**Python & Data Structures Laboratory B.Tech. 3rd Semester**



**Name : Harshit Kumar**

**Roll Number : 21ETMC412011**

**Section/Course : [M&C] Mathematics and Computing**

**Department : Computer Science and Engineering**

**Faculty of Engineering & Technology**

**Ramaiah University of Applied Sciences**

**Ramaiah University of Applied Sciences**

Private University Established in Karnataka State by Act No. 15 of 2013

|  |  |
| --- | --- |
| Faculty | Engineering & Technology |
| Programme | B. Tech. in AIML, ISE and MC |
| Year/Semester | 2021 / 3rd |
| Name of the Laboratory | Python & Data Structures Laboratory |
| Laboratory Code | 21CSL206A |

# List of Lab Experiments

|  |  |  |  |
| --- | --- | --- | --- |
| **No** | **Lab Experiment** | **Marks (10)** | **Faculty Signature** |
| 1 | Array |  |  |
| 2 | Linked List |  |  |
| 3 | Stack |  |  |
| 4 | Queue |  |  |
| 5 | Binary Tree |  |  |
| 6 | Binary Search Tree |  |  |
| 7 | Heap |  |  |
| 8 | AVL Tree |  |  |
| 9 | Quick Sort |  |  |
| 10 | Merge Sort |  |  |
| **Total Marks (100)** | |  |  |

**Signature of the Faculty In-charge**

# Experiment 1

**Title of the laboratory experiment**: Array

**1. Aim:**

To understand and implement the basic operations in arrays using python.

**2. Objective:**

To execute the below operations:

1. Traverse − print all the array elements one by one.
2. Insertion − Adds an element at the given index.
3. Deletion − Deletes an element at the given index.
4. Search − Searches an element using the given index or by the value.
5. Update − Updates an element at the given index.

**3. Exercise:**

To develop a python to perform the below tasks:

1. Create your own list of your favourite five sportsperson. Using this find out,
2. Length of the list.
3. Add a sixth sportsperson at the end of this list.
4. You realize that you need to add the sixth sportsperson after the second sportsperson, so remove it from the list first and then add it after the second sportsperson.
5. Now you don't like two sportspersons. Now remove those two and replace them with any other two sportspersons.
6. Sort the sportspersons list in alphabetical order (hint: use the dir() functions to list down all functions available in the list).
7. Create a list of all even numbers between number x and number y.
8. Sort the sportspersons list in alphabetical order (hint: use the dir() functions to list down all functions available in the list). The number x should be your age, and the number y should be your father's or mother's age.

**4. Experimental Procedure**

1. Algorithm design

“Write the pseudocode of the main operations of the given data structure”

**algorithm1:**

Step 1: Create an array list of 5 elements , ' l ' .

Step 2: Find the length of list using function len().

Step 3: Add 6th element at the end of array using append() function .

Step 4: print modified arraylist , l

Step 5: Remove last element of array using pop() and then add it after the second element using function ().

Step 6: print modified arraylist , l

Step 7: Remove any 2 elemnts from list using their index number by remove() function

Step 8 : And insert new elements using insert()

Step 9: print modified arraylist , l

Step 10: Sort the list using sort()

Step 11: Display the sorted list in alphatbetic order.

**algorithm2:**

Step 1: Enter the input integer in " x "

Step 2: Enter the input integer in " y "

Step 3: create empty list ' l '

Step 4: for i in range(x,y+1):

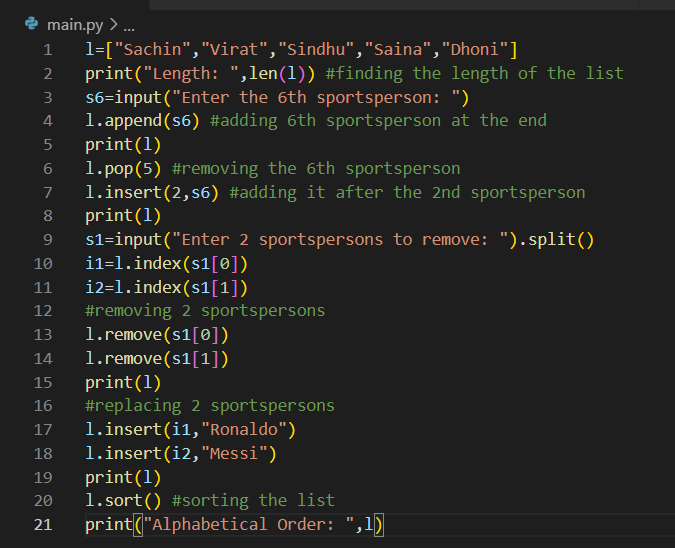
if i%2==0 , then append(i) in list ' l '

Step 5: Print the modified list ' l '

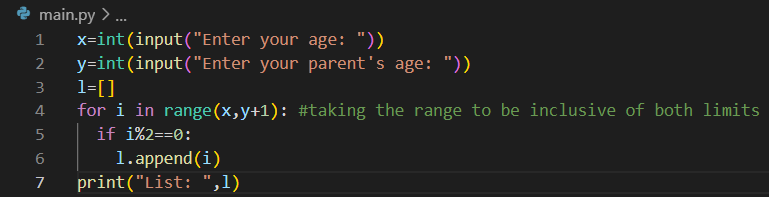
1. Program

“Paste the screenshot of the executed python code”

**Program-1:**

****

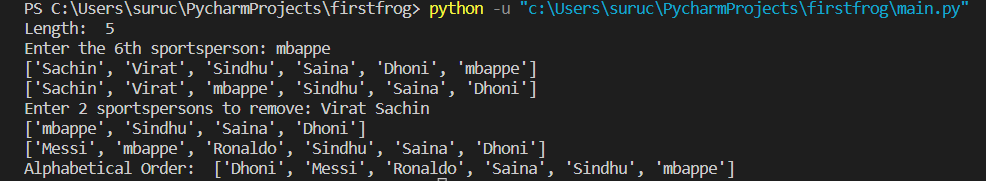
**Program-2:**

****

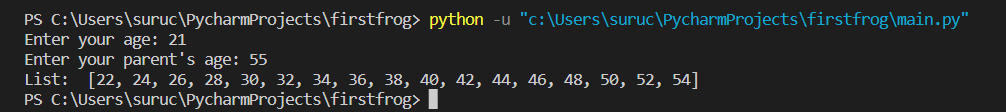
1. Presentation of the results

“Paste the output of the program”

**Output-1:**

****

**Output-2:**

****

1. Analysis and discussions

“Discuss the time complexities of all the operations of the given data structure”

The time complexity of operations on an array data structure can vary depending on the specific implementation and the algorithm used. However, in general, here are the time complexities of common operations on an array:

1. Accessing an element: O(1) Accessing a specific element in an array is a constant time operation. This is because arrays provide direct access to any element using its index.
2. Searching for an element: O(n) In the worst case, searching for an element in an unsorted array requires checking every element, which takes O(n) time where n is the number of elements in the array. In a sorted array, a binary search can be used to achieve a time complexity of O(log n).
3. Inserting an element at the end: O(1) Inserting an element at the end of an array is a constant time operation because it does not require any shifting of elements. However, inserting an element at the beginning or in the middle of an array may require shifting all subsequent elements, which takes O(n) time.
4. Removing an element from the end: O(1) Removing an element from the end of an array is a constant time operation because it does not require any shifting of elements. However, removing an element from the beginning or in the middle of an array may require shifting all subsequent elements, which takes O(n) time**.**
5. Sorting an array: O(n log n) Sorting an unsorted array using a comparison-based sorting algorithm, such as quicksort or mergesort, takes O(n log n) time in the average case. However, some special cases, such as already sorted or nearly sorted arrays, may be sorted faster.
6. Merging two sorted arrays: O(n) Merging two sorted arrays takes O(n) time, where n is the total number of elements in the two arrays. This is because each element is compared only once during the merging process.
7. Resizing an array: O(n) Resizing an array requires creating a new, larger array and copying all the elements from the old array to the new array. This takes O(n) time where n is the number of elements in the array.

Top of Form

# Experiment 2

**Title of the Laboratory Exercise**: Linked List

**1. Aim:**

To understand and implement the basic operations in Circular Doubly Linked List using python.

