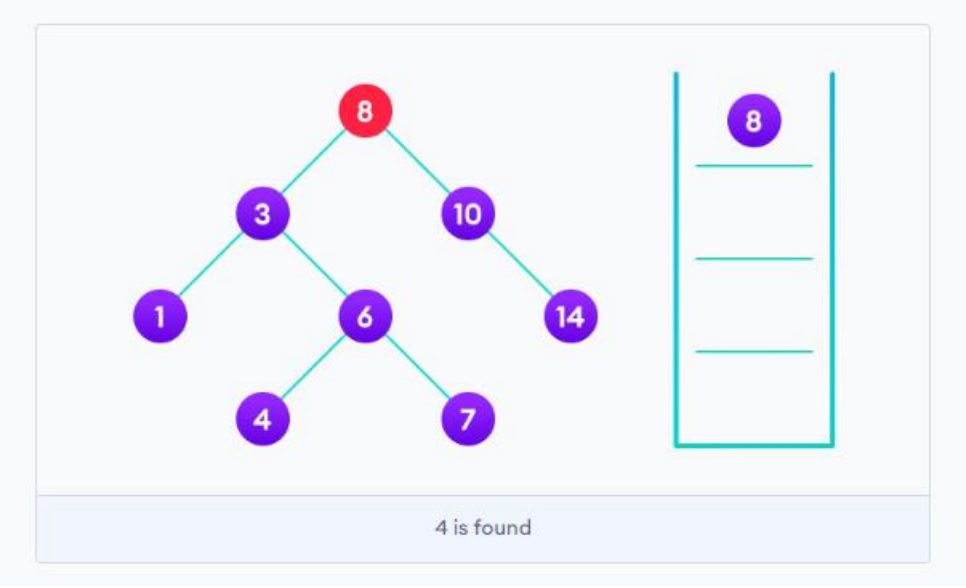
Algorithm:

```
If root == NULL
   return NULL;
If number == root->data
   return root->data;
If number < root->data
   return search(root->left)
If number > root->data
   return search(root->right)
```

Let us try to visualize this with a diagram

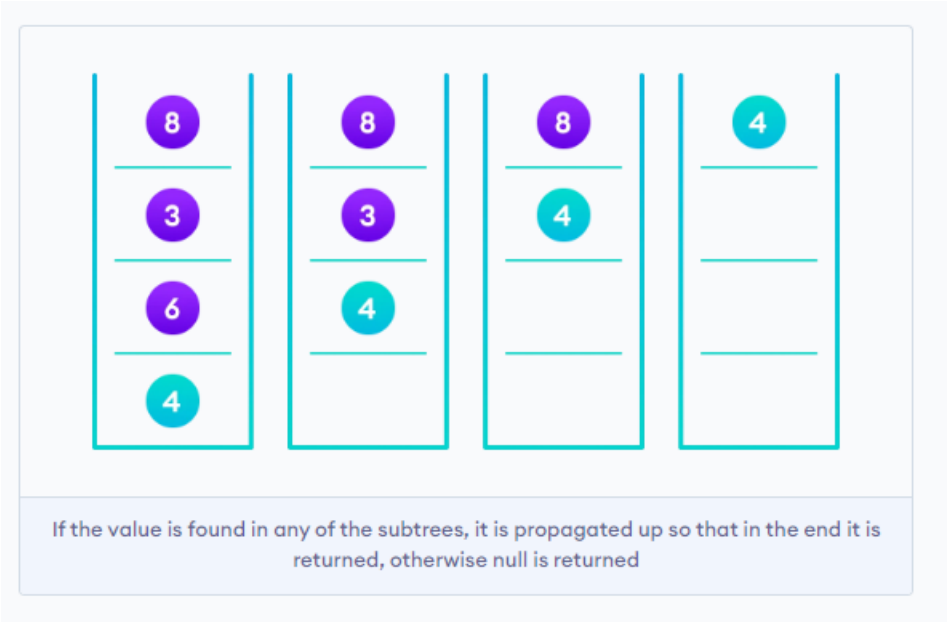






If the value is found, we return the value so that it gets propagated in each recursion step as shown in the image below.

If you might have noticed, we have called return search (struct node*) four times. When we return either the new node or NULL, the value gets returned again and again until search(root) returns the final result.

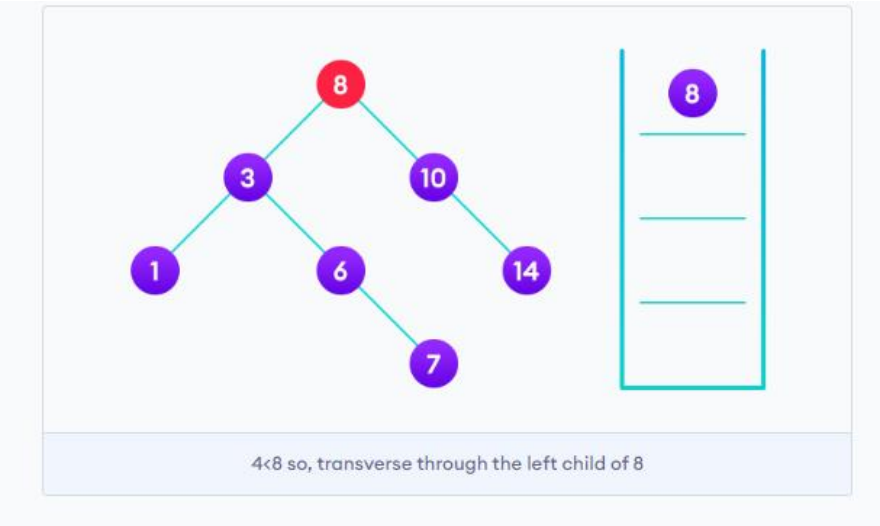


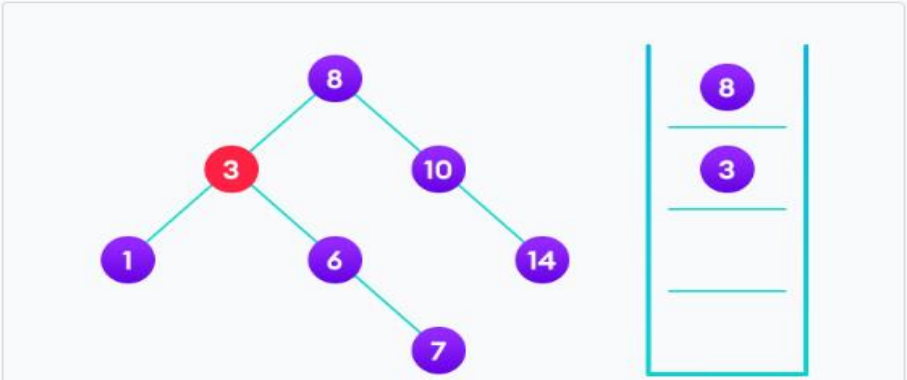
If the value is not found, we eventually reach the left or right child of a leaf node which is NULL and it gets propagated and returned.

Insert Operation

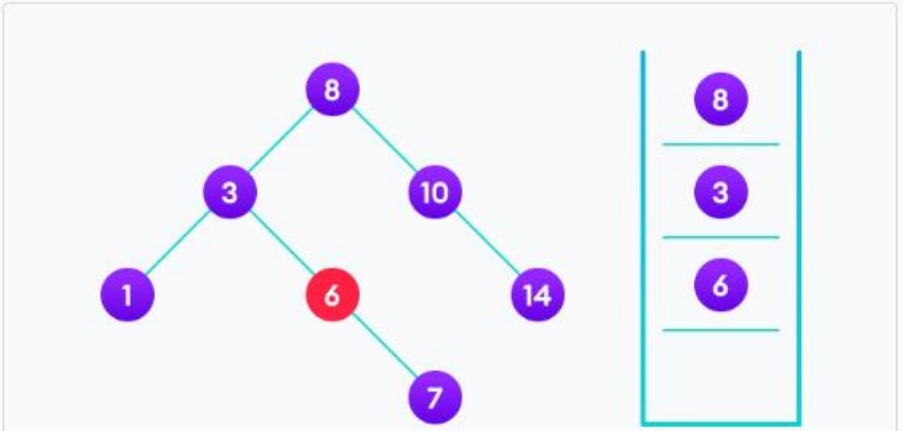
Inserting a value in the correct position is similar to searching because we try to maintain the rule that the left subtree is lesser than root and the right subtree is larger than root.

We keep going to either right subtree or left subtree depending on the value and when we reach a point left or right subtree is null, we put the new node there.

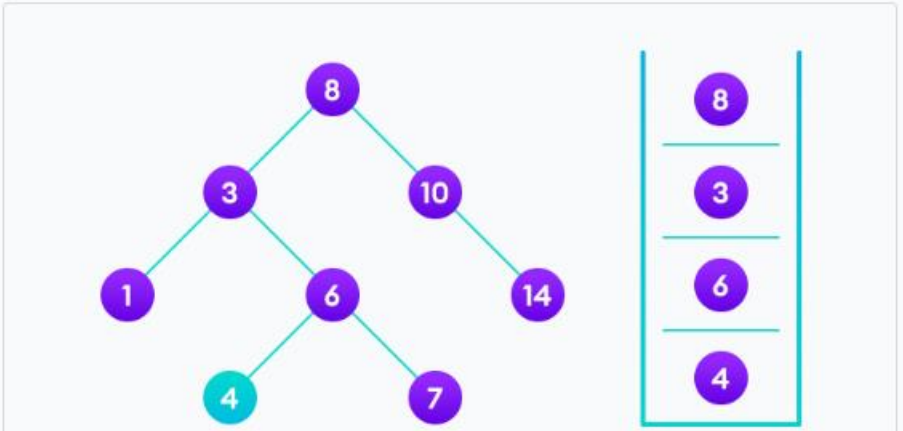




4>3 so, transverse through the right child of 8



4<6 so, transverse through the left child of 6



Insert 4 as a left child of 6

We have attached the node but we still have to exit from the function without doing any damage to the rest of the tree. This is where the `return node;` at the end comes in handy. In the case of `NULL`, the newly created node is returned and attached to the parent node, otherwise the same node is returned without any change as we go up until we return to the root.

This makes sure that as we move back up the tree, the other node connections aren't changed.

