Practical Workbook

CS-218

Data Structures and Algorithms

Name	: ______________________
Year	: ______________________
Batch	: ______________________
Roll No	: ______________________
Department:	______________________

Department of Computer & Information Systems Engineering
NED University of Engineering & Technology

Practical Workbook
CS-218
Data Structures and Algorithms

Prepared by:

Ms. Fauzia Yasir
Ms. Ibshar Ishrat

Revised in:

August 2019

Department of Computer & Information Systems Engineering
NED University of Engineering & Technology

Introduction

This workbook has been compiled to assist the conduct of practical classes for CS-218 Data Structures and Algorithms. Practical work relevant to this course aims at providing students understanding of the concepts of information organization and manipulation in order to pursue advanced courses in this discipline.

The goal of this workbook, therefore, is to present elegant, yet fundamentally simple ideas for the organization and manipulation of data.

The Course Profile of CS-218 Data Structures and Algorithms lays down the following Course Learning Outcome:

“Practice with algorithms for widely used computing operations” (C3, CLO-3)

All lab sessions of this workbook have been designed to assist the achievement of the above CLO. A rubric to evaluate student performance has been provided at the end of the workbook.

Lab sessions 1 & 2 cover basic Memory model and built-data structures along with its operations. Lab session 3 is about binary search algorithm. Lab sessions 4 & 5 discuss matrix operations, storage and retrieval. Lab session 6 deals with Linked list implementation and its operations. Lab sessions 7 & 8 covers stack data structure and its application in processing arithmetic expressions. Lab session 9 demonstrates recursive algorithms. Lab session 10 deals with sorting algorithms on list of elements. Lab session 11 discusses queue data structure. Lab session 12 explains implementation of binary trees. Lab session 13 covers heap along with its operations. Lab session 14 deals with graph and its traversal algorithms.

Contents

Lab Session 01

Explore memory models with example programming language

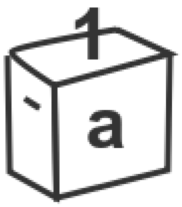
Variables and memory management

In traditional programming languages, variables are often thought of as being named memory locations. Applying that idea to Python, you might think of a variable in Python as a place, corresponding to a location in the computer's memory where you can store an object, but that's not an accurate assumption.

For example, when you do an assignment like the following in C, it actually creates a block of memory space so that it can hold the value for that variable.

```
int a = 1;
```

Think of it as putting the value assigned in a box with the variable name as shown below.



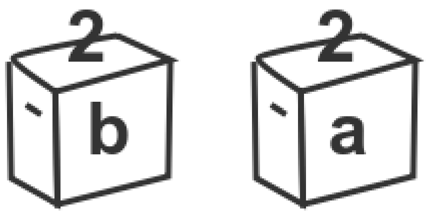
And for all the variables you create a new box is created with the variable name to hold the value. If you change the value of the variable the box will be updated with the new value. That means doing

```
a = 2;
```

will result in:

Assigning one variable to another makes a copy of the value and put that value in the new box.

```
int b = a;
```



In Python variables work like tags. When you do an assignment in Python, it tags the value with the variable name.

```
a = 1
```

and if you change the value of the variable, it just changes the tag to the new value in memory. You don’t need to do the housekeeping job of freeing the memory here. Python's Automatic Garbage Collection does it for you. When a value is without names/tags it is automatically removed from memory.

```
a = 2
```

Assigning one variable to another makes a new tag bound to the same value as show below.

```
b = a
```

Other languages have 'variables'. Python has 'names'.

Python's memory management

As you have seen before, a value will have only one copy in memory and all the variables having this value will refer to this memory location. For example when you have variables `a`, `b`, `c` having a value

10, it doesn't mean that there will be 3 copies of `10`s in memory. There will be only one `10` and all the variables `a`, `b` and `c` will point to this value. Once a variable is updated, say you are doing `a += 1` a new value `11` will be allocated in memory and `a` will be pointing to this.

Let's check this behavior with Python Interpreter. Start the Python Shell and try the following for yourselves.

```
>>> a = 10

>>> b = 10

>>> c = 10

>>> id(a), id(b), id(c)

(140621897573616, 140621897573616, 140621897573616)

>>> a += 1

>>> id(a)

140621897573592
```

`id()` will return an objects memory address (object's identity). As you have noticed, when you assign the same integer value to the variables, we see the same ids. But this assumption does not hold true all the time. See the following for example:

```
>>> x = 500

>>> y = 500

>>> id(x)

4338740848

>>> id(y)

4338741040
```

What happened here? Even after assigning the same integer values to different variable names, we are getting two different ids here. These are actually the effects of CPython optimization we are observing here. CPython implementation keeps an array of integer objects for all integers between -5 and 256. So when we create an integer in that range, they simply back reference to the existing object.

Let's take a look at strings now.

```
>>> s1 = 'hello'
>>> s2 = 'hello'

>>> id(s1), id(s2)

(4454725888, 4454725888)

>>> s1 == s2

True

>>> s1 is s2

True

>>> s3 = 'hello, world!'

>>> s4 = 'hello, world!'

>>> id(s3), id(s4)

(4454721608, 4454721664)

>>> s3 == s4

True

>>> s3 is s4

False
```

When the string was a simple and shorter one, the variable names where referring to the same object in memory. But when they became bigger, this was not the case. This is called interning, and Python does interning (to some extent) of shorter string literals (as in `s1` and `s2`) which are created at compile time. But in general, Python string literals create a new string object each time (as in `s3`and `s4`). Interning is runtime dependent and is always a trade-off between memory use and the cost of checking if you are creating the same string. There's a built-in `intern()` function to forcefully apply interning.

Now we will try to create custom objects and try to find their identities.

```
>>> class Foo:

...     pass

...

>>> bar = Foo()

>>> baz = Foo()
>>> id(bar)

140730612513248

>>> id(baz)

140730612513320
```

As you can see, the two instances have different identities. That means, there are two different copies of the same object in memory. When you are creating custom objects, they will have unique identities unless you are using Singleton Pattern which overrides this behavior (in `__new__()`) by giving out the same instance upon instance creation.

Exercise

1. For the given program, find the memory address(es) of the variable *n* for 5 iterations. Note down the observations.

```
#print in triangle
x=int(input("enter the number"))
for n in range(0,x):
    n +=1
    print ("*" *(0+n))

for n in range(-x,0):
    n +=1
    print ("*" *(0-n+1))
```

____

____

____

____

____

2. Write a Python script to concatenate following dictionaries to create a new one. Find the addresses of the three dictionaries and the concatenated dictionary as well. Note you're your observations. Can you find the address of individual key-value pair?
   Sample Dictionary :

```
dic1={1:10, 2:20}
dic2={3:30, 4:40}
dic3={5:50,6:60}
```

   Expected Result : `{1: 10, 2: 20, 3: 30, 4: 40, 5: 50, 6: 60}`

Write the program here. Attach the printout of output

3. Write down a python script to remove duplicates from the list. Note down the addresses of list before and after the removal of duplicates.

Write program here. Attach the printout of output

4. Write a Python program to count the elements in a list until an element is a tuple. Note down the addresses of the tuple as well as the list.

Write program here. Attach the printout of output

Lab Session 02

Practice following operations with built-in data structures
i. Insertion
ii. Deletion
Iii. Searching

Data structures

Data structures are used to store a collection of related data.
There are four built-in data structures in Python – list, tuple, dictionary and set. In this lab session, insertion, deletion and searching algorithms would be covered for list. Other data structures are left to be explored in the exercise section.

ALGORITHMS FOR LIST
List is a type of container that is used to store multiple data at the same time. The elements in a list are ordered and can be heterogeneous. Lists are mutable objects. The elements are indexed according to a definite sequence.

Algorithm A2.1: ins(data, pos)
This algorithm inserts the data at the position specified in the argument.

1. Slice a list from index 0 till pos. Call it list1.
2. The list from pos till length of the initial list will be list 2.
3. Concatenate the element or sub-list with list 1.
4. Concatenate the list obtained from step 3 with list 2.
5. Print the final list

Algorithm A2.2: dele(data)
This algorithm removes the data if available in the list.
1. For `i` in the range of length of the list, check whether `data` is there at any index position.
2. If there is a match, slice the list from 0 till index value `i,` call it list 1.
3. The list from `i+1` till length of the initial list will be list 2.
4. Concatenate the list obtained from step 2 with list 2.
5. Print the final list.

Algorithm A2.3: search(data)
This algorithm finds the data in the list and returns the position of the data value.
1. For `i` in the range of length of the list, check whether `data` is there at any index position.
2. If there is a match, return the index value `i.`
3. Print the value of the index.