**2. Objective:**

To execute the below operations in Circular Doubly Linked List:

1. Insert: Inserts an element after a specific value.
2. Delete: Deletes an element having a specific value.
3. Display: Prints the elements in the forward direction as well as in the reverse direction.

**3. Exercise:**

In a Circular Doubly Linked List class, implement the below four operations:

**def insert\_after\_value(self, data\_after, data\_to\_insert):**

# Search for first occurance of data\_after value in linked list

# Now insert data\_to\_insert after data\_after node

**def remove\_by\_value(self, data):**

# Remove first node that contains data

**def print\_forward(self):**

# This method prints list in forward direction. Use node.next. Use a print statement to print the nodes in forward direction starting from the first node to the last node.

**def print\_backward(self):**

# Print linked list in reverse direction. Use node.prev for this. Use a print statement to print the nodes in backward direction starting from the last node to the first node.

Now make following calls,

LL = LinkedList()

LL.insert\_values(["Red","Yellow","Purple","Orange"])

LL.print()

LL.insert\_after\_value("Yellow","Blue") # insert Blue after Yellow

LL.print()

LL.remove\_by\_value("orange") # remove Orange from linked list

LL.print()

LL.remove\_by\_value("Green")

LL.print()

LL.remove\_by\_value("Red")

LL.remove\_by\_value("Yellow")

LL.remove\_by\_value("Blue")

LL.remove\_by\_value("Purple")

LL.print()

LL.print\_forward()

LL.print\_backward()

**4. Experimental Procedure**

1. Algorithm design

“Write the pseudocode of the main operations of the given data structure”

**algorithm:**

Step 1: Construct a class of linkedlist.

Step 2: Add elements in the empty list.

Step 3: Print the modified list.

* Step 4: Inserts an element after a specific value.

Step 5: Print the linkedlist.

Step 6: Removing first node that contains data

Step 7: print the linkedlist

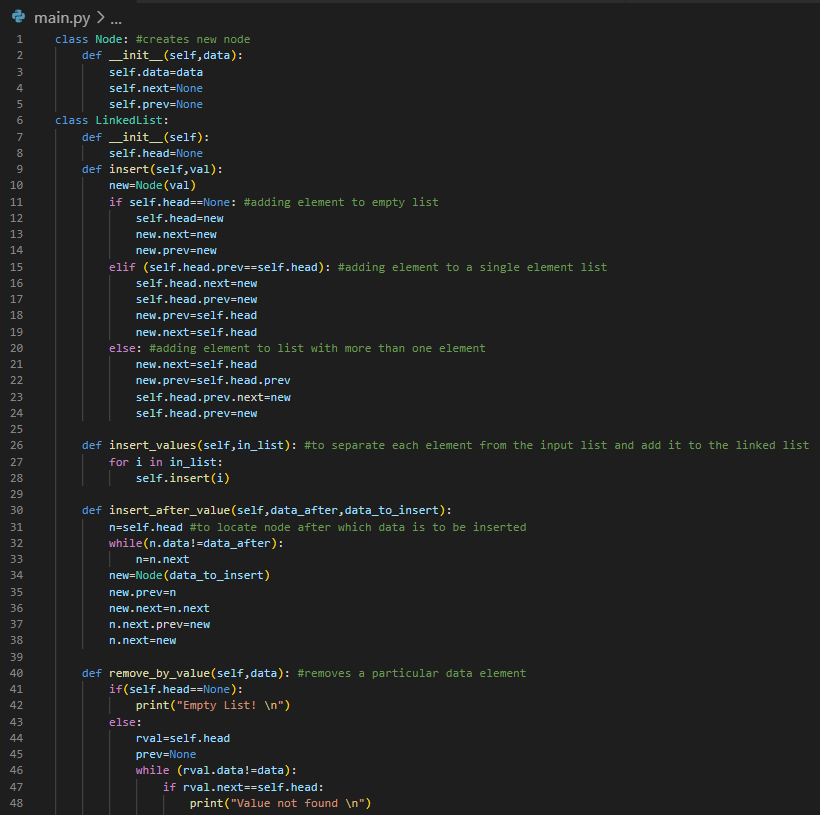
Step 8 : Use a print statement to print the nodes in forward direction starting from the first node to the last node.

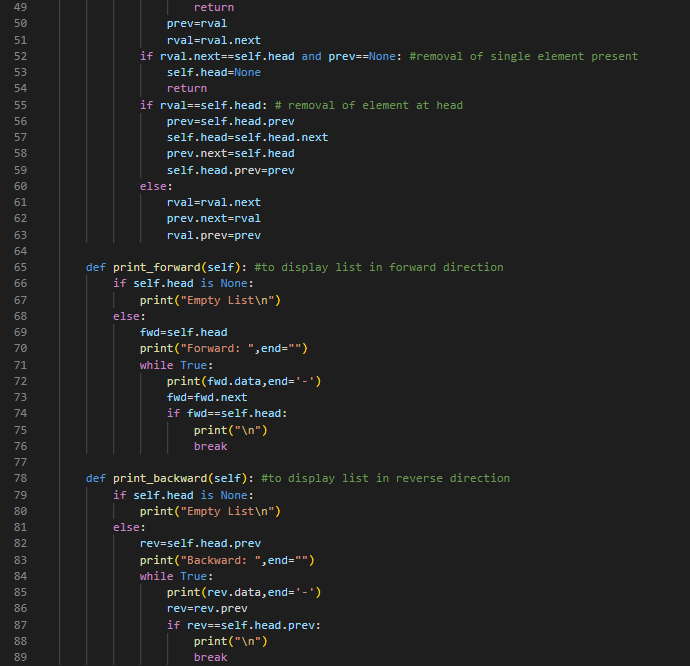
Step 9: Use a print statement to print the nodes in backward direction starting from the last node to the first node.

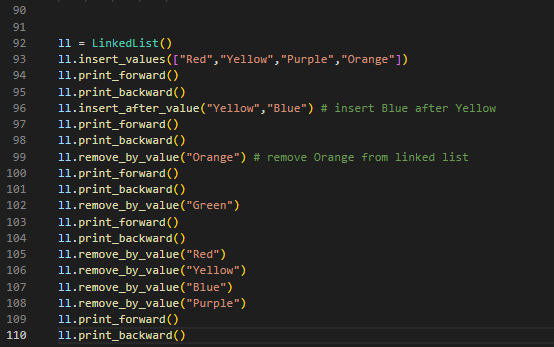
Step 10 : Display the linkedlist

1. Program

“Paste the screenshot of the executed python code”

**Program: **

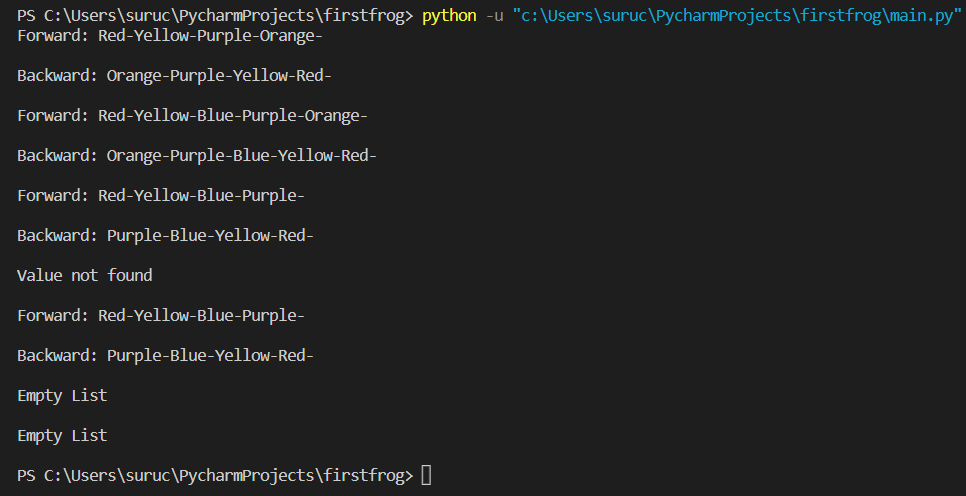
****

****

1. Presentation of the results

“Paste the output of the program”

**Output:**

****

1. Analysis and discussions

“Discuss the time complexities of all the operations of the given data structure”

The time complexity of operations on a linked list data structure can vary depending on the specific implementation and the algorithm used. However, in general, here are the time complexities of common operations on a singly linked list:

