



School of Business and Technology
Department of Computer Science

Course Code: CSC 211: DATA STRUCTURES AND ALGORITHMS NOTES.

Course Lecturer: N. Mutua

e-mail: mutuanicholis@gmail.com

Contact cell: +254 723782525

Lecture Time: 2-5PM

Course Purpose

The primary objective of this course is to teach students structures and algorithms which will allow them to write efficient programs designed to retrieve and store large amounts of data. Students will gain skills on how data may be structured and instructions sequenced in algorithms and programmes as well as the relationship between appropriate data and control structures and tasks from the real world.

Learning Outcomes

At the end of the course the students should be able to:

1. Have gained knowledge and skills in organizing and manipulating data in the computer storage.
2. Understand data structures and the design and analysis of computer algorithms
3. Apply principles of abstraction and encapsulation as expressed by data structures
4. Analyze algorithms to determine their efficiency (in terms of computation and memory resources)

CHAPTER ONE:

INTRODUCTION TO DATA STRUCTURES:

- **Data Structure:** This is a systematic way to organize data in order to use it efficiently.
- **Interface:** Each data structure has an interface. Interface represents the set of operations that a data structure supports. An interface only provides the list of supported operations, type of parameters they can accept and return type of these operations.
- **Implementation** – Implementation provides the internal representation of a data structure. Implementation also provides the definition of the algorithms used in the operations of the data structure.

Characteristics of a Data Structure

- **Correctness** – Data structure implementation should implement its interface correctly.
- **Time Complexity** – Running time or the execution time of operations of data structure must be as small as possible.
- **Space Complexity** – Memory usage of a data structure operation should be as little as possible.

Need for Data Structure

As applications are getting complex and data rich, there are three common problems that applications face now-a-days.

- **Data Search** – Consider an inventory of 1 million-(10^6) items of a store. If the application is to search an item, it has to search an item in 1 million-(10^6) items every time slowing down the search. As data grows, search will become slower.
- **Processor speed** – Processor speed although being very high, falls limited if the data grows to billion records.
- **Multiple requests** – As thousands of users can search data simultaneously on a web server, even the fast server fails while searching the data.

Data can be organized in a data structure in such a way that all items may not be required to be searched, and the required data can be searched almost instantly.

Execution Time Cases

There are three cases which are usually used to compare various data structure's execution time in a relative manner.

- **Worst Case** – This is the scenario where a particular data structure operation takes maximum time it can take. If an operation's worst case time is $f(n)$ then this operation will not take more than $f(n)$ time where $f(n)$ represents function of n .
- **Average Case** – This is the scenario depicting the average execution time of an operation of a data structure. If an operation takes $f(n)$ time in execution, then m operations will take $mf(n)$ time.

- **Best Case** – This is the scenario depicting the least possible execution time of an operation of a data structure. If an operation takes $f(n)$ time in execution, then the actual operation may take time as the random number which would be maximum as $f(n)$.

Basic Terminology

- **Data** – Data are values or set of values.
- **Data Item** – Data item refers to single unit of values.
- **Group Items** – Data items that are divided into sub items are called as Group Items.
- **Elementary Items** – Data items that cannot be divided are called as Elementary Items.
- **Attribute and Entity** – An entity is that which contains certain attributes or properties, which may be assigned values.
- **Entity Set** – Entities of similar attributes form an entity set.
- **Field** – Field is a single elementary unit of information representing an attribute of an entity.
- **Record** – Record is a collection of field values of a given entity.
- **File** – File is a collection of records of the entities in a given entity set.

LOCAL ENVIRONMENT SETUP

If you are still willing to set up your environment for C programming language, you need the following two tools available on your computer,

- (a) Text Editor and
- (b) The C Compiler.

Text Editor

This will be used to type your program. Examples of few editors include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim.

The C Compiler

The source code written in the source file is the human readable source for your program. It needs to be "compiled", to turn into machine language so that your CPU can actually execute the program as per the given instructions.

This C programming language compiler will be used to compile your source code into a final executable program.

Most frequently used and free available compiler is GNU C/C++ compiler. Otherwise, you can have compilers either from HP or Solaris if you have respective Operating Systems (OS).

Installation on UNIX/Linux

If you are using Linux or UNIX, then check whether GCC is installed on your system by entering the following command from the command line

Installation on Mac OS

If you use Mac OS X, the easiest way to obtain GCC is to download the Xcode development environment from Apple's website and follow the simple installation instructions. Once you have Xcode setup, you will be able to use GNU compiler for C/C++.

Installation on Windows

To install GCC on Windows, you need to install MinGW. To install MinGW, go to the MinGW homepage, www.mingw.org and follow the link to the MinGW download page. Download the latest version of the MinGW installation program, which should be named MinGW-<version>.exe.

