CS-218
DATA STRUCTURE

Dr. Hashim Yasin

National University of Computer and Emerging Sciences,

Faisalabad, Pakistan.

LINKED LIST

Inefficiency in Array-based List

- If insertions and deletions occur frequently throughout the list and, in particular, at the front of the list, then the array is not a good option.
- In order to avoid the cost of insertion and deletion operations, we need to ensure that the list is **NOT** stored **contiguously (together in sequence)**,
  - The reason is, entire parts of the list will need to be moved.
- Solution is, …

We look for an alternative implementation, that is *linked list*.

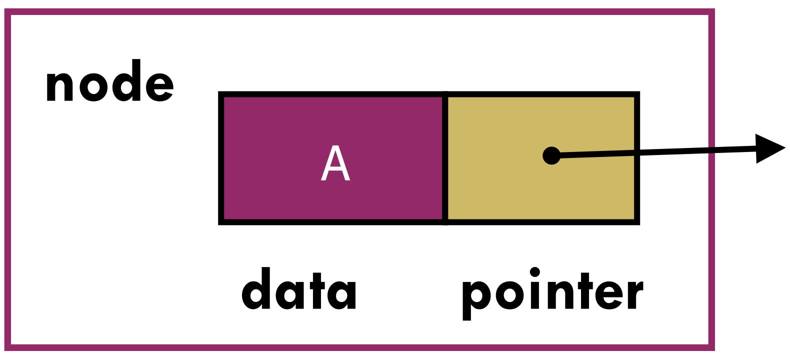
Linked List

- Linked lists
  - Linked List is a sequence of links which contains items. Each link contains a connection to another link.
- Basic operations of linked lists
  - Insert, find, delete, print, etc.
- Variations of linked lists
  - Single linked lists
  - Circular linked lists
  - Doubly linked lists

SINGLE LINKED LIST

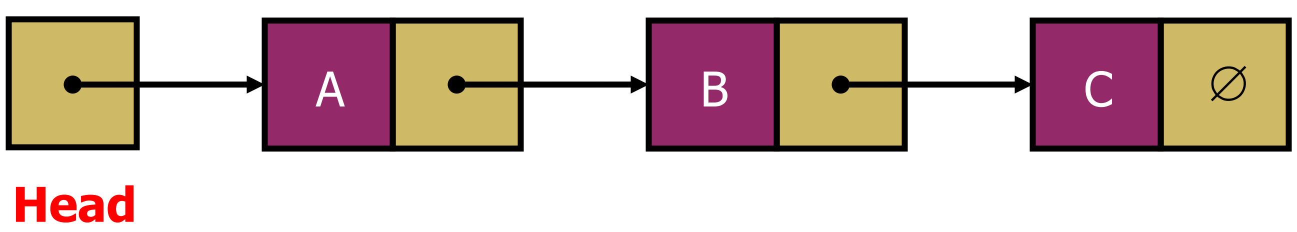
Linked List

- A *linked list* is a series of connected *nodes*
- **Each node** contains at least
  - A piece of data (any type)
  - A pointer to the next node in the list



Linked List

- A *linked list* is a series of connected *nodes*
- **Each node** contains at least
  - A piece of data (any type)
  - A pointer to the next node in the list
- ***Head***: pointer to the first node
- The last node points to `NULL`



A Simple Linked List Class

- We use two classes: **Node** and **List**
- Declare Node class *for the nodes*
  - data: double-type data in this example
  - next: (pointer of object type) a pointer to the next node in the list

```
class Node {
public:
        double data;            // data
        Node*  next;            // pointer to next
};
```

A Simple Linked List Class

- Declare `List` class, which contains
  - `head`: a pointer to the first node in the list.
    - If the list is empty initially, `head` is set to `NULL`
  - Operations on `List`

```
bool   IsEmpty()
Node*  InsertNode(int index, double x);
int    FindNode(double x);
int    DeleteNode(double x);
void   DisplayList(void);
```

A Simple Linked List Class

- Operations of `List`
  - `IsEmpty`: determine whether or not the list is empty
  - `InsertNode`: insert a new node at a particular position
  - `FindNode`: find a node with a given value
  - `DeleteNode`: delete a node with a given value
  - `DisplayList`: print all the nodes in the list