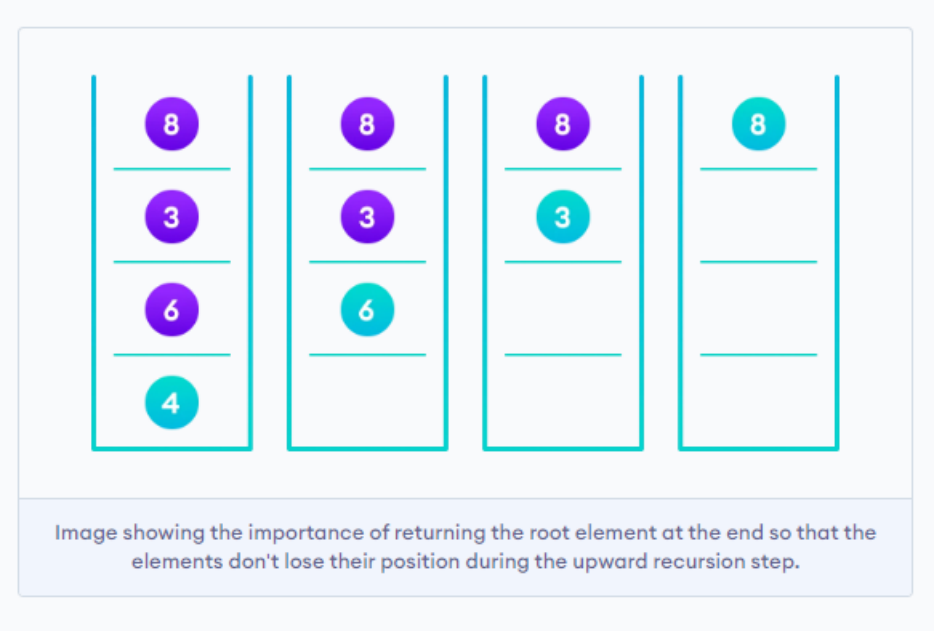


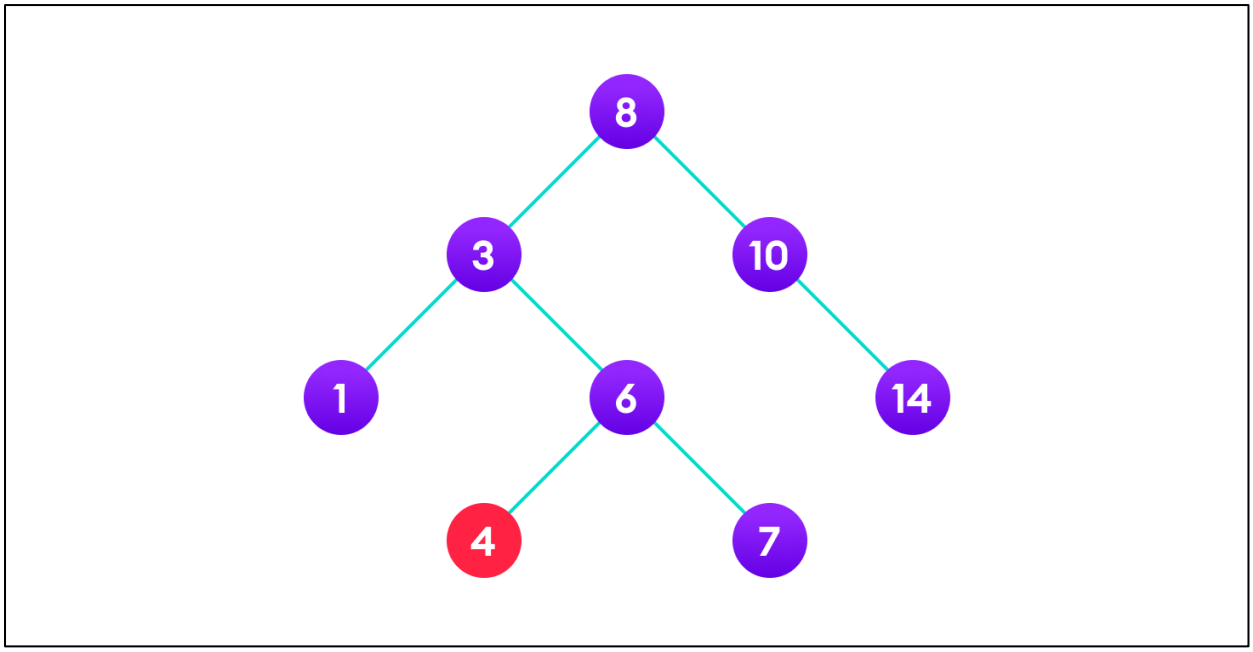
Image showing the importance of returning the root element at the end so that the elements don't lose their position during the upward recursion step.

Deletion Operation

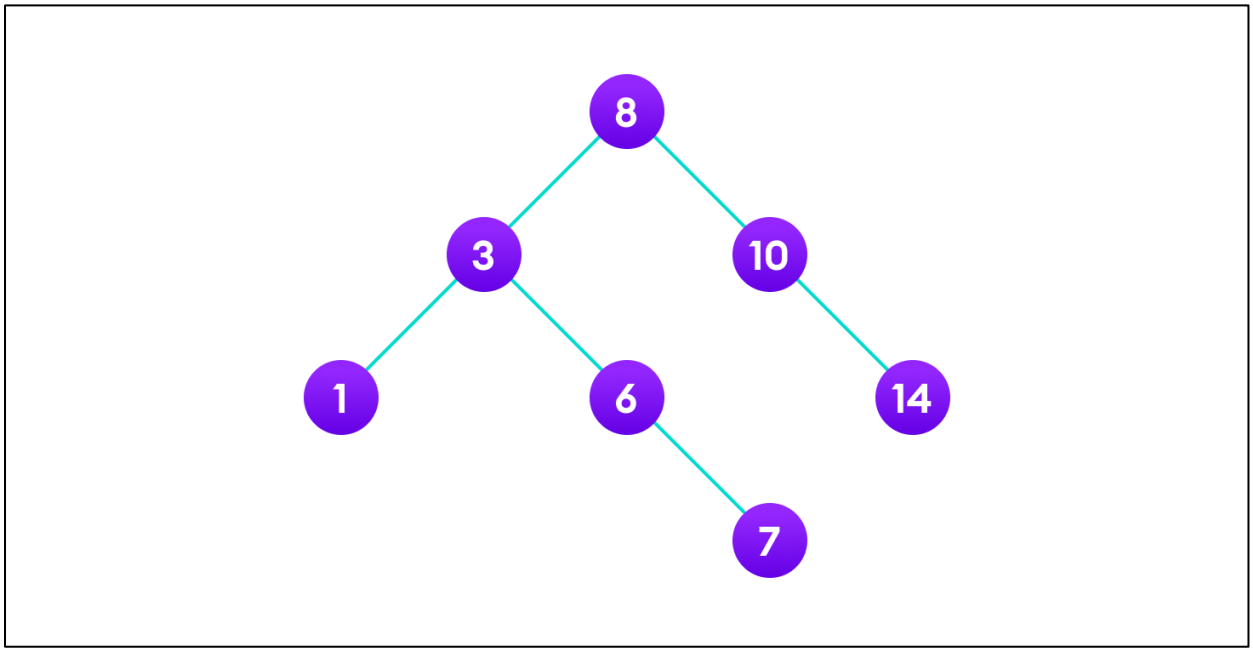
There are three cases for deleting a node from a binary search tree.

Case I

In the first case, the node to be deleted is the leaf node. In such a case, simply delete the node from the tree.



4 is to be deleted

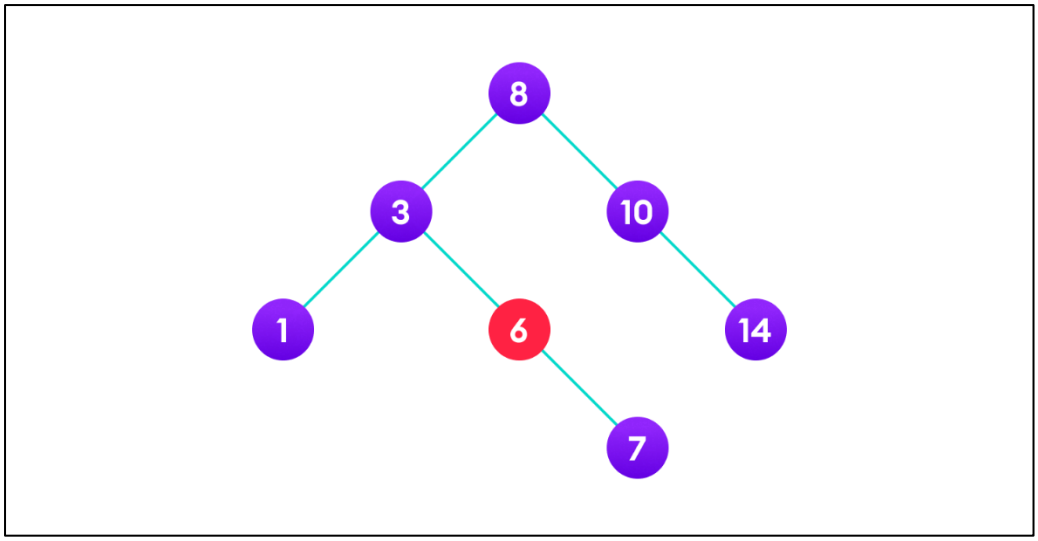


Delete the node

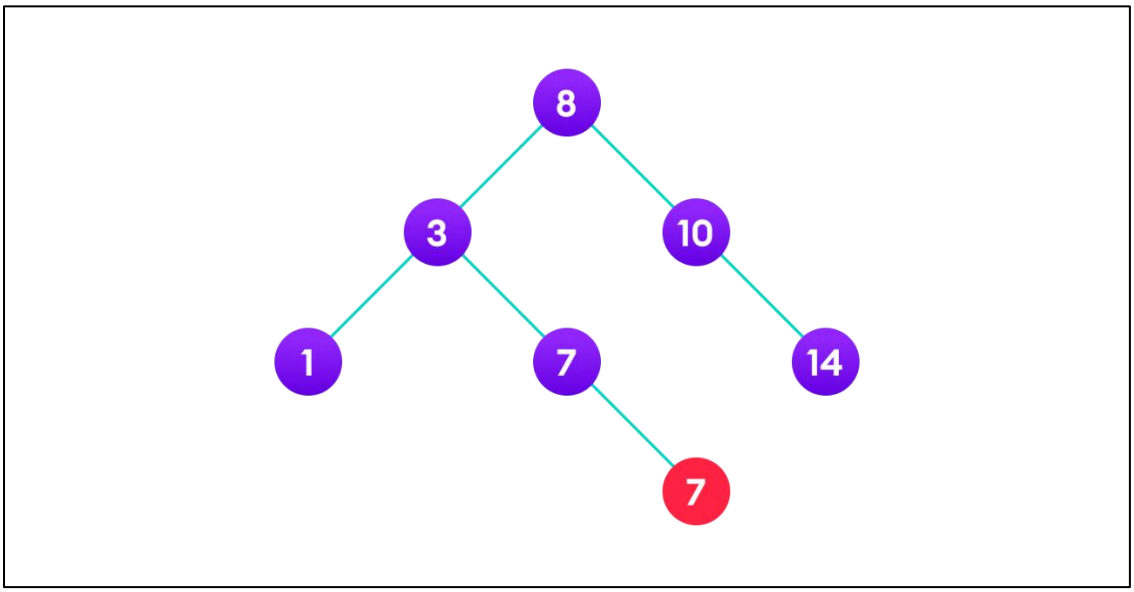
Case II

In the second case, the node to be deleted lies has a single child node. In such a case follow the steps below:

