ANNA UNIVERSITY

JAYA ENGINEERING COLLEGE

THIRUNINRAVUR-602 024

(An ISO 9001:2000 Certified Institution)

Email : info@jec.ac.in
Website: www.jec.ac.in

DEPARTMENT OF INFORMATION TECHNOLOGY

PRACTICAL RECORD

CD3281 DATA STRUCTURES AND ALGORITHMS LABORATORY

NAME	:	______________________________
REGNO.	:	______________________________
YEAR	:	______________________________
SEM	:	______________________________

ANNA UNIVERSITY
JAYA ENGINEERING COLLEGE

THIRUNINRAVUR-602 024

(An ISO 9001:2000 Certified Institution)
Email : info@jec.ac.in
Website: www.jec.ac.in

BONAFIDE CERTIFICATE

This is to certify that this is a bonafide record of work done by

Mr./Ms.________________________________ Reg.No.____________________________

of **B.TECH / Information Technology** in **CD3281 DATA STRUCTURES AND ALGORITHMS LABORATORY** in the **III** semester during **AUGUST 2022 to JANUARY 2023.**

Head of the Department **Staff-In-Charge**

Submitted For Practical Examination Held On ________________

Internal Examiner **External Examiner**

VISION OF THE INSTITUTE

To Achieve Excellence in Technical Education through Innovative Teaching and Multidisciplinary Research with Professionalism to Serve the Global Society.

MISSION OF THE INSTITUTE

Jaya Engineering College (JEC) will Endeavor

M1 -To provide state of art teaching and learning for Engineering and Technology, Research and Management studies.

M2 -To provide quality education, self discipline and ethical values.

M3 - To associate with R&D and industries to have connectivity with the society.

M4 – To impart knowledge to become empowered professionals in the field of Engineering and Management

VISION OF THE DEPARTMENT

To bridge the gap between the academician and the Industry. To impart quality education through innovative teaching and learning method.

MISSION OF THE DEPARTMENT

To develop employable software developing graduates with knowledge, skills and ethics; provide them with the professional and soft skills necessary to lead a successful career.

PROGRAMME EDUCATIONAL OBJECTIVES (PEO)

PEO 1

Demonstrate technical competence with analytical and critical thinking to understand andmeet the diversified requirements of industry, academia and research.

PEO 2

Exhibit technical leadership, team skills and entrepreneurship skills to provide businesssolutions to real world problems.

PEO 3

Work in multi-disciplinary industries with social and environmental responsibility, workethics and adaptability to address complex engineering and social problems

PEO 4

Pursue lifelong learning, use cutting edge technologies and involve in applied research todesign optimal solutions.

PROGRAMME OUTCOMES

ENGINEERING GRADUATES WILL BE ABLE TO:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OBJECTIVES (PSOs)

1. Have proficiency in programming skills to design, develop and apply appropriatetechniques, to solve complex engineering problems.
2. Have knowledge to build, automate and manage business solutions using cutting edgetechnologies
3. Have excitement towards research in applied computer technologies.

.

COURSE OUTCOMES

At the end of the course, learners will be able to:

- Implement ADTs as Python classes
- Design, implement, and analyse linear data structures, such as lists, queues, and stacks,according to the needs of different applications.
- Design, implement, and analyse efficient tree structures to meet requirements such assearching, indexing, and sorting
- Model problems as graph problems and implement efficient graph algorithms to solve them.

CO's-PO's&PSO'sMAPPING

CO's	PO's												PSO's		
	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3
1	3	2	2	2	3	-	-	-	1	1	2	2	-	-	-
2	3	2	2	1	3	-	-	-	2	2	2	2	-	-	-
3	3	2	2	2	3	-	-	-	3	1	1	2	-	-	-
4	3	2	2	1	3	-	-	-	1	2	3	1	-	-	-
Avg.	3	2	2	2	3	-	-	-	2	2	2	2	-	-	-

SYLLABUS

CD3281 DATA STRUCTURES AND ALGORITHMS LABORATORY **L T P C**
0 0 4 2

COURSE OBJECTIVES:

- To implement ADTs in Python
- To design and implement linear data structures – lists, stacks, and queues
- To implement sorting, searching and hashing algorithms
- To solve problems using tree and graph structures

LIST OF EXERCISES:

1. Implement simple ADTs as Python classes
2. Implement recursive algorithms in Python
3. Implement List ADT using Python arrays
4. Linked list implementations of List
5. Implementation of Stack and Queue ADTs
6. Applications of List, Stack and Queue ADTs
7. Implementation of sorting and searching algorithms
8. Implementation of Hash tables
9. Tree representation and traversal algorithms
10. Implementation of Binary Search Trees
11. Implementation of Heaps
12. Graph representation and Traversal algorithms
13. Implementation of single source shortest path algorithm
14. Implementation of minimum spanning tree algorithms

Total: 60 Periods

INDEX

LIST OF EXPERIMENTS

Ex. No.1 **Introduction of Data structures and Algorithms**
Date:

Data Structure

As already defined, the data structure is simply organizing the data in the memory, so that memory can be used efficiently and the process of data retrieval becomes fast. There are many kinds of data structures that are categorized into different categories, some data structures are array, stack, queue, linked list, etc.

How data structure relate to the Algorithm?

The data structure is a set of algorithms, that can be used to implement in any programming language to organize the data in the memory of the system. The data structure is not a programming language, the data structure is the concept of memory organization which is further written with the help of the algorithm and the algorithm is further programmed/ implemented using any programming language.

Why study Data structure and Algorithms?

