Lab Manual

CS2001 – Data Structures

Fall-2021

National University of Computer and Emerging Sciences-FAST
Karachi Campus

Acknowledgments

This Lab manual would not have been possible without the help team of faculty member of Data structure Fall 2021.

Faculty Members of Data Structure Fall 2021

- Dr Muhammed Rafi (Course Coordinator)
- Dr Syed Ali Raza
- Shahbaz Siddiqui
- Muhammad Sohail
- Mubashra Fayyaz
- Farah Sadia
- Mafaza mohi
- Erum Shaheen
- Muhammad Ali Shah Fatimi
- Aqsa Zahid

Data Structures Lab

Session 1

Course: Data Structures (CS2001) **Semester:** Fall 2021
Instructor: Muhammad Rafi **T.A:** N/A

Note:

- Maintain discipline during the lab.
- Listen and follow the instructions as they are given.
- Just raise hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

Task # 1

Using Command Line arguments

The main() in C/C++ is a very vital function. It is the main entry point for a program. There will be only one main in a program. It is generally defined with a return type of int and without parameters:

int main() { /* ... */ }

We can also give command-line arguments in C and C++. Command-line arguments are given after the name of the program in command-line shell of Operating Systems.

C:>\ myProgram.exe I am Rafi

To pass command line arguments, we typically define main() with two arguments : first argument is the number of command line arguments and second is list of command-line arguments.

int main(int argc, char *argv[]) { /* ... */ }
or

int main(int argc, char **argv) { /* ... */ }

Task # 2

Using Command Line arguments to read from the file and write to the file.

Example:

```
int main () {
  ofstream myfile;
  myfile.open ("example.txt");
  myfile << "Writing this to a file.\n";
  myfile.close();
  return 0;
}
```

There will be 2 text file for this task input_file.txt and output_file.txt. these two file names come as some arguments to the main program. You need to open the input_file.txt for reading and output_file for writing. If the file is not created already do create a file with this name. Now read the file character by character and write it back to output_file. After completing the entire input_file to output_file
Do close the 2 files.

Debugging in C++:
Steps to Remember:
Consider the following Code as an example

The following coding example illustrates how does a debugger executes the code, steps into which method's body, how does it steps out and when does it step over a function.

```
int a =5; int b=6; int c =3;
 if( a> b && a < c){
    cout << "A is the greatest of all three integers "<< a << endl;
}
  else if (b> a && b > c){
    cout << "B is the greatest of all three integers "<< b << endl;
}
  else {
    cout << "C is the greatest of all three integers << c << endl;
}

 return 0;
 }
```

The below task is to understand the Watching and Debugging the code segments.

Task # 3
The problem is to get a positive integer from the keyboard and find all its factors by division method. After completing all factor also produce the sum of all the factors. The program should stop when a user entered -1 as input.

Creating a watch for the iteration on factor and sum to find whether the process is correct or not.

Task # 4

This task is to understand the Watching and Debugging of functions and their effects.

Write the task # 3 code as function to return factors and sum. See effectively how step into a function and step out of a function works.

Pointers:
A pointer variable is one that contains the address, not the value, of a data object. A pointer can be declared as,
Int *intptr;
There are few important operations, which we will do with the pointers very frequently. (a) We define a pointer variable. (b) Assign the address of a variable to a pointer. (c) Finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations.

```
int main()

{

int var = 20; // actual variable declaration.
int *ip;       // pointer variable
ip = &var;
 // store address of var in pointer variable
// print the address stored in ip pointer
//Code
// access the value at the address available in pointer
   //Code
return 0;
}
```

Task # 5

Find the largest and the smallest element in an array using pointers. Create a function named MinMax having addresses of Min and Max from main function, use pointers so that it will directly affect the values of the variables defined inside the main function.

Dynamic Allocation of Objects:

[Dynamically allocate memory to 5 objects of a class.]
Just like basic types, objects can be allocated dynamically, as well.

But remember, when an object is created, the constructor runs. Default constructor is invoked unless parameters are added:

```
Fraction * fp1, * fp2, * flist;
fp1 = new Fraction;              // uses default constructor
fp2 = new Fraction(3,5);         // uses constructor with two parameters

flist = new Fraction[20];        // dynamic array of 20 Fraction objects
// default constructor used on each
```

Deallocation with delete works the same as for basic types:

```
delete fp1;
delete fp2;
delete [] flist;
```

Notation: dot-operator vs. arrow-operator:

dot-operator requires an object name (or effective name) on the left side

```
objectName.memberName                // member can be data or function
```

The arrow operator works similarly as with structures.

```
pointerToObject->memberName
```

Remember that if you have a pointer to an object, the pointer name would have to be dereferenced first, to use the dot-operator:

```
(*fp1).Show();
```

Arrow operator is a nice shortcut, avoiding the use or parentheses to force order of operations:

```
fp1->Show(); // equivalent to (*fp1).Show();
```

When using dynamic allocation of objects, we use pointers, both to single object and to arrays of objects. Here's a good rule of thumb:

For pointers to single objects, arrow operator is easiest:

```
fp1->Show();
fp2->GetNumerator();
fp2->Input();
```

For dynamically allocated arrays of objects, the pointer acts as the array name, but the object "names" can be reached with the bracket operator. Arrow operator usually not needed:

```
flist[3].Show();
flist[5].GetNumerator();

// note that this would be INCORRECT, flist[2] is an object, not a pointer flist[2]->Show();
```

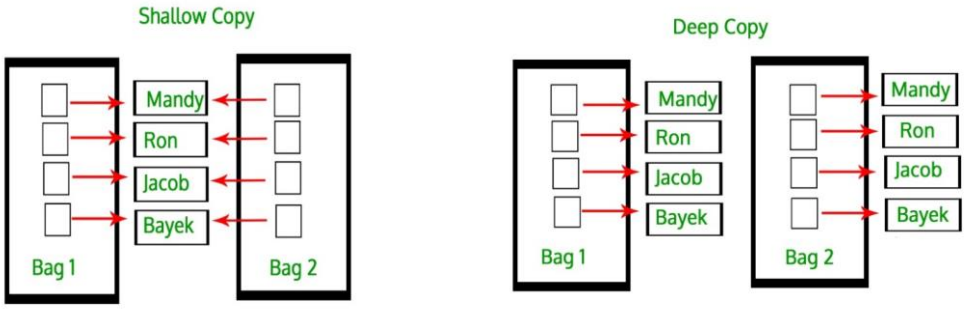
Task # 6 create an Animal class (having appropriate attributes and functions) and do following steps.

1. Dynamically allocate memory to 5 objects of a class.
2. Order data in allocated memories by Animal names in ascending order.

Rule of Three:

The Rule of Three is a rule of thumb in C++ (prior to C++ 11) which states that if a class defines one or more of the following, then it should probably explicitly define all three of these. The rule is also called Law of Big Three or The Big Three, three special functions which are:

1) Destructor
2) Copy Constructor
3) Overloaded/Copy Assignment Operator.



Example:

```
class Numbers{
//private members:

//public members

  //copy constructor
  /*
  Numbers(const ClassName &obj)
  {
Some code
  }*/

  ~Numbers() //destructor
  {
    delete ptr;
  }
};

int main(){

  int arr[ 4 ] = { 11, 22, 33, 44 };
  Numbers Num1( 4, arr );
  // this creates problem (if not using the copy constructor)
  Numbers Num2( Num1 );

  return 0;
}
```

Task # 7 The task is to understand the concept of shallow and deep copies, and to understand the logic behind Rule of Three.

Write a code that create a Class named Numbers, with private size and pointer variables , a public function to assign values to private member and a destructor to destroy the pointer and memory. Understand what

happens when you define an object by providing one object as an argument to another or simply assign one object to another.