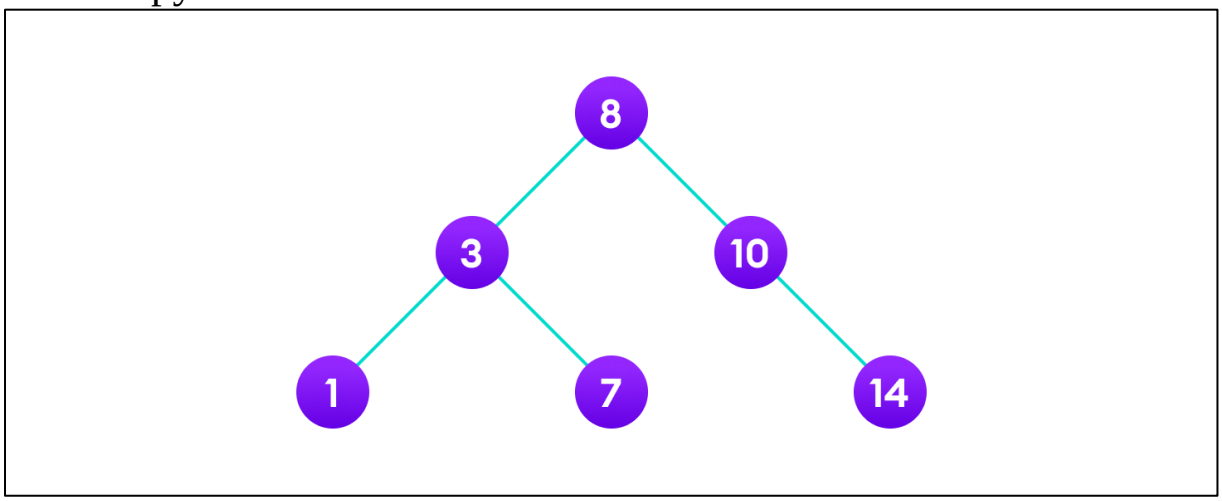
- Replace that node with its child node.
- Remove the child node from its original position.



6 is to be deleted



copy the value of its child to the node and delete the child

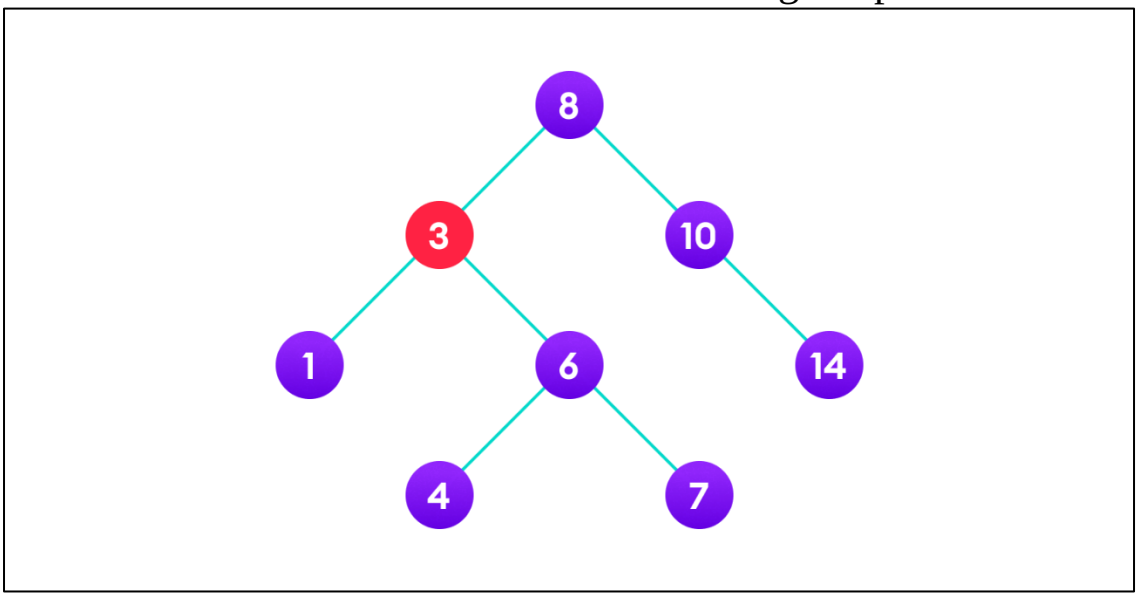


Final tree

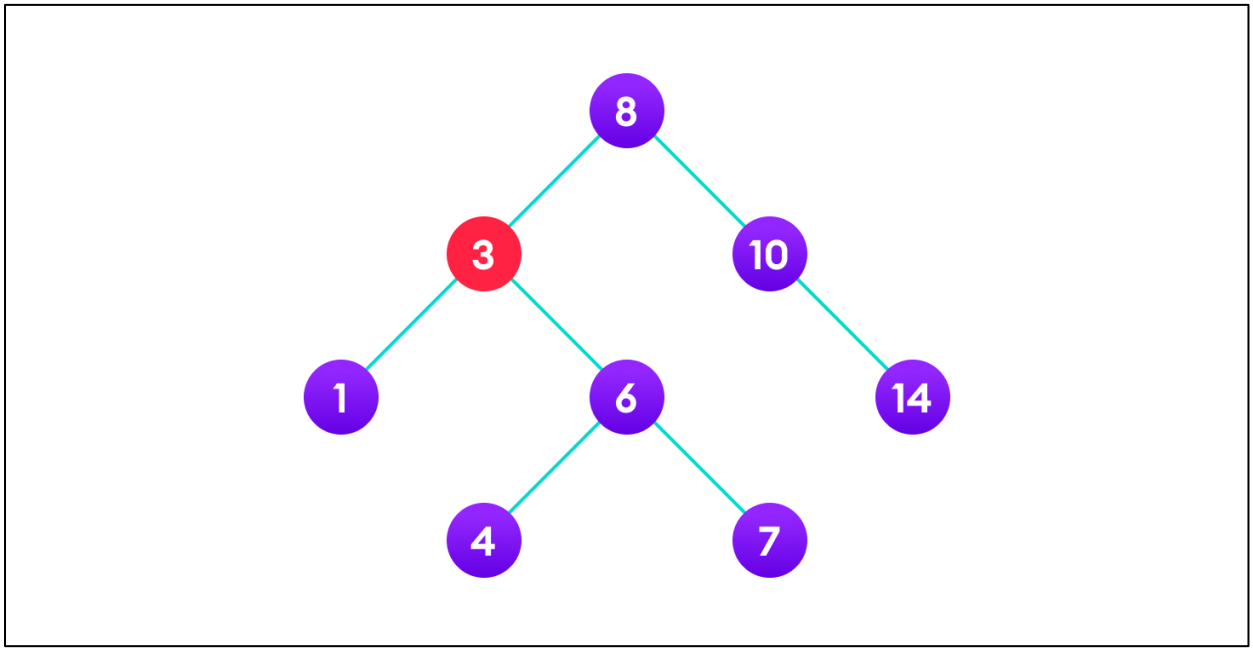
Case III

In the third case, the node to be deleted has two children. In such a case follow the steps below:

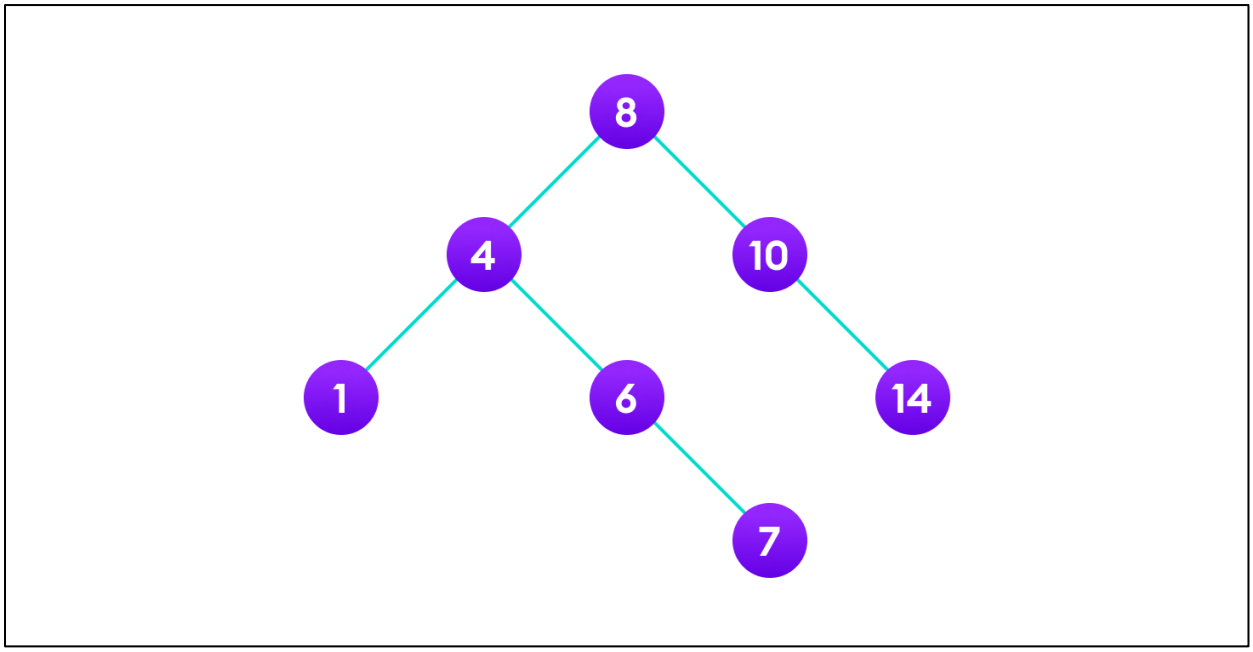
- Get the inorder successor of that node.
- Replace the node with the inorder successor.
- Remove the inorder successor from its original position.