1. Accessing an element: O(n) Accessing a specific element in a linked list requires traversing the list from the head node to the desired node, which takes O(n) time where n is the number of elements in the list.
2. Searching for an element: O(n) In the worst case, searching for an element in a singly linked list requires checking every element, which takes O(n) time where n is the number of elements in the list.
3. Inserting an element at the beginning: O(1) Inserting an element at the beginning of a singly linked list is a constant time operation because it does not require shifting of elements.
4. Inserting an element at the end: O(n) Inserting an element at the end of a singly linked list requires traversing the entire list to reach the last node, which takes O(n) time where n is the number of elements in the list.
5. Inserting an element in the middle: O(n) Inserting an element in the middle of a singly linked list requires traversing the list to find the insertion point, which takes O(n) time where n is the number of elements in the list. However, if the insertion point is known (such as inserting after a specific node), the time complexity can be reduced to O(1).
6. Removing an element from the beginning: O(1) Removing an element from the beginning of a singly linked list is a constant time operation because it does not require shifting of elements.
7. Removing an element from the end: O(n) Removing an element from the end of a singly linked list requires traversing the entire list to reach the last node, which takes O(n) time where n is the number of elements in the list.
8. Removing an element from the middle: O(n) Removing an element from the middle of a singly linked list requires traversing the list to find the node to be removed, which takes O(n) time where n is the number of elements in the list.
9. Reversing a linked list: O(n) Reversing a singly linked list requires traversing the entire list once and updating the pointers of each node, which takes O(n) time where n is the number of elements in the list.
10. Concatenating two linked lists: O(n) Concatenating two singly linked lists requires traversing the first list to reach the end, and then updating the last node to point to the head of the second list, which takes O(n) time where n is the number of elements in the first list.

Top of Form

# Experiment 3

**Title of the Laboratory Exercise**: Stack

**1. Aim:**

To understand and implement the basic operations in stack using python.

**2. Objective:**

To execute the below operations in stack:

1. Push: Pushing (storing) an element on the stack.
2. Pop: Removing (accessing) an element from the stack.
3. Peek: get the top data element of the stack, without removing it.
4. Check if stack is full.
5. Check if stack is empty.

**3. Exercise:**

1. Write a function in python that can reverse a string (your full name) using stack data structure. Create a function called “reverse\_myname” which does this operation.

Follow the steps given below to reverse a string using stack:

1. Create an empty stack.
2. One by one push all characters of string to stack by calling a push().
3. One by one pop all characters from stack and put them back to string
4. by calling a pop().
5. Create a Python function named "isit\_balanced" that determines if the string's paranthesis are balanced or not. "{}',"()" or "[]" are examples of parantheses.

**4. Experimental Procedure**

1. Algorithm design

“Write the pseudocode of the main operations of the given data structure”

**algorithm1:**

Step 1. Create an empty stack.

Step 2. Take input of your full name.

Step 3. Initialize an empty string.

Step 4. Loop through each character in the input name:

a. push the character onto the stack.

Step 5. Loop through each character in the input name:

a. pop the top character off the stack.

b. append the popped character to the empty string created in step #3.

Step 6. Return the reversed string.

**algorithm2:**

Step 1. Define the function isit\_balanced with argument string.

Step 2. Create a empty stack list.

Step 3. Loop through the characters of the string.

Step 4. If character is an opening bracket, push it to the stack list.

Step 5. If character is a closing bracket, pop the top most element from stack and check if the opening bracket corresponding to the closing bracket of current char.

Step 6. If they are not corresponding or stack is empty, return False.

Step 7. Finally, check if stack is empty.

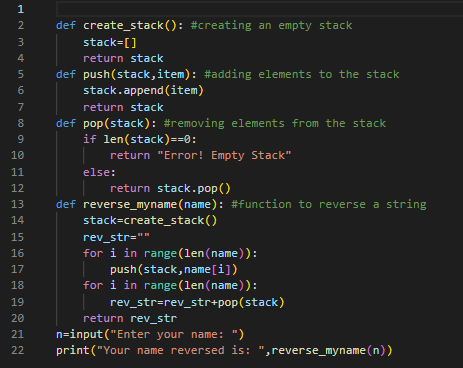
Step 8. If stack is empty, return True (as it means all the brackets have been matched and cancelled out).

Step 9. Otherwise, return False.

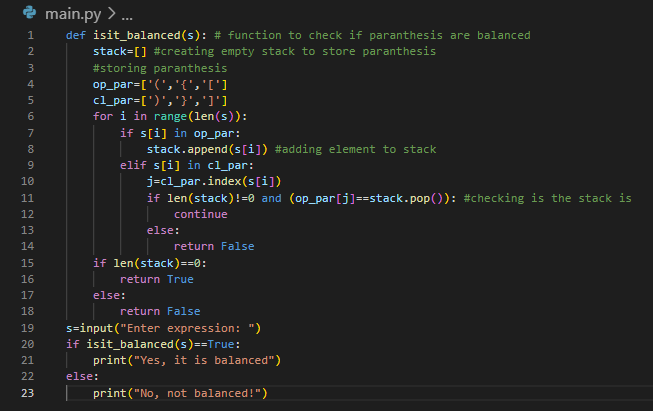
1. Program

“Paste the screenshot of the executed python code”

**Program-1:**

****

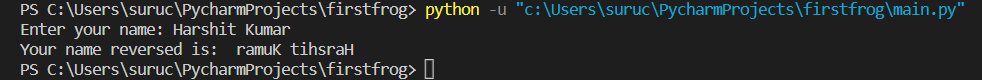
**Program-2:**

****

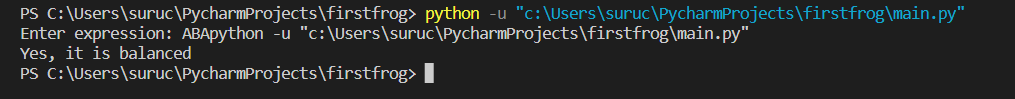
1. Presentation of the results

“Paste the output of the program”

**Output-1:**

****

**Output-2:**

****

1. Analysis and discussions

“Discuss the time complexities of all the operations of the given data structure”

The time complexity of stack operations depends on the implementation of the stack. In general, stack operations can be implemented using an array or a linked list.

If an array is used to implement the stack, the time complexity of the stack operations are as follows:

* Push operation: O(1) - This operation inserts an element at the top of the stack. Since the top of the stack is always at a fixed position, the push operation takes constant time.
* Pop operation: O(1) - This operation removes the element at the top of the stack. Since the top of the stack is always at a fixed position, the pop operation also takes constant time.
* Peek operation: O(1) - This operation returns the element at the top of the stack without removing it. Since the top of the stack is always at a fixed position, the peek operation also takes constant time.
* Search operation: O(n) - This operation searches for an element in the stack. It involves traversing the entire stack from top to bottom until the element is found. Therefore, the search operation takes O(n) time.

If a linked list is used to implement the stack, the time complexity of the stack operations are as follows:

* Push operation: O(1) - This operation inserts an element at the beginning of the linked list, which is the new top of the stack. Since the insertion is done at the beginning of the list, the push operation takes constant time.
* Pop operation: O(1) - This operation removes the element at the beginning of the linked list, which is the top of the stack. Since the removal is done at the beginning of the list, the pop operation also takes constant time.
* Peek operation: O(1) - This operation returns the element at the beginning of the linked list without removing it. Since the top of the stack is always the first element of the linked list, the peek operation also takes constant time.
* Search operation: O(n) - This operation searches for an element in the stack. It involves traversing the entire linked list from the beginning until the element is found. Therefore, the search operation takes O(n) time.

In summary, the push, pop, and peek operations of the stack take constant time, while the search operation takes linear time. The choice of implementation, array or linked list, can also affect the time complexity of the operations.

# Experiment 4

**Title of the Laboratory Exercise**: Queue

**1. Aim:**

To understand and implement the basic operations in deque using python.

**2. Objective:**

To execute the below operations in a full binary tree:

1. Insert an element at the front end of the deque.
2. Delete an element at the rear end of the deque.

**3. Exercise:**

Using the deque data structure, insert some elements at the front and delete an element at the rear end of the deque. The maximum size of the array is 6. Check the conditions of overflow and underflow before carrying out insertion and deletion, respectively.