Exercise

1. Implement algorithms A2.1 to A2.3 in the form of functions. Compare the implemented functions with built-in functions available for list data structure in terms of execution time. Note down your observations.
   Hint: Use `timeit` *module.*

Write program here. Attach printout of output

Write program here. Attach printout of output

2. Develop algorithms for insertion, deletion and searching for dictionary and tuple data structure. Implement all the algorithms and compare them with built-in functions available for respective operations in terms of execution time. Note down your observations.

Write program here. Attach printout of output

Write program here. Attach printout of output

Lab Session 03

Implement binary search algorithm

Binary search

Suppose DATA is an array that is sorted in increasing (or decreasing) numerical order or, equivalently, alphabetically. Then there is an extremely efficient searching algorithm, called *binary search*, which can be used to find the location LOC of a given ITEM of information in DATA.

The binary search algorithm applied to our array DATA works as follows: During each stage of our algorithm, our search for ITEM is reduced to a segment of elements of DATA:

DATA[BEG], DATA[BEG+1], DATA[BEG+2], . . ., DATA[END]

Note that the variables BEG and END denote, respectively, the beginning and end locations of the segment under consideration. The algorithm compares ITEM with the middle element DATA[MID] of the segment, where MID is computed as

MID = INT((BEG + END) / 2) (INT(A) refers to the integer value of A.)

If DATA[MID] = = ITEM, then the search is successful and we set LOC = MID.

Otherwise a new segment of DATA is obtained as follows:

a) If ITEM < DATA[MID], then ITEM can appear only in the left half of the segment: DATA[BEG], DATA[BEG+1], . . . , DATA[MID-1]

   So we reset END = MID –1 and begin searching again.

b) If ITEM > DATA[MID], then ITEM can appear only in the right half of the segment: DATA[MID+1], DATA[MID+2], . . . , DATA[END]

   So we reset BEG = MID +1 and begin searching again.

Initially, we begin with the entire array DATA; i.e., we begin with BEG = 0 and END = N-1.

If ITEM is not in DATA, then eventually we obtain BEG > END.

This condition signals that the search is unsuccessful, and in such a case we assign LOC = NULL. Here NULL is a value that lies outside the set of indices of DATA.

Algorithm

Algorithm A3.1: BINSEARCH(DATA, ITEM)

```
1. Set BEG = 0, END = N-1 and MID = INT((BEG + END) / 2)
2. Repeat steps 3 and 4 while BEG <= END AND DATA[MID] ≠ ITEM
3.     If ITEM < DATA[MID], then Set END = MID – 1, Else Set BEG = MID + 1
4.     MID = INT((BEG + END) / 2)
5. If DATA[MID] = ITEM, then Set LOC = MID Else Set LOC = NULL
6. Exit
```

Exercises

a) Write a program in Python that makes use of the above implementation as a function. Your program should input the array from the user. Also compare the execution time of function with built-in search function available and write down the results.

<u>Write program here. Attach printout of output</u>

b) Analyze the *binary search* algorithm for the best, average and worst cases. Express your results in Big O notation.

c) Construct a function to take input of an array from the user and prompt the user if he enters unsorted data. Modify the *binary search* algorithm presented here so as to insert the element in the array if the search remains unsuccessful. Make sure that the array remains sorted after the insertion. Implement the revised algorithm in Python.

Write the program here. Attach screenshot of output.

Write program here. Attach printout of output

Lab Session 04

Implement matrix operations using multidimensional arrays

Matrix

We consider two input matrices A and B of order nxn. We need to compute $C = AxB$, where C is of order nxn.

If A is of order $r_A x c_A$, and B is of order $r_B x c_B$, then multiplication is possible only if $c_A = r_B$. $C = AxB$ is of order $r_A x c_B$.

Algorithm

Algorithm A4.1: MATMUL(A,B)

Here A and B both are of order nxn.

```
1.    For i = 0 to n-1
2.      For j = 0 to n-1
3.    C[i][j] = 0
4.          For k = 0 to n-1
5.          C[i][j] = C[i][j] +A[i][k]*B[k][j]
6.          End for
7.      End for
8.    End for
9.    Return
```

Exercises

a) Implement the above algorithm in Python.

b) Considering A is of order $r_A x c_A$, and B is of order $r_B x c_B$, revise the above algorithm and implement it in Python. $c_A = r_B$, but r_A, r_B and c_B can be different quantities.

d) Analyze the algorithm MATMUL() and express your result in Big O notation.

Lab Session 05

Explore sparse matrices and their applications

Sparse matrices

Matrices with a high proportion of zero values are called sparse matrices. Sparse matrices can be useful for computing large-scale applications that dense matrices cannot handle. A sparse matrix data structure is described and used in subsequent algorithms for the matrix exponential, linear simulation, and frequency response matrix calculations. The sparse matrix techniques require no special computer hardware such as aprallel or vector processors. The algorithms are applied to several typical aircraft flight control system state models. Two general forms are given below. Zero entries are left empty.

$$A = \begin{bmatrix} 4 & & & \\ 3 & -5 & & \\ 1 & 6 & 2 & \\ 8 & 0 & 5 & 9 \end{bmatrix} \qquad B = \begin{bmatrix} 5 & -7 & & \\ 1 & 4 & 3 & \\ & 9 & -3 & 6 \\ & & 2 & 4 \end{bmatrix}$$

Triangular Matrix Tridiagonal Matrix

In triangular matrix, all entries on and below the main diagonal are non-zero. In tridiagonal matrix, all entries on the main diagonal and entries immediately above and immediately below the main diagonal are non-zero.

To conserve space, we store a sparse matrix in a uni-dimensional array, omitting zero entries.

Consider the triangular matrix A. It has 1 non-zero entry in row 0, 2 non-zero entries in row 1, 3 non-zero entries in row 2, and n non-zero entries in row n-1. Total non-zero entries will determine the size of the uni-dimensional array U.

Size of U = 1 + 2 + 3 + … + n = ½ * n * (n+1)

U[0] = A[0][0], U[1] = A[1][0], U[2] = A[1][1], U[3] = A[2][0] and so on.

To retrieve A[J][K], we assume, U[L] = A[J][K]. L represents the number of elements in U up to and including A[J][K]. In the rows before row J, there are 1 + 2 + 3 + … + J = ½ * J * (J+1) non-zero numbers. In row J, there are K+1 elements upto and including A[J][K]. Since the array U has starting index 0, we need to subtract 1 from L.

L = ½ * J * (J+1) + K + 1 – 1 = ½ * J * (J+1) + K

So, U[½ * J * (J+1) + K] = A[J][K]

Algorithms

A is a triangular sparse matrix of order nxn. U is a uni-dimensional array of size ½*n*(n+1).

Algorithm A5.1: STORETRIANGULAR(A)

```
1.  i=0
```

```
2. For j = 0 to n-1
3.     For k = 0 to j
4.     U[i] = A[j][k]
5.     i++
6.     End for
7. End for
```

Algorithm A5.2: RETRIEVETRIANGULAR(U)

```
1. For j = 0 to n-1
2.     For k = 0 to n-1
3.     If k > j, A[j][k] = 0
4.     Else A[j][k] = U[½*j*(j+1)+k]
5.     End for
6. End for
```

Exercises

a) Implement the algorithms A5.1 and A5.2 in Python.

b) Analyze the given algorithms and express your result in Big O notation.

c) Find the size of U for a tridiagonal matrix B. Find the formula for L if U[L] = B[J][K].

d) Develop an algorithm to store non-zero entries of B in U.

e) Develop an algorithm to retrieve B from U.

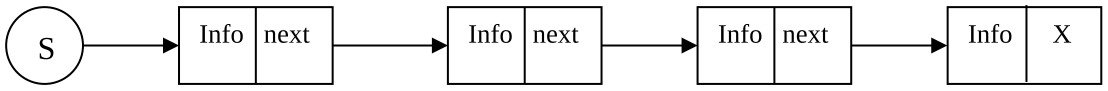
f) SciPy. (pronounced "Sigh Pie") is a free and open-source Python library used for scientific computing and technical computing. SciPy provides tools for creating sparse matrices using multiple data structures, as well as tools for converting a dense matrix to a sparse matrix. Write a program to define a 3 x 6 sparse matrix as a dense array, convert it to a CSR (Compressed sparse row) sparse representation, and then convert it back to a dense array.

Lab Session 06

Implement linked list and practice following operations:
i. Insertion
ii. Deletion
iii. Searching.

Linked list

A linked list consists of a set of nodes, each of which has two fields: an information field and a reference to the next node in the list.



