CHAPTER 1 – TIPS ON C PROGRAMMING

In this section, we will provide tips and good programming practices that will help you to become a better programmer. We assume that you are aware of programming concepts that was covered extensively in CS1110: Computer Programming using C.

Tips on Getting Started with Programming

- The only way to learn a new programming language is by writing programs (just reading the theory will not help, you have to get your hands dirty to learn programming)
- Do not expect your program to be correct at once; even the best programmers make mistakes/errors.
- It is all right if your program throws errors. Understand the error and fix it. Do not be afraid of errors. They are programmer’s friend.
- DO NOT BLINDLY COPY programs from others. It will never help you learn programming. Make mistakes but write your own program.
- DO NOT WRITE entire source code at once, rather write a small part, then test it and once it is working start with another portion. It becomes easy to debug your program this way.
- MAKE FUNCTIONS wherever possible. It helps in debugging and promotes reusability.
- Use DEBUGGER (like gdb) extensively to debug your program.
- BE CAREFUL while using pointers.
- INDENT you code properly. It promotes readability.
- Use PROPER NAMING CONVENTION for your variables and functions. It enhances readability.
- NEVER STOP LEARNING – no one is perfect in programming.

How to compile & run your first program on GNU/Linux OS

Writing a C program requires following 3 steps.

✓ Step 1: Write/Update the source code using a text editor.
✓ Step 2: Compile the source code. If error, go to Step 1 and fix your error, else go to Step 3
✓ Step 3: Run your program.

Step1: Write the source code

Open a text editor (like nano, gedit, vim etc.) and write the source code. To create a file named *hello.c* using nano text editor, write following command on your terminal:

```
nano hello.c
```

This command will open an empty file named hello.c. Copy following lines to the source file:

```
#include<stdio.h>
int main()
{
	printf("Hello World\n");
}
```

Step 2: Compile the source code

To compile the source code, write following command on your terminal

```
gcc hello.c
```

Note: We shall use gcc as our compiler.

If your source code has no any error, above command will create a executable file named *a.out* , otherwise it will show the error messages. If there is an error, fix it and continue with the compilation process.

Note: You can also create custom-named executable file. To create an executable file named hello, write following command during compilation:

```
gcc hello.c -o hello
```

Step 3: Run your program

To run your program, write following command on your terminal.

./a.out (default executable file) or

./hello (custom-named executable file)

Styling your source code (indentation)

Poor indentation and improper documentation makes it difficult to maintain and read your program. Good programming style (indentation) and proper documentation makes your program easier to read. It is advised to following proper styling while writing your program. You may follow the style guides given below:

1. Give proper name to your source code filename. For example if you write a program to find transpose of a matrix, better name for your source file would be *transpose_matrix.c* rather than writing *trans.c* or *t.c* or *transpose.c* or *xyz.c*.
2. Start your program with documentation section. Describe what your program does, mention about author and creation date also. A sample format is shown below.

```
/*
        Description: mention about your program
        Author: author(s) name
        Date: date of creation/updation
*/
```

3. Separate block of code/function with a blank line.
4. All the statements within a block {} should be indented (tab space).

   Code without indentation:

   ```
   int add(int a, int b)
   {
   int sum;
   sum = a+b;
   return sum;
   }
   ```

   Code with indentation *(each statement within a block starts only after a tab space)*

   ```
   int add(int a, int b)
   {
       int sum;
       sum = a+b;
       return sum;
   }
   ```

5. Comment your code wherever possible for better readability.
6. Always use parenthesis in if, else, for, while, do-while, etc.
7. Make your programming style consistent with all your programs.

Data Type & Variables

1. Use const to declare variables that will not change.
2. Don't depend on implicit declarations. Always declare a variable explicitly to avoid confusion.
3. Use #define to give names to constant value.

Statements

1. Be sure to use braces around statement list in if, else-if, while, for, do-while etc.
2. In a loop without a body, put the semicolon for the empty statement on a line by itself.
3. It is easier to read for loop that while loop because the expressions that control the loop are all together.
4. Use a default: clause in every switch statement.

Operators & Expressions

1. Be careful not to use = rather than == to perform a comparison.
2. Do not write an expression whose value depends on the order of evaluation.
3. Use the conditional operator rather than if to simplify expressions.
4. C does not have an explicit Boolean type to integers are used instead. The rule is “Zero is false and any non-zero value is true”.
5. Using sizeof to compute size of data types enhances portability.

Pointers & Arrays

1. Be careful not to deference an uninitialized pointer variable or a NULL pointer.
2. Be careful while using NULL pointer arguments to functions.
3. Be careful not to decrement pointer variable past the beginning of an array.
4. Set pointer variables to NULL when they are not pointing to anything useful.
5. Be careful to check for array bound explicitly in your program.
6. Always check the pointer returned from malloc for NULL.
7. Be careful not to access outside the bounds of dynamically allocated memory.
8. Be careful not to access dynamic memory after it has been freed.

CHAPTER 2 – SEARCHING IN AN ARRAY

Searching is an operation to find the location of the given element (key element) in a list or an array. The search is said to be successful if an element is found, else it is said to be unsuccessful. In this section, we are going to implement two standard methods – Linear search and Binary search

Idea behind Linear Search

In this method, the element to be found (key element) is sequentially searched from the beginning of an array (index 0) to the end of an array. If key element is found in the list, the program stops else it continues to search till it reaches end of the list. Linear search works well with unsorted array.

Idea behind Binary Search

This method works only with sorted array. In this technique, we first compare the key element with the element present at the center of the list. If it matches, the search is successful. Otherwise the list is divided into two parts – one starting from 1st element to center element (left part) and another from center element to the last element (right part). Now, the search will proceed in either of the two parts, depending upon whether the key element is greater or smaller than the center element. If the key element is smaller than center element, search is done at the left part else at the right part of an array.

You are supposed to implement linear search and binary search in this chapter by creating your own header file. The section below describes steps to create your own header file.

Creating your own header file

For demonstration purpose, let us take an example to addition of two numbers using header file. We are going to define addition function in a header file.

1. Create a header file with .h extension. Let's create add.h and write following code

```
int add(int a, int b)
{
        return a+b;
}
```

2. Now create your source file say (addition.c). Write following code.

```
#include<stdio.h>
#include “add.h”      // including our add.h header file. Use double quotes (“”)
int main()
```

```
{
        int x=10,y=20,sum;
        sum = add(x,y);
       print("Sum is: %d\n",sum);
}
```

3. Compile: *gcc addition.c*
4. Run: *./a.out*

Exercise Questions:

1. Write a program to implement linear search on 1-D array (Create your own header file)
2. Write a program to implement binary search on 1-D array (Create your own header file)

Additional Questions:

1. Write a menu-driven program to implement following operations on 1-D array (use header files):
    a. Insert an element at specified position.
    b. Delete an element at specified position.
    c. Search for an element.
    d. Display the array.
    e. Reverse the list.
    f. Move all zeros at the bottom of an array.

2. Write a program that creates an array of 100 random integers in the range 1 to 200 and then, using the linear search, search the array 100 times using randomly generated key in the same range. At the end of the program, display the following statistics:
    a. The number of searches completed.
    b. The number of successful searches.
    c. The number of unsuccessful searches.
    d. The percentage of successful searches.

CHAPTER 3 – MATRIX

Matrix is a 2-dimensional array. A sparse matrix is a two-dimensional array where the majority of elements have the value null (zero). Sparse matrix can be represented in less space if only non-zero elements are represented. The compact form of sparse matrix is a 3-tuple form where you only store row index, column index and value for each non-zero elements.

Example:

Sparse Matrix

$$\begin{matrix} 0 & 0 & 0 & -2 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & -2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 4 & -6 \end{matrix}$$

Compact form:

$$\begin{matrix} 0 & 3 & -2 \\ 0 & 5 & 1 \\ 1 & 5 & 5 \\ 2 & 4 & -2 \\ 4 & 3 & 1 \\ 5 & 0 & 1 \\ 5 & 4 & 4 \\ 5 & 5 & -6 \end{matrix}$$

Exercise Questions:

1. Write a program to convert a sparse matrix to its compact form.
2. Write a program to convert a compact form to a sparse matrix.
3. Write a program to find the transpose of compact form of a sparse matrix.

