**KIBABII UNIVERSITY**

**BIT 121: DATA STRUCTURES AND ALGORITHMS**

**First year**

**1.0 Introduction**

Data structures and algorithms are concerned with the coding phase of software engineering.

**1.1 Overview of data structures and Algorithms**

* Data may be arranged in many different ways; the logical or mathematical model of a particular organization of data constitutes data structure.
* Data structure is a way to organize data; a way of storing and organizing data in order to facilitate access and modification.
* Basically, a data structure is an arrangement of data in a computer’s memory (or sometimes on disk space).
* Example: library
  + is composed of elements (books)
  + Accessing a particular book requires knowledge of the arrangement of the books
* Examples of data structures are arrays, linked lists, queues, stacks, binary trees among others.
* The choice of a data structure depend on two considerations:

i) It must be rich enough in structure to mirror the actual relationships of the data in the real world. ii) The structure should be simple enough that one can effectively process the data when necessary.

* Knowledge of data structures is required of people who design and develop computer programs of any kind: systems software or applications software.

* **Algorithm** is used to manipulate the data contained in the data structures
* It is generally a computable set of steps to achieve a desired result
* Examples of algorithms are searching for a particular data item and sorting the data
* One major challenge of programming is to develop efficient algorithms for the processing of our data.
* The time and space it uses are two major measures of the efficiency of an algorithm.
* The best algorithm is the one which has a fine balance between time taken and memory consumption.

**Why Java**

* Java enables users to develop and deploy applications on the Internet for servers, desktop computers and small hand-held devices. i.e Java is a general purpose programming language.

**Characteristics of Java**

1. Java is object-oriented

* it allows reuse of code
* OOP provides greater flexibility, modularity and reusability through encapsulation, inheritance and polymorphism.

1. Java is distributed
   * Distributed computing involves several computers working together on a network. Java is designed to make distributed computing easy.
2. Java is robust
   * It has a run time exception handling feature to provide programming support for robustness.
3. Java is secure
   * Java implements several security mechanisms to protect system against harm caused by stray programs
4. Java is Architecture-Neutral
   * With a Java Virtual Machine, you can write one program that will run on any platform. vi) Java is Dynamic
   * Java was designed to adapt to an evolving environment. New code can be loaded on the fly without recompilation.

**Simple Java Program**

//This program prints Welcome to Java!

public class Welcome {

public static void main (String [] args)

{

System.out.println (“Welcome to Java!”);

}

}

* + Message is printed to the console
  + Java is case sensitive
  + Program file name should exactly match the class name. When saving file, save using class name.

**Anatomy of a Java Program**

1. **Comments** 
   * + - Line comment which is preceded by 2 slashes (//) in a line
       - Paragraph comment which is enclosed between /\* and \*/ in one or multiple lines
2. **Reserved words**

* Are key words that have specific meaning to the compiler and cannot be used for other purposes in the program. E.g the word class which the compiler understands that the word after class is the name of the class.
* Other reserved words are public, static, void etc.

1. **Modifiers** 
   * + Specific properties of data, methods and classes and how they can be used. Example modifiers are public, static, private etc.
     + A public datum, method or class can be accessed by other programs
     + A private datum or method cannot be accessed by other programs. Only the method defined in the class can access and modify what is private.
2. **Statements** 
   * + Represents an action or sequence of actions
     + The statement System.out.println ("Welcome to Java!”) is a statement to display the greeting "Welcome to Java!”
     + Every statement in Java ends with a semicolon (;)
3. **Blocks**

* A pair of braces in a program forms a block that groups components of a program.

1. Classes
   * + A class is a template or a blue print for objects. Example, “Welcome” is a class name
     + For all class names, the first letter should be upper case.
2. Methods

e.g System.out.println

* + - It is a method: a collection of statements that perform a sequence of operations to display a message to the console.
    - It is used by invoking a statement with a string argument
    - The string argument is enclosed within parentheses. In this case, the argument is "Welcome to Java!”

1. **main Method** 
   * + Provides control of program flow
     + Java program executes application by invoking the main method

public static void main (String [] args)

{

//Statements;

}

* + - **public** – means method can be viewed outside class by all programs
    - **static** means any changes on the state of data inside the function will be effected
    - **void** means the method returns nothing to the calling program
    - **main** - is the name of the function/method
    - **String** **[]** **args**- The function receives args which is a string as an argument. Arguments are data passed to the function to be used.
    - **System.out.println** – Java stores content in a package. Content is contained in package called System. To output content, begin by calling the package.
    - **println** is a function.
    - **System.out.println –** prints and curser moves to next line
    - **System.out.print** – prints and curser remains there

**Difference between Java and C**

* + - Java is a pure object oriented programming language, it uses concepts of Classes, Objects, Inheritance
    - C is function oriented. It uses concept of pointers. There is no reuse of code
    - The main difference between Java and C are speed, portability and object-orientation. Java was created for the purpose of making a language that could be implemented on many different types of computers (cell phone, Mac, PC etc). C on the other hand can only be run on a computer of the same type as the one that compiled the program.

**Example C program**

# include <stdio.h>

int main()

{

//printf()

displays the string printf(“Hello, World!”);

return 0;

}

* + - The # include <stdio.h> is a preprocessor directive/command that tells the compiler to include the contents of stdio.h (standard input and output) file in the program.
    - The stdio.h file contains functions such as scanf(), printf() to take input and display output respectively.
    - If we write printf() function without writing # include <stdio.h>, the program will not be compiled.
    - Execution of C program starts from main () function
    - The printf() is a library function to send formatted output to the screen. In this program, printf() displays Hello, World! Text on the screen.
    - The return 0; statement is the “exit status” of the program. i.e program ends with this statement.

**BASIC DATA STRUCTURES**

**2.1 Arrays**

* + - An array is a number of data items of the same type arranged contiguously in memory.
    - An array is the most commonly used data storage structure and it is built into most programming languages e.g C, C++, Java.
    - Arrays offer a convenient way for introducing data structures and for seeing how object-oriented programming and data structures relate to each other.

**2.2.1 One dimensional array**

* + - The simplest data structure is a linear (or one-dimensional) array.
    - A linear array is a list of a finite number n of homogenous data elements (i.e., data elements of the same type) such that:

i) The elements of the array are referenced respectively by the index set consisting of n consecutive numbers. ii) The elements of the array are stored respectively in successive memory locations

* + - If we choose the name A for the array, then the elements of A are denoted by the bracket notation A[1], A[2],

A[3], …, A[N]

* + - The number K in A[K]is called a subscript and A[K] is called a subscripted variable.

Example:

A linear array STUDENT consisting of the names of five students is shown below.

STUDENT

|  |
| --- |
| Jane |
| John |
| Mary |
| Peter |
| James |

From top,

STUDENT[0] = Jane; STUDENT[1] = John; STUDENT[2] = Mary; STUDENT[3] = Peter; STUDENT[4] = James;

**Creating an array**

* There are two kinds of data in Java: primitive types (such as int, double etc), and objects. Objects have a state and behavior.
* In Java, arrays are treated as objects.
* Accordingly one must use the new operator to create an array.  The syntax is:

DataType[] ArrayName;

ArrayName=new DataType[size]; Example

int[]Arr; // defines a reference to an array

Arr = new int[100]; // creates the array, and sets Arr to refer to it

**Or the equivalent single-statement approach:**

int[]Arr = new int[100];

* The [] operator is the sign to the compiler that we are naming an array object and not an ordinary variable.
* Because an array is an object, its name, Arr in the above code is a reference to an array. It's not the array itself.
* The array is stored at an address elsewhere in memory, and Arr holds only this address.

**Accessing Array Elements**

* Array elements are accessed using square brackets
* temp = Arr[3]; // get contents of fourth element of array
* x= Arr [5];// gets content of sixth element of array
* y= Arr [9]; // gets content of tenth element of array

**NB:** The first element is numbered 0, so that the indices in an array of 10 elements run from 0 to 9.

* If you use an index that is less than 0 or greater than the size of the array less 1, you'll get the "Array Index Out of Bounds" runtime error. **Initialization**
* Unless you specify otherwise, an array of integers is automatically initialized to 0 when it's created.  You can initialize an array of a primitive type to something besides 0 using the syntax:

int[] Arr = { 20, 34, 65, 92, 12, 15, 18, 21, 24, 27 };

* This single statement takes the place of both the reference declaration and the use of new to create the array.
* The numbers within the curly braces are called the *initialization list*.
* The size of the array is determined by the number of values in this list.

An array Example1

Int[] Array=new int[3];

Array[0]=1;

Array[1]=2; Array[2]=4; for (a=0;a< Array.Length;a++)

System.out.print(Array[a]);

# An array Example2

**NB:** Name given to the program should be the name of the class.

// demonstrates Java arrays class ArrayApp

{

public static void main(String[] args) //main method

{

int[ ] Arr; // reference

Arr = new int[10]; // make array int j; // loop counter

Arr[0] = 44; // insert 10 elements

Arr[1] = 98;

Arr[2] = 65;

Arr[3] = 87;

Arr[4] = 12;

Arr[5] = 32;

Arr[6] = 70;

Arr[7] = 23;

Arr[8] = 54; Arr[9] = 67; for(j=0; j<10; j++) // display elements

System.out.print(Arr[j] + " ");

} // end main ()

} // end class ArrayApp

* In the above program, we create an array called Arr, place 10 data items in it and display all the items.
* The output of the program looks like this:

44 98 65 87 12 32 70 23 54 67

# Example3

//creat an array of integers, put some values in the array and print each value to standard output (1D) public class ArrayDemo {

public static void main (String[] args) { //declares an array of integers

int[] anArray; //allocates memory for 5 integers

anArray = new int[5]; //initialize first element

anArray[0] = 20; //initialize second element

anArray[1] = 25; //and so forth

anArray[2] = 70;

anArray[3] = 55;

anArray[4] = 34;

System.out.println(anArray[0]);

System.out.println(anArray[1]);

System.out.println(anArray[2]);

System.out.println(anArray[3]);

System.out.println(anArray[4]);

}

}

**Output**

20

25 70

55

34

**Same as**

//create an array of integers, put some values in the array and print each value to standard output public class ArrayDemo {

public static void main (String[] args) { //declares an array of integers

int[] anArray;

//allocates memory for 5 integers

anArray = new int[5]; //initialize first element

anArray[0] = 20; //initialize second element

anArray[1] = 25; //and so forth

anArray[2] = 70;

anArray[3] = 55;

anArray[4] = 34;

System.out.println("Element at index 0:" + anArray[0]);

System.out.println("Element at index 1:" + anArray[1]);

System.out.println("Element at index 2:" + anArray[2]);

System.out.println("Element at index 3:" + anArray[3]);

System.out.println("Element at index 4:" + anArray[4]);

}

}

**Output**

Element at index 0:20

Element at index 1:25

Element at index 2:70

Element at index 3:55

Element at index 4:34

# Example4

//Accessing string elements in 1Dimensional Array

public class OneDArrayDemo {

public static void main (String[] args) {

String[]Names = new String[3];

Names[0]="Ann";

Names[1]="Tom";

Names[2]="Janet";

System.out.println(Names[0]);

System.out.println(Names[1]);

System.out.println(Names[2]);

}

}

**Output**

Ann

Tom

Janet

**Same as**

public class OneDArrayDemo {

public static void main (String[] args) {

String[]Names = new String[3];

Names[0]="Ann";

Names[1]="Tom";

Names[2]="Janet";

System.out.print(Names[0]+"\t");

System.out.print(Names[1]+"\t");

System.out.print(Names[2]);

}

}

**Output**

Ann Tom Janet

# An array Example5 (Exercise)

Write a Java program that prompts the user to enter 5 scores. The program should output the scores, average and sum.

import java.util.Scanner; //includes Scanner class for user input. It is found in java.util package public class Marks// Marks is the name of the class

{ public static void main(String[] Args) //Main method

{

int[] Score=new int[5];//Array declaration

double avg, sum; //Variables

Scanner sc=new Scanner(System.in); //creates object of Scanner class (sc) for(int i=0;i<5;i++ ) //variable called i is created and initialized to 0

{

System.out.print("\nScore"+(i+1)+":");

Score[i]=sc.nextInt();

}

System.out.print("\n\n==RESULTS==\n\n");

sum=0;

for(int i=0;i<5;i++)

{

System.out.print(Score[i]+"\t");

sum+=Score[i];//similar to sum=sum+Score[i]

}

System.out.print(sum/5+"\t"+ sum);

}

}

If for example the following numbers are input

Scores1: 45

Scores2: 67

Scores3: 89

Scores4: 54

Scores5: 67

==RESULTS==

45 67 89 54 67 64.4 322.0

**2.2.2 Two dimensional array**

A two-dimensional array is a collection of similar data elements where each element is referenced by two subscripts.