BASIC CONCEPTS OF ALGORITHMS

An algorithm is a sequence of steps that must be carried out in order to accomplish a particular task. Three things are to be considered while writing an algorithm: input, process, and output. The input that we give to an algorithm is processed with the help of the procedure and finally, the algorithm returns the output.

HISTORY OF ALGORITHMS.

The origin of the word algorithm is indirectly linked to India. A scholar in the, ‘House of wisdom’ in Baghdad, Abu Abdullah Muhammad Ibn Musa Al Khwarizmi, wrote a book about Indian numerals in which rules of performing arithmetic with such numerals were discussed. These rules were referred to as ‘algorism’ from which the word algorithm was derived. His book was translated into Latin in the 18th century. This was followed by the invention of Boolean algebra by George Boole and the creation of language in special symbols by Frege. The concept of algorithms, given its present form by a genius named Alan Turing, helped in the inception of artificial intelligence. Algorithms have been used for long in mathematics and computer science. Euclid’s theorem and the algorithm of Archimedes to calculate the value of ‘Pi’ are classic examples of algorithms. These events reinforced the belief that if a task is to be performed, then it must be performed with a predefined sequence of steps that are unambiguous and efficient. However, this belief would be challenged in the late 20th century with the introduction of non-deterministic algorithms. Not only in mathematics and computer science are there algorithms, they are part of our daily lives. When a person is taught how to make tea, even then an algorithm is edified. Algorithms are camouflaged as directions and rules, which form the basis of our existence. The challenge, however, is to make these algorithms efficient and robust.

Features/characteristics of algorithms.

Not all procedures can be called an algorithm. An algorithm should have the following characteristics –

- ❖ **Unambiguous** – Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- ❖ **Input** – An algorithm should have 0 or more well-defined inputs.
- ❖ **Output** – An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- ❖ **Finiteness** – Algorithms must terminate after a finite number of steps.
- ❖ **Feasibility** – Should be feasible with the available resources.

- ❖ **Independent** – An algorithm should have step-by-step directions, which should be independent of any programming code.
- ❖ It should be efficient both in terms of memory and time.

From the data structure point of view, following are some important categories of algorithms.

- **Search** – Algorithm to search an item in a data structure.
- **Sort** – Algorithm to sort items in a certain order.
- **Insert** – Algorithm to insert item in a data structure.
- **Update** – Algorithm to update an existing item in a data structure.
- **Delete** – Algorithm to delete an existing item from a data structure.

IMPORTANCE OF ALGORITHMS

- It helps in enhancing the thinking process. They are like brain stimulants that will give a boost to our thinking process.
- It helps in solving many problems in computer science, computational biology, and economics.
- Without the knowledge of algorithms we can become a coder but not a programmer.
- A good understanding of algorithms will help us to get a job. There is an immense demand of good programmers in the software industry who can analyse the problem well.
- Genetic algorithms and randomized approach will help us to retain that job in the changing market.

WAYS OF WRITING AN ALGORITHM

There are three basic ways of writing algorithms in programming. They include:

1. English like algorithm
2. Flowcharts
3. pseudocodes

English-Like Algorithm.

Problem – Design an algorithm to add two numbers and display the result.

```

Step 1 - START
Step 2 - declare three integers a, b & c
Step 3 - define values of a & b
Step 4 - add values of a & b
Step 5 - store output of step 4 to c
Step 6 - print c
Step 7 - STOP
  
```

Alternatively:




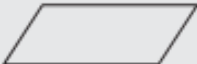


```

Step 1 - START ADD
Step 2 - get values of a & b
Step 3 -  $c \leftarrow a + b$ 
Step 4 - display c
Step 5 - STOP
  
```

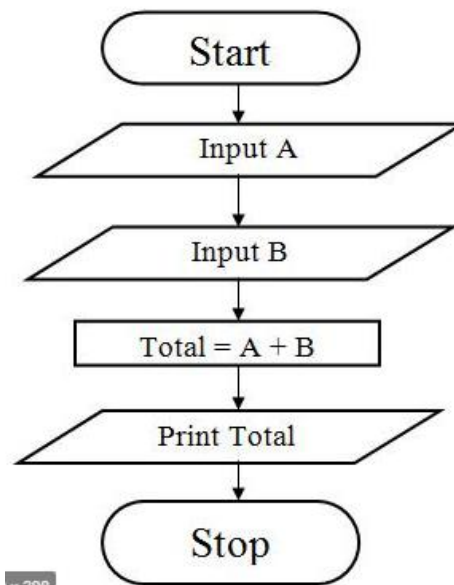
Flowchart.

This is a graphical representation of a computer program in relation to its sequence of functions (as distinct from the data it processes). Flowcharts use simple geometric shapes to depict processes and arrows to show relationships and process/data flow.

Flowcharts pictorially depict a process.