Linked implementations of a list can provide more efficient insertion and deletion of items than an array implementation, but they lack random access and require more memory. Such implementations are typically more difficult to write correctly than array implementations since the programmer has to keep careful track of the necessary references.

In Python, `Node` class (given below) can be used to create the nodes in a linked list.

```
class Node:
  def __init__(self, item, next=None):
       self.item = item
       self.next = next

  def getItem(self):
       return self.item

  def getNext(self):
       return self.next

  def setItem(self, item):
      self.item = item

  def setNext(self, next):
       self.next = next
```

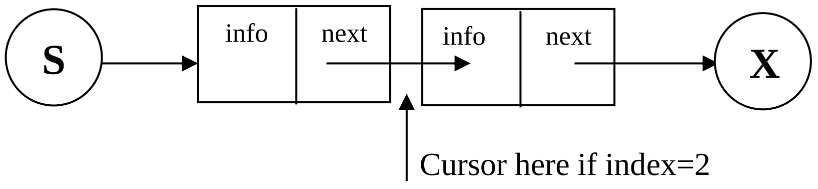
In the `Node` class there are two pieces of information: the item is a reference to a value in the list, and the next reference which points to the next node in the sequence.

Algorithms

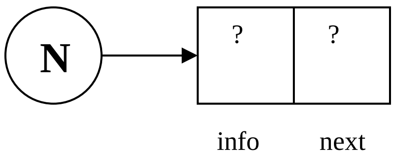
Linked List Creation
Create a linked list (object of class LinkedList) with reference to the first node in the linked list and the last item in the linked list. They both point to a head node to begin with. This node will always be in the first position in the list and will never contain an item.

Algorithm A6.1: Inserting a new node according to index position

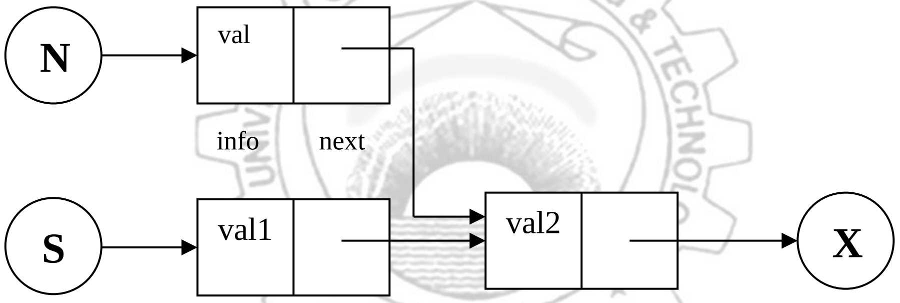
1. Declare a cursor variable that points to the first node initially and index variable indicating the position.
2. If index is less than the total number of elements in the list (nodes), then move the cursor till the reference pointing to the node at index position and perform steps 3 to 5, else perform step 3 only and insert the element at the end of list.



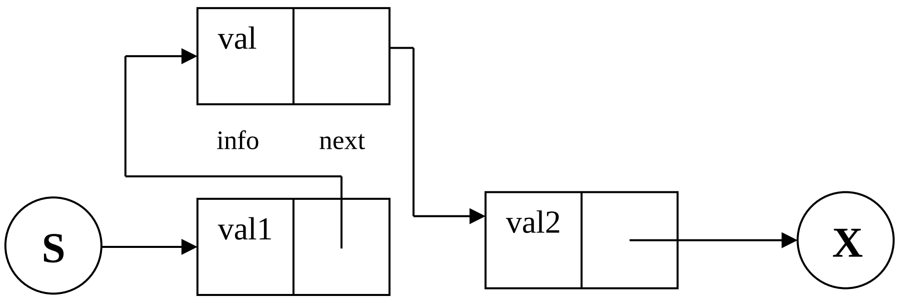
1. Create a new node object with the desired value (info) and reference to the next node.



2. Set the reference of the newly allocated node to point to the next variable of the node after the index value.



3. Change the link field of the node before the index value to point to the newly created node. Increment the total number of nodes in the list.



Algorithm A6.2: Searching for an item

1. Declare a cursor variable, N that is initially set to point to the first node in the list.
2. While (N points to a non-None node of the list)
   If value in the info field of N's referent equals the item to be searched, return the referred node in the cursor.
   Else advance the cursor variable to point to the next node in the list.
3. Return N's value, None, as the result of list search if item not found.

Algorithm A6.3: Deleting the node on the basis of match

After performing the steps mentioned in the above algorithm, follow the steps given below.

1. If the list has exactly one node, load None in the next variable of first node and return. Else declare and initialize precursor and cursor variables to point to the node to be deleted and the node previous to it respectively.
2. For deleting the desired node advance the cursor variable to point to the node next to the one to be deleted. Initialize the precursor variable to point to the node following the deleted one.

Exercises

a) Implement the linked list creation (as described under the heading of algorithm) in Python.

b) Implement all the algorithms as separate procedures and demonstrate their use in a program.

c) A doubly linked list is one in which there are two pointers (next variables) in a node, one pointing to the next node and the other pointing to the previous node. Design all above algorithms for a doubly linked list and use them in a program.

Lab Session 07

Implement stack operations

Stack

A stack is a list of elements in which an element may be inserted or deleted only at one end, called the top of stack, denoted TOS. This means, in particular, that elements are removed from a stack in the reverse order of that in which they were inserted into the stack. For this reason stack is regarded as *Last-In, First-Out* (LIFO) structure.

Special terminology is used for two basic operations associated with stacks:
a. “Push” is the term used to insert an element into a stack.
b. “Pop” is the term used to delete an element into a stack.

Suppose we push five elements a, b, c, d, e, f, in order, onto an empty stack. Thus the stack will be represented as follows:

STACK: a, b, c, d, e, f

The TOS will point to the f indicating that it is the most recently inserted item onto stack. Since the stack can’t be accessed anywhere except the TOS, therefore e can’t be removed before f.

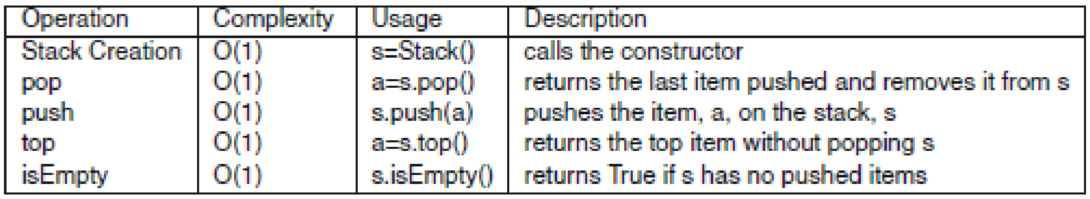
The key point in using stacks is that they are useful in the situations wherever decisions are postponed. For example,

- Keeping track of return addresses in case of procedure calls and returns. This feature is of course very valuable for interrupt processing.
- Implementing Recursion (to be discussed in a subsequent lab)
- Processing arithmetic expressions (to be discussed in a subsequent lab)

Algorithms

Before developing algorithms for stack operations, we must decide on stack implementation i.e., we should first answer the question how to represent a stack in the computer’s memory.

A Stack class can be implemented in at least a couple different ways to achieve the computation complexities outlined in this table. Either a list or a linked list will suffice.



Operation	Complexity	Usage	Description
Stack Creation	O(1)	s=Stack()	calls the constructor
pop	O(1)	a=s.pop()	returns the last item pushed and removes it from s
push	O(1)	s.push(a)	pushes the item, a, on the stack, s
top	O(1)	a=s.top()	returns the top item without popping s
isEmpty	O(1)	s.isEmpty()	returns True if s has no pushed items

Algorithm A7.1: PUSH (STACK, TOS, MAXSTK, ITEM)

This algorithm pushes an ITEM onto a stack.

```
1.  /* Check for OVERFLOW. An overflow is said to occur when an attempt is made to insert
    onto a stack when it already contains maximum number of elements */
    if (TOS = = MAXSTK – 1) then PRINT "OVERFLOW" and return.
2.  Set TOS = TOS + 1.
3.  Set STACK[TOS] = ITEM.
4.  Return.
```

Algorithm A7.2: POP (STACK, TOS, ITEM)

This algorithm deletes the top element of STACK and assigns it to the variable ITEM

```
1. /* Check for UNDERFLOW. An underflow is said to occur when an attempt is made to
   delete from an empty stack */
   if (TOS = = -1) then PRINT "UNDERFLOW"  and return.
2. Set ITEM = STACK[TOS].
3. Set TOS = TOS –1.
4. Return.
```

Exercises

a) Construct the stack class and implement the algorithms A7.1 and A7.2 for stack operations in Python.