3 is to be deleted



Copy the value of the inorder successor (4) to the node



Delete the inorder successor

Binary Search Tree Complexities

Time Complexity

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

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

Space Complexity

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

Binary Search Tree Applications

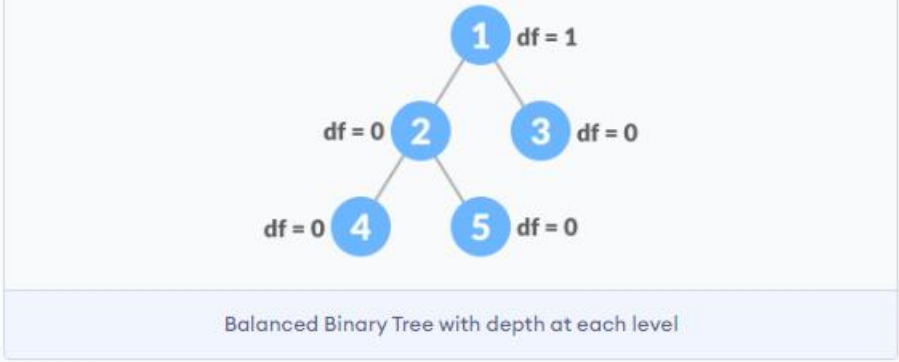
- In multilevel indexing in the database
- For dynamic sorting
- For managing virtual memory areas in Unix kernel

4. BALANCED BINARY TREE

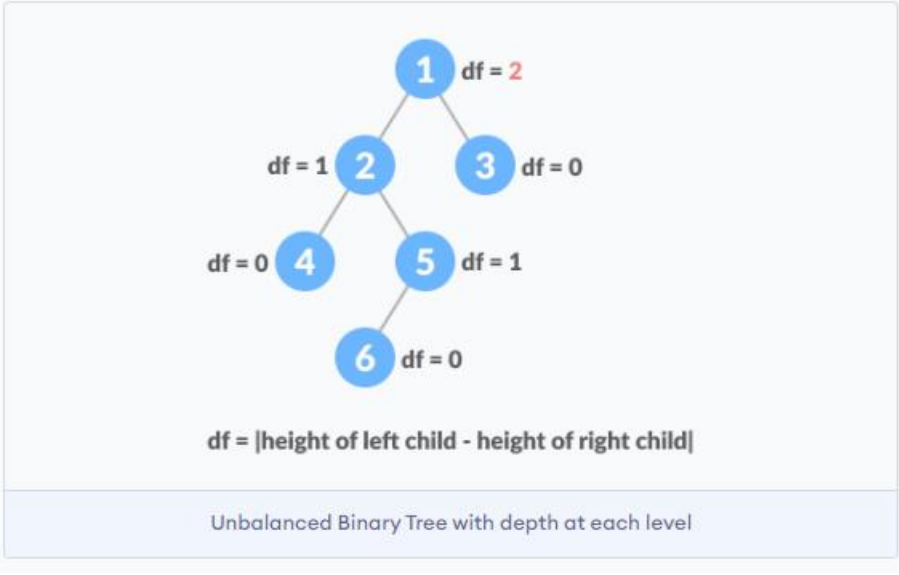
A balanced binary tree, also referred to as a height-balanced binary tree, is defined as a binary tree in which the height of the left and right subtree of any node differ by not more than 1.

To learn more about the height of a tree/node, visit Tree Data Structure.Following are the conditions for a height-balanced binary tree:

1. difference between the left and the right subtree for any node is not more than one
2. the left subtree is balanced
3. the right subtree is balanced



Balanced Binary Tree with depth at each level



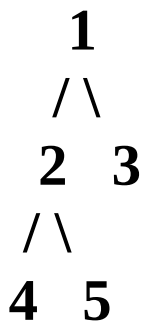
Unbalanced Binary Tree with depth at each level

Python Examples

The following code is for checking whether a tree is height-balanced.

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


def isHeightBalanced(root):
    if root is None:
        return True
    left_height = getHeight(root.left)
    right_height = getHeight(root.right)

    if abs(left_height - right_height) <= 1 and
isHeightBalanced(root.left) and isHeightBalanced(root.right):
        return True
    return False

def getHeight(node):
    if node is None:
        return 0
    return 1 + max(getHeight(node.left), getHeight(node.right))