# Creating an array

DataType[][] ArrayName;

ArrayName=new DataType[size][size];

# Or

DataType[][] ArrayName=new DataType[size][size];

Example:

int[][]Marks= new int[2][5];

2 represents the number of rows and 5 represents the number of columns.

**Initialization**

int[2][5] ArrayName= {{56, 45, 34, 32, 67}, {89, 54, 43, 78, 90}}; e.g

int[2][5] Marks= {{56, 45, 34, 32, 67}, {89, 54, 43, 78, 90}};

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 56 | 45 | 34 | 32 | 67 |
| 89 | 54 | 43 | 78 | 90 |

We can state that:

Marks[0][0]= 56

Marks[0][1]=45

Marks[0][2]=34

Marks[0][3]=32

Marks[0][4]=67

Marks[1][0]=89

Marks[1][1]=54

Marks[1][2]=43

Marks[1][3]=78

Marks[1][4]=90

**Example**

**//**Accessing elements in an array using corresponding number of index values public class TwoDArrayDemo {

public static void main (String[] args) {

String[][]Names = {{"Mr.", "Mrs.", "Ms."},{"Smith", "Jane"}};

//Mr. Smith

System.out.println(Names[0][0] + Names[1][0]);

//Ms. Jane

System.out.println(Names[0][2] + Names[1][1]);

}

}

**Output**

Mr.Smith

Ms.Jane

# Example (Exercise)

Write a java program that prompts the user to enter 5 scores of 3 students. The program should display the scores per students, their average and sum.

import ava.util.Scanner;

public class Marks2

{

public static void main(String[] Args)

{

int[][] Score=new int[3][5];

double avg,sum; int r,c;

Scanner sc=new Scanner(System.in);

for(r=0;r<3;r++)

{

System.out.print("\nStudent"+(r+1)+" Scores\n");

for(c=0;c<5;c++)

{

System.out.print("\nScore"+(c+1)+":");

Score[r][c]=sc.nextInt();

}

}

System.out.print("\n\n=====RESULTS=====\n\n");

for(r=0;r<3;r++)

{

sum=0;

System.out.print("\nStudent"+(r+1)+" Scores:");

for(c=0;c<5;c++)

{

System.out.print(Score[r][c]+"\t");

sum+=Score[r][c];

}

System.out.print(sum/5+"\t"+ sum);

}

}

}

The output is:

Student1 Scores

Score1:34

Score2:45

Score3:67

Score4:89

Score5:90

Student2 Scores

Score1:23

Score2:45

Score3:67

Score4:78

Score5:56

Student3 Scores

Score1:34

Score2:56

Score3:67

Score4:67

Score5:86

=====RESULTS=====

Student1 Scores:34 45 67 89 90 65.0 325.0

Student2 Scores:23 45 67 78 56 53.8 269.0

Student3 Scores:34 56 67 67 86 62.0 310.0

# Deleting elements from an array

public class ArrayOperations

{

public static void main(String[] Args)

{

int[] Marks=new int[4];

int i;

Marks[0]=42;

Marks[1]=67;

Marks[2]=89;

Marks[3]=69;

for(i=0;i<Marks.length;i++)

{

System.out.print(Marks[i]+"\t");

}

Marks[2]=0;

System.out.print("\n");

for(i=0;i<Marks.length;i++)

{

System.out.print(Marks[i]+"\t");

}

}

}

The output of the program is:

42 67 89 69

42 67 0 69

# Limitations of arrays

* Once an array is created, its size cannot be altered.
* Array provides inadequate support for inserting, deleting, sorting, and searching operations.

**3.0 Sorting**

Sorting refers to the operation of arranging data in ascending or descending order.

**Example**

Suppose an array DATA contains 8 elements as follows:

DATA: 77, 99, 44, 11, 88, 22, 66, 55

After sorting, DATA must appear in memory as follows:

DATA: 11, 22, 44, 55, 66, 77, 88, 99

There are several types of sorting algorithms:

* + - Insertion sort
    - Selection sort
    - Bubble sort

**Correctness**

An algorithm is correct with respect to a problem if for every instance of the inputs (that is, the specified properties of the inputs are satisfied) the algorithm halts, and the outputs produced satisfy the specified input/output relation.

**3.1 Insertion sort**

This is an in-place comparison-based sorting algorithm. An element which is to be 'inserted in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of Ο(n2), where **n** is the number of items.

**How Insertion Sort Works?** We take an unsorted array for our example.

Unsorted Array

Insertion sort compares the first two elements.

Insertion Sort

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion Sort

Insertion sort moves ahead and compares 33 with 27.

Insertion Sort

And finds that 33 is not in the correct position.

Insertion Sort

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

Insertion Sort

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

Insertion Sort

These values are not in a sorted order.

Insertion Sort

So we swap them.

Insertion Sort

However, swapping makes 27 and 10 unsorted.

Insertion Sort

Hence, we swap them too.

Insertion Sort

Again we find 14 and 10 in an unsorted order.

Insertion Sort

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

Insertion Sort

This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

**Algorithm**

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

**Step 1** − If it is the first element, it is already sorted. Return 1;

**Step 2** − Pick next element

**Step 3** − Compare with all elements in the sorted sub-list

**Step 4** − Shift all the elements in the sorted sub-list that is greater than the value to be sorted

**Step 5** − Insert the value

**Step 6** − Repeat until list is sorted

**Example2**

Suppose an array A contains 8 elements as follows:

77, 33, 44, 11, 88, 22, 66, 55

Pass 1: 77, 33, 44, 11, 88, 22, 66, 55

Pass 2: 33, 77, 44, 11, 88, 22, 66, 55

Pass 3: 33, 44, 77, 11, 88, 22, 66, 55

Pass 4: 11, 33, 44, 77, 88, 22, 66, 55

Pass 5: 11, 33, 44, 77, 88, 22, 66, 55

Pass 6: 11, 22, 33, 44, 77, 88, 66, 55

Pass 7: 11, 22, 33, 44, 66, 77, 88, 55

Pass 8: 11, 22, 33, 44, 55, 66, 77, 88

**Example 3**

Write a java program to carry out an insertion sort on the following elements.

74, 47, 85,69,45,60,10,56,43

public class InsertionSort

{

public static void main(String[ ] Args)

{

int[] Marks={74,47,85,69,45,60,10,56,43};

int i, j, temp, x; /\* i and j are control variables for the loops i.e determine indices of array. x is for displaying passes \*/

System.out.print("\nPass 1:");

for(i=0;i<9;i++)

{

System.out.print(Marks[i] + "\t"); // prints elements of the array as they are i.e non sorted

}

//Insertion Sort Begins

for(i=0;i<8;i++)

{

if(Marks[i]>Marks[i+1])

{

temp=Marks[i+1]; //47 will be allocated to temp

Marks[i+1]=Marks[i]; // 47 will be replaced by 74 Marks[i]=temp; //74 will be replaced by 47

for(j=i;j>0;j--)

{

if(Marks[j]<Marks[j-1])

{

temp=Marks[j-1];

Marks[j-1]=Marks[j];

Marks[j]=temp;

}

}

}

System.out.print("\nPass"+(i+2)+ ":");

for(x=0;x<9;x++)

{

System.out.print(Marks[x] + "\t");//Display elements in each pass

}

}

//End of Insertion Sort

}

}

**Output**

Pass 1: 74 47 85 69 45 60 10 56 43

Pass 2: 47 74 85 69 45 60 10 56 43

Pass 3: 47 74 85 69 45 60 10 56 43

Pass 4: 47 69 74 85 45 60 10 56 43

Pass 5: 45 47 69 74 85 60 10 56 43

Pass 6: 45 47 60 69 74 85 10 56 43

Pass 7: 10 45 47 60 69 74 85 56 43

Pass 8: 10 45 47 56 60 69 74 85 43

Pass 9: 10 43 45 47 56 60 69 74 85

**Insertion Sort runtimes**

**Best case:** insertion sort is efficient when the data is in sorted order. After making one pass through the data and making no insertions, insertion sort exits.

**Worst case:** insertion sort is inefficientif the numbers were sorted in reverse order.

**3.2 selection Sort**

The idea behind selection sort is that we put a list in order by placing each item in turn. In other words, we put the smallest item at the start of the list, then the next smallest item at the second position in the list, and so on until the list is in order.

Selection sort is inefficient for sorting large data volumes.

Selection sort is notable for its programming simplicity (applying swaps) and it can over perform other sorts in certain situations

***Example1.***

Sort {5, 1, 12, -5, 16, 2, 12, 14} using selection sort.



**Example 2**

Write a java program to carry out a selection sort on the following elements.

74, 47, 85, 69, 45, 60, 10, 56, 43

public class SelectionSort

{

public static void main(String[ ] Args)

{

int[] Marks={74,47,85,69,45,60,10,56,43};

int i,j,temp,x,ind; // ind keeps position of the smallest element in each pass. Others remain the same

for(i=0;i<8;i++)

{

ind=i;

temp=Marks[i];

for(j=i;j<9;j++)

{

if(Marks[j]<temp)

{

temp=Marks[j];

ind=j;

}

}

Marks[ind]=Marks[i];

Marks[i]=temp;

System.out.print("\nPass" + (i+1)+":");

for(x=0;x<9;x++)

{

System.out.print(Marks[x] + "\t"); // prints elements in each pass

}

}

}

}

**Output**

Pass 1: 10 47 85 69 45 60 74 56 43

Pass 2: 10 43 85 69 45 60 74 56 47

Pass 3: 10 43 45 69 85 60 74 56 47

Pass 4: 10 43 45 47 85 60 74 56 69

Pass 5: 10 43 45 47 56 60 74 85 69

Pass 6: 10 43 45 47 56 60 74 85 69

Pass 7: 10 43 45 47 56 60 69 85 74

Pass 8: 10 43 45 47 56 60 69 74 85

**Complexity analysis**

Selection sort stops, when unsorted part becomes empty. On every step number of unsorted elements decreased by one.

Number of swaps may vary from zero (in case of sorted array) to n - 1 (in case array was sorted in reversed order).

Fact, that selection sort requires n - 1 number of swaps at most, makes it very efficient in situations, when write operation is significantly more expensive, than read operation.

**3.3. Bubble sort**

The idea behind bubble sort is similar to the idea behind selection sort: on each pass through the algorithm, we place at least one item in its proper location.

The differences between bubble sort and selection sort lie in how many times data is swapped and when the algorithm terminates. Bubble sort performs more swaps in each pass

Like selection sort, bubble sort works by comparing two items in the list at a time.

Unlike selection sort, bubble sort will always compare two consecutive items in the list, and swap them if they are out of order.

If we assume that we start at the beginning of the list, this means that at each pass through the algorithm, the largest remaining item in the list will be placed at its proper location in the list.

**Algorithm**

1. Compare each pair of adjacent elements from the beginning of an array and, if they are in reversed order, swap them.
2. If at least one swap has been done, repeat step 1.

**Example 1**

Suppose the following numbers are stored in an array A:

32, 51, 27, 85, 66, 23, 13, 57

Pass1: we have the following comparisons:

1. Compare A1 and A2. Since 32<51, the list is not altered

32, 51, 27, 85, 66, 23, 13, 57

1. Compare A2 and A3. Since 51>27, interchange 51 and 27 as follows

32, 27, 51, 85, 66, 23, 13, 57

1. Compare A3 and A4. Since 51<85, the list is not altered

32, 27, 51, 85, 66, 23, 13, 57

1. Compare A4 and A5. Since 85>66, interchange 85 and 66 as follows

32, 27, 51, 66, 85, 23, 13, 57

1. Compare A5 and A6. Since 85>23, interchange 85 and 23 as follows

32, 27, 51, 66, 23, 85, 13, 57

1. Compare A6 and A7. Since 85>13, interchange 85 and 13 as follows

32, 27, 51, 66, 23, 13, 85, 57

1. Compare A7 and A8. Since 85>57, interchange 85 and 57 as follows

32, 27, 51, 66, 23, 13, 57, 85

At the end of the first pass, the largest number 85 has moved to the last position. However, the rest of the numbers are not sorted, even though some of them have changed their positions.