In the exercises (b) to (e), show the result of operation on the stack. If overflow or underflow occurs, check the appropriate box; otherwise, show the new contents of the array, the TOS, and ITEM. (*Note:* Some values in the array may not be elements in the stack.)

b) PUSH (STACK, ITEM)

STACK

C	A	M	E	L
0	1	2	3	4

TOS = 2
ITEM = ‘K’

Overflow? Underflow?

STACK

0	1	2	3	4

TOS = ____
ITEM = ______

c) PUSH (STACK, ITEM)

STACK

M	A	N	G	O
0	1	2	3	4

TOS = 4
ITEM = ‘X’

Overflow? Underflow?

STACK

0	1	2	3	4

TOS = ____
ITEM = ______

d) POP (STACK, ITEM)

STACK

L	E	M	O	N
0	1	2	3	4

TOS = 0
ITEM = ‘X’

Overflow? Underflow?

STACK

0	1	2	3	4

TOS = ____
ITEM = ______

e) POP (STACK, ITEM)

STACK

F	E	T	C	H
0	1	2	3	4

TOS = 4
ITEM = ‘K’

Overflow? Underflow?

STACK

0	1	2	3	4

TOS = ____
ITEM = ______

f) Show what is printed by the following segment of code.

```
ClearStack(STACK)          # Initializes STACK as empty
X = 4
Z = 0
Y = X + 1
PUSH (STACK, Y)
PUSH (STACK, Y + 1)
PUSH (STACK, X)
POP (STACK, Y)
X = Y + 1
PUSH (STACK, X)
PUSH (STACK, Z)
‘’’ empty ( ) is a function to test for empty stack ‘’’
while (!EMPTY (STACK)):
      POP (STACK, Z)
      print (Z)

print (“X = “, X)
print (“Y = “, Y)
print (“Z = “, Z)
```

g) Demonstrate the use of stack class and its operations.

Lab Session 08

Use stack data structure for processing arithmetic expressions.

Arithmetic notations

The way we are used to seeing arithmetic expressions is called **infix** notation – the operator is **in** between the operands. Infix notation can be fully parenthesized, or it can rely on a scheme of operator precedence, as well as the use of parentheses to override the rules, to express the order of evaluation within an expression. For instance, the operators x and / usually have a higher precedence over the operators + and –.

For example, a + b * c is sufficient to express that multiplication should be performed first. However, if we want to do the addition before multiplication, we must indicate this with the parentheses: (a + b) * c.

The problem with the infix notation is its ambiguity. We must resort to an agreed-upon scheme to determine how to evaluate the expression. There are other ways of writing expressions, however, that do not require parentheses or precedence schemes. Such two schemes are **postfix** and **prefix** notations.

- POSTFIX NOTATION

It refers to the notation in which the operator symbol is placed **post** (i.e., *after*) its operands. For example:

Infix	**Postfix**
A + B	AB+
A - B	AB-
A * B	AB*
A / B	AB/

We translate, step by step, the following infix expressions into postfix notation using brackets [ ] to indicate a partial translation:

(a + b) * c = [ab+] * c = ab+c*
a + (b * c) = a + [bc*] = abc*+

The fundamental property of postfix notation is that the order in which the operations are to be performed is completely determined by the positions of the operators and operands in the expression. Accordingly, one never needs parentheses when writing expressions in postfix notation.

- PREFIX NOTATION

It refers to the analogous notation in which the operator symbol appears **pre** (i.e., *before*) its operands.

Infix	**Prefix**
A + B	+AB
A - B	-AB
A * B	*AB
A / B	/AB

In the honor of its designer, a Polish mathematician Jan Lukasiewicz, it is referred to as **Polish** notation. It has the same property associated with it as described for postfix notation. (For obvious reason, postfix notation is also termed as reverse polish notation).

Algorithms

Algorithm A8.1: Evaluation of a Postfix Expression

Suppose P is an arithmetic expression written in postfix notation. The following algorithm that uses a stack to hold operands, evaluates the VALUE of P.

1. Add a right parenthesis ")" at the end of P. /* This acts as a sentinel */
2. Scan P from left to right and repeat steps 3 and 4 for each element of P until the sentinel ")" is encountered.
3. If an operand is encountered, push it onto stack.
4. If an operator **op** is encountered then:
   a) remove the two top elements of STACK. (Let T be the top element and NT be the next-to-top element.)
   b) evaluate NT **op** T
   c) place the result of (b) back on STACK.
5. Set VALUE equal to the top element of stack.
6. Exit

Algorithm A8.2: Transforming Infix Expressions into Postfix Expressions

Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

1. Push "(" onto STACK, and add ")" at the end of Q.
2. Scan Q from left to right and repeat steps 3 to 6 for each element of Q until the STACK is empty:
3. If an operand is encountered, add it to P.
4. If a left parenthesis is encountered, push it onto STACK.
5. If an operator **op** is encountered, then:
   a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same precedence as or higher precedence than **op**.
   b) Add **op** to STACK.
6. If a right parenthesis is encountered, then:
   a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a left parenthesis is encountered.
   b) Remove the left parenthesis. [Do not add the left parenthesis to P]
7. Exit

Algorithm A8.3: Evaluate Infix Expressions

Suppose Q is an arithmetic expression written in infix notation. This algorithm uses two stacks an operator stack (to hold operators and left parenthesis) and an operand stack (to hold numbers) then evaluates the VALUE of Q.

1. Push "(" onto operator STACK.
2. Scan Q from left to right and repeat steps 3 and 4 until operator stack is empty.
3. If an operand is encountered add it to operand stack.
4. If an operator is encountered call OPERATE procedure.
5. Set VALUE equal to the top element of the operand stack.

Algorithm A8.3.1: OPERATE(operator, operator stack, operand stack):

1. If the given operator is a left parenthesis we push it on the operator stack and return.
2. Else, while the precedence of the given operator is less than or equal to the

precedence of the top operator on the operator stack proceed as follows:

  a. Pop the top operator from the operator stack. Denote this by the variable *topOp*.
  b. If *topOp* is a +,–, ∗, or / then operate on the number stack by popping the operands, doing the operation, and pushing the result.
  c. Else if *topOp* is a left parenthesis then the given operator should be a right parenthesis and return.

3. Push the given operator on operator stack and return.

Exercises

a) Implement the algorithms A8.1, A8.2 and A8.3 in Python.

b) Compute the value of following postfix expressions using the algorithm A1.

(i) 1 4 18 6 / 3 + + 5 / +

Symbol Scanned	STACK
1	
4	
18	
6	
/	
3	
+	
+	
5	
/	
+	

(ii) 3 1 + 2 ^ 7 4 – 2 * + 5 –

Symbol Scanned	STACK
3	
1	
+	
2	
^	
7	
4	
–	
2	
*	
+	
5	
–	

c) Translate the following infix expression into its equivalent postfix expression using the algorithm A2.

((a + b) / c) ^ ((d – e) * f)

Symbol Scanned	Stack	Postfix P
(		
(		
a		
+		
b		
)		
/		
c		
)		
^		
(		
(		
d		

–		
e		
)		
*		
f		
)		

d) Compute the value of following infix expression using the algorithm A3.

(i) (6 + 5) * 4 - 9

Symbol Scanned	OPERATOR STACK	OPERAND STACK
(		
6		
+		
5		
)		
*		
4		
–		
9		

Lab Session 09

Implement recursive algorithms

Recursion

Recursion is an important concept in computer science. Many algorithms can be best described in terms of recursion. Recursion is the name for the case when a procedure P invokes itself or invokes a series of other procedures that eventually invokes the procedure P again. Then P is called a recursive procedure.

In order to make sure that program will not continue to run indefinitely, a recursive procedure must have the following two properties:
1. There must be certain criteria, called 'base criteria', for which the procedure does not call itself.
2. Each time the procedure does call itself (directly or indirectly), it must be closer to the baser criteria.

A recursive procedure with these two properties is said to be 'well-defined'.

- FACTORIAL FUNCTION

The product of the positive integers from 1 to n, inclusive, is called *n factorial* and is usually denoted by n!

$$n! = 1 \times 2 \times 3 \times \ldots (n-2) \times (n-1) \times n$$

We know that 0! = 1. For every positive integer n; n! = n x (n – 1)!

Accordingly, the factorial function may also be defined as follows:
a) If n = 0, then n! = 1.
b) If n > 0, then n! = n x (n – 1)!

Obviously this definition is recursive, since it refers to itself when it uses (n –1)!. However, (a) the value of n! is explicitly given when n = 0 (thus 0 is the base value); and (b) the value of n! for arbitrary n is defined in terms of a smaller value of n which is closer to the base value 0. Accordingly, the definition is not circular, or in other words, the procedure is well-defined.