**4. Experimental Procedure**

1. Algorithm design

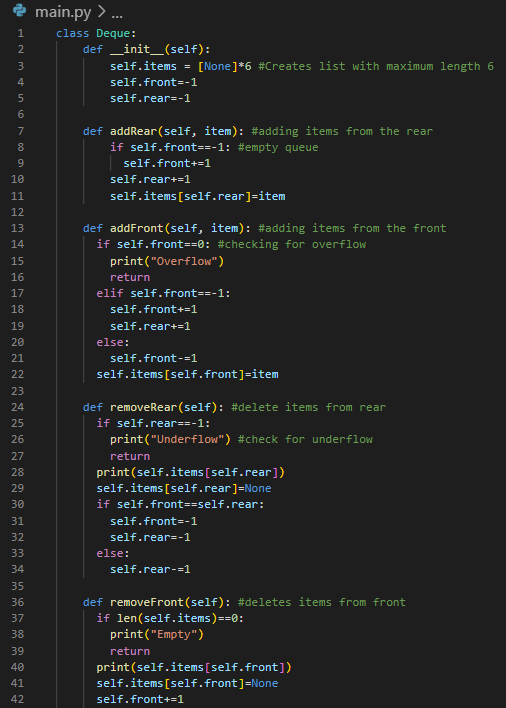
“Write the pseudocode of the main operations of the given data structure”

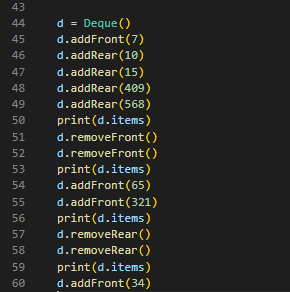
**algorithm:**

1. Initialize deque with empty array with size 6.
2. Define variable front and rear with initial values -1.
3. Define max size variable as 6.
4. Define Insert\_front procedure to insert an element at the front: a. Check if front is equal to 0 and size is greater than or equal to max size, print overflow error. b. If front is -1, set front and rear to 0 and insert element. c. If front is greater than 0, decrease front by 1 and insert element.
5. Define delete\_rear procedure to delete an element from the rear: a. Check if rear is -1, print underflow error. b. If rear is 0, delete element and set front and rear to -1. c. If rear is greater than 0, delete element and decrease rear by 1.
6. Define print\_deque procedure to print deque elements from front to rear: a. If front is -1, print empty deque message. b. If front is less than or equal to rear, loop through deque elements and print each element. c. If front is greater than rear, loop through deque elements from index 0 to rear and print each element, then loop through elements from index front to max size and print each element.
7. Call insert\_front and delete\_rear procedures to add and remove elements.
8. Call print\_deque to print deque elements.
9. End.
10. Program

“Paste the screenshot of the executed python code”

**Program:**

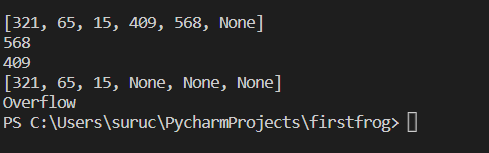
****

****

1. Presentation of the results

“Paste the output of the program”

**Output:**

****

1. Analysis and discussions

“Discuss the time complexities of all the operations of the given data structure”

The time complexity of queue operations depends on the implementation of the queue. In general, queue operations can be implemented using an array or a linked list.

If an array is used to implement the queue, the time complexity of the queue operations are as follows:

* Enqueue operation: O(1) - This operation inserts an element at the rear end of the queue. Since the rear end of the queue is always at a fixed position, the enqueue operation takes constant time.
* Dequeue operation: O(n) - This operation removes the element at the front end of the queue. Since the front end of the queue is not fixed and the elements need to be shifted, the dequeue operation takes O(n) time, where n is the size of the queue.
* Peek operation: O(1) - This operation returns the element at the front end of the queue without removing it. Since the front end of the queue is always at a fixed position, the peek operation takes constant time.
* Search operation: O(n) - This operation searches for an element in the queue. It involves traversing the entire queue from front to rear until the element is found. Therefore, the search operation takes O(n) time.

If a linked list is used to implement the queue, the time complexity of the queue operations are as follows:

* Enqueue operation: O(1) - This operation inserts an element at the end of the linked list, which is the rear end of the queue. Since the insertion is done at the end of the list, the enqueue operation takes constant time.
* Dequeue operation: O(1) - This operation removes the element at the beginning of the linked list, which is the front end of the queue. Since the removal is done at the beginning of the list, the dequeue operation also takes constant time.
* Peek operation: O(1) - This operation returns the element at the beginning of the linked list without removing it. Since the front end of the queue is always the first element of the linked list, the peek operation also takes constant time.
* Search operation: O(n) - This operation searches for an element in the queue. It involves traversing the entire linked list from the beginning until the element is found. Therefore, the search operation takes O(n) time.

In summary, the enqueue and peek operations of the queue take constant time, while the dequeue operation takes linear time in the case of an array implementation, and constant time in the case of a linked list implementation. The search operation takes linear time in both cases. The choice of implementation, array or linked list, can also affect the time complexity of the operations.

Top of Form

Bottom of Form

# Experiment 5

**Title of the Laboratory Exercise**: Binary Tree

**1. Aim:**

To understand and implement the basic operations in full binary tree using python.

**2. Objective:**

To execute the below operations in a full binary tree:

1. Search − Searches an element in a tree.
2. Insert − Inserts an element in a tree.
3. Pre-order Traversal − Traverses a tree in a pre-order manner.
4. In-order Traversal − Traverses a tree in an in-order manner.
5. Post-order Traversal − Traverses a tree in a post-order manner.

**3. Exercise:**

Construct a full binary tree with 10 nodes, where the data item inserted at every node should be a random value between 1 and 100. Add the following methods to the class named "FullBinaryTree" and perform the operation on the constructed full binary tree.

1. find\_min(): finds the minimum element stored in the constructed Full binary tree.
2. find\_max(): finds the maximum element stored in the constructed Full binary tree.
3. calculate\_sum(): calculates the sum of all elements stored in the constructed Full binary tree.
4. pre\_order\_traversal(): performs pre-order traversal of the constructed Full binary tree.
5. post\_order\_traversal(): performs post-order traversal of the constructed Full binary tree.
6. in\_order\_traversal(): performs in-order traversal of the constructed Full binary tree.

**4. Experimental Procedure**

1. Algorithm design

“Write the pseudocode of the main operations of the given data structure”

**algorithm:**

Step 1. Create a node

Step 2. assign values in node

Step 3. calaculate sum of all elements

Step 4. Perform post order traversal

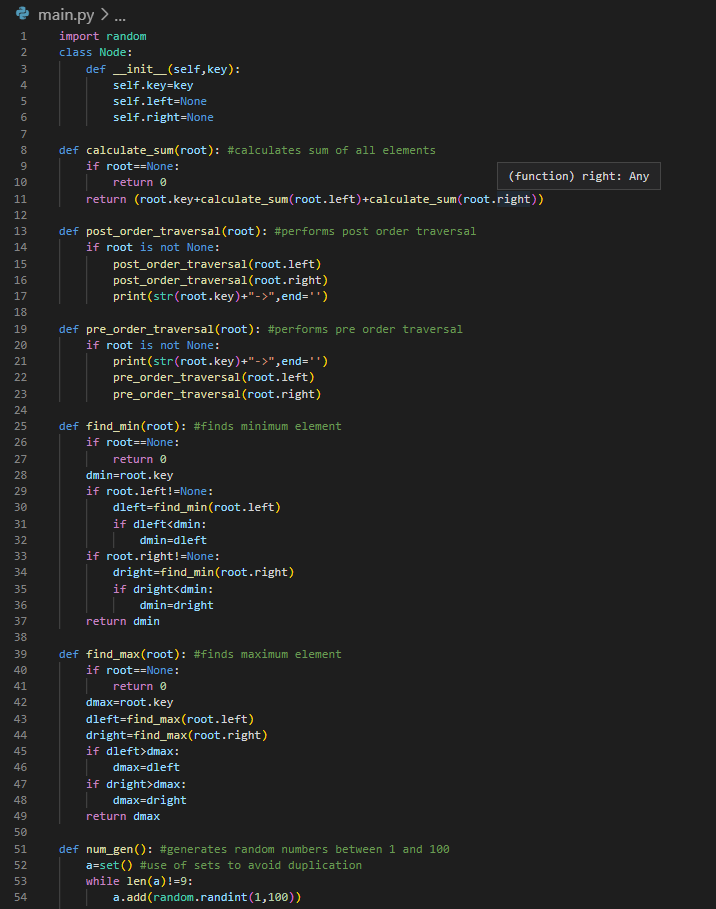
Step 5. Perform pre order traversal

Step 6. find minimum element and maximum element in binary tree

1. Program

“Paste the screenshot of the executed python code”

