CS-202

Dynamic Data Structures (Pt.1)

C. Papachristos
Autonomous Robots Lab
University of Nevada, Reno

Course Week

Course , Projects , Labs:

Monday	Tuesday	Wednesday	Thursday	Friday
			Lab (4 Sections)	
	CLASS	RL – Session	CLASS	
PASS Session	PASS Session	**Project DEADLINE**	***NEW Project***	

Your *Next* Project ?

7th Project Deadline was this Wednesday 4/4.

- NO Project accepted past the 24-hrs delayed extension (@ 20% grade penalty).
- Send what you have in time!

Today's Topics

Dynamic Data Structures

Array(s) *vs* Dynamically Growing/Shrinking Data Structures

Linked-List(s)

- Basics
- LL Node(s)
- Traversal
- Insertion
- Deletion
- Search

Dynamic Data Structures

Data Structures and Dynamic Memory

A Class wrapping a Data Structure:

- Implemented with $\boxed{\text{Dynamic Memory.}}$
- A simple container which:

  1) Holds a number of **`m_size`** elements of some Struct or Class type of data (in this example, named ***`Data`***).

  2),3) Can get instantiated empty/with a given initial size with all elements set to a specific value/as a copy of another structure, and properly deallocates its data when destroyed.

  4) Can set itself (assignment) to be a copy of another.

  5) Can resize itself to allow holding a specific total number of elements.

  6) Allows access to its elements for getting/modifying their value.

```
class DataStruct {
public:
DataStruct();                                          2)
DataStruct(size_t count,
           const Data & value);
DataStruct(const DataStruct & other);
~DataStruct();                                         3)
DataStruct& operator=(const                            4)
                 DataStruct & other);
void resize(size_t size);                              5)
Data & operator[](size_t pos);                         6)
const Data & operator[](size_t pos) const;
private:
 Data * m_data;                                        1)
 size_t m_size;
};
```

Data Structures and Dynamic Memory

A Class wrapping a Data Structure:

- Implemented with Dynamic Memory.
- Size can be modified in runtime, which is an important new ability compared to statically allocated arrays.

But ! Have to delete and re-allocate entire memory portion !

```
DataStruct& DataStruct::operator=(const DataStruct& other){
 if (this != &other){  //check for self-assignment
   delete [] m_data;
   …
     try{
       m_size = new double[other.m_size];
       for(…){ m_data[i] = other.m_data[i]; }
     } catch(...) { /* handle exception */ }
   …
 }
 return *this
}
```

```
class DataStruct {
public:
DataStruct();
DataStruct(size_t count,
           const Data & value);
DataStruct(const DataStruct & other);
~DataStruct();
DataStruct& operator=(const
                 DataStruct & other);
void resize(size_t size);

Data &operator[](size_t pos);
const Data &operator[](size_t pos) const;
private:
 Data * m_data;
 size_t m_size;
};
```

Data Structures and Dynamic Memory

A Class wrapping a Data Structure:

- Implemented with Dynamic Memory.
- Size can be modified in runtime, which is an important new ability compared to statically allocated arrays.

But ! Even for “simpler” operations …

```
DataStruct& DataStruct::resize(int size){
 int origSize = m_size;
 double* origData = m_data;
 try{
  if (size>0){
   m_size = size;
   m_data = new double[m_size];
   int minSize = m_size<=origSize ? m_size : origSize;
   for(int i=0;i<minSize;++i){ *m_data++ = *origData++; }
   delete [] origData;
  } else { m_size = 0; delete [] m_data; m_data = NULL; }
 } catch(...) { /* handle exception */ }
 return *this
}
```

Have to delete, reallocate, copy over for a simple resizing.

```
class DataStruct {
public:
DataStruct();
DataStruct(size_t count,
           const Data & value);
DataStruct(const DataStruct & other);
~DataStruct();
DataStruct& operator=(const
                  DataStruct & other);
void resize(size_t size);
Data &operator[](size_t pos);
const Data & operator[](size_t pos) const;
private:
 Data * m_data;
 size_t m_size;
};
```

Dynamic Data Structures

Data Structures and Dynamic Memory

A Class wrapping a Data Structure:

- Implemented with Dynamic Memory.
- Size can be modified in runtime, which is an important new ability compared to statically allocated arrays.

But ! And overly complicating others …

```
void DataStruct::push(const Data & value){
 resize(m_size + 1);
 if (m_size){ m_data[m_size] = value; }
}
void DataStruct::pop(const Data& value){
 resize(m_size - 1);
}
void DataStruct::insert(size_t pos, const Data & value){
 /* ? works in-between, have to rework with temporary vars ?*/
}
void DataStruct::erase(size_t pos){
 /* ? works in-between, have to rework with temporary vars ?*/
}
```

Even these cases incur the cost of re-allocating / copying (almost) the entire data range !

```
class DataStruct {
 public:
 DataStruct();
 DataStruct(size_t count, const Data & value);
 DataStruct(const DataStruct & other);
 ~DataStruct();
 DataStruct& operator=(const
             DataStruct& other);
 void resize(int size);
 void push(const Data & value);
 void pop(const Data & value);
 void insert(size_t pos, const Data & value);
 void erase(size_t pos);
 Data & operator[](size_tt pos);
 const Data & operator[](size_t pos) const;
 private:
 Data * m_data;
 size_t m_size;
};
```

Dynamic Data Structures

Dynamically growing (/shrinking) Data Structures

An organization of data in memory that:

- Flexibly grows/shrinks.
- Flexibly handles relationships between contained data:
  (Single or multiple relationships, connectivity, etc.)

The Linked-List (LL):

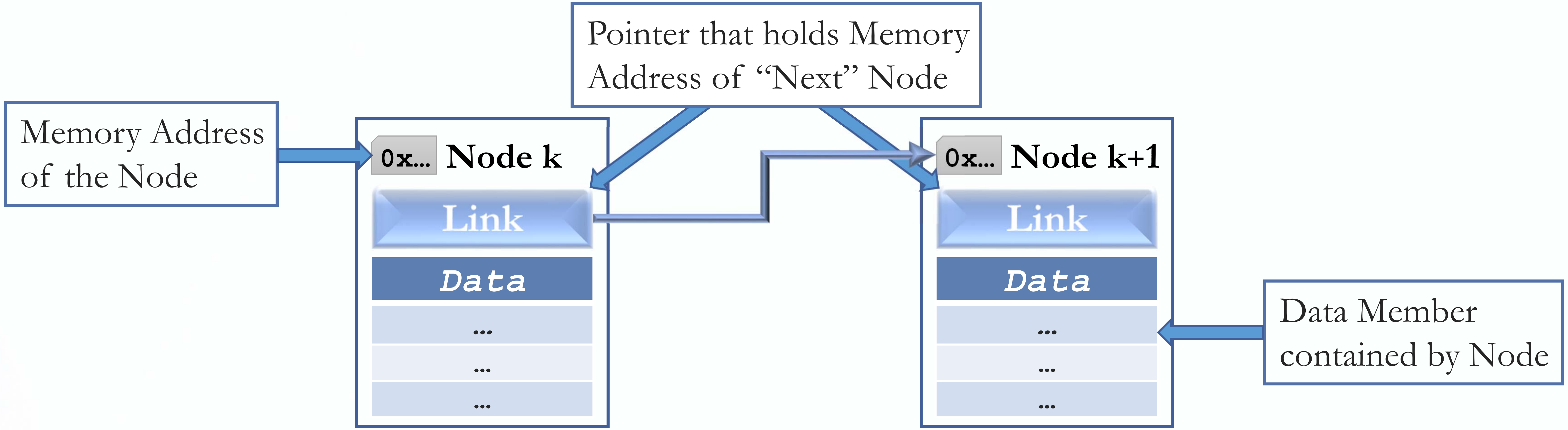
- Allows "easy" (i.e. computationally efficient) insertion and deletion.
- A simple Dynamic Data Structure (DDS).