With the advancement in technology, applications are getting data-rich and more complicated nowadays. Some common issues that most applications face today are:

Searching Speed: Due to the huge amount of data, the searching speed of data is getting slower as data increases. Consider there is an application with 10 million users' data, and for a single data request system has to search 10 million data every time. This huge amount of data leads to a slowing down speed of the application.

Processor speeds: The processor's speed is very high indeed, but if the data will be in billion so this will also slow down.

Multiple requests: Multiple requests are raised to the server at a time, if the system is not optimized then it can lead to server breakdown as it won’t be able to handle thousands of requests at a time.

To resolve all the above-mentioned issues, the data structure was introduced. With the help of data structure, we can minimize the search speed which will eventually increase the response time and processor speed as well. Data can be organized in such a manner, that while searching for a particular item we need not search entire data, Or the most relevant data can be kept on priority and it can be accessed more quickly than others, and many more. So, the data structure is the backbone of today’s highly efficient and faster-moving world.

If you study Data structure along with Algorithms, you will be able to:

Write optimized code: Once you will get to know about different data structures then you can code more efficient programs, you can identify which data structure is best for which kind of data, and code it accordingly.

Better utilization of Time & Memory: Time and memory are the most costlier element. With the help of data structure, you will be able to write code that is really quick and uses less memory space.

Job opportunities: There is a high demand for good Software developers worldwide and with sufficient knowledge of data structure and algorithms you can easily crack the interviews of Google, Microsoft, Amazon, and many more.

Foundation of Data Structure

Interface: The interface of a data structure depicts the types of operations supported by the data structure. Each data structure has an interface and the interface tells only about the supported operations, the type of parameters that are accepted, and the return type of the supported operation.

Implementation: Implementation of a data structure provides the information regarding the set of algorithms used in supported operations. It is an internal representation.

Time Complexity: Data structure is studied with the sole aim of optimization, so the running time/ execution time complexity must be as minimal as possible.

Space Complexity: Memory usage must be as minimum as possible in any data structure.

Correctness: The data structure must implement its interface(supported operations) correctly.

Types of Data Structure

The data structure is categorized as follows:

- **Primitive data Structure:** It is a primitive data type, that can hold a single value. Like: int, char, float, double, and pointer.

- **Non-primitive data structure:** Data structure that can hold more than one value, this is of two types:

- **Linear Data Structure:** When data is arranged in a sequential manner, where one element is connected to another element sequentially. Like: Array, Linked list, stack, queue, etc.
- **Non-Linear data structure:** When data is arranged in random order and one element is connected to "n" elements. Like: Tree and graph.

- The data structure is also classified as:
- **Static Data Structure:** Data structure whose size is allocated at compile time, and hence such data structure has a fixed size.
- **Dynamic Data Structure:** Data structure whose size is allocated at run time, and hence such data structure has a flexible size.

- Major Operations in Data Structure

- **Searching:** Finding any element in the data structure
- **Insertion:** Inserting any element into a given data structure
- **Deletion:** Deleting any element from a data structure.
- **Updation:** Updating any information/element from a given data structure, or replacing an existing element with another element.
- **Sorting:** Organizing all the elements of a data structure either in ascending or descending order.

Ex. No.2 **Simple ADT using array**
Date:

Aim:

To write an algorithm and program for simple ADT using array in Python programming.

Algorithm:

```
STEP 1: Start.
STEP 2: import array for array creation.
STEP 3: Creating an array with integer type.
STEP 4: Printing the value of integer array.
STEP 5: Creating an array with float type.
STEP 6: Printing the values of float array.
STEP 7: Stop
```

Program:

```
# Python program to demonstrate
# Creation of Array

# importing "array" for array creations
import array as arr
# creating an array with integer type
a = arr.array('i', [1, 2, 3])
# printing original array
print ("The new created array is : ", end =" ")
```

```
for i in range (0, 3):
  print (a[i], end =" ")
print()
# creating an array with float type
b = arr.array('d', [2.5, 3.2, 3.3])

# printing original array
print ("The new created array is : ", end =" ")
for i in range (0, 3):
  print (b[i], end =" ")
```

OUTPUT:

```
The new created array is :  1 2 3
The new created array is :  2.5 3.2 3.3
```

RESULT:

Thus the output is verified.

Ex. No. 3 **Simple ADT Using Queue.**

Date:

Aim:

To write an algorithm and program for simple ADT using Queue in Python programming.

Algorithm:

Step 1 − Check if the queue is full.

Step 2 − If the queue is full, produce overflow error and exit.

Step 3 − If the queue is not full, increment rear pointer to point the next empty space.

Step 4 − Add data element to the queue location, where the rear is pointing.

Step 5 − return success.

Program:

```
# Python program to
# demonstrate implementation of
# queue using queue module
from queue import Queue
# Initializing a queue
q = Queue(maxsize = 3)
# qsize() give the maxsize
# of the Queue
print(q.qsize())
# Adding of element to queue
```

```
q.put('a')

q.put('b')

q.put('c')

# Return Boolean for Full

# Queue

print("\nFull: ", q.full())

# Removing element from queue

print("\nElements dequeued from the queue")

print(q.get())

print(q.get())

print(q.get())

# Return Boolean for Empty

# Queue

print("\nEmpty: ", q.empty())

q.put(1)

print("\nEmpty: ", q.empty())

print("Full: ", q.full())

# This would result into Infinite

# Loop as the Queue is empty.

# print(q.get())
```

Output:

```
0

Full:  True
```

```
Elements dequeued from the queue
a
b
c
Empty:  True
Empty:  False
Full:  False
```

RESULT:

Thus the output is verified.