Pass2: only showing the interchanges

1. 27, 32, 51, 66, 23, 13, 57, 85
2. 27, 32, 51, 23, 66, 13, 57, 85
3. 27, 32, 51, 23, 13, 66, 57, 85
4. 27, 32, 51, 23, 13, 57, 66, 85

At the end of Pass 2, the second largest number, 66, has moved its way down to the next-to-last position.

Pass3: only showing the interchanges

1. 27, 32, 23, 51, 13, 57, 66, 85
2. 27, 32, 23, 13, 51, 57, 66, 85

Pass4: only showing the interchanges

1. 27, 23, 32, 13, 51, 57, 66, 85
2. 27, 23, 13, 32, 51, 57, 66, 85

Pass5: only showing the interchanges

1. 23, 27, 13, 32, 51, 57, 66, 85
2. 23, 13, 27, 32, 51, 57, 66, 85

Pass 6: 13, 23, 27, 32, 51, 57, 66, 85

**Example 2**

Write a java program to carry out a bubble sort on the following elements. 74,47,85,69,45,60,10,56,43

public class BubbleSort

{

public static void main(String[ ] Args)

{

int[] Marks={74,47,85,69,45,60,10,56,43};

int i,j, temp, x, y;// ind keeps position of the smallest element in each pass. others remain the same

for(i=0;i<9;i++)

{

temp=Marks[0];

for(j=0;j<8;j++)

{

if(Marks[j+1]<temp)

{

Marks[j]=Marks[j+1];

Marks[j+1]=temp;

}

temp=Marks[j+1];

}

System.out.print("\nPass" + (i+1)+ ":");

for(x=0;x<9;x++)

{

System.out.print(Marks[x] + "\t");// prints elements in each pass

}

//Checks whether the elements are sorted if so the loop is terminated

y=0;

for (x=0;x<8;x++)

{

if(Marks[x]>Marks[x+1])

{

y=1; //y is set to 1 if the elements are not sorted

break; // looping is terminated

}

}

if(y==0)

{

break;

}

}

}

}

**Output**

Pass 1: 47 74 69 45 60 10 56 43 85

Pass 2: 47 69 45 60 10 56 43 74 85

Pass 3: 47 45 60 10 56 43 69 74 85

Pass 4: 45 47 10 56 43 60 69 74 85

Pass 5: 45 10 47 43 56 60 69 74 85

Pass 6: 10 45 43 47 56 60 69 74 85

Pass 7: 10 43 45 47 56 60 69 74 85

**4.0. Non-linear structures**

A non-linear data structure is a data structure in which a data item is connected to several other data items. Example are: graphs, trees, heaps

**Trees**

A tree is a collection of nodes called elements. A tree stores elements hierarchically. With the exception of the top element, each element in a tree has a parent element and zero or more children elements. The top element is called the root of the tree.

**Binary trees**

A binary tree is a hierarchical structure. It has a root node and 0-2 children. If it has 2 children, they are the left child and the right child. Each child is itself a tree, it may have a left child and a right child. Examples of binary trees are shown

60 G

55

100

F

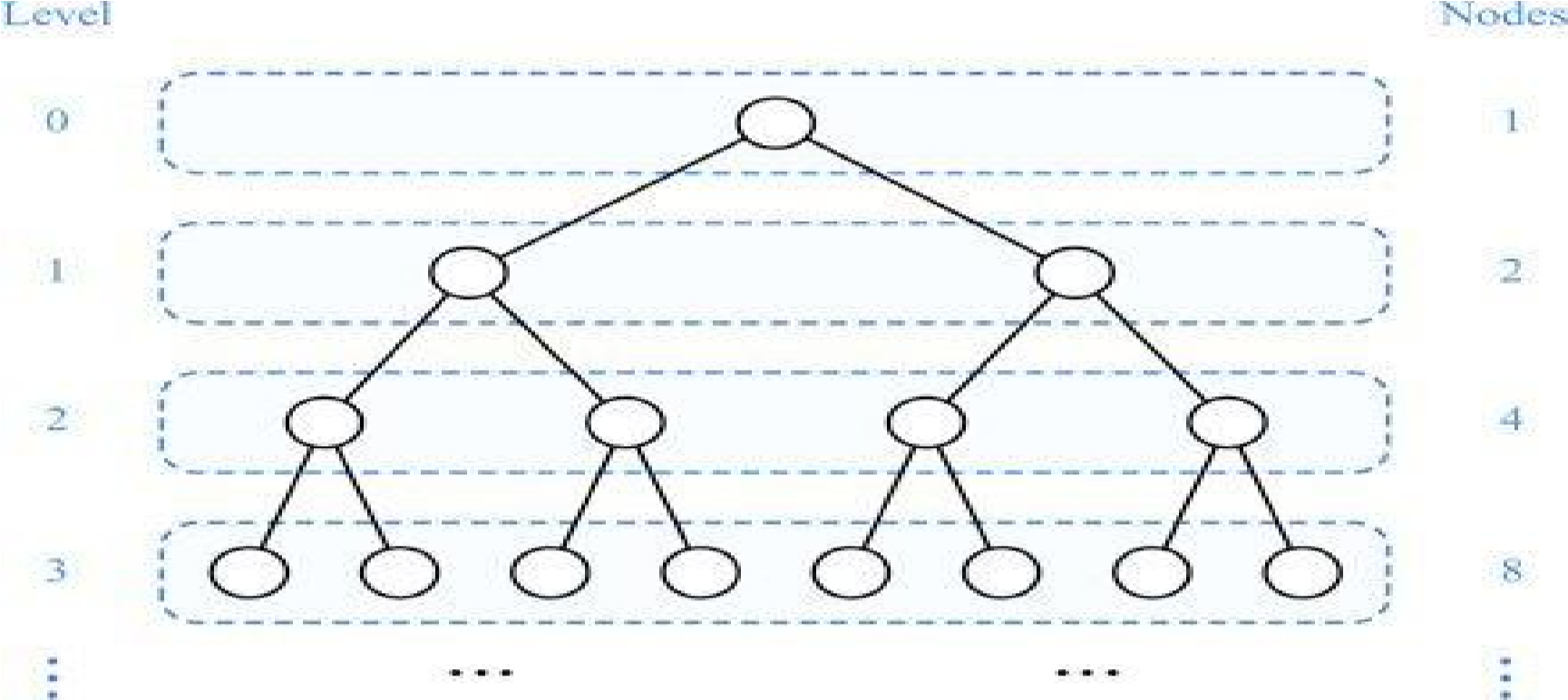
R

45 57 67 107 A M T

(A) (B)

A node without children is called a *leaf*.

* According to the definition of trees, a node can have any number of children.
* A binary tree is restricted to only having 0, 1, or at most 2 children.
* A complete binary tree is one where all the levels are full with exception to the last level and it is filled from left to right.
* A full binary tree is one where if a node has a child, then it has two children.



Differences between a general tree and binary tree

* A general tree is a **data structure** in that each node can have infinite number of children while a Binary tree is a data structure in that each node has at most **two nodes** left and right.
* Subtree of general tree are **not ordered** while Subtree of binary tree are **ordered**.

**Binary search trees**

A special type of binary tree called a *binary search tree* is often useful. A binary search tree (with no duplicate elements) has the property that for every node in the tree the value of any node in its left subtree is less than the value of the node and the value of any node in its right subtree is greater than the value of the node.

**Example:**

Draw the binary search tree for the following inputs. 60, 25, 36, 45, 54, 20, 64.

**Representing Binary Trees**

A binary tree can be represented using a set of linked nodes. Each node contains a value and two links named *left* and *right* that reference the left child and right child, respectively, as shown in Figure

below

.

60

55

100

57

45

67

107

root

class TreeNode { Object element;

TreeNode left;

TreeNode right;

public TreeNode(Object o) {

element = o;

}

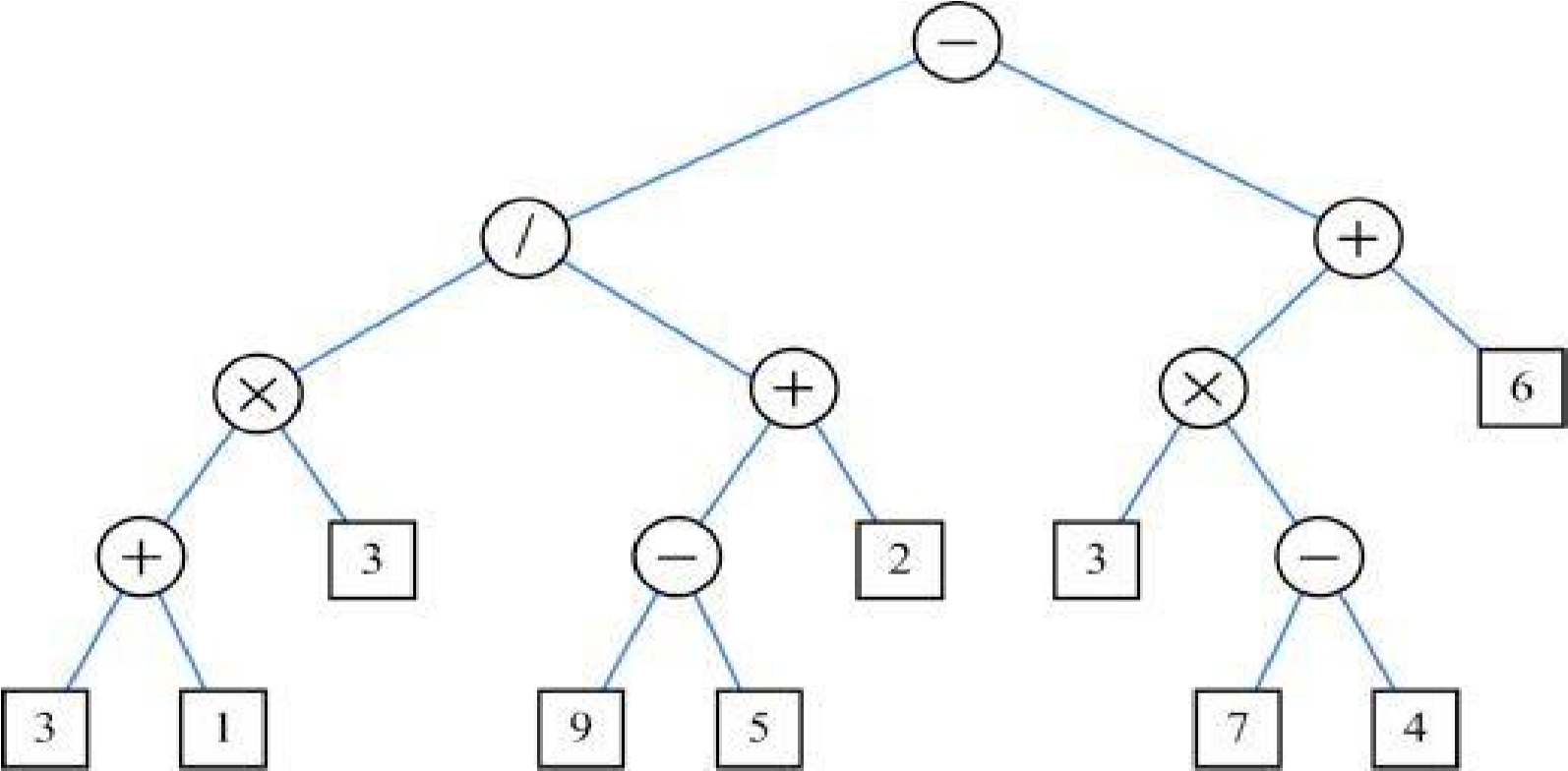
}

**An arithmetic** expression can be represented by a binary tree whose external nodes are associated with variables or constants, and whose internal nodes are associated with one of the operators, +, -, \* and /. An arithmetic expression tree is a proper binary tree, since each operator, +, -, \* and /, takes exactly two operands.