Linked-List(s)

Dynamically Data Structure paradigm

A container of elements that are represented as Nodes:

- Each Node contains Data.
- Each Node contains Address of "Next" Node in the Linked-List.

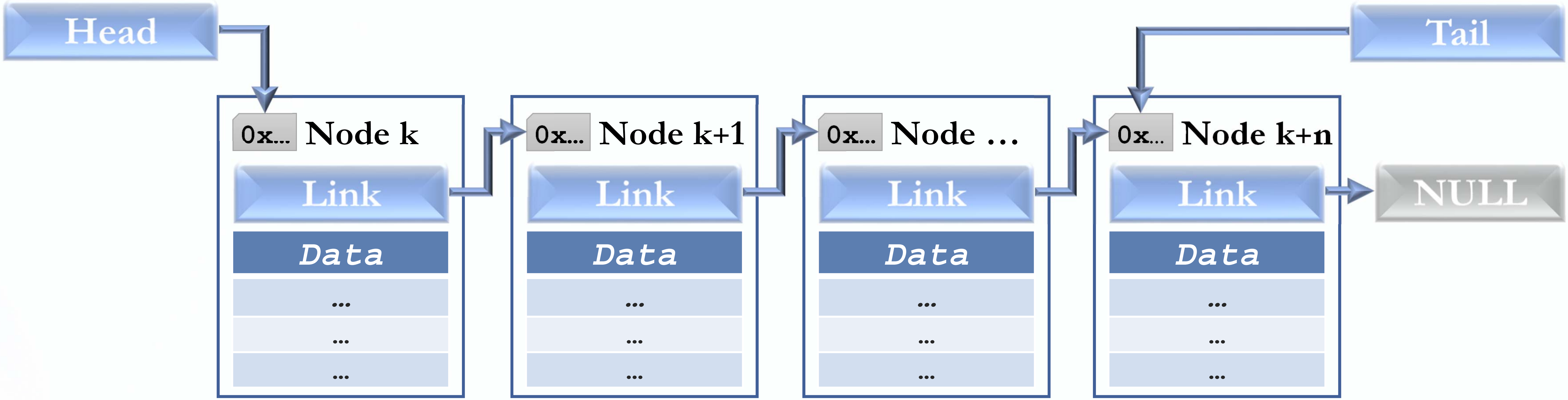


Linked-List(s)

Dynamically Data Structure paradigm

Example Linked-List

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



Linked-List Basics

Linked lists and Arrays are similar since they both store collections of data.

The Array's distinctive and advantageous features are derived from its nature, its strategy of allocating the memory for all its elements in one contiguous block of memory.

Linked-Lists use an entirely different strategy:

- Linked-Lists allocate memory for each element separately.
- Linked-Lists allocate memory for an element only when necessary.

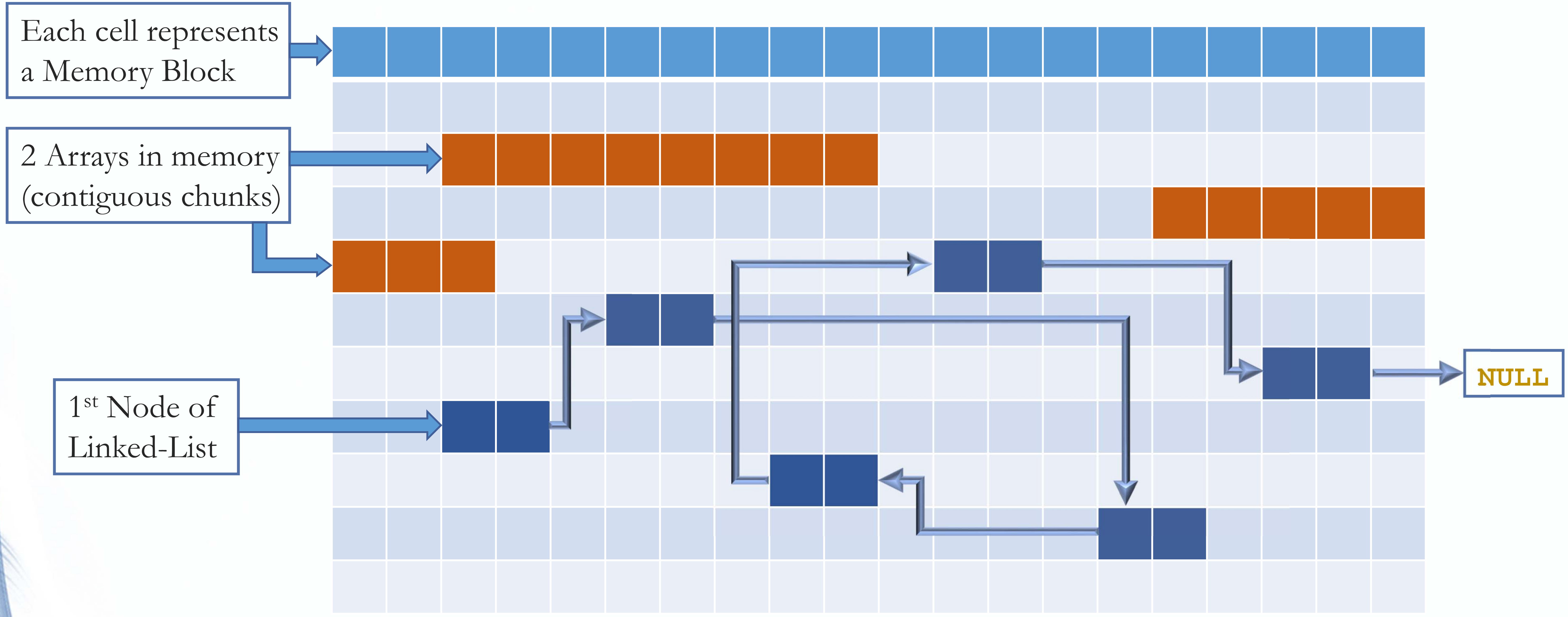
Linked-List Utility

Array disadvantages:

- The size of the array is (relatively) fixed, and allocation is on the basis of "seems large enough".
- Requires a contiguous block of memory.
- Just a small amount of useful elements in it and the remaining space wasted.
- If more elements than the declared size are needed code needs reconsideration.
- A number of operations including insertion in the middle of an array, element deletion, sorting, take time.

Linked-List(s)

Memory Representation



Linked-List(s)

Linked-List Utility

Appropriate to use LLs when the number of data elements to be represented in the Data Structure at the same time is unpredictable:

- Linked lists are Dynamic Data Structures, so the length of a LL can grow or shrink as necessary.
- Each Node does not necessarily follow the previous one in physical memory ordering.
- Linked-Lists can be maintained in sorted order by inserting or deleting an element at the proper point in the List.

Linked-List(s)

Linked-List Utility

Advantages:

Easy resizing (all the time too).
Easy insertion and deletion anywhere in the Linked-List.
Contiguous storage required on the single-Node level only.

Disadvantages:

Can’t use simple pointer-offset indexing to access.
Memory management required.
Additional memory required to store Pointer to “Next” Node.

