University of Kelaniya
Sri Lanka

# COSC 21063

## Lists

Lecturer : Ms.D.M.L.Maheshika Dissanayake

## Structure of Lesson

- Lists
  - Definition
  - Operations
  - Implementation

    - Contiguous implementation

    - Linked Implementation

## Learning Outcome:

By the end of this lesson you should be able to:

- Understand the concept of List
- Create the data structure List and define the necessary operations
- Implement the List using the contiguous and linked representation
- Solve simple problems using the List concept

## Lists

- A Linear List is an ordered linear sequence of information nodes, which may grow or shrink regulation throughout its lifetime. It is therefore an inherently dynamic data structure.
- e.g. a shopping list or book reference list,

## Lists

A list is a collection of set of elements with the following properties:

- In a list there is a linear order (called `followed by', or `next') defined on the elements: every element (except for one, called the *last* element ) is followed by one other element, and no two elements are followed by the same element. There is exactly one element in a list (the *first* element) that does not follow after any element.
- It is possible to access individual elements of a list. Insertion, deletion and retrieval may occur at any point of the list.

## Operations on a List

### CreateList

Inputs:  None

Output: L (a List)

Preconditions: None

Postconditions: L is created and is empty.

## ClearList

Input: L, a list

Output: L'

Preconditions: L is defined(i.e. L is created)

Postconditions: All entries in the list have been removed:
list is empty.

## IsListEmpty

Input: L

Outputs: IsListEmpty true or false

Preconditions: L is defined.

Postconditions: true if L is empty, false otherwise

## ListSize

Input: L

Output: S (non-negative integer)

Preconditions: L is defined

Postconditions: S = the number of elements in L

## InsertLast

Input: L and V (value/element/entry)

Output: L'

Preconditions: L is created and not full. V is an appropriate type for an element of L.

Postconditions: L' = L with V added as the last element of L. Storage is allocated as needed. If V already occurs in L, L' contains an additional copy of V.

## InsertList

Input: L , V (element) and P the position V to be inserted.

Output: L'

Preconditions: L is created and not full. V is an appropriate type for an element of L and $0 <= P < N$, where N is the number of entries in L

Postconditions: L' = L with V inserted at position P. The entry formerly in position P (provided $P < N$) and all later entries have their position numbers increased by one.

## DeleteList

Input: L and P the position of the element (say V) to be deleted.

Output: L' and V the deleted element

Preconditions: L is created and not empty and $0 <= P < N$, where N is the number of entries in L.

Postconditions: The element in position P of L has been returned as V and deleted from L. the entries in all later positions (provided $P < N-1$) have their position numbers decreased by one.

## RetrieveList

Input: L and P the position of the element (say V) to be retrieved.

Output: V the retrieved element

Preconditions: L is created and not empty and $0 <= P < N$, where N is the number of entries in L.

Postconditions: The element in position P of L has been returned as V and L remains unchanged.

## ReplaceList

Input: L , V the element, and P the position at which V is replaced.

Output: L’

Preconditions: L is created and not empty V is an appropriate type for an element of L and 0 <= P < N, where N is the number of entries in L.

Postconditions: The element in position P of L has been replaced by V. The other entries of remain unchanged. L’ = L after replacing V at position P.

