2

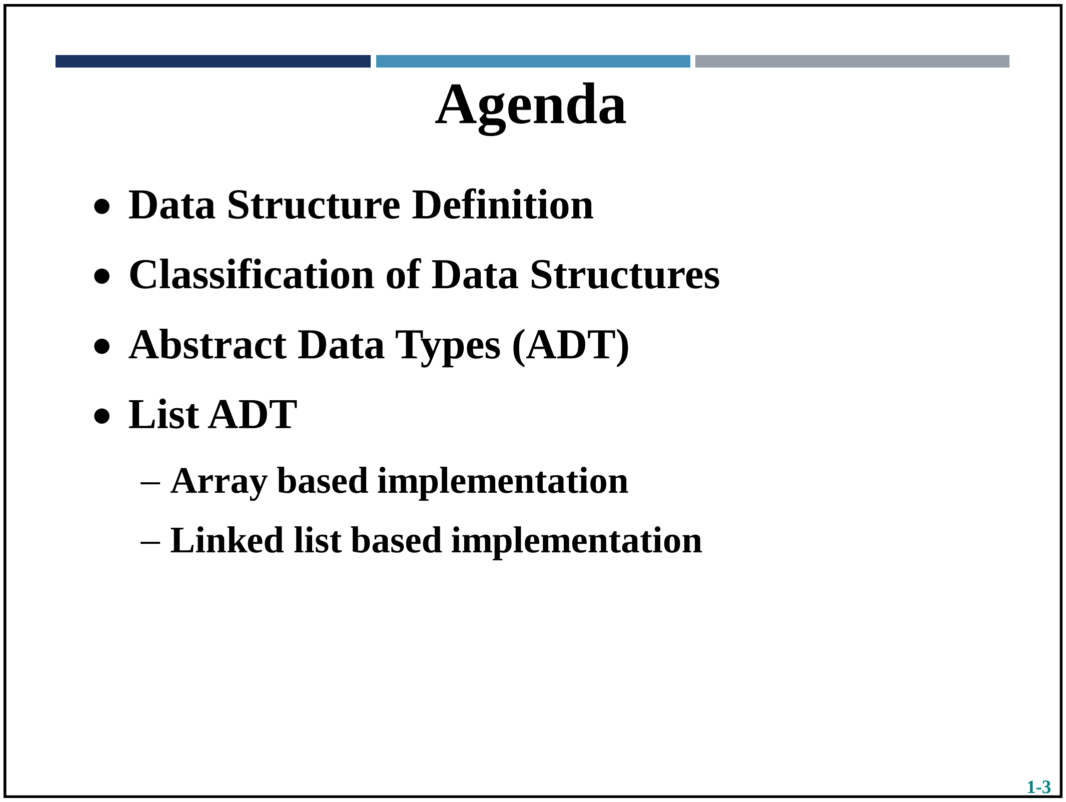
DATA STRUCTURES AND ALGORITHMS - (CIE 205)

LEC_2 LISTS & LINKED LISTS

SPRING (2023-2024)

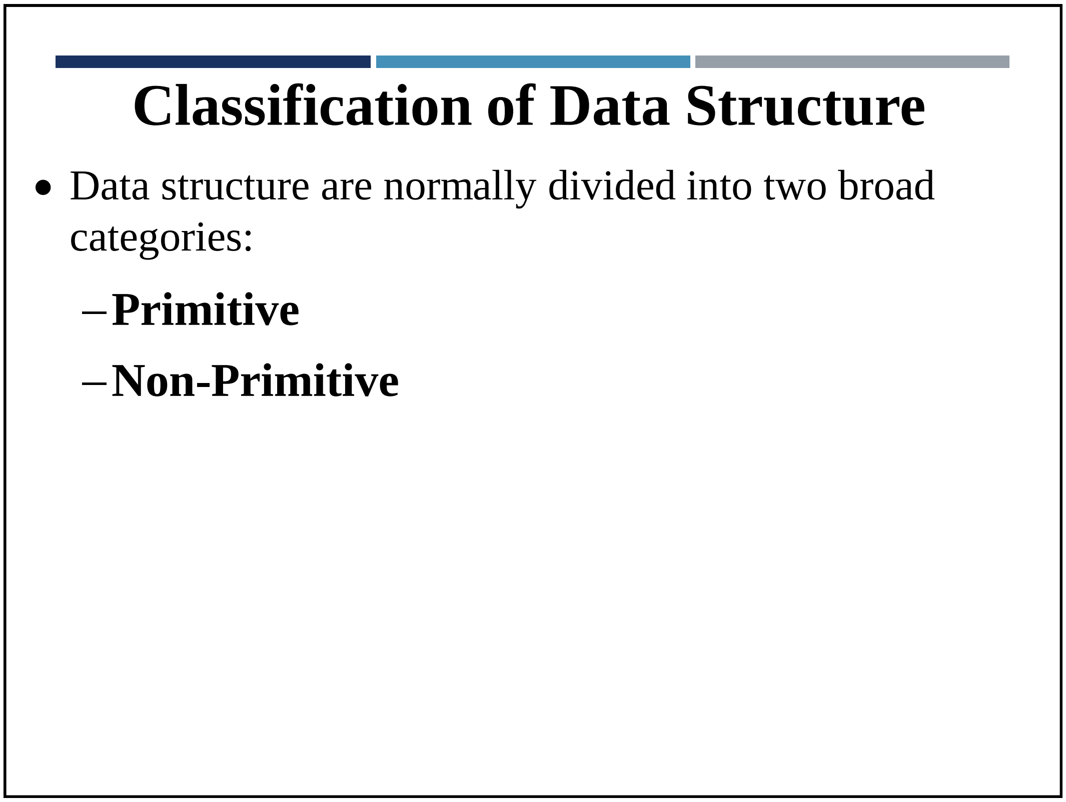
COURSE LEARNING OBJECTIVES

- Use arrays and linked lists to implement and apply linear lists and the related operations (insertion, deletion and traversal).
- Implement and use special linear lists (stacks and queues) and the related basic operations (push and pop; and enqueue and dequeue).
- Implement and use simple sorting algorithms.
- Apply searching algorithms including sequential search, binary search and hashing techniques.
- Apply different String Matching algorithms.
- Use the graph data structure and apply common graph algorithms.
- Implement and use the non-linear list tree data structures (Binary trees, Binary Search Trees) and the related operations (insert, delete and traverse).
- Compare and Select or improve an appropriate solution to a problem.
- Develop teamwork discipline through working in teams.



