Algorithms and Data Structures (CS211)

What this course will cover?

Lectures:

1. Introduction to algorithms and data structures
2. Algorithm/complexity analysis
3. Recursion
4. Lists, linked lists
5. Stacks, queues
6. Trees (Trees, Sets, Maps, Graphs)
7. Graph algorithms (Shortest-path, Dijkstra, ...)
8. Searching and Sorting algorithms

Textbooks:

- Michael T. Goodrich , Roberto Tamassia, Michael H. Goldwasser . “Data Structures and Algorithms in Java”, 6th edition 2014,

- Mark A. Weiss, "Data Structures and Algorithm Analysis in Java", 3th ed., Addison Wesley,2011 , ISBN: 0-132-57627-9.

- A. Drozdek, "Data Structures and Algorithms in Java", 3th edition, Cengage Learning, 2008.

Lecture 2: Algorithm

Algorithm

- **Algorithm** is an explicit and unambiguous sequence of elementary instructions
- describes actions on the input instance.
- There are many correct algorithms for solving the **same problem.**

Algorithm& Program

- **Algorithm**: outline the essence of a computational procedure, step-by-step instruction.
- **Program**: an implementation of an algorithm in some programming language.

What is a good algorithm?

Efficient:

- ❖Running time.
- ❖Size of algorithm.
- ❖Number of data elements.
- ❖Space used (The number of bits in an input number).

Algorithm Analysis

- **Why we should analyze algorithms?**
  - **Predict the resources that the algorithm requires**
    - **Computational time (CPU consumption)**
    - **Memory space (RAM consumption)**

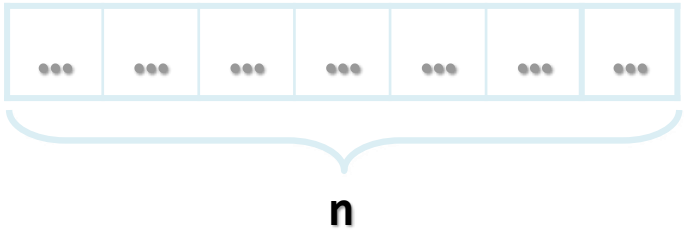
  - **The running time of an algorithm is:**
    - **The total number of primitive operations executed**
    - **Also known as algorithm complexity**

Time Complexity

- Worst-case
  - An upper bound on the running time for any input of given size
- Average-case
  - Assume all inputs of a given size are equally likely
- Best-case
  - The lower bound on the running time

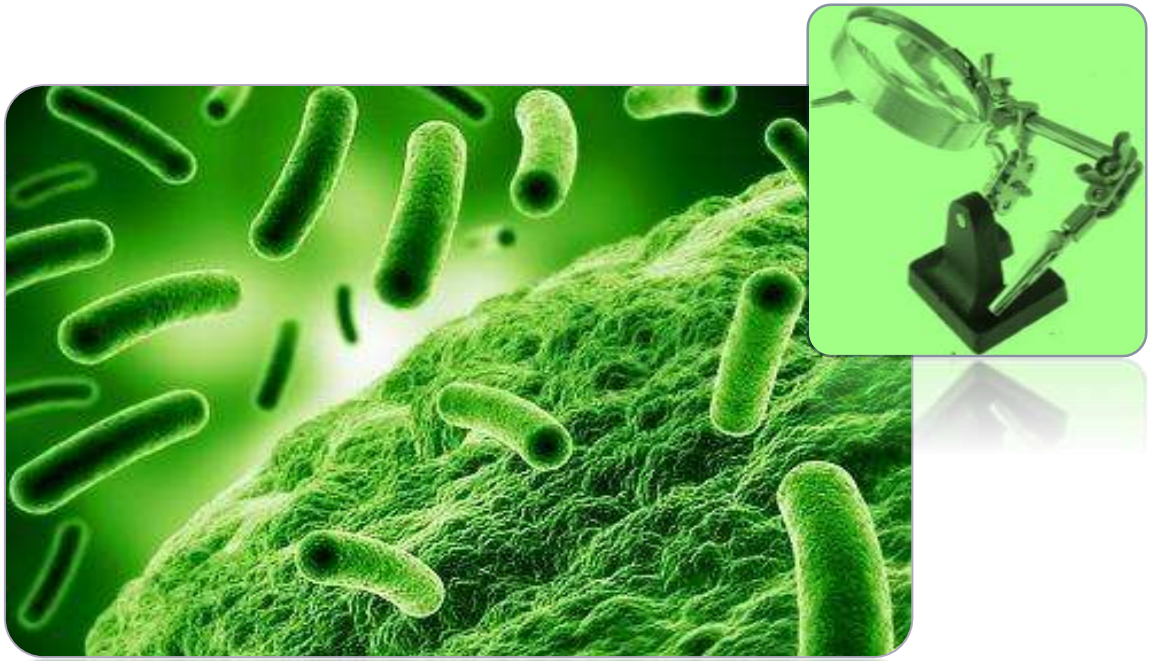
Time Complexity – Example

- Sequential search in a list of size n
  - Worst-case:
    - n comparisons
  - Best-case:
    - 1 comparison
  - Average-case:
    - n/2 comparisons
- The algorithm runs in linear time
  - Linear number of operations



Algorithms Complexity

- **Algorithm complexity** is rough estimation of the number of steps performed by given computation depending on the size of the input data
  - Measured through **asymptotic notation**
    - O(g) where g is a function of the input data size
  - Examples:
    - Linear complexity O(n) – all elements are processed once (or constant number of times)
    - Quadratic complexity $O(n^2)$ – each of the elements is processed n times



Analyzing Complexity of Algorithms

Examples

Complexity Examples(1)

```
int FindMaxElement(int[] array)
{
    int max = array[0];
    for (int i=0; i<int n; i++)
    {
        if (array[i] > max)
        {
            max = array[i];
        }
    }
    return max;
}
```

- Runs in O(n) where n is the size of the array
- The number of elementary steps is ~ n

Complexity Examples (2)

```
decimal Sum3(int n)
{
    decimal sum = 0;
    for (int a=0; a<n; a++)
        for (int b=0; b<n; b++)
            for (int c=0; c<n; c++)
                sum += a*b*c;
    return sum;
}
```

- Runs in cubic time $O(n^3)$
- The number of elementary steps is $\sim n^3$

Complexity Examples (3)

```
long SumMN(int n, int m)
{
    long sum = 0;
    for (int x=0; x<n; x++)
        for (int y=0; y<m; y++)
            sum += x*y;
    return sum;
}
```

- Runs in quadratic time O(n*m)
- The number of elementary steps is ~ n*m

Complexity Examples (4)

```
long SumMN(int n, int m)
{
    long sum = 0;
    for (int x=0; x<n; x++)
        for (int y=0; y<m; y++)
            if (x==y)
                for (int i=0; i<n; i++)
                    sum += i*x*y;
    return sum;
}
```

- **Runs in quadratic time O(n*m)**

Complexity Examples (5)

```
decimal Factorial(int n)
{
    if (n==0)
        return 1;
    else
        return n * Factorial(n-1);
}
```

- Runs in linear time O(n)
- The number of elementary steps is ~ n

Recursion

Reduction is the single most common technique used in designing algorithms.

Recursion is a particularly powerful kind of reduction. In which the method call itself.

Java program to sum the integers from 0 to n using recursion.

```
package javaapplication69;
public class JavaApplication69 {
static  int sum(int n){
  if(n>=1)return sum(n-1)+n;     return n;    }
    public static void main(String[] args) {
 int x= sum(5);
System.out.println(x);    }}
```

What is a data structure?

Any computer software deals with **data**

- **Organize** …. **Store** …. **Process** …. **Use** it!

Facebook data: Your account details (name, email, password), your friends list, your images and videos, your posts (replies, likes), Ads, groups, attached files … etc

Dictionary: There are many thousands of words? How to store the words? Sorted?

Google maps: I want to go From School to Home? From Cairo to London? How to store these locations/path ? How to efficiently find the path between 2 points?

Classification of Data Types

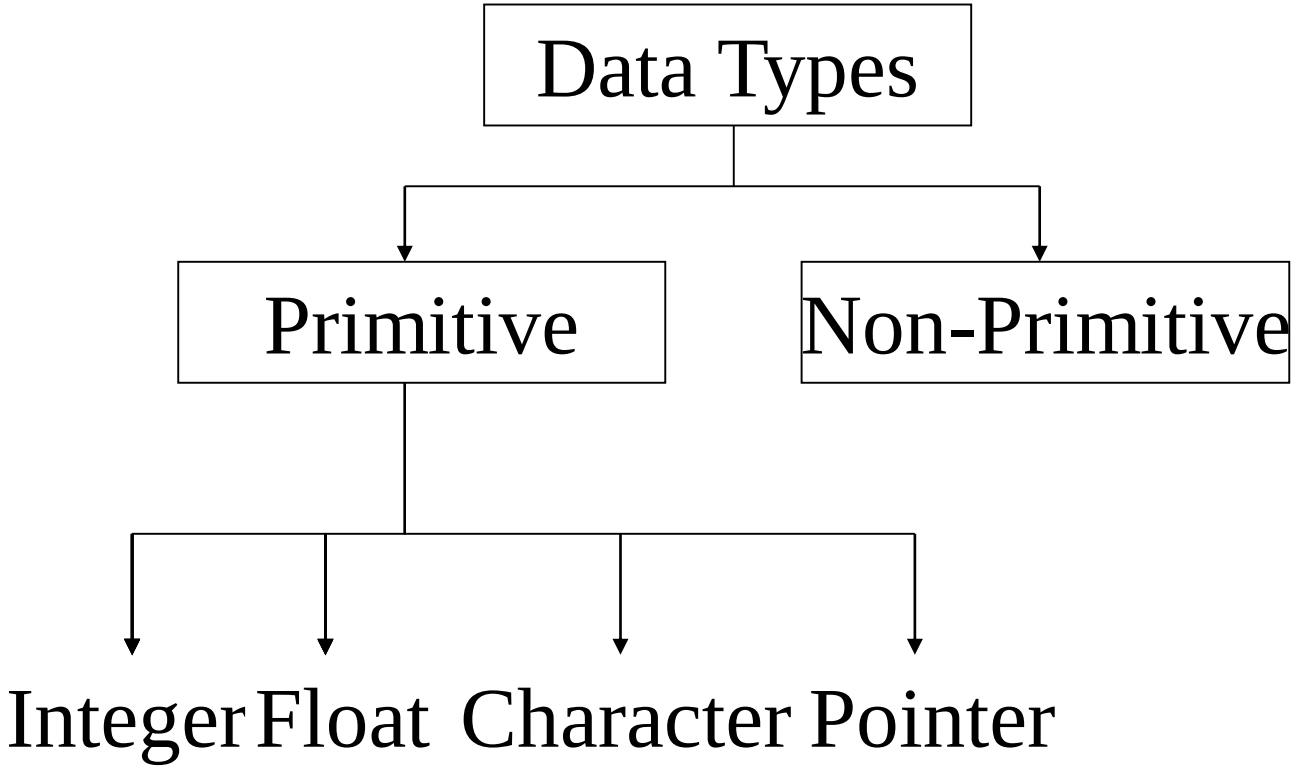
- Data types are normally divided into two categories:
  - Primitive or Simple Data Types
  - Non-Primitive or Structured Data Types