Ex. No.4

Date: **Simple ADT Using Stack**

Aim:

To write an algorithm and program for simple ADT using array in Python programming.

Algorithm:

```
Step 1 − Checks if the stack is empty.
Step 2 − If the stack is empty, produces an error and exit.
Step 3 − If the stack is not empty, accesses the data element at which top is pointing.
Step 4 − Decreases the value of top by 1.
Step 5 − Returns success.
```

Program:

```
// Simple ADT using Stack using List
# Python program to
# demonstrate stack implementation
# using list
stack = []
# append() function to push
# element in the stack
stack.append('a')
stack.append('b')
stack.append('c')
print('Initial stack')
print(stack)
# pop() function to pop
# element from stack in
# LIFO order
```

```
print('\nElements popped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())
print('\nStack after elements are popped:')
print(stack)
# uncommenting print(stack.pop())
# will cause an IndexError
# as the stack is now empty
```

Output:

```
Initial stack
['a', 'b', 'c']
Elements popped from stack:
c
b
a
Stack after elements are popped:
[]
```

RESULT:

Thus the output is verified.

Ex. No. 5 **Applications of stack**

Date:

Aim:

To write an algorithm and program for find factorial of a number using Recursion

Algorithm:

1. Conversion of Infix to Postfix

Algorithm for Infix to Postfix

Step 1: Consider the next element in the input.

Step 2: If it is operand, display it.

Step 3: If it is opening parenthesis, insert it on stack.

Step 4: If it is an operator, then

If stack is empty, insert operator on stack.

If the top of stack is opening parenthesis, insert the operator on stack

If it has higher priority than the top of stack, insert the operator on stack.

Else, delete the operator from the stack and display it, repeat Step 4.

Step 5: If it is a closing parenthesis, delete the operator from stack and display them until an opening parenthesis is encountered. Delete and discard the opening parenthesis.

Step 6: If there is more input, go to Step 1.

Step 7: If there is no more input, delete the remaining operators to output.

2. Infix to Prefix

Algorithm for Infix to Prefix Conversion:

Step 1: Insert “)” onto stack, and add “(” to end of the A .

Step 2: Scan A from right to left and repeat Step 3 to 6 for each element of A until the stack is empty .

Step 3: If an operand is encountered, add it to B .

Step 4: If a right parenthesis is encountered, insert it onto stack .

Step 5: If an operator is encountered then,

a. Delete from stack and add to B (each operator on the top of stack) which has same or higher precedence than the operator.

b. Add operator to stack.

Step 6: If left parenthesis is encountered then ,

a. Delete from the stack and add to B (each operator on top of stack until a left parenthesis is encountered).

b. Remove the left parenthesis.

Step 7: Exit

3. Postfix to Infix

Following is an algorithm for Postfix to Infix conversion:

Step 1: While there are input symbol left.

Step 2: Read the next symbol from input.

Step 3: If the symbol is an operand, insert it onto the stack.

Step 4: Otherwise, the symbol is an operator.

Step 5: If there are fewer than 2 values on the stack, show error /* input not sufficient values in the expression */

Step 6: Else,

a. Delete the top 2 values from the stack.

b. Put the operator, with the values as arguments and form a string.

c. Encapsulate the resulted string with parenthesis.

d. Insert the resulted string back to stack.

Step 7: If there is only one value in the stack, that value in the stack is the desired infix string.

Step 8: If there are more values in the stack, show error /* The user input has too many values

4. Prefix to Infix

Algorithm for Prefix to Infix Conversion

```
Step 1: The reversed input string is inserted into a stack -> prefixToInfix(stack)
Step 2: If stack is not empty,
        a. Temp -> pop the stack
        b. If temp is an operator,
            Write a opening parenthesis "(" to show -> prefixToInfix(stack)
            Write temp to show -> prefixToInfix(stack)
            Write a closing parenthesis ")" to show
        c. Else If temp is a space ->prefixToInfix(stack)
        d. Else, write temp to show if stack.top != space ->prefixToInfix(stack)
```

Program:

```
# Python program to convert infix expression to postfix
# Class to convert the expression
class Conversion:
    # Constructor to initialize the class variables
    def __init__(self, capacity):
        self.top = -1
        self.capacity = capacity
        # This array is used a stack
        self.array = []
        # Precedence setting
        self.output = []
        self.precedence = {'+': 1, '-': 1, '*': 2, '/': 2, '^': 3}
    # check if the stack is empty
```

```
def isEmpty(self):
    return True if self.top == -1 else False
# Return the value of the top of the stack
def peek(self):
    return self.array[-1]
# Pop the element from the stack
def pop(self):
    if not self.isEmpty():
        self.top -= 1
        return self.array.pop()
    else:
        return "$"
# Push the element to the stack
def push(self, op):
    self.top += 1
    self.array.append(op)
# A utility function to check is the given character
# is operand
def isOperand(self, ch):
    return ch.isalpha()
# Check if the precedence of operator is strictly
# less than top of stack or not
def notGreater(self, i):
    try:
        a = self.precedence[i]
        b = self.precedence[self.peek()]
```

```
            return True if a <= b else False
        except KeyError:
            return False
# The main function that
# converts given infix expression
# to postfix expression
def infixToPostfix(self, exp):
    # Iterate over the expression for conversion
    for i in exp:
        # If the character is an operand,
        # add it to output
        if self.isOperand(i):
            self.output.append(i)
        # If the character is an '(', push it to stack
        elif i == '(':
            self.push(i)
        # If the scanned character is an ')', pop and
        # output from the stack until and '(' is found
        elif i == ')':
            while((not self.isEmpty()) and
                self.peek() != '('):
                a = self.pop()
                self.output.append(a)
            if (not self.isEmpty() and self.peek() != '('):
                return -1
            else:
                    self.pop()
            # An operator is encountered
            else:
                while(not self.isEmpty() and self.notGreater(i)):
                    self.output.append(self.pop())
                self.push(i)
        # pop all the operator from the stack
        while not self.isEmpty():
            self.output.append(self.pop())
        print "".join(self.output)
# Driver's code
if __name__ == '__main__':
    exp = "a+b*(c^d-e)^(f+g*h)-i"
    obj = Conversion(len(exp))
    # Function call
    obj.infixToPostfix(exp)
```

Output:

```
    abcd^e-fgh*+^*+i-
```

RESULT:

Thus the output is verified.