Agenda

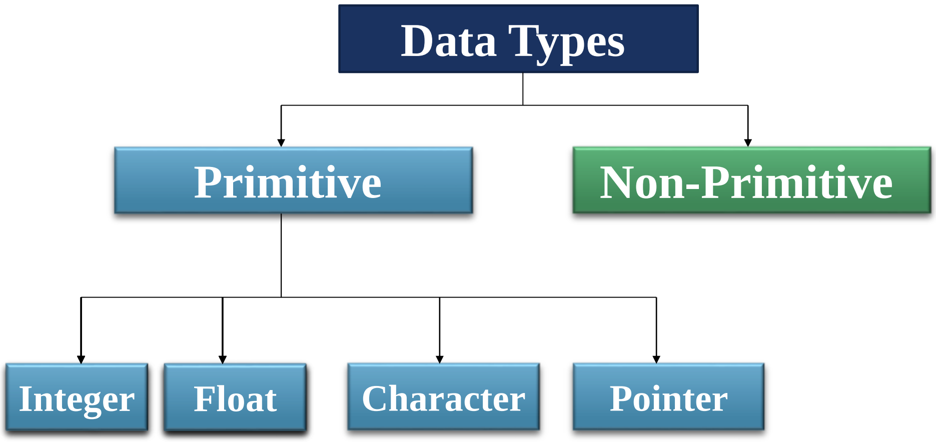
- **Data Structure Definition**
- **Classification of Data Structures**
- **Abstract Data Types (ADT)**
- **List ADT**
  - **Array based implementation**
  - **Linked list based implementation**



Classification of Data Structure

- Data structure are normally divided into two broad categories:
  - **Primitive**
  - **Non-Primitive**

Classification of Data Structure



Data Types

Primitive

Non-Primitive

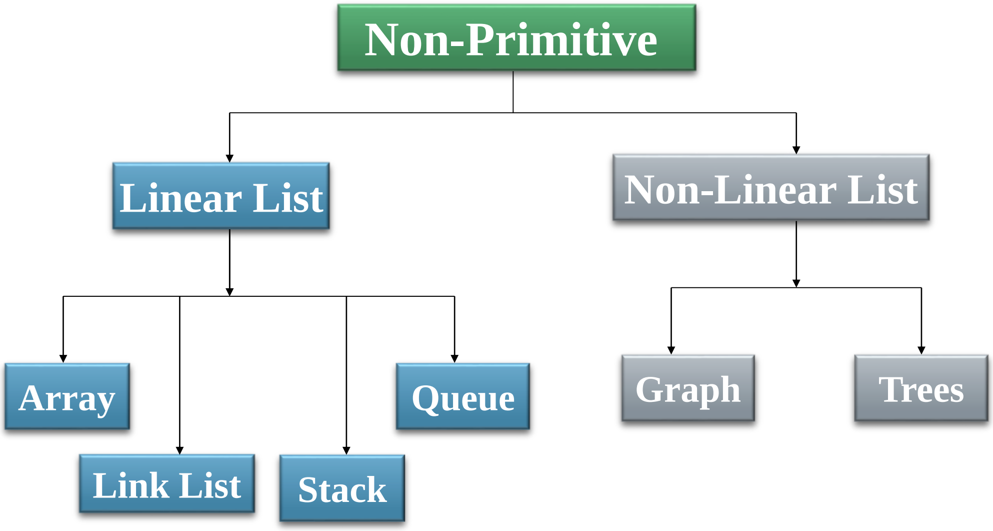
Integer

Float

Character

Pointer

Classification of Data Structure



Non-Primitive

Linear List

Non-Linear List

Array

Link List

Stack

Queue

Graph

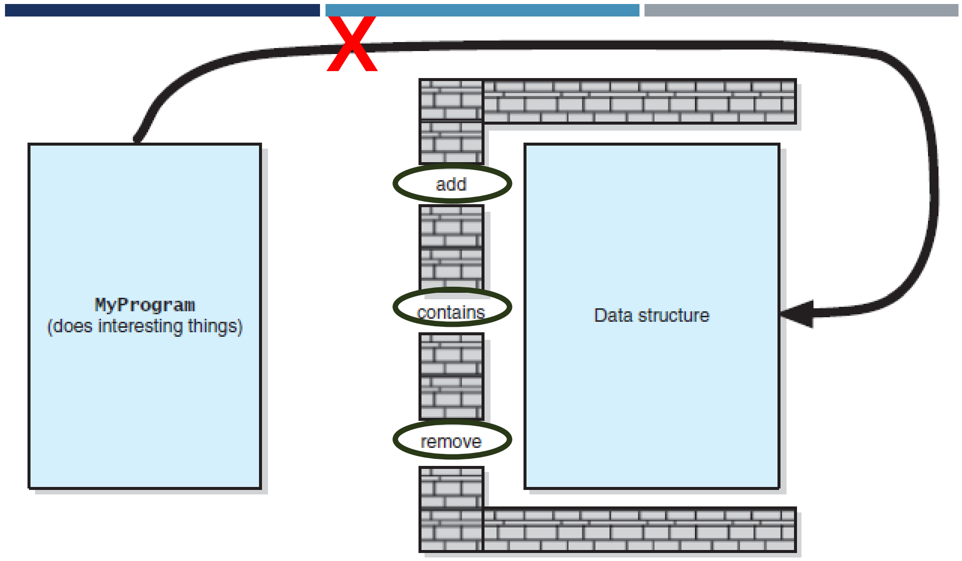
Trees

Abstract Data Types (ADTs)

- **ADT** is a type (or class) for objects **whose behavior is defined by a set of values and a set of operations**.
- The definition of ADT **only** mentions what operations are to be performed **but not** how these operations will be implemented.
- **ABSTRACTION**: it gives an implementation independent view.
- Separate the implementation of Abstract Data Type providing ONLY the interface (BEHAVIOR) to the ADT.