Reference:

Using command line arguments in C++

- **argc :** Argument Count (argc) is integer type and cannot be negative. It represents number of command-line arguments passed by the user including the absolute name of the program.
- **argv:** ARGument Vector (argv) is array of character pointers listing all the arguments.
- If argc is greater than zero, the array elements from argv[0] to argv[argc-1] will contain pointers to strings. Argv[0] is the name of the program , After that till argv[argc-1] every element is command -line arguments.

Filling as Command Line arguments:

- Write a program that read a file containing only one line as an input argument (input_file.txt)
- The program writes the line back into a new file as output. (output_file.txt)
- File operations in C/C++

Watching and Debugging C/C++ Programs:

- Creating a watch list of variables, seeing the values during the execution of the program.
- Stepping into a function and Stepping out of the function.
- Identifying potential value watch for debugging.

Pointers:

- Use of pointers, storing the address of a variable into a pointer variable and accessing the value it points.
- Value of the variables changes when the address hold by pointer variables changes.

Dynamic Allocation of Objects:

- Memory allocation and deallocation of objects in variables and arrays using default and copy constructors.

Rule of Three:

- Creating a shallow copy and then calling a destructor without explicit definition of copy constructor and assignment operator.
- Creating deep copy by explicitly defining copy constructor and assignment operator to resolve the anomaly caused when the destructor is called twice or more.

Lab1: Programming and Debugging		
Std Name:		**Std_ID:**
Lab1-Tasks	**Completed**	**Checked**
Task #1		
Task #2		
Task #3		
Task# 4		
Task# 5		
Task# 6		
Task# 7		

Instructor signature: _____

Data Structures Lab
Session 2

Course: Data Structures (CL2001) **Semester:** Fall 2021
Instructor: Farah Sadia **T.A:** N/A

Note:
- Maintain discipline during the lab.
- Listen and follow the instructions as they are given.
- Just raise your hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

Lab Title: Learn to implement a DynamicSafeArray with one/two dimensional pointers.
Objectives: Get the knowledge of how dynamic memory is used to implement 1D and 2D arrays which are more powerful as compared to the default array mechanism of the programming language C/C++ supports.
Tool: Dev C++

1D & 2D Array:

A **one-dimensional** array (or single dimension array) is a type of linear array. Accessing its elements involves a single subscript which can either represent a row or column index.
Syntax:

```
float mark[5];
int mark[5] = {19, 10, 8, 17, 9};
int mark[] = {19, 10, 8, 17, 9};
```

Like a 1D array, a **2D array** is a collection of data cells, all of the same type, which can be given a single name. However, a 2D array is organized as a matrix with a number of rows and columns.
Syntax:

```
float x[3][4];
int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};
        int c[][3] = {{1, 3, 0}, {-1, 5, 9}};
        int c[2][3] = {1, 3, 0, -1, 5, 9};
```

Task # 1

Write a C++ program to read elements in a matrix and check whether the matrix is an Identity matrix or not.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Identity matrix

Dynamic Memory Allocation for arrays:

Memory in your C++ program is divided into two parts

1. The **`stack –`** All variables declared inside the function will take up memory from the stack.
2. The **`heap –`** this is unused memory of the program and can be used to allocate the memory dynamically when the program runs.

A **dynamic array** is an array with a big improvement: **automatic resizing**.
One limitation of arrays is that they're fixed size, meaning you need to specify the number of elements your array will hold ahead of time.
A dynamic array expands as you add more elements. So you don't need to determine the size ahead of time.

Strengths:

1. **Fast lookups**. Just like arrays, retrieving the element at a given index takes O(1) time.
2. **Variable size.** You can add as many items as you want, and the dynamic array will expand to hold them.
3. **Cache-friendly.** Just like arrays, dynamic arrays place items right next to each other in memory, making efficient use of caches.

Weaknesses:

1. **Slow worst-case appends.** Usually, adding a new element at the end of the dynamic array takes **O(1)** time. But if the dynamic array doesn't have any room for the new item, it'll need to expand, which takes **O(n)** time.
2. **Costly inserts and deletes.** Just like arrays, elements are stored adjacent to each other. So adding or removing an item in the middle of the array requires "scooting over" other elements, which takes **O(n)** time.

Factors impacting performance of Dynamic Arrays:

The array's initial size and its growth factor determine its performance. Note the following points:

1. If an array has a **small size** and a **small growth factor**, it will keep on **reallocating** memory more often. This will **reduce** the performance of the array.
2. If an array has a **large size** and a **large growth factor**, it will have a **huge chunk** of **unused** memory. Due to this, resize operations may take longer. This will reduce the performance of the array.

The new Keyword:

In C++, we can create a dynamic array using the **new keyword**. The number of items to be allocated is specified within a pair of square brackets. The type name should precede this. The requested number of items will be allocated.

Syntax:

```
int *ptr1 = new int;
int *ptr1 = new int[5];
int *array { new int[10]{}};
int *array { new int[10]{1,2,3,4,5,6,7,8,9,10}};
```

Resizing Arrays:

The length of a dynamic array is set during the allocation time.However, C++ doesn't have a built-in mechanism of resizing an array once it has been allocated. You can, however, overcome this challenge by allocating a new array dynamically, copying over the elements, then erasing the old array.

Dynamically Deleting Arrays:

A dynamic array should be deleted from the computer memory once its purpose is fulfilled. The delete statement can help you accomplish this. The released memory space can then be used to hold another set of data. However, even if you do not delete the dynamic array from the computer memory, it will be deleted automatically once the program terminates.

Syntax:

```
delete ptr;
delete[] array;
```

NOTE: To delete a dynamic array from the computer memory, you should use delete[], instead of delete. The [] instructs the CPU to delete multiple variables rather than one variable. The use of delete instead of delete[] when dealing with a dynamic array may result in problems. Examples of such problems include **memory leaks, data corruption, crashes,** etc.

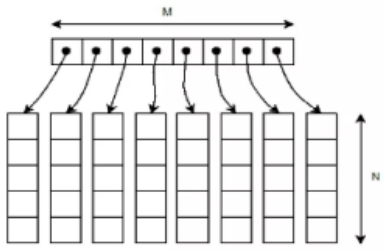
Example:

Single Dimensional Array:

```
#define N 10
// Dynamically Allocate Memory for 2D Array in C++
int main(){
    // dynamically allocate memory of size N
    // assign values to allocated memory
    // print the 1D array
    // deallocate memory
}
```

Two Dimensional Array Using Array of Pointers:

We can dynamically create an array of pointers of size M and then dynamically allocate memory of size N for each row as shown below.



Example:

```
// Dynamically Allocate Memory for 2D Array in C++
int main(){
        // dynamically create array of pointers of size M
        int** A = new int*[M];
        /* initialize random seed: */
        srand (time(NULL));
        // dynamically allocate memory of size N for each row
        for (int i = 0; i < M; i++)
                A[i] = new int[N];

        // assign values to allocated memory
        // print the 2D array

        // deallocate memory using delete[] operator
        for (int i = 0; i < M; i++)
                delete[] A[i];
        delete[] A;
        return 0;
}
```

Task # 2

Write a program that will read **10 integers** from the keyboard and place them in an **array**. The program then will sort the array into **ascending** and **descending** order and print the sorted list. The program must not change the original array or not create any other integer arrays.

Task # 3

Write a C++ program to **rearrange** a given **sorted** array of **positive** integers . Note: In the final array, **first element** should be **maximum** value, **second minimum** value, **third second maximum** value , **fourth second minimum** value, **fifth third maximum** and so on.

Safe Array:

In C++, there is **no check** to determine whether the **array index** is **out of bounds**.
During program execution, an out-of-bound array index can cause **serious problems**. Also, recall that
in C++ the array index starts at 0.
Safe array solves the out-of-bound array index problem and allows the user to begin the array index starting at any integer, positive or negative.
"**Safely**" in this context would mean that access to the array elements must not be **out of range**. ie. the position of the element must be **validated** prior to access.