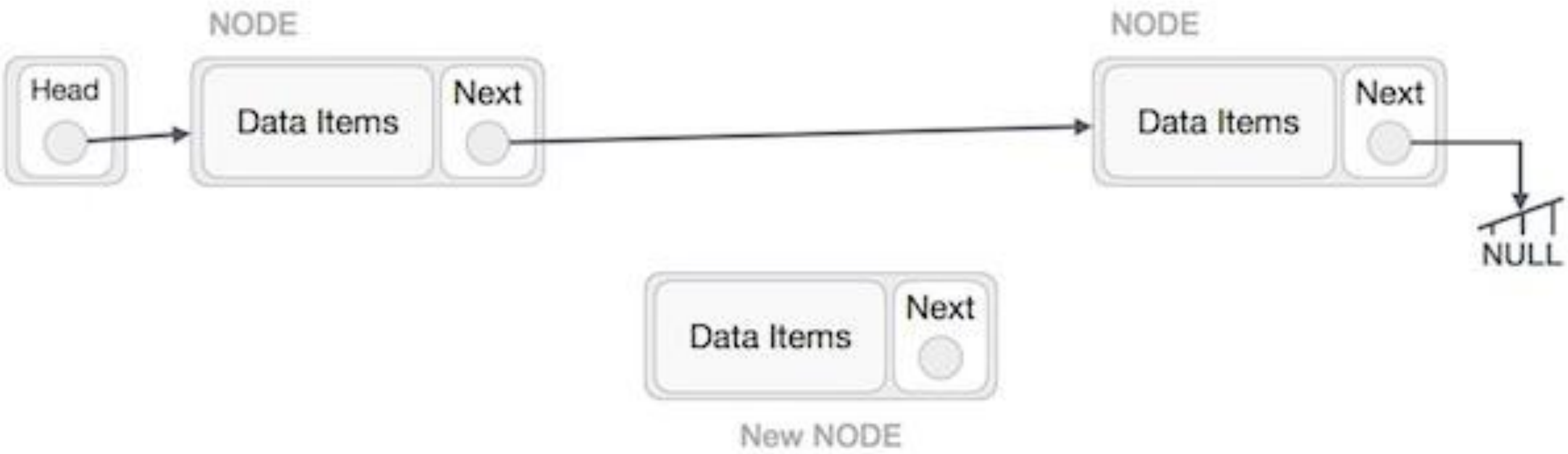
A Simple Linked List Class


```
class List {
public:
      List(void) {                    // constructor
            head = NULL;
      }
      ~List(void);                    // destructor

      bool  IsEmpty() { return head == NULL; }
      Node* InsertNode(int index, double x);
      int   FindNode(double x);
      int   DeleteNode(double x);
      void  DisplayList(void);
private:
      Node* head;  // a pointer to the first node
};
```