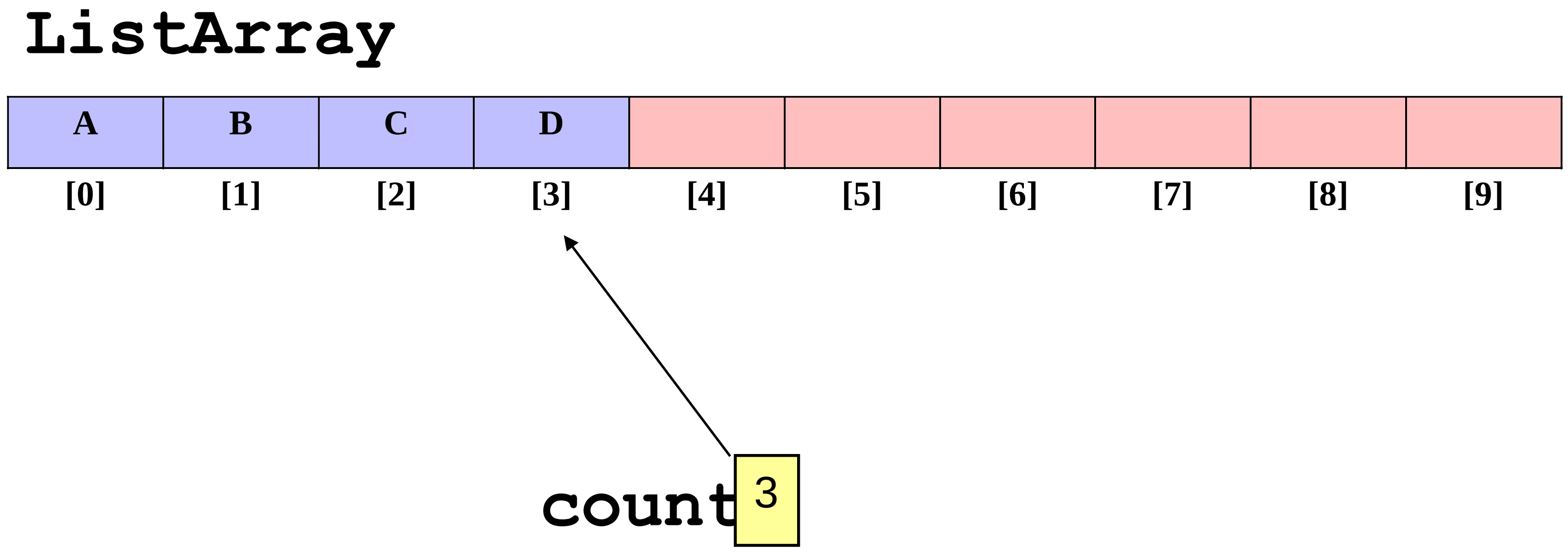
## Contiguous Implementation

- Define a List in java
- Identify the information needed.
  - An array to hold the elements of the List.
  - A variable count to point to the last element of the list. This variable can also be used to identify the number of elements in the List.

## CreateList

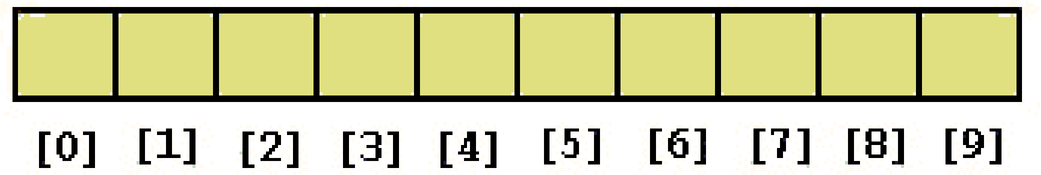
```
    public class List {
    private int maxSize ;
    private int position;
    private int[] ListEntry;

    List(int size)
    {
        maxSize = size;
        ListEntry = new int[maxSize];
        position= -1;
    }
}
```

# An Array-Based List

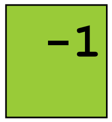


# Initial Picture – Empty LIST

**Name: ListArray**
**Type: Array**



**Name: count**
**Type: Integer**



## IsListEmpty

```
boolean IsListEmpty()
 {
   return (position==-1);
 }
```

## IsListFull

```
boolean IsListFull()
 {
     return (position== maxSize-1);
  }
```

## ListSize

```
int ListSize()
{
        return (position+1);
}
```

## InsertLast

```
void InsertLast(int x)
  {
     if (IsListFull())
       System.out.println("Attempt to insert at the end of a full list");
     else
        ListEntry[++position] = x;
  }
```

## InsertList

```
void InsertList(int p,  int element)
 {     int i;
        if (IsListFull())
           System.out.println("Attempt to insert an entry into a full list");
        else if (p < 0 || p > ListSize())
           System.out.println("attempt to insert a position not in the list");
        else
        {
           for( i = ListSize(); i >p; i--)
              ListEntry[i] = ListEntry[i-1];
           ListEntry[p] = element;
           position++;
        }
    }
```

- If we insert an entry at the end of the list, then the function executes only a small, constant number of commands.
- If, on the other extreme, we insert an entry at the beginning of the list then the function must move every entry in the list to make room, so if the list is long, it will do much more work.
- In the average case, where we assume that all possible insertions are equally likely, the function will move about half the entries of the list.
- Thus we say that the amount of work that this function does is approximately proportional to n, the length of the list.

