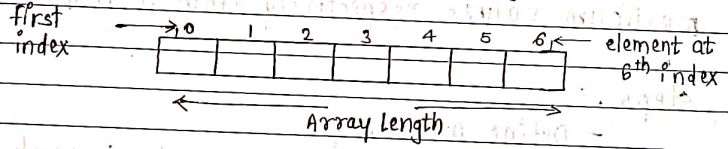
Experiment no. 01

Aim : Problems on Arrays.

Theory: Arrays in Python, are often implemented as lists, are used to store multiple elements of the same type.

Basic operations on array include finding sums, maximums, minimums and sorting.

These tasks are commonly performed using python's inbuilt functions like sum(), min(), max(), sorted(). Understanding these operations are essential for efficient data manipulations in python.



Problems on array

1) sum of array

steps :

- define an array (or list) in python.
- initialize a variable to zero.
- Loop through array & add each element to initialized variable.
- Print the final sum.

Alternatively, python provides a inbuilt() function directly return the sum of elements.

2) Maximum and minimum value from an Array:-

Here, we can use two approaches :-

a) manual approach : Initialise two variables minn and maxx with the first element of the array, then iterate through the array while comparing each element with these variables, update them whenever a larger and a smaller value is found.

b) Built-in functions : Python offers the min() & max() functions, which returns the minimum & maximum values respectively from a list.

Steps :

- Define an Array.
- Use a loop to compare each element or use the max() & min() functions to get the values.
- print minimum and maximum values.

3) Sort the elements of an array.

Python provides several approaches to sort an array-

a) manual sorting : Implement a sorting algorithm that rearranges the elements step by step.

b) Built-in method : Python's sort() method sorts the list in ascending order while the sorted() function returns a new sorted list.

Steps:

- Define an array.
- Use sorting algorithm as python's sort method.
- print the sorted array.

Conclusion: These programs helps in practicing basic operations with arrays & familiarize you with loops, conditions and built-in functions in python.

Experiment no. 02

Aim : Problems on string

Theory :

Strings

— Strings in python are surrounded by either single quotation or double quotation marks.

— We can display a string literal with the print() function as :- print("Hello") or print('Hello')

Assigning string to a variable

— Assigning a string to a variable is done with the variable name followed by an equal sign & the string as :—

eg. a = "Hello"
print(a)

Multiline strings

— You can assign a multiline string to a variable by using three quotes :—

a = """ Lorem ipsum
consectetur elit
aliqua."""
print(a)

Indexing in strings

— Square brackets can be used to access elements of a string as :- a = "Hello World"
print(a[1]) ⇒ Output : e.

Looping through strings

— since strings are similar to arrays, we can loop through the characters within it using a 'for' loop.

eg. for 'x' in 'banana':
    print(x)

output: b
a
n
a
n
a

String Concatenation

— strings can be concatenated i.e combined using a '+' operator. eg. print("hello" + "world")

String modification

— Python provides a several set of functions for modifications of strings.

- upper() : converts entire string to upper case.
- lower() : converts entire string to lowercase.
- capitalize() : capitalizes the first letter of string.
- title() : capitalizes the first letter of every word in a string.

- strip() : Removes leading and trailing whitespace also can remove specific characters.

- split(seperator) : splits the string into a list using a seperator.

- join(iterable) : joins the elements of an iterable list into a single string, seperated by a given string.

- len() : We make use of a len() function to get the length of a string.

Conclusion : These programs show functions or the operations on strings in python, following the best practices for handling string datatypes effectively.

Experiment No :- 3

Aim : To implement sorting methods — Bubble Sort and Insertion Sort
To implement linear search and binary search.

Theory : Sorting algorithms are essential in computer science for ~~ascending~~ arranging data in a specific order, usually in ascending & descending data order. Different sorting methods vary in efficiency, use case and complexity.

(1) Bubble Sort
It is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. This process repeats until the list is sorted.

Time complexity :-
Best case : $O(n)$. when the list is already sorted.
Worst case : $O(n^2)$. when the list is in reverse order.

How it works ?
- starts from the first element and compare it with the next one.
- If the first is greater than the second, swap them.
- Move to the next pair and repeat.
- Repeat the entire process until no swaps are needed.

(2) Insertion sort

Insertion sort works by building a sorted list, one element at a time. It removes an element from the unsorted part, compares it with element in the sorted part & places it in the correct position.

Time complexity

Best case :- O(n) when the list is already sorted.

Worst case :- $O(n^2)$

How it works?

- start with the second element and compares it to the first.
- Insert it in the correct position.
- Move to the next element & repeat until the whole list is sorted.