For example in the member function to allow the user to set a value of the array at a particular location:

```
void set(int pos, Element val){     //set method
        if (pos<0 || pos>=size){
                cout<<"Boundary Error\n";
        }
        else{
                Array[pos] = val;
        }
}
```

Task # 4

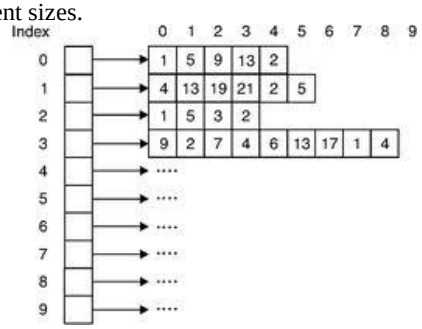
Write a program that creates a **2D array** of **5x5** values of **type boolean**. Suppose indices represent people and that the value at **row i**, **column j** of a **2D array** is true just in case **i and j** are **friends** and **false otherwise**. Use initializer list to instantiate and initialize your array to represent the following configuration: (*** means "friends"**)

	0	1	2	3	4
0		*		*	*
1	*		*		*
2		*			
3	*				*
4	*	*		*	

Write a method to check whether **two people** have a **common friend**. For example, in the example above, **0 and 4** are **both friends with 3** (so they have a common friend), whereas **1 and 2** have **no common friends**.

Jagged Array:

Jagged array is nothing but it is **an array of arrays** in which the member arrays can be in different sizes.



Example:

```
int **arr = new int*[3];
        int Size[3];
        int i,j,k;
for(i=0;i<3;i++){
cout<<"Row "<<i+1<< " size: ";
cin>>Size[i];
arr[i] =new int[Size[i]];
}
for(i=0;i<3;i++){
for(j=0;j<Size[i];j++){
cout<<"Enter row " <<i+1<<" elements: ";
cin>>*(*(arr + i) + j);
}
}
// print the array elements
// deallocate memory using delete[] operator
```

Task # 5

Write a program to calculate the GPA of students of all subjects of a single semester . Assume all the courses have the same credit hour (let's assume 3 credit hours).

	Data Structure	**Programming for AI**	**Digital Logic Design**	**Probability & Statistics**	**Finance & Accounting**
Ali	3.66	3.33	4.0	3.0	2.66
Hiba	3.33	3.0	3.66	3.0	---
Asma	4.0	3.66	2.66	---	---
Zain	2.66	2.33	4.0	---	---
Faisal	3.33	3.66	4.0	3.0	3.33

Lab2: DynamicSafeArray with one/two dimensional pointers		
Std Name:		**Std_ID:**
Section:		
Lab1-Tasks	**Completed**	**Checked**
Task #1		
Task #2		
Task #3		
Task# 4		
Task# 5		

Instructor signature: _____

Data Structures Lab
Session 3

Course: Data Structures (CS2001) **Semester:** Fall 2021
Instructor: Shahbaz Siddiqui **T.A:** N/A

Note:

- Lab manual cover following below recursion topics
  {Base Condition, Direct and Indirect Recursion, Tailed Recursion, Nested Recursion, Backtracking}
- Maintain discipline during the lab.
- Just raise hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

Base Condition in Recursion

Sample Code

```
int Funct(int n)
{   if (n < = 1) // base case      return 1;
   else
      return Funct (n-1);
}
```

Key Points: In the above example, base case for n < = 1 is defined and larger value of number can be solved by converting to smaller one till base case is reached.

Task-1:

a. Generate the following sequence with recursive approach
   1 , 3 , 6 , 10 , 15 , 21 , 28 . . . .
b. Generate the following sequence with recursive approach
   1 , 1 , 2 , 4 , 7 , 11 , 16 , 22 . . . .

Direct and Indirect Recursion

Sample Code (Direct Recursion)

```
void X()
{   // Some code....
   X();
   // Some code...
}
```

Sample Code (In-Direct Recursion)

```
void indirectRecFun1()
{   // Some code...
   indirectRecFun2();
   // Some code...
}
void indirectRecFun2()
{    // Some code...
   indirectRecFun1();
   // Some code...
}
```

Task-2:

a. **Write a indirect recursive code for the above task-1 (a,b) part with same approach as defined in the above sample code of In-Direct Recursion**

Tailed and Non Tailed Recursion

Sample Code (Non tailed Recursion)

```
void Funct (int a)
{
   if (a < 1)  return;
   cout << "The current Output is  " << n;

   // recursive call
   Funct (n-1);
}
```

Sample Code (Tailed Recursion)

```
unsigned Funct 1(int n, int a)
{
   if (n == 1)  return a;

   return Funct 1(n-1, n*a);
}


 int Funct 2(unsigned int n)
 {
```

```
  return Funct 1(n, 1);
 }

int main()
{
   cout << Funct 2(5);
   return 0;
}
```

Task 3:
Sort The Unsorted Numbers with both tail recursive and Normal recursive approach
Sample Input and Output
Given array is
12 11 13 5 6 7
Sorted array is
5 6 7 11 12 13

Nested Recursion

Sample Code

```
#include <iostream>
using namespace std;

int fun(int n)
{
   if (n > 100)
      return n - 10;

   // A recursive function passing parameter
   // as a recursive call or recursion inside
   // the recursion
   return fun(fun(n + 11));
}
int main()
{
   int r;
   r = fun(95);

   cout << " " << r;

   return 0;
}
```

Task 4:

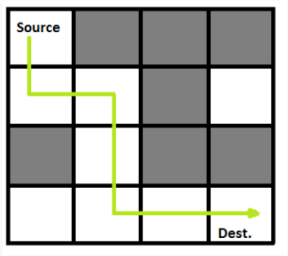
Dry run the outputs of the upper code in order to find out how the recursive calls are made

Backtracking

Sample Pseudocode

```
void findSolutions(n, other params) :
    if (found a solution) :
        solutionsFound = solutionsFound + 1;
        displaySolution();
        if (solutionsFound >= solutionTarget) :
            System.exit(0);
        return

    for (val = first to last) :
        if (isValid(val, n)) :
            applyValue(val, n);
            findSolutions(n+1, other params);
            removeValue(val, n);
```

A Maze is given as N*N binary matrix of blocks where source block is the upper left most block i.e., maze[0][0] and destination block is lower rightmost block i.e., maze[N-1][N-1]. A rat starts from source and has to reach the destination. The rat can move only in two directions: forward and down.



In the maze matrix, 0 means the block is a dead end and 1 means the block can be used in the path from source to destination. Note that this is a simple version of the typical Maze problem. For example, a more complex version can be that the rat can move in 4 directions and a more complex version can be with a limited number of moves.

Task-5

A. Design the function with recursive approach to find the number of existing destination path in the above provided sample code link
B. Change the Maze with following configuration .Find the optimal path to reach the destination with recursive approach

```
int maze[N][N] = { { 0, 0, 0, 1 },
                  { 0, 1, 1, 1 },
                  { 0, 1, 1, 0 },
```

Sample Code
https://www.geeksforgeeks.org/rat-in-a-maze-backtracking-2/

Reference:

Base Condition in Recursion

- In the recursive program, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems

All recursive algorithms must have the following:

1. Base Case (i.e., when to stop)
2. Work toward Base Case
3. Recursive Call (i.e., call ourselves)

The "work toward base case" is where we make the problem simpler (e.g., divide list into two parts, each smaller than the original). The recursive call, is where we use the same algorithm to solve a simpler version of the problem. The base case is the solution to the "simplest" possible problem (For example, the base case in the problem 'find the largest number in a list' would be if the list had only one number... and by definition if there is only one number, it is the largest).