root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

if isHeightBalanced(root):
    print('The tree is balanced')
else:
    print('The tree is not balanced')
```

output: The tree is balanced



Balanced Binary Tree Applications

- AVL tree
- Balanced Binary Search Tree

5. SPLAY TREE:

Splay tree is a self-adjusting binary search tree data structure, which means that the tree structure is adjusted dynamically based on the accessed or inserted elements. In other words, the tree automatically reorganizes itself so that frequently accessed or inserted elements become closer to the root node.

Splay trees are a type of self-adjusting binary search tree introduced by Sleator and Tarjan in 1985. They provide efficient search, insertion, and deletion operations in O(log n) amortized time complexity. The key idea is to bring the most recently accessed or inserted element to the root by performing splaying, a sequence of tree rotations. Splay trees are useful for dynamic data structures requiring fast access, like caching and network routing. However, they lack guaranteed balance and may perform poorly in worst-case scenarios, making them unsuitable for real-time or safety-critical systems.

Overall, splay trees are a powerful and versatile data structure that offers fast and efficient access to frequently accessed or inserted elements. They are widely used in various applications and provide an excellent tradeoff between performance and simplicity.

- The key feature of a splay tree is that each time an element is

accessed, it is moved to the root of the tree, creating a more balanced structure for subsequent accesses.

- Splay trees are characterized by their use of rotations, which are local transformations of the tree that change its shape but preserve the order of the elements.
- Rotations are used to bring the accessed element to the root of the tree, and also to rebalance the tree if it becomes unbalanced after multiple accesses.

OPERATIONS IN SPLAY TREE

- **Insertion:** Insert a new element as you would in a regular binary search tree, then perform rotations to bring the new element to the root.
- **Deletion:** Find the element to delete using a binary search, then remove it based on its number of children. Promote a single child or replace with the successor for two children.
- **Search:** Perform a binary search, splaying the accessed element to the root if found. If not found, splay the last visited node.
- **Rotation:** Use Zig rotations to bring a node to the root, and Zig-Zig rotations to balance the tree after multiple accesses in the same subtree

Here's a step-by-step explanation of the rotation operations:

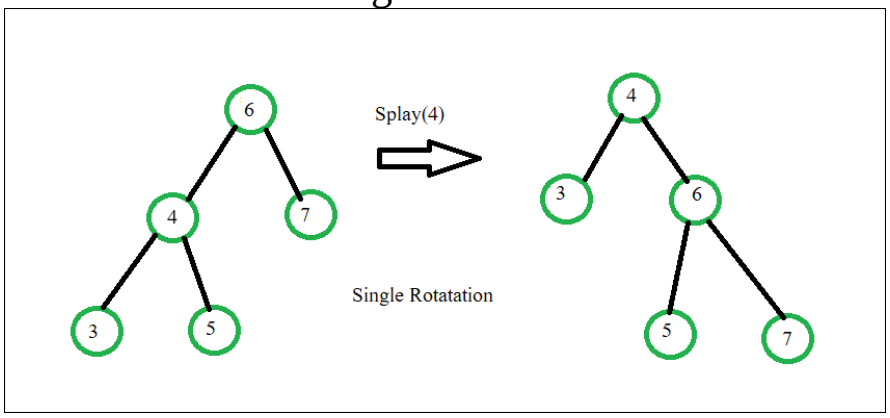
- **Zig Rotation:** If a node has a right child, perform a right rotation to bring it to the root. If it has a left child, perform a left rotation.
- **Zig-Zig Rotation:** If a node has a grandchild that is also its child's right or left child, perform a double rotation to balance the tree. For example, if the node has a right child and the right child has a left child, perform a right-left rotation. If the node has a left child and the left child has a right child, perform a left-right rotation.
- **Note:** The specific implementation details, including the exact rotations used, may vary depending on the exact form of the splay tree.

Rotations in Splay Tree

1. Zig Rotation
2. Zag Rotation
3. Zig – Zig Rotation
4. Zag – Zag Rotation
5. Zig – Zag Rotation
6. Zag – Zig Rotation

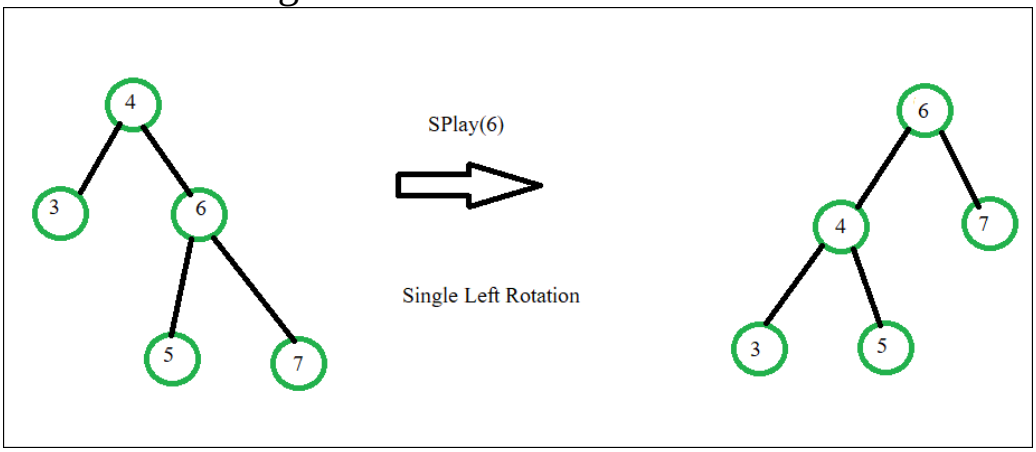
1) Zig Rotation:

The Zig Rotation in splay trees operates in a manner similar to the single right rotation in AVL Tree rotations. This rotation results in nodes moving one position to the right from their current location. For example, consider the following scenario:



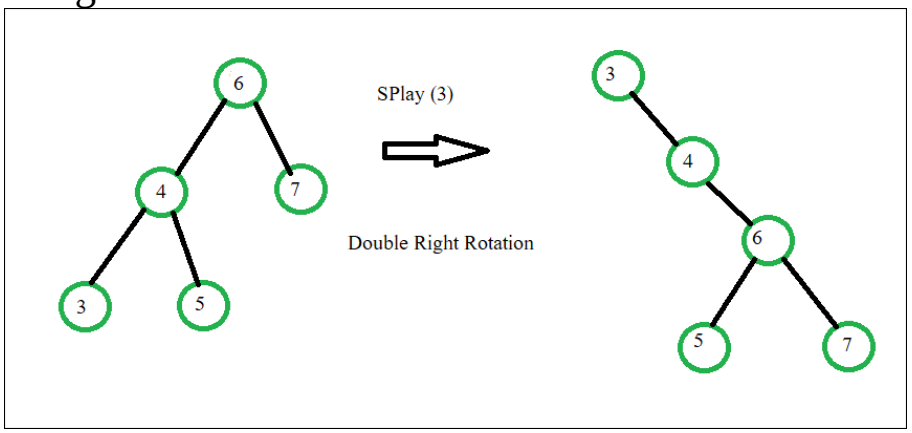
2) Zag Rotation:

The Zag Rotation in splay trees operates in a similar fashion to the single left rotation in AVL Tree rotations. During this rotation, nodes shift one position to the left from their current location. For instance, consider the following illustration:



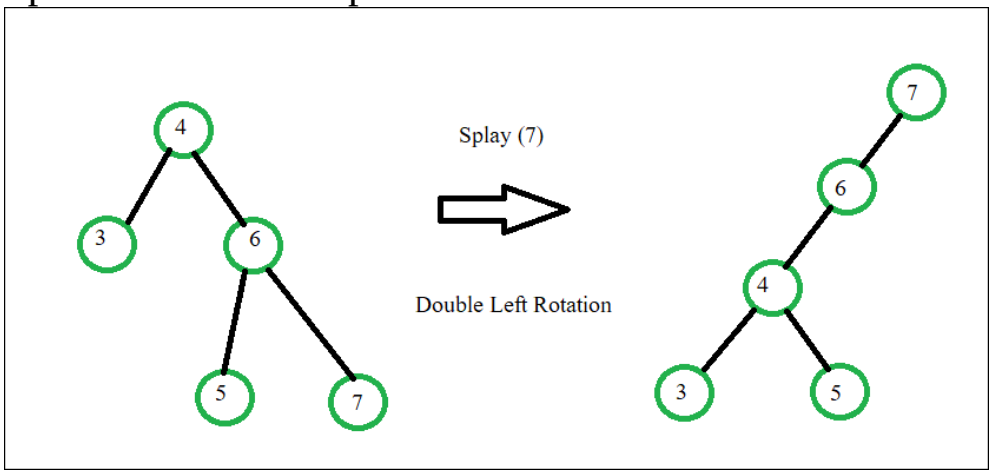
3) Zig-Zig Rotation:

The Zig-Zig Rotation in splay trees is a double zig rotation. This rotation results in nodes shifting two positions to the right from their current location. Take a look at the following example for a better understanding:



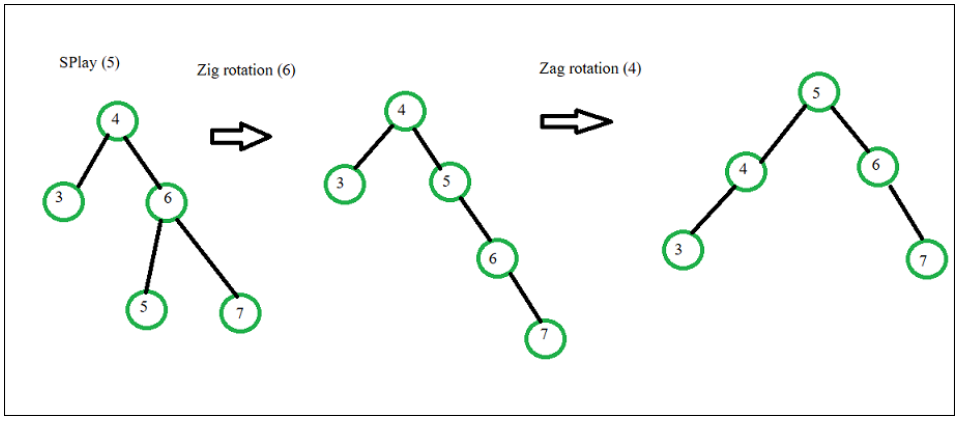
4) Zag-Zag Rotation:

In splay trees, the Zag-Zag Rotation is a double zag rotation. This rotation causes nodes to move two positions to the left from their present position. For example:



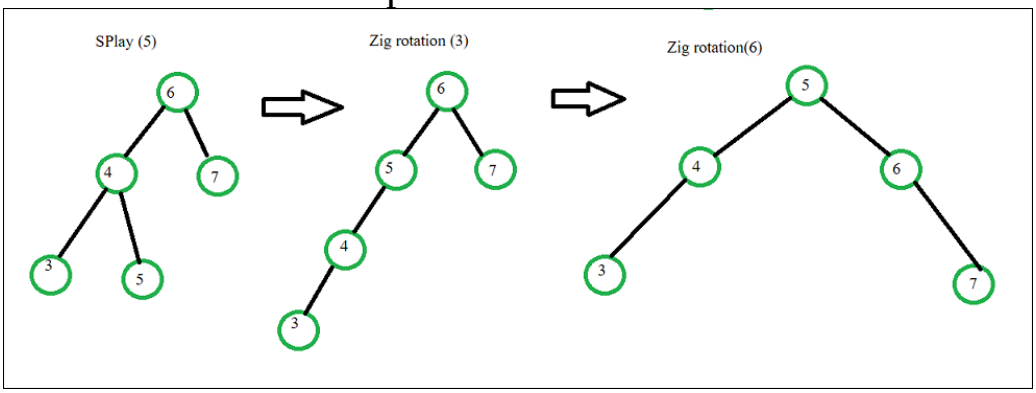
5) Zig-Zag Rotation:

The Zig-Zag Rotation in splay trees is a combination of a zig rotation followed by a zag rotation. As a result of this rotation, nodes shift one position to the right and then one position to the left from their current location. The following illustration provides a visual representation of this concept:



6) Zag-Zig Rotation:

The Zag-Zig Rotation in splay trees is a series of zag rotations followed by a zig rotation. This results in nodes moving one position to the left, followed by a shift one position to the right from their current location. The following illustration offers a visual representation of this concept:



Drawbacks of splay tree data structure:

- Unbalanced Trees
- Memory Usage
- Complexity
- Reorganization Overhead
- Limited Use Cases

Applications of the splay tree:

- Caching
- Database Indexing
- File Systems
- Data Compression
- Text Processing

- Graph Algorithms
- Online Gaming

Sure, here are some advantages and disadvantages of splay trees, as well as some recommended books for learning more about the topic:

Advantages of Splay Trees:

1. Splay trees have amortized time complexity of O (log n) for many operations, making them faster than many other balanced tree data structures in some cases.
2. Splay trees are self-adjusting, meaning that they automatically balance themselves as items are inserted and removed. This can help to avoid the performance degradation that can occur when a tree becomes unbalanced.

Disadvantages of Splay Trees:

1. Splay trees can have worst-case time complexity of O(n) for some operations, making them less predictable than other balanced tree data structures like AVL trees or red-black trees.
2. Splay trees may not be suitable for certain applications where predictable performance is required.

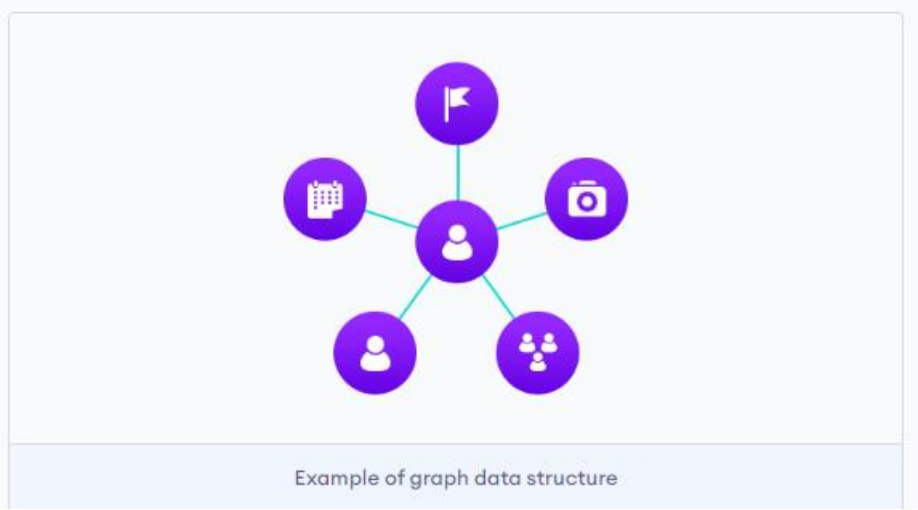
6. GRAPH DATA STUCTURE

Graph Data Structure is. Also, you will find representations of a graph.

A graph data structure is a collection of nodes that have data and are connected to other nodes.

Let's try to understand this through an example. On facebook, everything is a node. That includes User, Photo, Album, Event, Group, Page, Comment, Story, Video, Link, Note...anything that has data is a node.

Every relationship is an edge from one node to another. Whether you post a photo, join a group, like a page, etc., a new edge is created for that relationship.

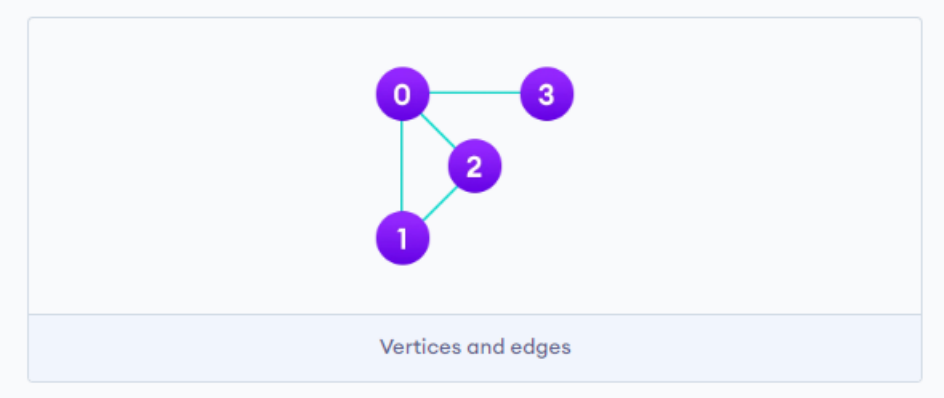


Example of graph data structure

All of facebook is then a collection of these nodes and edges. This is because facebook uses a graph data structure to store its data.

More precisely, a graph is a data structure (V, E) that consists of

- A collection of vertices V
- A collection of edges E, represented as ordered pairs of vertices (u,v)
- a graph contains vertices that are like points and edges that connect the points



Vertices and edges

In the graph,

V = {0, 1, 2, 3}
E = {(0,1), (0,2), (0,3), (1,2)}
G = {V, E}

Graph Terminology:

- **Adjacency:** A vertex is said to be adjacent to another vertex if there is an edge connecting them. Vertices 2 and 3 are not adjacent because there is no edge between them.
- **Path:** A sequence of edges that allows you to go from vertex A to vertex B is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.
- **Directed Graph:** A graph in which an edge (u,v) doesn't necessarily mean that there is an edge (v, u) as well. The edges in such a graph are represented by arrows to show the direction of the edge.

Graph Representation:

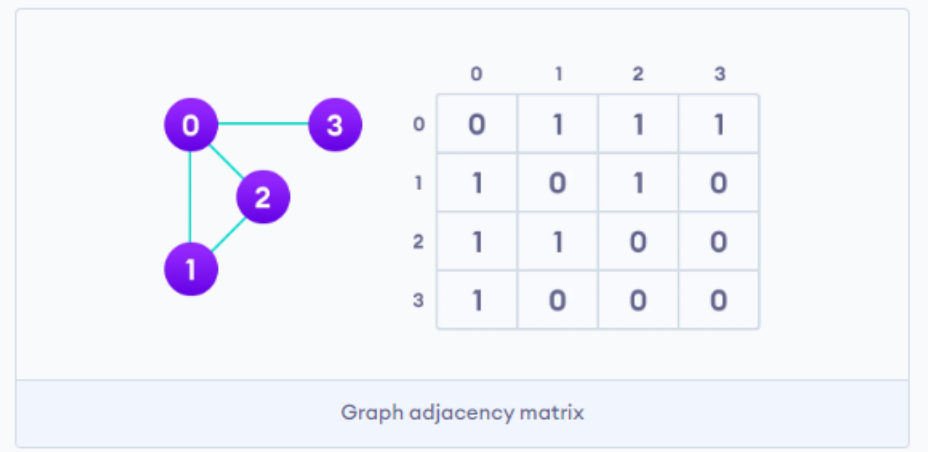
Graphs are commonly represented in two ways:

1. Adjacency Matrix

An adjacency matrix is a 2D array of V x V vertices. Each row and column represent a vertex.

If the value of any element a[i][j] is 1, it represents that there is an edge connecting vertex i and vertex j.

The adjacency matrix for the graph we created above is



Since it is an undirected graph, for edge (0,2), we also need to mark edge (2,0); making the adjacency matrix symmetric about the diagonal.

Edge lookup (checking if an edge exists between vertex A and vertex B) is extremely fast in adjacency matrix representation but we have to reserve space for every possible link between all vertices(V x V), so it requires more space.

2. Adjacency List

An adjacency list represents a graph as an array of linked lists.

The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

The adjacency list for the graph we made in the first example is as follows:

An adjacency list is efficient in terms of storage because we only need to store the values for the edges. For a graph with millions of vertices, this can mean a lot of saved space.

Graph Operations:

The most common graph operations are:

- Check if the element is present in the graph
- Graph Traversal
- Add elements (vertex, edges) to graph
- Finding the path from one vertex to another

7. GRAPH TRAVERSAL TECHNIQUE IN DATA STRUCTURE

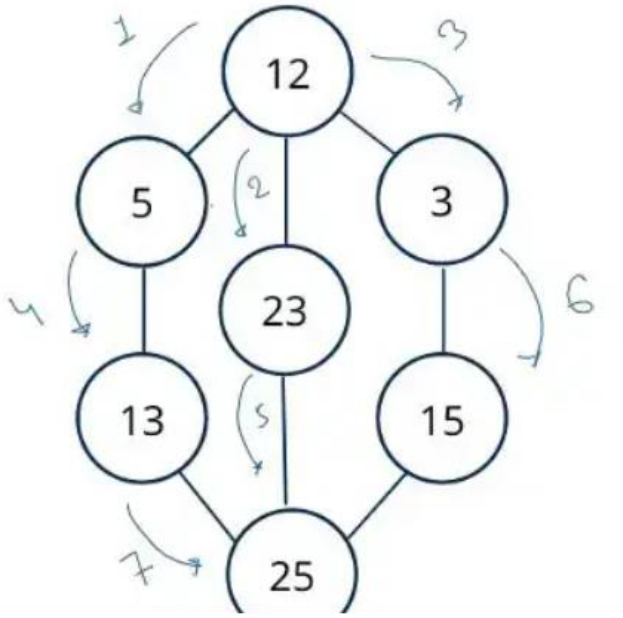
Graph traversal is a technique to visit each nodes of a graph G. It is also use to calculate the order of vertices in traverse process. We visit all the nodes starting from one node which is connected to each other without going into loop.
Basically, in graph it may happen sometime visiters can visit one node more than once. So, this may cause the going the visiters into infinite loop. So, to protect from this infinite loop condition we keep record of each vertex. Like if visiters visited vertex then the value will be zero if not, then one.

There are two graph traversal techniques

1. Breadth-first search
2. Depth-first search

Breadth-first search

Breadth-first search graph traversal techniques use a queue data structure as an auxiliary data structure to store nodes for further processing. The size of the queue will be the maximum total number of vertices in the graph.



1. Using a Queue (FIFO):