Ex. No. 6(a) **Factorial Using Recursion**

Date:

Aim:

To write an algorithm and program for find factorial of a number using Recursion.

Algorithm:

```
Step 1: Start
Step 2: take input from the user for finding the factorial.
Step 3: Create a variable 'factorial' and assign the value 1.
Step 4: if(number<0):
            print 'cannot be calculated.
        elif ( number == 1):
            print 1
        else:
            for i in range(1, number+1):
                factorial*=i
Step 5: print factorial
Step 7: Stop
```

Program:

```
def recur_factorial(n):
  if n == 1:
     return n
  else:
     return n*recur_factorial(n-1)
num = 7
```

```
# check if the number is negative
if num < 0:
   print("Sorry, factorial does not exist for negative numbers")
elif num == 0:
   print("The factorial of 0 is 1")
else:
   print("The factorial of", num, "is", recur_factorial(num))
```

Output:

```
The Factorial of 7 is 5040.
```

RESULT:

Thus the output is verified.

Ex. No.6(b) **Sum of Sequence using recursion**

Date:

Aim:

To write an algorithm and program for find factorial of a number using Recursion.

Algorithm:

```
Step 1 − Start
Step 2 – using the function sum of sequence is calculate.
Step 3 – Total as been initialized as ‘0’.
Step 4 – Range function is used to print the values are added.
Step 5 – Print the sum.
Step 6 –Stop.
```

Program:

```
def sum(n):
    total=0
for index in range (n+1):
    total+=index
return total
result total
result=sum(5)
print(result)
```

Output:

```
The sum is 15.
```

Result:

Thus the output is verified.

Ex. No.7 **Implement list using Python**

Date:

Aim:

To write an algorithm and program for implement list.

Algorithm:

```
Step 1 − Start
Step 2 – Creating the list.
Step 3 – Viewing the element in list.
Step 4 – assign and accessing data.
Step 5 – Append the list using append().
Step 6 –Extend the list using exciting.
Step 7 –Stop.
```

Program:

```
List1=[1,2,3,4]
Print(list1[1])
Print(list1[1:4])
Print(list1[-1])
List1=[]
For I in range (0,11):
    Print(list1[i])
List1=[1,2,3,4]
List1[2]=5
Print(list1)
```

```
List1.extend([1,2,3])
Print(list1)
List1=[1,2,3,4,5]
Print(list1)
Del list[2]
Print(list1)
```

Output :

```
2
[2,3,4]
4
0
1
2
3
4
5
6
7
8
9
10
[1,2,5,4,6]
```

Result:

Thus the output is verified.

Ex. No.8(a) **Traversing in a linked list**

Date:

Aim:

To write an algorithm and program for traverse of linked list.

Algorithm:

Step 1: Start

Step 2: Create classes Node and Linked List for creating a node and liking with the head node.

Step 3: Define proper methods in the class such as append, display for adding elements and display operations in the linked list.

Step 4: Create an instance for the linked list class.

Step 5: Call the methods.

Step 6: Stop

Program:

```
class Node:
  def _init_(self, dataval=None):
    self.dataval = dataval
    self.nextval = None
class SLinkedList:
  def _init_(self):
    self.headval = None
  def listprint(self):
    printval = self.headval
    while printval is not None:
        print (printval.dataval)
        printval = printval.nextval
list = SLinkedList()
list.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")
# Link first Node to second node
list.headval.nextval = e2
# Link second Node to third node
e2.nextval = e3
list.listprint()
```

Output:

```
Mon

Tue

Wed
```

Result:

Thus the output is verified.

Ex. No.8(b) **Inserting a linked list at front**

Date:

Aim:

To write an algorithm and program for insertion at front of linked list.

Algorithm:

Step 1: Start

Step 2: Create classes Node and Linked List for creating a node and liking with the head node.

Step 3: Define proper methods in the class such as append, display for adding elements and display operations in the linked list.

Step 4: Create an instance for the linked list class.

Step 5: Call the methods.

Step 6: Stop

Program:

```
class Node:
  def _init_(self, dataval=None):
    self.dataval = dataval
    self.nextval = None
class SLinkedList:
  def _init_(self):
    self.headval = None
# Print the linked list
  def listprint(self):
    printval = self.headval
```

```
      while printval is not None:
         print (printval.dataval)
         printval = printval.nextval
   def AtBegining(self,newdata):
      NewNode = Node(newdata)
# Update the new nodes next val to existing node
   NewNode.nextval = self.headval
   self.headval = NewNode
list = SLinkedList()
list.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")
list.headval.nextval = e2
e2.nextval = e3
list.AtBegining("Sun")
list.listprint()
```

Output:

```
Sun

Mon

Tue

Wed
```

Result:

Thus the output is verified.

Ex. No.8(c) **Inserting a linked list at End**

Date:

Aim:

To write an algorithm and program for insertion at End of linked list.