**Example**

A binary tree representing an arithmetic expression. The tree represents the expression

((((3 + 1) \*3) / ((9 – 5) + 2)) – ((3 \* (7 – 4)) + 6))



**Inserting an Element to a Binary Tree**

If a binary tree is empty, create a root node with the new element. Otherwise, locate the parent node for the new element node. If the new element is less than the parent element, the node for the new element becomes the left child of the parent. If the new element is greater than the parent element, the node for the new element becomes the right child of the parent. Here is the algorithm:

**Insert 101 into the following tree.**

60

55

100

57

45

67

107

root

if (root == null)

root = new TreeNode(element);

else {

// Locate the parent node

current = root;

while (current != null)

if (element value < the value in current.element)

{ parent = current;

current = current.left;

}

else if (element value > the value in current.element)

{

parent = current;

current = current.right;

}

else

return false; // Duplicate node not inserted

// Create the new node and attach it to the parent node

if (element < parent.element)

parent.left = new TreeNode(element);

else

parent.right = new TreeNode(element);

return true; // Element inserted

}

The tree will be:

60

55

100

57

45

67

107

root

parent

101

**Inserting 59 into the Tree**

60

55

100

57

45

67

107

root

59

101

**Tree Traversal**

Tree traversal is the process of visiting each node in the tree exactly once. There are several ways to traverse a tree: ***inorder***, ***preorder***, ***postorder****,* ***depth****-****first***traversals.

* The inorder traversal: visit the left subtree of the current node first, then the current node itself, and finally the right subtree of the current node (LNR).
* The preorder traversal: visit the current node first, then the left subtree of the current node, and finally the right subtree of the current node (NLR).
* The postorder traversal : visit the left subtree of the current node first, then the right subtree of the current node, and finally the current node itself (LRN).
* The breadth-first traversal is to visit the nodes level by level. First visit the root, then all children of the root from left to right, then grandchildren of the root from left to right, and so on.

60

55

100

57

45

67

107

root

59

101

For example, in the tree in Figure above,

**The inorder is:** 45 55 57 59 60 67 100 101 107.

**The postorder is:** 45 59 57 55 67 101 107 100 60.

**The preorder is:** 60 55 45 57 59 100 67 107 101.

**The breadth-first traversal is:** 60 55 100 45 57 67 107 59 101.

**Traverse the following binary tree in inorder, postorder, and preorder**

George

Adam

Michael

Daniel

Jones

Tom

root

Peter

**Inorder:** Adam, Daniel George, Jones, Michael, Peter, Tom

**Postorder:** Daniel Adam, Jones, Peter, Tom, Michael, George

**Preorder:** George, Adam, Daniel, Michael, Jones, Tom, Peter

**Example 1**

Represent the following expression tree **(a+b\*c)\*(d/e-f)** and write the prefix and postfix expression.



\*



b



c



d



e



/



a



+



\*



-



f

i) Prefix expression: \*+a\*bc-/def

ii) Postfix expression: abc\*+de/f-\*

**Example 2**

Draw a binary tree to evaluate the following fully parenthesized expression and evaluate it.

(((8\*3) + 2)/(21-(2^3)))

**BINARY TREE FOR THE ABOVE EXPRESSION**



\*



+



/



-



^



2



8



3



2



3



2

**EVALUATION OF THE FULLY PARENTHESIZED EXPRESSION**

|  |  |
| --- | --- |
| ( | ( |
| ( | (( |
| ( | ((( |
| 8 | (((8 |
| \* | (((8\* |
| 3 | (((8\*3 |
| ) | (((8\*3) |
| + | ((24+ |
| 2 | ((24+2 |
| ) | ((24+2) |
| / | (26/ |
| ( | (26/( |
| 21 | (26/(21 |
| - | (26/(21- |
| ( | (26/(21-( |
| 2 | (26/(21-(2 |
| ^ | (26/(21-(2^ |
| 3 | (26/(21-(2^3 |
| ) | (26/(21-(2^3) |
| ) | (26/(21-8) |
| ) | (26/13) |
| End Result | 2 |

Hence, (((8\*3)+2)/(21-(2^3))) = 2

**Heap**

A *heap* is a binary tree with the following properties:

* It is a complete binary tree.
* Each node is greater than or equal to any of its children.

**Complete Binary Tree**

A binary tree is *complete* if every level of the tree is full except that the last level may not be full and all the leaves on the last level are placed left-most. For example, in Figure below, the binary trees in (a) and (b) are complete, but the binary trees in (c) and (d) are not complete. Further, the binary tree in (a) is a heap, but the binary tree in (b) is not a heap, because the root (39) is less than its right child (42).

42 39 42 42

32

32

39

32

42

32

39 32

22 29 14 33 22 29 14 22 14 33 22 29

**Representing a Heap**

For a node at position *i*, its left child is at position *2i+1* and its right child is at position *2i+2*, and its parent is (*i-1)/2*. For example, the node for element 39 is at position 4, so its left child (element 14) is at 9 (*2\*4+1*), its right child (element 33) is at 10 (*2\*4+2*), and its parent (element 42) is at 1 (*(4-1)/2*).

[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10][11][12][13]

42 59 32 39 44 13 22 29 14 33

62

30

17

9

[10][11]

parent

lef

t

right

22

29 14 33

30

17

9

32

39 44

13

42

59

62

3  *1*

3

5

1

*22*

11

19

3

|  |  |  |
| --- | --- | --- |
| (a) After adding 3 | (b) After adding 5 | (c) After adding 1 |

|  |  |
| --- | --- |
| **Adding Elements to the Heap**  Adding 3, 5, 1, 19, 11, and 22 to a heap, initially empty |  |
| *3 5* | 5 |

19

*11*

1

3 5

3

*19*

5

1

(d) After adding 19 (e) after adding 11 (f) After adding 22

**Rebuild the heap after adding a new node**

Adding 88 to the heap

22 22 *88*

11

19

11

*88*

11

22

|  |  |  |
| --- | --- | --- |
| 3 5 1 *88*  (a) Add 88 to a heap | 3 5 1 19  (b) After swapping 88 with 19 | 3 5 1 19  (b) After swapping 88 with 22 |

**Removing the Root and Rebuild the Tree**

Removing root 62 from the heap

29 14 33

22

30

17

9

13

39 44

32

42

59

*62*

Move 9 to root

29 14 33

22

30

17

13

39 44

32

59

42

*9*

Swap 9 with 59

29 14 33

22

30

17

32

13

39 44

4

2

9

59

Swap 9 with 44

29 14 33

22

30

17

32

39

9

13

4

2

44

59

Swap 9 with 30

59

42 44

32 39 30 13

22 29 14 33 *9* 17

**5.0. Searching**

Searching for items is a task that we do every day. Searching is also a common task in computer programs. We search for all occurrences of a word in a file in order to replace it with another word.

**5.1. Linear search**

The simplest search algorithm is linear search. In linear search, we look at each item in the list in turn, quitting once we find an item that matches the search term or once we’ve reached the end of the list. Our “return value” is the index at which the search term was found, or some indicator that the search term was not found in the list.

**Algorithm for linear search**

for (each item in list) { compare search term to current item if match, save index of matching item break

} return index of matching item, or -1 if item not found

**Java implementation of linear search**

public int sequentialSearch(int item, int[] list) {

// if index is still -1 at the end of this method, the item is not

// in this array. int

index = -1;

// loop through each element in the array. if we find our search

// term, exit the loop.

for (int i=0; i<list.length; i++)

{

if (list[i] == item)

{

index = i; break;

}

} return index;

}

**Example**

import java.util.Scanner; public class LinearSearch

{

public static void main(String[] Args)

{

int[] Marks={56,47,84,68,62,38};

String optn="Y";

int key;

int result=-1; int i;

int max=Marks.length;

Scanner sc=new Scanner(System.in); System.out.print("The Original Values: "); for(i=0;i<max;i++)

{

System.out.print(Marks[i]+"\t");

}

while(optn.equals("Y") || optn.equals("y"))

{

System.out.print("\nEnter Value to Search:"); key=sc.nextInt();

result=-1;

System.out.print("\nSearched Values:"); for(i=0;i<max;i++)

{

System.out.print(Marks[i] + "\t");

if (Marks[i]==key)

{

result=i; break;

}

}

System.out.print("\nSearch Result: "+ result);

System.out.print("\nEnter Y to continue or Any Key to terminate:");

optn=sc.next();

}

}

}

In case the value to search is 68,

Searched Values: 56 47 84 68

Search Result: 3

**Performance of linear search**

When comparing search algorithms, we only look at the number of comparisons, since we don’t swap any values while searching. Often, when comparing performance, we look at three cases:

* Best case: What is the fewest number of comparisons necessary to find an item?

* Worst case: What is the most number of comparisons necessary to find an item?

* Average case: On average, how many comparisons does it take to find an item in the list?

For linear search, our cases look like this:

* Best case: The best case occurs when the search term is in the first slot in the array. The number of comparisons in this case is 1.

* Worst case: The worst case occurs when the search term is in the last slot in the array, or is not in the array. The number of comparisons in this case is equal to the size of the array. If our array has N items, then it takes N comparisons in the worst case.

* Average case: On average, the search term will be somewhere in the middle of the array. The number of comparisons in this case is approximately N/2.

**5.2. Binary search**

Binary search exploits the ordering of a list. The idea behind binary search is that each time we make a comparison, we eliminate half of the list, until we either find the search term or determine that the term is not in the list. We do this by looking at the middle item in the list, and determining if our search term is higher or lower than the middle item. If it’s lower, we eliminate the upper half of the list and repeat our search starting at the point halfway between the first item and the middle item. If it’s higher, we eliminate the lower half of the list and repeat our search starting at the point halfway between the middle item and the last item. **Algorithm for binary search** set first = 1, last = N, mid = N/2 while (item not found and first < last) { compare search term to item at mid if match save index break else if search term is less than item at mid, set last = mid-1 else set first = mid+1 set mid = (first+last)/2

} return index of matching item, or -1 if not found **Java implementation of binary search**

public int binarySearch(int item, int[] list) {

// if index = -1 when the method is finished, we did not find the

// search term in the array int index = -1;

// set the starting and ending indexes of the array; these will

// change as we narrow our search int low = 0; int high = list.length-1; int mid;

// Continue to search for the search term until we find it or

// until our ‘‘low’’ and ‘‘high’’ markers cross while (high >= low) {

mid = (high + low)/2;// calculate the midpoint of the current array if (item < list[mid]) { // value is in lower half, if at all

high = mid - 1;

} else if (item > list[mid]) {

// value is in upper half, if at all low = mid + 1;

} else {

// found it! break out of the loop index = mid; break;

} } return index;

}

**Performance of binary search**

The best case for binary search still occurs when we find the search term on the first try. In this case, the search term would be in the middle of the list.

The worst case for binary search occurs when the search term is not in the list, or when the search term is one item away from the middle of the list, or when the search term is the first or last item in the list.

The following are examples to determine the number of comparisons.

Suppose we have a list of four integers: {1, 4, 5, 6}. We want to find 2 in the list.

According to the algorithm, we start at the second item in the list, which is 4. Our search term, 2, is less than 4, so we throw out the last three items in the list and concentrate our search on the first item on the list, 1. We compare 2 to 1, and find that 2 is greater than 1. At this point, there are no more items left to search, so we determine that 2 is not in the list. It took two comparisons to determine that 2 is not in the list.

Now suppose we have a list of 8 integers: {1, 4, 5, 6, 9, 12, 14, 16}. We want to find 9 in the list. Again, we find the item at the midpoint of the list, which is 6. We compare 6 to 9, find that

9 is greater than 6, and thus concentrate our search on the upper half of the list: {9, 12, 14, 16}. We find the new midpoint item, 12, and compare 12 to 9. 9 is less than 12, so we concentrate our search on the lower half of this list (9). Finally, we compare 9 to 9, find that they are equal, and thus have found our search term at index 4 in the list. It took three comparisons to find the search term. **Example**

public class BinarySearch

{

public static void main(String[] Args)

{

int[] Marks={10,25,34,38,56,78,85,96};

int i;

int low = 0; int item=10;

int index = -1;

int high = Marks.length-1;

int mid;

// Continue to search for

// until our ‘‘low’’ and

System.out.print ("The Original Values Are:");

for(i=0;i<Marks.length;i++)

{

System.out.print(Marks[i]+"\t");

}

System.out.print("\nThe Searched Values Are:");

while (high >= low)

{

mid = (high + low)/2;

System.out.print(Marks[mid]+"\t");

if (item < Marks[mid])

{

high = mid - 1;//search the lower half

}

else if (item > Marks[mid])

{

// value is in up

low = mid + 1;

}

else// means value is the same

{

// found it! break

index = mid;

break;

}

}

System.out.print("\nThe result is: "+index);

}

}

Given that we are searching the value 10, the result would be

The Original Values Are: 10 25 34 38 56 78 85 96

The Searched Values Are: 38 25 10

The result is: 0

**Abstract data types**

Abstract Data Type (ADT) - An abstract data type is a data declaration packaged together with the operations that are meaningful on the data type. i.e an ADT is a type with associated operations, but whose representation is hidden.