S. No.	Name	Element Representation	Meaning
1.	Start/End		An oval is used to indicate the beginning and end of an algorithm.
2.	Arrows		An arrow indicates the direction of flow of the algorithm.
3.	Connectors		Circles with arrows connect the disconnected flowchart.
4.	Input/Output		A parallelogram indicates the input or output.
5.	Process		A rectangle indicates a computation.
6.	Decision		A diamond indicates a point where a decision is made.

Problem – Design an algorithm to add two numbers and display the result.



Pseudocode.

This is a notation resembling a simplified programming language, used in program design. The pseudocode has an advantage of being easily converted into any programming language. This way of writing algorithm is most acceptable and most widely used. In order to write a pseudocode, one must be familiar with the conventions of writing it.

THEY INCLUDE:

1. *Single line comments start with //*
2. *Multi-line comments occur between /* and */*
3. *Blocks are represented using brackets. Blocks can be used to represent compound statements or the procedures.*
4. *Statements are delimited by semicolon.*
5. *Assignment statements indicates that the result of evaluation of the expression will be stored in the variable.*
6. *The boolean expression 'x > y' returns true if x is greater than y, else returns false.*
7. *The boolean expression 'x < y' returns true if x is less than y, else returns false.*
8. *The boolean expression 'x <= y' returns true if x is less than or equal to y, else returns false.*
9. *The boolean expression 'x >= y' returns true if x is greater than or equal to y, else returns false.*
10. *The boolean expression 'x != y' returns true if x is not equal to y, else returns false.*
11. *The boolean expression 'x == y' returns true if x is equal to y, else returns false.*
12. *The boolean expression 'x AND y' returns true if both conditions are true, else returns false.*
13. *The boolean expression 'x OR y' returns true if any of the conditions is true, else returns false.*
14. *The boolean expression 'NOT y' returns true if the result of x evaluates to false, else returns false.*
15. *if< condition >then< statement >*
16. *This condition is an enhancement of the above 'if' statement. It can also handle the case where the condition isn't satisfied.*

Problem – Design an algorithm to add two numbers and display the result.

1. BEGIN
2. NUMBER a, b, sum
3. OUTPUT(“input number a”)
4. INPUT a
5. OUTPUT(“input number b”)
6. INPUT b
7. Sum = a + b
8. OUTPUT sum
9. END

Data Definition

Data Definition defines a particular data with the following characteristics.

- **Atomic** – Definition should define a single concept.
- **Traceable** – Definition should be able to be mapped to some data element.

- **Accurate** – Definition should be unambiguous.
- **Clear and Concise** – Definition should be understandable.

Data Object

Data Object represents an object having a data.

Data Type

Data type is a way to classify various types of data such as integer, string, etc. which determines the values that can be used with the corresponding type of data, the type of operations that can be performed on the corresponding type of data. There are two data types –

- Built-in Data Type
- Derived Data Type

Built-in Data Type

Those data types for which a language has built-in support are known as Built-in Data types. For example, most of the languages provide the following built-in data types.

- Integers
- Boolean (true, false)
- Floating (Decimal numbers)
- Character and Strings

Derived Data Type

Those data types which are implementation independent as they can be implemented in one or the other way are known as derived data types. These data types are normally built by the combination of primary or built-in data types and associated operations on them. For example –

- List
- Array
- Stack
- Queue

Basic Operations

The data in the data structures are processed by certain operations. The particular data structure chosen largely depends on the frequency of the operation that needs to be performed on the data structure.

- Traversing
- Searching
- Insertion
- Deletion
- Sorting
- Merging

Array Data Structures

Array is a very basic data structure provided by every programming language. Let's talk about an example scenario where we need to store ten employees' data in our C/C++ program including name, age and salary. One of the solutions is to declare ten different variables to store employee name and ten more to store age and so on. Also you will need some sort of mechanism to get information about an employee, search employee records and sort them. To solve these types of problem C/C++ provide a mechanism called Arrays.

An array is simply a number of memory locations, each of which can store an item of data of the same data type and which are all referenced through the same variable name. Array may be defined abstractly as finite order set of homogeneous elements. So we can say that there are finite numbers of elements in an array and all the elements are of same data type. Also array elements are ordered i.e. we can access a specific array element by an index.

Terms

- **Element** – Each item stored in an array is called an element.
- **Index** – Each location of an element in an array has a numerical index, which is used to identify the element.

The general form of declaring a simple (one dimensional)
Array is `array _type variable name[array_size];`

in your C/C++ program you can declare an array like
`int Age[10];`

Here array type declares base type of array which is the type of each element in array. In our example array type is `int` and its name is `Age`. Size of the array is defined by `array_size` i.e. 10. We can access array elements by index, and first item in array is at index 0. First element of array is called lower bound and its always 0. Highest element in array is called upper bound. In C programming language upper and lower bounds cannot be changed during the execution of the program, so array length can be set only when the program is written.