Algorithm:

Step 1: Start

Step 2: Create classes Node and Linked List for creating a node and liking with the head node.

Step 3: Define proper methods in the class such as append, display for adding elements and display operations in the linked list.

Step 4: Create an instance for the linked list class.

Step 5: Call the methods.

Step 6: Stop

Program:

```
class Node:
   def _init_(self, dataval=None):
      self.dataval = dataval
      self.nextval = None
class SLinkedList:
   def _init_(self):
      self.headval = None
# Function to add newnode
   def AtEnd(self, newdata):
      NewNode = Node(newdata)
```

```
        if self.headval is None:
            self.headval = NewNode
            return
        laste = self.headval
        while(laste.nextval):
            laste = laste.nextval
        laste.nextval=NewNode
# Print the linked list
    def listprint(self):
        printval = self.headval
        while printval is not None:
            print (printval.dataval)
            printval = printval.nextval
list = SLinkedList()
list.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")
list.headval.nextval = e2
e2.nextval = e3
list.AtEnd("Thu")
list.listprint()
```

Output:

```
Mon

Tue

Wed

Thu
```

Result:

Thus the output is verified.

Ex. No.8(d) **Inserting a linked list between two nodes**

Date:

Aim:

To write an algorithm and program for insertion linked list between two nodes.

Algorithm:

```
Step 1: Start
Step 2: Create classes Node and Linked List for creating a node and liking with the head node.
Step 3: Define proper methods in the class such as append, display for adding elements and display operations in the linked list.
Step 4: Create an instance for the linked list class.
Step 5: Call the methods.
Step 6: Stop
```

Program:

```
class Node:
   def _init_(self, dataval=None):
      self.dataval = dataval
      self.nextval = None
class SLinkedList:
   def _init_(self):
      self.headval = None
# Function to add node
   def Inbetween(self,middle_node,newdata):
      if middle_node is None:
            print("The mentioned node is absent")
            return
        NewNode = Node(newdata)
        NewNode.nextval = middle_node.nextval
        middle_node.nextval = NewNode
# Print the linked list
    def listprint(self):
        printval = self.headval
        while printval is not None:
            print (printval.dataval)
            printval = printval.nextval
list = SLinkedList()
list.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Thu")
list.headval.nextval = e2
e2.nextval = e3
list.Inbetween(list.headval.nextval,"Fri")
list.listprint()
```

Output:

```
Mon

Tue

Fri

Thu
```

Result:

Thus the output is verified.

Ex. No.9(a) **Bubble Sort**

Date:

Aim:

To write an algorithm and program for Bubble sort.

Algorithm:

```
Step 1: Repeat For i = 0 to N-1
Step 2: Repeat For j = i + 1 to N - I
Step 3: If Arr[ j ] > Arr[ i ]
            swap Arr[ j ] and Arr[ i ]
            [end of step 2]
            [end of step 1]
Step 4: stop
```

Program:

```
# Python program for implementation of Bubble Sort
def bubbleSort(arr):
    n = len(arr)
     # Traverse through all array elements
    for i in range(n):
        # Last i elements are already in place
        for j in range(0, n-i-1):
            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
# Driver code to test above
if _name_ == "_main_":
  arr = [5, 1, 4, 2, 8]
  bubbleSort(arr)
  print("Sorted array is:")
  for i in range(len(arr)):
      print("%d" % arr[i], end=" ")
```

Output:

```
Sorted array:
1 2 4 5 8
```

Result:

Thus the output is verified.

Ex. No.9(b) **Quick Sort**

Date:

Aim:

To write an algorithm and program for Quick sort.

Algorithm:

Step 1 − Choose the highest index value has pivot

Step 2 − Take two variables to point left and right of the list excluding pivot

Step 3 − left points to the low index

Step 4 − right points to the high

Step 5 − while value at left is less than pivot move right

Step 6 − while value at right is greater than pivot move left

Step 7 − if both step 5 and step 6 does not match swap left and right

Step 8 − if left ≥ right, the point where they met is new pivot

Program:

```
# Python3 implementation of QuickSort
# Function to find the partition position
def partition(array, low, high):
    # Choose the rightmost element as pivot
    pivot = array[high]
    # Pointer for greater element
    i = low - 1
    # Traverse through all elements
    # compare each element with pivot
    for j in range(low, high):
        if array[j] <= pivot:
            # If element smaller than pivot is found
            # swap it with the greater element pointed by i
            i = i + 1
            # Swapping element at i with element at j
            (array[i], array[j]) = (array[j], array[i])

    # Swap the pivot element with
    # e greater element specified by i
    (array[i + 1], array[high]) = (array[high], array[i + 1])
    # Return the position from where partition is done
    return i + 1
# Function to perform quicksort
def quick_sort(array, low, high):
    if low < high:
  # Find pivot element such that
        # element smaller than pivot are on the left
        # element greater than pivot are on the right
        pi = partition(array, low, high)
         # Recursive call on the left of pivot
        quick_sort(array, low, pi - 1)
         # Recursive call on the right of pivot
        quick_sort(array, pi + 1, high)
# Driver code
array = [10, 7, 8, 9, 1, 5]
```

```
quick_sort(array, 0, len(array) - 1)
 print(f'Sorted array: {array}')
```

Output:

```
Sorted array:
1 5 7 8 9 10
```

Result:

Thus the output is verified.

Ex. No. 9(c) **Merge Sort**

Date:

Aim:

To write an algorithm and program for merge sort.