**Program:**

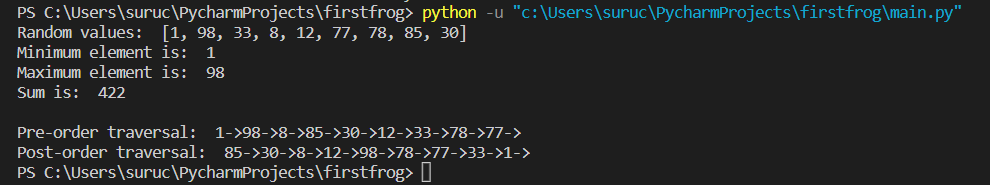
****

****

1. Presentation of the results

“Paste the output of the program”

**Output:**

****

1. Analysis and discussions

“Discuss the time complexities of all the operations of the given data structure”

The time complexity of the operations on a binary tree data structure depends on the specific implementation and the type of binary tree, such as a binary search tree, a heap, or a balanced tree. Here are the time complexities of some common operations on binary trees:

1. Insertion: a. Binary Search Tree: O(h), where h is the height of the tree. In the worst case, when the tree is unbalanced, the height can be O(n), where n is the number of nodes in the tree, so the worst-case complexity is O(n). b. Heap: O(log n), where n is the number of nodes in the tree. c. Balanced Tree (e.g. AVL tree, Red-Black tree): O(log n), where n is the number of nodes in the tree.
2. Deletion: a. Binary Search Tree: O(h), where h is the height of the tree. In the worst case, when the tree is unbalanced, the height can be O(n), where n is the number of nodes in the tree, so the worst-case complexity is O(n). b. Heap: O(log n), where n is the number of nodes in the tree. c. Balanced Tree (e.g. AVL tree, Red-Black tree): O(log n), where n is the number of nodes in the tree.
3. Searching: a. Binary Search Tree: O(h), where h is the height of the tree. In the worst case, when the tree is unbalanced, the height can be O(n), where n is the number of nodes in the tree, so the worst-case complexity is O(n). b. Heap: O(n), where n is the number of nodes in the tree. c. Balanced Tree (e.g. AVL tree, Red-Black tree): O(log n), where n is the number of nodes in the tree.
4. Traversal: a. In-order, Pre-order, Post-order Traversal: O(n), where n is the number of nodes in the tree. b. Level-order Traversal: O(n), where n is the number of nodes in the tree.
5. Finding Minimum/Maximum: a. Binary Search Tree: O(h), where h is the height of the tree. In the worst case, when the tree is unbalanced, the height can be O(n), where n is the number of nodes in the tree, so the worst-case complexity is O(n). b. Heap: O(1).

In summary, the time complexities of binary tree operations depend on the type of tree and the implementation. The worst-case time complexity for binary search tree operations can be O(n) in the case of an unbalanced tree, while the worst-case complexity for balanced trees is O(log n). The heap operations have a time complexity of O(log n), while the traversal operations have a time complexity of O(n). The time complexity of finding the minimum/maximum value depends on the type of tree; for a heap, it is O(1), while for a binary search tree, it is O(h), where h is the height of the tree.

# Experiment 6

**Title of the Laboratory Exercise**: Binary Search Tree

**1. Aim:**

To understand and implement the basic operations in Binary Search Tree using python.

**2. Objective:**

To execute the below operations in a Binary Search Tree (BST):

1. Search − Searches an element in a BST.
2. Insert − Inserts an element in a BST.
3. Delete − Deletes an element in a BST.
4. Check the balance of the BST.
5. Determine the height of the BST.

**3. Exercise:**

Construct a binary search tree with the below values: {12, 35, 14, 97, 36, 65, 89}. Write a python program to perform the following operations:

1. Insert a new element which is having a value equivalent to the “last two digits of your roll number”.
2. To determine the height of the constructed BST.
3. Delete any element from the constructed BST.
4. To check if the constructed BST is Balanced or not.

**4. Experimental Procedure**

1. Algorithm design

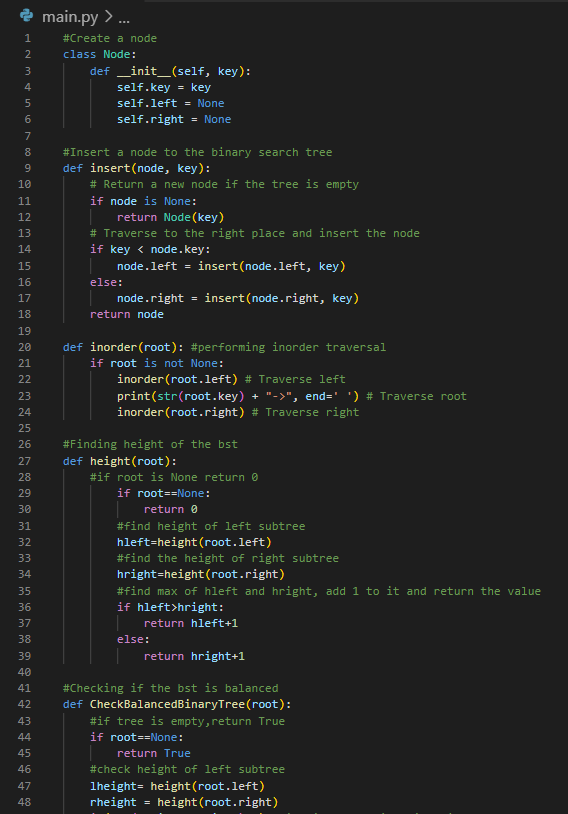
“Write the pseudocode of the main operations of the given data structure”

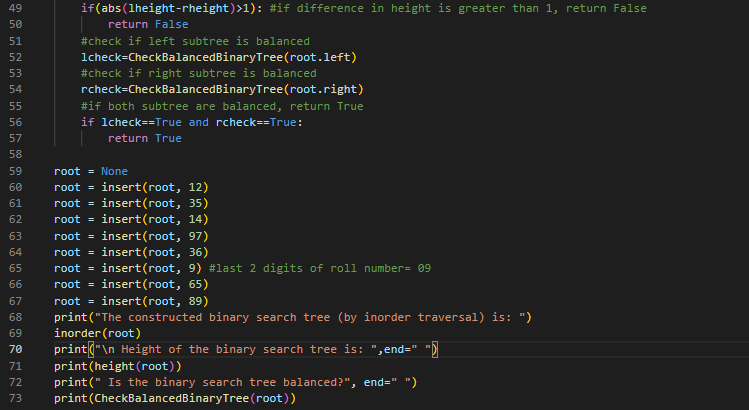
**algorithm:**

1. Create a BinarySearchTree class
2. Create a Node class with attributes for value, left child and right child
3. Define an init method for the BinarySearchTree class that initializes an empty root node
4. Define a method called insert that takes a value and recursively searches the tree until it finds a node with no children, and inserts the new node in the appropriate place
5. Define a method called find\_height that recursively traverses the tree and returns the maximum depth of the tree
6. Define a method called delete that takes a value to be deleted, and recursively searches the tree until it finds the node containing that value. If the node has no children, simply remove it. If the node has one child, replace the node with its child. If the node has two children, find the minimum value in the right subtree, replace the node with that value, and then delete the duplicate node
7. Define a method called is\_balanced that recursively checks the heights of the left and right subtrees, and returns True if the maximum height difference is less than or equal to 1; otherwise, returns False
8. Program

“Paste the screenshot of the executed python code”

**Program:**

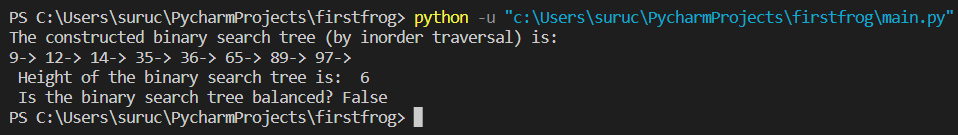
****

****

1. Presentation of the results

“Paste the output of the program”