```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])

    while queue:
        vertex = queue.popleft()
        if vertex not in visited:
            visited.add(vertex)
            print(vertex)  # Process the vertex (or perform any other
desired operation)
            queue.extend(graph[vertex] - visited)

# Example usage:
graph = {
    'A': {'B', 'C'},
    'B': {'A', 'D', 'E'},
    'C': {'A', 'F'},
    'D': {'B'},
    'E': {'B', 'F'},
    'F': {'C', 'E'}
}

bfs(graph, 'A')
```

2. Using a Queue (Python list):

```
def bfs(graph, start):
    visited = set()
    queue = [start]

    while queue:
        vertex = queue.pop(0)
        if vertex not in visited:
            visited.add(vertex)
            print(vertex)  # Process the vertex (or perform any other
```

```
desired operation)
            queue.extend(graph[vertex] - visited)

# Example usage:
graph = {
    'A': {'B', 'C'},
    'B': {'A', 'D', 'E'},
    'C': {'A', 'F'},
    'D': {'B'},
    'E': {'B', 'F'},
    'F': {'C', 'E'}
}

bfs(graph, 'A')
```

output:

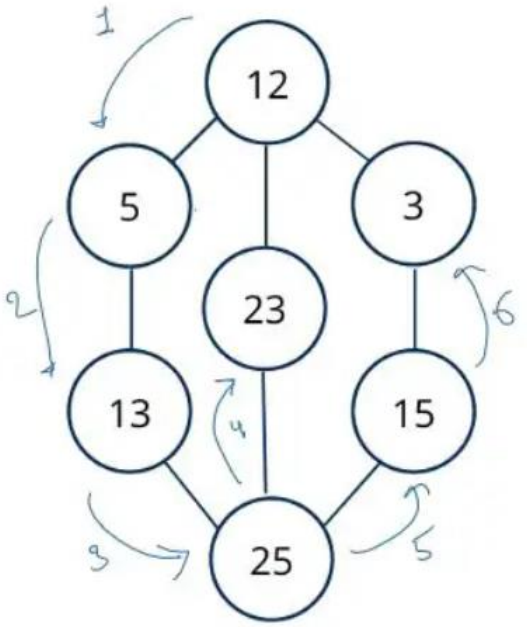
```
A
B
C
D
E
F
```

In both implementations, we maintain a visited set to keep track of visited vertices and ensure we don't process them multiple times. The algorithm starts with the start vertex and uses a queue to store the vertices to be visited. We iterate until the queue is empty, dequeueing a vertex, processing it, and enqueueing its unvisited neighboring vertices. This ensures that vertices at each level are visited before moving to the next level, resulting in a breadth-first traversal.

Depth-first search

DFS stands for Depth First Search, is one of the graph traversal algorithms that use Stack data structure. In DFS Traversal go as deep as possible of the graph and then backtrack once reached a vertex that has all its adjacent vertices already visited.

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



DFS traversal for tree and graph is similar but the only difference is that a graph can have a cycle but the tree does not have any cycle. So, in the graph we have additional array which keeps the record of visited array to protect from infinite loop and not visited again the visited node.

Implementation

```
class Node:
    def __init__(self,value):
        self.value=value
        self.left=None
        self.right=None
    def depth_first_search(root):
        if root is None:
```

```
            return
        print(root.value)
        depth_first_search(root.left)
        depth_first_search(root.right)

root=Node(50)
root.left=Node(10)
root.right=Node(25)
root.left.left=Node(18)
root.left.right=Node(36)
root.right.left=Node(22)
root.right.right=Node(46)

depth_first_search(root)
```

output:

```
50
10
18
36
25
22
46
```

8. Comparing Sorting Algorithms

At this point, it might be useful for us to take a moment and consider all the algorithms we have studied in this book to sort an *n*-element sequence.

Among the sorting algorithms discussed, there is no clear "best" algorithm that fits all scenarios. The choice of the best sorting algorithm depends on various factors such as efficiency, memory usage, and stability, as well as the specific properties of the application at hand. Here is a simplified summary of the remaining sorting algorithms:

1. Insertion Sort and Selection Sort:

- Both have an average and worst-case time complexity of O(n^2).
- Selection Sort is generally considered inefficient and is not commonly used.
- Insertion Sort can be efficient for small input sizes or nearly sorted arrays.

2. Heap Sort, Merge Sort, and Quick Sort:

- All have an average time complexity of O(n log n).
- Heap Sort has a worst-case time complexity of O(n log n), making it suitable for worst-case scenarios.
- Merge Sort has good stability and is a good choice when stability is a requirement.
- Quick Sort is usually faster in practice due to its efficient partitioning and cache-friendly nature.

3. Bucket Sort and Radix Sort:

- Both have a linear time complexity for certain types of keys.
- Bucket Sort is efficient when the input is uniformly distributed across a range.
- Radix Sort is suitable for sorting integers with a fixed number of digits or keys with fixed-length representations.

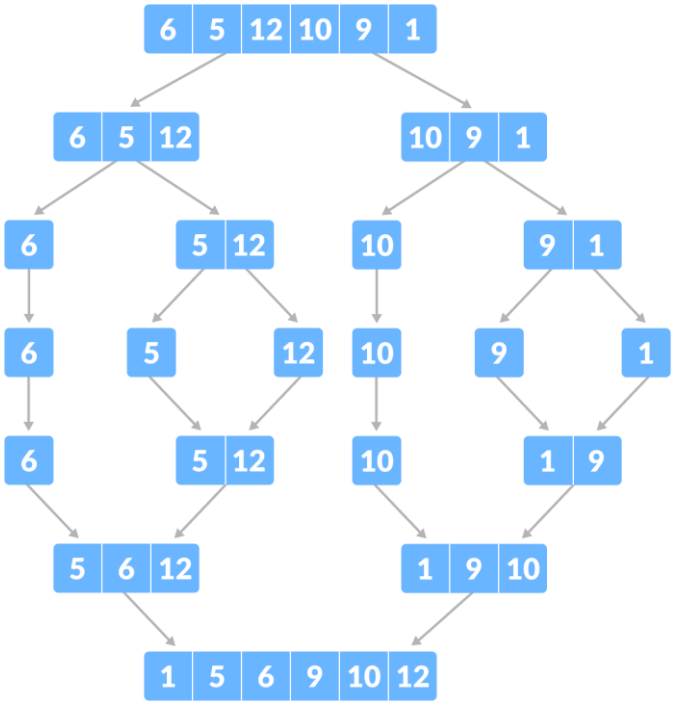
In practice, the default sorting algorithm used by programming languages and systems has evolved over time. Modern implementations often combine different sorting algorithms based on input size, such as using Insertion Sort for small arrays and switching to a more efficient algorithm for larger arrays.

When choosing a sorting algorithm, consider the size of the input, the distribution of the data, the stability requirement, memory constraints, and any specific characteristics of the application. It is essential to analyze the properties of the data and conduct performance tests to determine the most suitable sorting algorithm for a given scenario.

9. Merge Sort Algorithm

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.



Merge Sort example

Divide and Conquer Strategy

Using the **Divide and Conquer** technique, we divide a problem into subproblems. When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem. Suppose we had to sort an array `A`. A subproblem would be to sort a sub-section of this array starting at index `p` and ending at index `r`, denoted as `A[p..r]`.

Divide

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

Conquer

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

Combine

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

MergeSort Algorithm

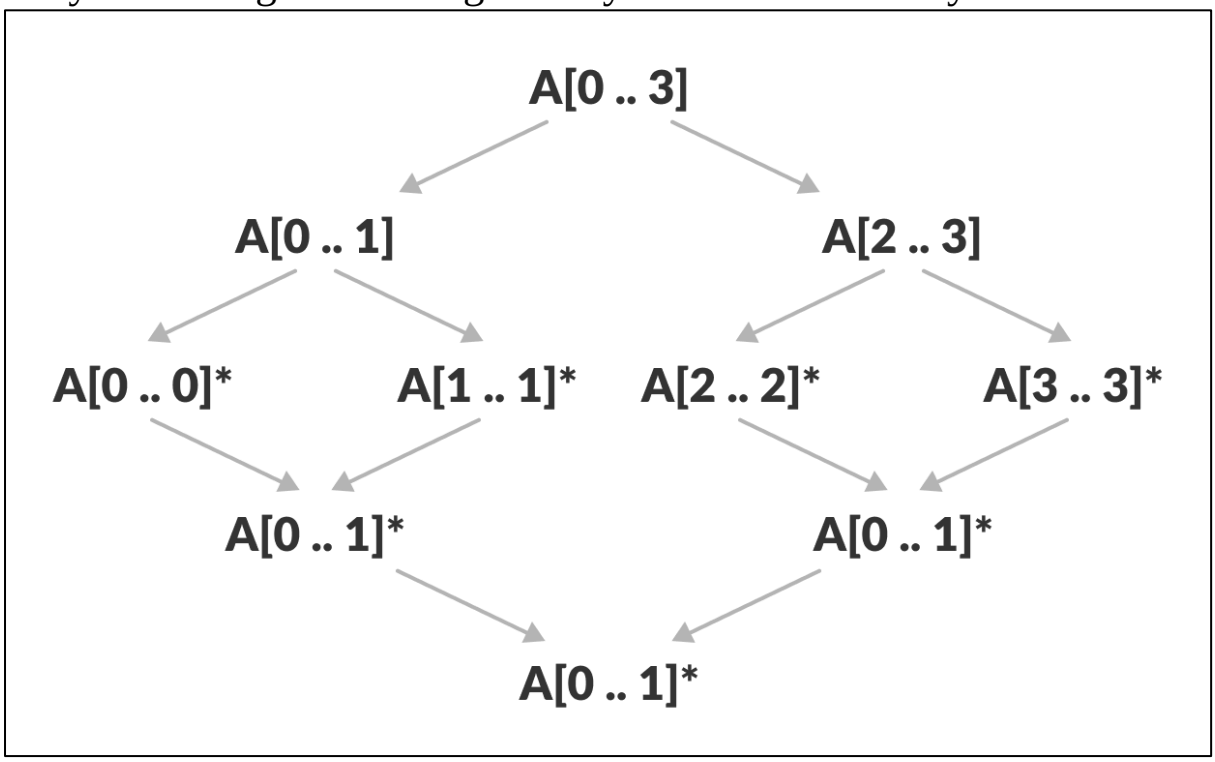
The MergeSort function repeatedly divides the array into two halves until we reach a stage where we try to perform MergeSort on a subarray of size 1 i.e. `p == r`.

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

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

To sort an entire array, we need to call `MergeSort(A, 0, length(A)-1)`. As shown in the image below, the merge sort algorithm recursively

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



Merge sort in action

The merge **Step of Merge Sort**

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

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

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

Have we reached the end of any of the arrays?

No:

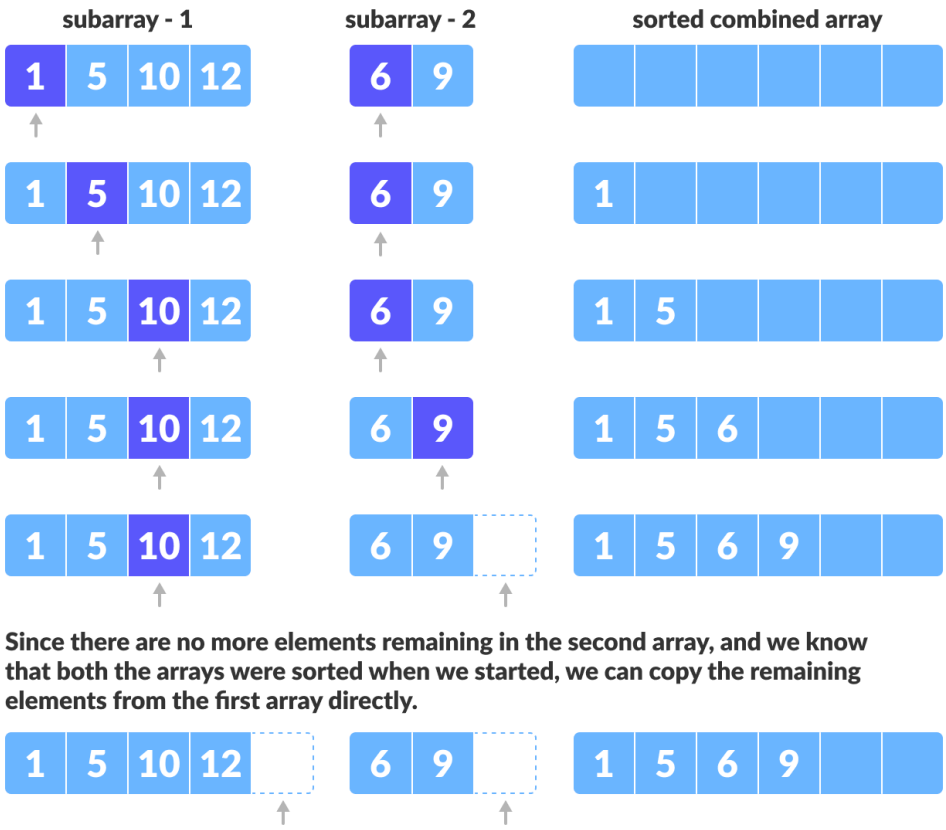
Compare current elements of both arrays

Copy smaller element into sorted array

Move pointer of element containing smaller element

Yes:

Copy all remaining elements of non-empty array



Merge step

Merge( ) Function Explained Step-By-Step

A lot is happening in this function, so let's take an example to see how this would work.

As usual, a picture speaks a thousand words.

Merging two consecutive subarrays of array

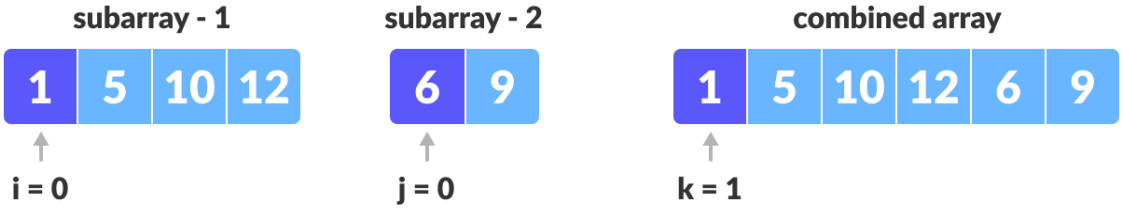
The array A[0..5] contains two sorted subarrays A[0..3] and A[4..5]. Let us see how the merge function will merge the two arrays.

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

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

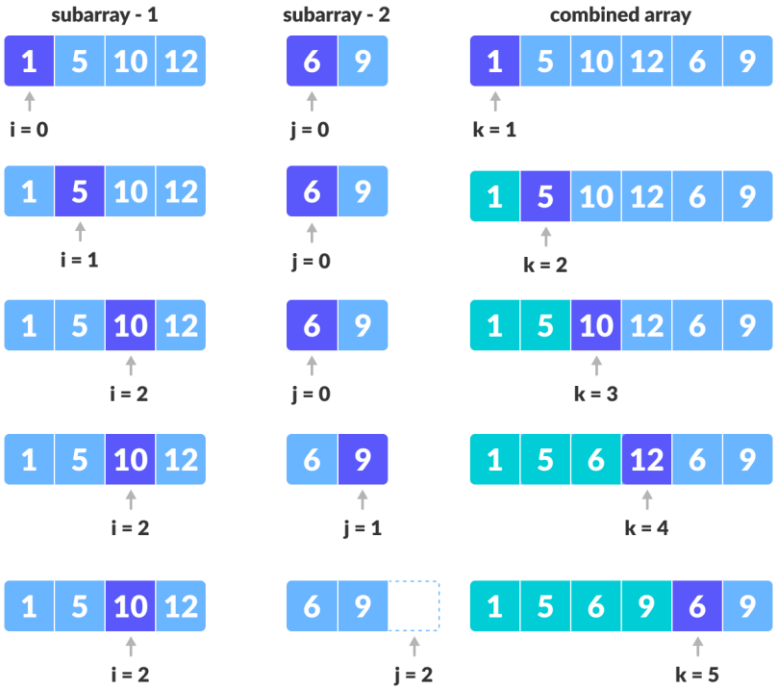
Create copies of subarrays for merging

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



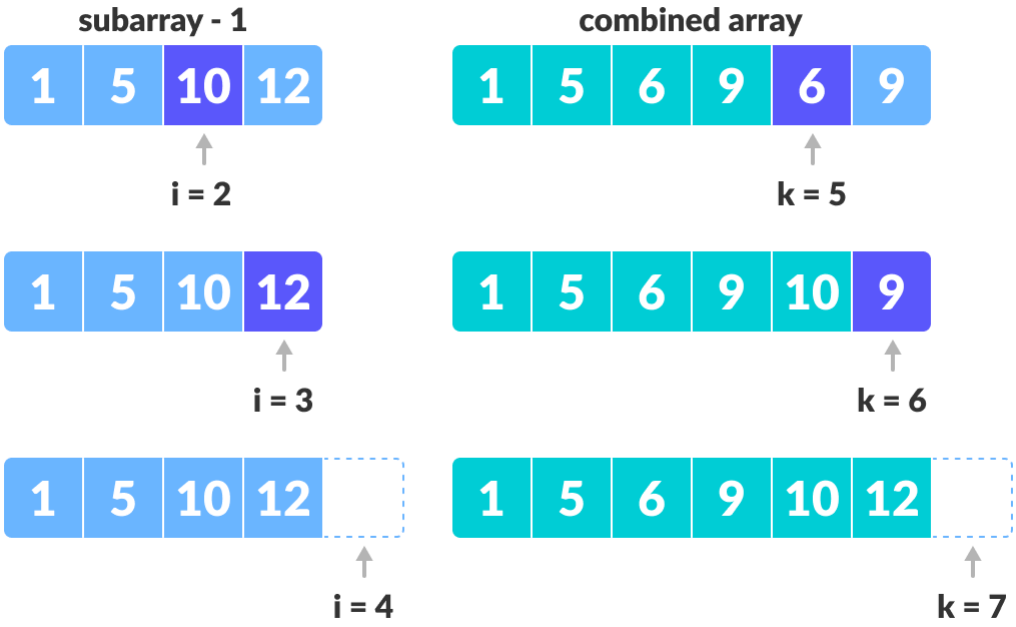
Maintain indices of copies of sub array and main array

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

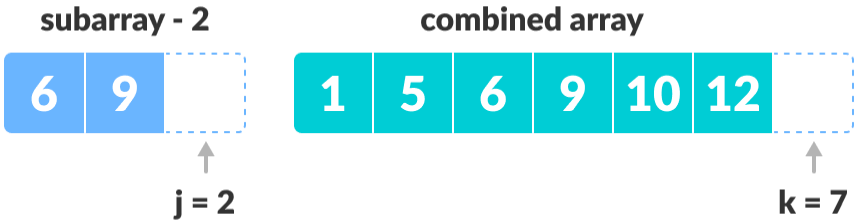


Comparing individual elements of sorted subarrays until we reach end of one

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



Copy the remaining elements from the first array to main subarray



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

Merge Sort Code in Python

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
```

```
    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])

    return result

arr = input("Enter the list of numbers separated by space: ").split()
arr = [int(num) for num in arr]

print("Input Array:", arr)
sorted_arr = merge_sort(arr)
print("Sorted Array:", sorted_arr)
```