Additional Questions:

1. Write a program to find the determinant of a given matrix.
2. Write a program to check whether a given matrix is singular.(A matrix is called singular it the determinant value of a matrix is 0)
3. Write a program to check whether a given square matrix is orthogonal. (A matrix is said to be orthogonal, if the matrix obtained by multiplying the matrix with its transpose is an identity matrix)

}

CHAPTER 4 – STACKS

A stack is an ordered collection of elements where the insertion and deletion operation takes place at one end only, known as top of the stack (TOS). As all deletion and insertion in a stack is done from one end, the last added element will be the first one to be removed. This is the reason why stack is also called as Last-in-First-out (LIFO) type of list.

We use following terms for stack:

Push: for adding/inserting element to a stack

Pop: for removing element from a stack

Peek: display the element at the TOS

The procedure to perform array implementation of stack is given below:

StackInitialization()

```
{
    Let MAX be the size of the stack.
    Let S[MAX] be the stack.
    Let top be the TOS. Initialize top=-1
}
```

StackPush(n)

```
{
    If(top = MAX-1)
        Display “Stack overflow”
    Else
        top++
        S[top] = n
}
```

StackPop()

```
{
    If (top is -1)
        Display “Underflow”
    Else
        n = S[top]
        Display “n deleted”
        top = top-1
}
```

Exercise Questions:

1. Write a menu-driven program with following operations to implement stack using array:
    a. Push
    b. Pop
    c. Peek
    d. Display
2. Write a program to convert given infix expression to postfix expression strictly using stacks.
3. Write a program to evaluate the given postfix expression strictly using stacks.

Additional Questions:

1. Write a program to implement a text editor which can process a line of text using a stack. Consider two characters: erase character (#) and kill character (@). The erase character has the effect of canceling the previous uncanceled character. For example, the string *abc#d##d* is really the string *ae*. The kill character cancels all previous characters on the current line.

2. Write a function called *copyStack* that copies the contents of one stack into another. The function must have two arguments of type stack, one for the source stack and one for the destination stack. The order of the stack must be identical.

CHAPTER 5 - QUEUE

Queue is an abstract data type or a linear data structure, just like stack data structure, in which the first element is inserted from one end called the **REAR**(also called **tail**), and the removal of existing element takes place from the other end called as **FRONT**(also called **head**). This makes queue as **FIFO** (First in First Out) data structure, which means that element inserted first will be removed first. The process to add an element into queue is called Enqueue and the process of removal of an element from queue is called Dequeue. There are different types of Queue like ***Linear Queue, Circular Queue, Double ended queue, Priority Queue***

Implementation of Queue Data Structure

Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array.

Inserting an element in a linear queue

Queues maintain two data pointers, front and rear. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

- 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.

Algorithm for Enqueue operation

```
procedure enqueue (data)

  if queue is full
    return overflow
  endif

  rear ← rear + 1
  queue[rear] ← data
  return true

end procedure
```

Deleting an element in a linear queue

Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation −

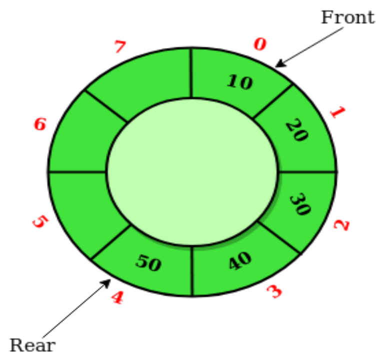
- **Step 1** − Check if the queue is empty.
- **Step 2** − If the queue is empty, produce underflow error and exit.
- **Step 3** − If the queue is not empty, access the data where **front** is pointing.
- **Step 4** − Increment **front** pointer to point to the next available data element.
- **Step 5** − Return success.

Algorithm for dequeue operation

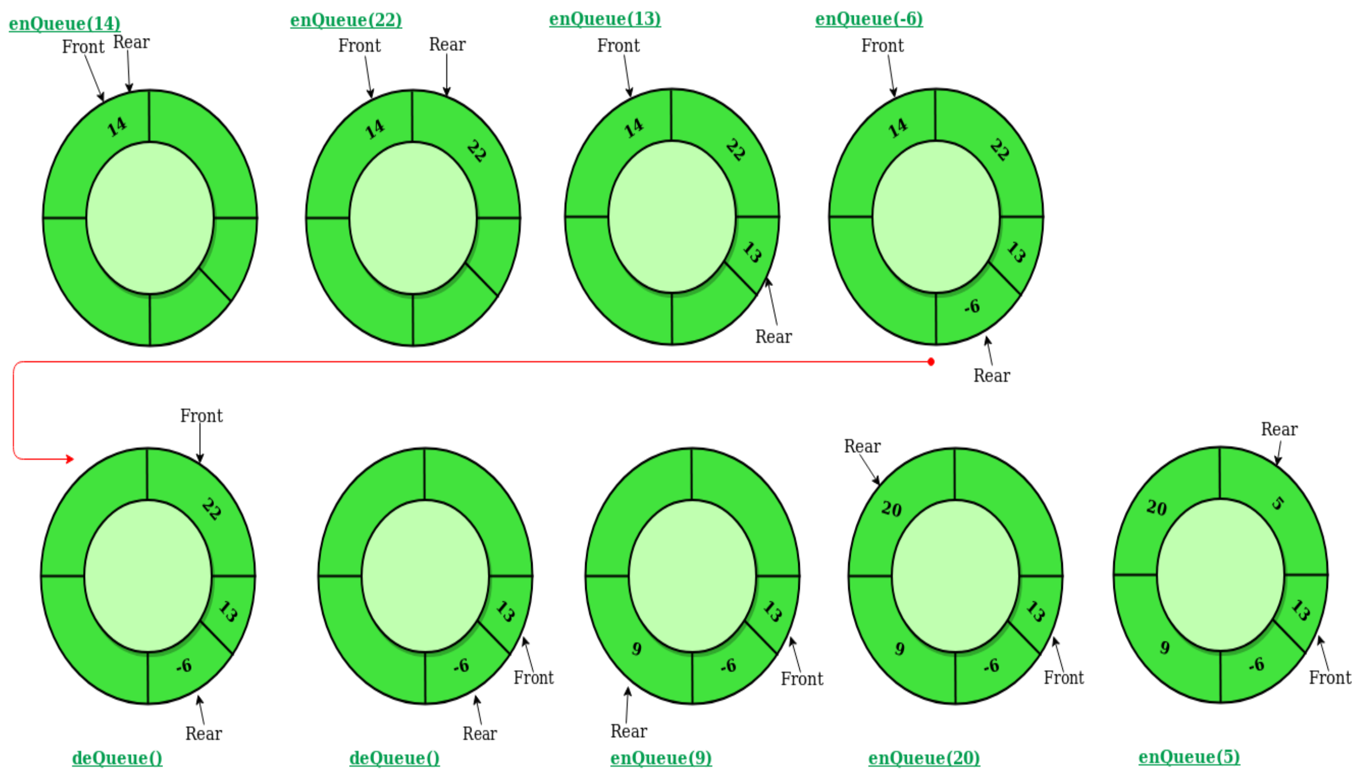
```
procedure dequeue

  if queue is empty
    return underflow
  end if

  data = queue[front]
  front ← front + 1
  return true

end procedure
```

Circular Queue

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called **'Ring Buffer'**.



In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we cannot insert the next element even if there is a space in front of queue.



Algorithm for inserting a value to a circular queue

In a circular queue, enQueue() is a function which is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at rear position. The enQueue() function takes one integer value as parameter and inserts that value into the circular queue. We can use the following steps to insert an element into the circular queue...

Step 1: Check whether queue is FULL. ((rear == SIZE-1 && front == 0) || (front == rear+1))

Step 2: If it is FULL, then display "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.

Step 3: If it is NOT FULL, then check rear == SIZE - 1 && front != 0 if it is TRUE, then set rear = -1.

Step 4: Increment rear value by one (rear++), set queue[rear] = value and check 'front == -1' if it is TRUE, then set front = 0.

Algorithm for deleting a value from a circular queue

In a circular queue, deQueue() is a function used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position. The deQueue() function doesn't take any value as parameter. We can use the following steps to delete an element from the circular queue...

Step 1: Check whether queue is EMPTY. (front == -1 && rear == -1)

Step 2: If it is EMPTY, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.

Step 3: If it is NOT EMPTY, then display queue[front] as deleted element and increment the front value by one (front ++). Then check whether front == SIZE, if it is TRUE, then set front = 0. Then check whether both front - 1 and rear are equal (front -1 == rear), if it TRUE, then set both front and rear to '-1' (front = rear = -1).