Abstract Data Types (ADTs)

- Implementation of ADT is **hidden** from user
- Elements of the same ADT are all of the same type.
- **Instances** of the same ADT may each consist of different element BUT the behavior of the ADT is the same for all instances.



The program does not access the data structure!
The program only accesses the provided operations of the Abstract Data Type (ADT)! (The Interface or Behavior)

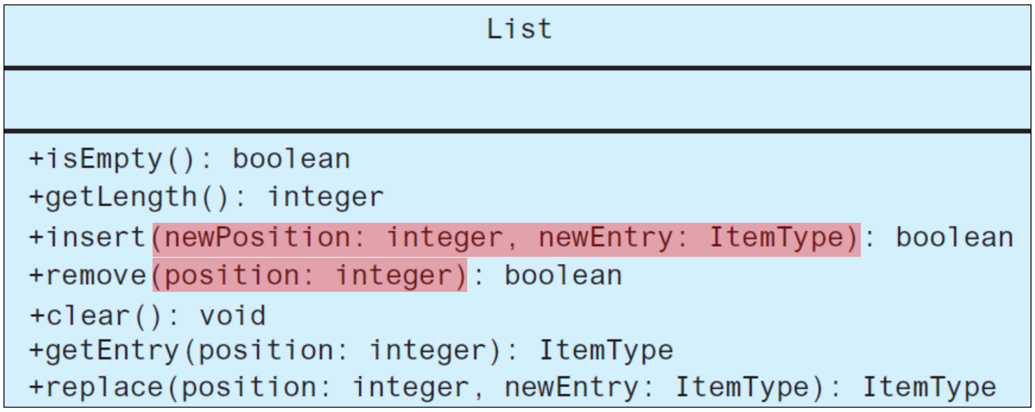
List ADT

- A list contains elements of the **same type** arranged in **sequential order**.
- Operations that can be performed on Lists are :
  - Print List
  - Add new item (insert)
  - Remove item (delete)
  - Remove At
  - Look at (get, Find) entry at given position on list.
  - Replace (set) entry at given position on list.
  - Is empty
  -

The List ADT

- The form of a general list: A_1, A_2, A_3, …, A_N;
- The size of this list is N;
- An **empty list** is a special list of size 0;
- For any list except the empty list, we say that A_{i+1} follows (or succeeds) A_i (i<N) and that A_{i-1} precedes A_i (i>1);
- The first element of the list is A_1, and the last element is A_N. We will not define the predecessor of A_1 or the successor of A_N.
- The position of element A_i in a list is i.

Specifying the ADT List

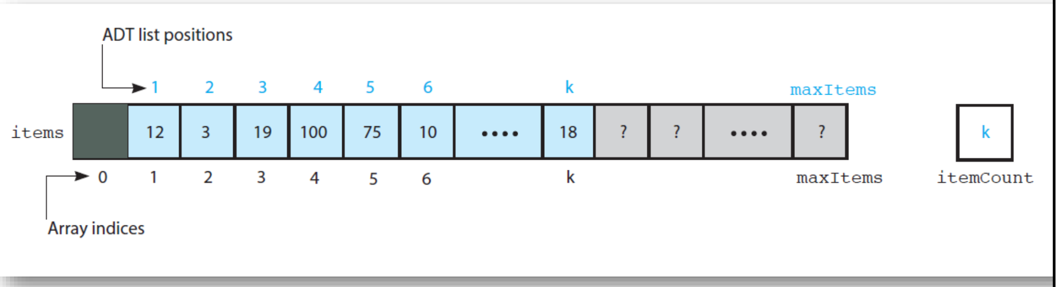


```
List

+isEmpty(): boolean
+getLength(): integer
+insert(newPosition: integer, newEntry: ItemType): boolean
+remove(position: integer): boolean
+clear(): void
+getEntry(position: integer): ItemType
+replace(position: integer, newEntry: ItemType): ItemType
```

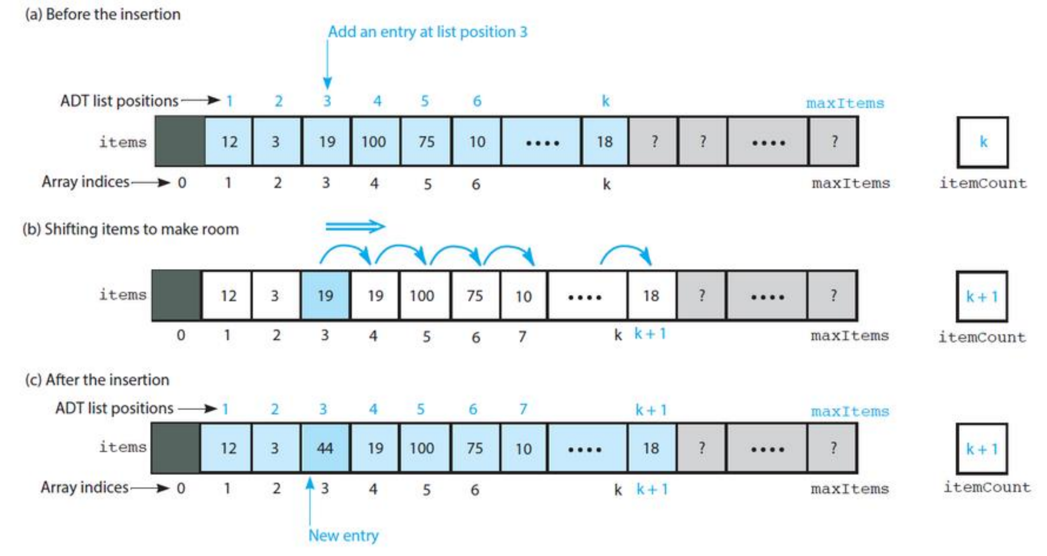
List ADT

- Lists can be implemented using
  - **Arrays** Data Structure
  - **Linked Lists** Data Structure