Array has 10 elements

Age 0	Age 1	Age 2	Age 3	Age 4	Age 5	Age 6	Age 7	Age 8	Age 9
30	32	54	32	26	29	23	43	34	5

Note: One good practice is to declare array length as a constant identifier. This will minimise the required work to change the array size during program development. Considering the array we declared above we can declare it like.

```
#define NUM_EMPLOYEE 10 int Age[NUM_EMPLOYEE];
```

Initialising an array

Initialisation of array is very simple in C programming. There are two ways you can initialise arrays.

- Declare and initialise array in one statement.
- Declare and initialise array separately.

Look at the following C code which demonstrates the declaration and initialisation of an array.

```
int Age [5] = { 30, 22, 33, 44, 25};
```

Alternatively,

```
int Age [5];  
Age [0]=30;  
Age [1]=22;  
Age [2]=33;  
Age [3]=44;  
Age [4]=25;
```

Array can also be initialised in a way that array size is omitted, in such case compiler automatically allocates memory to array.

```
int Age [ ] = { 30, 22, 33, 44, 25};
```

Let's write a simple program that uses arrays to print out number of employees having salary more than 3000.

Array in C Programming

```
#include <windows.h>  
#include <stdio.h>  
#include <stdlib.h>  
#define NUM_EMPLOYEE 10  
int main(int argc, char *argv[]){  
int Salary[NUM_EMPLOYEE], lCount=0,gCount=0,i=0;  
printf("Enter employee salary (Max 10)\n ");  
for (i=0; i<NUM_EMPLOYEE; i++){  
printf("\nEnter employee salary: %d - ",i+1);  
scanf("%d",&Salary[i]); } for(i=0; i<NUM_EMPLOYEE; i++)  
{
```

```

    if(Salary[i]<3000)
    lCount++;
    else gCount++;
}
printf("\nThere are {%d} employee with salary more than 3000\n",gCount);
printf("There are {%d} employee with salary less than 3000\n",lCount); printf("Press ENTER to
continue...\n");
getchar();
return 0;
}

```

Array in C++ Programming

```

#include <cstdlib>
#include <iostream>
#define NUM_EMPLOYEE 10 using namespace std;
int main(int argc, char *argv[])
{
    int Salary[NUM_EMPLOYEE], lCount=0,gCount=0,i=0;
    cout << "Enter employee salary (Max 10) " << endl;
    for (i=0; i<NUM_EMPLOYEE; i++)
    {
        cout << "Enter employee salary: - " << i+1 << endl;
        cin >> Salary[i];
    }
    for(i=0; i<NUM_EMPLOYEE; i++)
    {
        if(Salary[i]<3000) lCount++;
        else
        gCount++;
    }
    cout << "There are " << gCount << " employee with salary more than 3000" << endl
    << "There are " << lCount << " employee with salary less than 3000" << endl;
    system("PAUSE");
    return EXIT_SUCCESS;
}

```

Basic Operations

Following are the basic operations supported by an array.

- **Traverse** – print all the array elements one by one.
- **Insertion** – Adds an element at the given index.
- **Deletion** – Deletes an element at the given index.
- **Search** – Searches an element using the given index or by the value.
- **Update** – Updates an element at the given index.

How to declare and initialize multi-dimensional arrays

Often there is need to manipulate tabular data or matrices. For example if employee salary is increased by 20% and you are required to store both the salaries in your program. Then you will need to store this information into a two dimensional arrays. C/C++ gives you the ability to have arrays of any dimension.

int Salary[10][2];

This defines an array containing 10 elements of type int (integer). Each of these elements itself is an array of two integers. So to keep track of each element of this array is we have to use two indices. One is to keep track of row and other is to keep track of column.

Initialising multidimensional arrays

Multidimensional arrays can also be initialised in two ways just like one dimensional array. Two braces are used to surround the row element of arrays. If you are initialising more than one dimension then you will have to use as many braces as the dimensions of the array are.

```
int Salary [5][2] = {
```

```
{2300, 460},
```

```
{3400, 680},
```

```
{3200, 640},
```

```
{1200, 240},
```

```
{3450, 690} };
```

```
int Salary [5][2] ={0}; //This will initialise all the array elements to 0
```

```
int Salary [5][2];
```

```
Salary [0][0]=2300;
```

```
Salary [1][0]=3400;
```

```
Salary [2][0]=3200;
```

```
Salary [3][0]=1200;
```

```
Salary [4][0]=3450;
```

```
Salary [0][1]=460;
```

```
Salary [1][1]=680;
```

```
Salary [2][1]=640;
```

```
Salary [3][1]=240;
```

```
Salary [4][1]=690;
```

