CS-202

Dynamic Data Structures (Pt.2)

C. Papachristos
Autonomous Robots Lab
University of Nevada, Reno

Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday		Sunday
			Lab (8 Sections)			
	CLASS		CLASS			
PASS Session	PASS Session	**Project DEADLINE**	NEW Project	PASS Session		PASS Session

Your Smart Pointer(s) *extra-grade* Project X Deadline is this Wednesday 11/7.

- PASS Sessions held Friday-Sunday-&-Monday-Tuesday, get all the help you need!
- 24-hrs delay after Project Deadline incurs 20% grade penalty.
- Past that, NO Project accepted. Better send what you have in time!

Today's Topics

Dynamic Data Structures

Forward – Linked List(s)
- Node-based
- Array-based

Doubly – Linked List(s)

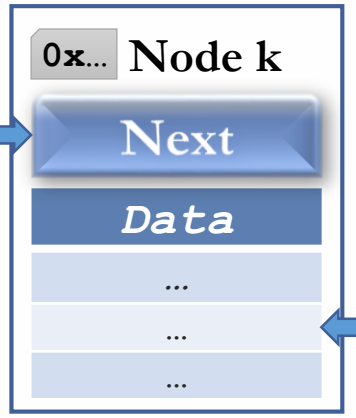
Forward – Linked List Node(s)

The Node(s) (of the FLL)

A Node of the FLL is an element of the Dynamic Data Structure.

- Typically a `class` encapsulating Data Object(s).
- Holds Data & Pointer to associate "Next" Node in the Forward–Linked List.
- List implementation has to Access & Mutate it.

Necessary: Maintains association(s) to other LL Nodes.
Can:
- Point to another Node.
- Be `NULL`.



Stored Data can be simple data types (`int`, `double`, …) or complex ones (`class`es/ `struct`s).

Provide Get/Set methods or give class List direct access to class Node members.

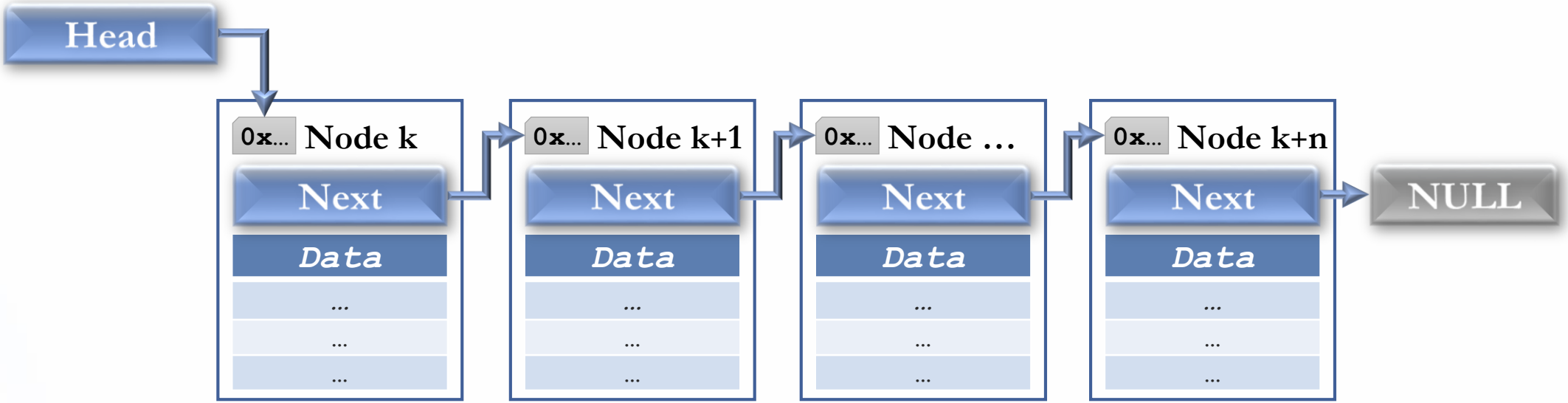
```
class Node {
 public:
 // ctor(s)
 // dtor
 // get - set methods
 …
 friend class ForwardList;
 …
 private:
 int m_data;
 Node * m_next;
};
```

Forward – Linked List (*Node*-based)

The Linked-List

The Structure:

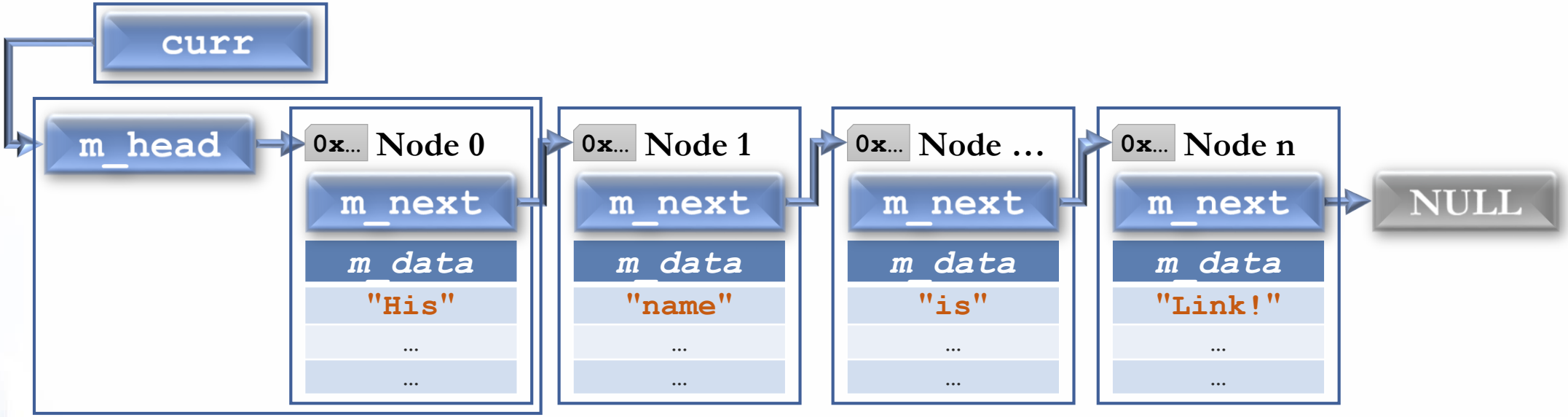
- Each Node contains Data and the Address of "Next" Node in the Linked-List.
- Linked-List has a Head pointer to "First" Node.



Forward – Linked List (*Node*-based)

Linked-List Traversal

To perform LL Traversal, a control loop is used:

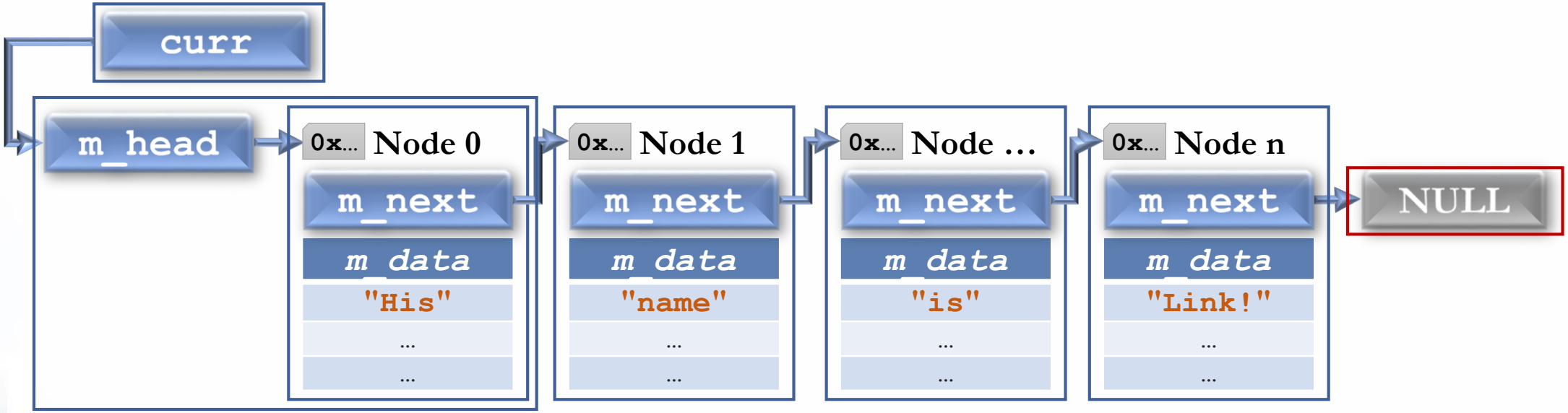



```
for (Node * curr = m_head; curr!=NULL; curr = curr->m_next)
  cout << curr->m_data << endl; //(overloaded) insertion for m_data type
```

Forward – Linked List (*Node*-based)

Linked-List Traversal

To perform LL Traversal, a control loop is used:

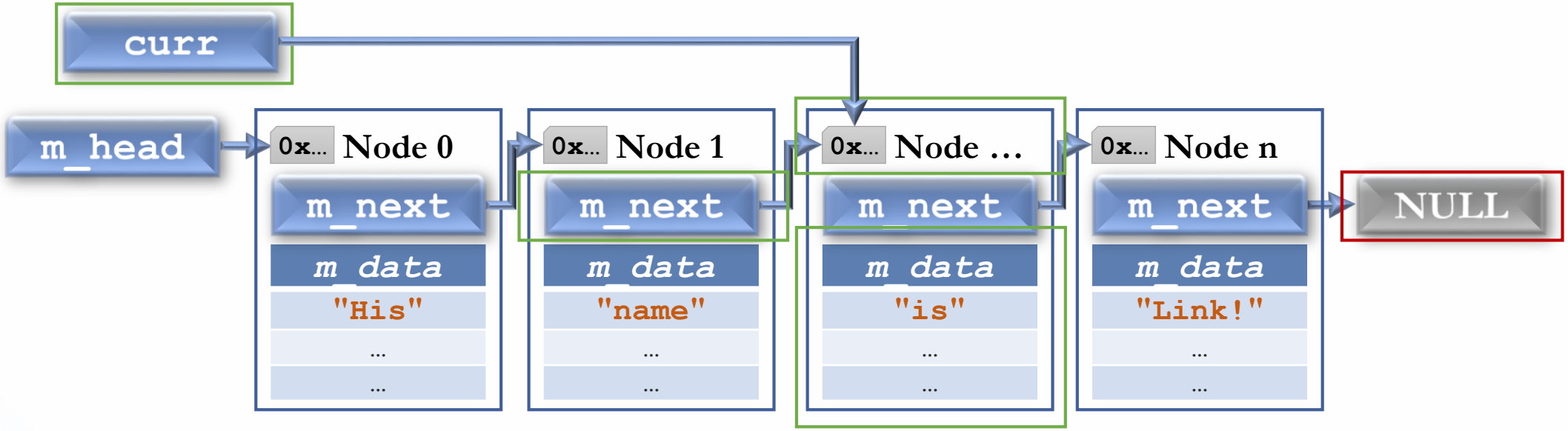



```
for (Node * curr = m_head; curr!=NULL; curr = curr->m_next)
  cout << curr->m_data << endl; //(overloaded) insertion for m_data type
```

Forward – Linked List (*Node*-based)

Linked-List Traversal

To perform LL Traversal, a control loop is used:

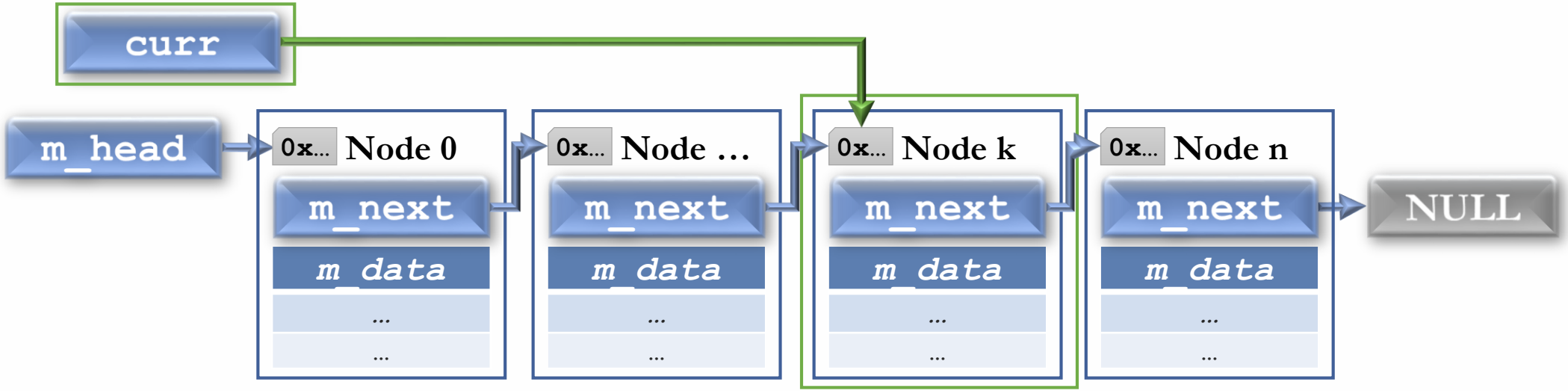



```
for (Node * curr = m_head; curr!=NULL; curr = curr->m_next)
  cout << curr->m_data << endl; //(overloaded) insertion for m_data type
```

Forward – Linked List (*Node*-based)

Linked-List Insertion

Find target Node to insert *after*.

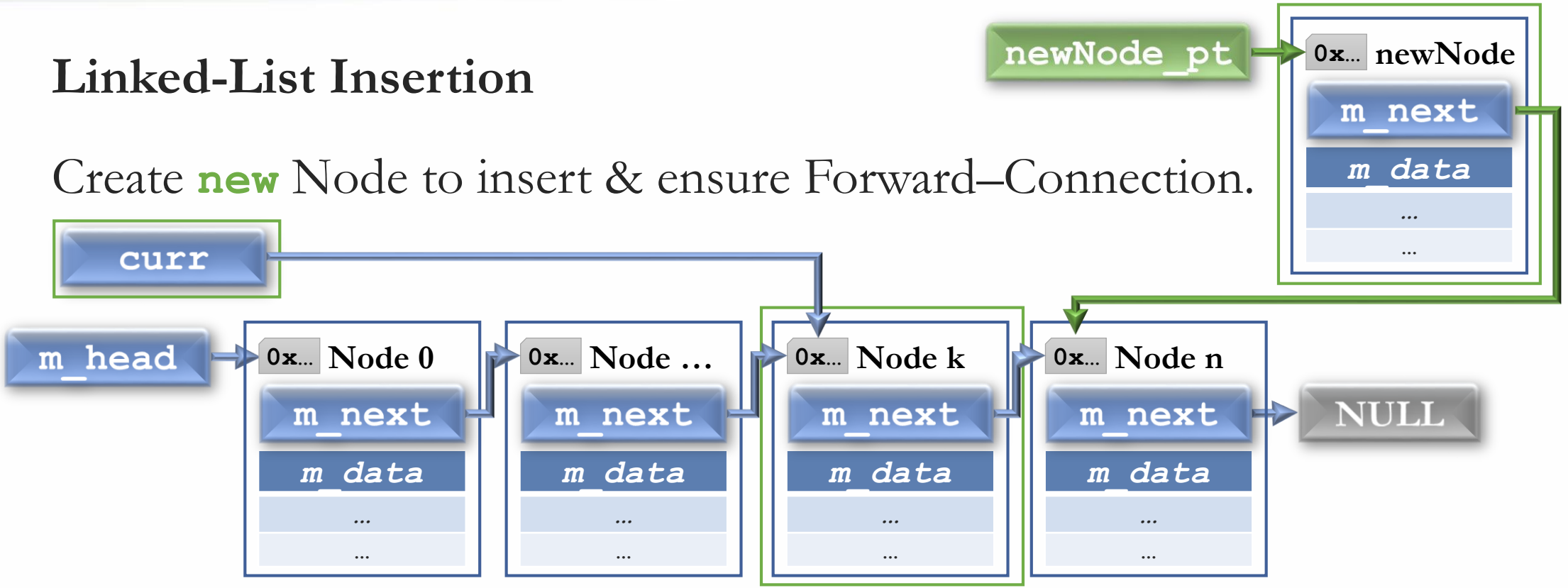



```
for (Node * curr = m_head; curr!=NULL; curr = curr->m_next)
  if ( curr->m_data == … ){
    Node * newNode_pt = new Node(data, curr->m_next);
    curr->m_next = newNode_pt;
  }
}
```

Forward – Linked List (*Node*-based)

Linked-List Insertion

Create **new** Node to insert & ensure Forward–Connection.

