

# Data Structures & Algorithms (COMP2113)

## Lecture # 17 Basic Data Structures | Part 01 Lists



Dr. Aftab Akram (PhD Computer Science, China)

Lecturer, Division of Science & Technology

University of Education, Lahore



# List – as Data Structure

- A list is a finite, ordered sequence of data items known as elements.
- The most important concept related to lists is that of position, i.e., there is a first element in the list, a second element, and so on.
- Each list element has a data type.
- In the simplest form, all list elements have same data type.
- The operations defined as part of the list ADT do not depend on the elemental data type.



# List – as Data Structure

- A list is said to be **empty** when it contains no elements.
- The number of elements currently stored is called the **length** of the list.
- The beginning of the list is called the **head**, the end of the list is called the **tail**.
- There might or might not be some relationship between the value of an element and its position in the list.
- For example, sorted lists have their elements positioned in ascending order of value, while unsorted lists have no particular relationship between element values and positions.



# List – as Data Structure

- if there are  $n$  elements in the list, they are given positions 0 through  $n - 1$  as  $\langle \underline{a_0}, \underline{a_1}, \underline{a_2}, \dots, \underline{a_{n-1}} \rangle$ .
- The subscript indicates an element's position within the list.
- Using this notation, the empty list would appear as  $\langle \text{ } \rangle$ .



# List -- Operations

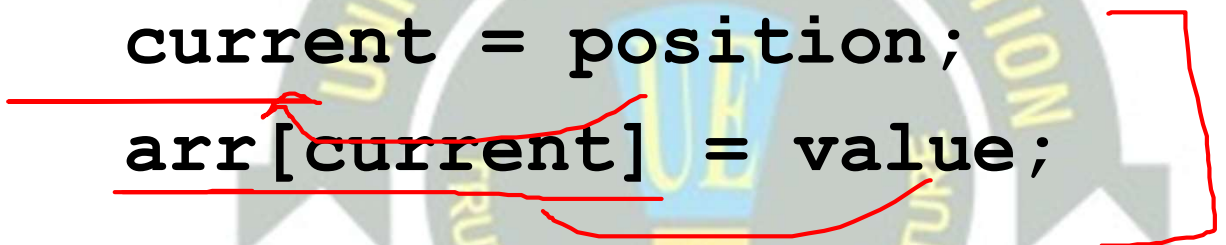
- Following Operations are defined for List as an ADT:
  - **insert** – inserts an elements at the current position.
  - **append** – insert an element at the end of list.
  - **remove** – remove and return current element.
  - **movToPos** – moves current to a position.
  - **moveToStart** – sets the current position to first element.
  - **moveToEnd** – sets the current position to last element.
  - **next** – move the current position to next element.
  - **prev** – move the current position to previous element.
  - **clear** – clears all elements in the list.
  - **getValue** – returns a pointer to current element.
  - **length** – returns the length of the list.
  - **currentPos** – returns the position of current element.
  - **find** – returns true if an element is present in the list.

20



# Pseudocode

```
function insert(value, position) {  
    current = position;  
    arr[current] = value;  
}
```



# Pseudocode

```
function append(value) {  
    current = last; n-1  
    arr[current] = value;  
}
```





# Pseudocode

```
function remove(position) {  
    current = position;  
    arr[current] = 0;  
}
```





# Pseudocode

```
function clear(array, n) {  
    for (int i=0; i<n; i++)  
        array[i]=0;  
}
```

$i \leftarrow n-1$



# Array-Based List Implementation

- Lists can be implemented using arrays or linked lists.
- Array implementation of lists is simpler.
- However, size of lists are fixed.
- Array is collection of elements stored in contiguous memory locations.
- Each element of the array is of same type. *int*
- Array elements can be accessed using element indexes which start at 0 and last index is  $n - 1$  for an array of size  $n$ . *random access*



# Arrays in C++

- Array Declaration
  - `type name [elements];`
  - `int arr[5];`
  - `int arr[n];`
- Array Initialization
  - `int arr[5] = {};`
  - `int arr[5] = {16,2,77,40,12071};`
  - `int arr[] = {16,2,77,40,12071};`
  - `int arr[5] = {10,20,30};`  

1   2   3   4   5



# Arrays in C++

- Accessing Array
  - name[index];
  - Accessing 3<sup>rd</sup> element in the array:
    - arr[2]; 0, 1, 2
  - Indexes should be between 0 and  $n - 1$ .
- Storing elements in an array:
  - for(int i=0; i<n, i++) cin>>arr[i];
- Reading elements from an array:
  - for(int i=0; i<n, i++) cout<<arr[i];



# Next Lecture

- In next lecture, we will discuss linked list based implementation of lists.