The code below demonstrates two dimension arrays. It uses the same example of employee salary to increment it by 20% and adds it to actual salary then print current salary, increment and new salary. Two dimensional Array in C Programming

```
#include <windows.h>

#include <stdio.h>

#include <stdlib.h>

#define NUM_EMPLOYEE 10

int main(int argc, char *argv[]){ //initialise Salary of each employee

int Salary[NUM_EMPLOYEE][2]={

{2300,0},

{3400,0},

{3200,0},

{1200,0}, {3450,0},

{3800,0}, {3900,0},

{2680,0},

{3340,0},

{3000,0} };

int lCount=0,gCount=0,i=0;

for(i=0; i<NUM_EMPLOYEE; i++){

Salary[i][1] = ((Salary[i][0]*20)/100); }

printf("Initial Salary + Increment = Total Salary\n"); for (i=0; i<NUM_EMPLOYEE; i++)

{

printf("%d\t%d\t%d\n",Salary[i][0],Salary[i][1],Salary[i][0]+Salary[i][1]);

} printf("Press ENTER to continue...\n"); getchar(); return 0; } Two dimensional array in C++

Programming #include <cstdlib> #include <iostream> #define NUM_EMPLOYEE 10 using

namespace std; int main(int argc, char *argv[]){

//initialise Salary of each employee

int Salary[NUM_EMPLOYEE][2]={

{2300,0},
```

{3400,0},

{3200,0},

{1200,0},

{3450,0}

Using C++, write a program that implements algorithm for inserting data elements into one dimensional array? Solution:

```
#include <iostream .h>
```

```
include name space std
```

```
{
```

```
int i=0,x=0;
```

```
int a{};
```

```
for (i=0;i<10;i++)
```

```
cin>>a>>endl;
```

```
{
```

```
for(x=0;x<i;x++)
```

```
{
```

```
cout<<a<<endl;
```

```
}
```

Review Questions

EXERCISE 1. Using C++,write a program that implements algorithm for inserting data elements into one dimensional array?

EXERCISE 2. Illustrate the concept of single linked list using C + + programming language.

EXERCISE 3. Write a C++ program that implements the algorithms for pushing, popping and deleting data elements from the stack data structure EXERCISE 4. Illustrate how queue data structure is different from stack data structure

EXERCISE 5. Construct a binary tree and apply the three traversal techniques on the following expression (A+B)*(C-D)

EXERCISE 6. Discuss the concept of graph data structure?

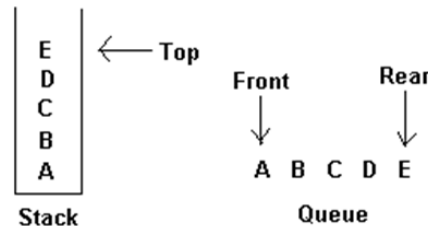
CHAPTER TWO:

2.0 Stack Data Structure

Two of the more common data objects found in computer algorithms are stacks and queues. Both of these objects are special cases of the more general data object, an ordered list.

A **stack** is an ordered list in which all insertions and deletions are made at one end, called the top.

A **queue** is an ordered list in which all insertions take place at one end, the rear, while all deletions take place at the other end, the front. Given a stack $S = (a[1], a[2], \dots, a[n])$ then we say that $a[1]$ is the bottommost element and element $a[i]$ is on top of element $a[i-1]$, $1 < i \leq n$. When viewed as a queue with $a[n]$ as the rear element one says that $a[i+1]$ is behind $a[i]$, $1 < i \leq n$.



The restrictions on a stack imply that if the elements A,B,C,D,E are added to the stack, in that order, then the first element to be removed/deleted must be E. Equivalently we say that the last element to be inserted into the stack will be the first to be removed. For this reason stacks are sometimes referred to as Last In First Out (LIFO) lists.

Is a data structure that utilizes the concept of last in First Out (LIFO).

All the operations take place at the top.

Areas where stack can be used include:

1. Plates in a cafeteria
2. Mathematical evaluations (postfix, infix, prefix)
3. Number conversion
4. Program execution

2.1. Operations of a stack

1. Dynamically initialize a stack (create)

By assigning a stack structure with $\text{Top} = 0$

2. Test whether the stack is full

If the $\text{top} = \text{max size of stack}$

3. Test whether the stack is empty

If the $\text{top} = 0$

4. Pushing items into the stack

(a) Increment top by 1

(b) Insert the item

5. Popping items from the stack (Remove the item)

Decrease top by 1

2.1.1. The stack class

Class stack

{

Int top;

Int stackarray[50];

Public:


```

Stack( );
Int emptystack( );
Int fullstack( );
Void push(int item);
Void pop(int &item);
};

```

Example _. Using stack structure write a program for displaying numbers in the reverse order