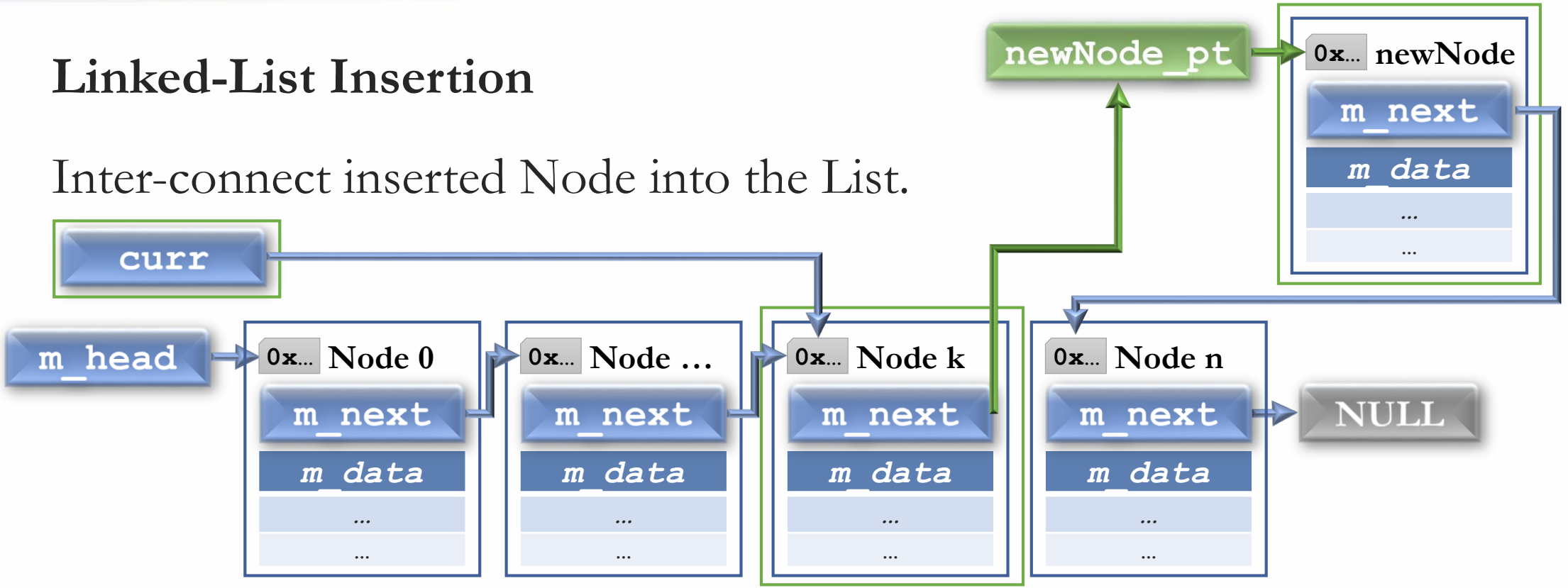



```
for (Node * curr = m_head; curr!=NULL; curr = curr->m_next)
  if ( curr->m_data == … ){
    Node * newNode_pt = new Node(data, curr->m_next);
    curr->m_next = newNode_pt;
  }
}
```

Forward – Linked List (*Node*-based)

Linked-List Insertion

Inter-connect inserted Node into the List.

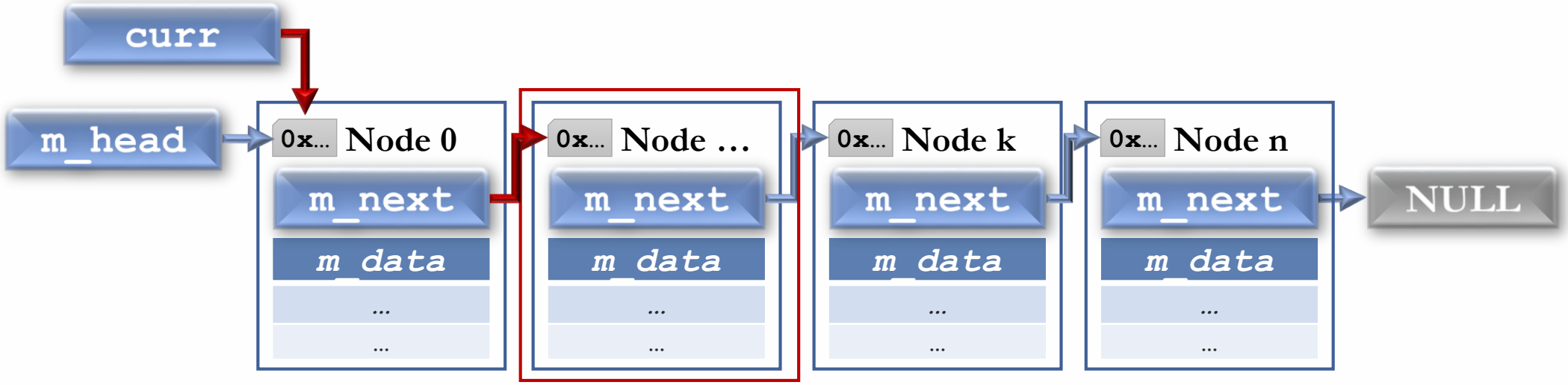



```
for (Node * curr = m_head; curr!=NULL; curr = curr->m_next)
  if ( curr->m_data == … ){
    Node * newNode_pt = new Node(data, curr->m_next);
    curr->m_next = newNode_pt;
  }
}
```

Forward – Linked List (*Node*-based)

Linked-List Node Deletion

Find the *predecessor* of the target Node to delete.

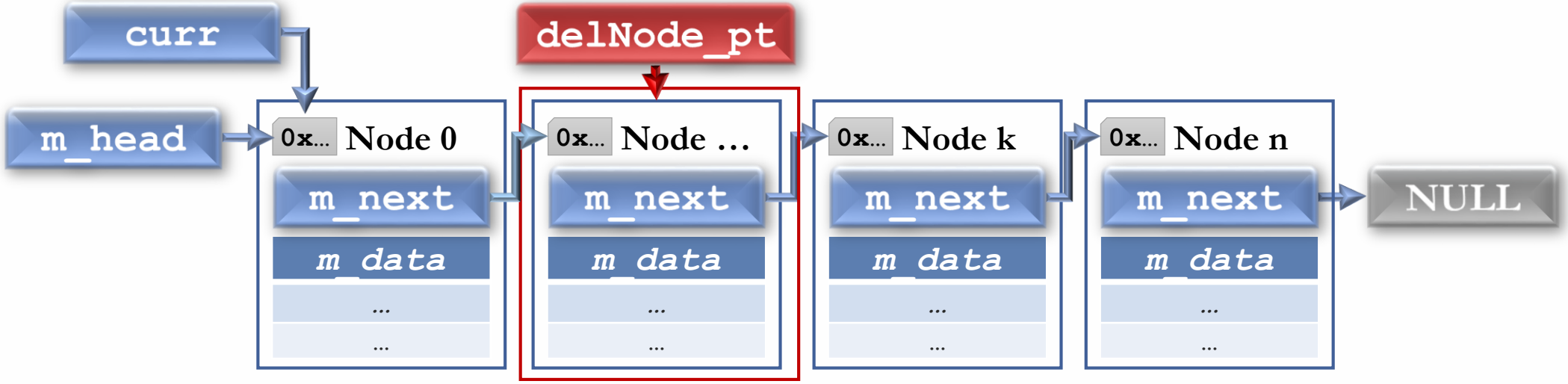



```
for (Node * curr = m_head; curr!=NULL; curr = curr->m_next)
  if ( curr->m_next && curr->m_next->m_data == … ){
    Node * delNode_Pt = curr->m_next;
    curr->m_next = delNode_pt->m_next;
    delete delNode_pt;
  }
```

Forward – Linked List (*Node*-based)

Linked-List Node Deletion

Memorize the Node to be **deleted**.

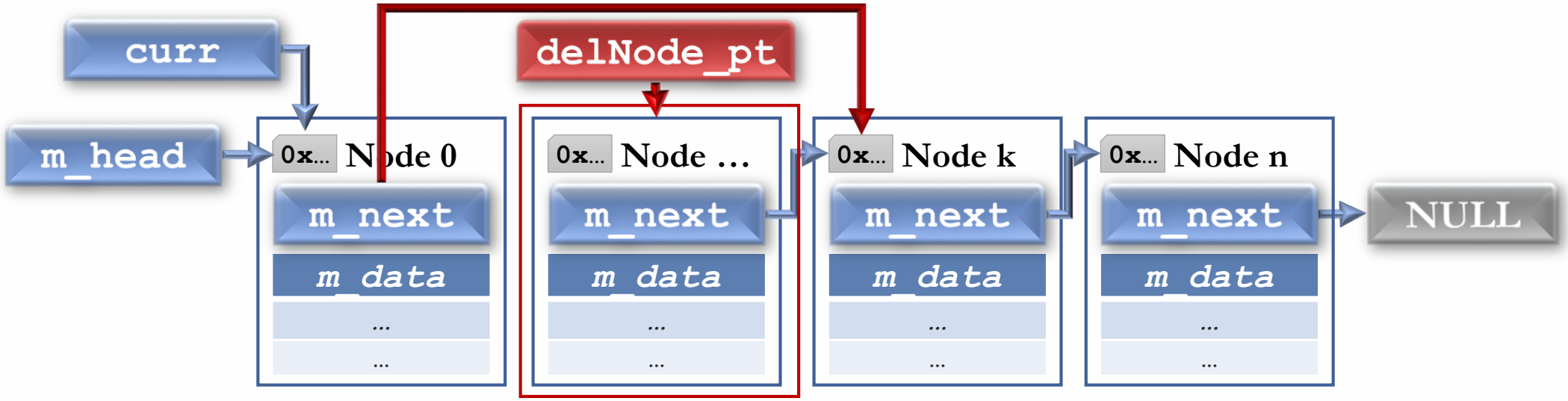



```
for (Node * curr = m_head; curr!=NULL; curr = curr->m_next)
  if ( curr->m_next && curr->m_next->m_data == … ){
    Node * delNode_Pt = curr->m_next;
    curr->m_next = delNode_pt->m_next;
    delete delNode_pt;
  }
```

Forward – Linked List (*Node*-based)

Linked-List Node Deletion

Change the Link of *predecessor* Node.