(3) Linear search

Linear search is the simplest search algorithm. It checks each element in the list sequentially until the desired element is found or the end of the list is reached. Linear search can be applied to both sorted & unsorted data. Each element is compared one by one.

Time complexity

- Best case - O(1): If the desired element is the first element.
- Worst case - O(n). If the desired element is the last element or not in the list.

(4) Binary search

Binary search is a more efficient algorithm for searching in a sorted list. It uses a divide and conquer approach by repeatedly dividing the list in half & comparing the middle element to the target value. The list must be sorted.

Time complexity

Best case - O(1) - If the middle element is target.

Worst case - O(log n) as the search space is halved with each comparison.

How it works?

- Compares the middle element of list to the target.
- If the middle element is equal to the target, returns its index.
- If the middle element is less than the target, repeat the process on the right half of the list.
- Continue until the target is found or the search space is executed.

Conclusion:

In this practical, we successfully implemented various sorting algorithms and search techniques (linear & Binary search).

Done

Experiment no. 4

Aim : To implement programs on matrix.

Theory :

(1) Matrix Addition

— Two matrices can be added if and only if they have the same dimensions (same no. of rows & columns)

$$c[i][j] = A[i][j] + B[i][j]$$

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 6 & 6 \\ 7 & 8 \end{bmatrix} \Rightarrow C = \begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

(2) Matrix Multiplication:

— If A is m x n matrix and B is n x p matrix, their product will be on m x p matrix.

$$c[i][j] = \sum_{i=1}^{n} A[i][k] \cdot B[k][j]$$

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 \cdot 5 + 2 \cdot 7 & 1 \cdot 6 + 2 \cdot 8 \\ 3 \cdot 5 + 4 \cdot 7 & 3 \cdot 6 + 4 \cdot 8 \end{bmatrix}$$

$$= \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

(3) Matrix Determinant

The determinant helps in determining whether a matrix is invertible (non-zero determinant) or singular (non-determinant)

$A = \begin{bmatrix} a & c \\ b & d \end{bmatrix} \rightarrow \det(A) = ad - bc$

eg.

$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \rightarrow \det(A) = (1 \cdot 4) - (2 \cdot 3)$

$= 4 - 6$

$= -2$

(4) matrix Transpose

Transpose of a matrix formed by swapping its rows & columns. If A is an m x n matrix, its transpose A^T will be n x m matrix.

$A^T[i][j] = A[j][i]$

$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$, $A^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$

Conclusion :-

In this experiment, we implemented various matrix operations such as addition, multiplication, find the determinant & calculating the transpose.

Experiment no. 5

Aim : To implement the linked list and also perform all the following operations :-

(i) Insertion at end
(ii) Deletion at beginning
(iii) Deletion at end
(iv) Insertion at beginning
(v) print
(vi) search
(vii) exit

Theory :

(1) Linked List

A linked list is a linear data structure in which the elements, called nodes are not stored in contiguous memory locations instead, each node contains two parts.

- Data : The actual value or data, the node holds
- Pointer*(next) : A reference to the next node in the sequence, unlike arrays linked list allows for dynamic memory allocation.
- Insertion at beginning : A new node is created & inserted at the start of linked list.
- Insertion at End : A new node is created & inserted at the end of the linked list.
- Deletion at beginning : The head node is removed & the pointer is updated to the second node in the list.

• Deletion at end: The last node in the list is removed and the second last last node's pointer is updated to null.

• print list:

The linked list is finally traversed from head node to the last node, while printing the value of each other.

• search:

The linkedlist is traversed to find a particular value; if the value is found, its position from the list is returned.

Time complexity

— Insertion at the end of a singly linked list takes o(n) time.

— space complexity: Insertion at the end requires o(1) space, as we are adding single new node.

— Adding elements into linked list has time complexity of o(1), best case.

— Worst case time complexity is o(n) when we have to iterate over all the items within the linked list.

Conclusion:

The implementation of linked list & the basic operations, provided a strong understanding of dynamic data structures.

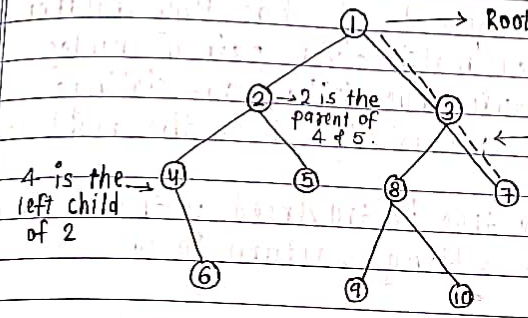
Experiment no. 06

Aim : Problems on Binary Tree.

Theory :

Binary Tree

- Binary Tree is a non-linear data structure where each node at most two children.



