1. Develop an algorithm and write a program to demonstrate basic data structures

Algorithm: demonstrate basic data structures

Step1: create variables of the required data type

Step2: Perform operations using the variables

Step3: Print result

Step4: find the required memory size using suitable function

Step5: Print memory size.

Step6:Exit

1. Develop an algorithm and write a program to demonstrate ADT with all its operations

Step1: Create a Abstract Data Type called car with Make, Fuel and Milage as attributes

Initialize Make, Fuel and Milage

display car details

update car details

Step2: Create an object of type car

Step3: Display the attribute values of the object using display function

Step4: Update the contents of the object using update function

Step5: Display the updated attribute values of the object using display function

Step6: Exit

1. Develop an algorithm and write a program to implement an ADT and compute time and space complexity

Algorithm: Implementing ADT

Step1: Create a Abstract Data Type called Add with **n1,n2,n3,n4,n5** and **sum** as attributes

Write a constructor to Initialize n1,n2,n3,n4,n5 and sum

Write a function **find\_sum** to find the sum of numbers

Write a function **display** to print the result

Step2: Create an object **c1** of type Add

Step3: Find the sum of numbers using c1.find\_sum()

Step4: Display the result using c1.display

Step5: Display the memory utilization using sys.getsizeof(c1)

Step6: Display time elapsed for the above computation using time.time()

Step7: Exit

1. Develop an algorithm and write program using arrays in ADTs , compute time and space complexity

Algorithm: Implementing ADT with array

Step1: Create a Abstract Data Type called Add with integer array **arr** and **sum** as attributes

Write a constructor to Initialize **array** and **sum**

Write a function **find\_sum** to find the sum of integers in the array

Write a function **display** to print the result

Step2: Create an object **c1** of type Add

Step3: Find the sum of numbers using c1.find\_sum()

Step4: Display the result using c1.display

Step5: Display the memory utilization using sys.getsizeof(c1)

Step6: Display time elapsed for the above computation using time.time()

Step7: Exit

1. Use suitable algorithm design strategy to search for a key in a given small unordered list and implement using Python

This problem can be solved using Brute force technique –**linear search algorithm**

Algorithm: Linear\_search(A[0..n-1])

[input: Array of unsorted elements]

[output: Index of the Element equal to key or -1]

[Search by comparing array element one by one with Key ]

Step 1: for i = 0 to n-1

if(A[i] == key)

return(i)

return(-1)

[Main function]

[input: Array of unsorted elements]

[output: Index of the Element equal to key or display “Not found”]

Step 1: Input Array elements

Step 2: result = Linear\_search(A[0..n-1])

Step 3: if(result == -1)

Display “Not Found”

else

Display(“Found at position:”, result)

Step 4: Stop

**Analysis of Linear search**

**Best case:**

Time efficiency O(1), if the key is the first element of the array.

**Worst case:**

Time efficiency is O(n), if the key is the last element of the array.

1. Use suitable algorithm design strategy to sort a small list of elements and implement using python(**Brute force technique-Bubble sort**)

Algorithm: Bubble\_sort(A[0..n-1])

[input: Array of unsorted elements]

[output: Array of sorted elements]

Step 1: for i = 0 to n-2

for j= 0 to n-i-2

if(A[j] > A[j + 1])

swap A[j] and A[j + 1]

Step 2: return

[Main function]

[input: Array of unsorted elements]

[output: Sorted Array]

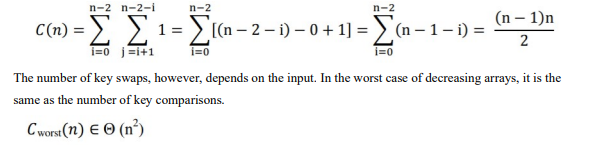
Step 1: Input Array elements

Step 2: Bubble\_sort(A[0..n-1])

Step 3: Display sorted Array

Step 4: Exit

Analysis of Bubble sort algorithm:



1. Use suitable algorithm design strategy to sort a small list of elements and implement using python(**Brute force technique-Selection sort**)
2. Use suitable algorithm design strategy to reduce the problem size(sort a big list) to solve it, and implement using python(**Decrease and conquer-Insertion sort**)

Algorithm: insertion\_sort(A[0..n-1])

[input: Array of unsorted elements]

[output: Array of sorted elements]

Step 1: for i = 0 to n-1

Key = A[i]

J =i – 1

Whle(j >=0) && (A[j] > key)

A[j+1] = A[j]

J = j – 1

A[j+1] = key

Step 2: return

[Main function]

[input: Array of unsorted elements]

[output: Sorted Array]

Step 1: Input Array elements

Step 2: insertion\_sort(A[0..n-1])

Step 3: Display sorted Array

Step 4: Exit

Analysis of insertion sort algorithm:

The best case running time of the insertion sort is **O(n)**.