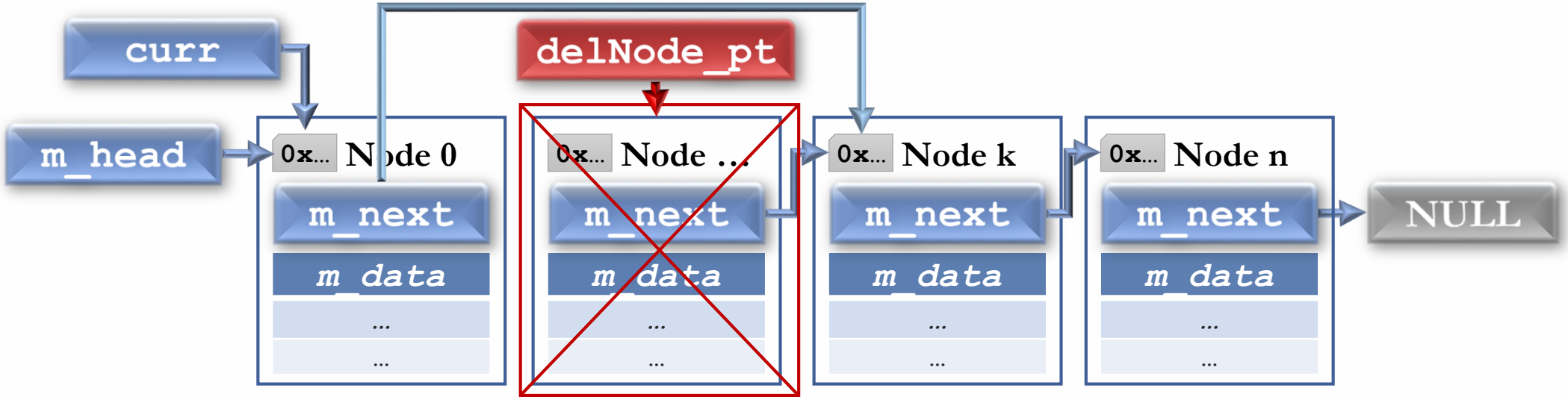



```
for (Node * curr = m_head; curr!=NULL; curr = curr->m_next)
  if ( curr->m_next && curr->m_next->m_data == … ){
    Node * delNode_Pt = curr->m_next;
    curr->m_next = delNode_pt->m_next;
    delete delNode_pt;
  }
```

Forward – Linked List (*Node*-based)

Linked-List Node Deletion

Deallocate dynamic memory of target Node.



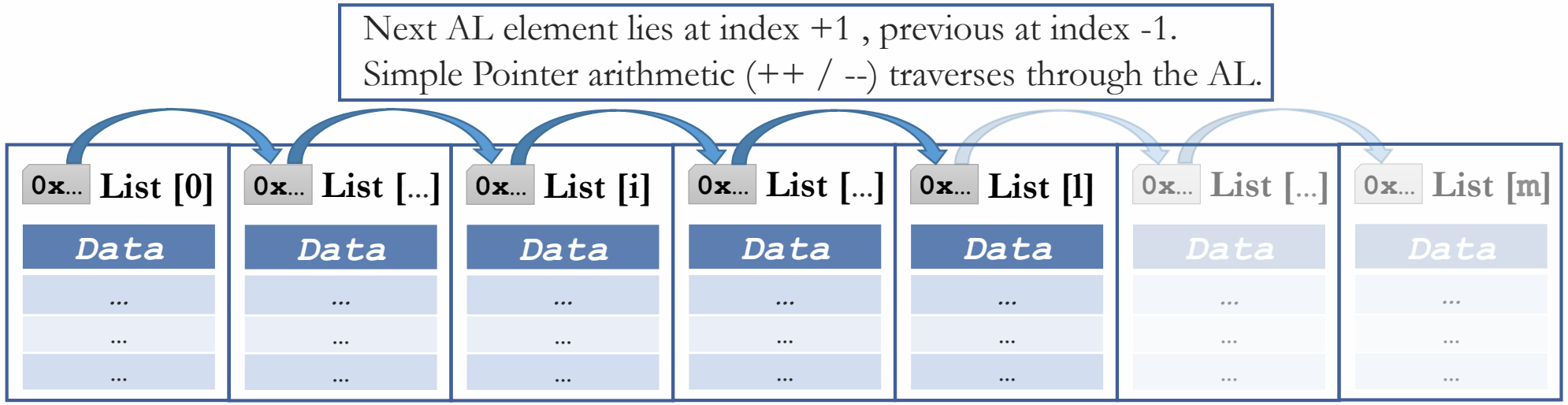

```
for (Node * curr = m_head; curr!=NULL; curr = curr->m_next)
  if ( curr->m_next && curr->m_next->m_data == … ){
    Node * delNode_Pt = curr->m_next;
    curr->m_next = delNode_pt->m_next;
    delete delNode_pt;
  }
```

Forward – Linked List (*Array*-based)

The Elements(s) (of the AL)

An Item of the AL is an element of the Dynamic Data Structure.

- Typically Data Object(s).
- Contiguously stored within an array.
- Data Organization implicitly determined by Array order.

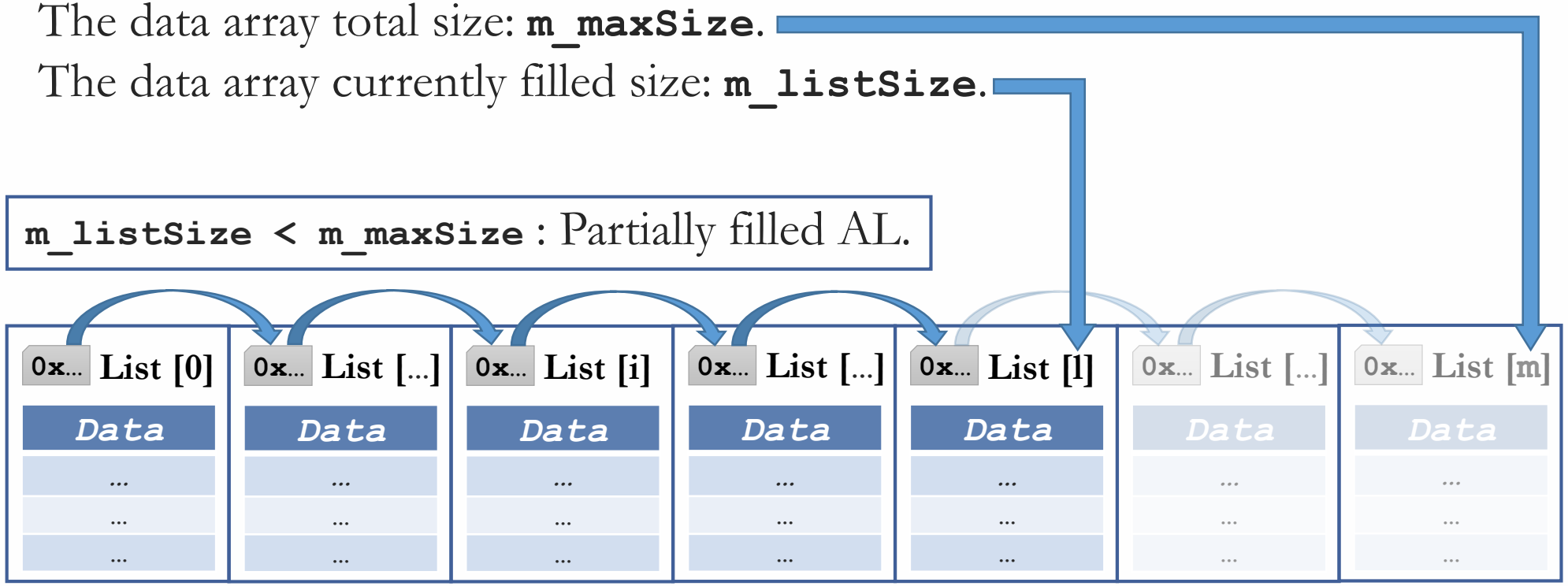


Forward – Linked List (*Array*-based)

The Structure (of the AL)

The AL can be variably filled. Use variables to track:

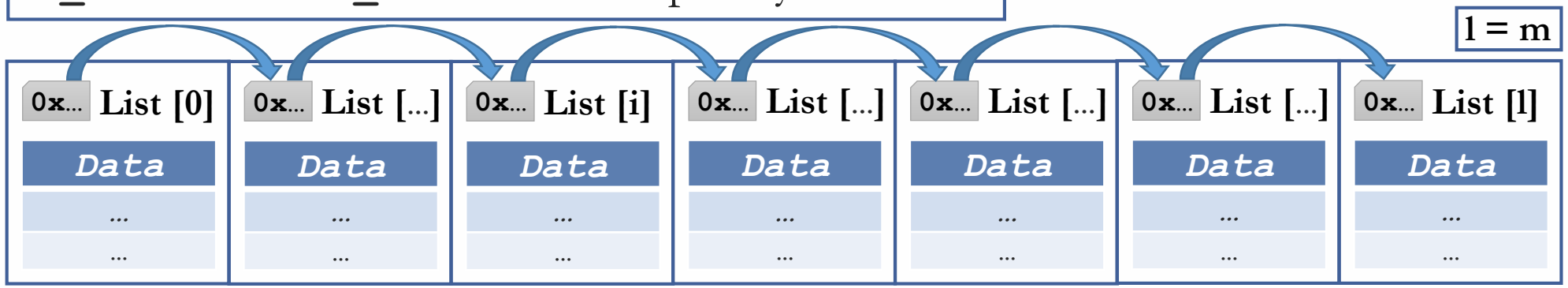
- The data array total size: **`m_maxSize`**.
- The data array currently filled size: **`m_listSize`**.