output

```
Enter the list of numbers separated by space: 8 3 5 1 4
Input Array: [8, 3, 5, 1, 4]
Sorted Array: [1, 3, 4, 5, 8]
```

Merge Sort Complexity

Time Complexity

Best	O(n*log n)

Worst	O(n*log n)
Average	O(n*log n)
Space Complexity	O(n)
Stability	Yes

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

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

Merge Sort Applications

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

10. QUICKSORT

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).
   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.

2. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
3. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

Working of Quicksort Algorithm

1. Select the Pivot Element

There are different variations of quicksort where the pivot element is selected from different positions. Here, we will be selecting the rightmost element of the array as the pivot element.

Select a pivot element

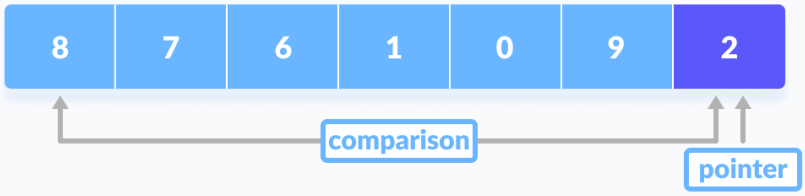
2. Rearrange the Array

Now the elements of the array are rearranged so that elements that are smaller than the pivot are put on the left and the elements greater than the pivot are put on the right.

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

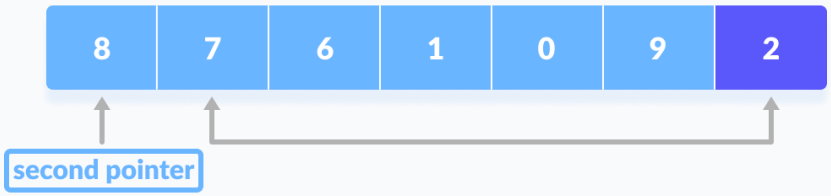
Here's how we rearrange the array:

1. A pointer is fixed at the pivot element. The pivot element is compared with the elements beginning from the first index.



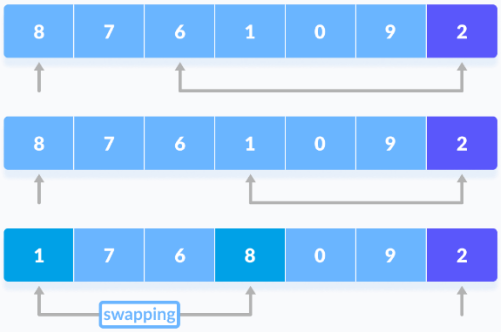
Comparison of pivot element with element beginning from the first index

2. If the element is greater than the pivot element, a second pointer is set for that element.



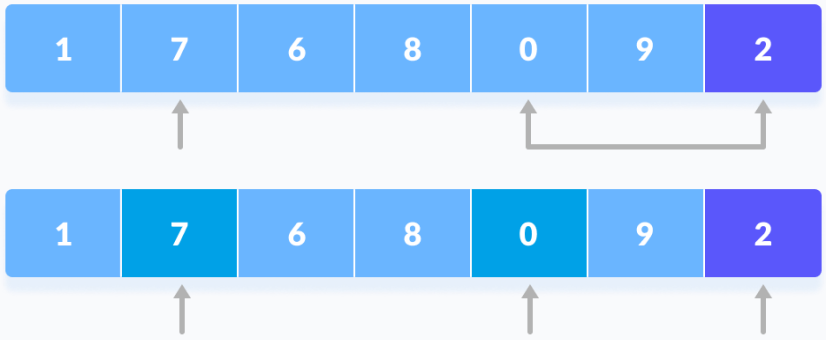
If the element is greater than the pivot element, a second pointer is set for that element.

3. Now, pivot is compared with other elements. If an element smaller than the pivot element is reached, the smaller element is swapped with the greater element found earlier.



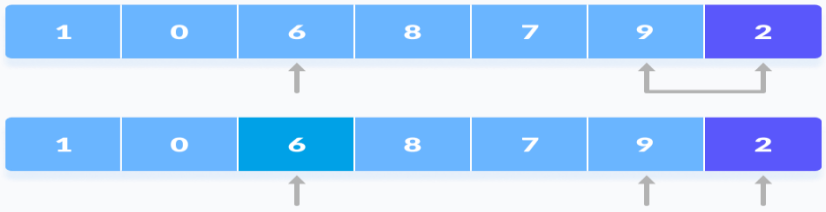
Pivot is compared with other elements.

4. Again, the process is repeated to set the next greater element as the second pointer. And, swap it with another smaller element.



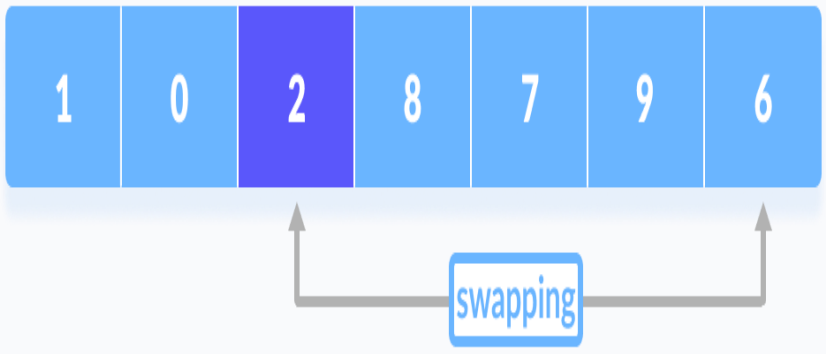
The process is repeated to set the next greater element as the second pointer.

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



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

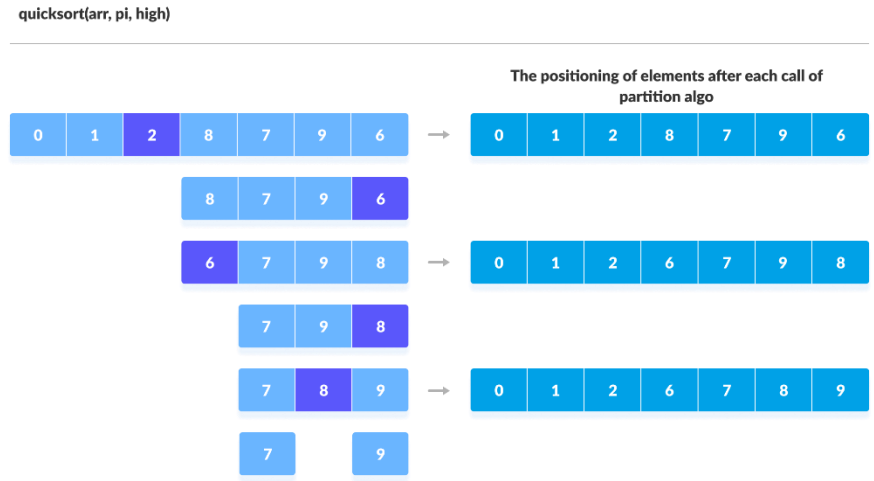
6. Finally, the pivot element is swapped with the second pointer.



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

3. Divide Subarrays

Pivot elements are again chosen for the left and the right sub-parts separately. And, **step 2** is repeated.



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

The subarrays are divided until each subarray is formed of a single element. At this point, the array is already sorted.

Quicksort Code in Python

```
def quick_sort(arr):
        if len(arr)<=1:
                return arr
pivot=arr[len(arr)//2]
left=[x for x in arr if x<pivot]
middle=[x for x in arr if x==pivot]
right=[x for x in arr if x>pivot]
return quick_sort(left)+middle+quick_sort(right)

arr=[]
n=int(input("Enter number of elements: "))
for i in range(n):
        element=int(input("Enter element {}: ".format(i+1)))
```

```
        arr.append(element)

print("Unsorted array: ",arr)
sorted_arr=quick_sort(arr)
print("Sorted array: ",sorted_arr)
```

output:

```
Enter number of elements: 5
Enter element 1: 23
Enter element 2: 76
Enter element 3: 48
Enter element 4: 56
Enter element 5: 1
Unsorted array:  [23, 76, 48, 56, 1]
Sorted array:  [1, 23, 48, 56, 76]
```

Quicksort Complexity

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

1. Time Complexities

- **Worst Case Complexity [Big-O]**: $O(n^2)$
  It occurs when the pivot element picked is either the greatest or the smallest element.
  This condition leads to the case in which the pivot element lies in an extreme end of the sorted array. One sub-array is always empty and another sub-array contains $n - 1$ elements. Thus,

quicksort is called only on this sub-array.
However, the quicksort algorithm has better performance for scattered pivots.

- **Best Case Complexity [Big-omega]**: O(n*log n)
  It occurs when the pivot element is always the middle element or near to the middle element.
- **Average Case Complexity [Big-theta]**: O(n*log n)
  It occurs when the above conditions do not occur.

2. Space Complexity

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

Quicksort Applications

Quicksort algorithm is used when

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

Similar Sorting Algorithms

- Insertion Sort
- Merge Sort
- Selection Sort
- Bucket Sort