Solution:

```
#Include < iostream.h>
```

```

Const int maxsize = 20;
{
Int top;
Int stackarray[maxsize];
Public:
Int emptystack( );
Int fullstack( );
Void push(int item);
Void pop(int & item);
};
Stack : : stack( )
{
Top = 0;
}
Int stack : : emptystack( )
{
Return top == 0;
}
Int stack : : fullstack( )
{
Return top == maxsize;
} Void stack : : push(int item)
{
Top ++ ;
Stack away (top) = item;
}
Void stack : : Pop (int &item)
{
Item = stackarray[top]
top --;
}
void main( )
{
Stack S;

Int i,n,x;

```

```

<out << "\n how many numbers";
cin>>n;
cout << "\n Enter numbers in";
For (i=1;i <= n; i++)
{ cin>>x;
s.push(x);
}
While (! S . emptystack( ))
{
S . pop (x);
cout <<x<<"\n";
}
} _

```

2.1.2. Application of the stack

Example: Converting Numbers from Base 10 to any other given base

Algorithm

1. Request for the number
2. Request for the base to convert to
3. While number >0
 - i) Compute remainder
 - ii) Push remainder into the stack
 - iii) Compute next number (the quotient become the next number)
4. Display the content of stack (pop)

Example Using stack, write a program for converting a number from base 10 to any other base (1-9)

Solution

```

# include <iostream.h>
Const int max size = 50;
{
class stack
Int top;
Int stackarray(maxsize)
Public:
Stack ( );
Int emptystack( );
Int fullstack( );
Void push(int item);
};
{
Top = 0;
}
int stack : : emptystack ( )
{
Return top == 0;
}
Int stack: : fullstack ( )

```

```

{return top == max size;
}
Void stock : : push (int item)
{
Top ++;
Stackarray (top) = item;
}
Void stock : : pop (int pitem)
{
Item = stackarray (top);
Top = - ; }
Void main( )
{
Stack s ;
Int n ; // number
Int btest ; // base to convert to Int remainder; // remainder
cout << "\n Enter base to convert to “.
cin >>btest;
While (n>0)
{
Remainder= n % btest;
s.push (remainder);
n=n /btest;
}
While (! S.empty stack (j )
{
S.Pop (remainder);
Count << remainder << “ “;
}

```

Review Questions

EXERCISE 7. _

Explain briefly the meaning of the following terms

- i. data type
- ii. Abstract data type (ADT)
- iii. Pointers
- iv. Data structure

EXERCISE 8. _

For each of the following situations, which of these ADT's (1 through 4) would be most appropriate:

- i. A queue,
 - ii. A stack,
 - iii. A list,
 - iv. none of these.
- i. The customers at a Kenchicken's counter who take numbers to make their turn
 - ii. Integers that need to be sorted
 - iii. Arranging plates in the cafeteria

- iv. People who are put on hold when they call Kenya Airways to make reservations
- v. Converting infix to postfix expression

CHAPTER THREE:

3.0 Queue Data Structure

Queues are data structures that, like the stack, have restrictions on where you can add and remove elements. To understand a queue, think of a cafeteria line: the person at the front is served first, and people are added to the line at the back. Thus, the first person in line is served first, and the last person is served last. This can be abbreviated to First In, First Out (FIFO). The cafeteria line is one type of queue. Queues are often used in programming networks, operating systems, and other situations in which many different processes must share resources such as CPU time.

Queue is a data structure that utilizes the concept of first-in-First Out (FIFO) Inserting takes place at the rear and deleting takes place at the front In a circular queue we need a counter that keep track of the of element in a queue.

We need a generalized approach to compute the rear and the front position;

$\text{Rear} = (\text{rear} + 1) \% \text{maxsize},$

$\text{Front} = (\text{front} + 1) \% \text{maxsize}$

When inserting for the very first time, we need to adjust in position of front from zero to 1.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues –

- **enqueue()** – add (store) an item to the queue.
- **dequeue()** – remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are –

- **peek()** – Gets the element at the front of the queue without removing it.
- **isfull()** – Checks if the queue is full.
- **isempty()** – Checks if the queue is empty.

3.1. QUEUE Abstract Data Type

```
Const int maxsize =50;
```

```
Clas queue
```

```
{
```

```
Int rear,front,count;
```

```
Int queuearray [maxisize];
```

```
Public:
```

```
Queue ( );
```

```
int fullqueue ( );
```

```
Void insertqueue (int item)
```

```
Void deletequeue (int item);
```

```
};
```

```
Queue : : Queue ( )
```

```

{
rear = 0;
front = 0;
count = 0;
}
Int Queue :: emptyqueue ( )
{
Return count == 0;
}
Int Queue :: full queue ( )
{
Return count == maxsize;
}
Void Queue :: interqueue (int item)
{
If (count == 0)
Front ++;
Rear = (rear + 1) % maxsize;
Queuearray [rear ] = item;
Count ++;
}
Void Queue: : deletequeue (int item)
{
Item = queuearray [front];
Front = (front + 1)% maxsize;
Count -- ; }

```

Example

Using Queue structure, write a program for displaying numbers in the same order of insertion?