Classification of Data Types

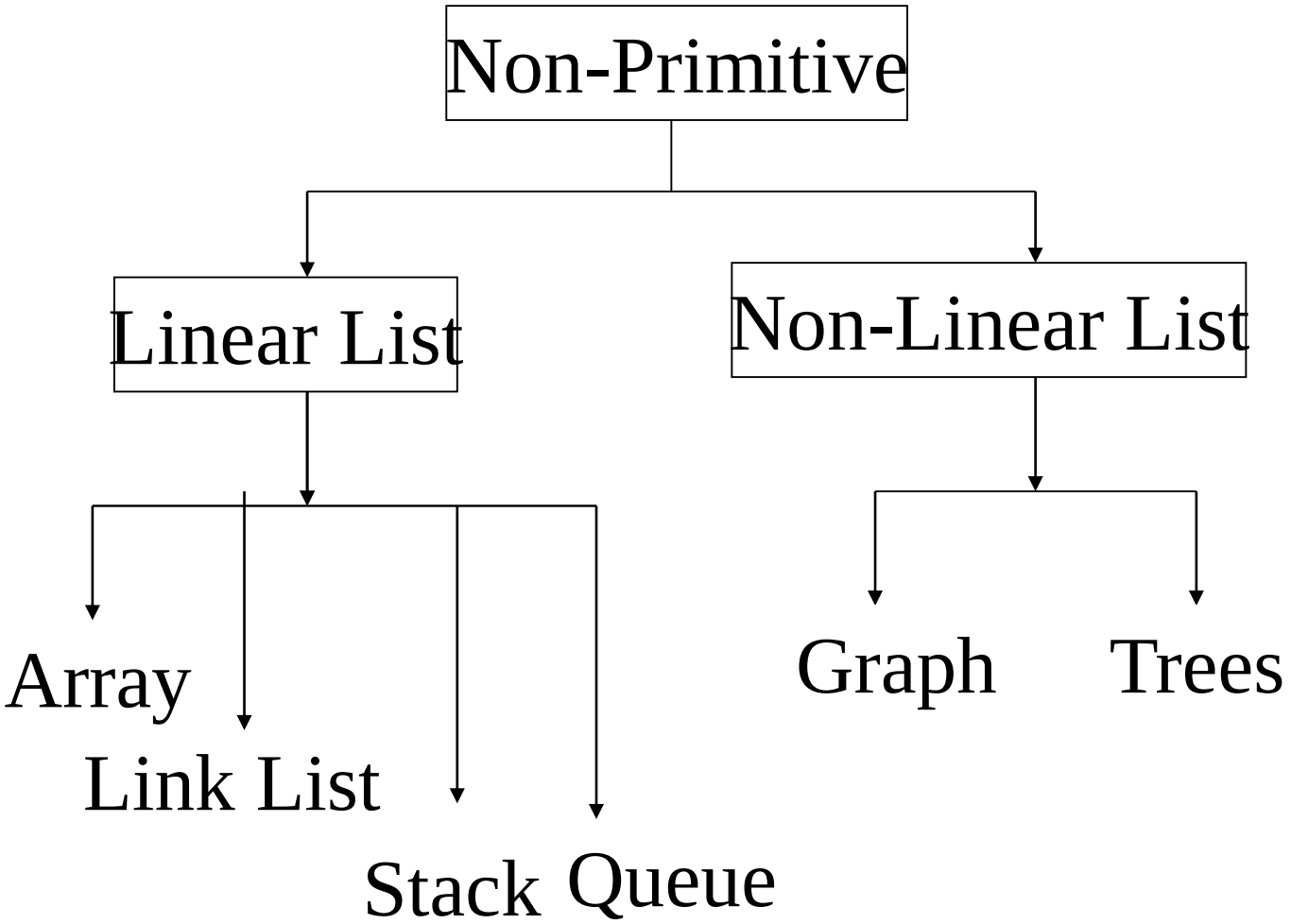
- **Simple Data** types: also known as atomic data types → have no component parts. E.g. int, char, float, etc.

- **Structured Data** types: hold a collection of data values. This collection will generally consist of the primitive data types.

Classification of Data Types



Classification of Data Types



Non-Primitive Data Types

- The most commonly used operation on data structure are broadly categorized into following types:
  - Create
  - Selection
  - Updating
  - Searching
  - Sorting
  - Merging
  - Delete

Data Structures

data structure: is a way to organize information to enable *efficient* computation over that information

A data structure supports certain *operations*, each with a:

- Meaning: what does the operation do/return
- Performance: how efficient is the operation

Examples:

- ***List*** with operations **insert** and **delete**
- ***Stack*** with operations **push** and **pop**

Abstract Data Types (ADTs)

- in Object-oriented programming, **abstraction** is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it.

- **Abstraction**? Anything that hides details & provides only the essentials.
- Examples: an integer $165 = 1.10^2+6.10^1+5.10^0$, procedures/subprograms, etc.

- **Abstract Data Types (ADTs):** Simple or structured data types whose implementation details are hidden…

Characteristics of Data Structures:

Data Structure	Advantages	Disadvantages
Array	Quick inserts Fast access if index known	Slow search Slow deletes Fixed size
Ordered Array	Faster search than unsorted array	Slow inserts Slow deletes Fixed size
Stack	Last-in, first-out access	Slow access to other items
Queue	First-in, first-out access	Slow access to other items
Linked List	Quick inserts Quick deletes	Slow search
Binary Tree	Quick search Quick inserts (If the tree remains balanced)	Deletion algorithm is complex
Graph	Best models real-world situations	Some algorithms are slow and very complex

Data Structures Efficiency

Data Structure	Add	Find	Delete	Get-by-index
Array (T[])	O(n)	O(n)	O(n)	O(1)
Linked list (LinkedList<T>)	O(1)	O(n)	O(n)	O(n)
Resizable array list (List<T>)	O(1)	O(n)	O(n)	O(1)
Stack (Stack<T>)	O(1)	-	O(1)	-
Queue (Queue<T>)	O(1)	-	O(1)	-

Choosing Data Structure

- **Arrays (T[])**
  - **Use when fixed number of elements should be processed by index**
- **Resizable array lists (List<T>)**
  - **Use when elements should be added and processed by index**
- **Linked lists (LinkedList<T>)**
  - **Use when elements should be added at the both sides of the list**
  - **Otherwise use resizable array list (List<T>)**

Choosing Data Structure

- **Stacks (Stack<T>)**
  - **Use to implement LIFO (last-in-first-out) behavior**
  - **List<T> could also work well**
- **Queues (Queue<T>)**
  - **Use to implement FIFO (first-in-first-out) behavior**
  - **LinkedList<T> could also work well**
- **Hash table based dictionary (Dictionary<K,T>)**
  - **Use when key-value pairs should be added fast and searched fast by key**
  - **Elements in a hash table have no particular order**

lecture 2: Array

Lecture content:

- ARRAYS
  - Meaning of Array
  - An *array* is a data structure
  - Arrays in Java
  - Declaring Arrays
  - An array features
- FUNCTIONS (METHODS)
  - Method declaration & definition with parameters
  - Method calling
- Pointer
- Struct

Meaning of Array

You learned how to store a single data point in a variable.

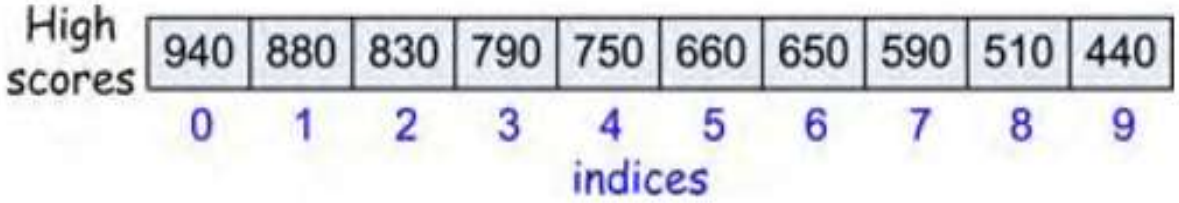
Then, in the Datatypes, you learned that its best to store data of a certain type in a variable with that respective type:

- store a character string in a String variable,
- store a whole number, integer in an *int* variable,
- store a floating point number in a *double* variable, and so on.

Now that we are becoming more advanced Java programmers, we want to store a collection of data in a single data structure for later manipulation.

Array

- **Array** is a collection of variables of the same type. Each variable(cell) has an index, refer to the value stored in that cell.



Making an array in a Java program involves three distinct steps:

- Declare the array name.
- Create the array.
- Initialize the array values.

We refer to an array element by putting its index in square brackets after the array name : the code a[i] refers to element i of array a[] .

Pointer type

- The ***pointer* type** is a simple type whose domain elements are *memory addresses*.
- **Pointer types** give the programmer access to the memory through indirect addressing.
- **The pointer type** plays an all-important role in abstract data structure implementations.
- Java doesn't have pointers but has references. All objects in java are references and can use them like pointers.