Exercise Questions:

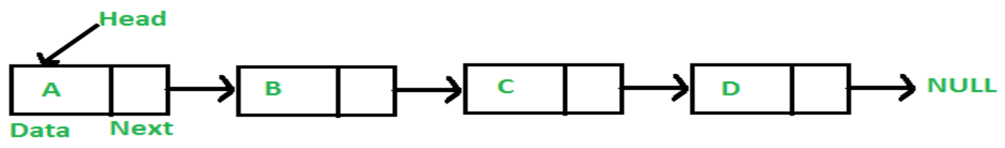
1. Write a menu-driven program to implement following operations on a linear queue using arrays: (a) Enqueue (b) Dequeue (c) Display
2. Write a program menu-driven program to implement following operations on a circular queue using array: (a) Enqueue (b) Dequeue (c) Display.

Additional Questions:

1. Write a program to implement priority queue using array.
2. Write a program to copy one queue to another when the queue is implemented as a linked list.

CHAPTER 6 - LINKED LIST

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at contiguous location; the elements are linked using pointers.



A linked list is represented by a pointer to the first node of the linked list. The first node is called head. If the linked list is empty, then value of head is NULL.
Each node in a list consists of at least two parts:
1) data
2) pointer to the next node

Types of Linked Lists

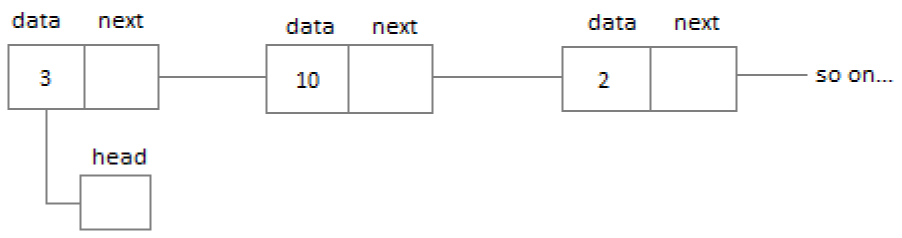
There are 3 different implementations of Linked List available, they are:

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

Singly Linked List

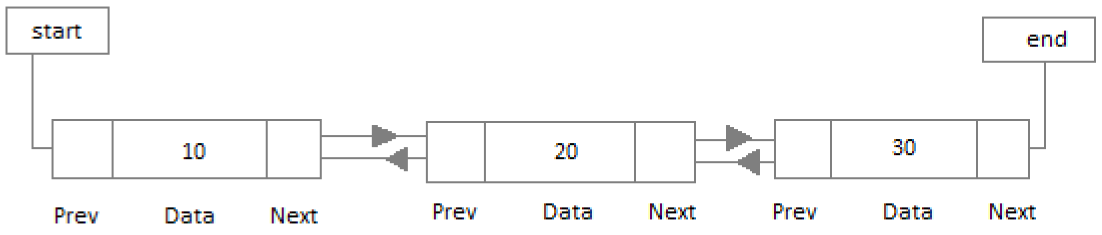
Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node in the sequence of nodes.

The operations we can perform on singly linked lists are insertion, deletion and traversal.



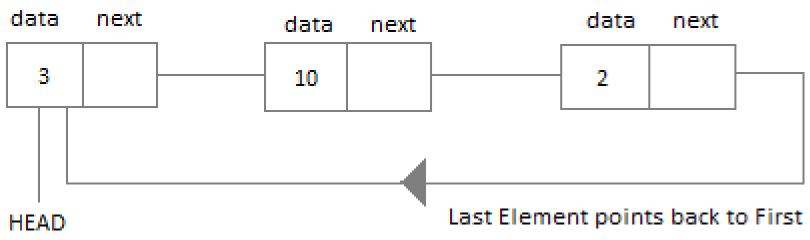
Doubly Linked List

In a doubly linked list, each node contains a data part and two addresses, one for the previous node and one for the next node.



Circular Linked List

In circular linked list the last node of the list holds the address of the first node hence forming a circular chain.



Implementation of Single Linked list

In a single linked list we perform the following operations...

- Insertion
- Deletion
- Display

Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.

Step 1: Include all the header files which are used in the program.
Step 2: Declare all the user defined functions.
Step 3: Define a Node structure with two members data and next
Step 4: Define a Node pointer 'head' and set it to NULL.
Step 4: Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

In a single linked list, the insertion operation can be performed in three ways. They are as follows...

- Inserting At Beginning of the list

- Inserting At End of the list
- Inserting At Specific location in the list

Inserting At Beginning of the list
We can use the following steps to insert a new node at beginning of the single linked list...

Step 1: Create a newNode with given value.
Step 2: Check whether list is Empty (head == NULL)
Step 3: If it is Empty then, set newNode→next = NULL and head = newNode.
Step 4: If it is Not Empty then, set newNode→next = head and head = newNode.

Inserting At End of the list
We can use the following steps to insert a new node at end of the single linked list...

Step 1: Create a newNode with given value and newNode → next as NULL.
Step 2: Check whether list is Empty (head == NULL).
Step 3: If it is Empty then, set head = newNode.
Step 4: If it is Not Empty then, define a node pointer temp and initialize with head.
Step 5: Keep moving the temp to its next node until it reaches to the last node in the list (until temp → next is equal to NULL).
Step 6: Set temp → next = newNode.

Inserting At Specific location in the list (After a Node)
We can use the following steps to insert a new node after a node in the single linked list...

Step 1: Create a newNode with given value.
Step 2: Check whether list is Empty (head == NULL)
Step 3: If it is Empty then, set newNode → next = NULL and head = newNode.
Step 4: If it is Not Empty then, define a node pointer temp and initialize with head.
Step 5: Keep moving the temp to its next node until it reaches to the node after which we want to insert the newNode (until temp1 → data is equal to location, here location is the node value after which we want to insert the newNode).
Step 6: Every time check whether temp is reached to last node or not. If it is reached to last node then display 'Given node is not found in the list!!! Insertion not possible!!!' and terminate the function. Otherwise move the temp to next node.
Step 7: Finally, Set 'newNode → next = temp → next' and 'temp → next = newNode'

Deletion
In a single linked list, the deletion operation can be performed in three ways. They are as follows...

- Deleting from Beginning of the list
- Deleting from End of the list
- Deleting a Specific Node

Deleting from Beginning of the list
We can use the following steps to delete a node from beginning of the single linked list...

Step 1: Check whether list is Empty (head == NULL)
Step 2: If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
Step 3: If it is Not Empty then, define a Node pointer 'temp' and initialize with head.
Step 4: Check whether list is having only one node (temp → next == NULL)
Step 5: If it is TRUE then set head = NULL and delete temp (Setting Empty list conditions)
Step 6: If it is FALSE then set head = temp → next, and delete temp.

Deleting from End of the list
We can use the following steps to delete a node from end of the single linked list...

Step 1: Check whether list is Empty (head == NULL)
Step 2: If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
Step 3: If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.
Step 4: Check whether list has only one Node (temp1 → next == NULL)
Step 5: If it is TRUE. Then, set head = NULL and delete temp1. And terminate the function. (Setting Empty list condition)
Step 6: If it is FALSE. Then, set 'temp2 = temp1 ' and move temp1 to its next node. Repeat the same until it reaches to the last node in the list. (until temp1 → next == NULL)
Step 7: Finally, Set temp2 → next = NULL and delete temp1.

Deleting a Specific Node from the list
We can use the following steps to delete a specific node from the single linked list...

Step 1: Check whether list is Empty (head == NULL)
Step 2: If it is Empty then, display 'List is Empty!!! Deletion is not possible' and terminate the function.
Step 3: If it is Not Empty then, define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with head.
Step 4: Keep moving the temp1 until it reaches to the exact node to be deleted or to the last node. And every time set 'temp2 = temp1' before moving the 'temp1' to its next node.

Step 5: If it is reached to the last node then display 'Given node not found in the list! Deletion not possible!!!'. And terminate the function.
Step 6: If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
Step 7: If list has only one node and that is the node to be deleted, then set head = NULL and delete temp1 (free(temp1)).
Step 8: If list contains multiple nodes, then check whether temp1 is the first node in the list (temp1 == head).
Step 9: If temp1 is the first node then move the head to the next node (head = head → next) and delete temp1.
Step 10: If temp1 is not first node then check whether it is last node in the list (temp1 → next == NULL).
Step 11: If temp1 is last node then set temp2 → next = NULL and delete temp1 (free(temp1)).
Step 12: If temp1 is not first node and not last node then set temp2 → next = temp1 → next and delete temp1 (free(temp1)).