Solution:

```

Include <io stream.h>
Void main ( )
{
Queue q ;

Int I,n,x;
Count<< "\n Enter the Number \n";
For (I=1 ; i<=n ; I ++ )
{
C;>>x ;
Q .insert queue (x) ;
}
Cout << "\n show number \n";
While (q. ! emptyqueue ( ) )
{
Q . delete queue (x);
Count << x << "\n";
}
}

```

}

Review Questions

EXERCISE 10.

Describe how deletion of a node in between the linked list can be carried out illustrated your answer with a diagram?

EXERCISE 11.

Beginning with an empty binary search tree what binary search tree is formed when you insert the following values in the order

i. W,T,N,J,E,B,A

ii. A,B,W,J,N,T,E

EXERCISE 12. _

1. Explain the importance of a head node (1 mark)
2. State two advantages of linked list over arrays (2 marks)
3. Each element of a doubly linked structure has three fields. State the three fields illustrating your answer with a diagram (2 marks)
4. Describe the procedure of deleting an element at position P in a doubly linked list, illustrating your answer with a diagram (4 marks)
5. State one advantage of circular list (2 marks)

CHAPTER FOUR:

4.0 LINKED LIST DATA STRUCTURE

4.1. What is Linked List?

A linked list is a data structure that consists of a sequence of data records such that in each record there is a field that contains a reference (i.e., a link) to the next record in the sequence.

The linked list is a useful data structure that can dynamically grow according to data storage requirements. This is done by viewing data as consisting of a unit of data and a link to more units of data. Linked list is useful in the implementation of dynamic arrays, stacks, strings and sets. The link list is the basic ADT in some languages, for example, LISP.

Linked lists are most useful in environments with dynamic memory allocation.

With dynamic memory allocation dynamic arrays can grow and shrink with less cost than in a static memory allocation environment. Linked lists are also useful to manage dynamic memory environments. Dramatically a linear linked list can be viewed as follows:

Each data element has an associated link to the next item in the list. The last item in the list has no link. The first element of the list is called the head, the last element is called the tail.

Linked lists can be implemented in most languages. Languages such as Lisp and Scheme have the data structure built in, along with operations to access the linked list. Procedural languages, such as C, or object-oriented languages, such as C++ and Java, typically rely on mutable references to create linked lists.

Terms:-

- **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.
- **LinkedList** – A Linked List contains the connection link to the first link called First.

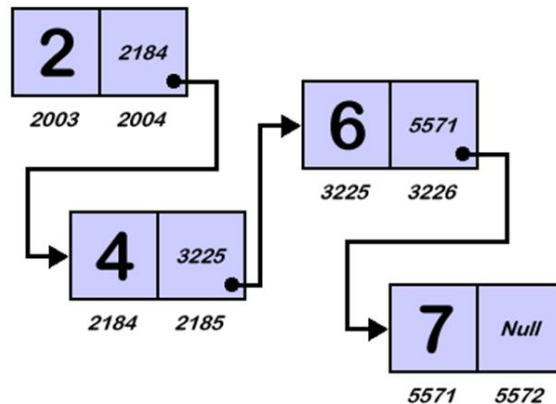
4.1.1. Pros and Cons of Linked Lists

Linked lists overcome all the major limitations of arrays such as:

The size of array is fixed. Though it can be deferred until the array is created at runtime, but after that it remains fixed.

If you allocate a large space for arrays, the most of the space is really wasted.

In case of small arrays declared, the code breaks in instances when more data elements are used than array size.



Inserting new elements in the front is potentially expensive because existing elements need to be shifted over to make room. Linked lists have their own strengths and weaknesses, but they happen to be strong where arrays are weak. The array's features all follow from its strategy of allocating the memory for all its elements in one block of memory. Linked lists use an entirely different strategy. Linked lists allocate memory for each element separately and only when necessary.

Linked List Basic by Nick Parlante

A user does not have to declare the size of the linked list at the beginning of the program; you can add and remove objects from the list by merely unlinking them.

In addition, you can sort the members of a linked list without actually moving data objects around by changing the links.

The cost of flexibility in a linked list is speed of access. You can't just reach in and grab the tenth element, for example, like you would in the case of an array. Instead, you have to start at the beginning of the list and link ten times from one object to the next.

4.1.2. What is the good and bad thing about linked list?

The good thing about a linked list is its runtime expandability. Unlike arrays, linked lists are much more flexible in dynamically allocating space for data.

The bad thing about linked lists is its difficulty in accessing an object at random; you can't just reach and grab the element.

4.1.3. Types of linked lists

Linearly linked lists

In linearly linked lists, the last node of a list contains a null reference to indicate it as the tail or end of the list. This type of list is also called an open linked list.