 **List**

A list is a data structure to store data in sequential order. For example, a list of students, a list of available rooms, a list of cities, and a list of books, etc. can be stored using lists. The common operations on a list are usually the following:

· Retrieve an element from the list.

· Insert a new element to the list.

· Delete an element from the list.

· Find how many elements are in the list.

· Find if an element is in the list.

· Find if the list is empty.

**Two Ways to Implement Lists**

There are two ways to implement a list.

* One is to use an array to store the elements. The array is dynamically created. If the capacity of the array is exceeded, create a new larger array and copy all the elements from the current array to the new array.
* The other approach is to use a linked list. A linked list consists of nodes. Each node is dynamically created to hold an element. All the nodes are linked together to form a list.

**Disadvantages of arrays are**

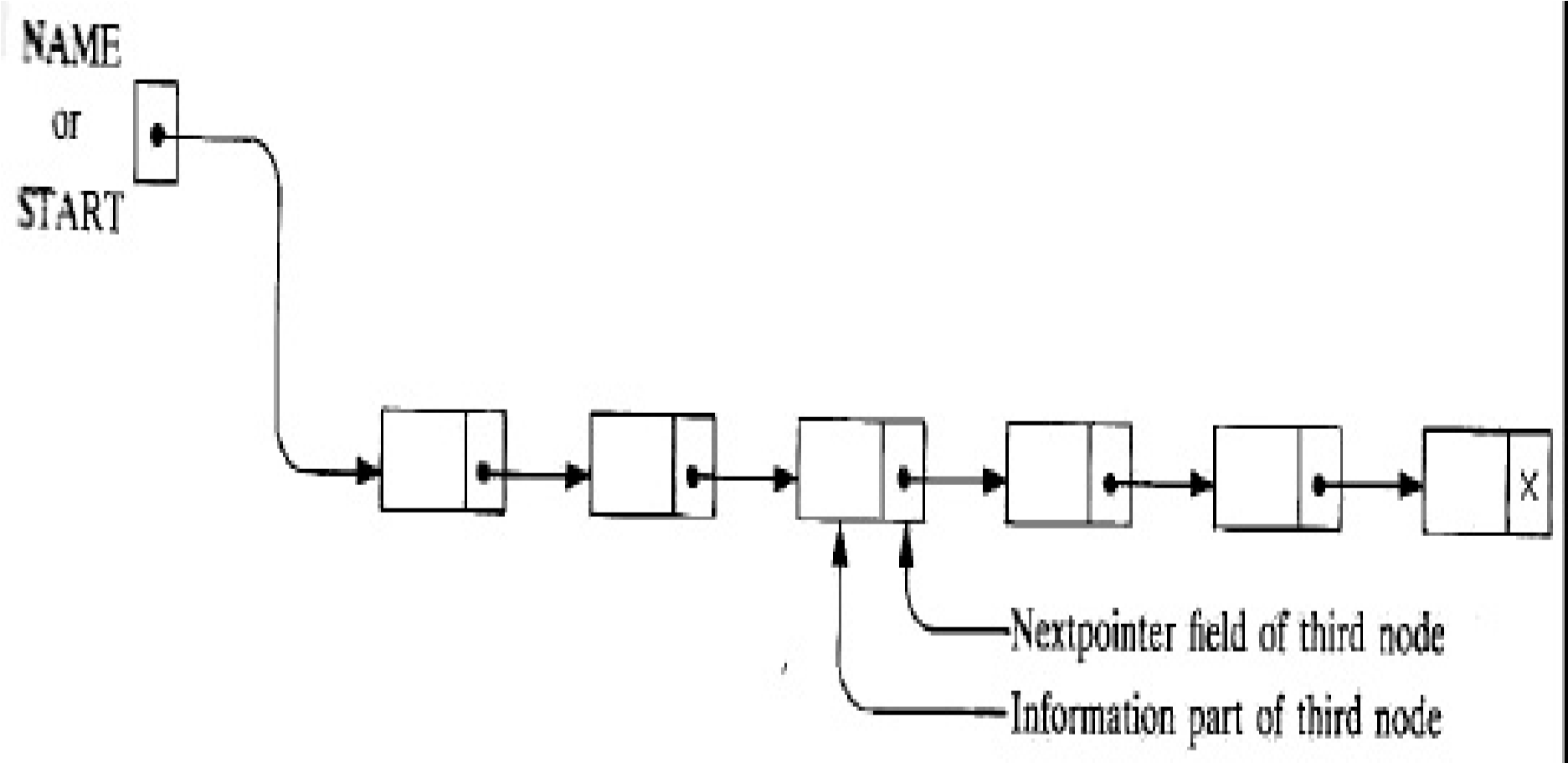
1. The size of the array is fixed for example, 100 elements.
2. Because of (1), the most convenient thing for programmers to do is to allocate arrays which seem "large enough" (e.g. the 100 in the scores example). Although convenient, this strategy has two disadvantages: (a) most of the time there are just 20 or 30 elements in the array and 70% of the space in the array really is wasted. (b) If the program ever needs to process more than 100 scores, the code breaks.

**Linked list**

Linked list is a linear collection of data elements, called nodes, where the linear order is given by means of pointers.

That is, each node is divided into two parts: the first part contains the information of the element, and the second part, called the link field or next pointer field, contains the address of the next node in the list. A list that has no nodes is called the null list or empty list and is denoted by the null pointer in the variable START.

Figure below is a diagram of a linked list with 6 nodes.



Each node is pictured with 2 parts. The left part represents the information part of the node which may contain an entire record of data items (e.g., Name, Address,..). The right part represents the pointer field of the node and there is an arrow drawn from it to the next node in the list. The pointer of the last node contains a special value, called the null pointer, denoted by x, which is an invalid address. In actual practice, 0 or a negative value is used for the null pointer.

A linked list also contains a list pointer variable, called START or NAME, which contains the address of the first node in the list, hence there is an arrow drawn from START to the first node. This address is needed to trace through the list.

**Similarity between linked list and array**

Linked lists and arrays are similar since they both store collections of data.

**Memory Allocation**

The maintenance of linked lists in memory assumes the possibility of inserting new nodes into the lists and hence requires some mechanism that provides unused memory space for the new nodes.

Also, some mechanism is required whereby the memory space of deleted nodes becomes available for future use. Together with the linked lists in memory, a special list is maintained which consists of unused memory cells.

This list, which has its own pointer, is called the list of available space or the free-storage list or the free pool.

Suppose our linked lists are implemented by parallel arrays and suppose insertions and deletions are to be performed on our linked lists.

Then the unused memory cells in the arrays will also be linked together to form a linked list using AVAIL as its list pointer variable. (Hence this free-storage list will also be called the AVAIL

list.) Such a data structure will frequently be denoted by writing

LIST (INFO, LINK, START, AVAIL)

**Representation of Linked Lists in Memory**

Let LIST be a linked list. Then LIST will be maintained in memory, unless otherwise specified or implied, as follows.

First of all, LIST requires two linear arrays-let’s call them here INFO and LINK-such that INFO[K] and LINK[K] contain, respectively, the information part and the next pointer field of a Node of LIST.

LIST also requires a variable name-such as START which contains the location of the beginning of the list, and a next pointer sentinel-denoted by NULL-which indicates the end of the list.

Since the subscripts of the arrays INFO and LINK will usually be positive, we will choose NULL = 0, unless otherwise stated.

We require two tests. First we have to check to see whether we have reached the end of the list; i.e., first we check to see whether PTR= NULL

If not, then we check to see whether INFO [PTR] = ITEM

**Garbage Collection**

Suppose some memory space becomes reusable because a node is deleted from a list or an entire list is deleted from a program.

The operating system of a computer may periodically collect all the deleted space onto the freestorage list.

Any technique, which does this collection, is called **garbage collection.** Garbage collection usually takes place in two steps.

 First the computer runs through all lists, tagging those cells which are currently in use, and then the computer runs through the memory, collecting all untagged space onto the free-storage list.

The garbage collection may take place when there is only some minimum amount of space or no space at all left in the free-storage list, or when the CPU is idle and has time to do the collection. Generally speaking, the garbage collection is invisible to the programmer.

**Overflow and Underflow**

Sometimes new data are to be inserted into a data structure but there is no available space, i.e., the free-storage list is empty. This situation is usually called overflow.

Overflow occurs when

AVAIL = NULL and there is an insertion

The term underflow refers to the situation where one wants to delete data from a data structure that is empty.

Underflow occurs when START = NULL and there is a deletion.

**Insertion into a Linked List**

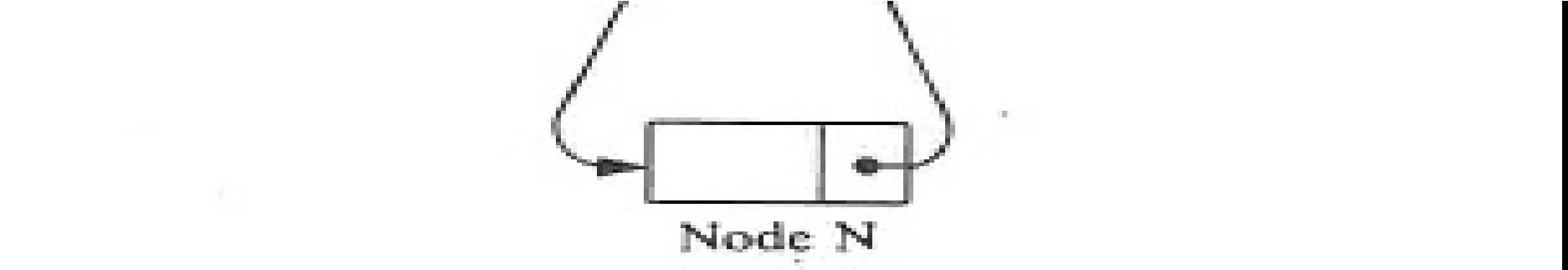
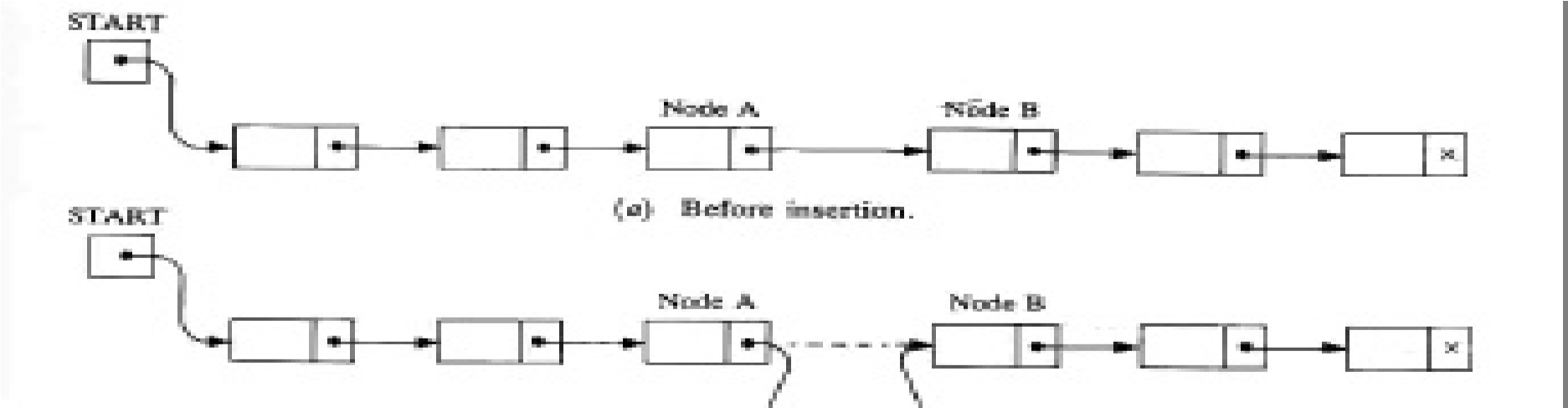
Let LIST be a linked list with successive nodes A and B, as pictured in Fig. (a)

Suppose a node N is to be inserted into the list between nodes A and B. The diagram of such an insertion appears in Fig. (b). That is, node A now points to the new node N, and node N points to node B, to which A previously pointed.