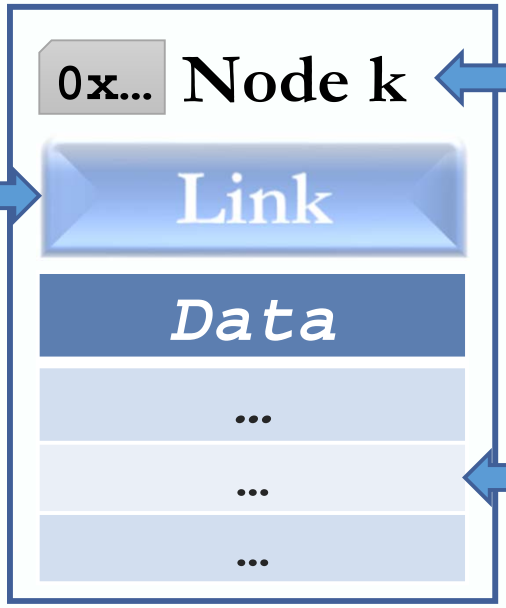
The Node(s) (of the LL)

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

- Often represented as a `class` (as a `struct` too).
- Can have other characteristic values (a "name", "state", etc.).
- Should have Data, which are the data of DDS.
- Has to have a Pointer to associate "Next" Node in the LL.

```
class Node {
 public:
 // ctor(s)
 // dtor
 // get - set methods
 …
 private:
 char * m_name
 int m_data;
 Node * m_link;
};
```

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



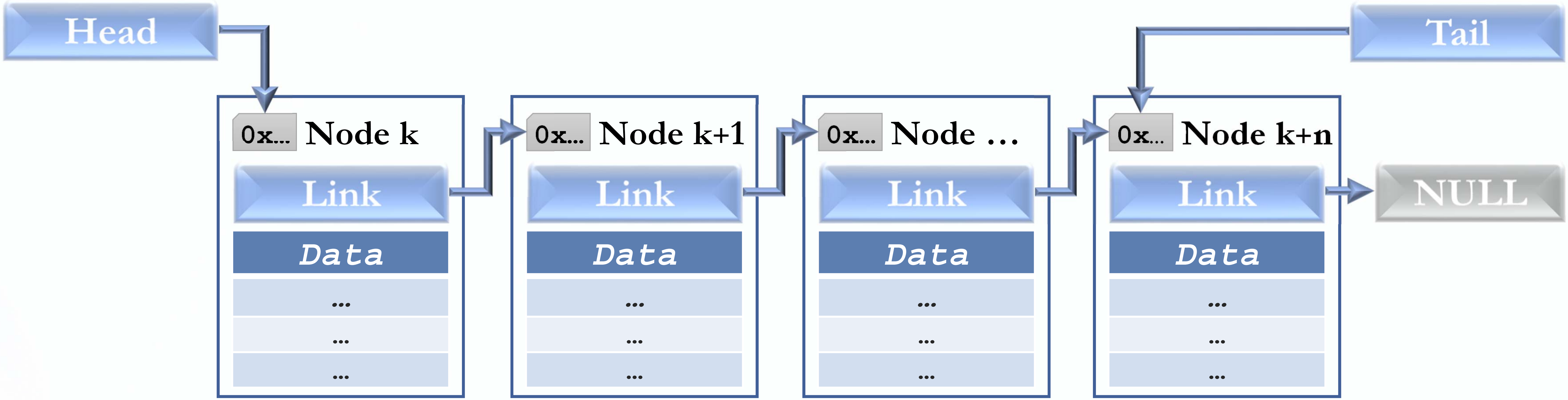
Name and other characteristic values are optional.

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

The Linked-List

An example Linked-List:

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



The Head Pointer (of the LL)

The previously illustrated box labeled "head" is not a Node:

```
Node * head;
```

It is a simple Pointer to a Node, set to point to the first Node in LL.

- Head used to "maintain" the start of the LL.
- It is also usable as an argument to functions:

Examples:

```
(*head).m_data = 12;
head->m_data = 12;
cin >> head->m_name;
```

Set **`count`** member of LL's starting Node (through dereferencing or direct arrow notation).

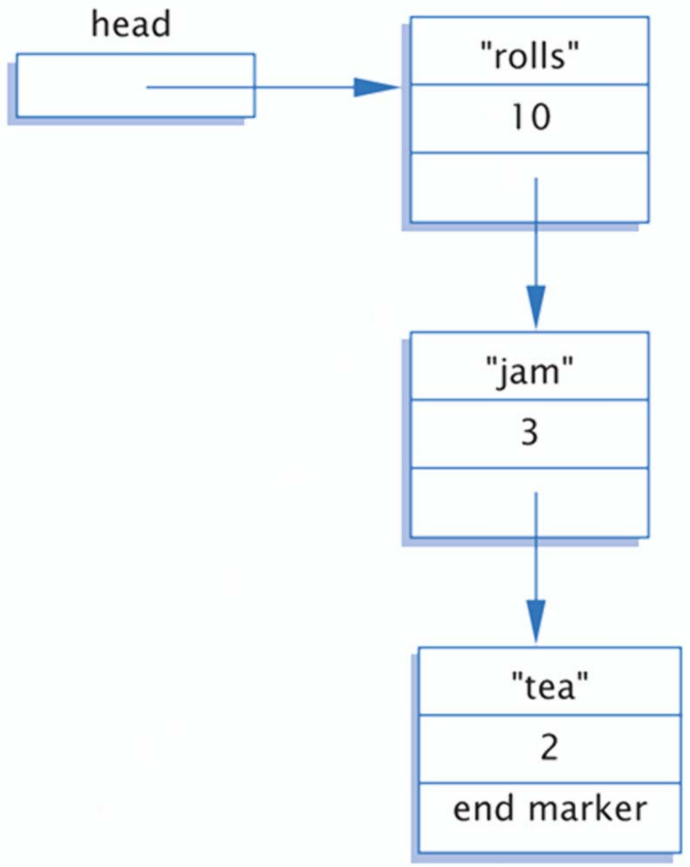
Directly read-in to a member of LL's starting Node through **`head`** Pointer.

Linked-List(s)

Nodes and Pointers

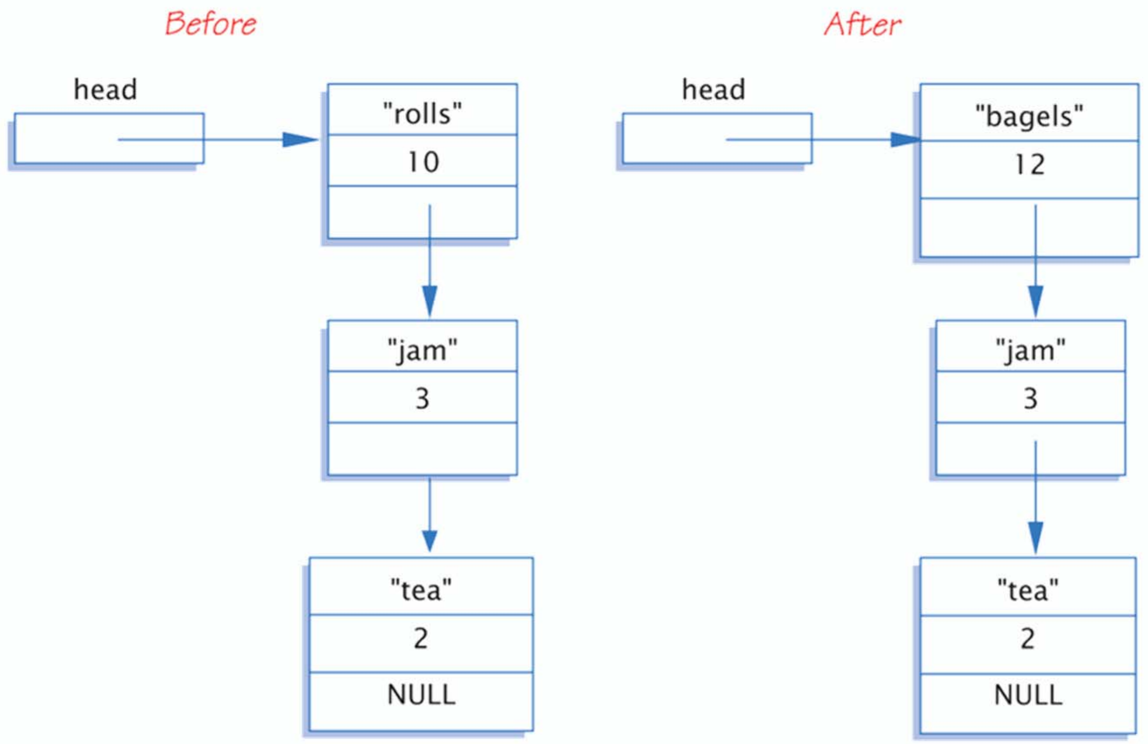
Visualization of:

- A Linked-List DDS.



- Accessing a Node (the "First" Node) through the Head Pointer.

```
head->m_data = 12;
strcpy (head->m_name , "bagels");
```



Linked-List(s)

End Marker (of the LL)

The "Last" Node in the Linked-List points to `NULL`.

- By convention, indicates "No further links after this Node".
- It provides end-marker functionality similarly to how partially-filled arrays are used (but without wasting extra space).

Overall:

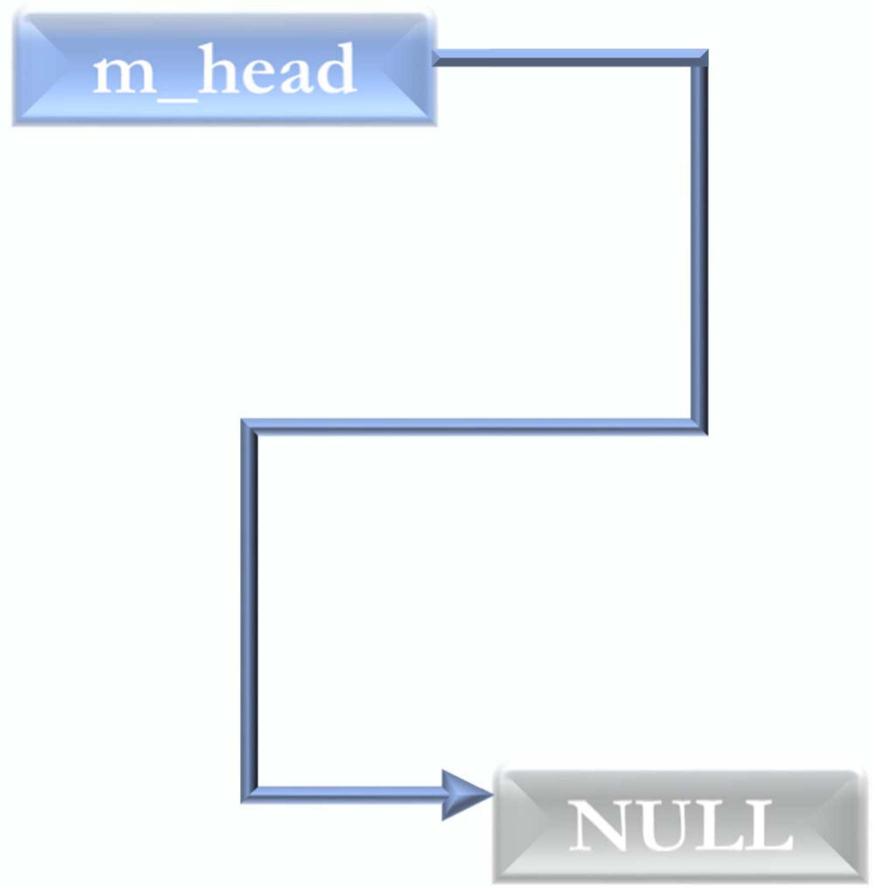
- Each LL Node points to either another Node, or to `NULL`.
- Only one link exists per-Node in the LL implementation.

The "Empty" List

An "Empty" Linked-List is a single Node pointer.

- The LL Head.
- Assigned to `NULL`.

```
Node * m_head = NULL;
```



"First" Node creation

Declares a pointer variable m_head.
Empty LL, so set to NULL pointer.

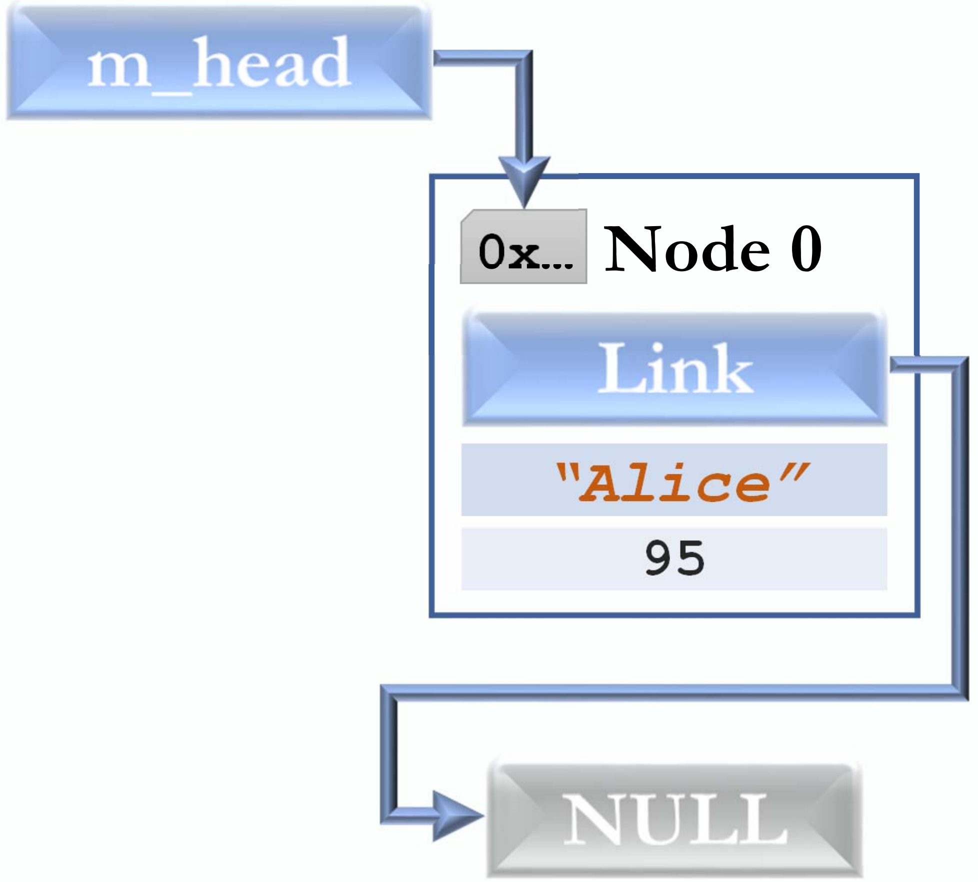
```
Node * m_head = NULL;
```

Dynamically allocate new Node.
The First in the LL, so assigned to head.

```
m_head = new Node;
```

Set head Node data.
Link set to NULL since it's the only node.