Displaying a Single Linked List
We can use the following steps to display the elements of a single linked list...

Step 1: Check whether list is Empty (head == NULL)
Step 2: If it is Empty then, display 'List is Empty!!!' and terminate the function.
Step 3: If it is Not Empty then, define a Node pointer 'temp' and initialize with head.
Step 4: Keep displaying temp → data with an arrow (--->) until temp reaches to the last node
Step 5: Finally display temp → data with arrow pointing to NULL (temp → data ---> NULL).

Algorithm for reversing a linked list

Iterative Method:

1. Initialize three pointers prev as NULL, curr as head and next as NULL.

2. Iterate trough the linked list. In loop, do following.

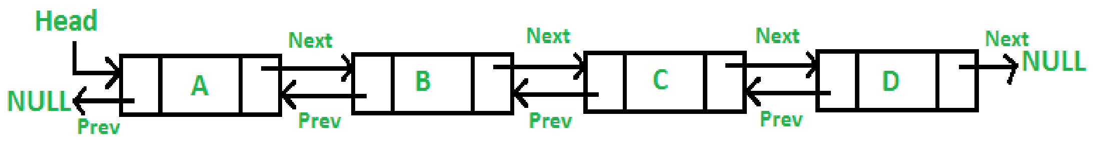
```
// Before changing next of current,

// store next node

                    next = curr->next

// This is where actual reversing happens

                    curr->next = prev

// Move prev and curr one step forward
```

prev = curr

curr = next

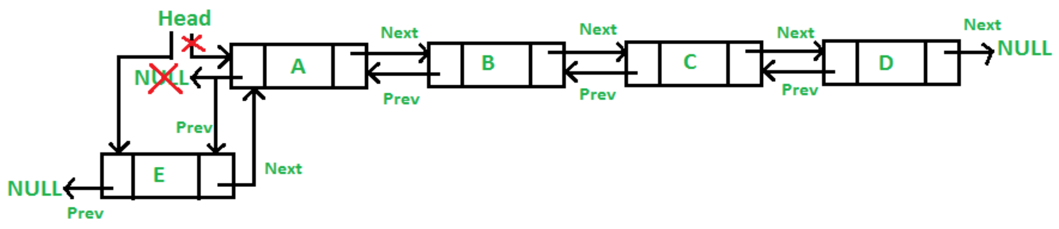
Doubly Linked List

A **D**oubly **L**inked **L**ist (DLL) contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list.

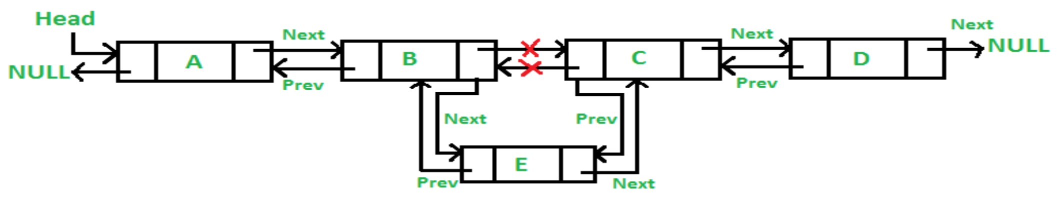


Insertion
A node can be in four ways
1) At the front of the DLL
2) At any position of the DLL
3) At the end of the DLL

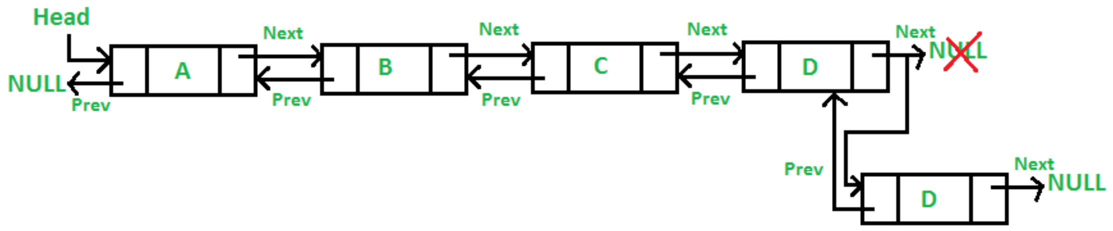
1) Add a node at the front:
The new node is always added before the head of the given Linked List. And newly added node becomes the new head of DLL.



2) Add a node after a given node:
We are given pointer to a node as prev_node, and the new node is inserted after the given node.



3) Add a node at the end:
The new node is always added after the last node of the given Linked List. Since a Linked List is typically represented by the head of it, we have to traverse the list till end and then change the next of last node to new node.



Similarly deletion can also be done in the following ways:

A node can be deleted in four ways
1) From the front of the DLL
2) From any position of the DLL
3) From the end of the DLL

Exercise Questions:

1. Write a menu-driven program to implement following the basic operations in a single linked list. Make use of functions for each of these operations.
    a. Insertion at beginning
    b. Insertion at end
    c. Insertion at any position
    d. Deletion from beginning
    e. Deletion from end
    f. Deletion from any position
    g. Display the list
    h. Count number of elements in a list
2. Write a program to reverse a singly linked list.

3. Write a menu-driven program to implement basic operations in a Doubly Linked List. Make use of functions for each of these operations.
    a. Insertion at beginning
    b. Insertion at end
    c. Insertion at any position
    d. Deletion from beginning
    e. Deletion from end
    f. Deletion from any position
    g. Display the list
    h. Count number of elements in a list
4. Write a menu-driven program with following operations to implement stack using linked list: (a)Push (b) Pop (c) Peek (d) Display
5. Write a menu-driven program to implement following operations on a linear queue using linked list: (a) Enqueue (b) Dequeue (c) Display
6. Write a program menu-driven program to implement following operations on a circular queue using linked list: (a) Enqueue (b) Dequeue (c) Display.

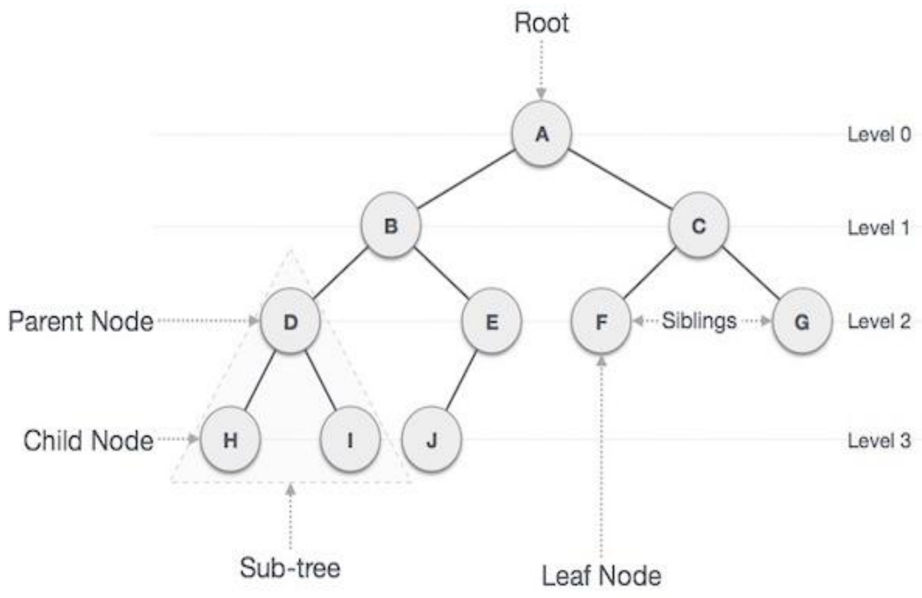
Additional Questions:

1. Write a program to find the sum two polynomials on a single variable. Represent each term (having coefficient and exponent) of a polynomial as a node of a linked list. Display the sum polynomial.
2. Write a program to sort a given linked list of integers using bubble sort.