Array Based Implementation



Array Based Implementation (Insert)



(a) Before the insertion

Add an entry at list position 3

ADT list positions → 1 2 3 4 5 6 k maxItems

items | 12 | 3 | 19 | 100 | 75 | 10 | •••• | 18 | ? | ? | •••• | ? | k itemCount

Array indices → 0 1 2 3 4 5 6 k maxItems

(b) Shifting items to make room

items | 12 | 3 | 19 | 19 | 100 | 75 | 10 | •••• | 18 | ? | •••• | ? | k + 1 itemCount

0 1 2 3 4 5 6 7 k k + 1 maxItems

(c) After the insertion

ADT list positions → 1 2 3 4 5 6 7 k + 1 maxItems

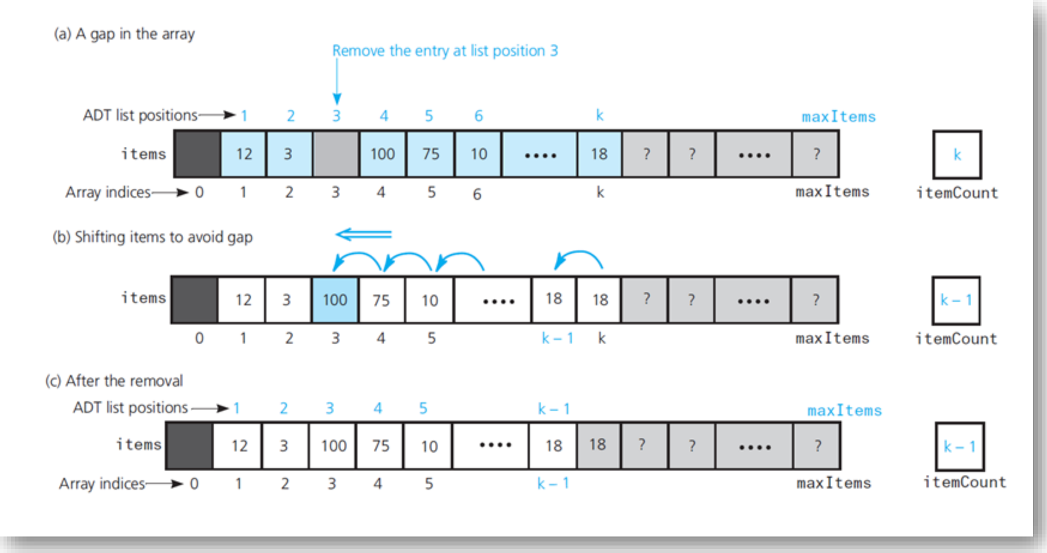
items | 12 | 3 | 44 | 19 | 100 | 75 | 10 | •••• | 18 | ? | •••• | ? | k + 1 itemCount

Array indices → 0 1 2 3 4 5 6 k k + 1 maxItems

New entry

Shifting items for insertion

Array Based Implementation (Remove)



(a) A gap in the array

Remove the entry at list position 3

ADT list positions → 1 2 3 4 5 6 k maxItems

items | 12 | 3 | | 100 | 75 | 10 | •••• | 18 | ? | ? | •••• | ? | k itemCount

Array indices → 0 1 2 3 4 5 6 k maxItems

(b) Shifting items to avoid gap

items | 12 | 3 | 100 | 75 | 10 | •••• | 18 | 18 | ? | ? | •••• | ? | k – 1 itemCount

0 1 2 3 4 5 k – 1 k maxItems

(c) After the removal

ADT list positions → 1 2 3 4 5 k – 1 maxItems

items | 12 | 3 | 100 | 75 | 10 | •••• | 18 | 18 | ? | ? | •••• | ? | k – 1 itemCount

Array indices → 0 1 2 3 4 5 k – 1 maxItems

Shifting items for Deletion

Array based Implementation

- **Disadvantages:**
  - An **estimate of the maximum size of the list is required**, even if the array is dynamically allocated. Usually this requires a high overestimate, which wastes considerable space.
  - Insertion and deletion are expensive. For example, inserting at position 0 requires first pushing the entire array down one spot to make room. (***Array Shifting***)

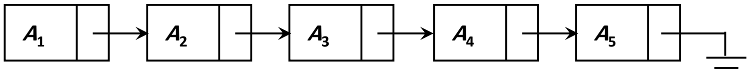
Because the running time for insertions and deletions is so slow and the list size must be known in advance, simple arrays are generally not used to implement lists.

Linked Lists

- Another way to organize data items
  - Place them within objects—usually called **nodes**
  - Linked together into a “chain,” one after the other

The linked list consists of a series of structures, which are **not necessarily adjacent in memory**.

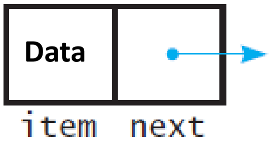
Each data item needs a **pointer** to lead to the next data item(*next pointer*).



A linked list

Linked Lists

- Each data item needs a **pointer** to lead to the next data item(*next pointer*).

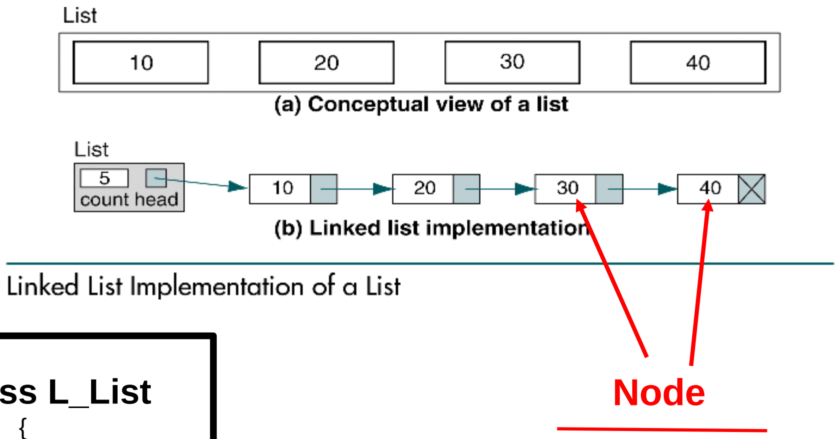


A node is implemented as a structure OR a class

```
struct Node
 { int data;
    Node*  next;
  };
```

```
class Node
 { int data;
    Node*  next;
 public:
     ……….
// set and get functions   };
```