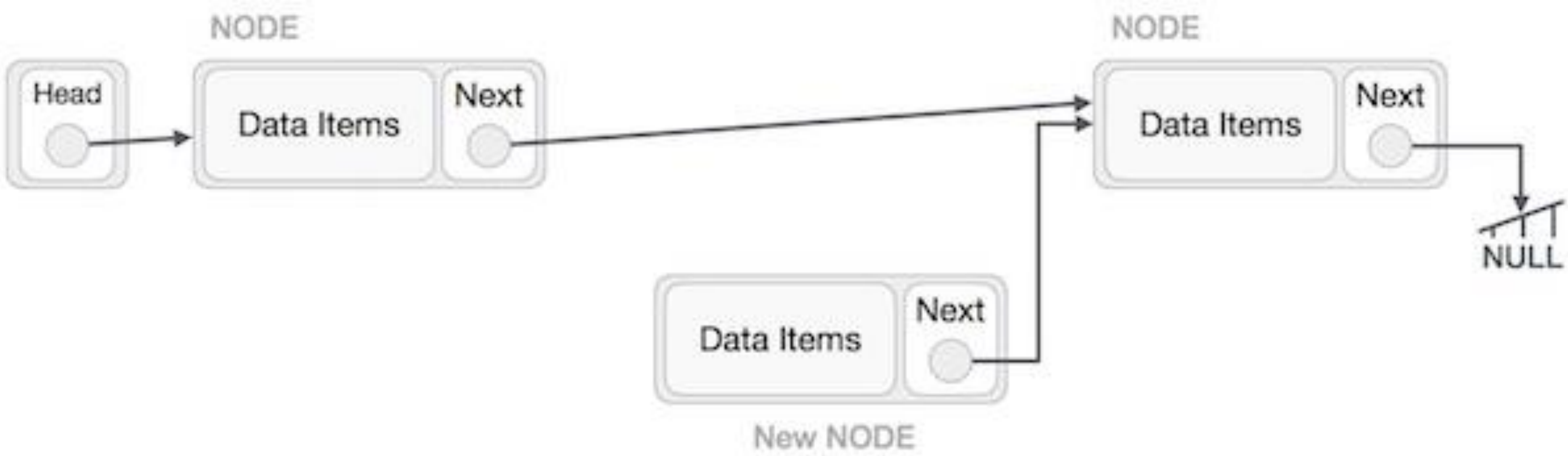
INSERTION

Inserting a New Node

Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode) and **C** (RightNode).

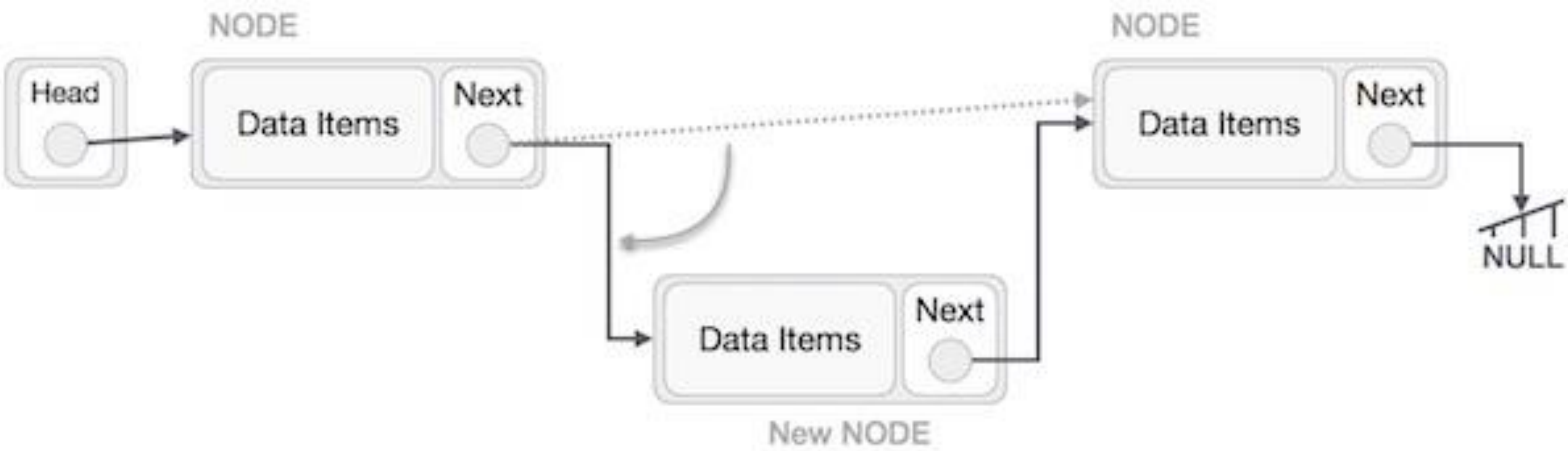


Inserting a New Node




```
NewNode.next -> RightNode;
```

Inserting a New Node




```
LeftNode.next -> NewNode;
```

Inserting a New Node

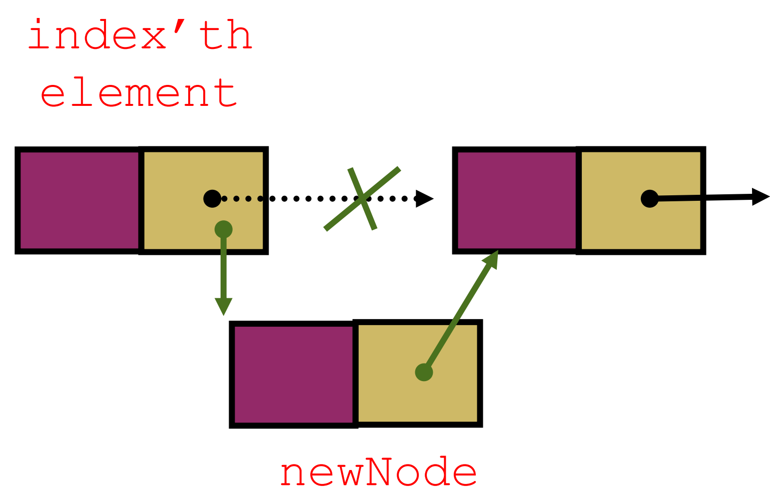
This will put the new node in the middle of the two.