Direct and Indirect Recursion

- A function X is called direct recursive if it calls the same function X. A function X is called indirect recursive if it calls another function say X_new and X_new calls fun directly or indirectly.

Tailed and Non Tailed Recursion

- In Tail Recursion you complete your calculations first, then call the recursive function, passing the results of your current step to the next recursive step. As a result, the last

sentence is written as (return (recursive-function prams)). The return value of any recursive action is essentially the same as the return value of the next recursive call.

- It's worth noting that tail recursion is essentially the same as looping. It's not merely a question of compiler optimization.

Nested Recursion:

- In this recursion, a recursive function will pass the parameter as a recursive call. That means **"recursion inside recursion".**
- In this recursion, there may be more than one functions and they are calling one another in a circular manner

Bactracking

- Backtracking is an improvement of the brute force approach. It systematically searches for a solution to a problem among all available options. In backtracking, we start with one possible option out of many available options and try to solve the problem if we are able to solve the problem with the selected move then we will print the solution else we will backtrack and select some other option and try to solve it. If none if the options work out we will claim that there is no solution for the problem
- Backtracking is a form of recursion. The usual scenario is that you are faced with a number of options, and you must choose one of these. After you make your choice you will get a new set of options; just what set of options you get depends on what choice you made. This procedure is repeated over and over until you reach a final state. If you made a good sequence of choices, your final state is a goal state; if you didn't, it isn't
- Three Types of Backtracking solution which are **Decision Problem** – In this, we search for a feasible solution.**Optimization Problem** – In this, we search for the best solution.
  Enumeration Problem – In this, we find all feasible solutions.

Lab3: Implementing Recursion, solving a simple maze		
Std Name:		**Std_ID:**
Lab1-Tasks	**Completed**	**Checked**
Task #1		
Task #2		
Task #3		
Task# 4		
Task# 5		

Data Structures Lab
Session 4

Course: Data Structures (CS2001) **Semester:** Fall 2021
Instructor: **T.A:** N/A

Note:

- Lab manual cover following below elementary sorting algorithms **{Bubble, insertion, selection, shell sort}**
- Maintain discipline during the lab.
- Just raise hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

Task-1:

Given an array of strings arr[]. Sort given strings using Bubble Sort and display the sorted array.

Key Points:

1. Bubble Sort, the two successive strings arr[i] and arr[i+1] are exchanged whenever arr[i]> arr[i+1]. The larger values sink to the bottom and hence called sinking sort. At the end of each pass, smaller values gradually "bubble" their way upward to the top and hence called bubble sort.

Task-2:

Develop C++ solution such that day month and year are taken as input for 5 records and perform Sorting Dates based on year using Selection Sort.

Key Points:

void selectionSort(int *array, int size) {
Find the smallest element in the array and exchange it with the element in the first position.
Find the second smallest element in the array and exchange it with the element in the second position.
Continue this process until done.
}

Task-3:

Develop an implementation of insertion sort that moves larger items to the right one position rather than doing full exchanges.

Key Points:

void insertionSort (int *array, int size) {
Choose the second element in the array and place it in order with respect to the first element.
Choose the third element in the array and place it in order with respect to the first two elements.

Continue this process until done.
Insertion of an element among those previously considered consists of moving larger elements one position to the right and then inserting the element into the vacated position

}

Task 4:

Given an array **arr[ ]** of length **N** consisting cost of **N** toys and an integer **K** the amount with you. The task is to find maximum number of toys you can buy with **K** amount.

Example 1:
Input:
N = 7
K = 50
arr[] = {1, 12, 5, 111, 200, 1000, 10}
Output: 4
Explaination: The costs of the toys
you can buy are 1, 12, 5 and 10.
Example 2:
Input:
N = 3
K = 100
arr[] = {20, 30, 50}
Output: 3
Explaination: You can buy all toys

Task 5:

Given an unsorted array arr[0..n-1] of size n, find the minimum length subarray arr[…] such that sorting this subarray makes the whole array sorted.
Examples:
1) If the input array is [10, 12, 20, 30, 25, 40, 32, 31, 35, 50, 60], your program should be able to find that the subarray lies between the indexes 3 and 8.
2) If the input array is [0, 1, 15, 25, 6, 7, 30, 40, 50], your program should be able to find that the subarray lies between the indexes 2 and 5.

Key Points:

Traverse the array from left to right tracking the max, saving the last found index value of 'j' which will be less than max
Traverse the array from right to left tracking the min, saving the last found index value of 'i' which will be greater than max
Sort the segment array from index i to j

Task 6:

A clerk at a shipping company is charged with the task of rearranging a number of large crates in order of the time they are to be shipped out. Thus, the cost of compares is very low relative to the cost of exchanges (move the crates). The warehouse is nearly full: there is extra space sufficient to hold any one of the crates, but not two. Which sorting method should the clerk use?

Discussion:

- Selection sort builds the final diagonal by exchanging the next-smallest element into position. The main cost of selection sorts is the comparisons required to find that element. Comparisons are not depicted in this representation except for a delay to make the time take for a comparison comparable to the cost of an exchange. The algorithm is slow at the beginning (because it has to scan through most of the array to find the next minimum) and fast at the end (because it has to scan through only a few elements).Selection sort is easy to implement; there is little that can go wrong with it. However, the method requires O(N^2) comparisons and so it should only be used on small files. There is an important exception to this rule. When sorting files with large records and small keys, the cost of exchanging records controls the running time. In such cases, selection sort requires O(N) time since the number of exchanges is at most N.

- Bubble sort works like selection sort, but its cost is explicit in this representation because it uses exchanges (data movement) to move the minimum element from right to left, one position at a time.

- Insertion sort maintains a ordered subarray that appears as a diagonal of black dots that moves from left to right sweeping up each new element that it encounters. Each new element is inserted into the diagonal. Insertion sort is an O(N^2) method both in the average case and in the worst case. For this reason, it is most effectively used on files with roughly N < 20. However, in the special case of an almost sorted file, insertion sort requires only linear time.

- Time complexity of shell sort is O(n^2). Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left. Shell algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as **interval**. This interval is calculated based on Knuth's formula as −

  Knuth's Formula
  h = h * 3 + 1
  where −
  h is interval with initial value 1

Reference:

For algorithms:
http://web.mit.edu/1.124/LectureNotes/sorting.html
For complexities and comparison
https://www.geeksforgeeks.org/analysis-of-different-sorting-techniques/
Algorithm visualizer
https://visualgo.net/en

Lab4: Elementary Sorting Techniques		
Std Name:		**Std_ID:**
Lab1-Tasks	**Completed**	**Checked**
Task #1		
Task #2		
Task #3		
Task# 4		
Task# 5		

Data Structures Lab

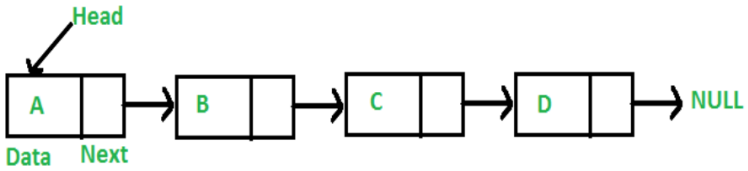
Session 5

Course: Data Structures (CS2001) **Semester:** Fall 2021
Instructor: Muhammad Rafi **T.A:** N/A

Note:

- Maintain discipline during the lab.
- Listen and follow the instructions as they are given.
- Just raise hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