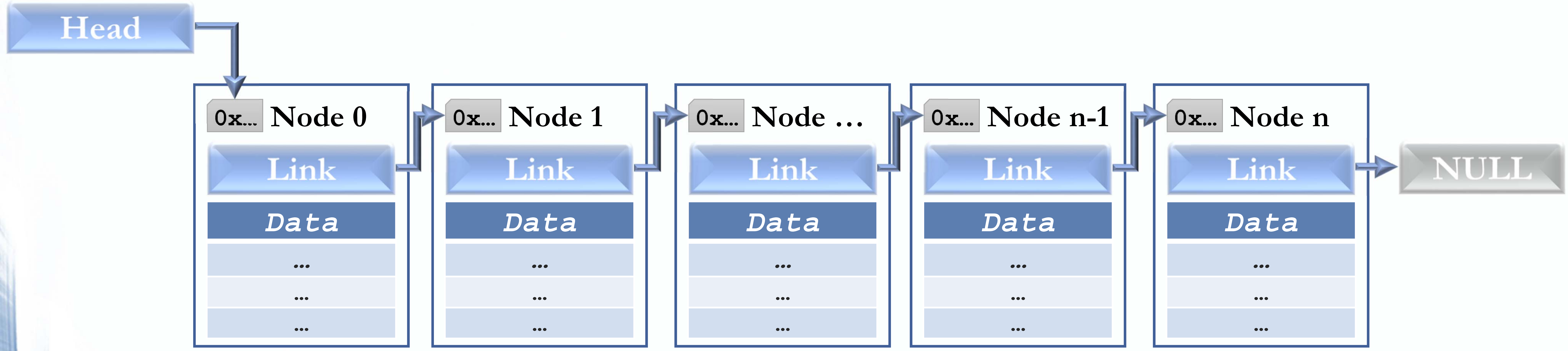
```
m_head->setName("Alice");
m_head->setData(95);
m_head->setLink(NULL);
```



Linked-List(s)

Memory Management and Linked-List(s)

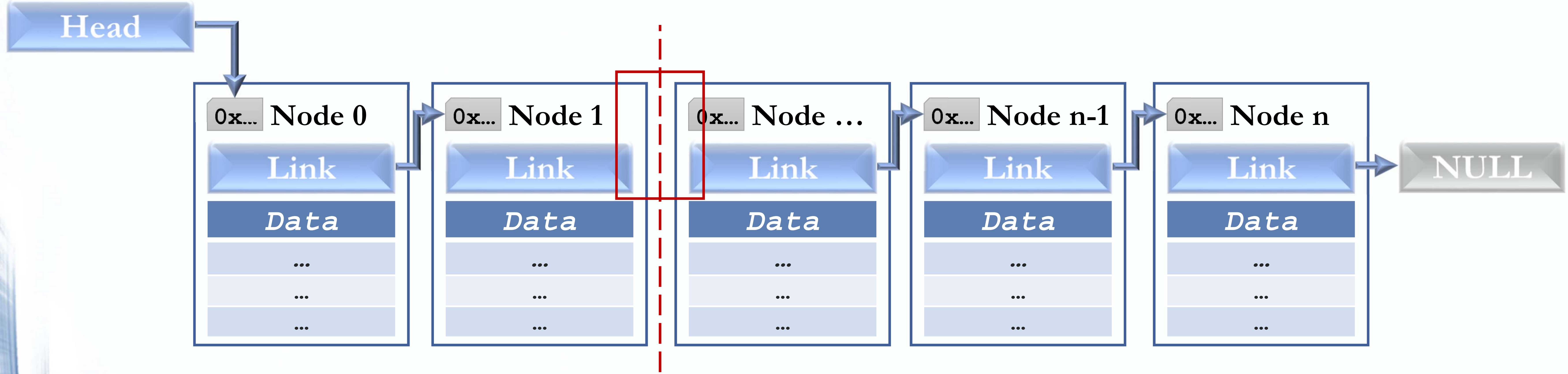
Important to ensure that none of the Nodes will go “missing”.



- Must keep track of where Links point to.
- With careless coding, Nodes can get lost in memory (and there might be no way to recover them).

Memory Management and Linked-List(s)

Important to ensure that none of the Nodes will go "missing".



- Only 1 Link per-Node exists.
- Losing just 1 Link breaks connectivity and isolates a part of the LL.

Linked-List Implementation

Some required functionalities to fully implement a working LL:

- Constructor – `ctor()`
- Destructor – `dtor()`
- `insert()`
- `remove()`
- `empty()`
- `size()`
- `at()`
- `output()`

Linked-List(s)

Linked-List Implementation

Linked-Lists are handled differently under specific circumstances:

- ➢ Linked-List is empty.
- ➢ Linked List has only one element.
- ➢ Linked List has multiple elements.
- ➢ Changing something with the "First" or "Last" Node.

Keep in mind when coding with LLs.

- ➢ Dummy Nodes can alleviate some of these concerns.

Linked-List Traversal

To perform LL Traversal, a control loop is used:

- Initialization

  Set the currently pointed-to Node `curr` to the "First" Node in the LL.

- Termination Condition

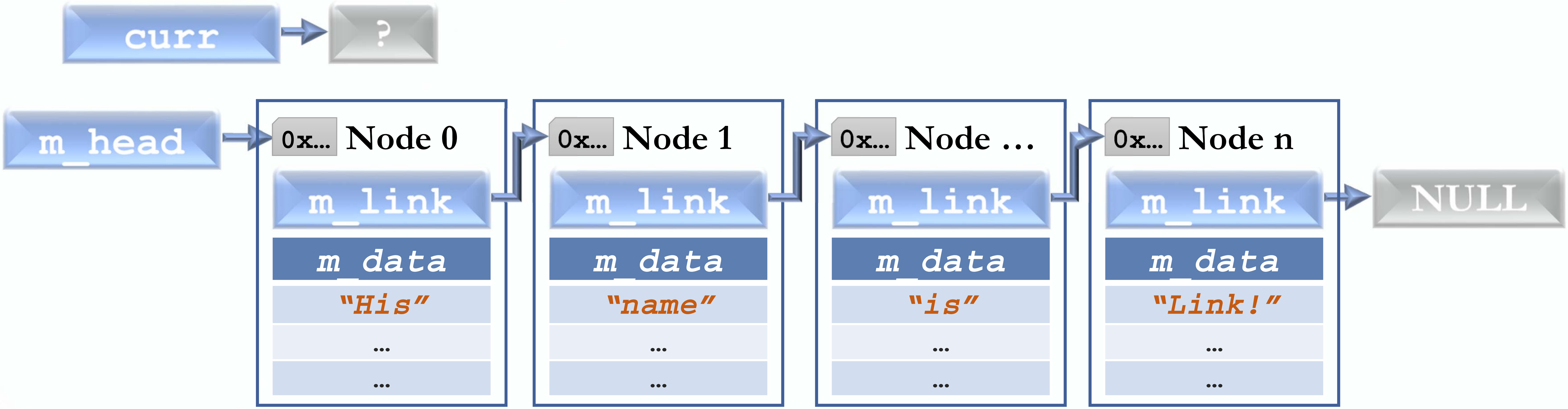
  Continue until we hit the End Marker of the LL (`NULL`).

- Modification

  Move from one Node to another using the "Next" Node Pointer `m_link`.