The new list should look like this –

Details … Inserting a New Node

The main steps are:

- ✓ Locate `index`'th element
- ✓ Allocate memory for the `newNode`
- ✓ Point the `newNode` to its successor
- ✓ Point the `newNode`'s predecessor to the `newNode`



Details … Inserting a New Node

- `Node* InsertNode(int index, double x)`
  - Insert a node with data equal to `x` after the `index`’th elements, i.e.,
    - when `index = 0`, insert the node as the first element;
    - when `index = 1`, insert the node after the first element, and so on)
  - If the insertion is successful, return the inserted node, otherwise, return `NULL`.
    - (If `index` is < 0 or > length of the list, the insertion will fail.)

Details … Inserting a New Node

- Possible cases of `InsertNode`
  1. Insert into an empty list
  2. Insert in front
  3. Insert at back
  4. Insert in middle
- But, in fact, only need to handle **two cases**
  - Insert as the first node (Case 1 and Case 2)
  - Insert in the middle or at the end of the list (Case 3 and Case 4)

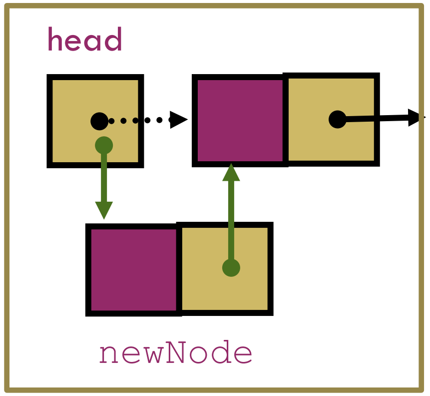
```
Node* List::InsertNode(int index, double x){
   if (index < 0) return NULL;

   int currIndex     =     1;
   Node* currNode    =     head;
   while (currNode && index > currIndex) {
        currNode     =     currNode->next;
        currIndex++;
   }
   if (index > 0 && currNode == NULL) return NULL;

   Node* newNode     =     new Node;
   newNode->data     =     x;
   if (index == 0) {
        newNode->next =    head;
        head          =    newNode;
   }
   else {
        newNode->next  =   currNode->next;
        currNode->next =   newNode;
   }
   return newNode;
}
```

Try to locate `index`'th node. If it doesn't exist, return `NULL`.

```
Node* List::InsertNode(int index, double x){
   if (index < 0) return NULL;

   int currIndex     =      1;
   Node* currNode    =      head;
   while (currNode && index > currIndex) {
       currNode      =      currNode->next;
       currIndex++;
   }
   if (index > 0 && currNode == NULL) return NULL;

   Node* newNode     =      new Node;
   newNode->data     =      x;
   if (index == 0) {
       newNode->next =      head;
       head           =     newNode;
   }
   else {
       newNode->next  =     currNode->next;
       currNode->next =     newNode;
   }
   return newNode;
}
```

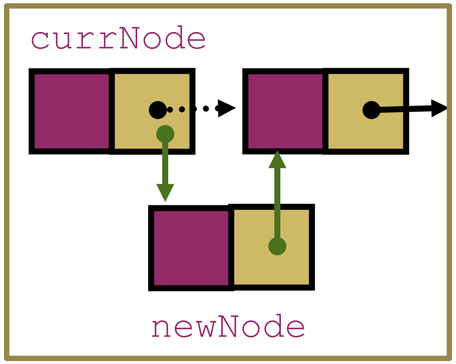
Create a new node.

```
Node* List::InsertNode(int index, double x){
   if (index < 0) return NULL;

   int currIndex      =      1;
   Node* currNode     =      head;
   while (currNode && index > currIndex) {
        currNode      =      currNode->next;
        currIndex++;
   }
   if (index > 0 && currNode == NULL) return NULL;

   Node* newNode      =      new Node;
   newNode->data      =      x;
   if (index == 0) {
        newNode->next =      head;
        head          =      newNode;
   }
   else {
        newNode->next   =    currNode->next;
        currNode->next  =    newNode;
   }
   return newNode;
}
```