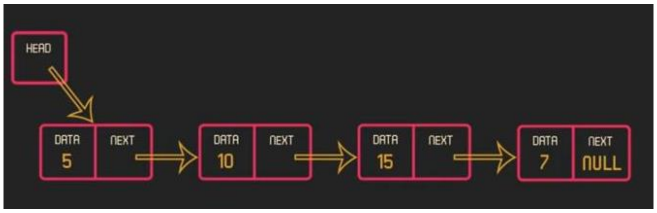
Linked List Implementation

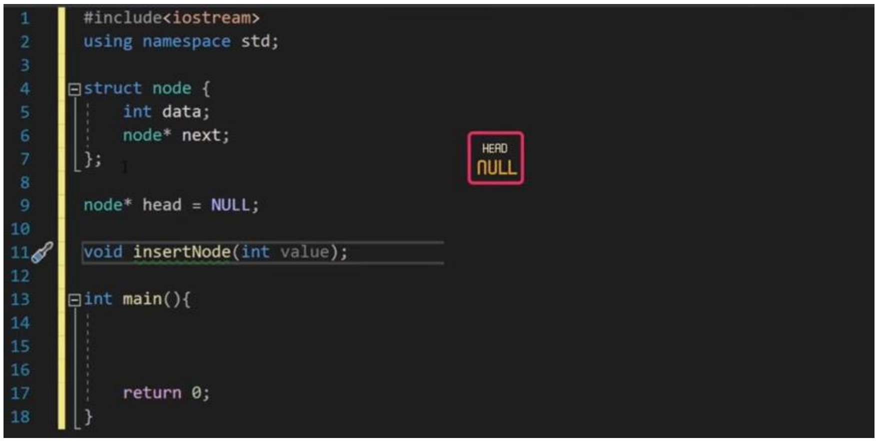


Linked List Implementation of a List

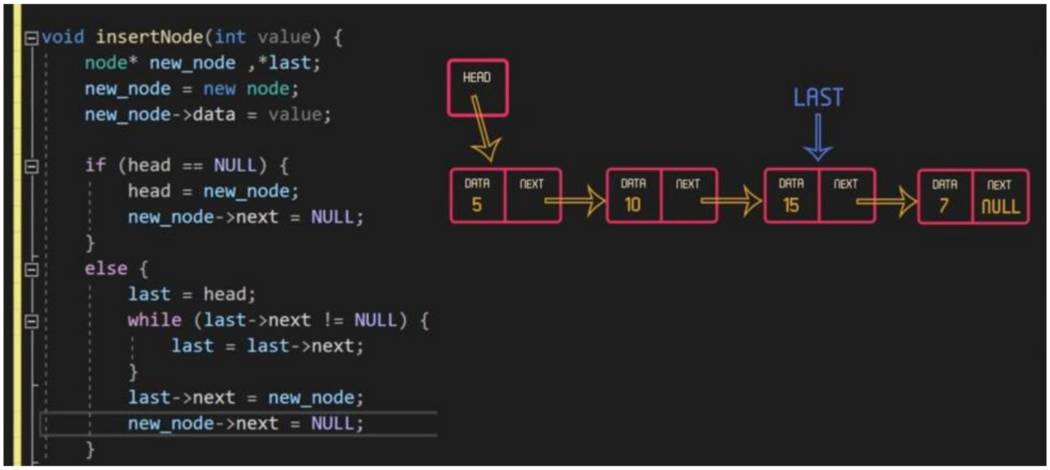
```
class L_List
   {
     Node*  head;
     int itemCount;
   };
```

Linked List Implementation

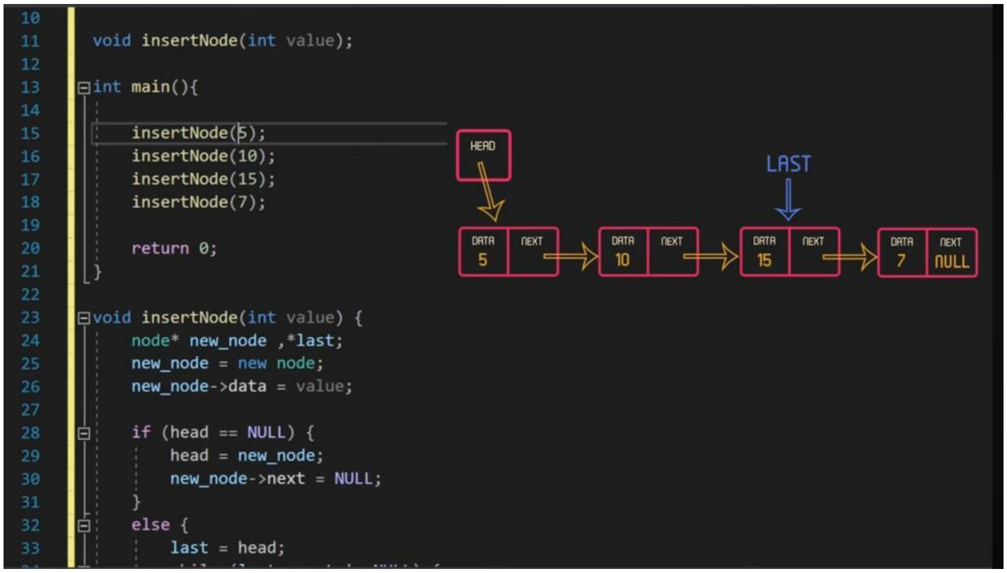



```
#include<iostream>
using namespace std;

struct node {
    int data;
    node* next;
};

node* head = NULL;

void insertNode(int value);

int main(){



    return 0;
}
```