Suppose our linked list is maintained in memory in the form

LIST (INFO, LINK, START, AVAIL)

Fig a and fig b



**Insertion Algorithms**

Algorithms, which insert nodes into linked lists, come up in various situations.

Two of them are:

The first one inserts a node at the beginning of the list, the second one inserts a node after the node with a given location,

All the algorithms assume that the linked list is in memory in the form LIST (INFO, LINK, START, AVAIL) and that the variable ITEM contains the new information to be added to the list.

Since our insertion algorithms will use a node in the AVAIL list, all of the algorithms will include the following steps:

1. Checking to see if space is available in the AVAIL list. If not, that is, if AVAIL = NULL, then the algorithm will print the message OVERFLOW.
2. Removing the first node from the AVAIL list. Using the variable NEW to keep track of the location of the new node, this step can be implemented by the pair of assignments (in this order)

NEW: = AVAIL, AVAIL: = LINK [AVAIL]

1. Copying new information into the new node. In other words,

INFO [NEW]: = ITEM

**Inserting at the Beginning of a List**

Suppose our linked list is not necessarily sorted and there is no reason to insert a new node in any special place in the list. Then the easiest place to insert the node is at the beginning of the list.

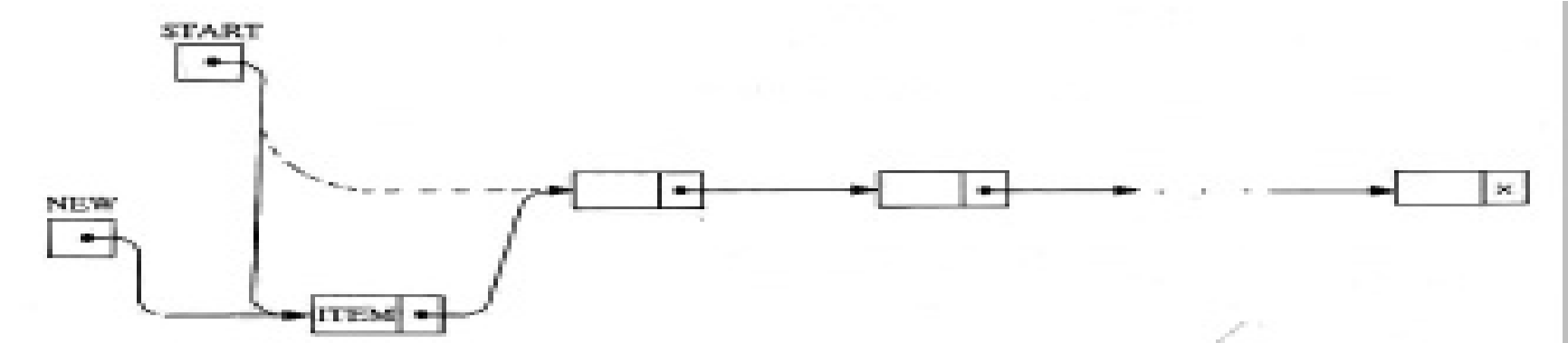
An algorithm that does so follows:

**INSFIRST (INFO, LINK, START, AVAIL, ITEM).**This algorithm inserts

ITEM as the first node in the list.

1. [OVERFLOW?] If AVAIL = NULL, then: Write: OVERFLOW, and Exit.
2. [Remove first node from AVAIL list.] Set NEW:= AVAIL and AVAIL:= LINK[AVAIL].
3. Set INFO[NEW]:= ITEM. [Copies new data into new node.]
4. Set LINK[NEW]:= START. [New node now points to original first node.]
5. Set START:= NEW. [Changes START so it points to the new node.] 6. Exit.

Fig: Insertion at the beginning of a list



**Inserting after a Given Node**

Suppose we are given the value of LOC where either LOC is the location of a node A in a linked LIST or LOC = NULL.

The following is an algorithm which inserts ITEM into LIST so that ITEM follows node A or, when LOC = NULL, so that ITEM is the first node. Let N denote the new node (whose location is NEW). If LOC = NULL, then N is inserted as the first node in LIST

Otherwise, let node N point to node B (which originally followed node A) by the assignment

LINK [NEW]: = LINK [LOC] and let node A point to the new node N by the assignment LINK [LOC] : = NEW

INSLOC (INFO, LINK, START, AVAIL, LOC, ITEM)

This algorithm inserts ITEM so that ITEM follows the node with location LOC or inserts ITEM as the first node when LOC = NULL.

1. [OVERFLOW?] If AVAIL = NULL, then: Write: OVERFLOW, and Exit.
2. [Remove first node from AVAIL list] Set NEW:= AVAIL and AVAIL:= LINK[AVAIL].
3. Set INFO [NEW]:= ITEM. [Copies new data into. new node.]
4. If LOC = NULL, then: [Insert as first node.] Set LINK [NEW]: = START and START: =

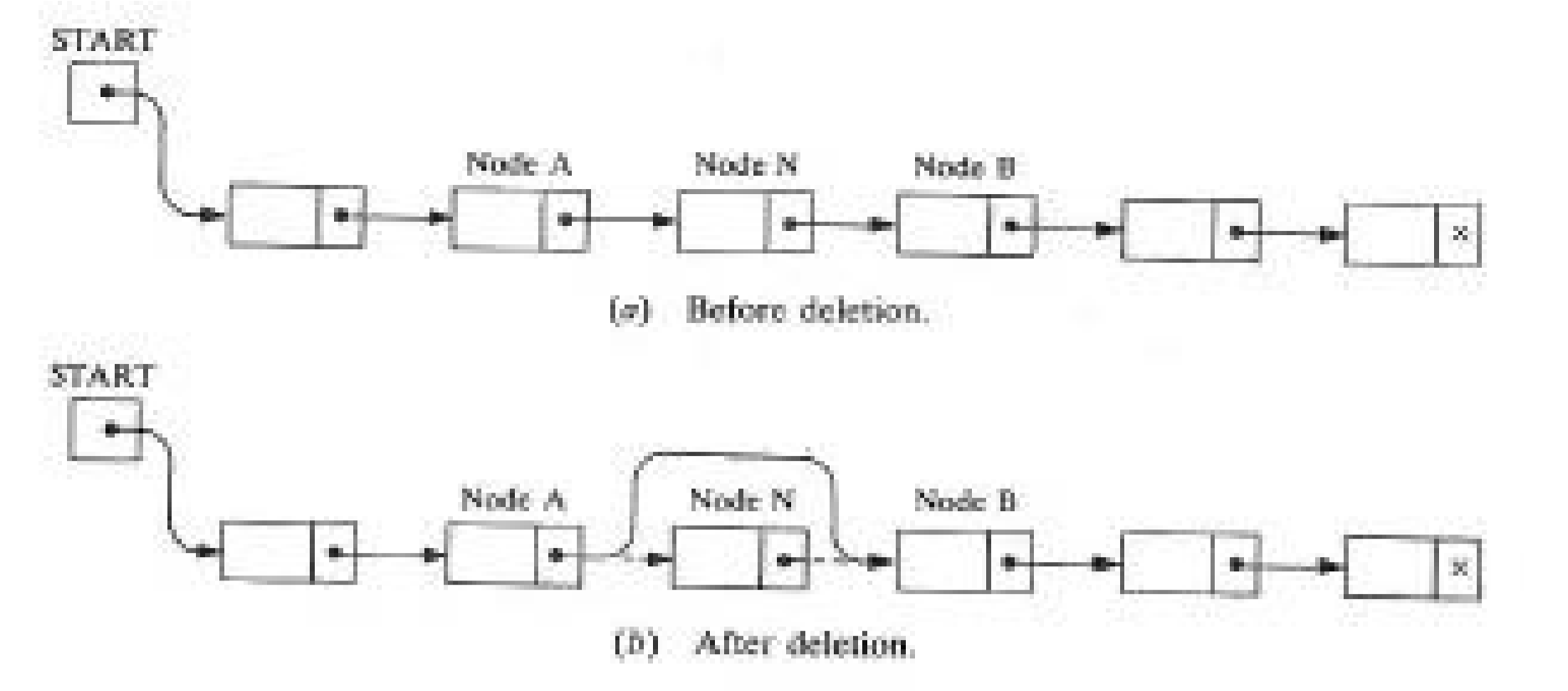
NEW. Else: [Insert after node with location LOC.] Set LINK [NEW]: = LINK [LOC] and LINK[LOC]: = NEW. [End of If structure.] 5. Exit.

**Deletion from a Linked List**

Let LIST be a linked list with a node N between nodes A and B, as pictured in Fig.(a). Suppose node N is to be deleted from the linked list.

The diagram of such a deletion appears in Fig. (b). the deletion occurs as soon as the next pointer field of node A is changed so that it points to node B. (when performing deletions, one must keep track of the address of the node which immediately precedes the node that is to be deleted.)

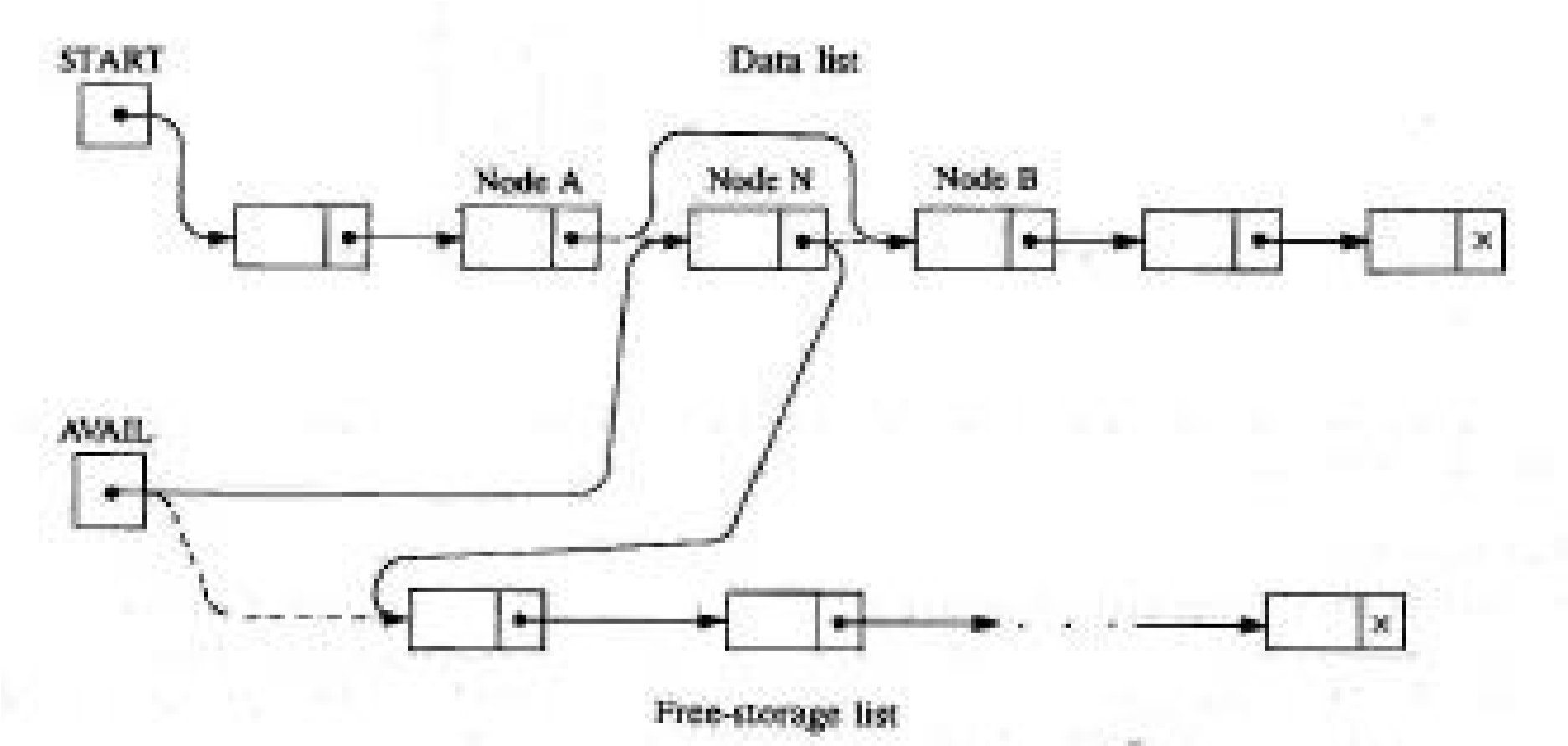
Suppose our linked list is maintained in memory in the form LIST (INFO, LINK, START, AVAIL)



The previous figure does not take into account the fact that, when a node N is deleted from our list, we will immediately return its memory space to the AVAIL list. Specifically, for easier processing, it will be returned to the beginning of the AVAIL list.