Algorithm:

```
step 1: start
step 2: declare array and left, right, mid variable
step 3: perform merge function.
    if left > right
        return
    mid= (left+right)/2
    mergesort(array, left, mid)
    mergesort(array, mid+1, right)
    merge(array, left, mid, right)
step 4: Stop
```

Program:

```
# Python program for implementation of MergeSort
# Merges two subarrays of arr[].
# First subarray is arr[l..m]
# Second subarray is arr[m+1..r]
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m
```

```
# create temp arrays
L = [0] * (n1)
R = [0] * (n2)
# Copy data to temp arrays L[] and R[]
for i in range(0, n1):
    L[i] = arr[l + i]
for j in range(0, n2):

    R[j] = arr[m + 1 + j]
# Merge the temp arrays back into arr[l..r]
i = 0     # Initial index of first subarray
j = 0     # Initial index of second subarray
k = l     # Initial index of merged subarray
while i < n1 and j < n2:
    if L[i] <= R[j]:
        arr[k] = L[i]
        i += 1
    else:
        arr[k] = R[j]
        j += 1
    k += 1
# Copy the remaining elements of L[], if there
# are any
while i < n1:
    arr[k] = L[i]
    i += 1
```

```
        k += 1
    # Copy the remaining elements of R[], if there
    # are any
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1
# l is for left index and r is right index of the
# sub-array of arr to be sorted
 def mergeSort(arr, l, r):
    if l < r:
        # Same as (l+r)//2, but avoids overflow for
        # large l and h
        m = l+(r-l)//2
        # Sort first and second halves
        mergeSort(arr, l, m)
        mergeSort(arr, m+1, r)
        merge(arr, l, m, r)
# Driver code to test above
arr = [12, 11, 13, 5, 6, 7]
n = len(arr)
print("Given array is")
for i in range(n):
    print("%d" % arr[i],end=" ")
mergeSort(arr, 0, n-1)
print("\n\nSorted array is")
```

```
for i in range(n):
    print("%d" % arr[i],end=" ")
```

Output:

```
Given array is
12 11 13 5 6 7
Sorted array is
5 6 7 11 12 13
```

Result:

Thus the output is verified.

Ex. No. 9(d) **Selection Sort**

Date:

Aim:

To write an algorithm and program for Selection sort.

Algorithm:

Step 1 - Select the minimum element of the list and swap it with the first element (for Ascending order).

Step 2: In every comparison, if any element is found smaller than the selected element, then both are swapped.

Step 3: Repeat the same procedure with the element in the next position in the list until the entire list is sorted.

Program:

```
# Selection sort in Python
# time complexity O(n*n)
#sorting by finding min_index
def selectionSort(array, size):
    for ind in range(size):
        min_index = ind
        for j in range(ind + 1, size):
            # select the minimum element in every iteration
            if array[j] < array[min_index]:
                min_index = j
         # swapping the elements to sort the array
        (array[ind], array[min_index]) = (array[min_index], array[ind])
```

```
arr = [-2, 45, 0, 11, -9,88,-97,-202,747]
size = len(arr)
selectionSort(arr, size)
print('The array after sorting in Ascending Order by selection sort is:')
print(arr)
```

Output:

```
The array after sorting in Ascending Order by selection sort is:
[-202, -97, -9, -2, 0, 11, 45, 88, 747]
```

Result:

Thus the output is verified.

Ex. No. 9(e) **Insertion Sort**

Date:

Aim:

To write an algorithm and program for Insertion sort.

Algorithm:

STEP 1: Iterate from arr[1] to arr[N] over the array.

STEP 2: Compare the current element (key) to its predecessor.

STEP 3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Program:

```
# Python program for implementation of Insertion Sort

# Function to do insertion sort

def insertionSort(arr):

    if (n := len(arr)) <= 1:

      return

    for i in range(1, n):

        key = arr[i]

        # Move elements of arr[0..i-1], that are

        # greater than key, to one position ahead

        # of their current position

        j = i-1

        while j >=0 and key < arr[j] :

                arr[j+1] = arr[j]

                j -= 1
        arr[j+1] = key
#sorting the array [12, 11, 13, 5, 6] using insertionSort
arr = [12, 11, 13, 5, 6]
insertionSort(arr)
print(arr)
```

Output:

```
Sorted array is:
[5, 6, 11, 12, 13]
```

Result:

Thus the output is verified.

Ex. No. 9(f) **Linear search**

Date:

Aim:

To write an algorithm and program for Linear search.

Algorithm:

STEP 1: Start from the leftmost element of given arr[] and one by one compare element x with each element of arr[]

STEP 2: If x matches with any of the element, return the index value.

STEP 3: If x doesn’t match with any of elements in arr[] , return -1 or element not found.

Program:

```
def linear_Search(list1, n, key):
    # Searching list1 sequentially
    for i in range(0, n):
        if (list1[i] == key):
            return i
    return -1
list1 = [1 ,3, 5, 4, 7, 9]
key = 7
n = len(list1)
res = linear_Search(list1, n, key)
if(res == -1):
    print("Element not found")
else:
    print("Element found at index: ", res)
```

Output:

```
Element found at index:  4
```

Result:

Thus the output is verified.

Ex. No. 9(g) **Binary search**

Date:

Aim:

To write an algorithm and program for Binary search.