Linked List Implementation


```
void insertNode(int value) {
    node* new_node ,*last;
    new_node = new node;
    new_node->data = value;

    if (head == NULL) {
        head = new_node;
        new_node->next = NULL;
    }
    else {
        last = head;
        while (last->next != NULL) {
            last = last->next;
        }
        last->next = new_node;
        new_node->next = NULL;
    }
```

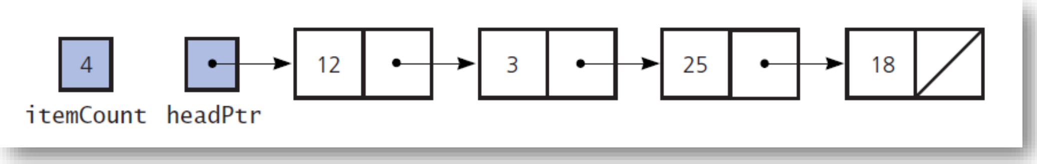
Linked List Implementation

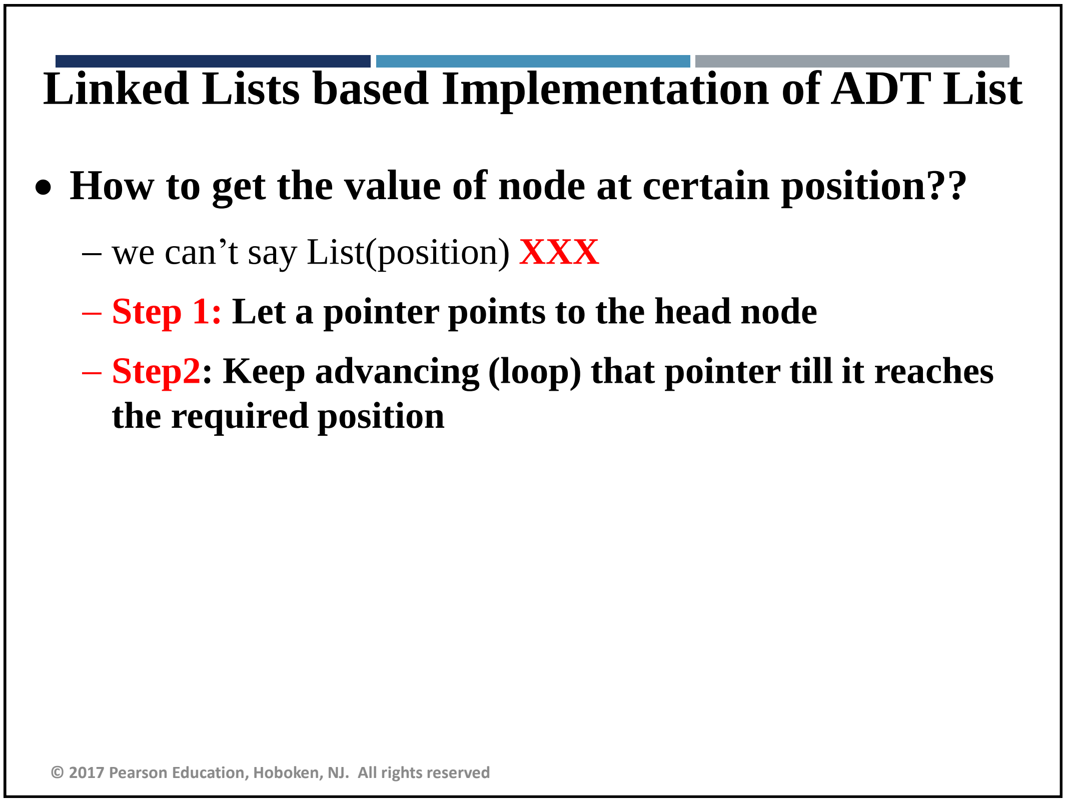


```
void insertNode(int value);

int main(){

    insertNode(5);
    insertNode(10);
    insertNode(15);
    insertNode(7);

    return 0;
}

void insertNode(int value) {
    node* new_node ,*last;
    new_node = new node;
    new_node->data = value;

    if (head == NULL) {
        head = new_node;
        new_node->next = NULL;
    }
    else {
        last = head;
```

Linked Lists based Implementation of ADT List

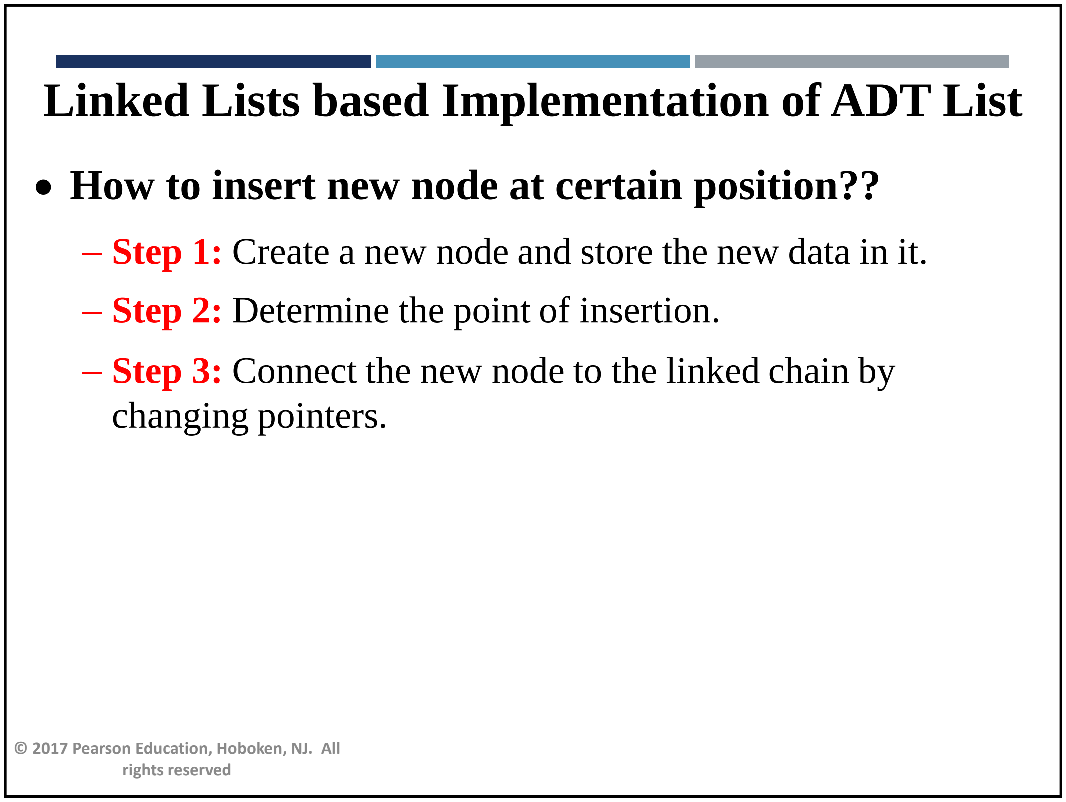
- A link-based implementation of the ADT list