Example 1

For example , the following code makes an array of n numbers of type double
all initialized to： zero

```
double[] a; // declare the array

a = new double[n]; // create the array of length n

 for (int i = 0; i < n; i++) // elements are indexed from 0 to n-1.

 a[i] = 0.0; // initialize all elements to 0.0
```

Declaring Arrays

```
int[] p = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
//p is reference (pointer) to the performed array.
int[] p;
x=p;
//x is reference (pointer) to the same array.
```

```
double[] a;
a = new double[10];
for (int k=0; k < a.length; k++)
 {   a[k] = 1.0;   }
```

Example 2

```
/** Counts the number of times an integer 11 appears in an array
**/
int[] a = {2, 11, 5, 7, 11, 13, 17, 11, 23, 29};
int count = 0;
for (int k=0; k < a.length; k++) // note the use of the "for each"
                                                        //loop
{
    if (a[k] == 11)        // check if the current element equals 11
       count++; }
```

An array features

Here is the list of most important array features you must know (i.e. be able to program)

- copying and cloning
- insertion and deletion
- searching and sorting

Cloning an Array :

copies all of the cells of a into a new array and assigns p to point to that new array

```
int[] a = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
int[]p;
p=a.clone();    //COPYING A INTO P
for (int k=0; k < a.length; k++)
    System.out.println(p[k]);
```

```
run:
2
3
5
7
11
13
17
19
23
29
```

Illustrating Two-Dimensional Array

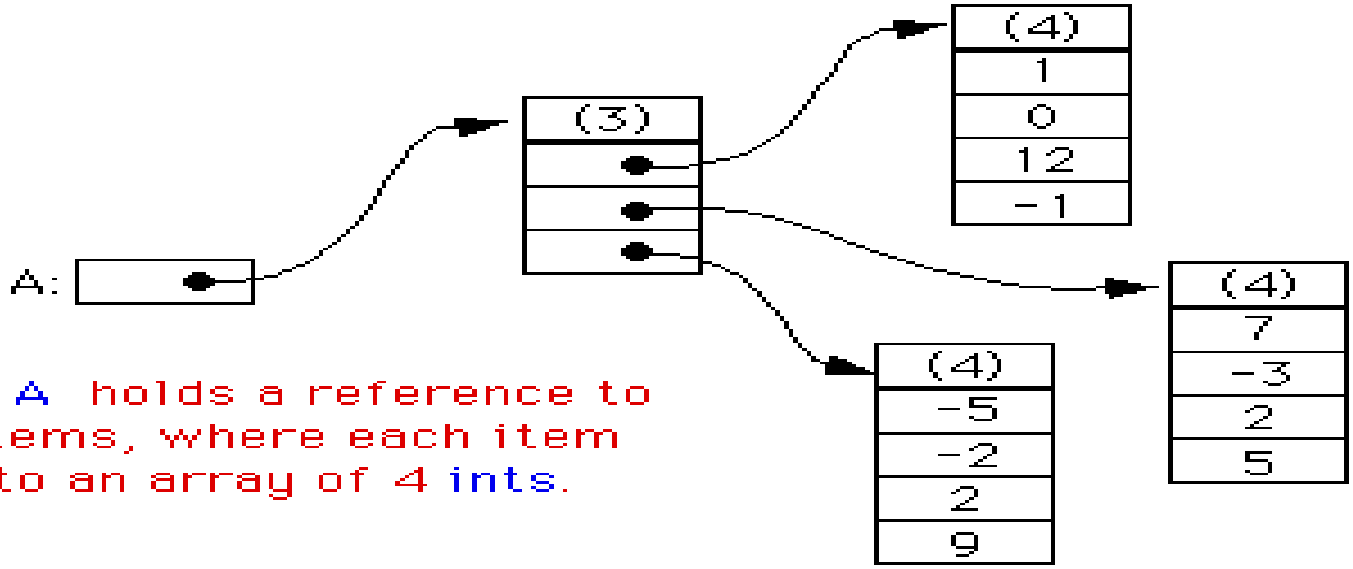
In a 2D array, we generally consider the first index to be the row, and the second to be the column:

a[*row, col*]

If you create an array `A = new int[3][4]`, you should think of it as a "matrix" with 3 rows and 4 columns.

A:

1	0	12	-1
7	-3	2	5
-5	-2	2	9



But in reality, A holds a reference to an array of 3 items, where each item is a reference to an array of 4 ints.

Processing Multidimensional Arrays

- Multidimensional arrays are often processed using for statements.

  To process all the items in a two-dimensional array, a for statement needs to be nested inside another.

  int[ ][ ] A = new int[3][4];

  The following loop stores 0 into each location in two dimensional array A :

```
for (int row = 0; row < 3; row++)
{
 for (int column = 0; column < 4; column++)
      {
        A [row][column] = 0;
      }
}
```

Home Work 1 :

- Using a mathematical function in the Java Math class and handling of two–dimensional arrays;
- - calculates the max – min value?

```
public class MaxMin{
    public static void main(String[]args)        {
          double mat[ ] [ ]= { {2.3, 5.1, 9.9}, {8.3, 4.5, 7.7},{
    5.2, 6.1, 2.8}}

        int n = mat.length;
        int m= mat[0].length;
        double maxmin = 0.0;
        for (int j = 0; j < m; j++){
          double min = mat[j][0];
          for (int i = 1; I <n ; i ++){
               min = Math.min(min,mat[i][j]);
          }
          if (j==0){
             maxmin = min;
          }else{
            maxmin = Math.max(maxmin, min);
          }
        }
        System.out.println("The max-min value is" + maxmin);
         }
    }
```

- Running the program produces the following result:

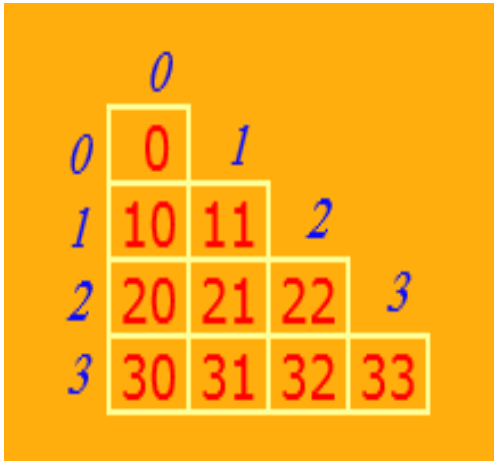
  The max-min value is 4.5

Ragged array ((Nonrectangular)OR(not regular)):

Ragged arrays are arrays of arrays such that:

member arrays can be of different sizes.
we can create a 2-D arrays but with variable number of columns in each row.

These type of arrays are also known as Jagged arrays.



each row can differ from the others.

Home Work 2:

- Give an example on Ragged array that declares a single-dimensional array having three elements, each of which is a single-dimensional array of integers.
- The first element is an array of 3 integers, the second is an array of 4 integers, and the third is an array of 2 integers. It uses initializers to fill the array elements with values.
-

```java
import java.util.Arrays;

class JaggedArray
{
    // Program to illustrate Jagged array in Java
    public static void main(String[] args)
    {
        // Declare a jagged array containing three elements
        int[][] arr = new int[3][];

        // Initialize the elements
        arr[0] = new int[] { 1, 2, 3 };
        arr[1] = new int[] { 4, 5, 6, 7 };
        arr[2] = new int[] { 8, 9 };

        // print the array elements
        for (int[] row : arr)
            System.out.println(Arrays.toString(row));
    }
}
```

Output:

```
[1, 2, 3]
[4, 5, 6, 7]
[8, 9]
```

Array of strings:

```
string s[]={“this” , “is” , “my” , “toy”};

for (int k=0; k < s.length; k++)
    System.out.print(s[k] + “    “);
```

Home Work 3:find maximum value

```
package array4;
import java.util.Scanner;
public class Array4 {
    public static void main(String[] args) {
Scanner input = new Scanner( System.in );
System.out.print("Enter three floating-point values : ");
double number1 = input.nextDouble();
double number2 = input.nextDouble();
double number3 = input.nextDouble();
double result = maximum( number1, number2, number3 );
System.out.println("Maximum is: " + result);
}
public static double maximum( double x, double y, double z )
{double maximumValue = x;
if ( y > maximumValue )
maximumValue = y;
if ( z > maximumValue )
maximumValue = z;
return maximumValue;}}
```

```
Enter three floating-point values : 4.5
1.2
9.7
Maximum is: 9.7
```

Home Work 4: Find average

```
package array3;
public class Array3 {
        public static void main(String[] args)
   {
      double d1 = 10.0;
      double d2 = 20.0;
      double d3 = 30.0;
      double d4 = 40.0;
     System.out.println( average(d1,d2));
     System.out.println( average(d1,d2,d3));
     System.out.println( average(d1,d2,d3,d4));
   }
        public static double average(double...numbers )    {
      double total = 0.0;
      for(double x: numbers)
         total+=x;
      return total /numbers.length ;      }     }
```

```
run:
15.0
20.0
25.0
```

Home work 5: Write a program in Java that:

- calculates the total in terms of the index of the row and column (two-dimensional Array([4] [3])
- print the output.

```
public class Array2 {
public static void main(String[] args) {
int[][]total=new int[4][3];
int[][]result; int i,j;
result=total;

for(i=0;i<total.length;i++)
{for(j=0;j<total[i].length;j++)
total[i][j]=(i+j);}

for(i=0;i<result.length;i++)
{for(j=0;j<result[i].length;j++)
System.out.print(" "+result[i][j]);
System.out.println();}
   }}
```

```
0 1 2
 1 2 3
 2 3 4
 3 4 5
```

Home work 6: Write a program in Java using scanner class to read the input and find the maximum and minimum value .

```
package array7;
import java.util.*;
public class Array7 {
    public static void main(String[] args) {
    Scanner input=new Scanner(System.in);
    int m [][]=new int[2][5];
    for(int i=0;i<2;i++)
    {for(int j=0;j<5;j++)
            m[i][j]=input.nextInt();}
    int max_element=max(m);
    int min_element=min(m);
    System.out.println("max="+max_element);
    System.out.println("min="+min_element);
    }
```

```
static int max(int f[][]){
    int max=f[0][0];
    for(int i=0;i<2;i++)
        for(int j=0;j<5;j++)
            if (f[i][j]>max)  max=f[i][j];
    return max;}

 static int min(int f[][]){
    int min=f[0][0];
    for(int i=0;i<2;i++)
        for(int j=0;j<5;j++)
            if (f[i][j]<min)  min=f[i][j];
    return min;}
}
```

struct

Consider a data structure representing a person that includes a first name, last name, and birthday. The data structure look in various procedural language.

```
struct MEMBER
{
        String   FIRSTNAME;
        String    LASTNAME;
        int     BIRTHYEAR;
};
```

Java definitively has no structs, The equivalent in Java to a struct would be class :

```
Class MEMBER
{
        public   String      FIRSTNAME  ;
        public   String      LASTNAME  ;
        Public   int         BIRTHYEAR;
};
```

lecture 3: Linked list

Lecture content:

- Arrays disadvantages
- Linked list introduction
- **What is a linked list?**
- **Advantages of Linked List**
- **Disadvantages of Linked List**
- **Types of Linked List**
- **Linked List operations**

Arrays disadvantages:

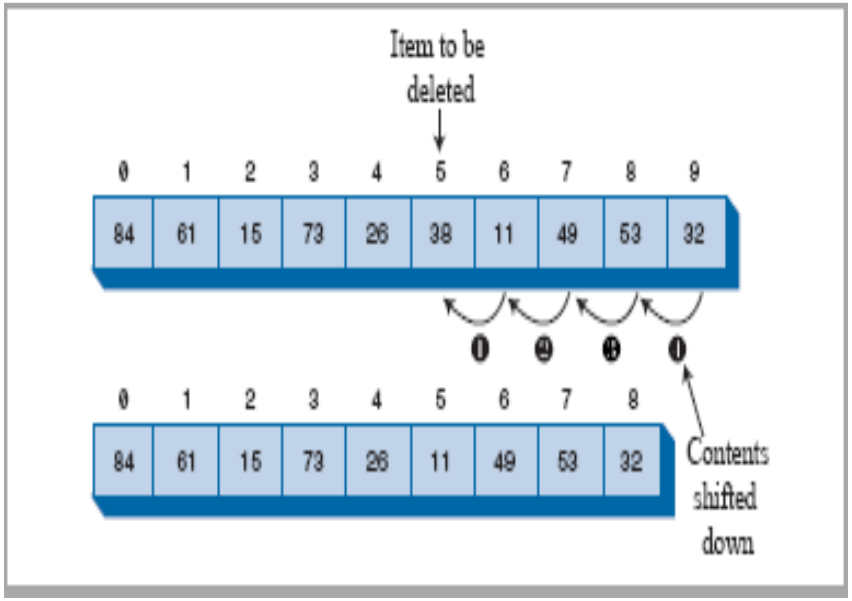
1) Memory Wastage.

   if we declare an array of size 10

   and store only 6 elements in it

   then the space of 4 elements

   are wasted



2) In an unordered array, searching is slow.