Suppose P is a recursive procedure. During the running of an algorithm or a program that contains P, we associate a *level number* with each given execution of procedure P as follows. The original execution of procedure P is assigned level 1; and each time procedure P is executed because of a recursive call, its level is 1 more than the level of the execution that has made the recursive call. The *depth* of recursion of a recursive procedure P with a given set of arguments refers to the maximum level number of P during its execution.

Algorithm

This algorithm calculates n! and returns the value in the variable FACT.
Algorithm A9.1: FACTORIAL(FACT, n)

```
1.  if (n = = 0) then set  FACT = 1 and return.
2.  FACT = n x FACTORIAL(FACT, n –1 )
3.  Return.
```

Algorithm A9.2: Quick Sort

```
QUICK(A, N, BEG, END, LOC)
        Here A is an array with N elements. Parameters BEG and END contain the boundary values
        of the sublist of A to which this procedure applies. LOC keeps track of the position of the first
        element A[BEG] of the sublist during the procedure. The local variables LEFT and RIGHT
        will contain the boundary values of the list of elements that have not been scanned.
        1. [Initialize.] Set LEFT:= BEG, RIGHT := END and LOC :=BEG.
        2. [Scan from right to left.]
                (a) Repeat while A[LOC] ≤ A[RIGHT] and LOC ≠ RIGHT:
                        RIGHT := RIGHT - 1,
                        [End of loop.]
                (b) If LOC = RIGHT, then: Return.
                (c) If A[LOC] > [RIGHT], then:
                    (i) [Interchange A[LOC] and A[RIGHT],]
                                Temp := A[LOC], A[LOC] := A[RIGHT],
                                A[RIGHT] := TEMP.
                    (ii) Set LOC := RIGHT.
                    (iii) Go to Step 3.
                    [End of if structure.]
        3. [Scan from left right, ]
                (a) Repeat while A[LEFT] ≤ A[LOC] and LEFT ≠ LOC:
                        LEFT := LEFT + 1,
                        [End of loop.]
                (b) If LOC = LEFT, than: Return.
                (c) If A[LEFT] > A[LOC], then
                    (i) [Interchange A[LEFT] and A[LOC],]
                                Temp := A[LOC], A[LOC] := A[LEFT],
                                A[LEFT] := TEMP.
                    (ii) Set LOC := LEFT.
                    (iii) Go to Step 2.
                    [End of if structure.]
(Quicksort) This algorithm sorts an array A with N elements.
        1. [Initialize. ] TOP := NULL.
        2. [Push boundary values of A onto stacks when A has 2 or more elements.]
           If N > 1, then: TOP := TOP + 1, LOWER[1] := 1, UPPER[1] := N.
        3. Repeate Steps 4, to 7 while TOP ≠ NULL.
        4.      [Pop sublist from stacks.]
                Set BEG := LOWER[TOP], END := UPPER[ TOP],
                TOP := TOP -1.
        5.      Call QUICK(A, N, BEG, END, LOC). [Procedure 6.5.]
        6.      [Push left sublist onto stacks when it has 2 or more elements.]
                If BEG <  LOC -1, then :-
                        TOP := TOP + 1, LOWER[TOP] := BEG,
                        UPPER[TOP] = LOC -1.
        7.      [Push right sublist onto stacks when it has 2 or more elements.]
                If LOC + 1 <  END, then:
                        TOP := TOP + 1, LOWER[TOP] := LOC + 1,
                        UPPER[TOP] := END.
                [End of if structure.]
        8.      Exit
```

Exercises

a) Construct a program in Python to solve the following.
This exercise will demonstrate how recursion can simplify solution of some complicated problems. It addresses the problem of making combinations of a certain size out of a total group of elements. For example, if we have twenty different books to pass out to four students, we can easily see that - to be equitable - we should give each student five books. But how many combinations of five books can be made out of a group of twenty books? There is a mathematical formula to solve this problem. Given that C is the total number of combinations, Group is the total size of the group to pick from, Members is the size of each subgroup, and Group > = Members,

$$C(Group,Members)= \begin{cases} Group & if\ Members = 1 \\ 1 & if\ Members = Group \\ C(Group-1, Members-1) + C(Group-1, Members) & if\ Group > Members > 1 \end{cases}$$

I. Find C (4, 3).

II. Find C (6, 4) and record your observations.

b) Implement in Python, the recursive solution for factorial. Demonstrate its use in a program.

c) Implement the recursive algorithm for the problem *Towers of Hanoi* in Python.

d) Implement the *Ackermann Function* in Python.

$$A(m,n) = \begin{cases} n+1 & m = 0 \\ A(m-1,1) & m > 0, n = 0 \\ A(m-1, A(m, n-1) & m > 0, n > 0 \end{cases}$$

e) Implement the algorithm A9.2 in Python and demonstrate its use for the following lists.

ii. 44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66

iii. D, A, T, A, S, T, R, U, C, T, U, R, E, S

Lab Session 10

Implement algorithms to insert and delete items in queue data structure

Queue

A queue is a linear list of elements in which deletions can take place only at one end, called the *front*, and insertions can take place only at the other end, called the *rear*. Queues are also called first-in first-out (FIFO) lists, since the first element in a queue will be the first element out of the queue. In other words, the order in which elements enter a queue is the order in which they leave. This contrasts with stacks, which are last-in first-out (LIFO) lists.

An important example of a queue in computer science occurs in a timesharing system, in which programs with the same priority form a queue while waiting to be executed.

In general, queues are used in situations where a fairness criterion is to be adopted by rendering services on the basis of order of request.

Array implementation of queues requires a linear array **elements** for containing queue elements, and two integer variables: **front**, to hold the array index of the front element of queue; and **rear**, to hold the array index of the rear element of queue.

Adding an element to a queue is called ENQUEUE operation and removing an element from a queue is called DEQUEUE operation.

Algorithms

Algorithm A10.1: ADDONE (i)

This algorithm increments a number i using modulo arithmetic, so the number returned is always greater than -1 and less than the total size of the array used to hold the queue.

```
1. i=(i+1) mod maxlength
2. Return i
```

Algorithm A10.2: MAKENULL (Q)

This algorithm initializes the queue structure, so that it holds nothing. This structure has two integer memebers, front and rear, and an array member, elements.

```
1. Q.front = 0
2. Q.rear = maxlength-1
```

Algorithm A10.3: EMPTY (Q)

This returns TRUE iff Q does not contain any element.

```
1. If ADDONE (Q.rear) = Q.front return TRUE
2. Return FALSE
```

Algorithm A10.4: FRONT (Q)

This algorithm is a function that returns the first element on queue Q.

```
1. If EMPTY (Q) then write ‘queue is empty’
2. Else return Q.elements[Q.front]
```

Algorithm A10.5: ENQUEUE (x, Q)

This algorithm inserts element x at the end of queue Q.

```
1. If ADDONE(ADDONE(Q.rear)) = Q.front then write ‘queue is full’
2. Else:
   a. Q.rear = ADDONE (Q.rear)
   b. Q.elements [Q.rear] = x
```

Algorithm A10.6: DEQUEUE (var Q: QUEUE);

This algorithm deletes the first element of queue Q.

```
4. If EMPTY (Q) then write ‘queue is empty’
5. Else Q.front = ADDONE (Q.front)
```

Exercises

a) Construct the queue class and implement the above algorithms in Python.

b) Investigate the purpose of algorithm ADDONE. (*Hint*: Consider the case i = maxlength).

c) In the light of your answer to exercise (b), is it just to call Q.elements a **circular array**? Explain.

d) Implement EMPTY, ENQUEUE, DEQUEUE algorithms in Python using linked list.

e) Implement priority queue in Python using
   i. Linked list.
   ii. Arrays

Lab Session 11

Implement sorting algorithms on a list of elements

Insertion sort

Suppose an array A with *n* elements A[1], A[2], …., A[N] is in memory. The insertion sort algorithm scans A from A [1] to A [N], inserting each element A[K] into its proper position in the previously sorted sub array A[1], A[2], …, A[K-1]. That is:

Pass 1. A[1] by itself is trivially sorted.
Pass 2. A[2] is inserted either before or after A[1] so that: A[1], A[2] is sorted.
Pass 3. A[3] is inserted into its proper place in A[1], A[2], that is, before A[1], between A[1] and A[2], or after A[2], so that: A[1], A[2],A[3] is sorted.
Pass 4. A[4] is inserted into its proper place in A[1], A[2], A[3] so that:
A[1], A[2], A[3], A[4] is sorted.
……………………………………………………………………………………...
Pass N. A[N] is inserted into its proper place in A[1],A[2],…,A[N-1] so that:
A[1], A[2],……,A[N] is sorted.