Linked-List Traversal

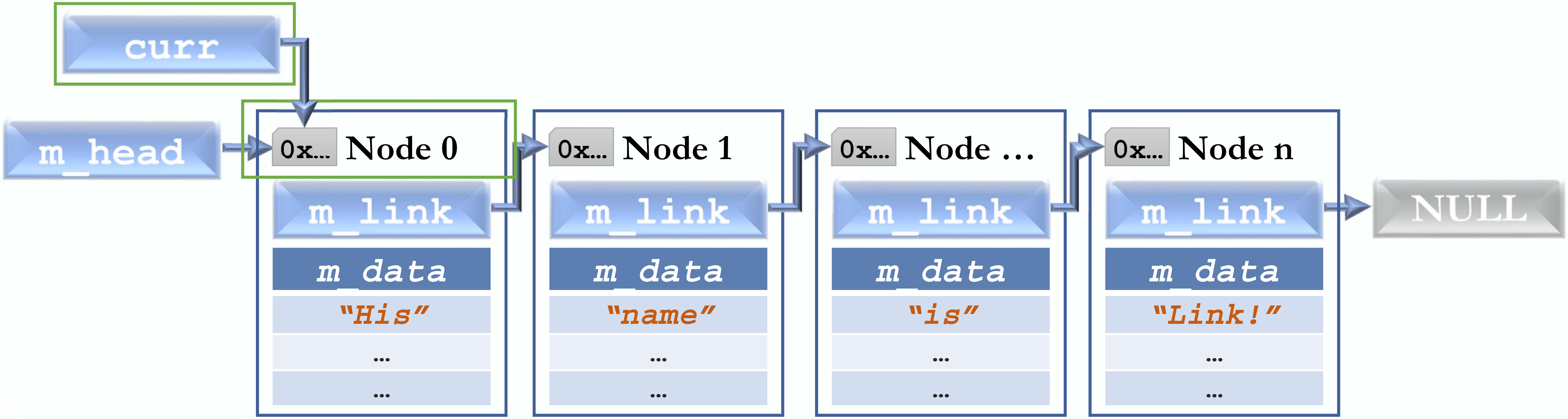
To perform LL Traversal, a control loop is used:




```
for (Node * curr = list.m_head; curr!=NULL; curr = curr->m_link)
```

Linked-List Traversal

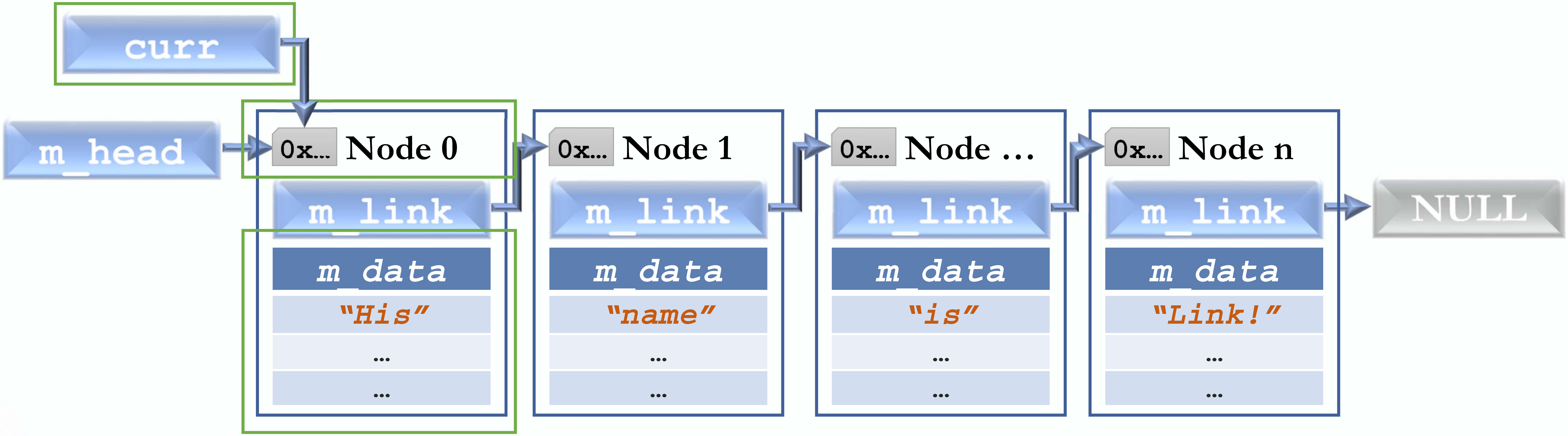
To perform LL Traversal, a control loop is used:




```
for (Node * curr = list.m_head; curr!=NULL; curr = curr->m_link)
```

Linked-List Traversal

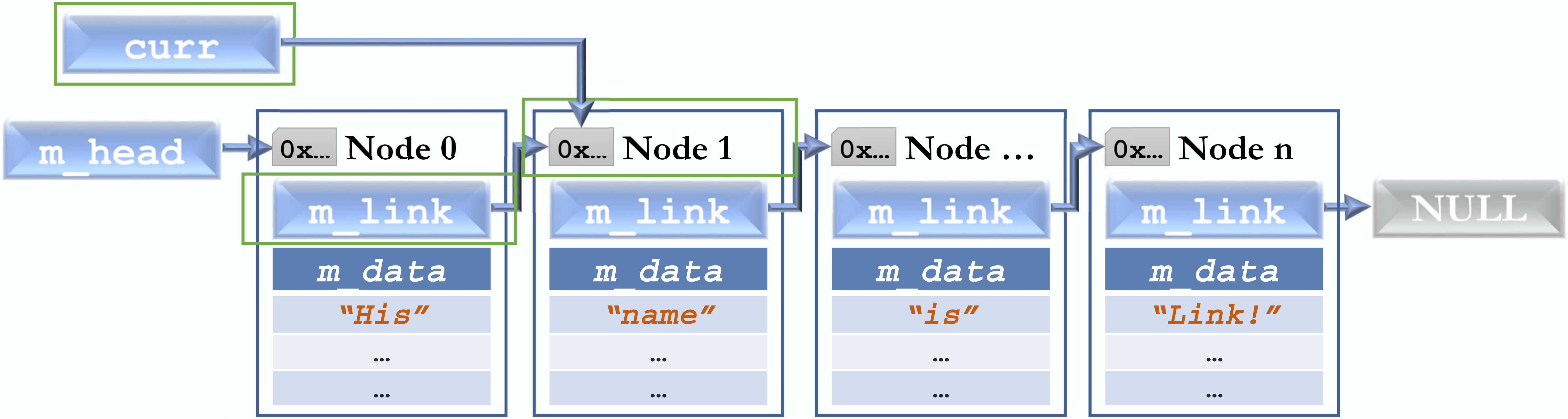
To perform LL Traversal, a control loop is used:




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

Linked-List Traversal

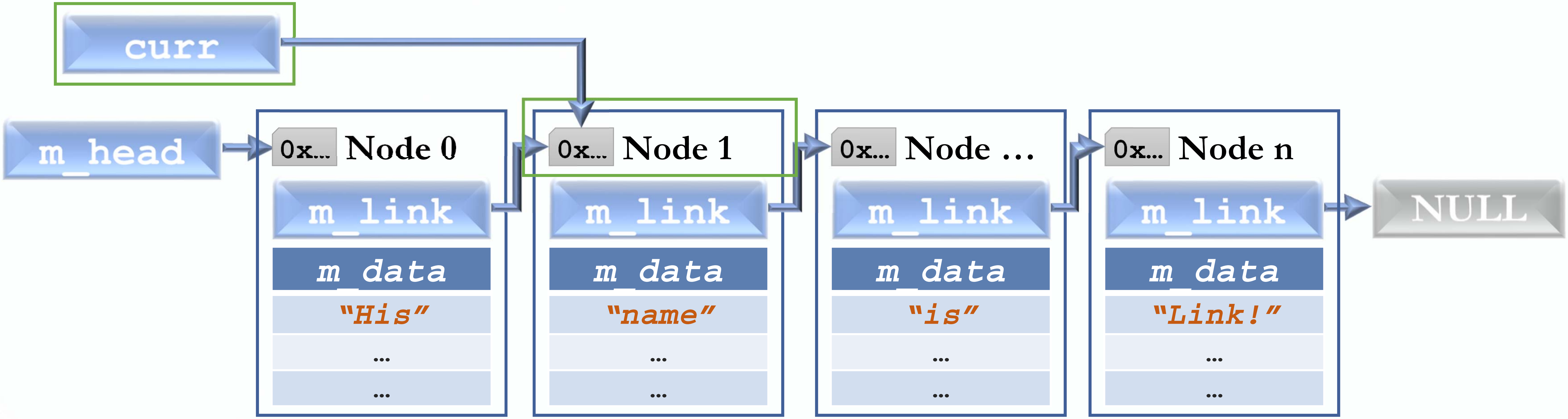
To perform LL Traversal, a control loop is used:




```
for (Node * curr = list.m_head; curr!=NULL; curr = curr->m_link)
```

Linked-List Traversal

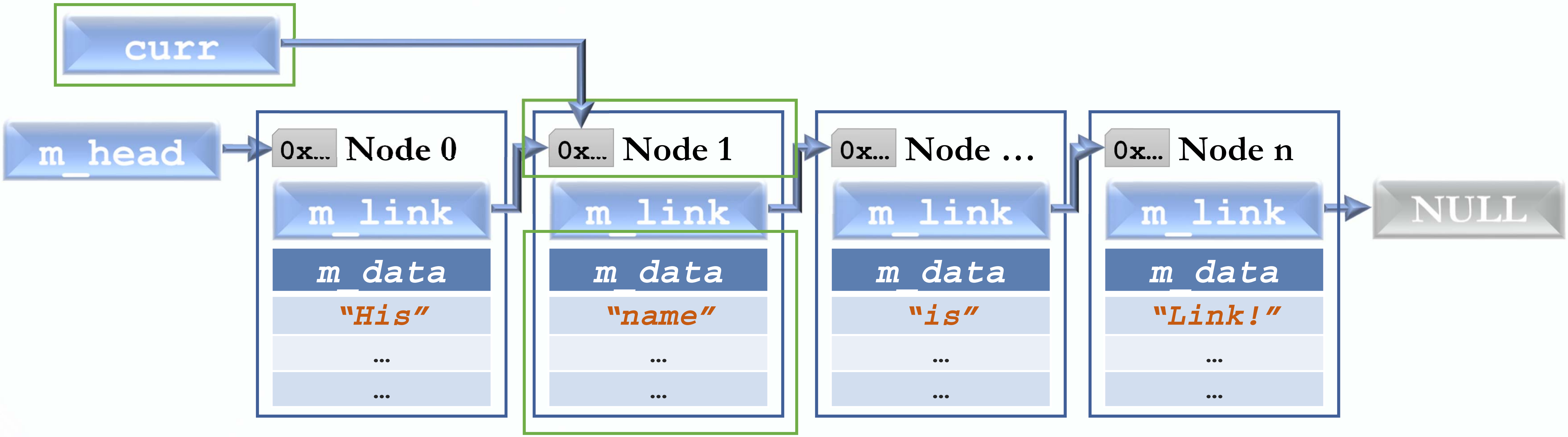
To perform LL Traversal, a control loop is used:




```
for (Node * curr = list.m_head; curr!=NULL; curr = curr->m_link)
```

Linked-List Traversal

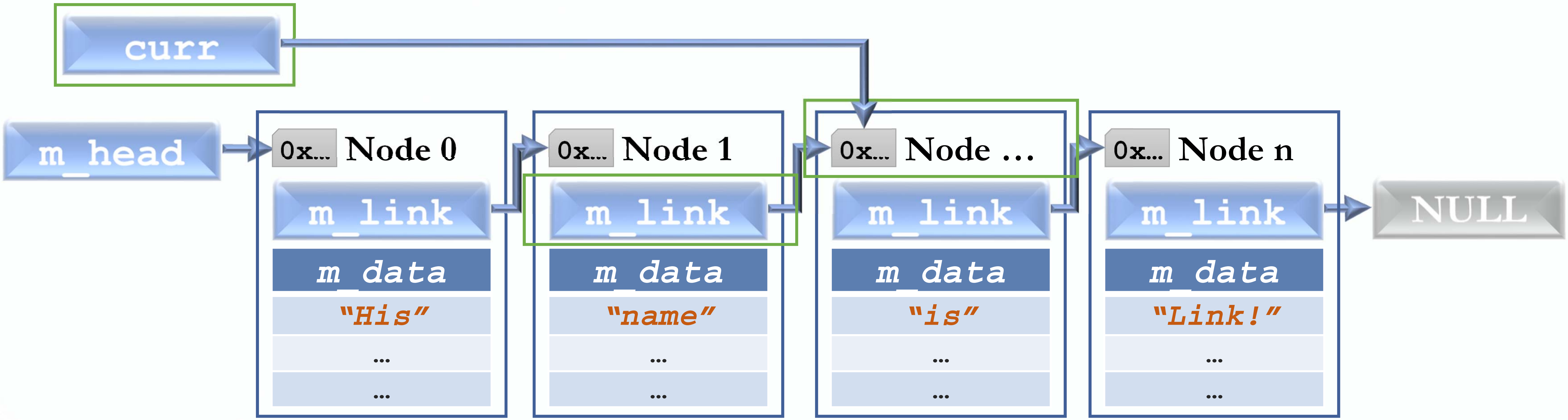
To perform LL Traversal, a control loop is used:




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

Linked-List Traversal

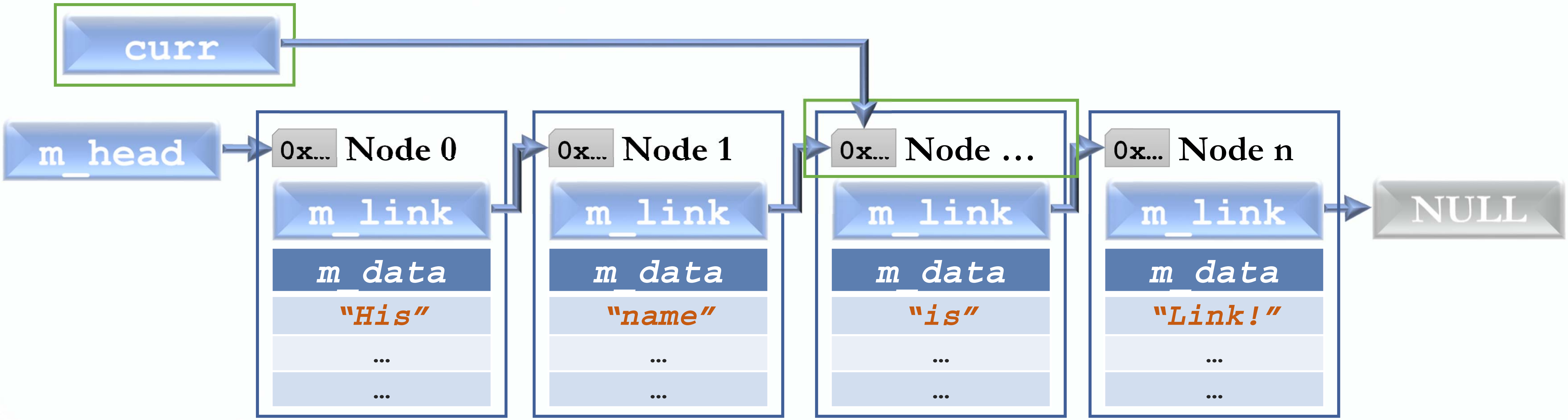
To perform LL Traversal, a control loop is used:




```
for (Node * curr = list.m_head; curr!=NULL; curr = curr->m_link)
```