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



Task # 1: Implement a SinglyLinkedList class

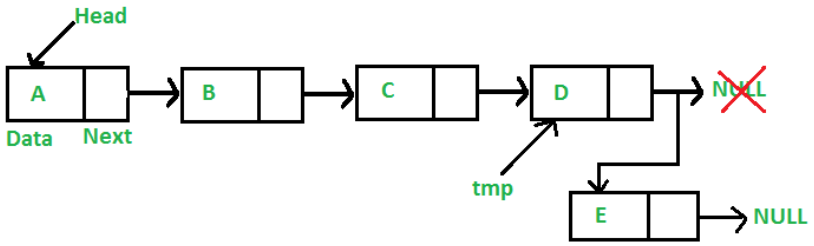
Singly Linked List is an important bifurcation of Linked List data structure.
It is called Singly as it holds one data member and one link member associated to each node in the list. In order to create a SinglyLinkedList class, we first need a helper class to implement nodes. Each node object will have 3 public data members (a) Int type Key (unique to each node) (b) data (c) next pointer. Along that we will use a default constructor and a parameterized constructor of Node class to manipulate those data members.
The SinglyLinkedList class will only have a public Node type pointer variable "head" to point to the first node of the list. Together with the help of default and parameterized constructor the head's value will be manipulated in the SinglyLinkedList class.

```
//Node Object
Class Node
{
        //public members: key, data, next
         Node ()
        {
                //initialize both key and data with zero while next pointer with NULL;
        }
```

```
        Node (int k, int d)
        {
                //assign the data members initialized in default constructor to these arguments.
        }
};

//SinglyLinkedList Object
Class SinglyLinkedList
{
        // create head node as a public member
        //Default Constructor
        SinglyLinkedList()
        {
                //initialize the head pointer will NULL;
        }
        //Parameterized Constructor with node type 'n' pointer
        SinglyLinkedList (Node* n)
        {
                //Assign the head pointer to value of pointer n;
        }
};
```

Task # 2: Add a node at the end of a Singly Linked List.

Add a Node at the End:
In this task, the new node is always added after the last node of the given Linked List.
For example, if the given Linked List is 5->10->15->20->25 and we add an item 30 at the end, then the Linked List becomes 5->10->15->20->25->30.
Since a Linked List is typically represented by the head of it, we have to traverse the list till the end and then change the next to last node to a new node.



To append a node at the end of the list, first check if the If there exists a node already with that key or not. For that you may need a helper function inside the SinglyLinkedList class to perform the test. In case a node already exists with that key value, Intimate the programmer to use another key value to append a node. On the contrary, If the node the doesn't exist, append a node at the end. Before that check if the list has some node or not, i.e. Check if the head pointer is null or not. If it is null, access

the head and assign node n to it. Otherwise, traverse through the list to find the that node whose next is Null, i.e. the last node in the list. Then, assign the node n to next pointer of the last node. Also, make sure the next pointer of node n (new last node) is null.

```
//Creating a Node type helper Function named nodeExists (argument key)
 Node nodeExists (int k)
{
        //temporary pointer var to hold Null value;
        //node type pointer to hold head pointers value;
        // loop condition (Traverse through the node until the ptr variable points to null)
        {
         If the ptr variable's key = passed key argument
        {
                //assign the pointer ptr to temp variable;
        }
         Else
        {
                //assign the pointer ptr to point to the next pointer's value.
        }
        return the temp variable;
}

//Append Function
 Void appendNode (Node* n, int data)
{
        //create new node and assign data to it.
        //check if the head pointer points to Null or not with a condition
         if (head  == NULL)
        {
                //If it does, assign the head pointer the passed pointer 'n'. This will put the address of
        node n in the head pointer.
        }

        // Else traverse through the list to find the next pointer holding Null as address (last node)
        Else
        {
                if (nodeExists(n->key) != NULL  ) {
                   // Print an intimation that a node holding passed key already exists.
                }
                else
                {
                        If (head != NULL)
                        {
                        // assign the head pointer to a new pointer of type Node (say, ptr).
                        //loop through the list using ptr until the next of any node contains null.
                        //when ptr->next = NULL. Then assign the n node to ptr->next to append that
                        node after the last found node.
                        }
```

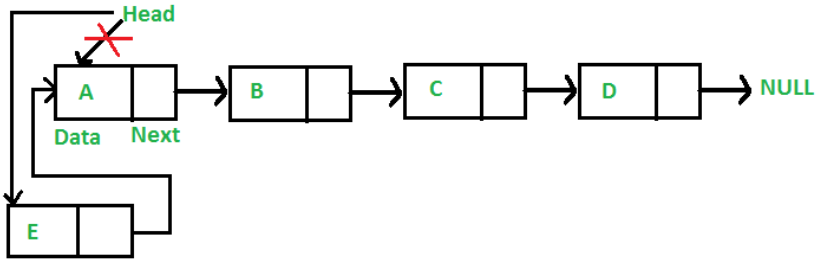
Commented [A1]: Do we need it?

Commented [A2R1]:

```
        }
}}
```

Task # 3: Add a node at the front of a Singly Linked List (Prepend a new node)

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 the Linked List.
For example, if the given Linked List is 10->15->20->25 and we add an item 5 at the front, then the Linked List becomes 5->10->15->20->25.
Let us call the function that adds at the front of the list is push ().
The push () must receive a pointer to the head pointer, because push must change the head pointer to point to the new node.



To prepend a node, simply check if the key that is passed already exists or not, if yes, intimate the programmer to pass a new key value or else assign the new head node's value to the next pointer of new node. Then set the new head to be the new node's address.

```
 //Prepending a Node.

 void prependNode(Node* n)
{
// checking the value of node's key if the node with that key already exists.

    if (nodeExists(n->key) != NULL  ) {
   // Print an intimation that a node holding passed key already exists.
    }
    else
    {
      // new node's next is pointing to the head i.e. address of first node.
      // since we have changed the head pointer's value from first node to the new node, now the new
.     // head will be pointing to address of new node.
    }
}
```

Task # 4: Add a node after a given node in a Singly Linked List

To insert a new node after some node, create a void function named insertNodeAfter () carrying two arguments, one for the key of the node after which the insertion is to be done, the new node.

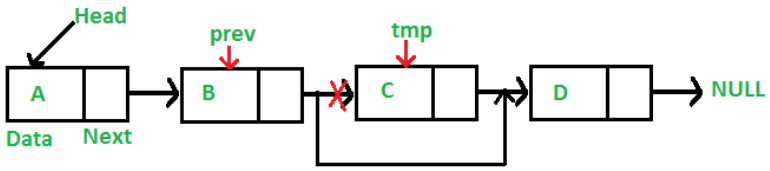
```
//Inserting a new node after some node.

Void insertAfterNode (key , new node)
{
        //creating a node type pointer that calls nodeExists () and passes the key argument into it to
        check if there exists a node with this key value
         Node * ptr = nodeExists(k);
         If(ptr == NULL)
        {
                //print a message saying no node exists with that key
        }
        Else
        {
                //check if any key with that already exists to avoid duplication
                if (nodeExists(n->key) != NULL  )
                {
                        //Print an intimation that a node holding passed key already exists. Append a new
                node with different key value.
                }
                Else
                {
                        //now the next pointer of new node will be holding the address kept in next
                        pointer of ptr.
                        //assign the address of node to the next pointer of the previous node (ptr).
                        //print a message that anode is inserted;
}
```

Task # 5

Delete a node from a Singly Linked List

- Delete Last node
- Delete any other node

To delete a node from the linked list, we need to do the following steps.
1) Find the previous node of the node to be deleted.
2) Change the next of the previous node to hold the node next to the node to be deleted.
3) Free memory for the node to be deleted.




```
void deleteNode(key)
{
    // create a Node type pointer to hold head (say, temp)
    // create a Node type pointer to hold previous node (say, prev)

    // check if head contains the key
    if (temp!=NULL && temp->key==key){
            // assign next of temp to temp, which will unlink the node pointed by head
            // delete the temp node
            // return
            }
    Else{
            While(temp!=NULL && temp->key != key){          // traverse the list until temp is not
                                                            NULL //and temp's key is same as the key.
                    // set prev to temp
                    // set temp to temp's next pointer
            }
            If(temp == NULL){    // key not found.
                    // return
                    }
            // unlink by setting temp's next to prev's next.
            // free memory by deleting temp
            }
}
```

Task # 6

Update a node in a Singly Linked List

Updating Linked List or modifying Linked List means replacing the data of a particular node with the new data. Implement a function to modify a node's data when the key is given. First, check if the node exists using the helper function nodeExists.

```
void updateNode(key, new_data)
{
```

```
        // call the nodeExists function and hold the return value in a Node type pointer (say, ptr)
        If(ptr!=NULL){
                // set ptr's data as new data
        }
        Else{
                // print a message sating node does not exist.
        }
}
```

Task # 7

Solve the following problem using a Singly Linked List.