CHAPTER 7 - TREE

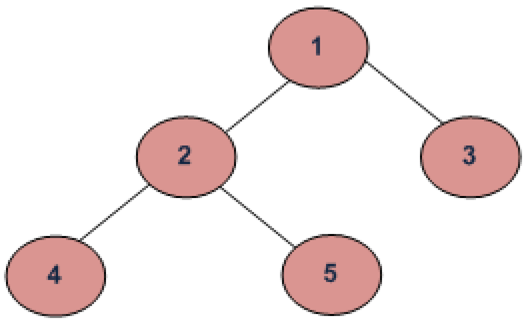
A tree is a data structure made up of nodes or vertices and edges without having any cycle. The tree with no nodes is called the **null** or **empty** tree.

A **binary tree** is a tree data structure in which each node has at most two children, which are referred to as the *left child* and the *right child.*



Tree Traversals (Inorder, Preorder and Postorder)

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



:
(a) Inorder (Left, Root, Right) : 4 2 5 1 3
(b) Preorder (Root, Left, Right) : 1 2 4 5 3
(c) Postorder (Left, Right, Root) : 4 5 2 3 1

Inorder Traversal:

Algorithm Inorder (tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Preorder Traversal:

Algorithm Preorder (tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Postorder Traversal:

Algorithm Postorder (tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

Exercise Questions:

1. Write a program to implement tree traversal (Preorder, Inorder, Postorder) in a binary tree using array and liked list.
2. Write a program to implement following operations on a binary search tree:
   a. Insertion of a node
   b. Searching of a node
   c. Deletion of a node

Additional Questions:

1. Write a program to represent an arithmetic expression in a binary tree called as expression tree. Evaluate the expression tree and display the result of evaluation.

CHAPTER 8 – GRAPH

Graph is a data structure that consists of following two components:

1. A finite set of vertices called as nodes (V)
2. A finite set of ordered pair of form (u,v) called as edges (E)

Graph can be represented using: (a) Adjacency matrix (b) Adjacency list

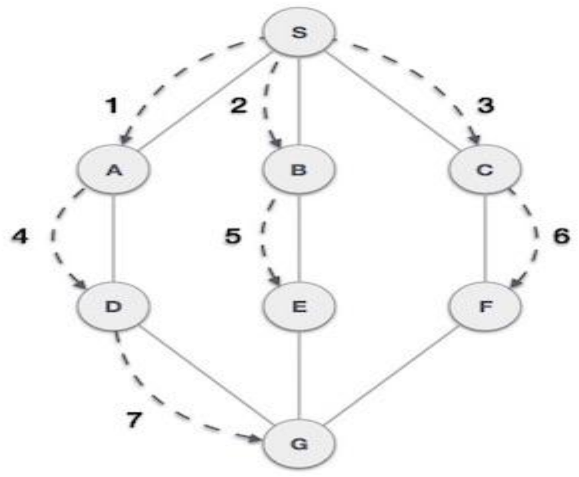
Adjacency Matrix: It is a 2D array of size v *v, where v is the no. of vertices in a graph (G). Let adj[][] be a matrix. If adj[i][j] = 1, it indicates that there is edge from node (i) to node (j). Adjacency matrix can also be used to represent weighted graph. If adj[i][j] = w, then there is an edge from vertex (i) to vertex (j) with weight w.

Adjacency List: In this, an array of linked list is used. The size of the array is equal to the number of vertices. If the array is arr[], then the entry arr[i] represents a linked list of vertices adjacent to ith vertex.

Breadth First Search or BFS for a Graph

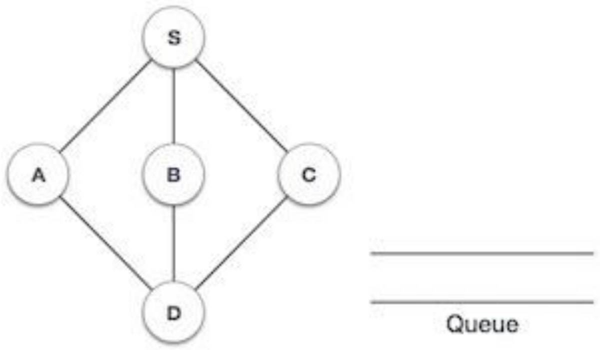
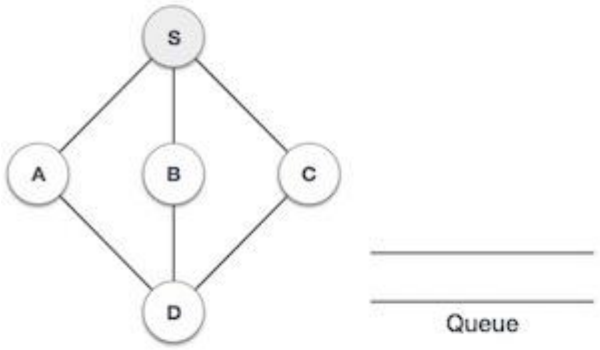
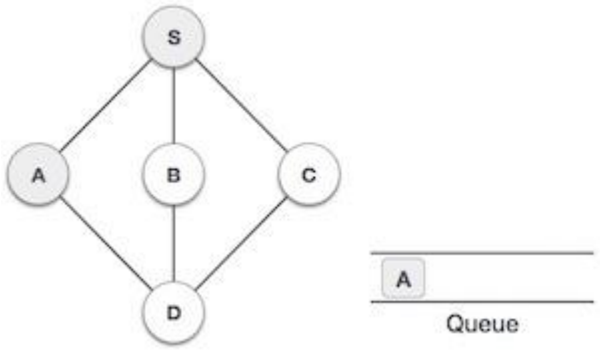
Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

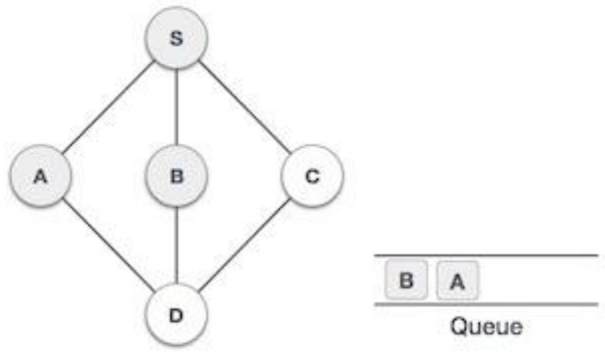
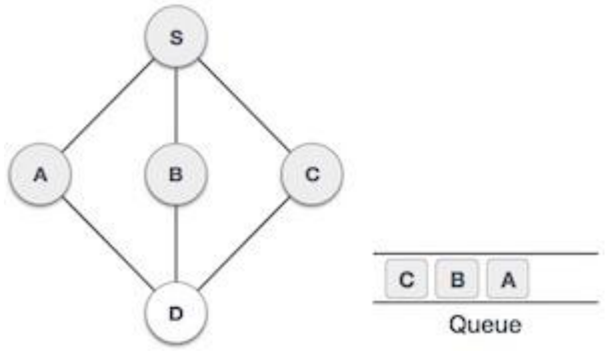
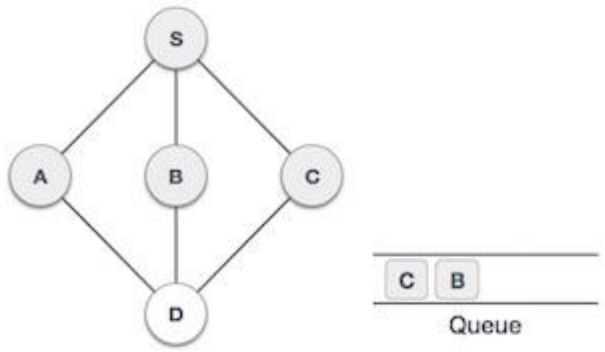
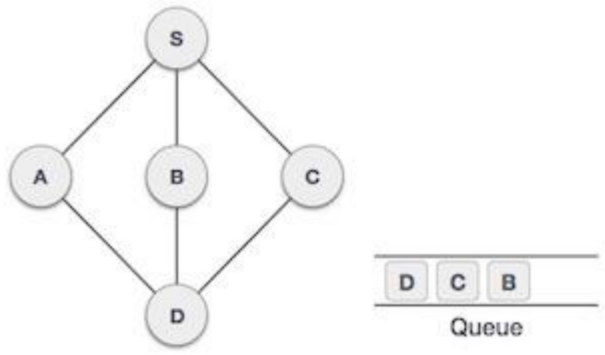
Breadth First Search (BFS) uses a breadth ward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- Rule 1 − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- Rule 2 − If no adjacent vertex is found, remove the first vertex from the queue.
- Rule 3 − Repeat Rule 1 and Rule 2 until the queue is empty.