3) In an ordered array, insertion is slow.

4) In both kinds the size is fixed.

5) In both kinds of arrays, deletion is slow.

   In deletion algorithm is the assumption that holes

   are not allowed in the array , the occupied cells

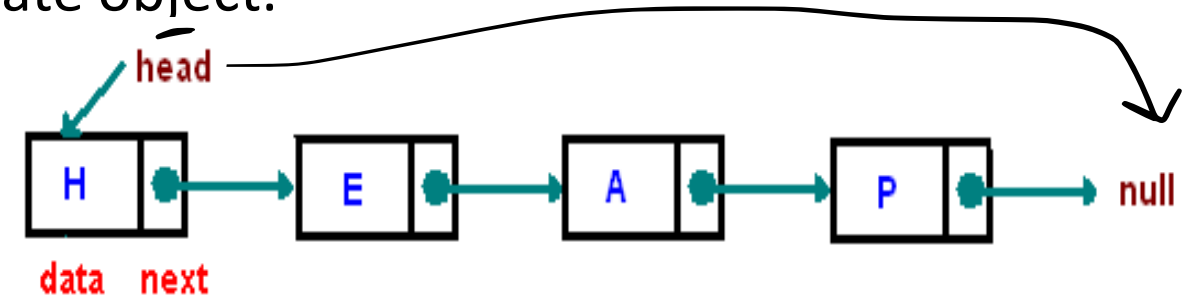
   must be arranged contiguously: **no holes allowed**.

Linked list introduction

- Linked list data storage structure that solves some of these problems
- Linked lists are probably the second most commonly used general-purpose storage structures after arrays.
- In fact, you can use a linked list in many cases in which you use an array, unless you need frequent random access to individual items using an index.
- Linked lists aren't the solution to all data storage problems, but they are conceptually simpler than some other popular structures such as trees.

What is a linked list?

A linked list is a linear data structure where each element is a separate object.



- ✓ Each element (we will call it a **node**) of a list is comprising of two items - the data and a reference to the next node.
- ✓ The last node has a reference to null.
- ✓ The entry point into a linked list is called the **head** of the list.
- ✓ It should be noted that head is not a separate node, but the reference to the first node.
- ✓ If the list is empty then the head is a null reference.

- ✓In a linked list, each data item is embedded in a link (node).
- ✓A *link (node)* is an object of a class called something like Linklist.
- ✓Each object contains a pointer to the next link in the list.

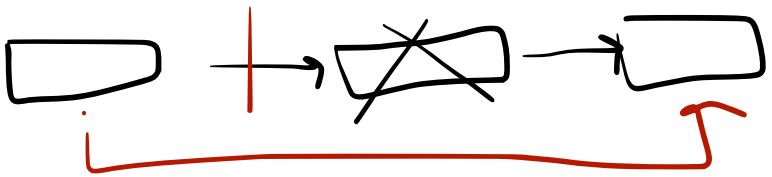
Advantages of Linked List:

1. **Dynamic Data Structure**

   •Linked list is a dynamic data structure so it can grow and shrink at runtime by allocating and de-allocating memory. So there is no need to give initial size of linked list لا تحتاج لحجز حجم من البداية.

2. **Easily Insertion and Deletion**

   Insertion and deletion of nodes are really easier. Unlike array here we don't have to shift elements after insertion or deletion of an element. In linked list we just have to update the address present in next pointer of a node.



3. **No Memory Wastage**

   •As size of linked list can increase or decrease at run time so there is no memory wastage

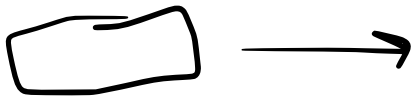
Disadvantages of Linked List:

1. **More Memory Usage**

   •More memory is required to store elements in linked list as compared to array. Because in linked list each node contains a pointer and it requires extra memory for itself.

2. **Traversal** اجتياز او وصول

   •Elements or nodes traversal is difficult in linked list. We can not randomly access any element as we do in array by index. For example if we want to access a node at position n then we have to traverse all the nodes before it. So, time required to access a node is large.

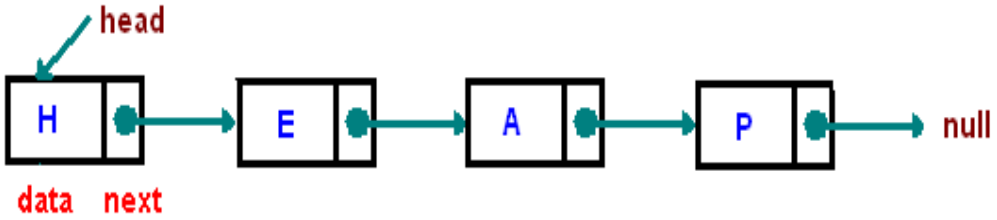


3. **Reverse Traversing**

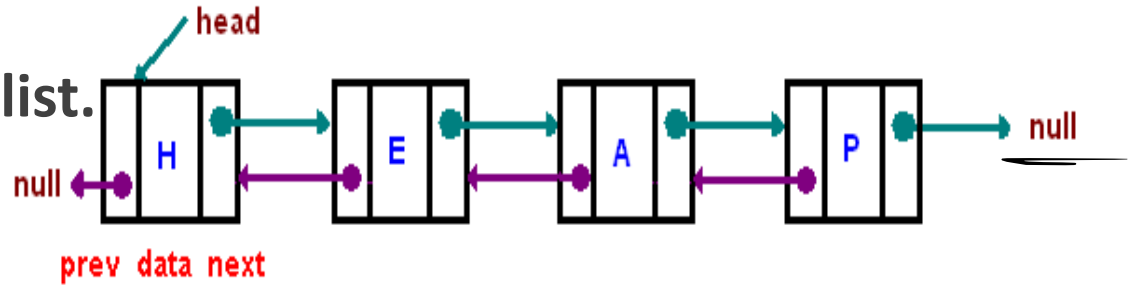
   •In linked list reverse traversing is really difficult. In case of doubly linked list its easier but extra memory is required for back pointer hence wastage of memory.

Types of Linked List:

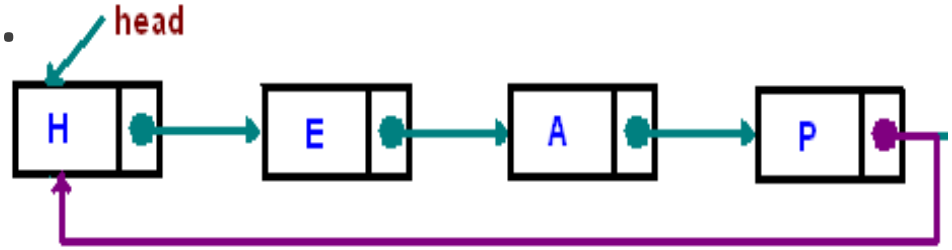
- **Single linked list.**



- **Double linked list.**



- **Circle linked list.**



Types of Linked List:

- **Single linked list.**

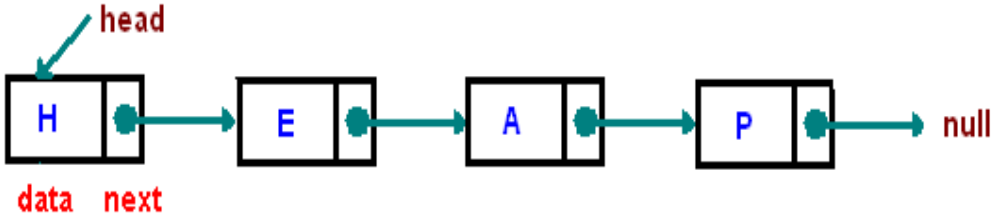
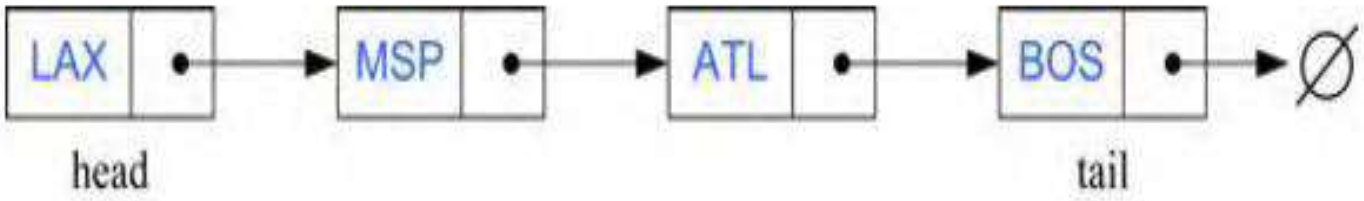


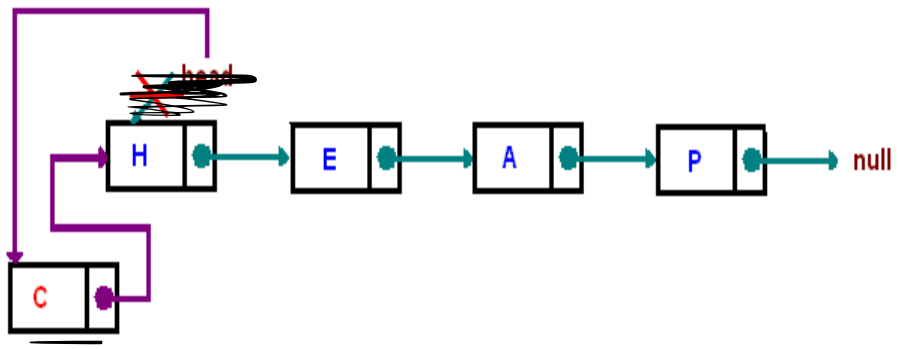
Figure 3.10: Example of a singly linked list whose elements are strings indicating airport codes. The `next` pointers of each node are shown as arrows. The **`null`** object is denoted as ∅.