Forward – Linked List (*Array*-based)

The Structure (of the AL)

`m_listSize = m_maxSize` : Completely filled AL.



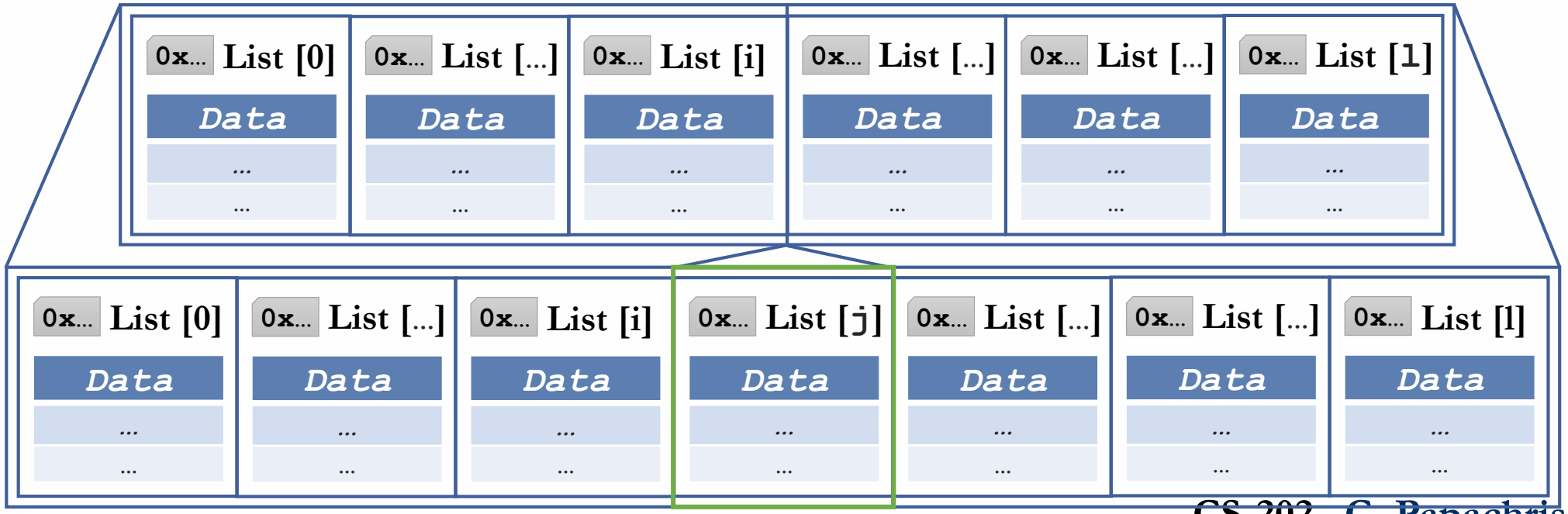
`m_listSize = 0` : Completely empty AL.

Forward – Linked List (*Array*-based)

Insertion (into the AL)

Need to "*Shuffle*" – upwards.

- If partially filled and new data "fits", no re-allocation.
- Otherwise, allocate new data array, copy contents, delete old data array.

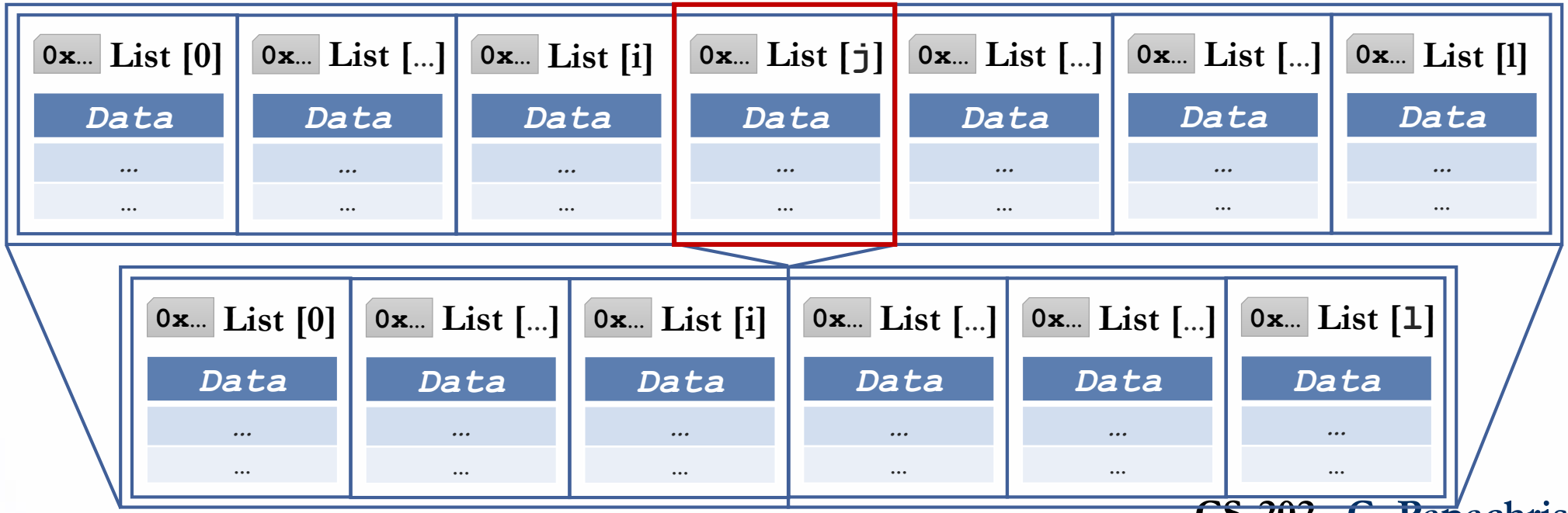


Forward – Linked List (*Array*-based)

Deletion (from the AL)

Need to "*Shuffle*" – downwards.

- Copy over data from right to left, overwriting array elements.
- Update necessary auxiliary variables – such as: **`m_listSize`**, **`m_maxSize`**



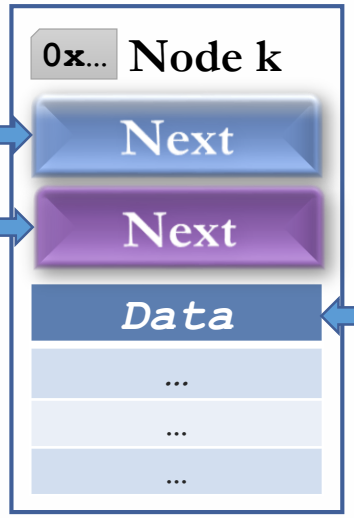
Doubly – Linked List Node(s)

The Node(s) (of the DLL)

A Node of the DLL is an element of the Dynamic Data Structure.

- Typically a `class` encapsulating Data Object(s).
- Holds Data & Pointer to associate "Next" Node as well as "Previous" Node in the Doubly–Linked List.
- List implementation has to Access & Mutate it.

Necessary: Maintain association(s) to other LL Nodes.
Can:
- Point to another Node.
- Be `NULL`.



Stored Data can be simple data types (`int`, `double`, …) or complex ones (`class`es/ `struct`s).

Provide Get/Set methods or give class List direct access to class Node members.

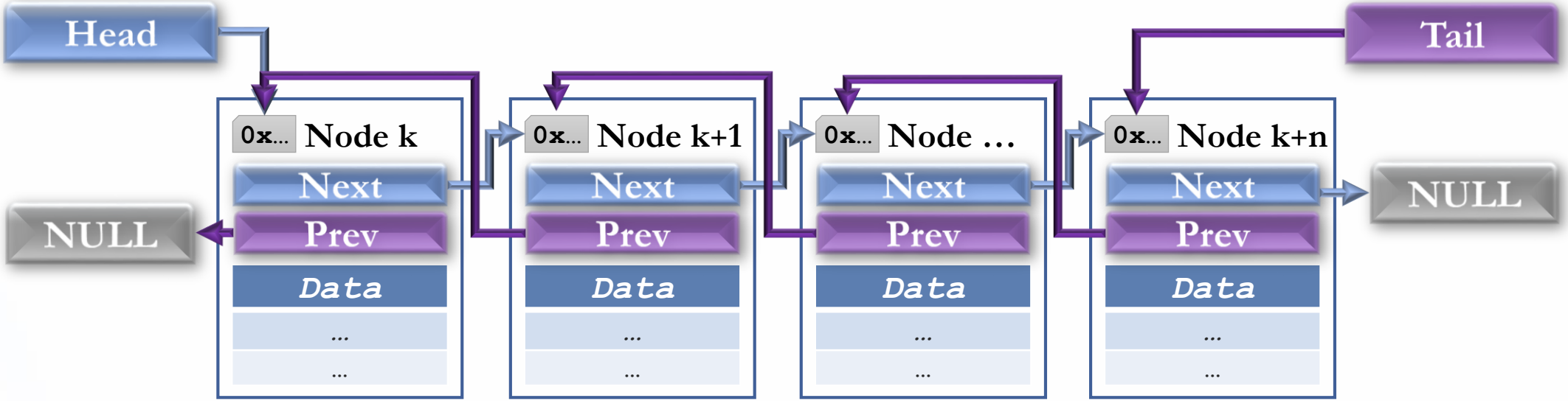
```
class Node {
 public:
 // ctor(s)
 // dtor
 // get - set methods
 …
 friend class List;
 …
 private:
 int m_data;
 Node * m_next;
 Node * m_prev;
};
```

Doubly – Linked List(s)

The Doubly Linked-List

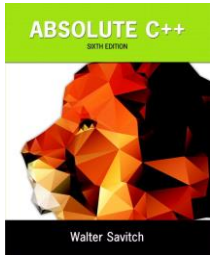
An example Doubly Linked-List:

- Each Node additionally contains Address of "Previous" Node in the Linked-List.
- Linked-List has a Head & a Tail pointer to respective "First" & "Last" Node.