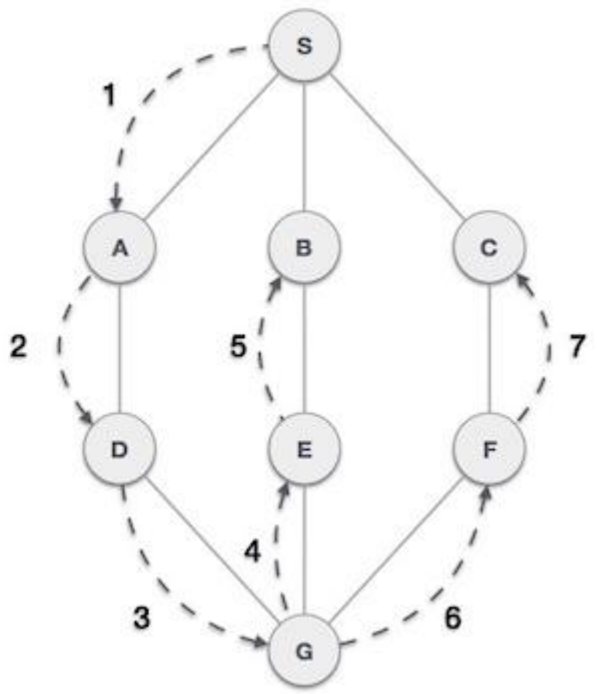
Step	Traversal	Description
1		Initialize the queue.
2		We start from visiting **S**(starting node), and mark it as visited.
3		We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.

4		Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.
5		Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.
6		Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.
7		From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

- At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

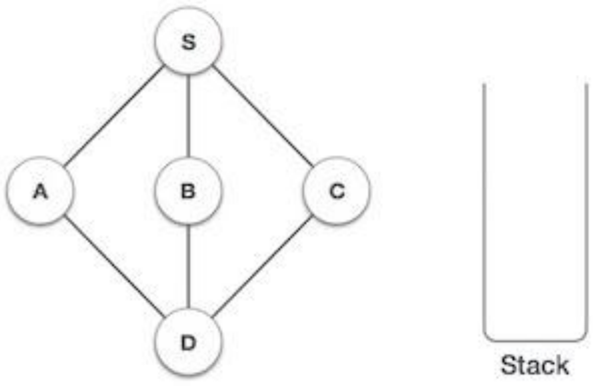
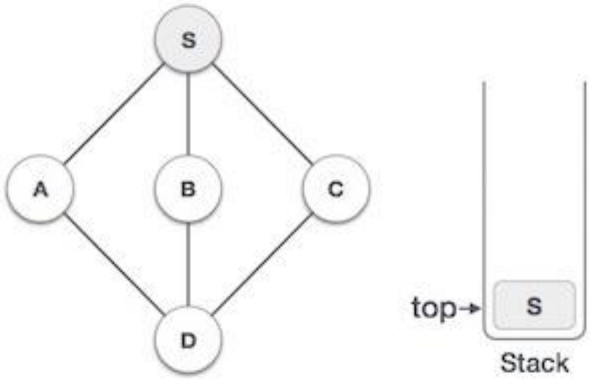
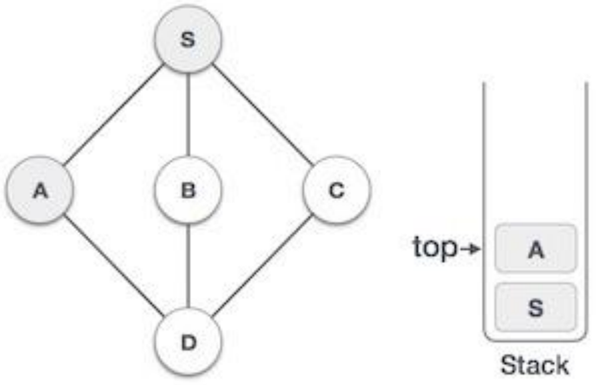
Depth First Search or DFS for a Graph

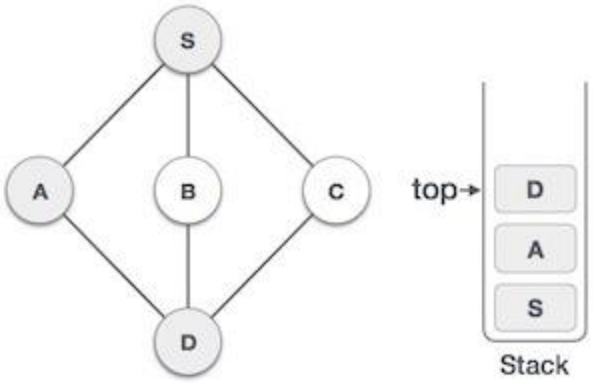
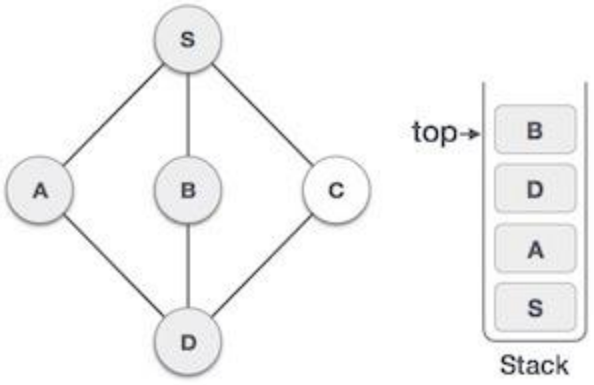
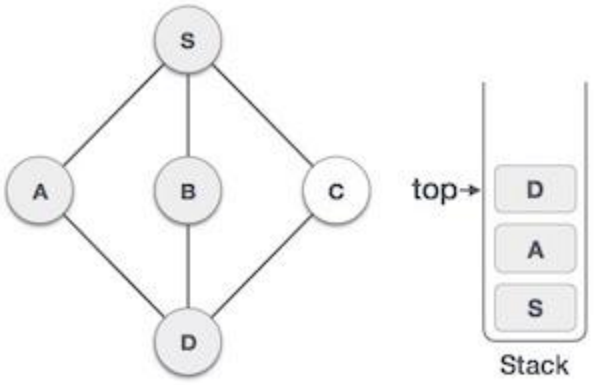
Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

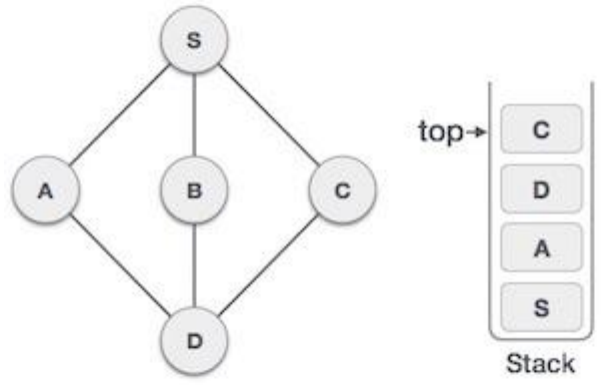


As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

- **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** − Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1		Initialize the stack.
2		Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3		Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S**and **D** are adjacent to **A** but we are concerned for unvisited nodes only.

4		Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.
5		We choose **B**, mark it as visited and put onto the stack. Here **B**does not have any unvisited adjacent node. So, we pop **B**from the stack.
6		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.

7		Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

Exercise Questions:

1. Write a program to represent a graph using adjacency matrix and adjacency list.
2. Write a program to implement Depth First Search traversal in a graph.

Additional Questions:

1. Write a program to implement Breadth First Search traversal in a graph.

ADDITIONAL PAGE

APPENDIX

1. Dynamic Memory Allocation

Dynamic Memory Allocation can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.

C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

1. malloc()
2. calloc()
3. free()
4. realloc()

1.**malloc() :** "malloc" or "memory allocation" method is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form.