Algorithm:

```
STEP 1: Sort the array in ascending order.
STEP 2: Set the low index to the first element of the array and the high index to the last element.
STEP 3: Set the middle index to the average of the low and high indices.
STEP 4: If the element at the middle index is the target element, return the middle index.
STEP 5: If the target element is less than the element at the middle index, set the high index to the middle index – 1.
STEP 6: If the target element is greater than the element at the middle index, set the low index to the middle index + 1.
STEP 7: Repeat steps 3-6 until the element is found or it is clear that the element is not present in the array.
```

Program:

```
# Iterative Binary Search Function method Python Implementation
# It returns index of n in given list1 if present,
# else returns -1
def binary_search(list1, n):
    low = 0
    high = len(list1) - 1
    mid = 0
    while low <= high:
        # for get integer result
        mid = (high + low) // 2
        # Check if n is present at mid
        if list1[mid] < n:
            low = mid + 1
        # If n is greater, compare to the right of mid
        elif list1[mid] > n:
            high = mid - 1
        # If n is smaller, compared to the left of mid
        else:
            return mid
            # element was not present in the list, return -1
    return -1
# Initial list1
list1 = [12, 24, 32, 39, 45, 50, 54]
n = 45
# Function call
result = binary_search(list1, n)
if result != -1:
    print("Element is present at index", str(result))
else:
    print("Element is not present in list1")
```

Output:

```
Element is present at index 4
```

Result:

Thus the output is verified.

Ex. No.10 **Hash table**

Date:

Aim:

To write an algorithm and program for Hash Table.

Algorithm:

Step 1: Is to generate a hash function that converts the search key to an array index.

Step 2: Various keys should ideally correspond to different indices.

Step 3:Because this ideal is typically out of reach, we must accept the chance that two or more distinct keys hash to the same array index.

Step 4: As a result, the second component of a hashing search is a collision-resolution procedure that handles this issue. Collision Resolution

Program:

```
# Python program to demonstrate working of HashTable
hashTable = [[],] * 10
def checkPrime(n):
    if n == 1 or n == 0:
        return 0
    for i in range(2, n//2):
        if n % i == 0:
            return 0
    return 1
def getPrime(n):
    if n % 2 == 0:
        n = n + 1
    while not checkPrime(n):
        n += 2
    return n
def hashFunction(key):
    capacity = getPrime(10)
    return key % capacity
def insertData(key, data):
    index = hashFunction(key)
    hashTable[index] = [key, data]
def removeData(key):
    index = hashFunction(key)
    hashTable[index] = 0
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'], [], [], [], []]
[[], [], 0, [432, 'mango'], [213, 'banana'], [654, 'guava'], [], [], [], []]
```

Result:

Thus the output is verified.

Ex. No.11 **Binary tree traversal**

Date:

Aim:

To write an algorithm and program for binary tree traversal.

Algorithm:

Preorder:

```
Step 1 - Visit the root node
Step 2 - Traverse the left subtree recursively.
Step 3 - Traverse the right subtree recursively.
```

Postorder:

```
Step 1 - Traverse the left subtree recursively.
Step 2 - Traverse the right subtree recursively.
Step 3 - Visit the root node.
```

Inorder:

```
Step 1 - Traverse the left subtree recursively.
Step 2 - Visit the root node.
Step 3 - Traverse the right subtree recursively.
```

Program:

```
# Tree traversal in Python
class Node:
    def __init__(self, item):
        self.left = None
        self.right = None
        self.val = item
def inorder(root):
    if root:
        # Traverse left
        inorder(root.left)
        # Traverse root
        print(str(root.val) + "->", end='')
        # Traverse right
        inorder(root.right)
def postorder(root):
    if root:
        # Traverse left
        postorder(root.left)
        # Traverse right
        postorder(root.right)
        # Traverse root
        print(str(root.val) + "->", end='')
def preorder(root):
    if root:
        # Traverse root
        print(str(root.val) + "->", end='')
        # Traverse left
        preorder(root.left)
        # Traverse right
    preorder(root.right)
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
print("Inorder traversal ")
inorder(root)
print("\nPreorder traversal ")
preorder(root)
print("\nPostorder traversal ")
postorder(root)
```

Output:

```
Inorder traversal
4->2->5->1->3->
Preorder traversal
1->2->4->5->3->
Postorder traversal
4->5->2->3->1->
```

Result:

Thus the output is verified.

Graph traversal

Ex. No.12(a) **Breadth First Traversal**

Date:

Aim:

To write an algorithm and program for Breadth first traversal.

Algorithm:

```
Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until QUEUE is empty
Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).
Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set
their STATUS = 2
(waiting state)
[END OF LOOP]
Step 6: EXIT
```

Program:

```
# BFS algorithm in Python
import collections
# BFS algorithm
def bfs(graph, root):
    visited, queue = set(), collections.deque([root])
    visited.add(root)
    while queue:
        # Dequeue a vertex from queue
        vertex = queue.popleft()
        print(str(vertex) + " ", end="")
        # If not visited, mark it as visited, and
        # enqueue it
        for neighbour in graph[vertex]:
            if neighbour not in visited:
                visited.add(neighbour)
                queue.append(neighbour)
if __name__ == '__main__':
    graph = {0: [1, 2], 1: [2], 2: [3], 3: [1, 2]}
    print("Following is Breadth First Traversal: ")
    bfs(graph, 0)
```

Output:

```
Following is Breadth First Traversal:
0 1 2 3
```

Result:

Thus the output is verified.

Ex. No.12(b) **Depth First traversal**

Date:

Aim:

To write an algorithm and program for Depth first traversal.