**Output:**

****

1. Analysis and discussions

“Discuss the time complexities of all the operations of the given data structure”

The time complexities of operations on a binary search tree (BST) data structure depend on the specific implementation and the properties of the tree. Here are the time complexities of some common operations on BST:

1. Insertion: The time complexity of inserting a new node in a BST is O(h), where h is the height of the tree. In the worst case, when the tree is unbalanced, the height can be O(n), where n is the number of nodes in the tree, so the worst-case complexity is O(n).
2. Deletion: The time complexity of deleting a node in a BST is also O(h), where h is the height of the tree. In the worst case, when the tree is unbalanced, the height can be O(n), where n is the number of nodes in the tree, so the worst-case complexity is O(n).
3. Searching: The time complexity of searching for a key in a BST is O(h), where h is the height of the tree. In the worst case, when the tree is unbalanced, the height can be O(n), where n is the number of nodes in the tree, so the worst-case complexity is O(n).
4. Traversal: There are three types of traversal operations on BST: In-order, Pre-order, and Post-order. The time complexity of each of these traversal operations is O(n), where n is the number of nodes in the tree.
5. Finding Minimum/Maximum: The time complexity of finding the minimum/maximum value in a BST is O(h), where h is the height of the tree. In the worst case, when the tree is unbalanced, the height can be O(n), where n is the number of nodes in the tree, so the worst-case complexity is O(n).
6. Finding Successor/Predecessor: The time complexity of finding the successor or predecessor of a node in a BST is also O(h), where h is the height of the tree. In the worst case, when the tree is unbalanced, the height can be O(n), where n is the number of nodes in the tree, so the worst-case complexity is O(n).

In summary, the time complexities of operations on a BST are proportional to the height of the tree. If the tree is balanced, the height is O(log n), where n is the number of nodes in the tree. However, if the tree is unbalanced, the height can be as high as O(n), resulting in worst-case time complexity of O(n) for some operations.

# Experiment 7

**Title of the Laboratory Exercise**: Heap

**1. Aim:**

To understand and implement the basic operations in Heap using python.

**2. Objective:**

To execute the below operations in a Heap:

https://medium.com/techie-delight/heap-practice-problems-and-interview-questions-b678ff3b694c**3. Exercise:**

Implement a Python program that constructs an AVL tree having the following elements: Z, I, J, F, A, E, C, P, B, D, H, N. Consider the order of the elements in ascending order. Explain the rotations diagrammatically.

**4. Experimental Procedure**

1. Algorithm design

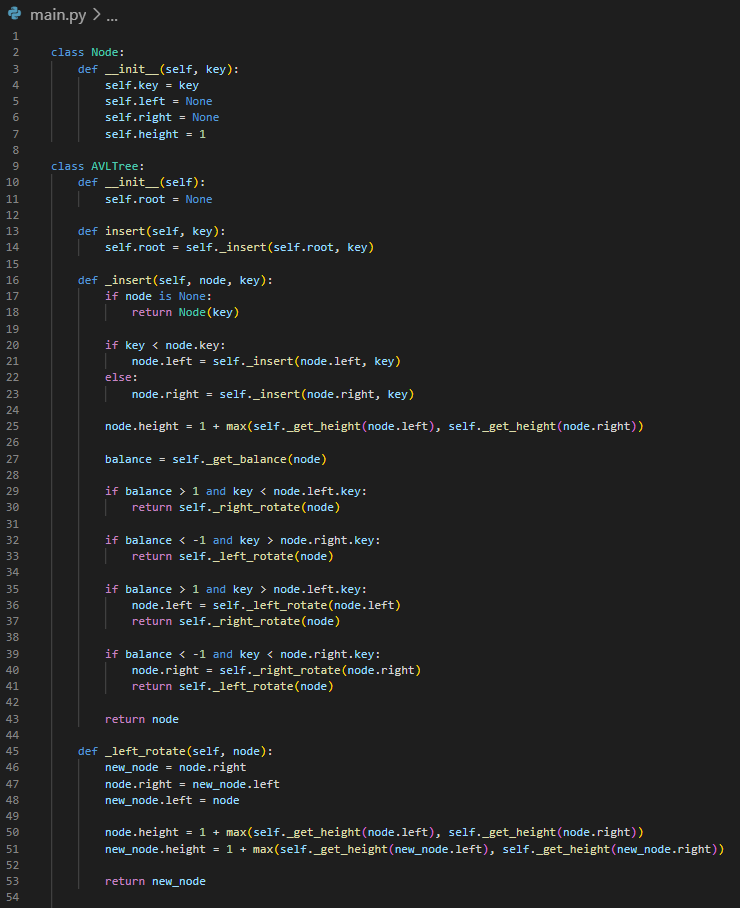
“Write the pseudocode of the main operations of the given data structure”

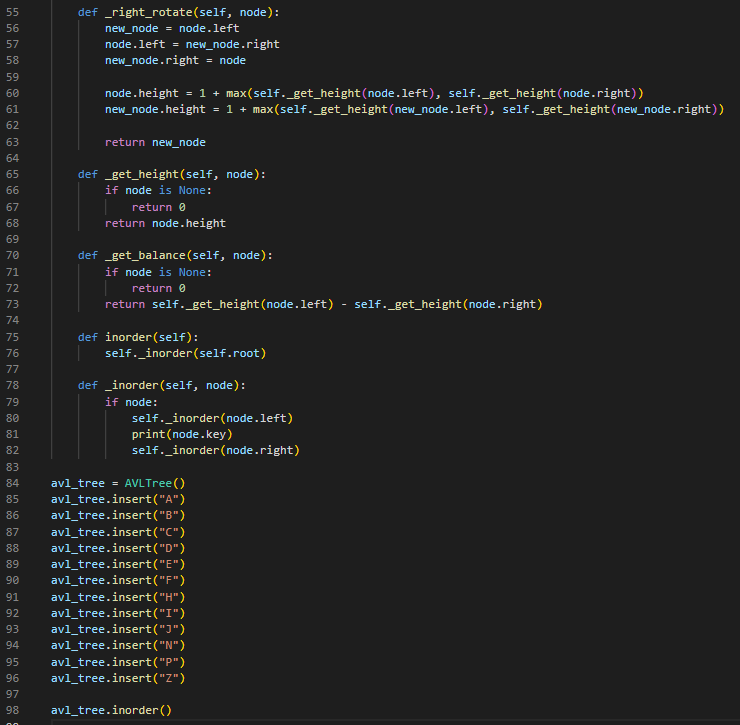
**algorithm:**

1. Create an empty heap tree
2. Insert Z into the heap
3. Insert I into the heap and perform necessary rotations
4. Insert J into the heap and perform necessary rotations
5. Insert F into the heap and perform necessary rotations
6. Insert A into the heap and perform necessary rotations
7. Insert E into the heap and perform necessary rotations
8. Insert C into the heap and perform necessary rotations
9. Insert P into the heap and perform necessary rotations
10. Insert B into the heap and perform necessary rotations
11. Insert D into the heap and perform necessary rotations
12. Insert H into the heap and perform necessary rotations
13. Insert N into the heap and perform necessary rotations
14. Print the final heap tree
15. Program

“Paste the screenshot of the executed python code”

**Program:**

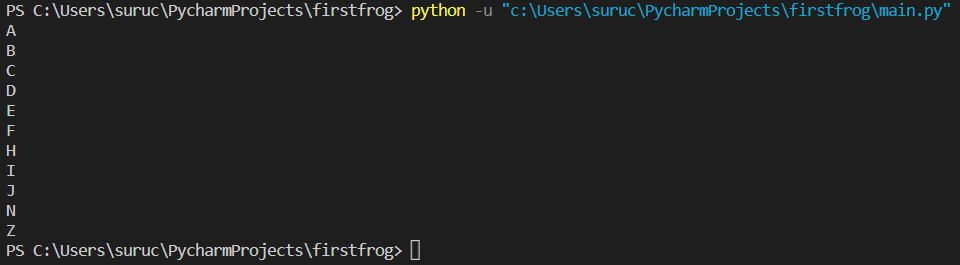
****

****

1. Presentation of the results

“Paste the output of the program”

**Output:**

****

1. Analysis and discussions

“Discuss the time complexities of all the operations of the given data structure”

The time complexities of operations on a heap data structure depend on the specific implementation and the properties of the heap. Here are the time complexities of some common operations on a binary heap:

1. Insertion: The time complexity of inserting a new element in a binary heap is O(log n), where n is the number of elements in the heap. The complexity arises due to the re-balancing process that takes place after inserting a new element.
2. Deletion: The time complexity of deleting an element from a binary heap is also O(log n), where n is the number of elements in the heap. The complexity arises due to the re-balancing process that takes place after deleting an element.
3. Finding the minimum/maximum: The time complexity of finding the minimum/maximum element in a binary heap is O(1) as it is stored at the root of the heap.
4. Heapify: The time complexity of converting an array of n elements into a binary heap is O(n) using the bottom-up heap construction algorithm.
5. Extracting minimum/maximum: The time complexity of extracting the minimum/maximum element from a binary heap is O(log n), where n is the number of elements in the heap. The complexity arises due to the re-balancing process that takes place after extracting the minimum/maximum element.
6. Merging: The time complexity of merging two binary heaps of size n1 and n2 is O(n1+n2).