This sorting algorithm is frequently used where *n* is small. There remains only the problem of deciding how to insert A[K] in its proper place in the sorted sub array A[1], A[2],…A[K-1]. This can be accomplished by comparing A[K] with A[K-1], comparing A[K] with A[K-2], comparing A[K] with A[K-3], and so on, until first meeting an element A[J] such that A[J] <= A[K]. Then each of the elements A[K-1], A[K-2],. ….,A[J+1] is moved forward one location, and A[K] is then inserted in the (J+1)th position in the array.

The algorithm is simplified if there always is an element A[J] such that A[J]<=A[K] ; otherwise we must constantly check to see if we are comparing A[K] with A[1]. This condition can be accomplished by introducing a sentinel element A[0] = - ∞ (or a very small number).

Shell sort

Shell sort is a generalization of insertion sort, which exploits the fact that insertion sort works efficiently on input that is already almost sorted. It improves on insertion sort by allowing the comparison and exchange of elements that are far apart. The last step of shell sort is a plain insertion sort, but by then, the array of data is guaranteed to be almost sorted.

The algorithm for shell sort can be defined in two steps:
Step 1: divide the original list into smaller lists.
Step 2: sort individual sub lists using insertion sort.

For dividing the original list into smaller lists, we choose a value *k*, which is known as *increment.* Based on the value of *k,* we split the list into *k* sub lists. For example, if our original list is x[0], x[1], x[2], x[3], x[4],….,x[99] and we choose 5 as the value for increment, *k* then we get the following sub lists :

First _list	= x[0], x[5], x[10], x[15]……x[95]
Second _list	= x[1], x[6], x[11], x[16]……x[96]
Third _list	= x[2], x[7], x[12], x[17]……x[97]
Fourth _list	= x[3], x[8], x[13], x[18]……x[98]
Fifth _list	= x[4], x[9], x[14], x[19]……x[99]

So the ith sub list will contain every kth element of the original list starting from index i–1.

For each iteration, the list is divided and then sorted. If we use the same value of *k*, we will get the same sub lists and every time we will sort the same sub lists, which will not result in the ordered final list. Note that sorting the five sub lists in the above example do not ensure that the full list is sorted. So we need to change the value of *k* for every iteration. Since number of sub lists we get are equal to the value of *k*, so if we decide to reduce the value of *k* after every iteration, we will reduce the number of sub lists also in every iteration. Eventually, when *k* will be set to 1, we will have only one sub lists. Hence we know the termination condition of our algorithm is *k* = 1.

It has been empirically found that larger number of increments gives more efficient results. Also, it is advisable not to use sequences like 1,2,4,8…. or 1,3,6,9…. since they may give almost the same elements in the sub lists to compare and sort which will not result in better performance.

Algorithms

Algorithm A11.1: INSERTION(A, N)
This algorithm sorts the array A with N elements using insertion sort.

```
1.  Set A[0] : = - ∞[Initializes sentinel element]
2.  Repeat steps 3 to 5 for k = 2,3,……N:
3.      Set TEMP : = A[K] and PTR : = K-1
4.      Repeat while TEMP < A[PTR] :
            (a)  Set A [PTR + 1]: = A [PTR].  [Moves element forward.]
            (b)  Set PTR : = PTR – 1
           [End of loop]
5.      Set A[PTR + 1] : = TEMP  [Insert element in proper place]
       [End of Step 2 loop]
6.  Return
```

Algorithm A11.2: SHELL(Num, N, key)
This algorithm sorts the array Num with N elements using Shell sort.

```
1.  Set k: = int (N/2)
2.  While k > 0 do:
    a.  For i from k to N-1, Repeat steps b, c, and e
    b.  Set Key : = Num[i]  and j : = i
    c.  While j = k and Num[j – k] > key, Repeat step d
    d.  Swap Num[j] and Num[j – k]
    e.  Set k : = int ( k/2.2)
3.  Use Insertion sort to sort remaining array of data.
```

Exercises

a) Implement algorithms A11.1 and A11.2 in Python.

b) Analyze algorithm A11.1 for the worst case. Express your results in Big O notation.

c) Analyze algorithm A11.2 for the worst case. Express your results in Big O notation.

Lab Session 12

Implement Binary Tree Algorithms

Trees

Trees are one of the most important data structures. They come in many forms. They provide natural representations for many kinds of data that occur in applications, and they are useful for solving a wide variety of algorithmic problems.

A tree is a collection of elements called *nodes*, one of which is distinguished as a *root*, with a relation ("parenthood") that places a hierarchical structure on the nodes. A node, like an element of a list, can be of any data type.

A structure with a unique starting node (the root), in which each node is capable of having *at most* two child nodes, and in which a unique path exists from the root to every other node is called a **binary tree**.

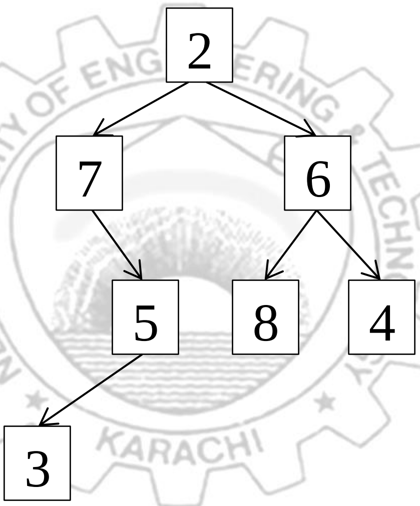


Fig 12.1 Expression Tree (An application of binary tree)

Relationships inside the tree are described using a mixture of family and tree-like terminology. The root node has two children: the node containing 7 is its left child and the one containing 6 is its right child. These two nodes are also said to be siblings. The nodes containing 8 and 4 are also siblings. The parent of node 5 is node 7. Node 3 is a descendant of node 7 and node 7 is an ancestor of node 3. A node that does not have any children is a leaf node. The depth of a node indicates how many edges are between it and the root node. The root node has a depth of zero. Nodes 7 and 6 have a depth of one and node 3 has a depth of three. The height or depth of a tree is the maximum depth of any node.

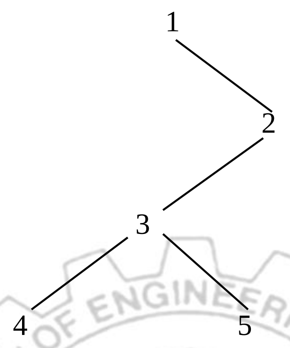
A binary tree has a natural implementation in linked storage. In the implementation to follow, the pointer variable **root** points to the root of the tree. With this pointer variable, it is easy to recognize an empty binary tree as precisely the condition root = NULL, and to create a new, empty binary tree we need only assign its root pointer to NULL.
One of the most important operations on a binary tree is traversal, moving through all the nodes of the binary tree, visiting each one in turn. As for traversal of other data structures, the action we shall take when we visit each node will depend on the application. For lists, the nodes come in a natural order from first to last, and traversal follows the same order. For trees, however, there are many different

orders in which we can traverse all the nodes. Let V, L and R respectively represent visiting root, traversing left subtree, and traversing right subtree. There are three standard traversal orders.

1.	PREorder:	V	L	R
2.	INorder:	L	V	R
3.	POSTorder :	L	R	V

These three names are chosen according to the step at which the given node is visited. With PREorder traversal, the root is visited before the subtrees; with INorder traversal, root is visited between them; and with POSTorder traversal, the root is visited after both of the subtrees.

As an example, consider the following binary tree:

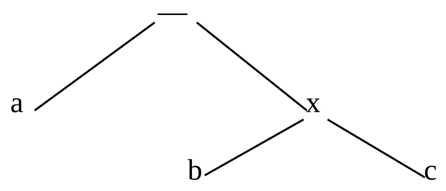


PREorder:	1	2	3	4	5
INorder:	4	3	5	2	1
POSTorder:	4	5	3	2	1

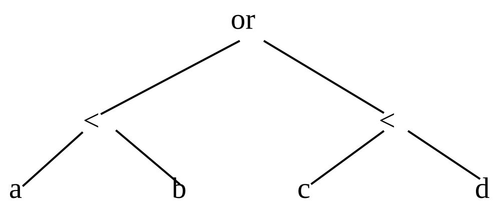
The choice of the names PREorder, INorder, and POSTorder for the three most important traversal methods is not accidental, but relates closely to a motivating example of considerable interest, that of expression trees.