Algorithm:

```
Step 1: SET STATUS = 1 (ready state) for each node in G
Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)
Step 3: Repeat Steps 4 and 5 until STACK is empty
Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)
Step 5: Push on the stack all the neighbors of N that are in the ready state (whose STATUS
= 1) and set their STATUS = 2 (waiting state)
[END OF LOOP]
Step 6: EXIT
```

Program:

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start)
    for next in graph[start] - visited:
        dfs(graph, next, visited)
    return visited
graph = {'0': set(['1', '2']),
         '1': set(['0', '3', '4']),
         '2': set(['0']),
         '3': set(['1']),
         '4': set(['2', '3'])}
dfs(graph, '0')
```

Output:

```
0
2
1
4
3
3
```

Result:

Thus the output is verified.

Ex. No.13 **Kruskal's algorithm**

Date:

Aim:

To write an algorithm and program for Kruskal's algorithm

Algorithm:

Step 1: Create a forest F in such a way that every vertex of the graph is a separate tree.

Step 2: Create a set E that contains all the edges of the graph.

Step 3: Repeat Steps 4 and 5 while E is NOT EMPTY and F is not spanning

Step 4: Remove an edge from E with minimum weight

Step 5: IF the edge obtained in Step 4 connects two different trees, then add it to the forest F

(for combining two trees into one tree).

ELSE

Discard the edge

Step 6: END

Program:

```
class Graph:
    def _init_(self, vertices):
        self.V = vertices
        self.graph = []
    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])
    # Search function
    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])
    def apply_union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        else:
            parent[yroot] = xroot
            rank[xroot] += 1
    #  Applying Kruskal algorithm
    def kruskal_algo(self):
        result = []
        i, e = 0, 0
        self.graph = sorted(self.graph, key=lambda item: item[2])
        parent = []
        rank = []
        for node in range(self.V):
            parent.append(node)
            rank.append(0)
        while e < self.V - 1:
            u, v, w = self.graph[i]
            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent, v)
            if x != y:
                e = e + 1
                result.append([u, v, w])
                self.apply_union(parent, rank, x, y)
        for u, v, weight in result:
            print("%d - %d: %d" % (u, v, weight))
g = Graph(6)
g.add_edge(0, 1, 4)
g.add_edge(0, 2, 4)
g.add_edge(1, 2, 2)
g.add_edge(1, 0, 4)
g.add_edge(2, 0, 4)
g.add_edge(2, 1, 2)
g.add_edge(2, 3, 3)
g.add_edge(2, 5, 2)
g.add_edge(2, 4, 4)
g.add_edge(3, 2, 3)
g.add_edge(3, 4, 3)
g.add_edge(4, 2, 4)
g.add_edge(4, 3, 3)
g.add_edge(5, 2, 2)
g.add_edge(5, 4, 3)
g.kruskal_algo()
```

Output:

```
1 - 2: 2

2 - 5: 2

2 - 3: 3

3 - 4: 3

0 - 1: 4
```

Result:

Thus the output is verified.

Ex. No.14 **Dijkstra's Algorithm**

Date:

Aim:

To write an algorithm and program for Dijkstra's algorithm

Algorithm:

Step 1: Mark the source node with a current distance of 0 and the rest with infinity.

Step 2: Set the non-visited node with the smallest current distance as the current node, lets say C.

Step 3: For each neighbour N of the current node C: add the current distance of C with the weight of the edge connecting C-N. If it is smaller than the current distance of N, set it as the new current distance of N.

Step 4: Mark the current node C as visited.

Step 5: Go to step 2 if there are any nodes are unvisited.

Program:

```
import sys
# Providing the graph
vertices = [[0, 0, 1, 1, 0, 0, 0],
            [0, 0, 1, 0, 0, 1, 0],
            [1, 1, 0, 1, 1, 0, 0],
            [1, 0, 1, 0, 0, 0, 1],
            [0, 0, 1, 0, 0, 1, 0],
            [0, 1, 0, 0, 1, 0, 1],
            [0, 0, 0, 1, 0, 1, 0]]
edges = [[0, 0, 1, 2, 0, 0, 0],
          [0, 0, 2, 0, 0, 3, 0],
          [1, 2, 0, 1, 3, 0, 0],
          [2, 0, 1, 0, 0, 0, 1],
          [0, 0, 3, 0, 0, 2, 0],
          [0, 3, 0, 0, 2, 0, 1],
          [0, 0, 0, 1, 0, 1, 0]]
# Find which vertex is to be visited next
def to_be_visited():
    global visited_and_distance
    v = -10
    for index in range(num_of_vertices):
        if visited_and_distance[index][0] == 0 \
            and (v < 0 or visited_and_distance[index][1] <=
                 visited_and_distance[v][1]):
            v = index
    return v
num_of_vertices = len(vertices[0])
visited_and_distance = [[0, 0]]
for i in range(num_of_vertices-1):
    visited_and_distance.append([0, sys.maxsize])
for vertex in range(num_of_vertices):
    # Find next vertex to be visited
    to_visit = to_be_visited()
    for neighbor_index in range(num_of_vertices):
        # Updating new distances
        if vertices[to_visit][neighbor_index] == 1 and \
                visited_and_distance[neighbor_index][0] == 0:
              new_distance = visited_and_distance[to_visit][1] \
                + edges[to_visit][neighbor_index]
              if visited_and_distance[neighbor_index][1] > new_distance:
                visited_and_distance[neighbor_index][1] = new_distance
            visited_and_distance[to_visit][0] = 1
i = 0
# Printing the distance
for distance in visited_and_distance:
    print("Distance of ", chr(ord('a') + i),
          " from source vertex: ", distance[1])
    i = i + 1
```

Output:

```
Distance of  a  from source vertex:  0
Distance of  b  from source vertex:  3
Distance of  c  from source vertex:  1
Distance of  d  from source vertex:  2
Distance of  e  from source vertex:  4
Distance of  f  from source vertex:  4
Distance of  g  from source vertex:  3
```

Result:

Thus the output is verified.