In summary, the time complexities of operations on a binary heap are proportional to the height of the heap, which is O(log n). The time complexity of inserting, deleting, and extracting the minimum/maximum element from a binary heap is O(log n). The time complexity of finding the minimum/maximum element in a binary heap is O(1).

# Experiment 8

**Title of the Laboratory Exercise**: AVL Tree

**1. Aim:**

To understand and implement the basic operations in AVL using python.

**2. Objective:**

To execute the below operations in an AVL Tree:

1. Left rotation
2. Right rotation
3. Left-Right rotation
4. Right-Left rotation

**3. Exercise:**

Implement a Python program that constructs an AVL tree having the following elements: Z, I, J, F, A, E, C, P, B, D, H, N. Consider the order of the elements in ascending order. Explain the rotations diagrammatically.

**4. Experimental Procedure**

1. Algorithm design

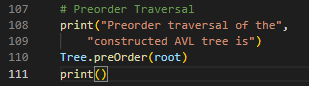
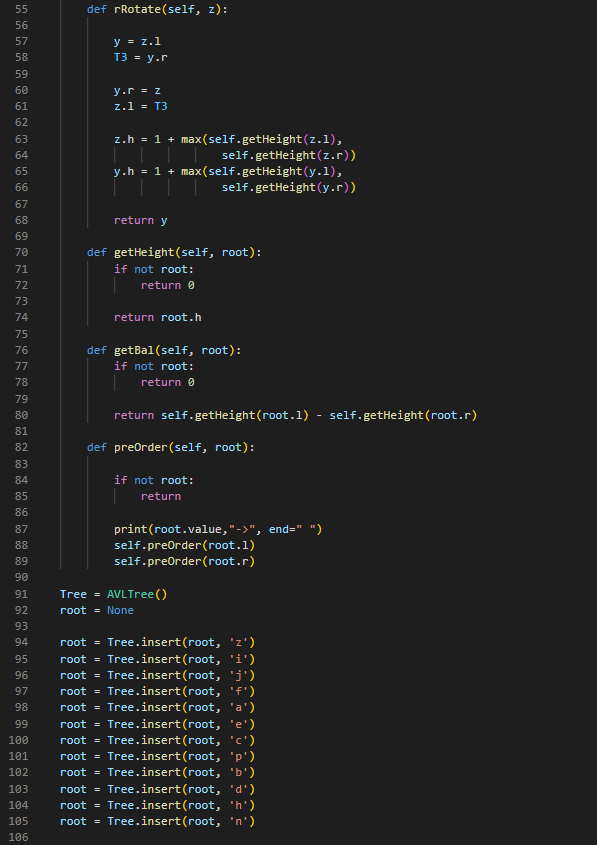
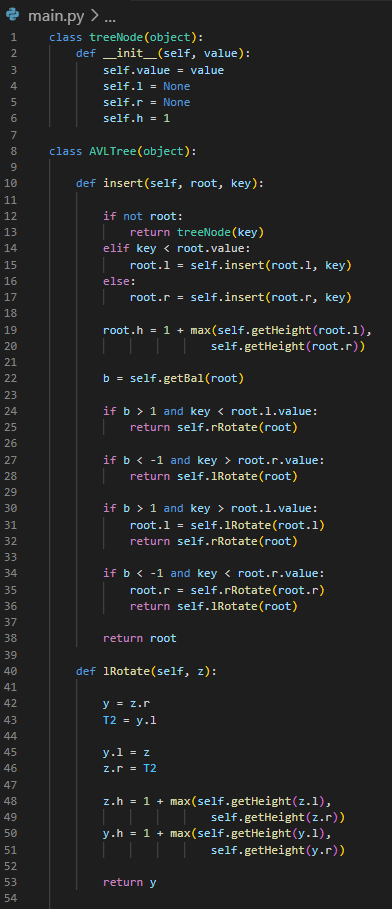
“Write the pseudocode of the main operations of the given data structure”

**algorithm:**

1. Create a class node with attributes value, left\_child, right\_child, and height.
2. Create a class AVLTree with methods add, balance\_factor, rotate\_left, rotate\_right, left\_right\_rotation, right\_left\_rotation, insert\_node, delete\_node, search.
3. Implement the add method to insert nodes into the AVL Tree.
4. Implement the balance\_factor method to calculate the balance factor of the node.
5. Implement the rotate\_left method to perform a left rotation on the node.
6. Implement the rotate\_right method to perform a right rotation on the node.
7. Implement the left\_right\_rotation method to perform a left-right rotation on the node.
8. Implement the right\_left\_rotation method to perform a right-left rotation on the node.
9. Implement the insert\_node method to insert the node into the AVL tree and balance it if necessary.
10. Implement the delete\_node method to remove a node from the AVL tree and balance it if necessary.
11. Implement the search method to search for a node in the AVL tree.
12. Program

“Paste the screenshot of the executed python code”

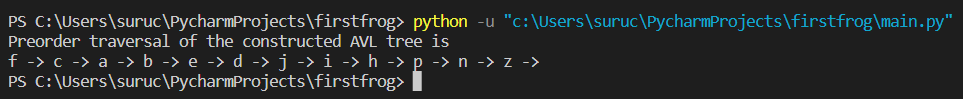
**Program:**

****

1. Presentation of the results

“Paste the output of the program”

**Output:**

****

1. Analysis and discussions

“Discuss the time complexities of all the operations of the given data structure”

An AVL tree is a self-balancing binary search tree, in which the heights of the left and right subtrees of any node differ by at most one. The time complexities of the basic operations on an AVL tree are as follows:

1. Search: The time complexity of searching for a value in an AVL tree is O(log n), where n is the number of nodes in the tree. This is because, like in a binary search tree, we can eliminate half of the remaining nodes at each level.
2. Insertion: The time complexity of inserting a new node in an AVL tree is O(log n), where n is the number of nodes in the tree. The insertion operation in an AVL tree involves two main steps: first, we perform a standard BST insertion to add the new node, and then we re-balance the tree by performing rotations, if necessary. Since a rotation can affect the height of the entire subtree, the re-balancing step takes at most O(log n) time.
3. Deletion: The time complexity of deleting a node from an AVL tree is also O(log n), where n is the number of nodes in the tree. Like in insertion, we first perform a standard BST deletion, and then we re-balance the tree by performing rotations, if necessary.
4. Balancing: The time complexity of balancing an AVL tree is O(1) per node. Whenever we add or remove a node from an AVL tree, we need to check whether the tree is still balanced and perform rotations if necessary. The balancing operation only affects the nodes along the path from the inserted/deleted node to the root, and hence its time complexity is proportional to the height of the tree, which is O(log n).

In summary, the time complexities of the basic operations on an AVL tree are O(log n), where n is the number of nodes in the tree. The AVL tree provides a worst-case guarantee of O(log n) time complexity for all basic operations, ensuring efficient performance even in the worst-case scenarios.