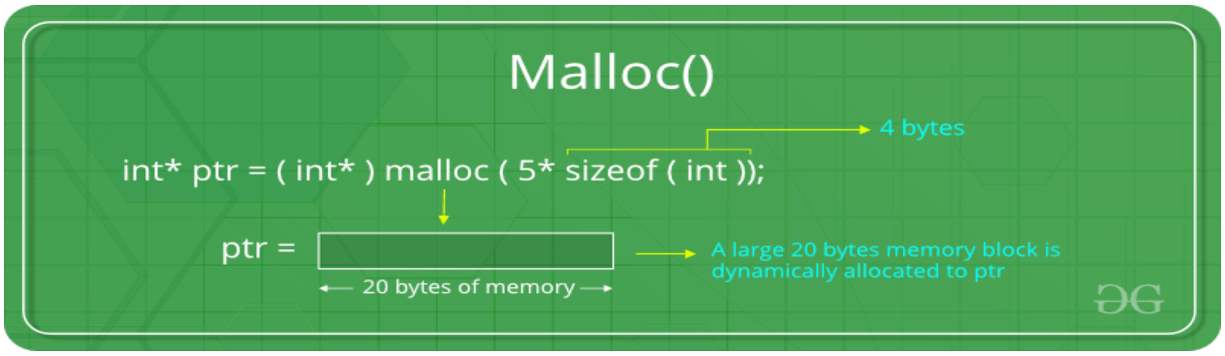
Syntax: **ptr = (cast-type*) malloc(byte-size)**

For Example:

ptr = (int*) malloc(100 * sizeof(int));

Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory.

And, the pointer ptr holds the address of the first byte in the allocated memory.


Malloc()
4 bytes
int* ptr = (int*) malloc (5* sizeof (int));
ptr =
20 bytes of memory
A large 20 bytes memory block is dynamically allocated to ptr

If the space is insufficient, allocation fails and returns a NULL pointer.

2. calloc( ): **“calloc”** or **“contiguous allocation”** method is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value ‘0’.

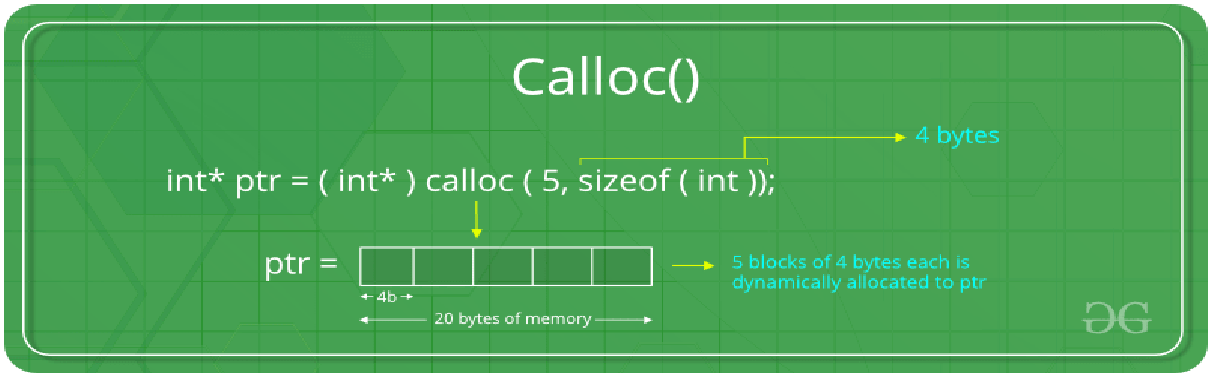
Syntax: **ptr = (cast-type*)calloc(n, element-size);**

For Example:

ptr = (float*) calloc(25, sizeof(float));

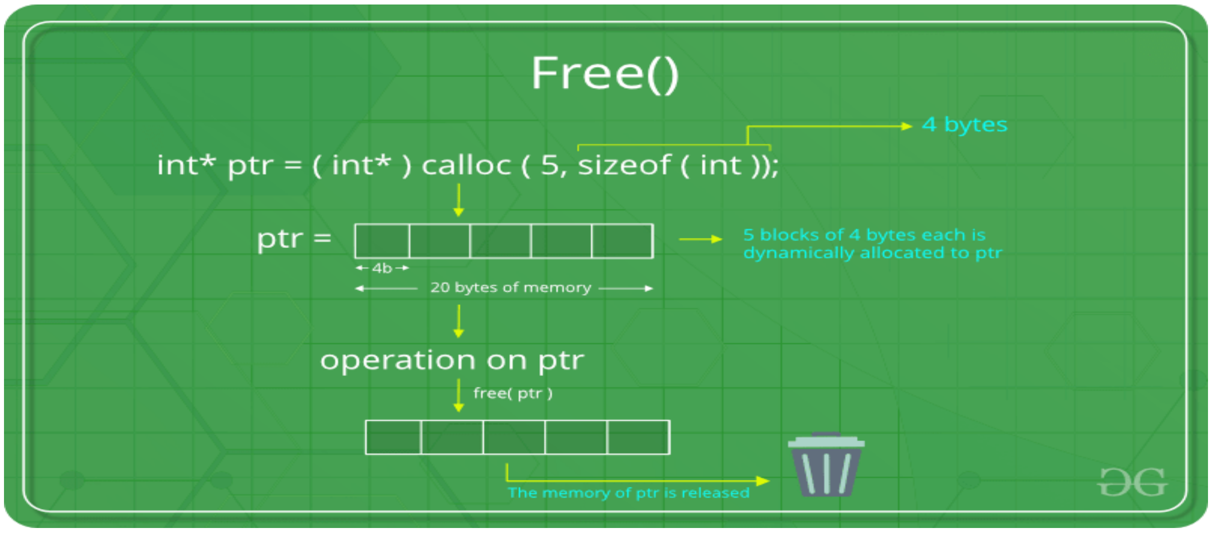
This statement allocates contiguous space in memory for 25 elements each with the size of float.

If the space is insufficient, allocation fails and returns a NULL pointer.


Calloc()
4 bytes
int* ptr = (int*) calloc (5, sizeof (int));
ptr =
5 blocks of 4 bytes each is dynamically allocated to ptr
4b
20 bytes of memory

3. free( ): “free” method is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() are not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

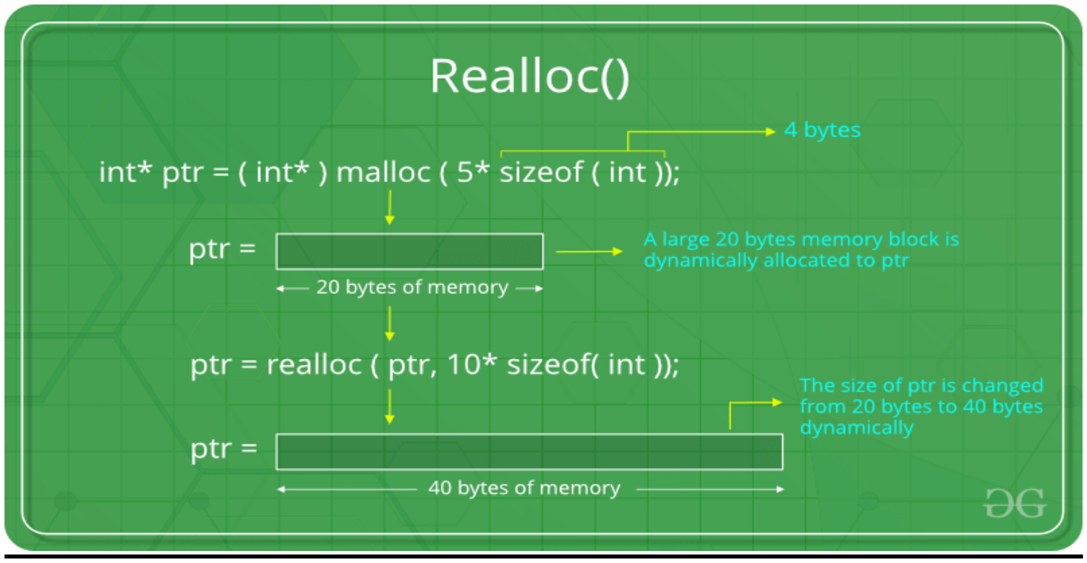
Syntax: free(ptr);


Free()
4 bytes
int* ptr = (int*) calloc (5, sizeof (int));
ptr =
5 blocks of 4 bytes each is dynamically allocated to ptr
4b
20 bytes of memory
operation on ptr
free(ptr)
The memory of ptr is released

4.realloc( ): "realloc" or **"re-allocation"** method is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**.

Syntax: ptr = realloc(ptr, newSize);

where ptr is reallocated with new size 'newSize'. If the space is insufficient, allocation fails and returns a NULL pointer.