Linked Lists based Implementation of ADT List

- **How to get the value of node at certain position??**
  - we can’t say List(position) **XXX**
  - **Step 1: Let a pointer points to the head node**
  - **Step2: Keep advancing (loop) that pointer till it reaches the required position**

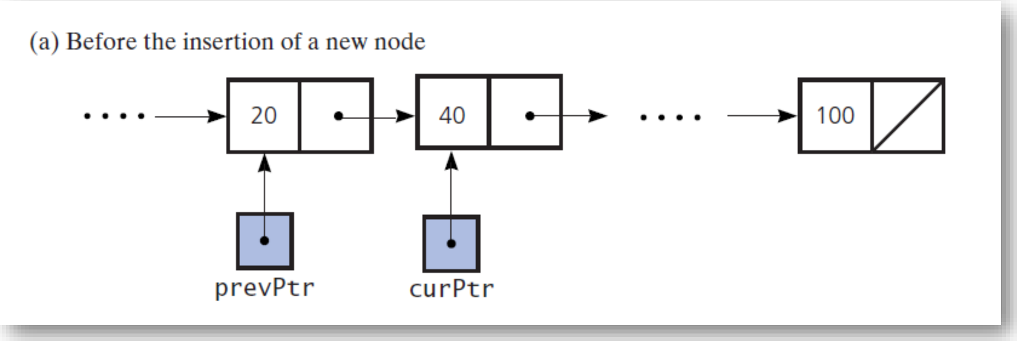


Linked Lists based Implementation of ADT List

- **How to insert new node at certain position??**
  - **Step 1:** Create a new node and store the new data in it.
  - **Step 2:** Determine the point of insertion.
  - **Step 3:** Connect the new node to the linked chain by changing pointers.