An expression tree is built from simple operators of an (arithmetic or logical) expression by placing operands as the leaves of a binary tree and the operators as the interior nodes. For example,

$\boxed{a - (b \times c)}$



$\boxed{(a < b) \text{ or } (c < d)}$



Algorithms

Declarations
Create a class `TreeNode` to represent the nodes of tree. Each node will have an instance variable to hold a reference to the data of the node and also variables for references to the left and right children. None object will be used for representing empty subtrees.

Using `TreeNode` we can easily create linked structures that directly mirror the binary trees.

For example, here's some code that builds a simple tree with three nodes:
```
left = TreeNode (1)
right = TreeNode (3)
root = TreeNode (2 , left , right)
```

Algorithm A12.1: PREorder Traversal

```
def traverse (tree) :
     if tree is not empty :
          process data at tree 's root  # preorder traversal
          traverse (tree 's left subtree)
          traverse (tree 's right subtree)
```

Applying this algorithm to the tree in Figure 12.1 processes the nodes in the order 2, 7, 5, 3, 6, 8, 4.

Algorithm A12.2: INorder Traversal

```
def traverse (tree) :
     if tree is not empty :
          traverse (tree 's left subtree)
          process data at tree 's root  # inorder traversal
          traverse (tree 's right subtree)
```

Applying this algorithm to the tree in Figure 12.1 processes the nodes in the order 7, 3, 5, 2, 8, 6, 4

Algorithm A12.3: POSTorder Traversal

```
def traverse (tree) :
     if tree is not empty :
          traverse (tree 's left subtree)
          traverse (tree 's right subtree)
          process data at tree 's root  # inorder traversal
```

Applying this algorithm to the tree in Figure 12.1 processes the nodes in the order 3, 5, 7, 8, 4, 6, 2.

Exercises

a) Draw expression tree for the following infix expressions and then give their postfix and prefix expressions.
i) log n!
ii) (A+B) * (C + log (D+E!) – sin (G/H))
iii) $x=(-b \pm \sqrt{}\,(b^2 - 4ac)\,) / 2a$

b) Write a function `TreeSize` that will count all the nodes of a linked binary tree.

c) Write a function `ClearTree` that will traverse a binary tree (in whatever order you find works best) and assign zero at all the nodes.

d) Implement an algorithm to perform a double-order traversal of a binary tree, meaning that each node of the tree, the algorithm first visits the node, then traverses the left subtree (in double order), then visits the node again, then traverses its right subtree (in double order).

Lab Session 13

Implement heaps and practice following operations
i. Insertion
ii. Deletion
iii. Sorting

Heap

A particular kind of binary tree, called a heap, has the following two properties:

1. The value of each node is greater than or equal to the values stored in each of its children.
2. The tree is perfectly balanced, and the leaves in the last level are all in the leftmost positions.

To be exact, these two properties define a max heap. If “greater” in the first property is replaced with “less,” then the definition specifies a min heap. This means that the root of a max heap contains the largest element, whereas the root of a min heap contains the smallest. A tree has the heap property if each non leaf has the first property. Due to the second condition, the number of levels in the tree is O(log n).
Elements in a heap are not perfectly ordered. We know only that the largest element is in the root node and that, for each node, all its descendants are less than or equal to that node.
An important observation is that because a complete binary tree is so regular, it can be represented in an array and no links are necessary. The array in Figure 13.2 corresponds to the heap in Figure 13.1

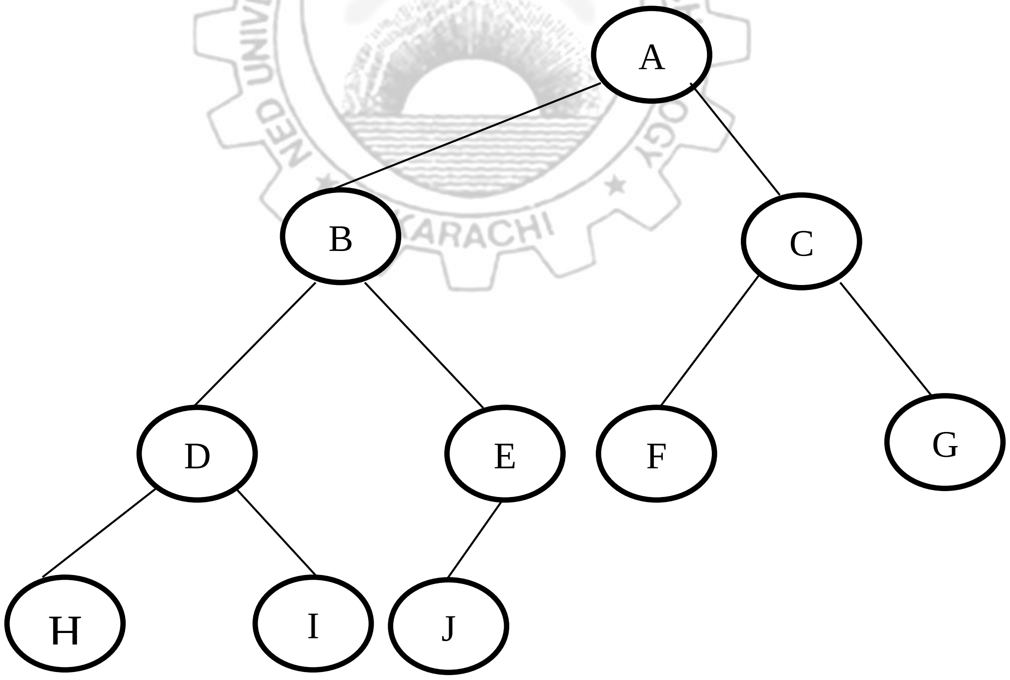


Figure 13.1- A complete binary tree

	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Figure 13.1- Array implementation of complete binary tree

For any element in array position i, the left child is in position 2i, the right child is in the cell after the left child (2i+1), and the parent is in position i/2. Thus, not only are links not required, but the operations required to traverse the tree are extremely simple and likely to be very fast on most computers.

Algorithms

Algorithm A13.1 : HeapInsertion (A[],x)
This algorithm will insert a new value x in a already created heap that is implemented as array A

1. Add a new element x to the end of an array A[]

2. Sift up the new element, while heap property is broken. Sifting is done as following: compare node's value with parent's value. If they are in wrong order, swap them.

Algorithm A13.2: HeapDelete (A[],x)
This algorithm will delete the minimum value in case on Min heap or maximum in case of Max heap which is the root of heap and store in x.

1. Set x:= A[1]
2. Copy the last value in the array to the root;
3. Decrease heap's size by 1;
4. Sift down root's value. Sifting is done as following:

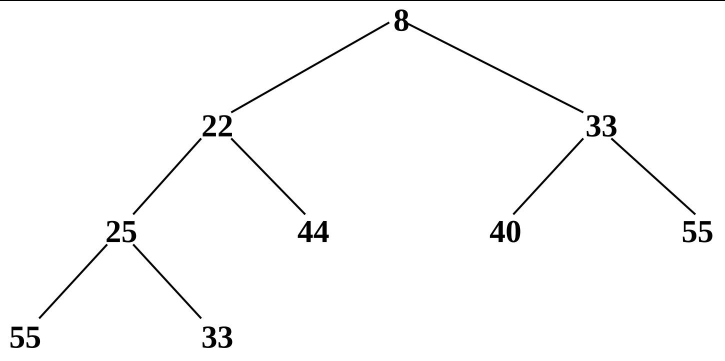
    i. if current node has no children, sifting is over;
    ii. if current node has one child: check, if heap property is broken, then swap current node's value and child value; sift down the child;
    iii. if current node has two children: find the smallest of them. If heap property is broken, then swap current node's value and selected child value; sift down the child.

Algorithm A13.3: HeapSort (A[],B[])
Let B is the list of unsorted elements .Heap sort algorithm is divided into two basic parts:

5. Creating a Heap A of the unsorted list/array B.
6. Then a sorted array is created by repeatedly removing the largest/smallest element from the heap, and inserting it into the array B. The heap is reconstructed after each removal.

Exercises

a) Implement above algorithms in Python.



b)

Consider the given Heap and insert item 11 using relevant algorithm and show result.

c)Test Heapsort program with following data and show result

(a) 44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66
(b) D, A, T, A, S, T, R, U, C, T, U, R, E, S

Lab Session 14

Implement graph and practice its traversal algorithms.

Graphs

Graphs are used to model a wide variety of problems in many different application areas. A graph is a set of **vertices** and the set of **edges** that connect the vertices. The edges can be classified as directed or undirected.