Insert as first element

```
Node* List::InsertNode(int index, double x){
   if (index < 0) return NULL;

   int currIndex    =     1;
   Node* currNode   =     head;
   while (currNode && index > currIndex) {
        currNode    =     currNode->next;
        currIndex++;
   }
   if (index > 0 && currNode == NULL) return NULL;

   Node* newNode    =     new Node;
   newNode->data    =     x;
   if (index == 0) {
        newNode->next =    head;
        head          =    newNode;
   }
   else {
        newNode->next  =   currNode->next;
        currNode->next =   newNode;
   }
   return newNode;
}
```

Insert after currNode

SEARCHING

Finding a Node

- `int FindNode(double x)`
  - Search for a node with the value equal to `x` in the list.
  - If such a node is found, return its position. Otherwise, return 0.

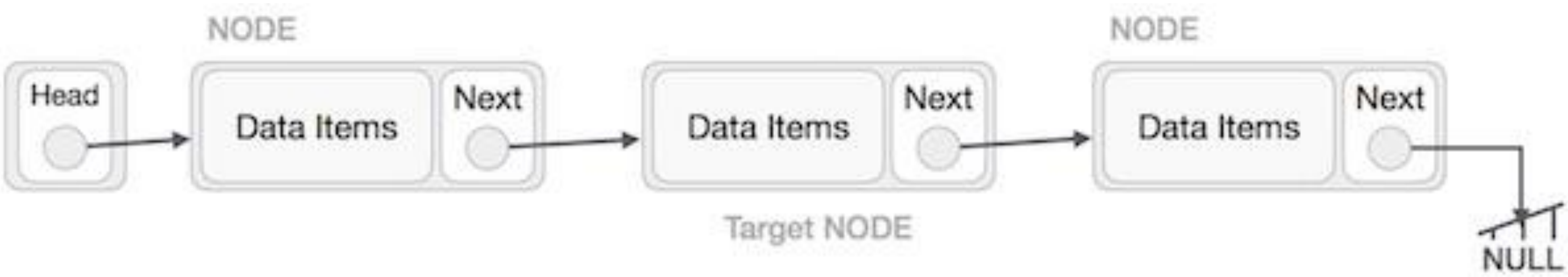
Finding a Node


```
int List::FindNode(double x) {
  Node* currNode =  head;
  int currIndex  =  1;
  while (currNode && currNode->data != x){
      currNode  =     currNode->next;
      currIndex++;
  }
  if (currNode)
      return currIndex;
  return 0;
}
```

DELETION

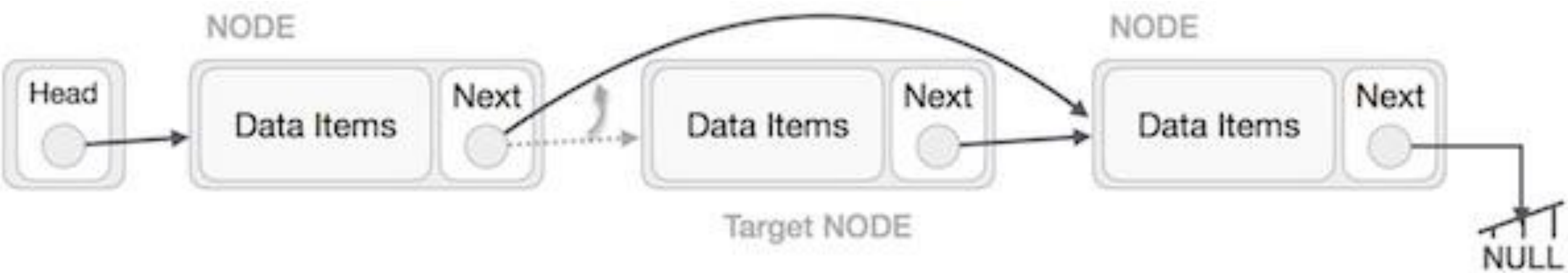
Deleting a Node

First, locate the target node to be removed, by using searching algorithms.