Given a Linked List of integers, write a function to modify the linked list such that all even numbers appear before all the odd numbers in the modified linked list. Also, keep the order of even and odd numbers same.
Examples:

```
Input: 17->15->8->12->10->5->4->1->7->6->NULL
Output: 8->12->10->4->6->17->15->5->1->7->NULL

Input: 8->12->10->5->4->1->6->NULL
Output: 8->12->10->4->6->5->1->NULL

// If all numbers are even then do not change the list
Input: 8->12->10->NULL
Output: 8->12->10->NULL

// If all numbers are odd then do not change the list
Input: 1->3->5->7->NULL
Output: 1->3->5->7->NULL
```

ab5: Singly Linked List		
Std Name:		**Std_ID:**
Lab1-Tasks	**Completed**	**Checked**
Task #1		
Task #2		
Task #3		
Task# 4		
Task# 5		
Task# 6		
Task# 7		

Instructor signature: _____

Data Structures Lab

Session 06

Course: Data Structures (CS2001) **Semester:** Fall 2021
Instructor: **T.A:** N/A

Note:

- Maintain discipline during the lab.
- Listen and follow the instructions as they are given.
- Just raise hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

Doubly Link List

```
class Node {
	public:
		int key;
		int data;
		Node * next;
		Node * previous;

		 Node() {
			key = 0;
			data = 0;
			next = NULL;
			previous = NULL;
		}

		Node(int k, int d) {
			key = k;
			data = d;
		}
};

class DoublyLinkedList {
	public:
		Node * head;

		DoublyLinkedList() {
			head = NULL;
		}

		DoublyLinkedList(Node * n) {
			head = n;
	}
```

```
        appendNode();
        prependNode();
        insertNodeAfter();
        deleteNodeByKey();
        updateNodeByKey();
};
```

Task-1:

Create a doubly link list and perform the mentioned tasks.

i. Insert a new node at the end of the list.
ii. Insert a new node at the beginning of list.
iii. Insert a new node at given position.
iv. Delete any node.
v. Print the complete doubly link list.

Circular Link List

```
class Node {
        public:
                int key;
                int data;
                Node * next;

                Node() {
                        key = 0;
                        data = 0;
                        next = NULL;
                }

                Node(int k, int d) {
                        key = k;
                        data = d;
                }
};

class CircularLinkedList {
        public:
                Node * head;

                CircularLinkedList() {
                        head = NULL;
                }


        appendNode();
        prependNode();
        insertNodeAfter();
        deleteNodeByKey();
```

```
        updateNodeByKey();
        print();
};
```

Task-2:

Create a circular link list and perform the mentioned tasks.

i. Insert a new node at the end of the list.
ii. Insert a new node at the beginning of list.
iii. Insert a new node at given position.
iv. Delete any node.
v. Print the complete circular link list.

Circular Double Link List

```
class  node
{
public:
                int info;
                node *next;
                node *prev;
};

class double_clist
{
        public:
        node *create_node(int);

        insert_begin();
        insert_last();
        insert_pos();
        delete_pos();
         update();
        display();
        node  *start, *last;

        double_clist()
        {
                start = NULL;
                last = NULL;
        }
};
```

Task-3:

Create a circular Double link list and perform the mentioned tasks.

i. Insert a new node at the end of the list.
ii. Insert a new node at the beginning of list.
iii. Insert a new node at given position.

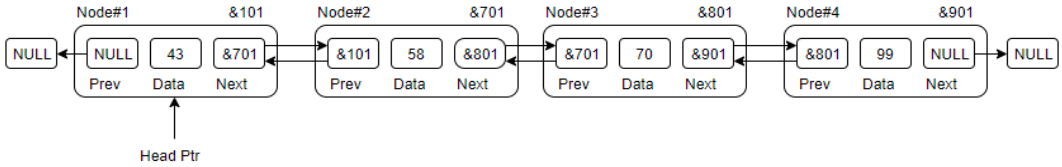
iv. Delete any node.
v. Print the complete circular double link list.

Task-4:
Give an efficient algorithm for concatenating two doubly-linked lists **L** and **M**, with head and tail preserved nodes, into a single list that contains all the nodes of **L** followed by all the nodes of **M**.

Reference:

1. Doubly Link List

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward or backward easily as compared to Single Linked List.



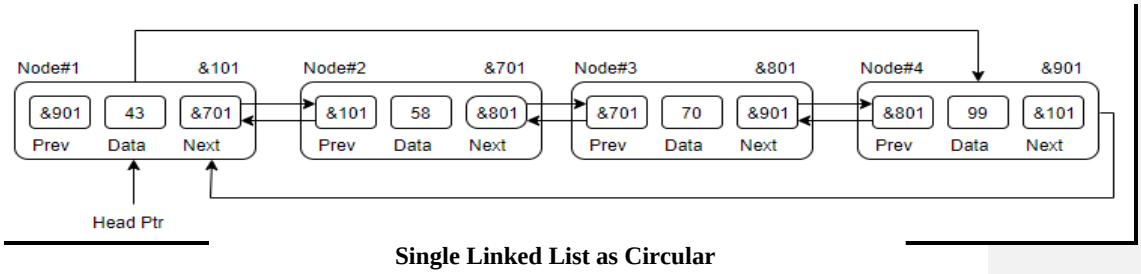
- **Link** − each link of a linked list can store a data called an element.
- **Next** − each link of a linked list contains a link to the next link called Next.
- **Prev** − each link of a linked list contains a link to the previous link called Prev.

2. Circular Link List

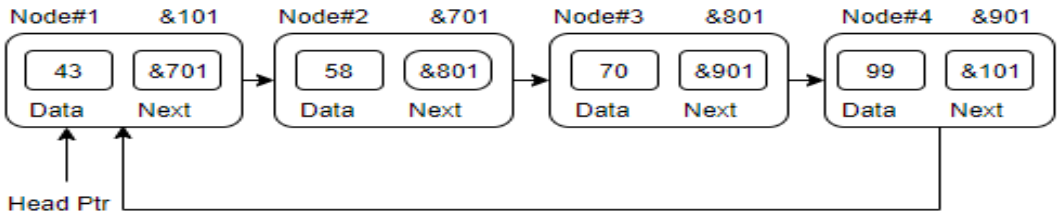
In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.

- The last link's next points to the first link of the list in both cases of singly as well as doubly linked list.
- The first link's previous points to the last of the list in case of doubly linked list.