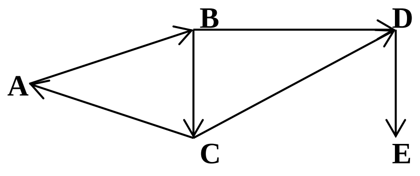


Figure 24.1 Sample Directed Graph

A **directed edge** is a one-way edge and an **undirected edge** is a bidirectional edge. In addition to naming the vertices and edges, attributes can be assigned to the edges and vertices such as a **weight** to an edge.

We will refer to a graph G as the set V of vertices and the set E of edges. Formally,
Mathematicians use the cardinality notation (for example, | V |) to indicate the number of elements in the set.

The term **degree** refers to the number of edges connected to a vertex. For a directed graph, a vertex has both an **in-degree** and an **out-degree** referring to the number of incoming edges and outgoing edges, respectively. Many problems and questions related to graphs require finding paths between vertices. A path from one vertex to another is a sequence of vertices such that there is an edge between each pair of consecutive vertices in the sequence. Most graphs with V vertices have at least V- I edges or the graph is not connected. Formally, a graph is **connected** if for every pair of vertices, there is a path between those two vertices.

In the coming sections adjacency list and its implementation would be discussed. Moreover, two fundamental graph algorithms known as the breadth-first search and depth-first search would be covered.

Adjacency list

The adjacency list of a given vertex V is the list of all adjacent vertices of V. The adjacency list representation is more commonly used since most graphs in real-world applications are sparse.

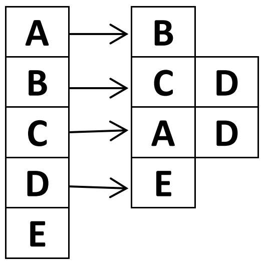


Figure 14.3 Adjacency list for the above graph

Using the built-in list data type, the following example shows a possible Python representation of the graph using a weight of one for each edge. The example also shows how to access the vertices and edges.

```
>>> g = [
...     ['A', [('B', 1)]],
...     ['B', [('C', 1), ('D', 1)]],
...     ['C', [('A', 1), ('D', 1)]],
...     ['D', [('E', 1)]],
...     ['E', []]]
>>> g[0]
['A', [('B', 1)]]
>>> g[0][0]
'A'
>>> g[0][1]
[('B', 1)]
>>> g[1][1]
[('C', 1), ('D', 1)]
```

Python's built-in dictionary is highly optimized and is generally a good choice to use for implementing your own data structures. The common way to implement a graph using a dictionary is to use the vertices as keys with each vertex key mapping to another dictionary that has the adjacent vertices as its keys. In the nested dictionary, each adjacent vertex key maps to the information about the edge (e.g., we could store the weight or a name for the edge as the value). For our sample graph, the dictionary representation and the results of accessing some of the items is the following:

```
>>> g = {
...     'A': {'B': 1},
...     'B': {'C': 1, 'D': 1},
...     'C': {'A': 1, 'D': 1},
...     'D': {'E': 1},
...     'E': {}}
>>> g['A']
{'B': 1}
>>> g['B']
{'C': 1, 'D': 1}
>>> g['B']['D']
1
```

The Python dictionary implementation also makes it easy to iterate over the vertices and over the adjacent vertices for a given vertex as the following code fragment shows.

```
# for each vertex
for v in g:
    print 'vertex', v
    # for each vertex adjacent to v
    for adj in g[v]:
        print adj, g[v][adj]
```

The output of the code fragment is

```
vertex A
B 1
vertex C
A 1
D 1
vertex B
C 1
D 1
vertex E
vertex D
E 1
```

Algorithms

In many problems, we wish to investigate all the vertices in a graph in some systematic order, just as with binary trees. In tree traversal, we had a root vertex with which we generally started; in graph, we do not have any one vertex singled out as special, and therefore the traversal may start at an arbitrary vertex. Although there are many possible orders for visiting the vertices of a graph, two methods are of particular importance.

Algorithm A14.1: Depth-First Traversal

```
dfs(g)
      for each vertex v in graph g :
            set v ' s starting time to 0
      t = 0
      for each vertex v in g :
            if v ' s start time i s 0 :
                  df s_traverse (g , v)

df s_traverse(g , v)
      t += 1
      set v ' s start time to t
      f or each vertex u adjacent to v :
            if u ' s start time i s 0 :
                  set u ' s parent to v
                  dfs_traverse( g , u)
      t += 1
      set v ' s end time to t
```

Algorithm A14.2: Breadth-First Traversal

```
set parent of each vertex to a default value such as None/NULL
set distance f or source vertex to 0
insert source vertex into queue
While queue is not empty
      remove a vertex v from queue
      for each vertex v adjacent to v
```

```
            if w's parent is None/NULL
                  set w's parent to v
                  set w's distance to 1 + v's distance
                  insert vertex w into queue
```

Exercises

a) A graph is **regular** if every vertex has the valence (that is, if it is adjacent to the same number of other vertices). For a regular graph, a good implementation is to keep the vertices in a linked list and the adjacency lists contiguous. The length of all the adjacency lists is called the **degree** of the graph. Write code snippet in Python for this implementation of regular graphs.

b) Write a Python function ReadGraph that will read from the shell the vertices in an undirected graph and lists of adjacent vertices.

c) Write a Python function WriteGraph that will write pertinent information specifying a graph to the shell.

d) Use the functions ReadGraph and WriteGraph to implement and test the ***breadth-first*** traversal algorithm.

e) Use the functions ReadGraph and WriteGraph to implement and test the ***depth-first*** traversal algorithm.

NED University of Engineering & Technology
Department of Computer and Information Systems Engineering

Course Code and Title: CS-218 Data Structures and Algorithms

Laboratory Session No. _____________ Date: ______________________

Software Use Rubric				
Skill Sets	Extent of Achievement			
	0	1	2	3
To what level has the student understood the problem?	The student has not understood the problem at all.	The student understands the problem inadequately.	The student understands the problem adequately.	The student understands the problem comprehensively.
To what extent has the student implemented the solution?	The solution has not been implemented.	The solution has syntactic and logical errors.	The solution has syntactic or logical errors.	The solution is syntactically and logically sound for the stated problem parameters.
How efficient is the proposed solution?	The solution does not address the problem adequately.	The solution exhibits redundancy and partially covers the problem.	The solution exhibits redundancy or partially covers the problem.	The solution is free of redundancy and covers all aspects of the problem.
How did the student answer questions relevant to the task?	The student answered none of the questions.	The student answered less than half of the questions.	The student answered more than half but not all of the questions.	The student answered all the questions.

Weighted CLO Score	
Remarks	
Instructor's Signature with Date	

NED University of Engineering & Technology
Department of Computer and Information Systems Engineering

Course Code and Title: CS-218 Data Structures and Algorithms

Laboratory Session No. _____________ Date: ______________________

Software Use Rubric				
Skill Sets	Extent of Achievement			
	0	1	2	3
To what level has the student understood the problem?	The student has not understood the problem at all.	The student understands the problem inadequately.	The student understands the problem adequately.	The student understands the problem comprehensively.
To what extent has the student implemented the solution?	The solution has not been implemented.	The solution has syntactic and logical errors.	The solution has syntactic or logical errors.	The solution is syntactically and logically sound for the stated problem parameters.
How efficient is the proposed solution?	The solution does not address the problem adequately.	The solution exhibits redundancy and partially covers the problem.	The solution exhibits redundancy or partially covers the problem.	The solution is free of redundancy and covers all aspects of the problem.
How did the student answer questions relevant to the task?	The student answered none of the questions.	The student answered less than half of the questions.	The student answered more than half but not all of the questions.	The student answered all the questions.

Weighted CLO Score	
Remarks	
Instructor's Signature with Date	

NED University of Engineering & Technology
Department of Computer and Information Systems Engineering

Course Code and Title: CS-218 Data Structures and Algorithms

Laboratory Session No. _____________ Date: ______________________

Software Use Rubric				
Skill Sets	Extent of Achievement			
	0	1	2	3
To what level has the student understood the problem?	The student has not understood the problem at all.	The student understands the problem inadequately.	The student understands the problem adequately.	The student understands the problem comprehensively.
To what extent has the student implemented the solution?	The solution has not been implemented.	The solution has syntactic and logical errors.	The solution has syntactic or logical errors.	The solution is syntactically and logically sound for the stated problem parameters.
How efficient is the proposed solution?	The solution does not address the problem adequately.	The solution exhibits redundancy and partially covers the problem.	The solution exhibits redundancy or partially covers the problem.	The solution is free of redundancy and covers all aspects of the problem.
How did the student answer questions relevant to the task?	The student answered none of the questions.	The student answered less than half of the questions.	The student answered more than half but not all of the questions.	The student answered all the questions.

Weighted CLO Score	
Remarks	
Instructor’s Signature with Date	