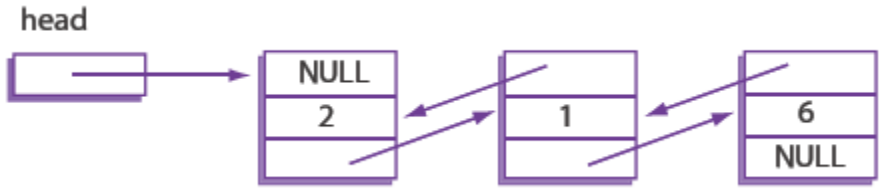
Doubly Linked Lists

- Single linked list
  - Can only follow links in one direction
- Doubly Linked List
  - Links to the next node and another link to the previous node
  - Can follow links in either direction
  - NULL signifies the beginning and end of the list
  - Can make some operations easier, e.g. deletion since we don't need to search the list to find the node before the one we want to remove



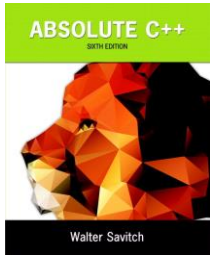
Modified Slides from:
Absolute C++ 6th Edition
Walter Savitch

Doubly Linked List Node




```
class DoublyLinkedIntNode {
  public:
    DoublyLinkedIntNode ( ) {}
    DoublyLinkedIntNode (int theData,
                         DoublyLinkedIntNode * previous,
                         DoublyLinkedIntNode * next)
        : m_data(theData),
          m_nextLink(next),
          m_previousLink(previous) {}

    DoublyLinkedIntNode * getNextLink( ) const { return m_nextLink; }
    DoublyLinkedIntNode * getPreviousLink( ) const { return m_previousLink; }
    int getData( ) const { return m_data; }
    void setData(int theData){ data = theData; }
    void setNextLink(DoublyLinkedIntNode * pointer) { m_nextLink = pointer; }
    void setPreviousLink(DoublyLinkedIntNode* pointer) { m_previousLink = pointer; }
  private:
    int m_data;
    DoublyLinkedIntNode * m_nextLink;
    DoublyLinkedIntNode * m_previousLink;
};
typedef DoublyLinkedIntNode * DoublyLinkedIntNodePtr
```

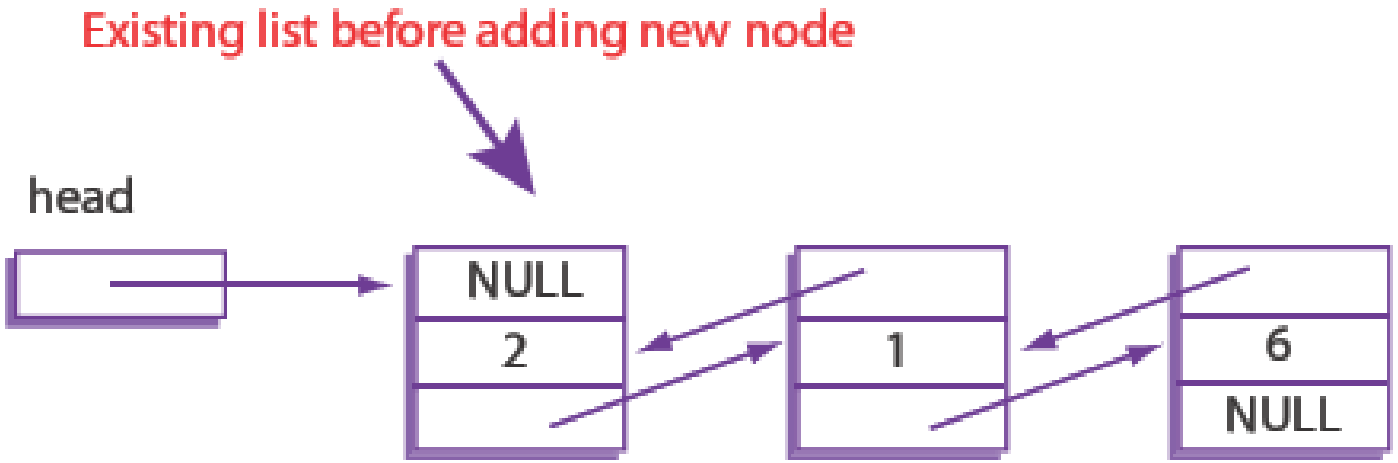


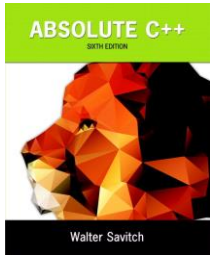
Modified Slides from:

Absolute C++ 6th Edition

Walter Savitch

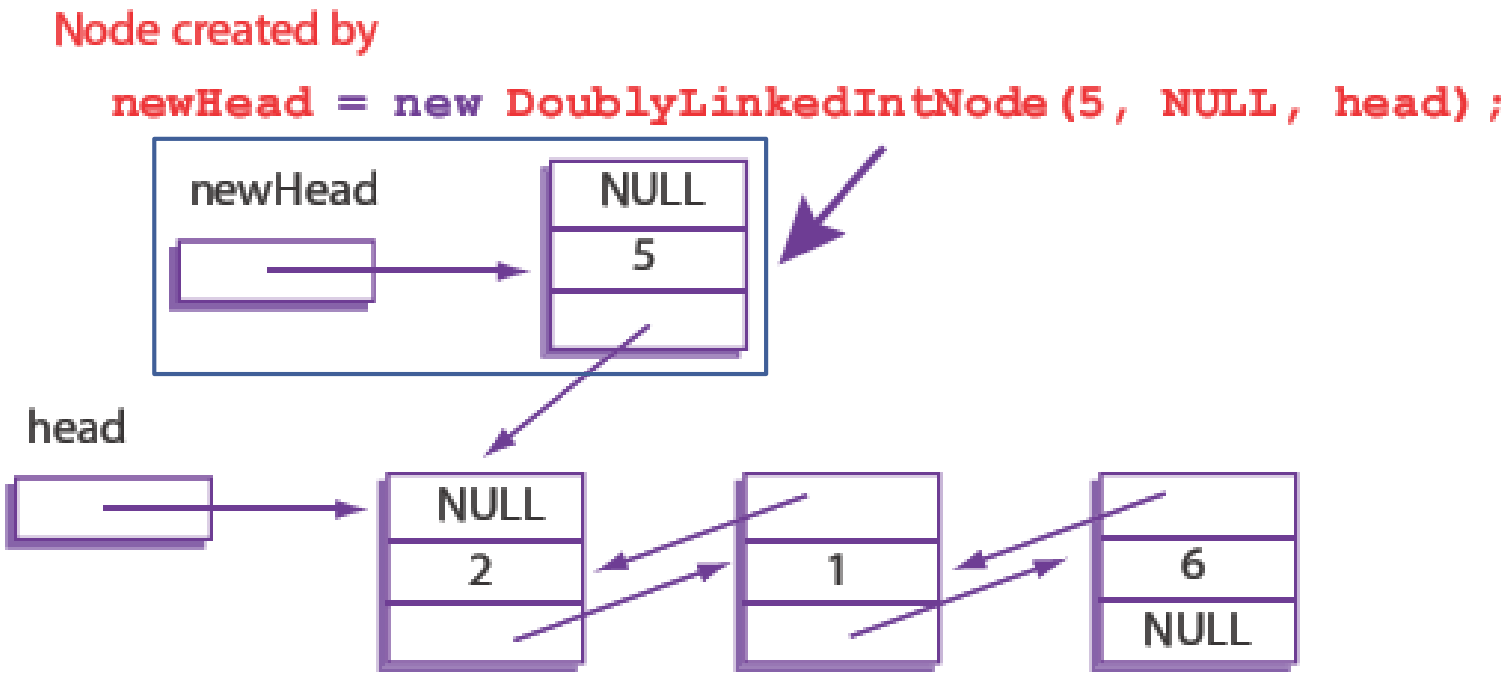
Adding a Node to the Front of a Doubly Linked List (1 of 4)