Thus a more exact diagram of such a deletion is the one in Fig. below. Observe that three pointer fields are changed as follows:

1. The nextpointer field of node A now points to node B, where node N previously pointed.
2. The nextpointer field of N now points to the original first node in the free pool, where AVAIL previously pointed.
3. AVAIL now points to the deleted node N.

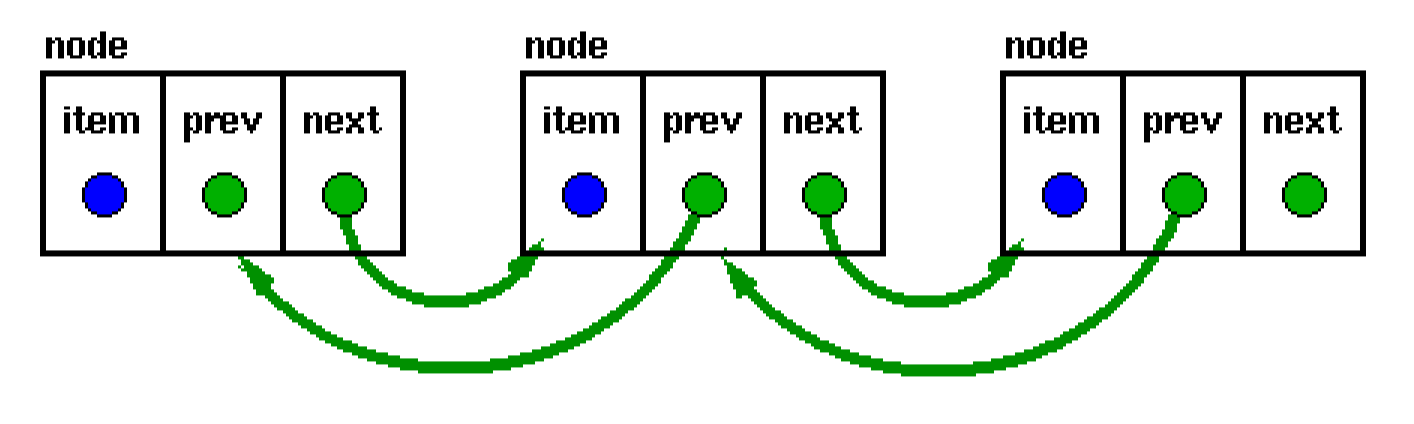


**Singly linked list**

* + - A singly linked list is a collection of nodes each node is a structure it consisting of an element and a pointer to a structure containing its successor, which is called a next pointer.

**Doubly linked list**

* + - Doubly linked list is a collection of nodes where each node is a structure containing the following fields
      1. Pointer to the previous node.
      2. Data.
      3. Pointer to the next node.



Doubly linked lists have a pointer to the preceding item as well as one to the next.

They permit scanning or searching of the list in both directions. To go backwards in a simple list, it is necessary to go back to the start and scan forwards. Many applications require searching backwards and forwards through sections of a list

**How to find target key in a linked list**

To find the target key in a linked list, you have to apply sequential search. Each node is traversed and compared with the target key, and if it is different, then it follows the link to the next node. This traversal continues until either the target key is found or if the last node is reached.

**Advantages of linked lists over arrays**

* + Array size is fixed and cannot change at run time, but in link list we can create memory according to requirement.
  + In array we define but at the run time it is not used so in that case memory is wasted. The memory distribution in linked lists is not contiguous unlike arrays, i.e., the memory is taken from the space available anywhere, and then a link is set with the previous node. However, arrays don not have this feature, and the memory distribution should be contiguous.
  + Insertions and deletion in linked lists are easier than in arrays.
  + It is easier to access elements in linked lists than in arrays

* + **Stacks**

A stack can be perceived as a special type of the list where insertions and deletions take place only at the one end, referred to as the top of a stack.

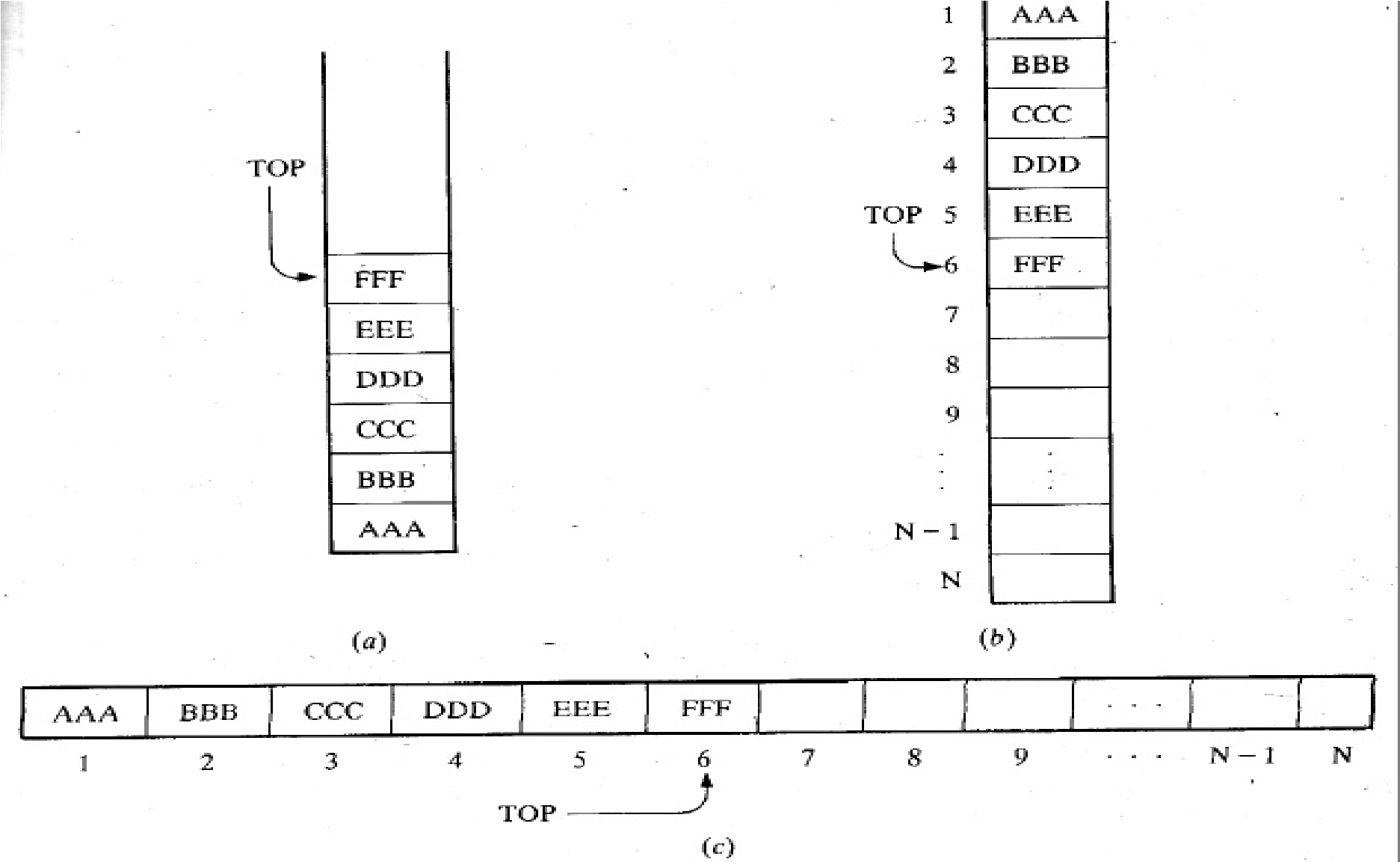
A stack is a list of elements in which an element may be inserted or deleted only at one end, called the top of the stack. This means, in particular, that elements are removed from a stack in the reverse order of that in which they were inserted into the stack; it is a linear structure in which items may be added or removed only at one end.

Special terminology is used for two basic operations associated with stacks:

(a) “Push” is the term used to insert an element into a stack. (b)“Pop” is the term used to delete an element from a stack.

Stacks are also called last-in first-out (LIFO) lists. This is because the last item to be added to a stack is the first item to be removed.

Suppose the following 6 elements are pushed, in order, onto an empty stack: AAA, BBB, CCC, DDD, EEE, FFF. The figure below shows three ways of picturing such a stack.



In this case, to remove an element from the stack, we start with the top most element which was last to be pushed in. the element is FFF followed by EEE and so on until the stack is empty.

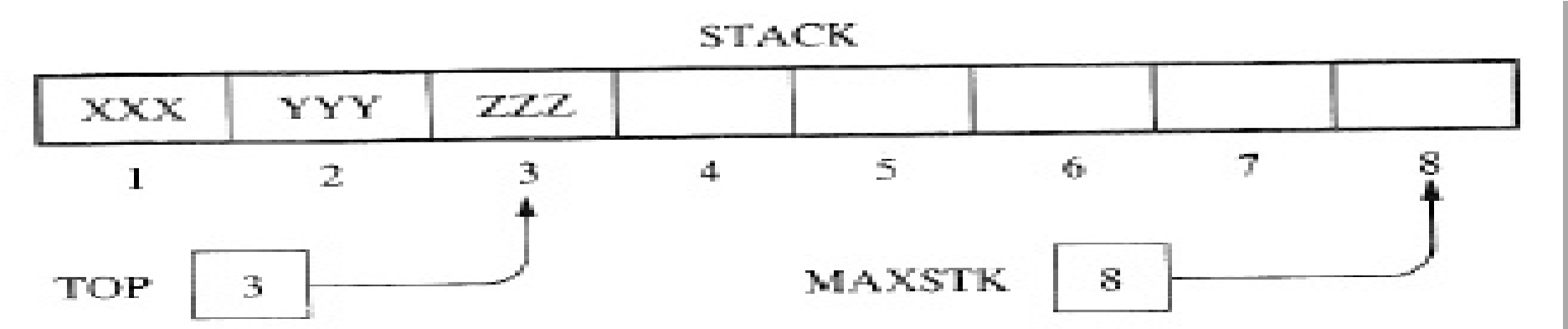
**Array Representation of Stacks**

Stacks may be represented in the computer in various ways, usually by means of a one-way list or a linear array.

Unless otherwise stated or implied, each of our stacks will be maintained by a linear array STACK; a pointer Variable TOP, which contains the location of the top element of the stack; and a variable MAXSTK which gives the maximum number of elements that can be held by the stack.

The condition TOP = 0 or TOP = NULL will indicate that the stack is empty. Example

Figure below pictures such an array representation of a stack.



Since TOP = 3, the stack has three-elements, XXX, YYY and ZZZ; and since MAXSTK = 8, there is room for 5 more items in the stack.

**PUSH and POP**

The operation of adding (pushing) an item onto a stack and the operation of removing (popping) an item from a stack may be implemented, respectively, by the following procedures, called PUSH and POP.

In executing the procedure PUSH, one must first test whether there is room in the stack for the new item; if not, then we have the condition known as overflow.

Analogously, in executing the procedure POP, one must first test whether there is an element in the stack to be deleted; if not, then we have the condition known as underflow.

**Procedure: PUSH**  PUSH (STACK, TOP, MAXSTK, ITEM)

//This procedure pushes an ITEM onto a stack.

1. [Stack already filled?] If TOP = MAXSTK, then: Print OVERFLOW, and Return.

2.Set TOP: =TOP+1. [Increases TOP by 1.]

1. Set STACK [TOP]: = ITEM. [Inserts ITEM in new TOP position.
2. Return.

**Procedure: POP**

POP (STACK, TOP, ITEM)

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

1. [Stack has an item to be removed?]If TOP = 0, then: Print: UNDERFLOW, and Return.
2. Set ITEM: = STACK [TOP]. [Assigns TOP element to ITEM.] 3. Set TOP: = TOP - 1. [Decreases TOP by 1.]