The best case occurs when the input array is already sorted. As the elements are already sorted, only one comparison is made on each pass, so that the time required is O(n).



The worst case time complexity of insertion sort is O(n2).

1. Use suitable algorithm design strategy to decompose the given list into sub lists and sort by solving the sub lists.(**Divide & conquer-Merge sort**)

[input: Array of unsorted elements]

[output: Array of sorted elements]

Algorithm: Simple Merge (A, low, mid, high)

Purpose: Merge two sorted arrays where the first array starts from low to mid and the second starts from mid + 1 to high

[input: Array of unsorted elements]

[output: Array of sorted elements]

Step 1: i = low, j = mid+1, k = low

Step 2: while ( i <= mid and j <= high)

If ( A[i] < A[j] ) then

C[k] = A[i]

i = i + 1

k = k +1

else

C[k] = A[j]

K = k + 1

End if

End while

Step 3: while (i <= mid)

C[k] = A[i]

K = k + 1

i = i + 1

end while

Step 4: while (j <= high)

C[k] = A[i]

K = k +1, j = j +1

End while

Step 5: for i = low to high

A[i] = C[i]

End for

Algorithm: Merge Sort (A, Low, High)

Purpose: sort the elements of the array between the lower bound and upper bound

Input: A is an unsorted vector with low and high as lower bound and upper

Output: A as a sorted vector

Step 1: if (low < high)

Mid = (low + high)/2

MergeSort(a, low, mid)

MergeSort(a, mid +1, high)

SimpleMerge(a, low, mid, high)

End if

# Time complexity of merge sort is given by

# T(n) =0 if n=1

# T(n) = T(n/2) + T(n/2) + n otherwise

# T(n/2) is the time required to sort the upper part of array

# T(n/2) is the time required to sort the lower part of array

# n is the time required to divide and merge one instance of a solution with n elements

# This can be simplified to get

# T(n) € θ(nlog2n)

1. Use suitable algorithm design strategy to decompose the given list into sub lists and search for the given key.(**Divide & conquer- Binary search**)

**Algorithm:Binary search(A, Low,High,key)**

Step 1:Low = 0

Step 2:High = n-1

Step 3:While(Low <= High)

Mid = (Low + High)/2

If (key = A[mid])

Return(mid)

If(key < A[mid])

Binary search(A, Low,Mid-1,key)

else

Binary search(A, Mid+1,High,key)

end if

end while

Step 4: Return(-1)

#Driver code

Step 1:Input Array A

Step 2:Input Key

Step 2:Result = Binary search(A,Low,High,Key)

Step 3:Print Result

# Analysis:

# Time complexity:

# Best Case:

# When the item to be searched is found in the middle of the array. Therefore the time complexity is given by Big Omega(1)

# Worst case:

# When the item to be searched is found in either in the beginning or at the end of the array. Therefore the time complexity is given by

# T(n) =1 if n=1

# T(n) = T(n/2) + 1 otherwise

# T(n/2) is the time required to search the upper or lower part of array

# 1 is the time required to compare middle element

# This can be simplified to get

# T(n) € θ(log2n)

# Average Case:

# T(n) € θ(log2n)

1. Use suitable algorithm design strategy to generate **Fibonacci sequence** and implement using python

Fib(n)

Step 1: Create list F=[0,1]

Step 2: For i=2 to n

Append(F[i-1] +F[i-2])

Return(f)

Step 1 :INPUT(n)

Step 2: Fib(n)

Step 3: EXIT

1. Write an algorithm to **create linked list**, inserting node in the front and deleting the node from front. Implement using Python