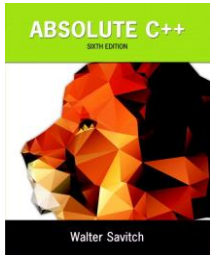




Modified Slides from:
Absolute C++ 6th Edition
Walter Savitch

Adding a Node to the Front of a Doubly Linked List (2 of 4)



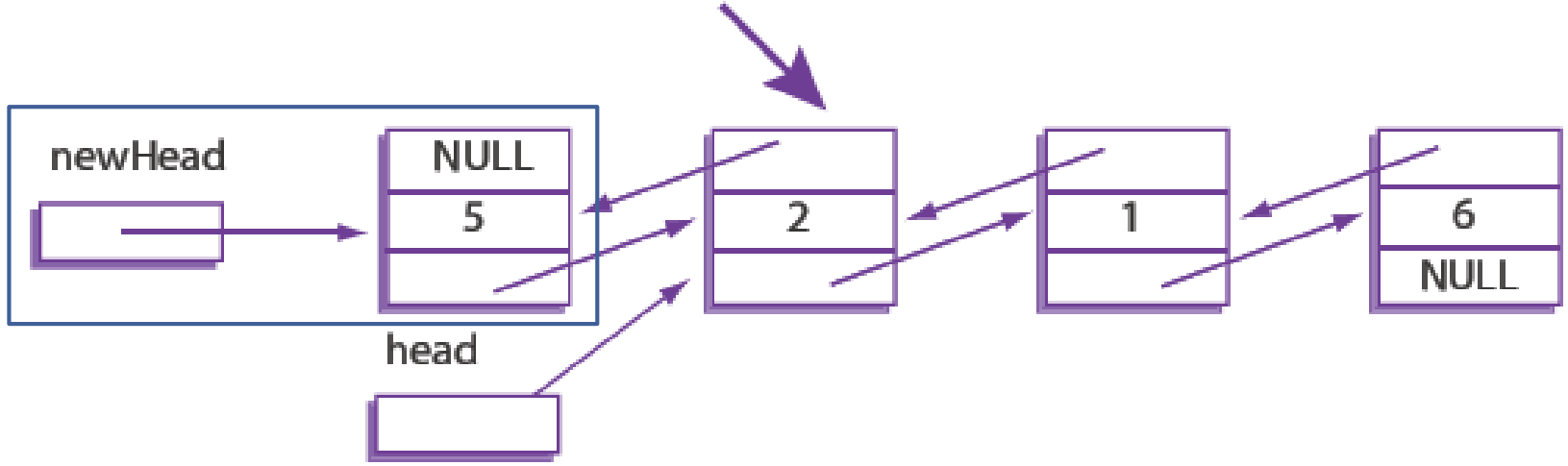


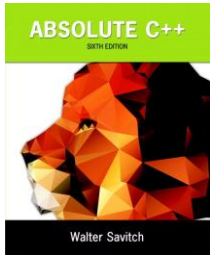
Modified Slides from:
Absolute C++ 6th Edition
Walter Savitch

Adding a Node to the Front of a Doubly Linked List (3 of 4)

Set the previous link of the original head node

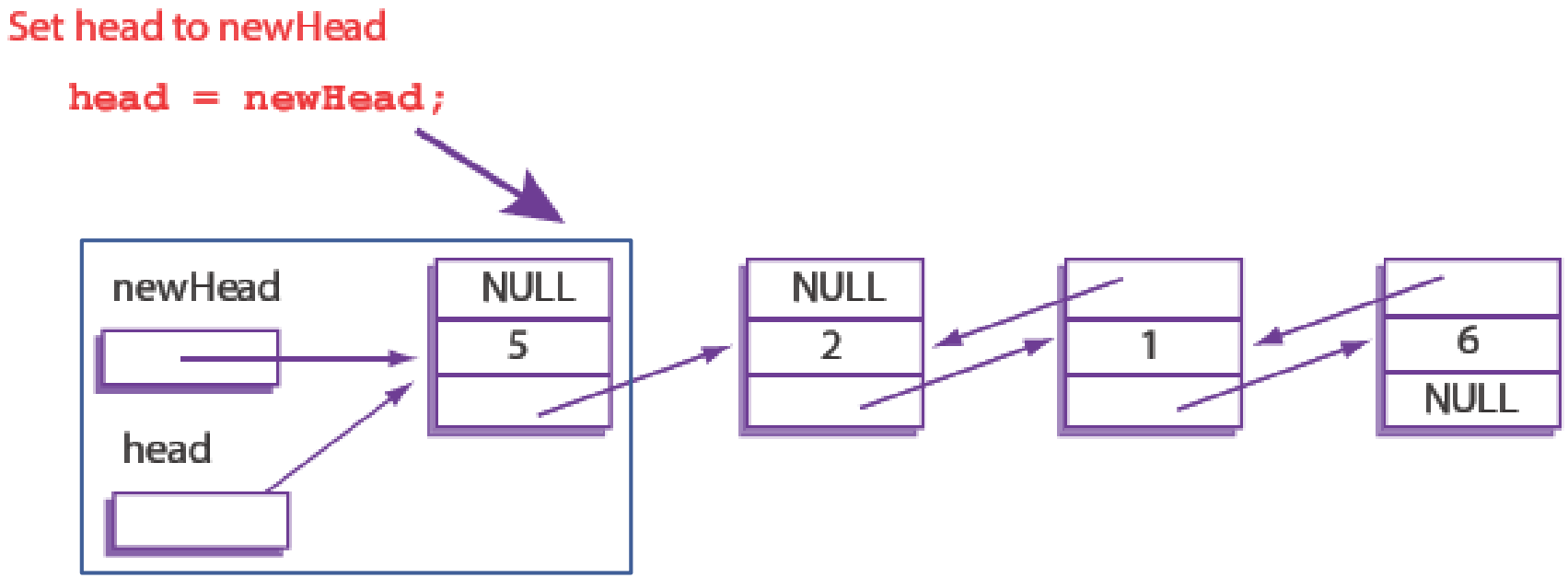
```
head->setPreviousNode(newHead);
```

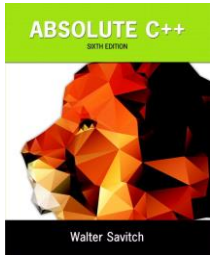




Modified Slides from:
Absolute C++ 6th Edition
Walter Savitch

Adding a Node to the Front of a Doubly Linked List (4 of 4)

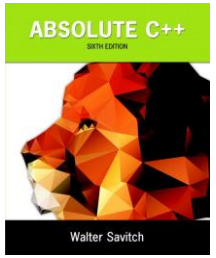




Modified Slides from:
Absolute C++ 6th Edition
Walter Savitch

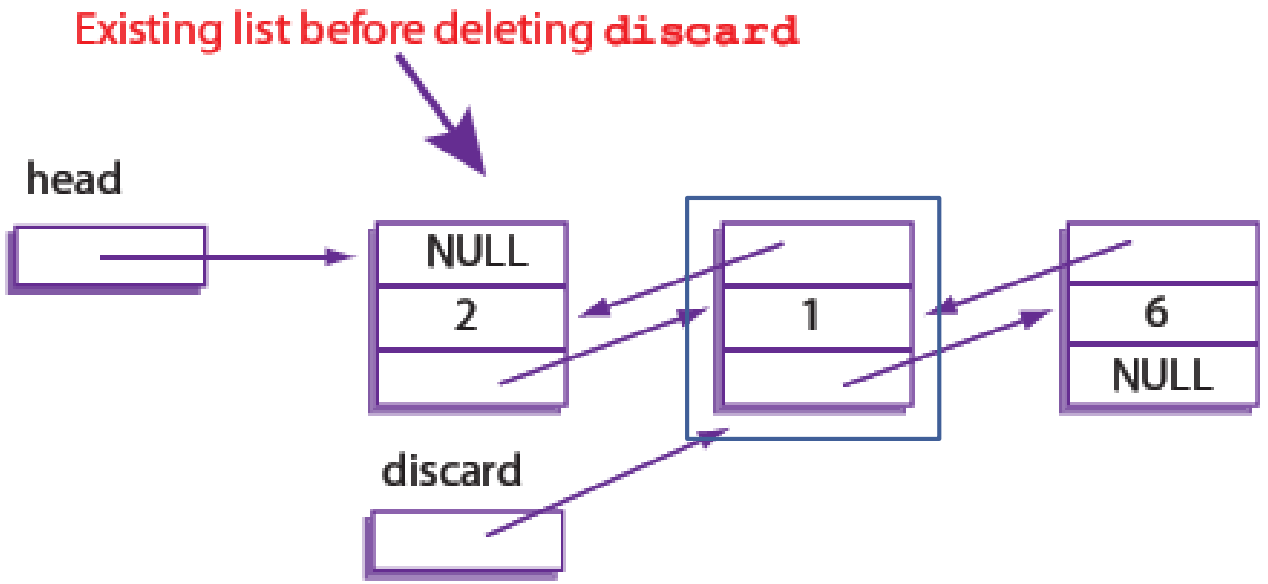
Deleting a Node from a Doubly Linked List