Deleting a Node

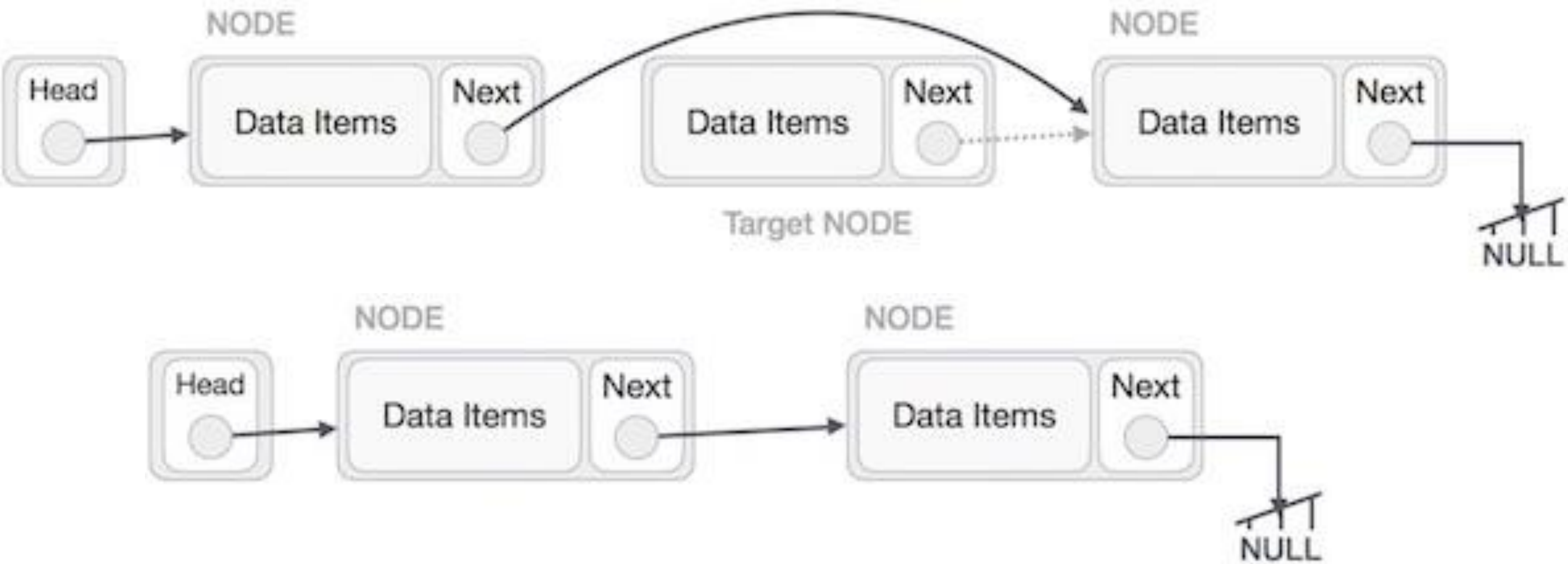
The left (previous) node of the target node now should point to the next node of the target node −




```
LeftNode.next -> TargetNode.next;
```

Deleting a Node

We remove what the target node is pointing at.




```
TargetNode.next -> NULL;
```

Deleting a Node

- `int DeleteNode(double x)`
  - Delete a node with the value equal to `x` from the list.
  - If such a node is found, return its position. Otherwise, return 0.
- Steps
  - Find the desirable node (similar to `FindNode`)
  - Release the memory occupied by the found node
  - Set the pointer of the predecessor of the found node to the successor of the found node
- Like `InsertNode`, there are two special cases
  - Delete first node
  - Delete the node in middle or at the end of the list

OTHER OPERATIONS

Printing All the Elements

- `void DisplayList(void)`
  - Print the data of all the elements
  - Print the number of the nodes in the list

Destroying the List

- `~List(void)`
  - Use the destructor to release all the memory used by the list.
  - Step through the list and delete each node one by one.