Doubly Linked List as Circular



Single Linked List as Circular



Basic Operations

- Traverse
- Insert
- Display
- Append
- Delete
- Prepend
- Count

Lab6: Doubly Link List & Circular Link List		
Std Name:		**Std_ID:**
Lab1-Tasks	**Completed**	**Checked**
Task #1		
Task #2		
Task #3		
Task# 4		
Task# 5		
Task# 6		

Data Structures Lab
Session 7

Course: Data Structures (CS2001) **Semester:** Fall 2021
Instructor: Shahbaz Siddiqui **T.A:** N/A

Note:

- Lab manual cover following below Stack and Queue topics
  {Stack with Array and Linked list , Application of Stack, Queue with Array and Linked List , Application of Queue }
- Maintain discipline during the lab.
- Just raise hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

Stack with Array

Sample Code of Stack in Array

```
class Stack {
    int top;

public:
    int a[MAX]; // Maximum size of Stack

    Stack() { top = -1; }
    bool push(int x);
    int pop();
    int peek();
    bool isEmpty();
};

bool Stack::push(int x)
{
    if (top >= (MAX - 1)) {
        cout << "Stack Overflow";
        return false;
    }
    else {
        a[++top] = x;
        cout << x << " pushed into stack\n";
        return true;
    }
}

int Stack::pop()
```

```
{
   if (top < 0) {
      cout << "Stack Underflow";
      return 0;
   }
   else {
      int x = a[top--];
      return x;
   }
}
int Stack::peek()
{
   if (top < 0) {
      cout << "Stack is Empty";
      return 0;
   }
   else {
      int x = a[top];
      return x;
   }
}

bool Stack::isEmpty()
{
   return (top < 0);
}
```

Task-1:

A. Design a Main class of upper code which perform the below task

1. Insert 10 Integers values in the stack
2. If the Insert input reach the Highest index of Array display the message Stack overflow
3. Remove the Inserted values till the Last value and print the message that stack is empty

Stack with Linked list

Sample Code of Stack in Array

```
struct Node
{
   int data;
   struct Node* link;
};

struct Node* top;
```

```
// Utility function to add an element
// data in the stack insert at the beginning
void push(int data)
{

   // Create new node temp and allocate memory
   struct Node* temp;
   temp = new Node();

   // Check if stack (heap) is full.
   // Then inserting an element would
   // lead to stack overflow
   if (!temp)
   {
      cout << "\nHeap Overflow";
      exit(1);
   }

   // Initialize data into temp data field
   temp->data = data;

   // Put top pointer reference into temp link
   temp->link = top;

   // Make temp as top of Stack
   top = temp;
}
```

Task-2:

A. Design a Main class of upper code which perform the below task

1. Insert 10 Integers values in the stack
2. Write a utility function for upper code to display all the inserted integer values in the linked list in forward and reverse direction both
3. Write utility function to pop top element from the stack

Application of Stack (convert infix expression to postfix)

Sample Pseudocode

```
Begin
  initially push some special character say # into the stack
  for each character ch from infix expression, do
    if ch is alphanumeric character, then
      add ch to postfix expression
    else if ch = opening parenthesis (, then
      push ( into stack
    else if ch = ^, then            //exponential operator of higher precedence
      push ^ into the stack
    else if ch = closing parenthesis ), then
      while stack is not empty and stack top ≠ (,
        do pop and add item from stack to postfix expression
      done

      pop ( also from the stack
    else
      while stack is not empty AND precedence of ch <= precedence of stack top element, do
        pop and add into postfix expression
      done

      push the newly coming character.
  done

  while the stack contains some remaining characters, do
    pop and add to the postfix expression
  done
  return postfix
End
```

Code Snippet

```
#include<bits/stdc++.h>
using namespace std;
//Function to return precedence of operators
int prec(char c)
{
if(c == '^')
return 3;
else if(c == '*' || c == '/')
return 2;
else if(c == '+' || c == '-')
return 1;
else
return -1;
}
```

```
int main()
{
   string exp = "a+b*(c^d-e)^(f+g*h)-i";
   infixToPostfix(exp);
   return 0;
}
```

Task-3:

A. Use the Upper code snippet implement the utility function with the help of array based stack **infixToPostfix** by using sample pseudocode

Queue with Array

```
using namespace std;

// A structure to represent a queue
class Queue {
public:
   int front, rear, size;
   unsigned capacity;
   int* array;
};

// function to create a queue
// of given capacity.
// It initializes size of queue as 0
Queue* createQueue(unsigned capacity)
{
   Queue* queue = new Queue();
   queue->capacity = capacity;
   queue->front = queue->size = 0;

   // This is important, see the enqueue
   queue->rear = capacity - 1;
   queue->array = new int[queue->capacity];
   return queue;
}
```

Task-4:

A. Use the Upper code snippet implement the following utility function in the Array based Queue
   1. Write a function **QueueCapacity** when the Queue is Full
   2. Write a function **ADDMember** when a new integer value is added in the array
   3. Write a function **RemoveMember** when any data member is remove from the queue

Queue with Linked list

Sample Code

```
#include <iostream>
using namespace std;
struct node {
  int data;
  struct node *next;};
struct node* front = NULL;
struct node* rear = NULL;
struct node* temp;
void Insert() {
  int val;
  cout<<"Insert the element in queue : "<<endl;
  cin>>val;
  if (rear == NULL) {
    rear = (struct node *)malloc(sizeof(struct node));
    rear->next = NULL;
    rear->data = val;
    front = rear;
  } else {
    temp=(edlist struct node *)malloc(sizeof(struct node));
    rear->next = temp;
    temp->data = val;
    temp->next = NULL;
  rear = temp;
  }}
```

Task-5:

B. Use the Upper code snippet implement the following utility function in the Link based Queue

4. Write a function **QueueCapacity** when the Queue is Full
5. Write a function **ADDMember** when a new integer value is added in the linkedlist
6. Write a function **RemoveMember** when any data member is remove from the queue

Application of Queue

Task-6: Read the Web link https://algs4.cs.princeton.edu/24pq/ .Implement the Elementary implementation of Priority queue using linked list as explain in Figure "Sequence of Operation in Priority Queue" .

Note: Your Base code will be in C++

Reference:

Stack with Array and Linked list

- Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out).

Mainly the following three basic operations are performed in the stack:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false

Application of Stack

- An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are –

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

Infix Notation

We write expression in **infix** notation, e.g. a - b + c, where operators are used **in**-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Prefix Notation

In this notation, operator is **prefix**ed to operands, i.e. operator is written ahead of operands. For example, **+ab**. This is equivalent to its infix notation **a + b**. Prefix notation is also known as **Polish Notation**.

Postfix Notation

This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfix**ed to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a + b**.

Sr.No.	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) * c	* + a b c	a b + c *
3	a * (b + c)	* a + b c	a b c + *
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) * (c + d)	* + a b + c d	a b + c d + *
6	((a + b) * c) - d	- * + a b c d	a b + c * d -

Table-1

Queue with Array and Linked list

- A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.
- **Following are the Operations on Queue**
  1.Enqueue: Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.
  2.Dequeue: Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.
  3.Front: Get the front item from queue.
  4.Rear: Get the last item from queue.

Applications of Queue

- A priority queue in c++ is a type of container adapter, which processes only the highest priority element, i.e. the first element will be the maximum of all elements in the queue, and elements are in decreasing order.

Difference between a queue and priority queue :

- Priority Queue container processes the element with the highest priority, whereas no priority exists in a queue.
- Queue follows First-in-First-out (FIFO) rule, but in the priority queue highest priority element will be deleted first.

- If more than one element exists with the same priority, then, in this case, the order of queue will be taken.

1. **empty()** – This method checks whether the priority_queue container is empty or not. If it is empty, return true, else false. It does not take any parameter.

2. **size()** – This method gives the number of elements in the priority queue container. It returns the size in an integer. It does not take any parameter.

3. **push()** – This method inserts the element into the queue. Firstly, the element is added to the end of the queue, and simultaneously elements reorder themselves with priority. It takes value in the parameter.

4. **pop()** – This method delete the top element (highest priority) from the priority_queue. It does not take any parameter.

5. **top()** – This method gives the top element from the priority queue container. It does not take any parameter.

6. **swap()** – This method swaps the elements of a priority_queue with another priority_queue of the same size and type. It takes the priority queue in a parameter whose values need to be swapped.