Linked List operations:

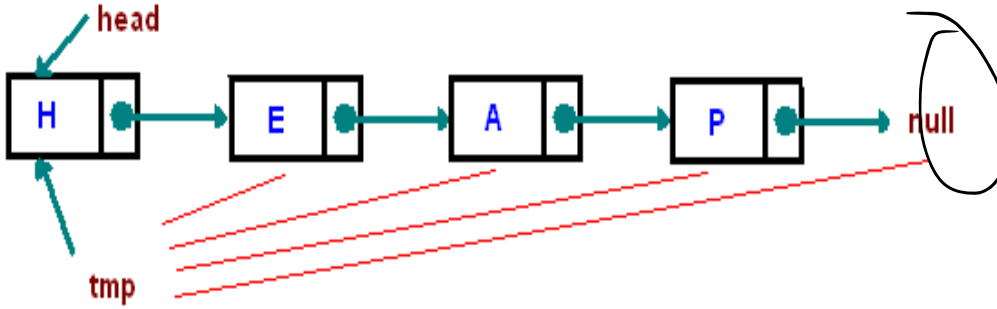
- **AddFirst.**

creates a node and prepends it at the beginning of the list



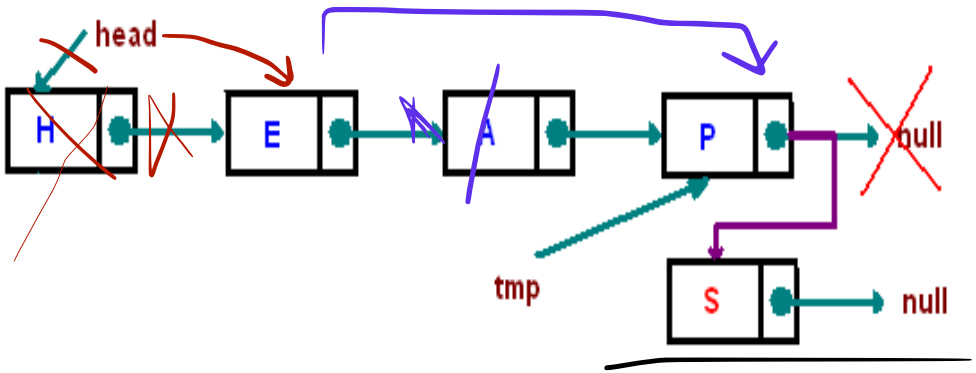
- **Traversing.**

Start with the head and access each node until you reach null. Do not change the head reference.

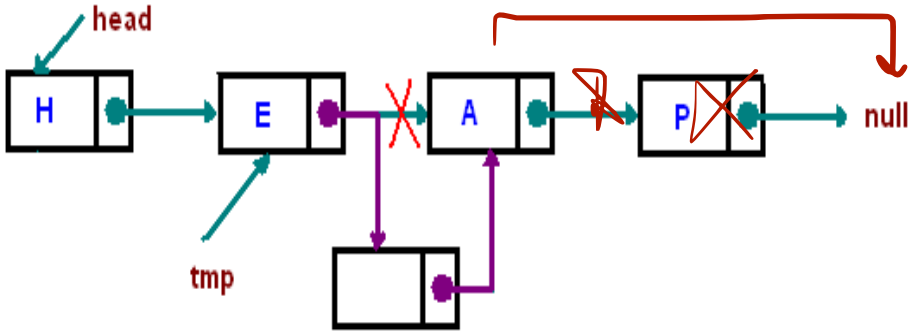


- **AddLast.**

  appends the node to the end



- **InsertAfter.**



- **RemoveFirst.**
- **RemoveLast.**
- **RemoveAt.**
- **……**

AddFirst:

```
Algorithm addFirst(v):
    v.setNext(head)          {make v point to the old head node}
    head ← v                 {make variable head point to new node}
    size ← size + 1          {increment the node count}
```

AddLast:

```
Algorithm addLast(v):
    v.setNext(null)          {make new node v point to null object}
    tail.setNext(v)          {make old tail node point to new node}
    tail ← v                 {make variable tail point to new node.}
    size ← size + 1          {increment the node count}
```

RemoveFirst:

```
Algorithm removeFirst():
    if head = null then
        Indicate an error: the list is empty.
    t ← head
    head ← head.getNext()          {make head point to next node (or null)}
    t.setNext(null)                {null out the next pointer of the removed node}
    size ← size − 1                {decrement the node count}
```

```
class Node
{
    private String elem;
    private Node next;
    public Node(String s , Node n)
    { elem=s; next=n; }
    public String getElem()
    { return elem; }
    public Node getNext()
    { return next; }
    public void setElem(String newElem)
    { elem=newElem; }
    public void setNext(Node newNext)
    { next=newNext; }
}
```

```
class SLink
{
    protected Node head;
    protected long size;
    public SLink()
    {    head=null; size=0;    }
    public void addFirst(Node
    v)
    {
        v.setNext(head);
        head=v;
        size++;
    }
```

```
public void deletefirst()

      {          String r=head.getElem();
head=head.getNext();
size =size-1;
 System.out.println(r);
}
```

```
public void addLast(Node v)
{
    Node tail=head;
    while(tail.getNext()!=null)
    {   tail=tail.getNext();   }
    tail.setNext(v);
    size++;
}
```

```
    public void displaylist()
    {
        Node cur=head;
        for(intk=0 ; k<size ; k++)
        {
            System.out.print(cur.getElem() + "
            ");
            cur=cur.getNext();
        }
        System.out.println();
    }
}
```

Main

```
SLink S1=new SLink();
Node N1=new Node("Heba",null);
S1.addFirst(N1);
S1.displaylist();
N1=new Node("Sara",null);
S1.addFirst(N1);
S1.displaylist();
N1=new Node("Ali",null);
S1.addFirst(N1);
S1.displaylist();
S1.deletefirst();
S1.displaylist();
```

```
run:
Heba
Sara Heba
Ali Sara Heba
Ali
Sara Heba
BUILD SUCCESSFUL
(total time: 1 second)
```

Home work

design java program that create a list of delegates of beauty company (delname , delno) the program contains function:

*Insertfirst function to insert new data at the beginning of the list.

*Insertlast function to insert new data at the last of the list.

*Display function to display the data of list.

```
package ls2;
class Node
{private String delname;  private String delno;  private Node next;
public Node(String name,String no,Node n)
{delname=name;delno=no; next=n;}
public String getname() {return delname;}
public String getno() {return delno;}
public Node getnext() {return next;}
public void setname(String newname)
{delname= newname;}
public void setno(String newno)
{delno= newno;}
public void setnext(Node newnext)
{next= newnext;} }
```

```
class SLink
{protected Node head;
protected long size;
public SLink()
{head=null;size=0;}

public void addfirst(Node v)
{v.setnext(head);
head=v;
size++;}

public void deletefirst()
 { String r=head.getname();
head=head.getnext();
size =size-1;
System.out.println(r);}
```

```
public void displaylist()
{Node cur=head;
for(int k=0;k<size;k++)
{System.out.print(cur.getname()+" " + cur.getno()+ " ");
cur=cur.getnext();}
System.out.println();}

public void addlast(Node v)        {
      Node tail=head;
      while (tail.getnext()!=null)
      {tail=tail.getnext();}
      tail.setnext(v);
      size++;}}
```

```
public class LS2 {
    public static void main(String[] args) {
        // TODO code application logic here
        SLink S1=new SLink();
        Node N1=new Node("sara","0553627744",null);
        S1.addfirst(N1);
        S1.displaylist();
          N1=new Node("samia","0542213200",null);
        S1.addfirst(N1);
    System.out.print("list: ");  S1.displaylist();
             N1=new Node("amira","0554344454",null);
      S1.addlast(N1);
           System.out.print("list after insert at last: ");
S1.displaylist();
}}
```

run:
sara 0553627744
list: samia 0542213200 sara 0553627744
list after insert at last: samia 0542213200 sara 0553627744
amira 0554344454
samia
sara 0553627744 amira 0554344454
BUILD SUCCESSFUL (total time: 1 second)

lecture 4: Double Linked list

Lecture content:

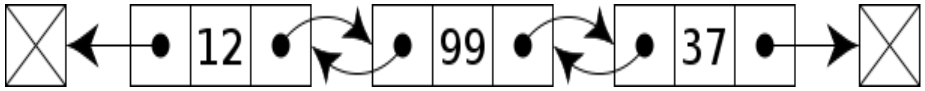
- Double Linked List introduction
- Advantages and Disadvantages of Doubly Linked List
- Double Linked List operations:
  - Add_First.
  - Add_After.
  - Add_Before.
  - Remove.
  - ……

Double Linked List introduction

- **a doubly linked list** is a linked data structure that consists of a set of sequentially linked records called nodes.

Each node contains two fields, called links, that are references to the previous and to the next node in the sequence of nodes.

The beginning and ending nodes' previous and next links, respectively, point to some kind of terminator, typically a sentinel node or null, to facilitate traversal of the list.



A doubly linked list whose nodes contain three fields :

- Data value,
- the link to the next node ,
- the link to the previous node

Advantages and Disadvantages of Doubly Linked List

Advantages:

1. We can traverse in both directions i.e. from starting to end and as well as from end to starting.
2. It is easy to reverse the linked list.
3. If we are at a node, then we can go to any node. But in linear linked list, it is not possible to reach the previous node.

Disadvantages:

1. It requires more space per space per node because one extra field is required for pointer to previous node.
2. Insertion and deletion take more time than linear linked list because more pointer operations are required than linear linked list.

Header and Trailer Sentinels

In order to avoid some special cases when operating near the boundaries of a doubly linked list, it helps to add special nodes at both ends of the list: a ***header*** node at the beginning of the list, and a ***trailer*** node at the end of the list. These "dummy" nodes are known as ***sentinels*** (or guards), and they do not store elements of the primary sequence. A doubly linked list with such sentinels is shown in Figure 3.19.