6, 5, 9, 10 are leaf nodes.

The topmost node is a root and the bottom most nodes are called leaves. A binary tree can be visualized as a hierarchical structure with the root at the top and leaves at the bottom.

Algorithm :-

Checking whether the tree given is a binary search tree.

1. Start at the root.
2. Check each node.

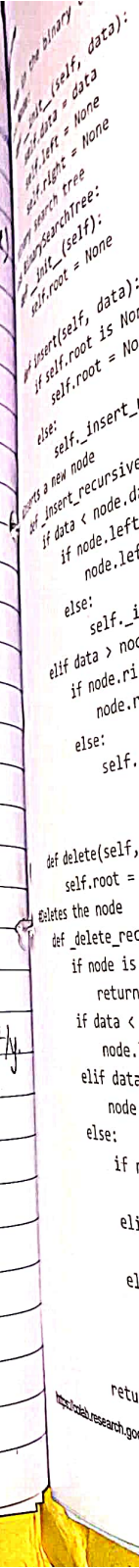
note :- left child node must be smaller.
3. Recursively verify (maximum bound at left subtree)
4. Base Case : An empty subtree is valid.
5. If all the nodes, satisfy the above condition, then the tree is a BST i.e Binary search tree.

Algorithm : Checking whether two nodes are siblings.

1. start from the root of tree.
2. check the current node, it has two children.
   — Check if they match given pair of nodes.
3. If found, return true (if they are siblings).
4. If not found, recursively check the right and left subtree.
5. If the whole tree is traversed without finding the nodes as siblings, return false.

Conclusion :

To verify a binary tree as BST, ensure that for each node the left child is smaller. This property allows searching for the node efficiently. To check sibling nodes, identify if two nodes share the same parent, useful in family tree structures or hierarchical data.



Experiment no: 07

Aim: Problems on Binary Search Tree

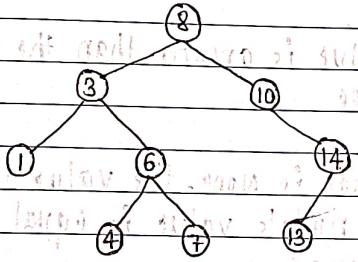
Theory:

Binary Search Tree

— Binary Search Tree is a data structure used for the organisation and storage of data in a sorted manner.

— Each node in Binary search Tree has at most two children, a left child and a right child, with the left child containing values less than parent node and the right child containing values greater than the parent node.

— This hierarchical structure allows for efficient searching, sorting, insertion and deletion operations on the data stored in the tree.



* BINARY SEARCH TREE *

Algorithm: Checking if a Binary Tree is a Binary Search Tree.

(i) In-order traversal

— Recursively, traverse the tree in-order.

- In-order traversal visits nodes in ascending order for BST.
- Keep track of previously visited node.

(ii) Check BST property
- For each node, compare its value with previous node's value.
  If current's node value is less than or equal to the previous node's value, the tree is not a BST.

(iii) Recursive Calls.
- Recursively check the left & right subtrees.

Algorithm : searching for a value in BST.

(i) Compare with Root
- Compare target value with root node's value.

(ii) Recursive search
- If target value is less than the root's value, search the left subtree.
- If target value is greater than the root's value, search the right subtree.

(iii) Base Case
- If the root node is None, the value is not found.
- If the root node's value is equal to target value, the value is found.

Conclusion :
By understanding these algorithms of Binary search Tree and their implementations, we can effectively work with binary search trees.

Experiment no: 08

Aim: Problems with stacks and queues.

Theory:

(i) Stack

— A stack is a linear data structure that follows the Last In, First Out (LIFO) principle. This means that the last element added to the stack is the first one to be removed.

Operations on stacks:

— operations associated with stack are —

Push: Adds an element to the top of the stack.

Pop: Removes & returns the top element of the stack.

Peek: Returns the top element of the stack without removing it.

IsEmpty: Checks if the stack is empty.

Size: Returns the no. of elements in the stack.

(ii) Queue

— A circular queue is an extended version of a normal queue where the last element of a queue is connected to the first element of the queue forming a circle.

— The operations are performed based on FIFO principle. It is also called 'Ring Buffer'.

Operations on stacks queue:

— operations associated with queue are —

Front : Get the front item from the queue.
Rear : Get the last item from the queue.
enqueue : This function is used to insert an element into the circular queue In a circular queue, the new element is always inserted at the rear position.
dequeue : This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from the front position.

Algorithm : To check if a tree is a BST i.e Binary Search Tree.

Recursive approach :
Base case : If the node is NULL, it is a BST.