**Algorithm: inserting the node at the beginning**

**Insert(HEAD)**

Step 1: Create NEW\_NODE

Step 2: SET NEW\_NODE -> DATA = VAL

Step 3 :SET NEW\_NODE -> NEXT = HEAD

Step 4: SET HEAD = NEW\_NODE

Step 5: EXIT

**Algorithm : Remove node at the beginning**

**Remove(HEAD)**

Step 1: **IF** HEAD = **NULL**

Write Empty list

EXIT

Step 2: SET PTR = HEAD

Step 3: SET HEAD = HEAD -> NEXT

Step 4: EXIT

**Algorithm : To search an element**

**Search(HEAD,key)**

Step 1: ptr = HEAD

Step 2: while ((ptr is not None) and (ptr.data != key)):

ptr = ptr->next

if(ptr is None):

print("key not found")

else:

print("key found")

Step 3: return

**Algorithm :Listprint()**

Step 1: ptr = Head

Step 2:if(ptr is None):

print("Empty list")

while ptr is not None:

print(ptr.data, end = "->")

ptr = ptr->next

print()

Step 3: Return

**Algorithm :Creating linked list by inserting at front, deleting from front and searching**

LinkedList()

Step 1:n= Input number of nodes required

Step 2:for i=0 to n-1

Insert(Node)

Step 2: Listprint()

Step 3:Remove(HEAD)

Step 4: key = input key to search

Step 5: Search(HEAD,key)

Step 6: Listprint()

Step 7: Stop

The time complexity for initializing a singly linked list is **O(1).**The space complexity is **O(1)** as no additional memory is required to initialize the linked list.

1. Implement Doubly Linked list
2. Implement Stack
3. Implement solution to Tower of Hanoi Problem

Tower\_of\_hanoi(n,s,d,a)

if n==1

Print(“Move disk 1 from s to d)

Return

Tower\_of\_hanoi(n-1,s,a,d)

Print(“Move disk n from s to d)

Tower\_of\_hanoi(n-1,a,d,s)

Step 1: n=3

Step 2: Tower\_of\_hanoi(n,’a’,’b’,’c’)

Step 3: Exit

1. Implement Simple Queue

[Algorithm to Create a simple Queue with capacity n. Queue has rear and front pointers.

rear pointer to insert element and front pointer to delete element. Insertation deletion

and diplay are the operations performed on Queue]

Step1: q = create queue with capacity n

Step2: print("Enter 1 to insert 2 to delete 3 to display 4 to Stop :")

Step3: read(choice)

Step4: while(choice == 1 or choice == 2 or choice == 3):

if(choice == 1):

insertqueue()

elif(choice == 2):

deletequeue()

elif(choice == 3):

display()

else:

break

print("Enter 1 to insert 2 to delete 3 to display 4 to Stop :")

read(choice)

Step5: Stop

[Function to delete value from the front end of queue]

deletequeue()

Step1: if(front == -1)

print("Queue Empty")

return

Step2: if(rear == front)

rear = front = -1

return

Step3: front = front + 1

Step4: return

[Function to insert value at the rear end of queue]

insertqueue()

Step1: if(rear == (capacity-1))

print("Overflow")

return

Step2: read(value)

Step3: rear = rear + 1

Step4: append value to rear end of the list

Step5: if(front == -1)

front = 0

Step6: return

[Function to display queue]

display()

Step1: if(front == -1)

print("Queue Empty")

Step2: i = front

Step3: while(i <= rear)

print(q[i])

1. Implement Priority Queue

[A simple implementation of Priority Queue]

Step1: Create a Priority Queue with required size

Step2: [insert into queue]

myQueue.insert(12)

myQueue.insert(1)

myQueue.insert(14)

myQueue.insert(7)

Step3: print("Queue created")

print(myQueue.queue)

Step4: print("Demonstrating deletion from priority Queue")

while not isEmpty()

print(myQueue.delete())

[Function for checking if the queue is empty]

isEmpty():

Step1: return (length(queue) == 0)

[Function for inserting an element in the queue]

insert(data)

Step1: Insert the data to queue(list)

[Function for deleting an element based on Priority]

delete()

Step1: max\_val = 0

Step2: for i in range(len(queue))

if queue[i] > queue[max\_val]

max\_val = i

Step3: item = queue[max\_val]

Step4: remove queue[max\_val] from queue

Step5: return item

1. Implement Binary search tree
2. Implement DFS
3. Implement BFS
4. Implement Hash function