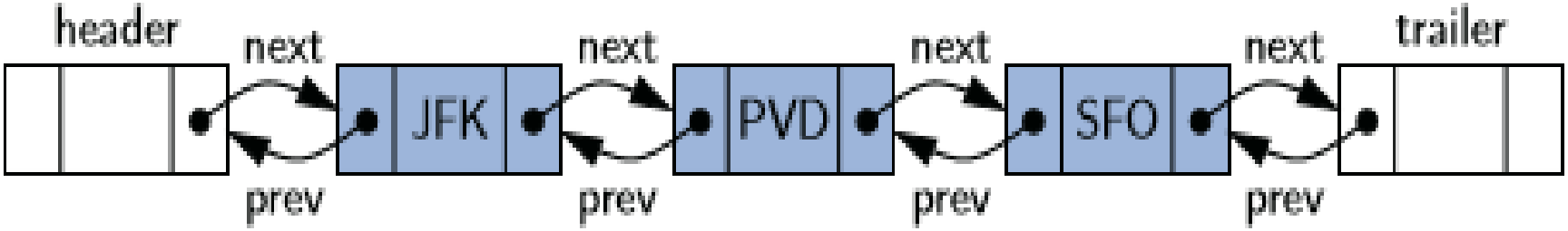
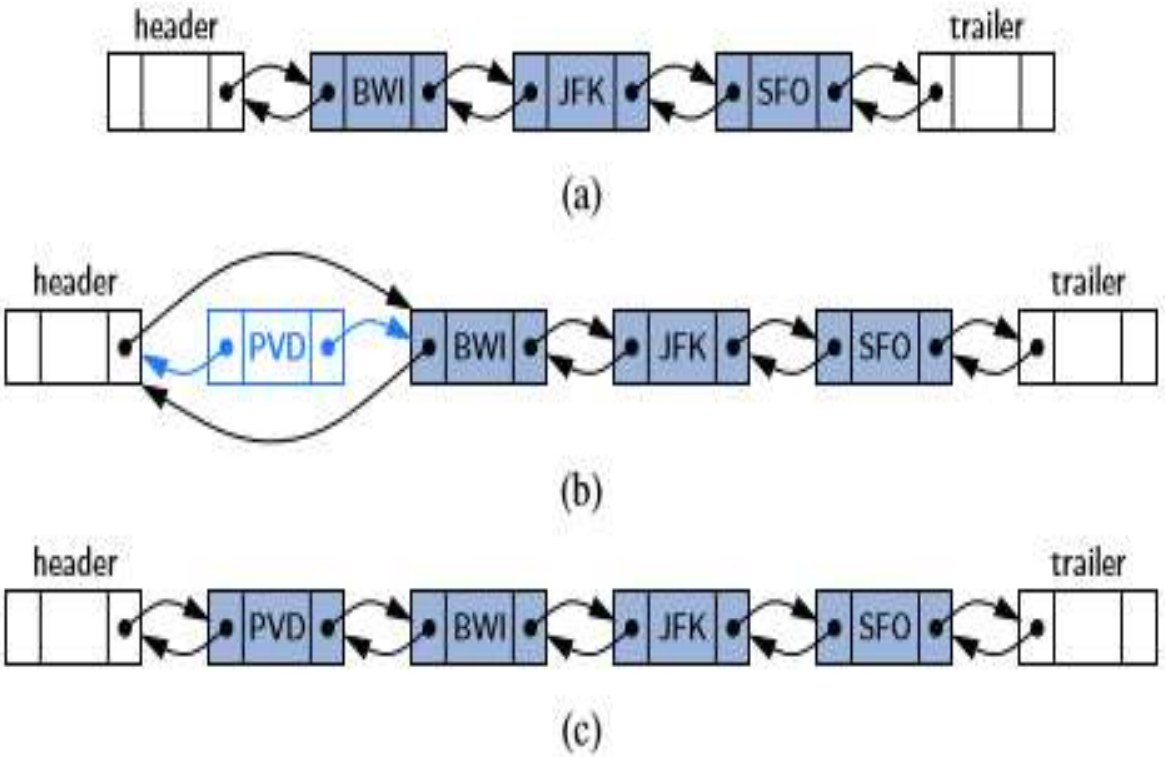
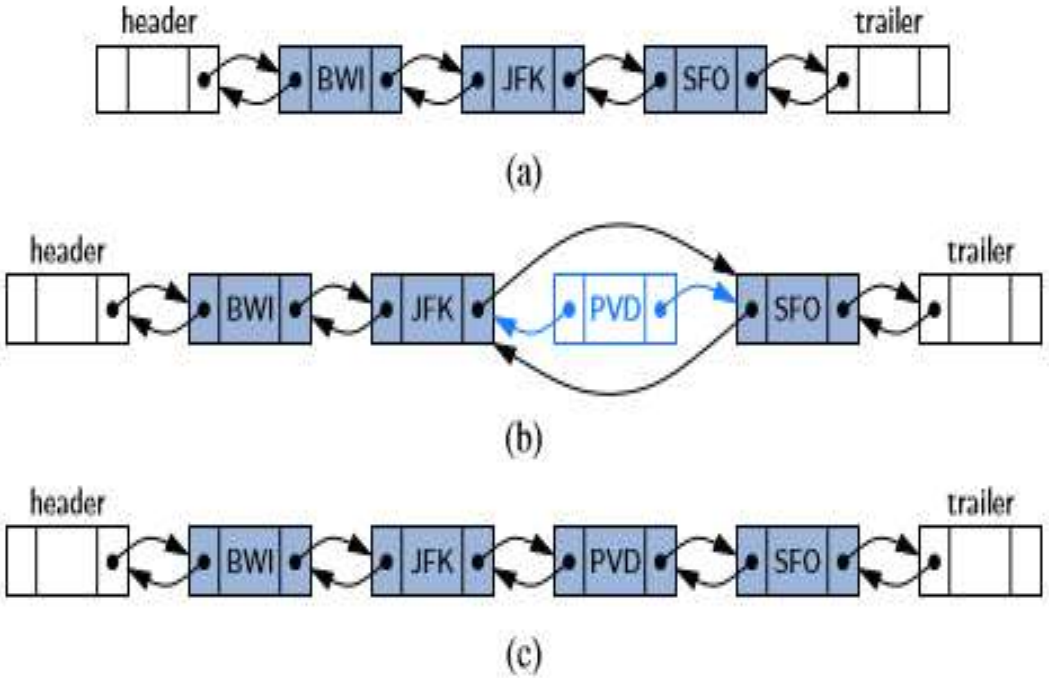


Figure 3.19: A doubly linked list representing the sequence { JFK, PVD, SFO }, using sentinels header and trailer to demarcate the ends of the list.

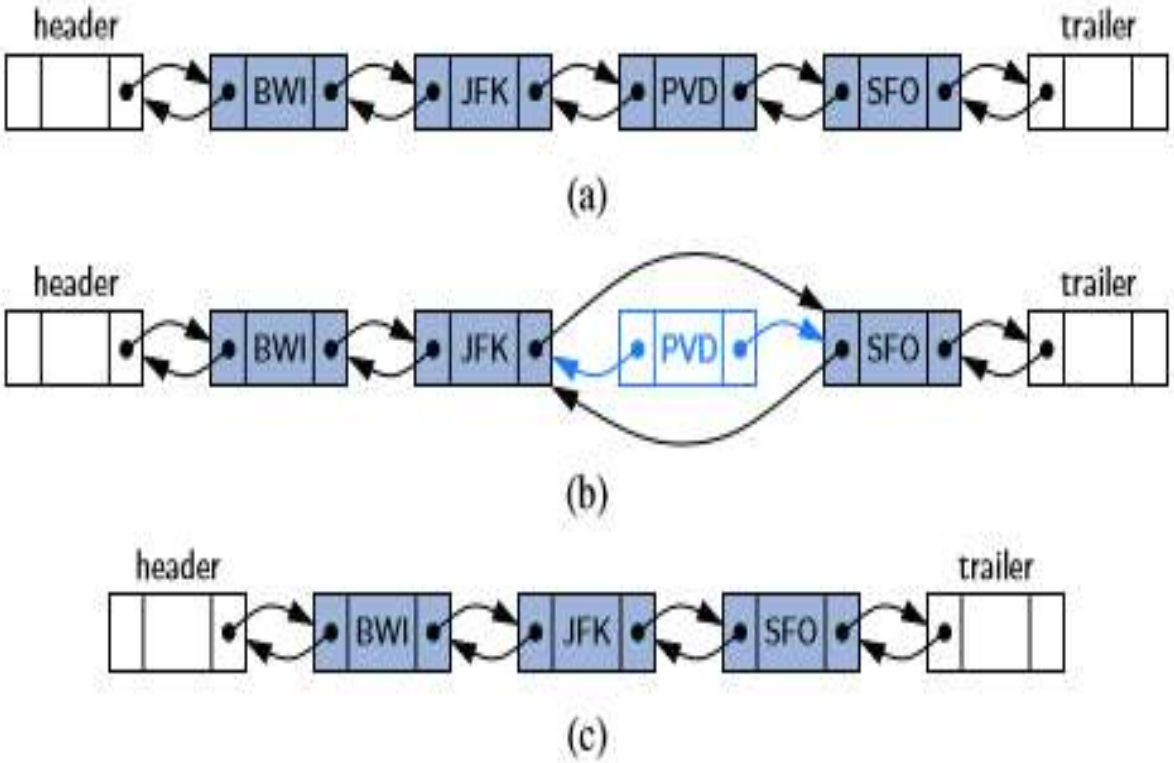
Inserting a node



Inserting a node



Deleting a node



Algorithm addFirst(v):

$w \leftarrow$ header.getNext() {first node}

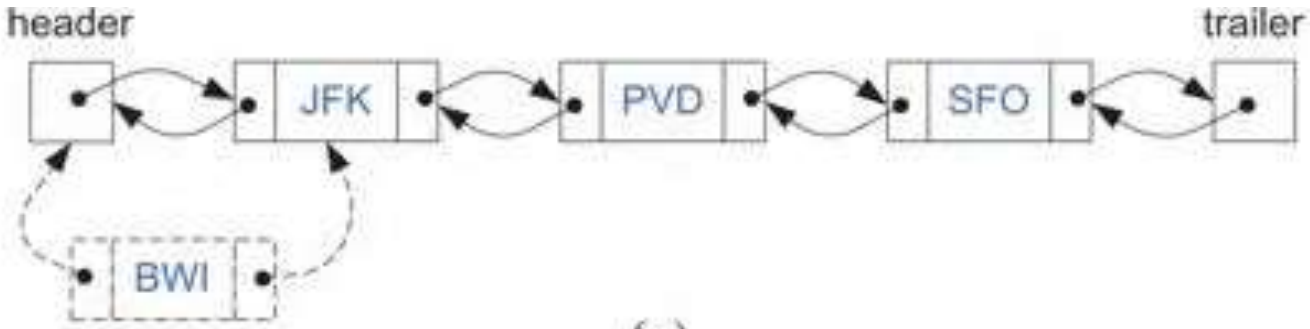
v.setNext(w)

v.setPrev(header)

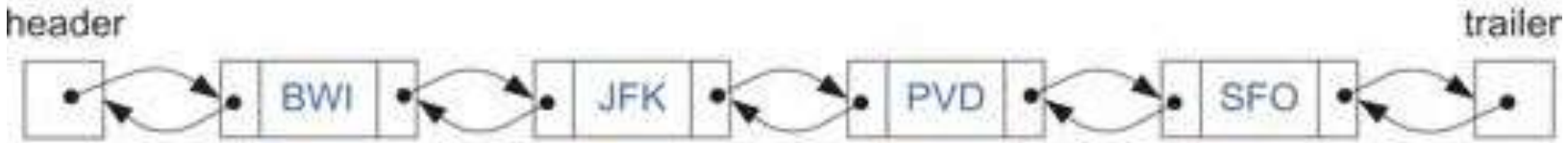
w.setPrev(v)

header.setNext(v)

size = size + 1



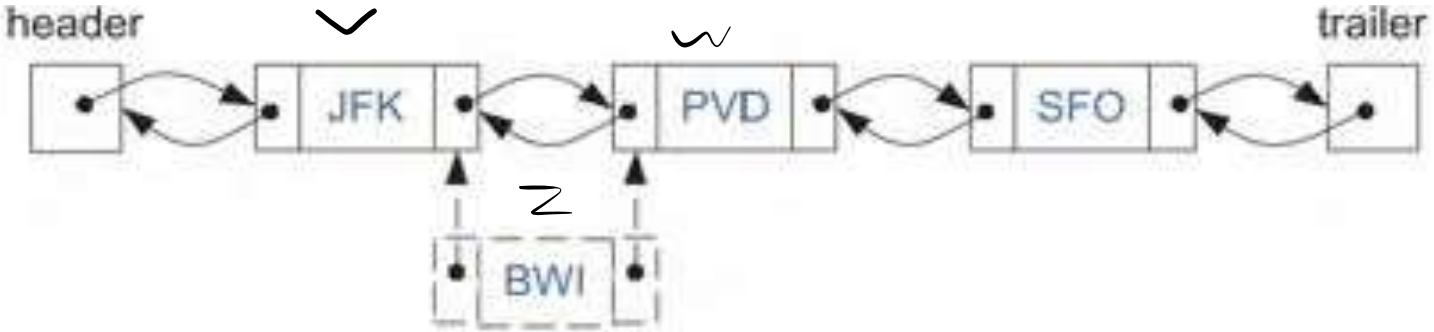
(a)