4. Return.

Example

Consider the stack in previous figure. We simulate the operation PUSH (STACK,

WWW):

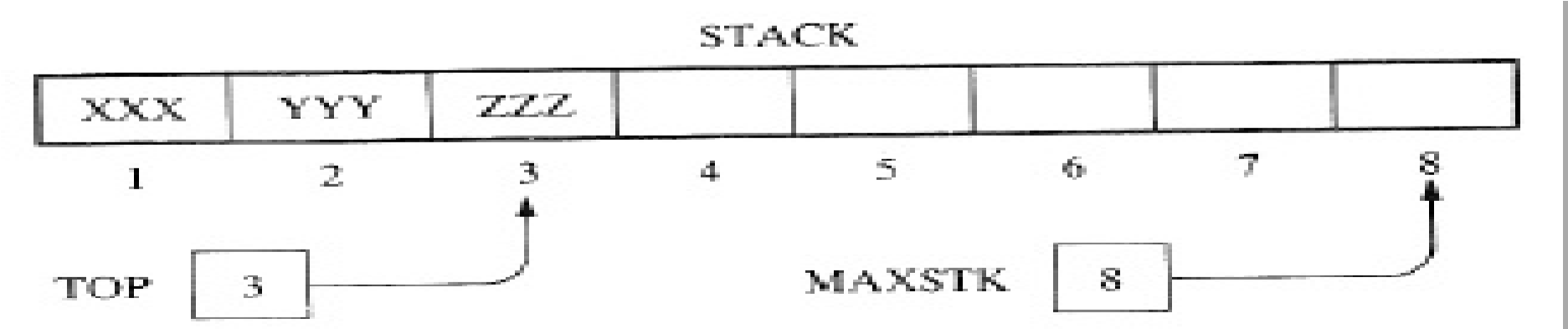
1. Since TOP = 3, control is transferred to Step 2.
2. TOP = 3 + 1 = 4.
3. STACK [TOP] = STACK [4] = WWW.
4. Return.

Note that WWW is now the top element in the stack.

Consider again the same stack. This time we simulate the operation POP (STACK, ITEM):

1. Since TOP = 3, control is transferred to Step 2.
2. ITEM = ZZZ.
3. TOP = 3 - 1 = 2.
4. Return.

Observe that STACK [TOP] = STACK [2] =YYY is now the top element in the stack.



**Application of stacks**

Polish notation

Polish notation, named after the Polish mathematician Jan Lukasiewicz, refers to the notation in which the operator symbol is placed before its two operands. For example,+AB –CD \*EF /GH

For most common arithmetic operations, the operator symbol is placed between its two operands. For example, A+B C-D E\*F G/H

This is called infix notation. With this notation, we must distinguish between (A+B)\*C and A + (B \* C) by using either parentheses or some operator-precedence convention

The order of the operators and operands in an arithmetic expression does uniquely determine the order in which the operations are to be performed. We translate, step by step, the following infix expressions into Polish notation using brackets [ ] to indicate a partial translation:

(A + B) \*C= [+AB]\*C = \*+ABC

A + (B\*C) = A + [\*BC] = +A\*BC

(A + B) / (C - D) = [+ AB] / [ - CD] = / + AB - CD

The fundamental property of Polish notation is that the order in which the operations are to be performed is completely determined by the positions of the operators and operands in the expression. One never needs parentheses when writing, expressions in Polish notation AB+ CD- EF\* GH/

**Reverse Polish notation**

Reverse Polish notation (RPN) is a method for representing expressions in which the operator symbol is placed after the arguments being operated on unlike in polish notation, in which the operator comes before the operands.

For example, the following RPN expression will produce the sum of 2 and 3, namely 5: 2 3 +.

Reverse Polish notation, also known as postfix notation, contrasts with the "infix notation" of standard arithmetic expressions in which the operator symbol appears between the operands.

**Infix, Postfix and Prefix notations**

Infix, Postfix and Prefix notations are three different but equivalent ways of writing expressions.

Infix notation: X + Y

Operators are written in-between their operands. This is the usual way we write expressions. An expression such as A \* ( B + C ) / D is usually taken to mean something like: "First add B and C

together, then multiply the result by A, then divide by D to give the final answer."

Postfix notation

Postfix notation (also known as "Reverse Polish notation"): XY +

Operators are written after their operands. The infix expression given above is equivalent to A B C + \* D /

The order of evaluation of operators is always left-to-right, and brackets cannot be used to change this order. Because the "+" is to the left of the "\*" in the example above, the addition must be performed before the multiplication.

Operators act on values immediately to the left of them. For example, the "+" above uses the "B" and "C". We can add (totally unnecessary) brackets to make this explicit: ( (A (B C +) \*) D /) .Thus, the "\*" uses the two values immediately preceding: "A", and the result of the addition. Similarly, the "/" uses the result of the multiplication and the "D".

Prefix notation

Prefix notation (also known as "Polish notation"): + X Y Operators are written before their operands. The expressions given above are equivalent to / \* A + B C D

Although Prefix "operators are evaluated left-to-right", they use values to their right, and if these values themselves involve computations then this changes the order that the operators have to be evaluated in. In the example above, although the division is the first operator on the left, it acts on the result of the multiplication, and so the multiplication has to happen before the division (and similarly the addition has to happen before the multiplication).

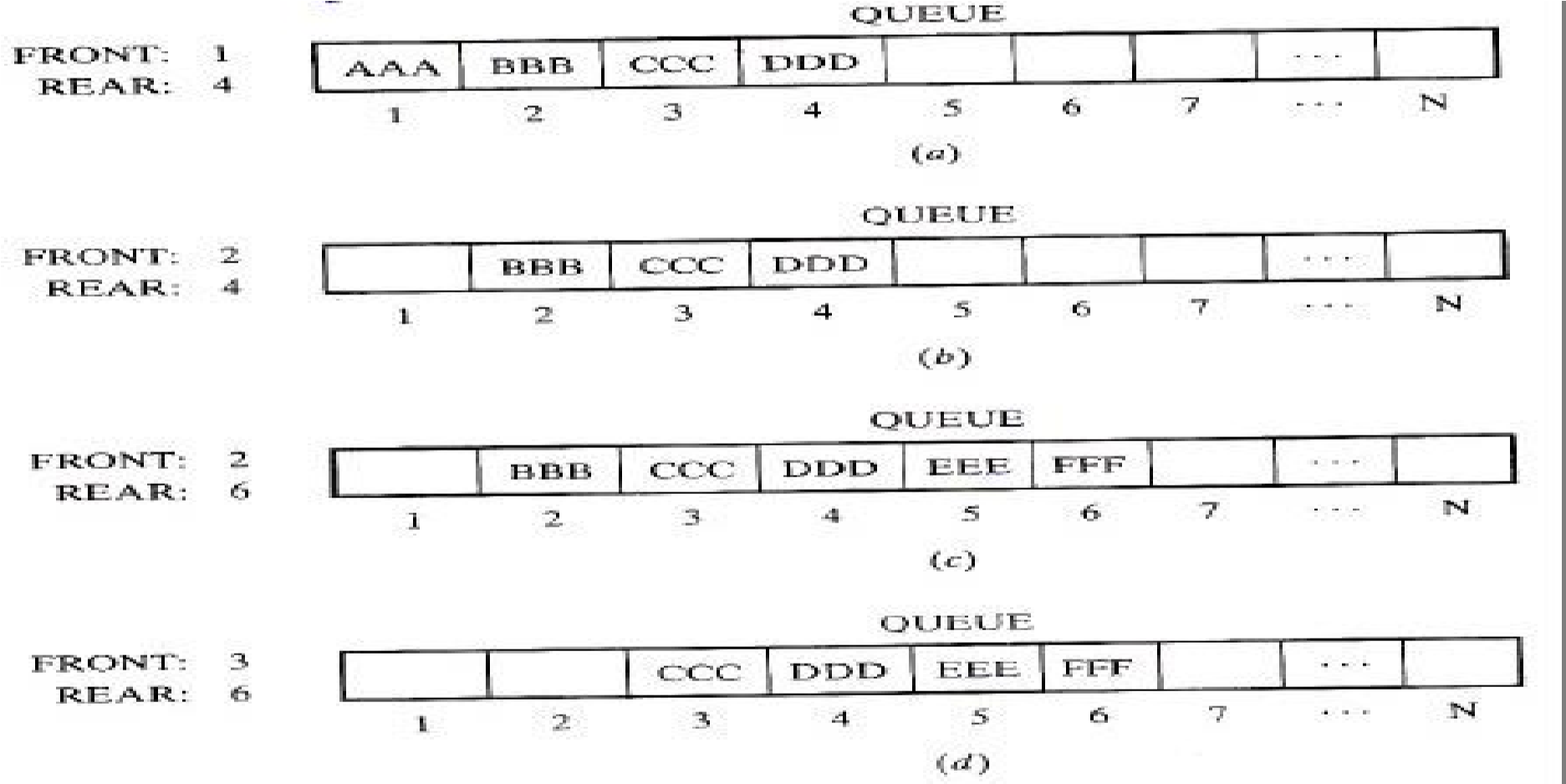
Note: In all three versions, the operands occur in the same order, and just the operators have to be moved to keep the meaning correct. (This is particularly important for asymmetric operators like subtraction and division: A - B does not mean the same as B - A; the former is equivalent to A B - or - A B, the latter to B A - or - B A).

Converting between notations

|  |  |  |
| --- | --- | --- |
| **Infix** | **Postfix** | **Prefix** |
| ( (A \* B) + (C / D) ) | ( (A B \*) (C D /) +) | (+ (\* A B) (/ C D) ) |
| ((A \* (B + C) ) / D) | ( (A (B C +) \*) D /) | (/(\*A(+BC))D) |
| (A \* (B + (C / D) ) ) | (A (B (C D /) +) \*) | (\* A (+ B (/ C D) ) ) |

**Queues**

* In real life a queue is a line of customers waiting for service of some kind. In most cases, the first customer in line is the next customer to be served.
* A queue is a particular kind of abstract data type or collection in which the entities in the collection are kept in order and the principal operations on the collection are the addition of entities to the rear terminal position, known as **enqueue**, and removal of entities from the front terminal position, known as **dequeue**.
* This makes the queue a First –in First- out (FIFO) data structure in that the first element added to the queue will be the first one to be removed.



* The access procedures for a queue include operations such as examining whether the queue is empty, inspecting the item at the front of the queue but not others, placing an item at the back of the queue, but at no other position, and removing an item from the front of the queue, but from no other position.

**QueueException Class**

* Adding an element to a full queue and removing an element from an empty queue would generate errors or exceptions
  + Queue overflow exception
  + Queue underflow exception Classes that handle these exceptions
  + QueueException extends RunTimeException
  + QueueOverflowException extends QueueException
  + QueueUnderflowException extends QueueException

**Implementation of Queues as Arrays** Instance variables

* + queueFront: keeps track of the first element
  + queueRear: keeps track of the last element
  + maxQueueSize: specifies the maximum size of the queues

**Empty Queue and Full Queue**

Methods isEmptyQueue and isFullQueue

public boolean isEmptyQueue()

{ return (count == 0);

} public boolean isFullQueue()

{ return (count == maxQueueSize);

}

**Circular Queue**

Circular Queue-Circular queue is a linear data structure. It follows FIFO principle. In circular queue the last node is connected back to the first node to make a circle. Circular linked list follow the First In First Out principle. Elements are added at the rear end and the elements are deleted at front end of the queue.

* A typical application of queues is resource allocation by operating systems on multi-user machines. For instance, if you share a printer with other users, a request to print enters a queue to wait its turn.

**Priority queue**

* In real life a queue is a line of customers waiting for service of some kind. In most cases, the first customer in line is the next customer to be served.
* There are exceptions, though. For example, at airports customers whose flight is leaving imminently are sometimes taken from the middle of the queue. Also, at supermarkets a polite customer might let someone with only a few items go first.
* The most general queueing discipline is priority queueing, in which each customer is assigned a priority, and the customer with the highest priority goes first, regardless of the order of arrival. The priority can be based on anything: what time a flight leaves, how many groceries the customer has, or how important the customer is.
* Priority Queue - Priority queue is a data structure that allows at least the following two operations.

1. Insert-inserts an element at the end of the list called the rear.
2. DeleteMin-Finds, returns and removes the minimum element in the priority