## DeleteList

```
int DeleteList( int  p) {
        int i,element;
       if (IsListEmpty())
           System.out.println("Attempt to delete an entry from an empty list");
       else if (p < 0 || p >= ListSize())
           System.out.println("attempt to delete a position not in the list");
       else {
           element = ListEntry[p];
           for( i = p; i < ListSize()-1; i++)
               ListEntry[i] = ListEntry[i+1];
           position--;
           return element;
           }
      return 0;  }
```

## RetrieveList

```
int  RetrieveList(int p ){
 int i,element;
 if (IsListEmpty()){
    System.out.println("Attempt to retrieve an entry from an empty list");
    return 0;}
else if (p < 0 || p >= ListSize()){
    System.out.println("attempt to retrieve an entry at a position not in the list");
    return 0; }
 else{
   element = ListEntry[p];
   return element;}
 }
```

# ReplaceList

```
void ReplaceList (int p,   int   x){
   int i;
   if (IsListEmpty())
       System.out.println("Attempt to replace an entry of an empty list");
   else if (p < 0 || p >= ListSize())
       System.out.println("attempt to replace an entry at a position not in the list");
   else
       ListEntry[p] = x;
}
```

## TraverselList

```
void TraverselList()
  {
      int i;
      for (i=0; i<position+1; i++)
          System.out.println(ListEntry[i]);
  }
```

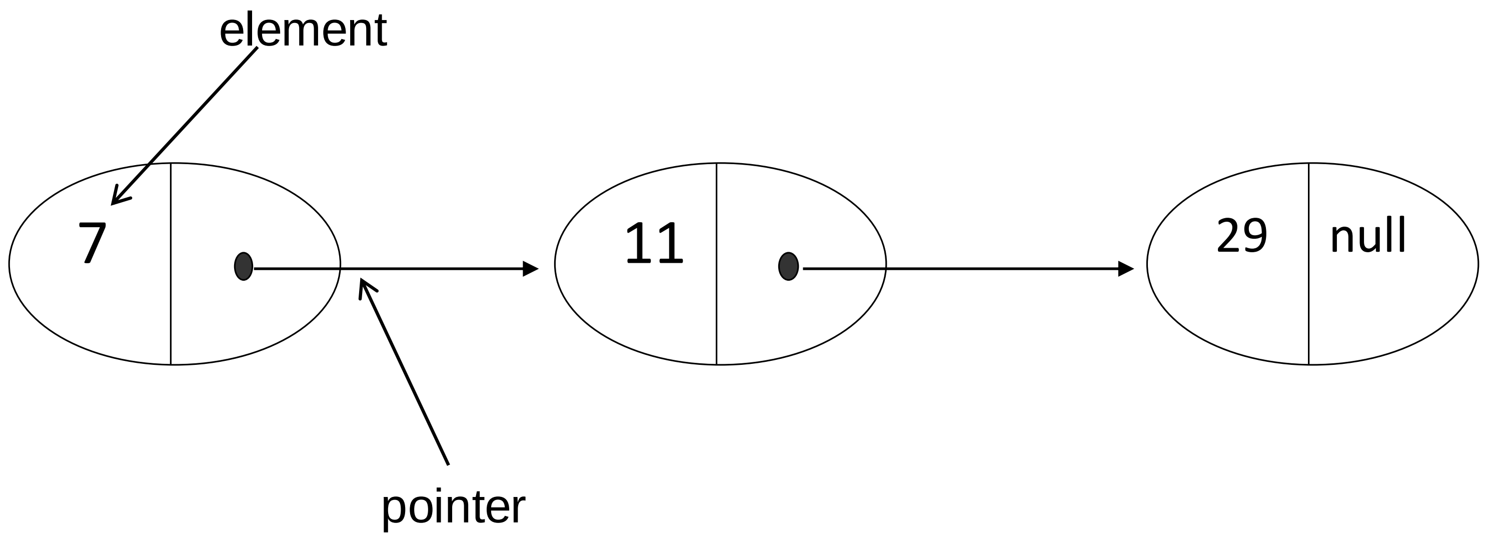
## Linked Implementation

- Consider the list of integers 29,7,11.
- We need one memory cell or node for each element in the list;

  In this memory cell we need to store two things:

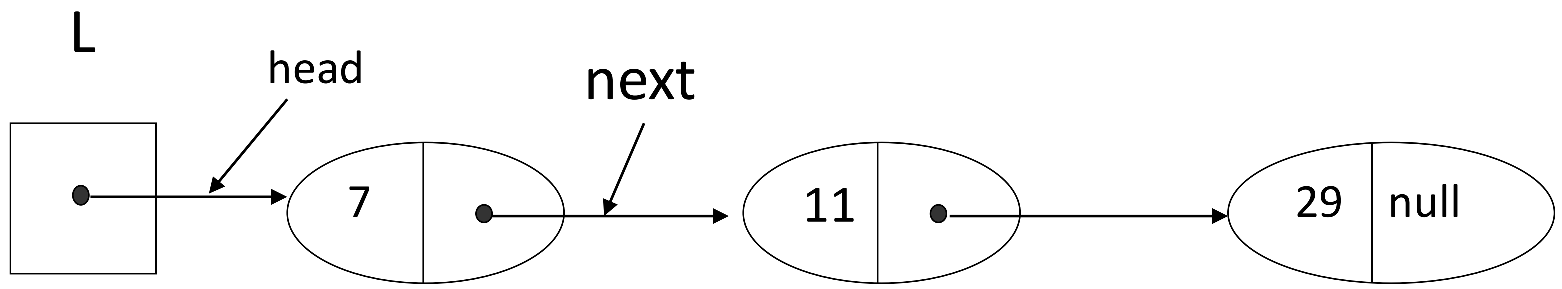
  - The value of the element, which is an integer in this example but could be anything
  - A pointer to the memory cell of the next element.