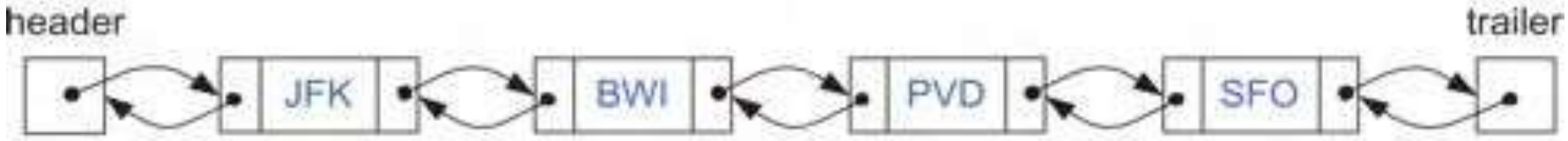
(b)

Algorithm addAfter(v,z):

$w \leftarrow v$.getNext() {node after v}
z.setPrev(v) {link z to its predecessor, v}
z.setNext(w) {link z to its successor, w}
w.setPrev(z) {link w to its new predecessor, z}
v.setNext(z) {link v to its new successor, z}
size $\leftarrow$ size + 1



(a)

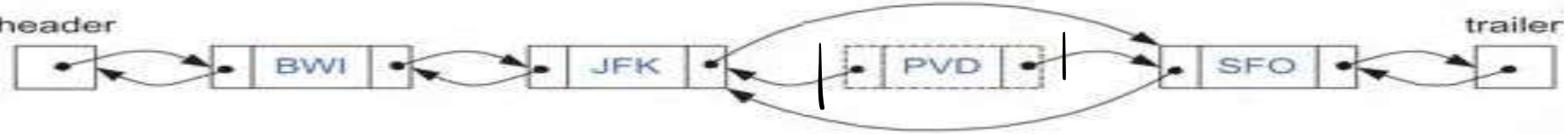


(b)

Algorithm remove(v):

$u \leftarrow v$.getPrev() {node before v}
$w \leftarrow v$.getNext() {node after v}
w.setPrev(u) {link out v}
u.setNext(w)
v.setPrev(**null**) {null out the fields of v}
v.setNext(**null**)
size $\leftarrow$ size $-$ 1 {decrement the node count}

(a)

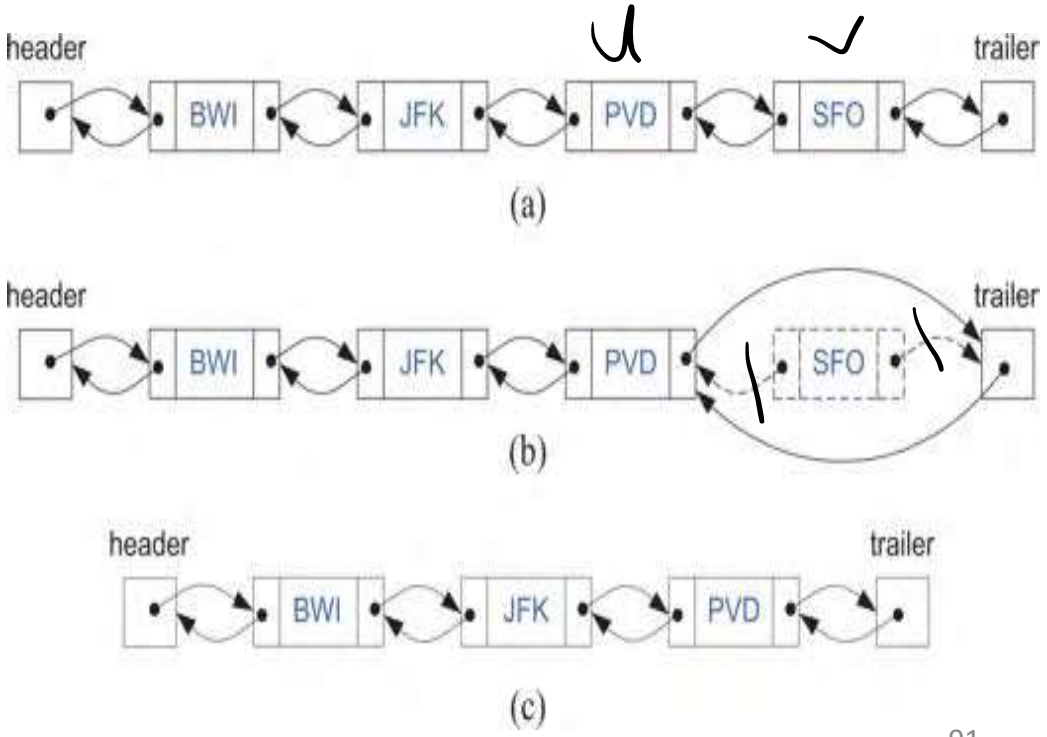


(b)

(c)

Algorithm removeLast():

```
if size = 0 then
    Indicate an error: the list is empty
v ← trailer.getPrev()          {last node}
u ← v.getPrev()                {node before the last node}
trailer.setPrev(u)
u.setNext(trailer)
v.setPrev(null)
v.setNext(null)
size = size − 1
```



```
class DNode
{
   public String name;
   public int id;
   public DNode next , prev;
   public DNode (String na ,int i,DNode nx , DNode p )
   {
      name=na;        id=i;
      next=nx;         prev=p;
   }}
```

```
class DList
{public int size; public DNode header , trailer;
    public DList()
    {size=0; header=new DNode(null,0,null,null);
        trailer=new DNode(null,0,header,null); header.next=trailer;
    }
public boolean isEmpty()
{   return size==0   ;}
public DNode getFirst() throws Exception
{if(isEmpty())    throw new Exception(“list is empty");
      else    return header.next;}
```

```
public DNode getLast() throws Exception
 {
     if(isEmpty()) throw new Exception(“list is empty");
     else return trailer.prev;
}


public DNode getprev(DNode v) throws Exception
{   if(v==header)
       throw new Exception(“can’t move back past the header of the list”);
   return v.prev ;}
```

```
public void addAfter(DNode v , DNode z) throws Exception
{
      DNode w=getnext(v);z.prev=v;z.next=w;
      w.prev=z; v.next=z ;size++;}

public void addBefore(DNode v , DNode z) throws Exception
{
     DNode u=getprev(v);
     z.prev=u; z.next=v; v.prev=z;
     v.next=z; size++ ;}
```

```
public DNode getnext(DNode v) throws Exception
{
    if(v==trailer)
throw new Exception(" can’t move forward past the trailer of the list ");
    return v.next ;
}

public void addFirst(DNode v) throws Exception
{      addAfter(header,v);   }

public void addLast(DNode v) throws Exception
{       addBefore(trailer,v);}
```

```
public void remove(DNode v) throws Exception
{
        DNode u=getprev(v);
        DNode w=getnext(v);
        w.prev=u;
        u.next=w;
        v.prev=null;
        v.next=null;
        size--;
}
```

```
   public void display()
   {DNode c=header.next;
         for(int i=0 ; i<size ; i++)
         {System.out.println(c.id + "   " + c.name ); c=c.next;
         }}}
public class JavaApplication11
{public static void main(String[] args) throws Exception
   {DList d1=new Dlist();
      DNode n1=new DNode("reham",1,null,null);
      d1.addFirst(n1);d1.display();
      }}
```

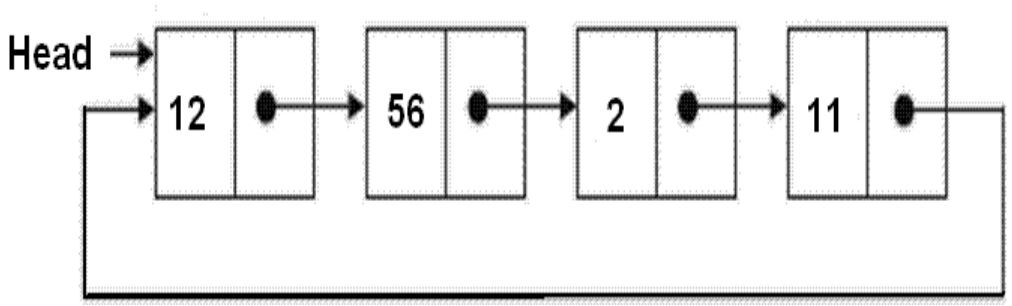
lecture 5: Circle Linked List

Lecture content:

- Circle Linked List introduction
- Circle Linked List category
- Circular Linked List Applications
- Advantages and Dis-Advantages of Circular Linked Lists
- Designing and Implementing a Circularly Linked List
- Operations on a Circularly Linked List

Introduction to Circular Linked List

- Circular linked list is a linked list where all nodes are connected to form a circle.
- A circular linked list can be a singly circular linked list or doubly circular linked list.



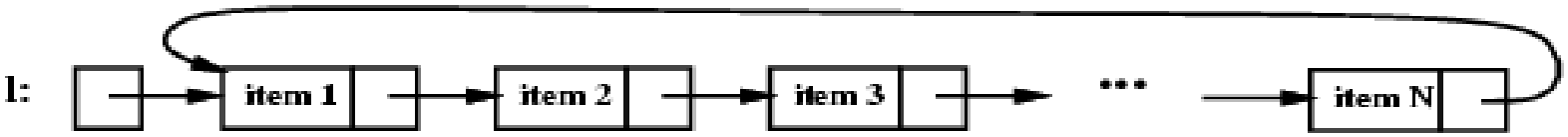
Circle Linked List category :

- Circle Linked List category :
- 1. Singly Linked List 2. Doubly Circular Linked List

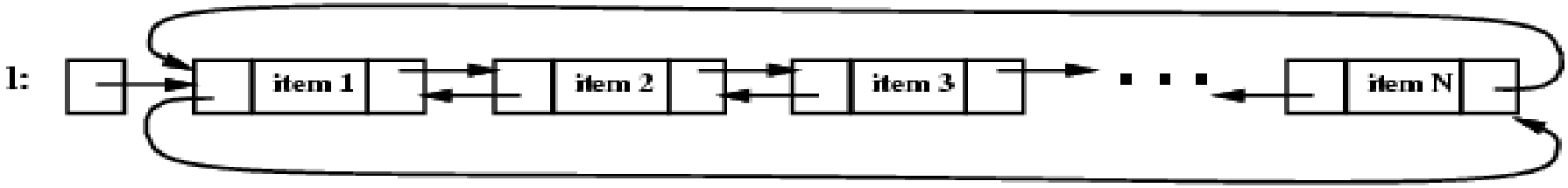
➢ In case of Singly Linked List, the last node points to the first.