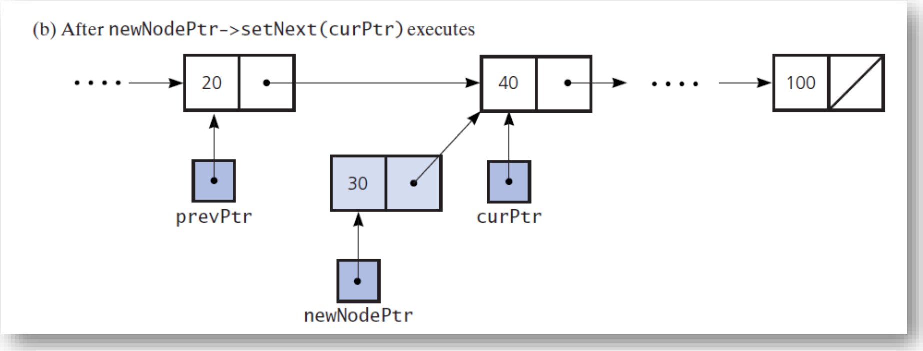
Node Insertion

- Inserting a new node between existing nodes of a linked chain



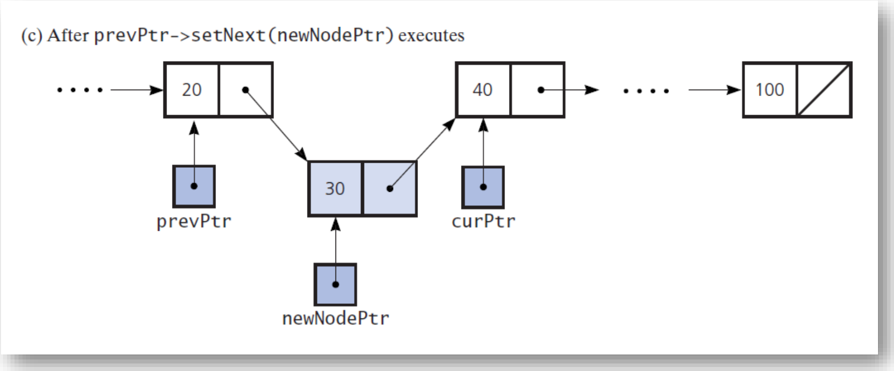
Node Insertion

- Inserting a new node between existing nodes of a linked chain



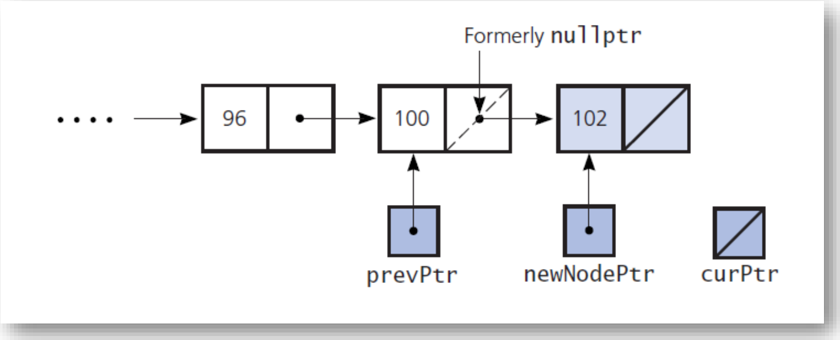
Node Insertion

- Inserting a new node between existing nodes of a linked chain

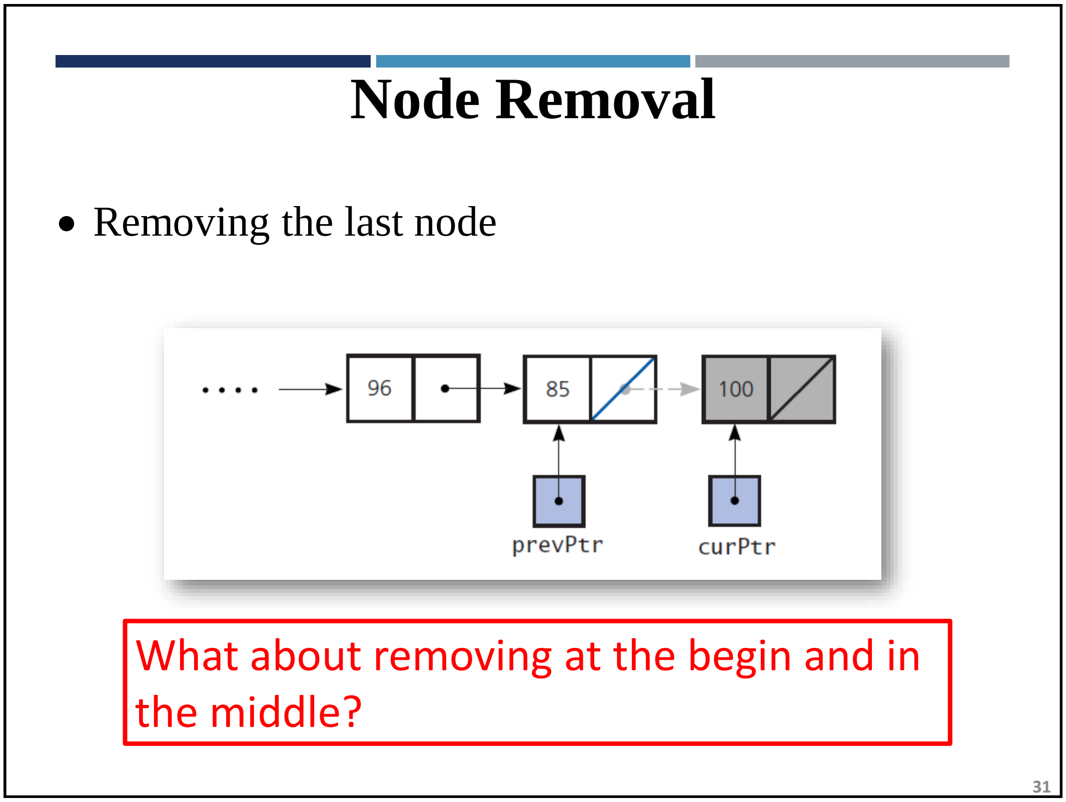

(c) After prevPtr->setNext(newNodePtr) executes

Node Insertion

- Inserting a new node between existing nodes of a linked chain



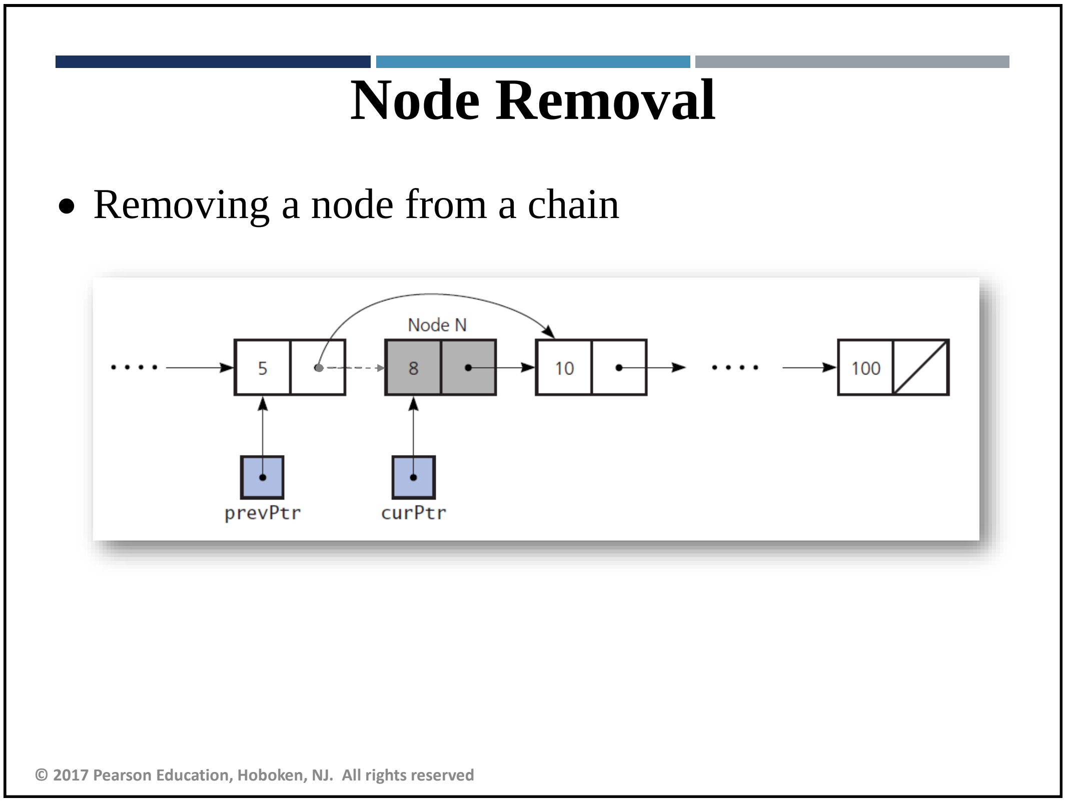
What about inserting at the begin?



Node Removal

- Removing the last node

What about removing at the begin and in the middle?



Node Removal

- Removing a node from a chain

مدينة زويل للعلوم والتكنولوجيا
Zewail City of Science and Technology

THANK YOU!

DR YASSER ELAWADY

YELAWADY@ZEWAILCITY.EDU.EG