7. **emplace()** – This method adds a new element in a container at the top of the priority queue. It takes value in a parameter.

Lab3: Stack and Queue (using both Arrays and Linked List)		
Std Name:		**Std_ID:**
Lab1-Tasks	**Completed**	**Checked**
Task #1		
Task #2		
Task #3		
Task# 4		
Task# 5		
Task# 6		

Data Structures Lab
Session 8

Course: Data Structures (CS2001) **Semester:** Fall 2021
Instructor: **T.A:** N/A

Note:

- Lab manual cover following topics
  {Tree, BST, Design and implement classes for binary tree nodes and nodes for general tree, Traverse the tree with the three common orders, Operation such as searches, insertions, and removals on a binary search tree and its applications}
- Maintain discipline during the lab.
- Just raise your hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

BST

KeyPoint: A Binary Search Tree (BST) is a binary tree with the following properties:

- The left subtree of a particular node will always contain nodes whose keys are less than that node's key.
- The right subtree of a particular node will always contain nodes with keys greater than that node's key. The left and right subtree of a particular node will also, in turn, be binary search trees

Task-1:

Build functionality named autoGrader which will assist DS teacher to check students BST assignments such that if given tree is BST assign 10 points if not then assign 0.

BST Insertion

Sample Code of class Nodes

```
Create class Nodes
class Node {    private:
 int key;
 string name;

 Node leftChild;
  Node rightChild; public:
 Node(int key, string name) {

      this.key = key;
      this.name = name;

 }
string toString() {
```

```
        return cout<<name<< " has the key " <<key<<endl;
         } };
```

Task-2: Complete the following Code:

A. Create class BinaryTree and create a function which add nodes in BST

```
class BinaryTree {
  private:   Node root;
 public:
 void addNode(int key, string name) {

       ----------------------// Create a new Node and initialize it




       // If there is no root this becomes root

       if (root == NULL) {

             ----------------------------

       } else {

             // Set root as the Node we will start with as we traverse the
tree


             ----------------------------
             // Future parent for new Node

             Node parent;

             while (true) {

                   // root is the top set the parent node to the root node

                   -------------------------

                   // Check if the new node should go on
                   // the left side of the parent node
                   Key is compared with that of root. If the key is less than
                   root, it is compared with root's left child key. If greater,
                   it is compared with the root's right child. Continue this
                   process until the new node is compared with a leaf node and
                   added either on the right or left child depending on its
                   key.

                   }
```

B. Implement main.cpp for the code provided such that a given array is passed to form a BST{ 15, 10, 20, 8, 12, 16, 25 }

Tree Traversals: Inorder, PreOrder, PostOrder

Pseudo code For Inorder Traversal (iteration)

```
1) Create an empty stack S.
2) Initialize current node as root
3) Push the current node to S and set current = current->left until
current is NULL
4) If current is NULL and stack is not empty then
     a) Pop the top item from stack.
     b) Print the popped item, set current = popped_item->right
     c) Go to step 3.
5) If current is NULL and stack is empty then we are done.
```

Task-3:

A. Write recursive algorithms that perform preorder and inorder tree walks.

Preorder Traversal approach.

1. Visit Node.
2. Traverse Node's left sub-tree.
3. Traverse Node's right sub-tree

B. Write the iterative code for preorder traversal.

BST Deletion

BST Deletion

1) ***Node to be deleted is the leaf:*** Simply remove from the tree.

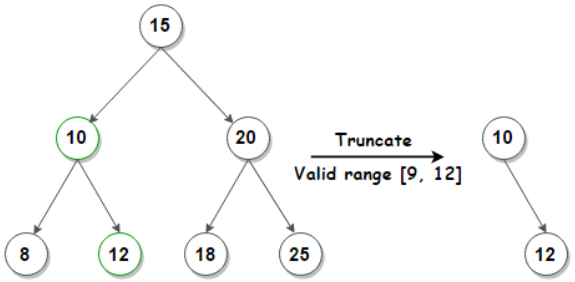
2) ***Node to be deleted has only one child:*** Copy the child to the node and delete the child

3) *Node to be deleted has two children:* Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

The important thing to note is, inorder successor is needed only when the right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in the right child of the node.

Task:4

Given a BST and a range of keys(values), remove nodes from BST that have keys outside the given range

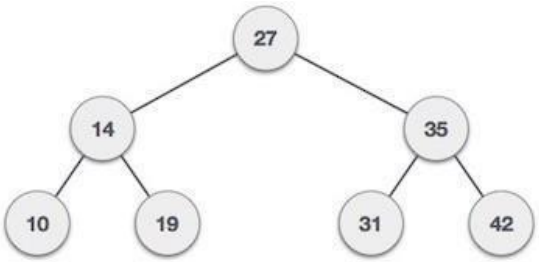


Summary Discussion

Binary search tree (BST)

Binary search tree (BST) or a lexicographic tree is a binary tree data structure which has the following binary search tree properties:

- Each node has a value.
- The key value of the left child of a node is less than to the parent's key value.
- The key value of the right child of a node is greater than (or equal) to the parent's key value.
- And these properties hold true for every node in the tree.



- **Subtree**: any node in a tree and its descendants.
- **Depth of a node**: the number of steps to hop from the current node to the root node of the tree.
- **Depth of a tree**: the maximum depth of any of its leaves.
- **Height of a node**: the length of the longest downward path to a leaf from that node.
- **Full binary tree**: every leaf has the same depth and every nonleaf has two children.
- **Complete binary tree**: every level except for the deepest level must contain as many nodes as possible; and at the deepest level, all the nodes are as far left as possible.
- **Traversal**: an organized way to visit every member in the structure.

Traversals

The binary search tree property allows us to obtain all the keys in a binary search tree in a sorted order by a simple traversing algorithm, called an in order tree walk, that traverses the left sub tree of the root in in order traverse, then accessing the root node itself, then traversing in in-order the right sub tree of the root node.

The tree may also be traversed in preorder or post order traversals. By first accessing the root, and then the left and the right sub-tree or the right and then the left sub-tree to be traversed in preorder. And the opposite for the post order.
The algorithms are described below, with Node initialized to the tree's root.

• **Preorder Traversal**

1. Visit Node.
2. Traverse Node's left sub-tree.
3. Traverse Node's right sub-tree.

• **In-order Traversal**

1. Traverse Node's left sub-tree.
2. Visit Node.

3. Traverse Node's right sub-tree

• Post-order Traversal

1. Traverse Node's left sub-tree.
2. Traverse Node's right sub-tree.
3. Visit Node

Searching

We use the following procedure to search for a node with a given key in a binary search tree. Given a pointer to the root of the tree and a key *k*, TREE-SEARCH returns a pointer to a node with key *k* if one exists; otherwise, it returns NIL.

```
TREE-SEARCH (x, k)
1 if x = NIL or k = key[x]
2     then return x
3  if k < key[x]
4     then return TREE-SEARCH (left[x], k)
5     else return TREE-SEARCH (right[x], k)
```

The procedure begins its search at the root and traces a path downward in the tree, as shown in Figure 13.2. For each node *x* it encounters, it compares the key *k* with *key*[*x*]. If the two keys are equal, the search terminates. If *k* is smaller than *key*[*x*], the search continues in the left subtree of *x*, since the binary-search-tree property implies that *k* could not be stored in the right subtree. Symmetrically, if *k* is larger than *key*[*k*], the search continues in the right subtree. The nodes encountered during the recursion form a path downward from the root of the tree, and thus the running time of TREE-SEARCH is *O*(*h*), where *h* is the height of the tree.

Reference:

http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap13.htm

Lab3: Binary Search Tree		
Std Name:		**Std_ID:**
Lab1-Tasks	**Completed**	**Checked**
Task #1		
Task #2		
Task #3		
Task# 4		