Realloc()
4 bytes
int* ptr = (int*) malloc (5* sizeof (int));
ptr =
A large 20 bytes memory block is dynamically allocated to ptr
20 bytes of memory
ptr = realloc (ptr, 10* sizeof(int));
The size of ptr is changed from 20 bytes to 40 bytes dynamically
ptr =
40 bytes of memory

2. Debugging a C Program

GDB is a debugger for C (and C++). GDB stands for GNU Debugger. ++). It allows you to do things like run the program up to a certain point then stop and print out the values of certain variables at that point, or step through the program one line at a time and print out the values of each variable after executing each line. It uses a command line interface. **gdb** is a command for invoking the debugger. The following basic steps needs to be followed to start using gdb:

1. Comiple your program (say *hello.c*) using -g flag. This informs your compiler to compile your code with symbolic debugging information included. ***gcc -g hello.c***

On successfull execution of given command, it will generate an executable file ***a.out***

[*Note: Ensure your program has no syntax error before running this command.]*

2. To start the dubugger along with the souce code displayed in the terminal, type following command: ***gdb a.out -tui***

Here, tui flag the Terminal User Interface is activated, hence your source code along with line numbers will be visible.

3. Just starting the debugger to run the program straight through isn't very useful—we need to stop execution and get into stepping mode. You need to set a breakpoint someplace you'd like to stop. You use the **break** or **b** command, and specify a location, which can be a function name, a line number, or a source file and line number. These are examples of locations, which are used by various other commands as well as **break**:

break main	Break at the beginning of the main() function
break 5	Break at line 5 of the current file
break hello.c:5	Break at line 5 of hello.c

Let’s say we issue a breakpoint at the beginning of the main() function, type following command: ***break main***

Note: To clear a breakpoint, use the clear command with the breakpoint location. You can also clear a breakpoint by number with the delete command.

4. To run the program under gdb, use ***run*** command. Type following on the terminal,

   Run

Note: If you haven't set any breakpoints, run command will simply execute the full program.

5. To execute the next statement, use **next** command (or **n**). This command moves you to the next statement in the current function (or returns to the function's caller if you've stepped off the end of the function.)

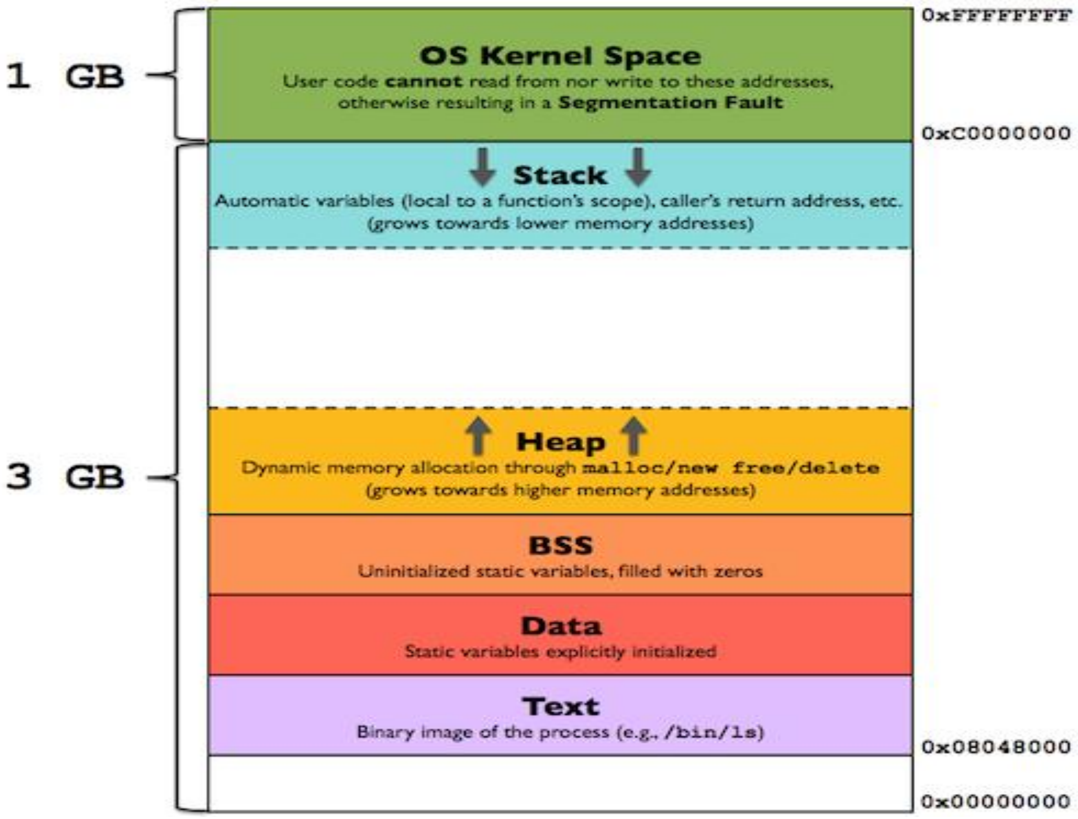
6. If you have a function you want to *step into* from your current function, and trace through that function line-by-line then use the **step** (or **s**) command to do this. It works just like **next**, except it steps into functions.

7. You can hit **CTRL-C** and that'll stop the program wherever it happens to be and return you to a "(gdb)" prompt

8. If you have some variables you wish to inspect over the course of the run, you can **display** them, but only if the variable is currently in scope. Each time you step the code, the value of the variable will be displayed (if it's in scope).
9. If you just want to one-off know the value of a variable, you can using the **print** command.

3. Segmentation Fault

A segmentation fault (aka segfault) is a common condition that causes programs to crash; they are often associated with a file named **core**. Segfaults are caused by a program trying to read or write an illegal memory location. Program memory is divided into different segments: a text segment for program instructions, a data segment for variables and arrays defined at compile time, a stack segment for temporary (or automatic) variables defined in functions, and a heap segment for variables allocated during runtime by functions, such as malloc. A segfault occurs when a reference to a variable falls outside the segment where that variable resides, or when a write is attempted to a location that is in a read-only segment. In practice, segfaults are almost always due to trying to read or write a non-existent array element, not properly defining a pointer before using it, or accidentally using a variable's value as an address.


1 GB
3 GB
0xFFFFFFFF
0xC0000000
0x08048000
0x00000000
OS Kernel Space
User code cannot read from nor write to these addresses, otherwise resulting in a Segmentation Fault
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)

The following three cases illustrate the most common types of array-related segfaults:

/* "Array out of bounds" error valid indices for array foo are 0, 1, ... 999 */

int foo[1000];

for (int i = 0; i <= 1000 ; i++)

foo[i] = i;

Here, array foo is defined for index = 0, 1, 2, ... 999. However, in the last iteration of the for loop, the program tries to access foo[1000]. This will result in a segfault if that memory location lies outside the memory segment where foo resides. Even if it doesn't cause a segfault, it is still a bug.

/* Illegal memory access if value of n is not in the range 0, 1, ... 999 */

int n;

int foo[1000];

for (int i = 0; i < n ; i++)

foo[i] = i;

Here, integer n could be any random value. As in case A, if it is not in the range 0, 1, ... 999, it might cause a segfault. Whether it does or not, it is certainly a bug.

```
/* Illegal memory access because no memory is allocated for foo2 */
float *foo, *foo2;
foo = (float*)malloc(1000);
foo2[0] = 1.0;
```

Here, the allocation of memory for variable foo2 has been overlooked, so foo2 will point to a random location in memory. Accessing foo2[0] will likely result in a segfault.

Another common programming error that leads to segfaults is oversight in the use of pointers. For example, the C function scanf() expects the address of a variable as its second parameter; therefore, the following will likely cause the program to crash with a segfault:

```
int foo = 0;
scanf("%d", foo); /* Note missing & sign ; correct usage would have been &foo */
```

The variable foo might be defined at memory location 1000, but the above function call would try to read integer data into memory location 0 according to the definition of foo.

Segfaults can also occur when your program runs out of stack space. This may not be a bug in your program, but may be due instead to your shell setting the stack size limit too small.

References:

- https://beej.us/guide/bggdb/
- https://web.eecs.umich.edu/~sugih/pointers/summary.html
- https://www.geeksforgeeks.org/gdb-step-by-step-introduction/
- https://kb.iu.edu/d/aqsj
- https://gabrieletolomei.wordpress.com/miscellanea/operating-systems/in-memory-layout/