Using Linked List

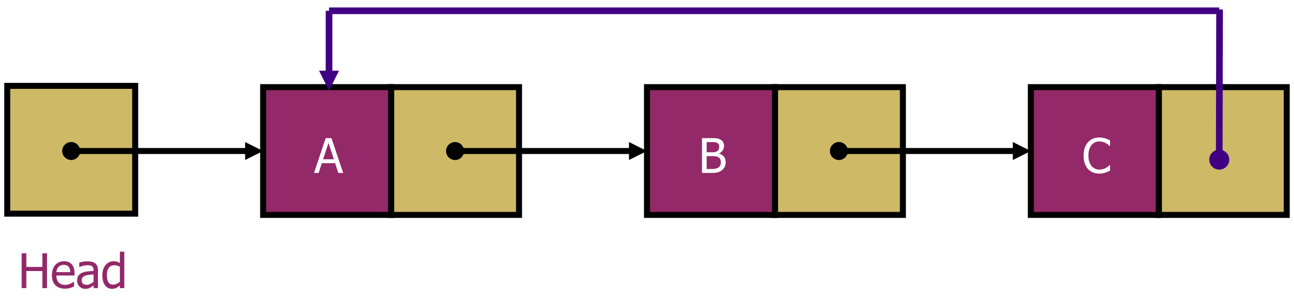

```
int main(void){
   List list;
   list.InsertNode(0, 7.0);     // successful
   list.InsertNode(1, 5.0);     // successful
   list.InsertNode(-1, 5.0);    // unsuccessful
   list.InsertNode(0, 6.0);     // successful
   list.InsertNode(8, 4.0);     // unsuccessful
   // print all the elements
   list.DisplayList();
   if(list.FindNode(5.0) > 0)
         cout << "5.0 found" << endl;
   else
         cout << "5.0 not found" << endl;
   list.DeleteNode(7.0);
   list.DisplayList();
   return 0;
}
```

Final Output

6
7
5
Number of nodes in the list: 3
5.0 found
6
5
Number of nodes in the list: 2

Circular Linked List

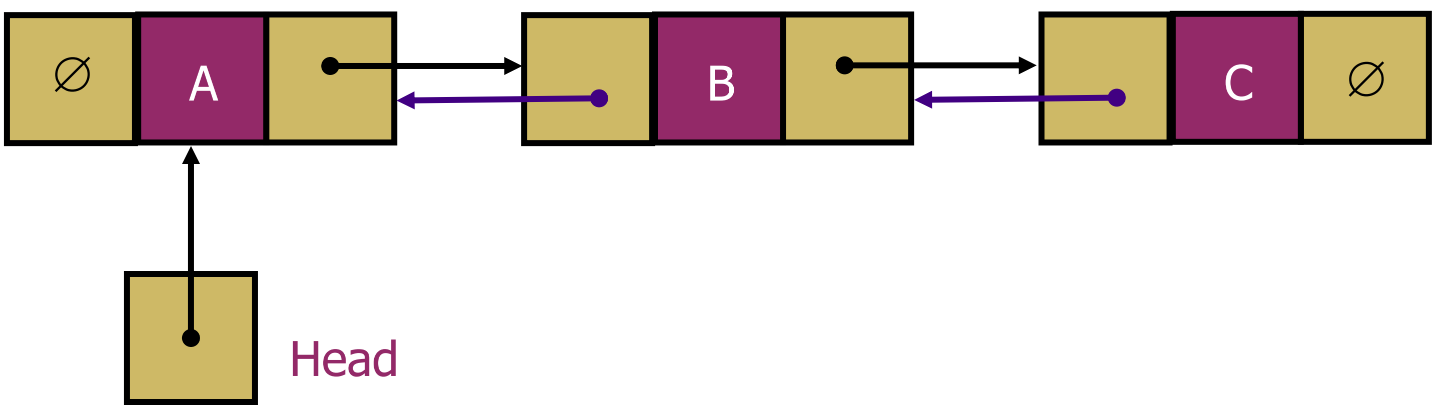
- The last node points to the first node of the list



- How do we know when we have finished traversing the list?
  - (Tip: check if the pointer of the current node is equal to the head.)

Doubly Linked List

- Each node points to not only successor but the predecessor
- There are two `NULL`:
  - at the first and last nodes in the list
- Advantage: given a node, it is easy to visit its predecessor. Convenient to traverse lists backwards



Reading Materials

- Nell Dale: Chapter # 2 (Section 2.1), Chapter # 3
- Schaum's Outlines: Chapter # 1
- Mark A. Weiss: Chapter # 3 (Section – 3.1, 3.2)
- Articles: (abstraction vs. encapsulation)