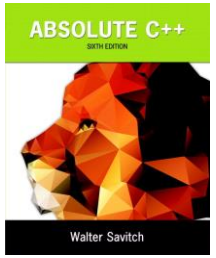
- Removing a node requires updating references on both sides of the node we wish to delete
- Thanks to the backward link we do not need a separate variable to keep track of the previous node in the list like we did for the singly linked list
  - Can access via node->previous



Modified Slides from:
Absolute C++ 6th Edition
Walter Savitch

Deleting a Node from a Doubly Linked List (1 of 4)



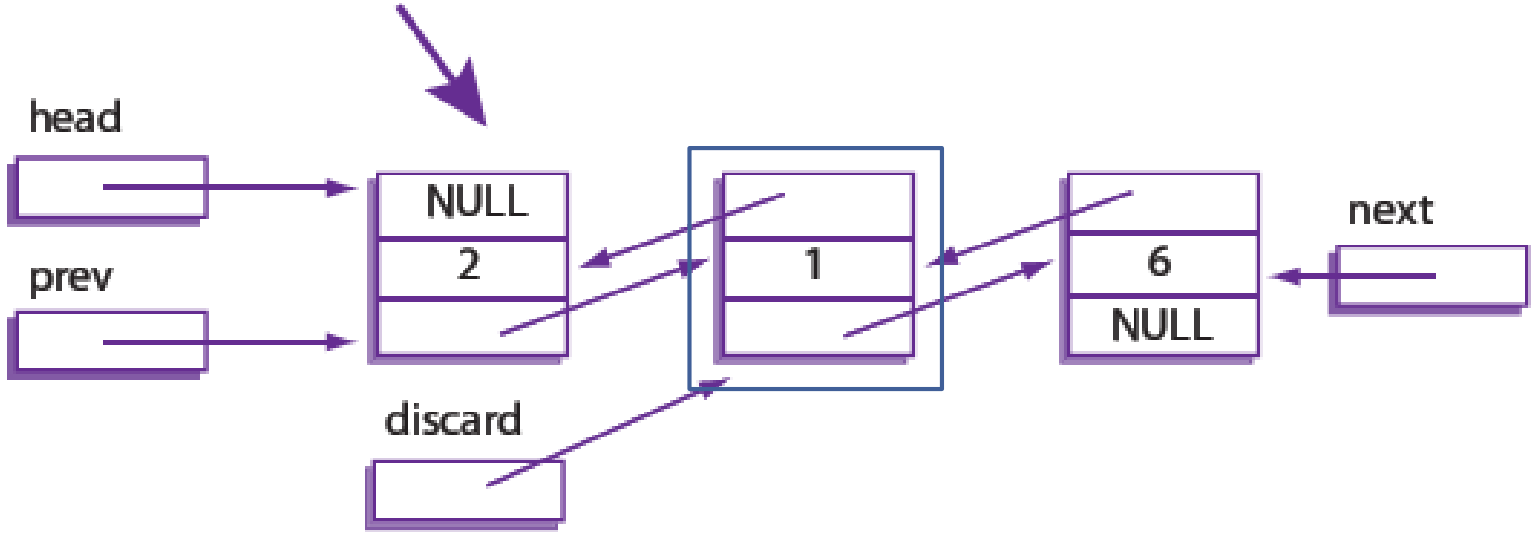


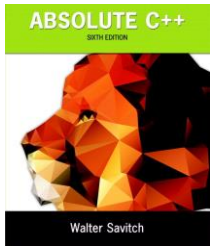
Modified Slides from:
Absolute C++ 6th Edition
Walter Savitch

Deleting a Node from a Doubly Linked List (2 of 4)

Set pointers to the previous and next nodes

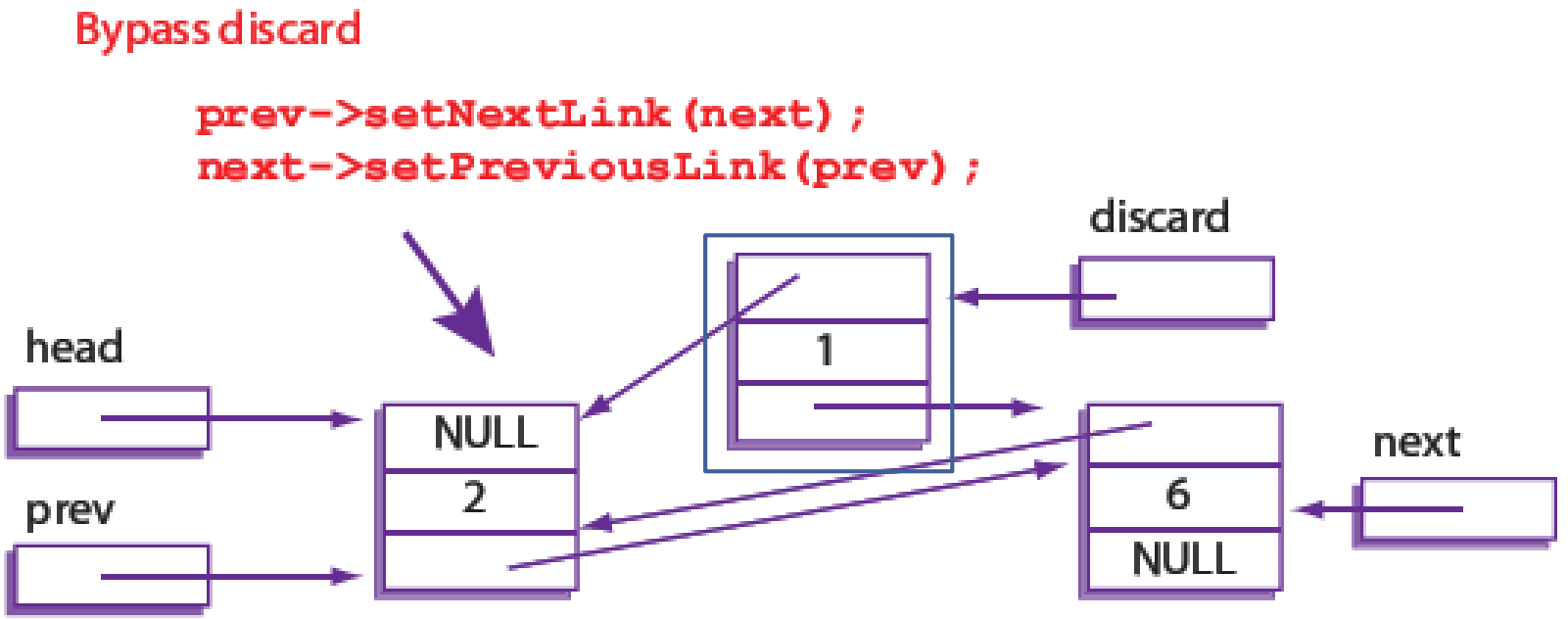
```
DoublyLinkedIntNodePtr prev = discard->getPreviousLink( );
DoublyLinkedIntNodePtr next = discard->getNextLink( );
```

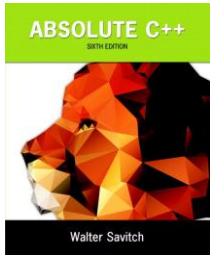




Modified Slides from:
Absolute C++ 6th Edition
Walter Savitch

Deleting a Node from a Doubly Linked List (3 of 4)



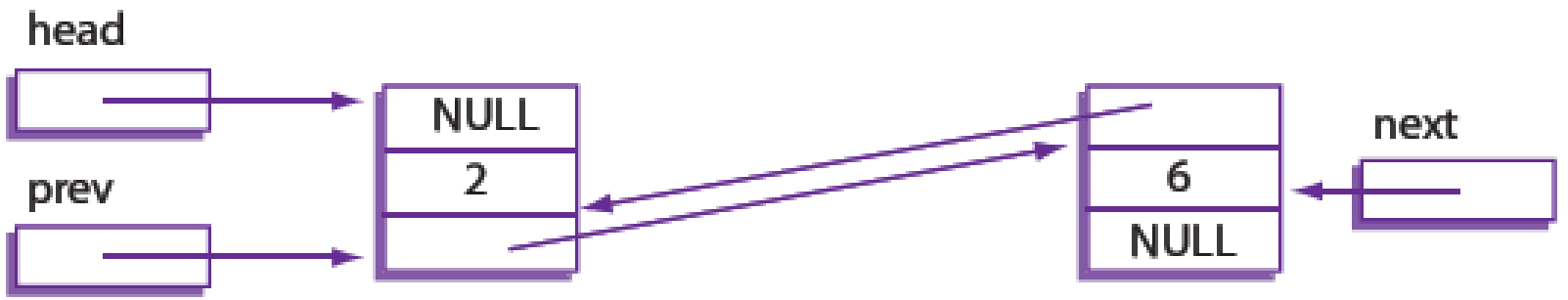


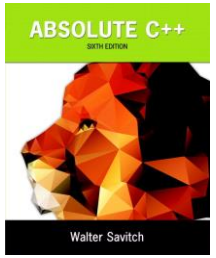
Modified Slides from:
Absolute C++ 6th Edition
Walter Savitch

Deleting a Node from a Doubly Linked List (4 of 4)

Delete discard

```
delete discard;
```





Modified Slides from:
Absolute C++ 6th Edition
Walter Savitch

CS-202

Time for Questions !