Circularly linked lists

In a circular linked list, the last node of a list contains a reference to the first node of the list. These lists are also called circular lists.

Singly Linked Lists

In singly linked lists, each node contains a link to the next node.

Doubly linked lists

In doubly linked lists, each node contains a link to the next node and yet another to link to the previous node. The two links may be called forwards and backwards or next and previous.

Multiply-linked Lists

In a multiply linked list, each node contains two or more link fields, each field being used to connect the same set of data records in a different order.

- _ A linked list is a structure consisting of nodes
- _ A node is a structure with two parts. One part stores the addresses of the next node.
- _ In a linked list they also have a special node referred to as the head. This node has only the information as to where the link structure starts.
- _ The last node in a linked list always point to NULL (Zero i.e. nothing)

Basic Operations

Following are the basic operations supported by a list.

- **Insertion** – Adds an element at the beginning of the list.
- **Deletion** – Deletes an element at the beginning of the list.
- **Display** – Displays the complete list.
- **Search** – Searches an element using the given key.
- **Delete** – Deletes an element using the given key.

4.1.4. How to insert a node at the beginning of a linked list

Steps

- _ Generate a new node
- _ Add information
- _ Establish position where to insert
- _ Let the link of the node at the position to insert points to the position of the new node
- _ Let the link of the new node points to the position where the node at the insertion points was pointing.

4.1.5. How to insert a node at the end of a linked list

Steps

- _ Generate a new node
- _ Add information
- _ Let the last node parts to position of the new node.
- _ Let the link of the new node points to NULL

4.1.6. Deleting first node in a linked list

Steps

- _ Let pointer points to the first node (position of the head)Let the points to the link of the first node
- _ Delete the node (free space)

4.1.7. Deleting a node that is in between the chain

Steps

- _ Note in position of the node to delete
- _ Let the link of the previous node points to the link of the previous node points to the link of the node to be deleted
- _ Free space

4.1.8. Deleting last node in the chain

Steps

- _ Let the link of the previous node points to null
- _ Delete the node (free space)

4.1.9. Example: Using the linked list concept, write a program for manipulating a stack

```
#include <iostream.h>
Class linkedstack
{
Private:
Structure linkedstacknode*link;
Linkedstacknode (int &item,Linked stackNode*head = NULL)
{
Data =item;
Link = head;
}
}
Linked stack Node* Top;
Public:
Linked stack ( )
{
TOP = NULL;
}
Void push (int item) ;
Void pop (int litem);
In empty ( )
{ :
Return TOP == NULL;
}
}
Void linkedstock : : push (int item)
{
Top = new linked stacknode (item,top);
}
Void linkedstock == pop ( int titem)
{
Linked stack node* ptr;
Ptr = TOP;
item = TOP-> Data;
Top = Top -> link;
Delete ptr;
}
Linkedstack s;
Int x, n;
Count<< "\nhow many Data",
Cin >.n;
For (I =I; ;<=n;i+ +)
{ Cin>>x;
S.push (x);
}
```

```

Count << "\n show Data",
Whilev (! S.empty (j)
{
s. pop (x);
count << "\n" << x;
}
Return0;
}

```

Example_.

Using the link list concept, write a program for manipulating a queue structure.

Solution:

```

Include < iostream .h>
Class linked Queue
{
Struct linkedQueuenode
{
Int Data;
linkedQueueNode*link;
LinkedQueue Node(int item, linkedQueue Node* mode modelink NULL)
{
Data = item;
Link = nodelinke;
};
LinkedQueuenode*rear,*front;
Public:
LinkedQueue
{
Front = NULL;
Rear = NULL;
}
Int emptyQueue ( )
{
Return front == Null;
}
Void add Q(int item)
{
If(front == NULL)
{
Rear = newlinkedQueuenode (item,rear);
From = rear;
} Else
{rear -link=new linkeQueueNode(item,rear);
Front = rear;
}
Else

```

```

{
Rear->link =new linkdQueueNode(item,rear->link)
Rear = rear -> link;
}
}
Void delete(intlitem)
{
Linke QueueNode *ptr;
Ptr =front;
Item=front ->data;
Front = front ->link;
Delete ptr;
}
};
Main c )
{ Inti,x,n;
LinkedQueue Q;
Count>>\n how many data?",
C"n >>n;
Count << "\n enter Data",
for( ; =1; ;<=n; ;+ +)
{
Cin >> x;
Q.add Q(x);
}
Count << " inshow Data".
While (1 Q.emptyqueue (j)
{
Q. DeleteQ(x)
Count << "\n" <<x;
}
Return O;
}

```

Review Questions

EXERCISE 13.

State the algorithm of Fibonacci sequence. Use your algorithm to write a program for computing Fibonacci sequence?

EXERCISE 14.

Trace the bubble sort algorithm as it sort the following array into ascending order: 20 80 40 25 60 30?