➢ In case of Doubly Circular Linked List, the last node points to the first and the first node points back to the last node.

Circular, singly linked list:



Circular, doubly linked list:



Circular Linked List Applications

- **Circular Linked Lists are useful for playing videos and sound files in looping mode.**
- **There are also steeping stone to implementing graphs.**

Advantages of Circular Linked Lists:

1. Any node can be a starting point:

We can traverse the whole list by starting from any point. We just need to stop when we have gone through the whole list i.e. the first visited node is visited again.

2. Useful for implementation of queue:

Unlike the usual implementation that needs a front and a rear pointer, we don’t need to maintain two pointers if we use circular linked list. We can maintain a pointer to the last inserted node which will always contain the address to the first node.

Circular lists are useful in applications to repeatedly go around the list, such as, a set of processes that should be time-shared.

For example:

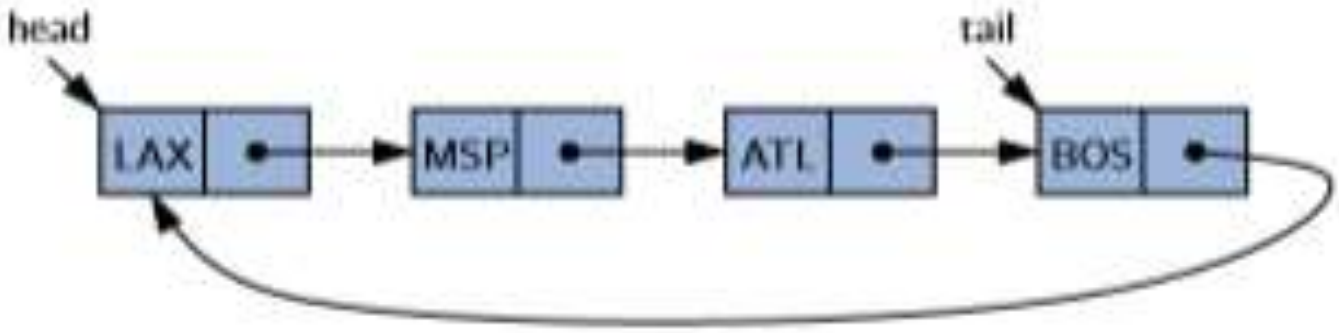
CPU Scheduling: When multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application.

Dis-Advantages of Circular Linked Lists:

- Possibility of an infinite loop.

Designing and Implementing a Circularly Linked List

Is a singularly linked list which the next reference of the tail node is set to refer back to the head of the list (rather than null).

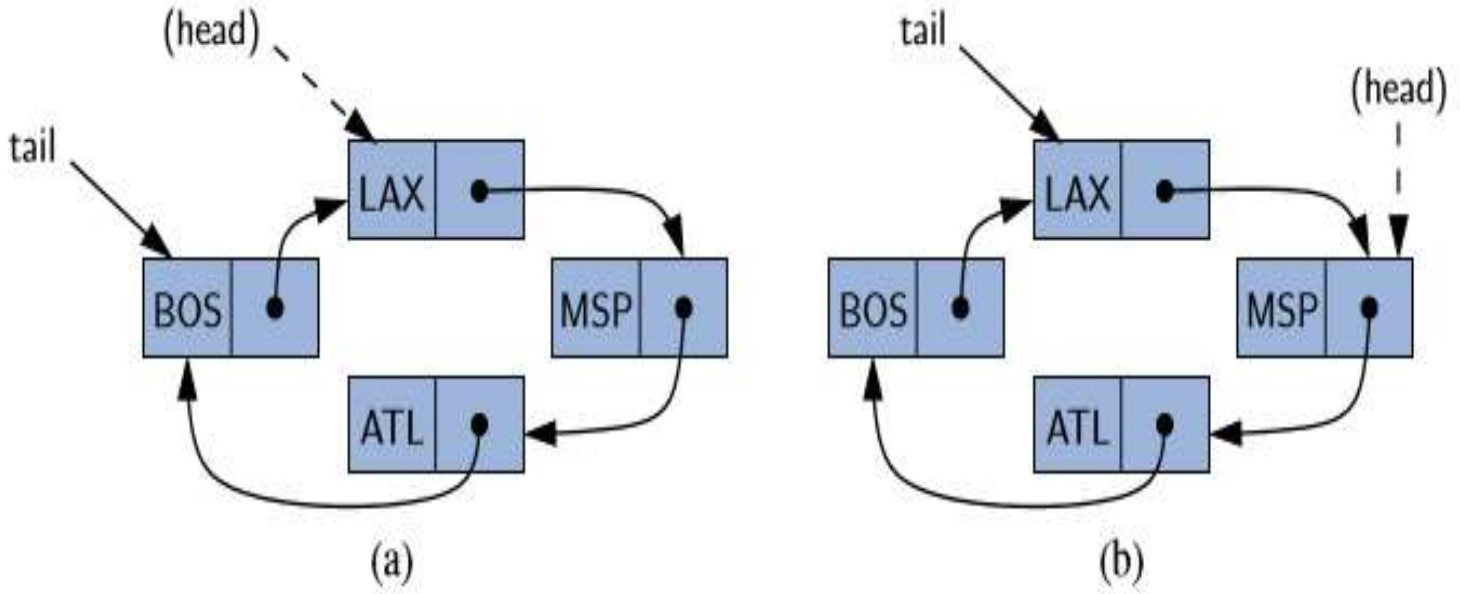


-head is tail.getNext()

Operations on a Circularly Linked List

- The operation rotate():

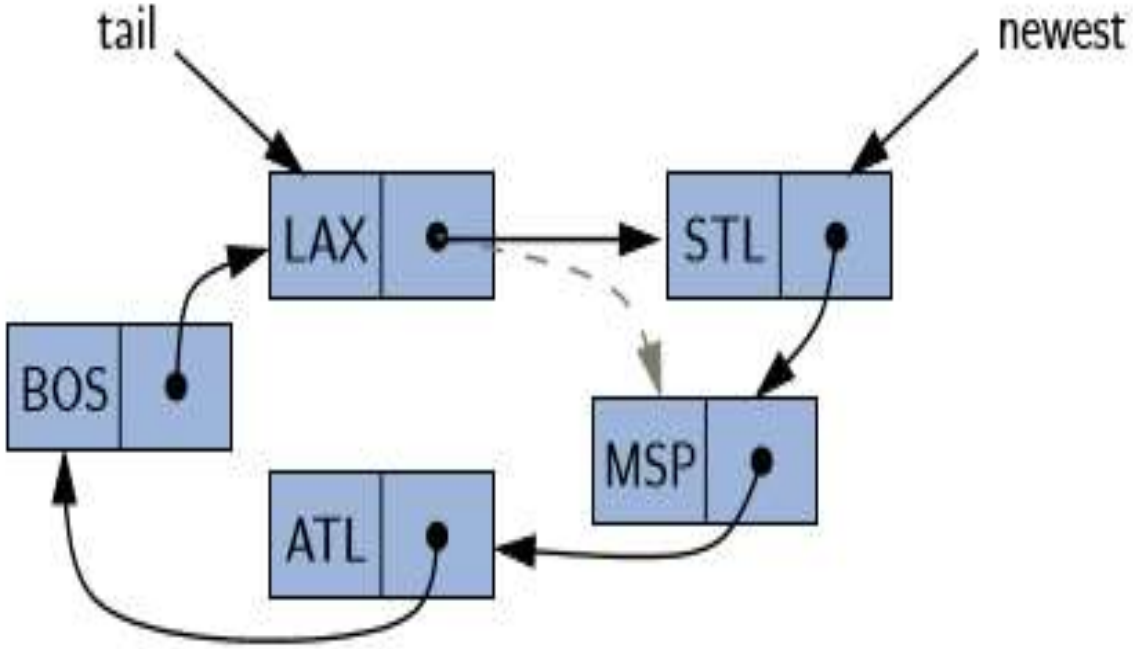
Moves the first element to the end of the list.



before the rotation, representing sequence {LAX, MSP, ATL, BOS};

after the rotation, representing sequence {MSP, ATL, BOS, LAX}.

- We can add a new element at the **front** of the list by creating a new node and linking it just after the tail of the list.



- STL is the first element of the list

```
package circle;
class Node
{       public String elem;      public Node next;
        public Node(String s , Node n)
        {   elem=s;   next=n;   }
 }
class CLinkedList
{
   public Node tail = null;
   public int size = 0;
   public CLinkedList()
   {}

   public boolean isEmpty()
   { return size == 0;}
```

```
public String first()
   {
      if (isEmpty())   return null;  return tail.next.elem;
   }

public String last()
   {        if (isEmpty()) return null;  return tail.elem;     }
```

Rotate method (update):

rotate the first element to the back of the list.

```
public void rotate()
   {    // rotate the first element to the back of the list
if (tail != null)      // if empty, do nothing
tail = tail.next; } // the old head becomes the new tail


public void addFirst(String e)
 {  if (size == 0)
{  tail = new Node(e, null);        tail.next=tail; }
                // link to itself circularly
 else {  Node newest = new Node(e, tail.next);
          tail.next=newest;        }        size++;     }
```

```
public void addLast(String e)
   {
      addFirst(e);                  // insert new element at front of list
      tail = tail.next;             // now new element becomes the tail
   }


public String removeFirst()
   {      if (isEmpty())            return null;
      Node head = tail.next;
 if (head == tail)                    // أي يوجد عقدة واحدة
 tail = null;                  // must be the only node left
      else    tail=head.next;     // removes ”head” from the list
      size=size-1;        return head.elem;    }
```

```
  public void display()
    {       Node c=tail.next;
        for( int k=0 ; k<size ; k++)       {
System.out.print(c.elem +  "    "); c=c.next;       }
        System.out.println();    }}

public class circle
{    public static void main(String[] args)
    {       CLinkedList C1 = new  CLinkedList();
   C1.addFirst("nora1"); C1.addFirst("nora2");
C1.addFirst("nora3"); C1.addFirst("nora4");
C1.display();    }  }
```