— For each node, check if its left subtree contains only nodes with values less than node's value.
— Check if its right subtree contains only nodes with values greater than the node's value.
— Recursively, check the left & right subtrees.

Implementation.

```
def is_bst_util(root, min_value, max_value):
    if root is None:
        return True
    if root.data < min_value or root.data > max_value:
        return False
    return is_bst_util(root.left, min_value, root.data-1) and
        is_bst_util(root.right, root.data+1, max_value)
```

```
def is_bst(root):
    return is_bst_util(root, float('infinity'), float('infinity'))
```

Algorithm : To check if a number is present in BST.

Iterative Approach :

— start from the root node.

— If the target value is equal to the current node's value, return True.

— If the target value is less than the current node's value, move to the left child.

— If the target value is greater than the current node's value, move to the right child.

— continue this process until the target value is found or the search reaches a leaf node.

Implementation :

```
def search_bst(root, target):
    while root:
        if root.data == target:
            return True
        elif root.data < target:
            root = root.right
        else:
            root = root.left
    return False.
```

Conclusion : These algorithms provide efficient ways to check if a given binary Tree is BST & to search a value within it while ensuring efficient search, insert & deletion.

Experiment no. 09

Aim : Problems on heap

Theory :

Heap is a special tree structure in which each parent node is less than or equal to its child node. Then it is called a Min heap. If each parent node is greater than or equal to its child node then it is called a max heap.

1] To check if a binary tree is a min heap or not.

Algorithm :

is_complete (index, n) :
— For each node, calculate the left & right child indices.
— Ensure that both children are within bounds & recursively checks if the subtrees are complete.

is_min_heap (index, n) :
— check if the current node satisfies the min-heap property by comparing it with its left & right children.
— Recursively check all the nodes.

is_min_heap_tree () :
— first, check if the tree is complete and ~~the~~ then check if the min-heap property holds for all nodes.

3) Check if the binary tree is max heap or not

is_complete (index_n):
— for each node, calculate the left and right child indices.
— Ensure that both children are within bounds & recursively check if the subtrees are complete.

is_max_heap (index_, n):
— check if the current node satisfies the max_heap property by comparing it with its left & right children.
— Recursively check all the nodes.

is_max_heap_tree()
— First, check if the tree is complete and then check if the max-heap property holds for all nodes.

Heaps are easy to implement if we understand the fundamentals of how trees & arrays work. The most commonly used variant heap data structure is - the binary heap, which can be represented as arrays.
Heaps are efficient because they guarantee logarithmic levels of data. Heaps are not suitable for search operations where they can degenerate to linear time complexity.

Conclusion: Understanding heaps is important when working with data structures & algorithms in computer programming.

Experiment no. 10

Aim : Problems on Greedy Algorithms.

Theory :

Greedy Algorithm

— Greedy Algorithm are a class of algorithm that make locally optimal choices at each step with the help of finding a global optimum solution.

1] Fractional knapsack problem using Greedy Algorithm.

— Fractional knapsack Problem is a problem where you are given a set of items, each with a weight & a value, and a knapsack that has a maximum weight capacity.

— Algorithm :-

— start.
— Input : List of items with value, weight & knapsack capacity.
— for each item : calculate value to weight ratio, ratio = value/weight.
— Sort the items in descending order of their value-to-weight ratio.
— Initialize total_value = 0.
— for each item in a sorted list, If item_weight <= remaining weight capacity of a knapsack —

then :-
(i) Add item value to total value.
(ii) Subtract item weight from remaining capacity.

- Else, take the function of the item that fits —
  - Add value of fraction of the item to total value
  - Fill the knapsack & break the loop.
- output the total-value in the knapsack.
- End.

2] Huffmann coding Problem using Greedy Algorithm.

- It is a lossless data compression algorithm that is used to compress data. It is a type of algorithm, where the goal is to assign shorter codes to more frequent characters & longer codes to less frequent characters.

- Algorithm

- start
- Input : List of characters with frequencies.
- Build a min-heap where each element is a node.
- While the heap contain more than one node :-
  a) extract two nodes with the smallest frequencies from the heap.
  b) Create a new node with the sum of frequencies of two nodes as its frequency.
  c) Set the extracted node as the left & right children of the new node.

d) Insert the new node back into the heap.

- After the heap contains only one node, it is the root of the Huffmann tree.
- Traverse tree from root to leaves.
  - assign '0' for free left traversal and '1' for right traversal to create the binary code.
- output : The huffmann codes for each character.
- End.

Conclusion :

Greedy algorithms are a well-liked method for resolving optimization issues in algorithm design & analysis. Thus, they select the most appropriate option based on the current situation, even if the current best result may not bring about the overall optimal result.