# Experiment 9

**Title of the Laboratory Exercise**: Quick Sort

**1. Aim:**

To implement Quick Sort Algorithm using Python

**2. Objective:**

1. To understand the concept of Quick Sort Algorithm
2. To learn how to implement Quick Sort Algorithm using Python
3. To analyze the time complexity of Quick Sort Algorithm

**3. Exercise:**

In this exercise, you will implement Quick Sort Algorithm using Python. Follow the steps below:

**Step 1:** Write a function called quick\_sort that takes an array of integers as input and returns a sorted array.

**Step 2:** Implement the Quick Sort Algorithm. The steps of the Quick Sort Algorithm are as follows:

i. Choose a pivot element from the array (can be the first or last element).

ii. Partition the array into two subarrays: one with elements less than or equal to the pivot, and one with elements greater than the pivot.

iii. Recursively sort the two subarrays.

**Step 3:** Test your implementation using a test case that includes a list of 10 unsorted integers.

**Step 4:** Analyze the time complexity of Quick Sort Algorithm.

**Step 5:** Submit your code along with a brief explanation of the Quick Sort Algorithm and its time complexity analysis.

Note: You can use the time module in Python to measure the time taken by your quick\_sort function to sort an array.

**4. Experimental Procedure**

1. Algorithm design

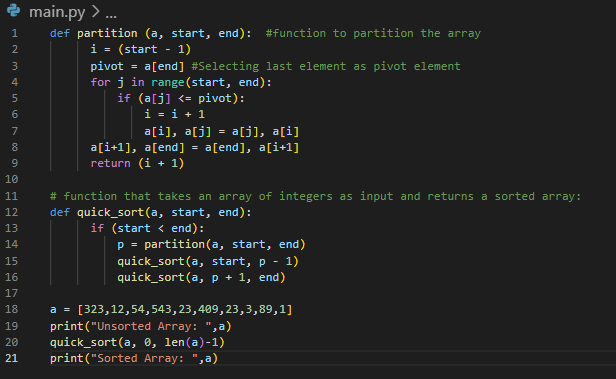
“Write the pseudocode of the main operations of the given sorting technique”

**algorithm:**

1. Define a function called quick\_sort that takes an array of integers as input.
2. Implement the Quick Sort Algorithm
3. Return the sorted array
4. Program

“Paste the screenshot of the executed python code”

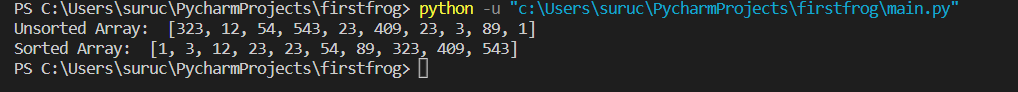
**Program:**

****

1. Presentation of the results

“Paste the output of the program”

**Output:**

****

1. Analysis and discussions

“Discuss the time complexities of all the operations of the given sorting technique”

Quick sort is a divide-and-conquer sorting algorithm that works by partitioning an array into two sub-arrays, sorting those sub-arrays recursively, and then combining them to produce a sorted array. The time complexities of the basic operations on quick sort are as follows:

1. Partition: The partitioning step in quick sort takes O(n) time, where n is the number of elements in the array. This step involves selecting a pivot element and partitioning the array into two sub-arrays, such that all elements to the left of the pivot are smaller, and all elements to the right of the pivot are larger.
2. Sorting: The sorting step in quick sort involves recursively sorting the left and right sub-arrays. The time complexity of this step depends on the size of the sub-arrays and the number of recursive calls made. In the best case, when the pivot divides the array into two equal halves, the time complexity is O(n log n). In the worst case, when the pivot is always the smallest or largest element, the time complexity is O(n^2). On average, the time complexity is O(n log n).
3. Choosing the pivot: The choice of pivot can greatly affect the time complexity of quick sort. In the worst case, when the pivot is always the smallest or largest element, the time complexity is O(n^2). To avoid this, various pivot selection strategies can be used, such as selecting the median element of the sub-array, selecting a random element, or using the "median-of-three" strategy.

In summary, the time complexity of quick sort depends on the size of the array and the choice of pivot. The partitioning step takes O(n) time, and the sorting step takes O(n log n) time on average, but can take up to O(n^2) time in the worst case. However, with appropriate pivot selection strategies, quick sort can achieve an average time complexity of O(n log n), making it one of the most efficient sorting algorithms in practice.

# Experiment 10

**Title of the Laboratory Exercise**: Merge Sort

**1. Aim:**

To implement Merge Sort Algorithm using Python

**2. Objective:**

1. To understand the concept of Merge Sort Algorithm
2. To learn how to implement Merge Sort Algorithm using Python
3. To analyze the time complexity of Merge Sort Algorithm

**3. Exercise:**

In this exercise, you are required to implement the merge sort algorithm using Python. Follow the instructions below:

**Step 1:** Define a function called "merge\_sort" that takes in a list of numbers as input and returns a sorted list using the merge sort algorithm.

**Step 2:** Implement the "merge" function, which will be used in the merge sort algorithm. This function takes two sorted sub-lists and merges them into a single sorted list. The merge function should return the merged list.

**Step 3:** Implement the "merge\_sort" function using recursion. The function should divide the input list into two halves, sort each half recursively, and then merge the two sorted halves using the "merge" function.

**Step 4:** Test your implementation using a test case that includes a list of 10 unsorted integers.

**Step 5:** Analyze the time complexity of your implementation.

**4. Experimental Procedure**

1. Algorithm design

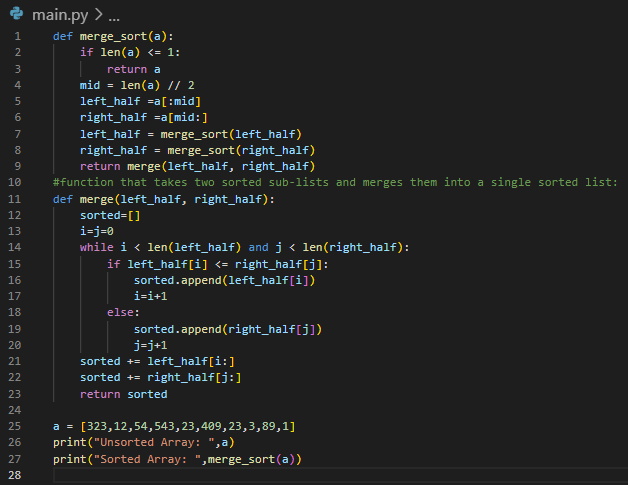
“Write the pseudocode of the main operations of the given sorting technique”

**algorithm:**

1. Define the function merge\_sort with a parameter "lst" for the list of numbers
2. Check if the length of the list is greater than 1 or not, if it is not greater than 1, return lst as it is.
3. Find the middle index of the list
4. Divide the list into two halves from the middle index, left\_half and right\_half.
5. Return the merged list by calling the merge function and passing the recursive call of merge\_sort function for left\_half and right\_half inputs.
6. Program

“Paste the screenshot of the executed python code”

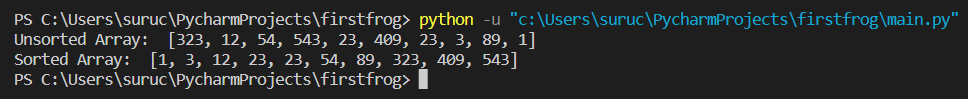
**Program:**

****

1. Presentation of the results

“Paste the output of the program”

**Output:**

****

1. Analysis and discussions

“Discuss the time complexities of all the operations of the given sorting technique”

Merge sort is a divide-and-conquer sorting algorithm that works by dividing the input array into two halves, sorting each half recursively, and then merging the two sorted halves into a single sorted array. The time complexities of the basic operations on merge sort are as follows:

1. Divide: The divide step in merge sort involves splitting the input array into two halves. This operation takes O(1) time.
2. Merge: The merge step in merge sort involves merging two sorted sub-arrays into a single sorted array. This operation takes O(n) time, where n is the total number of elements in the sub-arrays being merged.
3. Sorting: The sorting step in merge sort involves recursively sorting the left and right sub-arrays. The time complexity of this step depends on the size of the sub-arrays and the number of recursive calls made. In the worst case, the time complexity is O(n log n), where n is the number of elements in the input array.

In summary, the time complexity of merge sort is O(n log n), where n is the number of elements in the input array. This time complexity is guaranteed in all cases, regardless of the distribution of the input data. However, merge sort requires additional memory to store the sub-arrays during the sorting process, which can be a disadvantage in some situations.