## Linked Implementation Contd…



## Linked Implementation Contd…

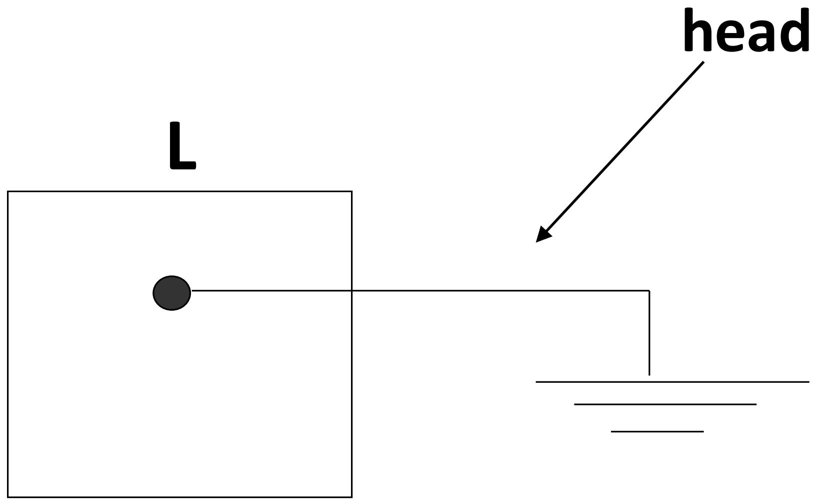
### Where is the List?



## Linked Implementation Contd…

- L is probably a different type than the ‘elements’ in the list, because it has no associated value,
- it has a pointer to point to the beginning of the list.
- If required we could have any additional information of the list. For example we may have the number of elements in L.

## How will we represent an empty List?



## Linked Implementation Contd…

- Define the structure of a cell or node.
- Identify the information needed for the node
  - The value of the element (in this example an integer, but could be anything)
  - A pointer to the next node

## Linked Implementation Contd…

```
public class Node
{
    int data;
    Node next;
}
```

## Define the List

The information for the List is:

- A pointer head to point to the beginning of the list
- A variable count indicating the number of elements of the List.

```
public class List
 {
    private Node head;
    private int count;
    public List()
    {
        head= null;
        count= 0;
    }
 }
```

## IsListEmpty

```
boolean  IsListEmpty()
{
   return (head == null);
}
```

## ListSize

```
int ListSize ()
  {
     return (count);
  }
```

## InsertLast

```
void InsertLast(int data)
  {
      Node node =new Node();
      node.entry =data;
      node.next =null;
      if (head == null)
      {
         head =node;
      }
      else
      {
       Node n =head;
```

## InsertLast Contd…

```
while (n.next!= null) {
            n=n.next;
          }
         n.next = node;
      }
      count++;

   }
```

## InsertList

```
public void InsertList(int p,int data)
  {
     Node node = new Node();
     node.entry = data;
     node.next = null;

     if(p < 0 || p > ListSize())
     {
        System.out.println("attempt to insert a position not in the list");
     }
```

## InsertList

```
else{
        Node n = head;
        for(int i=0;i<p-1;i++)
        {
           n = n.next;
        }
        node.next = n.next;
        n.next = node;
        count++;
     }

  }
```

## DeleteList

```
public void DeleteList(int p){
    if (IsListEmpty()) {
        System.out.printf("\ List is empty\n");
       }
    else {
        if(p < 0 || p > ListSize())
        {
            System.out.println("attempt to insert a position not in the list");
        }
        else if (p == 0)
        {
            head = head.next;
        }
```

## DeleteList Contd..

```
else {
            Node n = head;
            Node n1 = null;
            for (int i = 0; i < p - 1; i++) {
               n = n.next;
            }
            n1 = n.next;
            n.next = n1.next;
            n1 = null;
         }
count--;
      }
   }
```

## TraveseList

```
 public void TravelList()
   {
     Node node =head;
       while (node.next!= null) {
           System.out.println(node.entry);
           node = node.next;
       }
System.out.println(node.entry);

}
```

## Summary of main points

- Definition of a List
- Implementation of the List operation