Linked-List Traversal

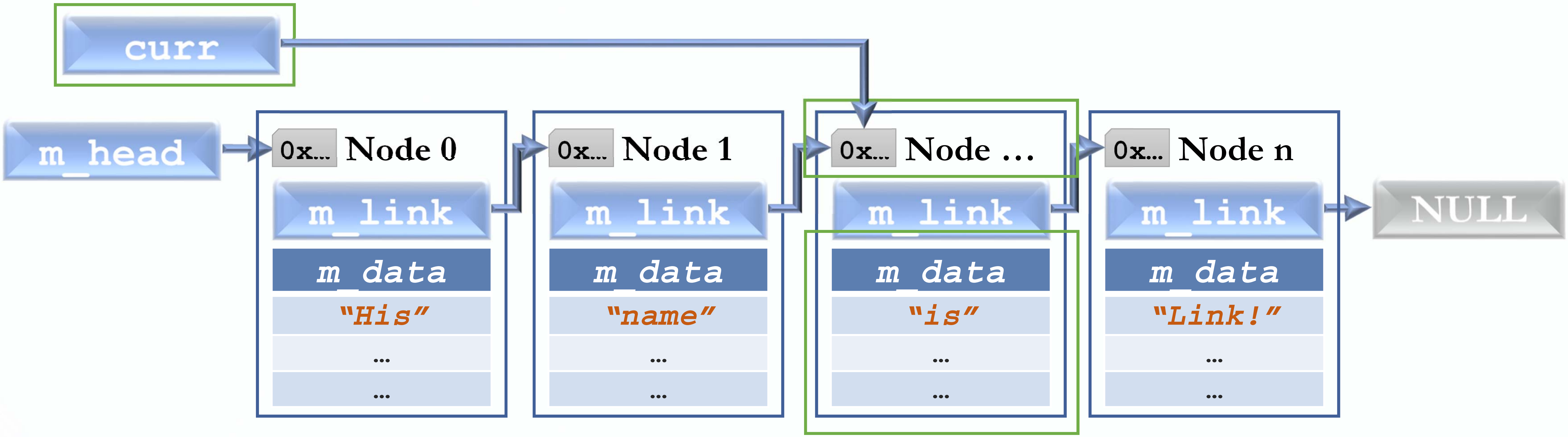
To perform LL Traversal, a control loop is used:




```
for (Node * curr = list.m_head; curr!=NULL; curr = curr->m_link)
```

Linked-List Traversal

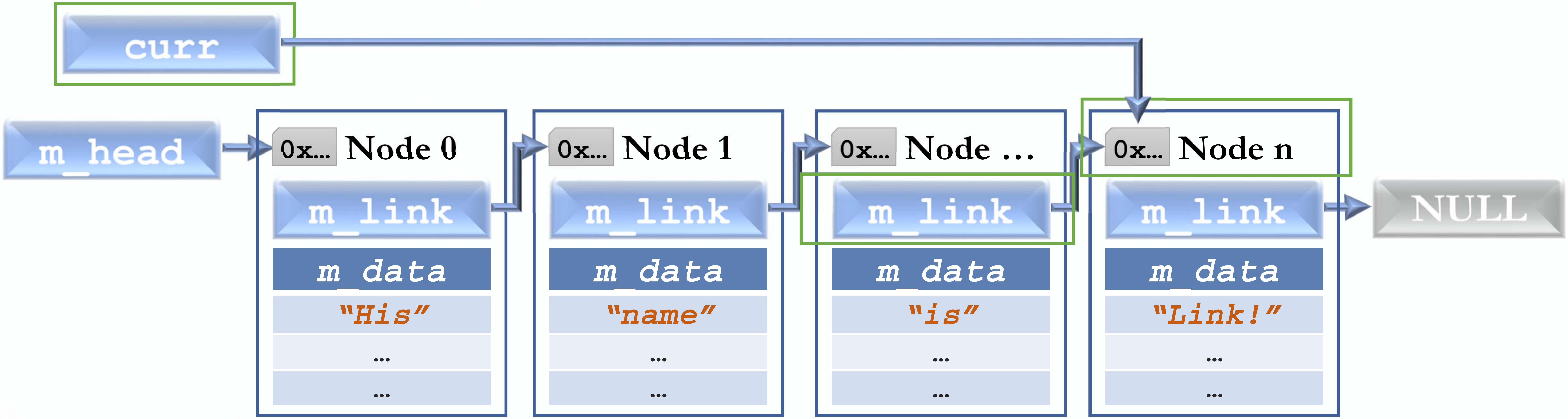
To perform LL Traversal, a control loop is used:




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

Linked-List Traversal

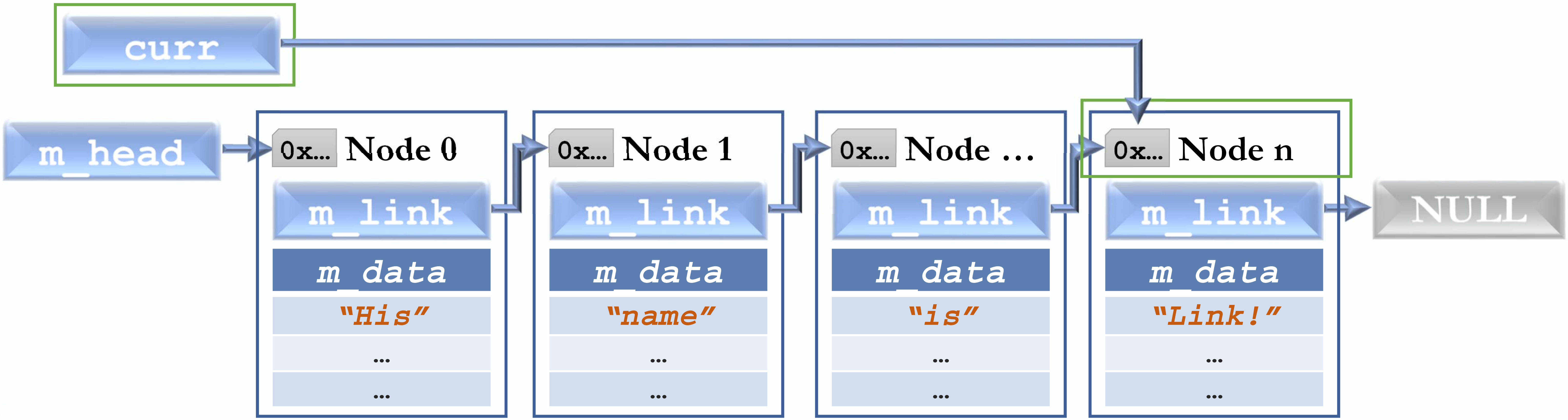
To perform LL Traversal, a control loop is used:




```
for (Node * curr = list.m_head; curr!=NULL; curr = curr->m_link)
```

Linked-List Traversal

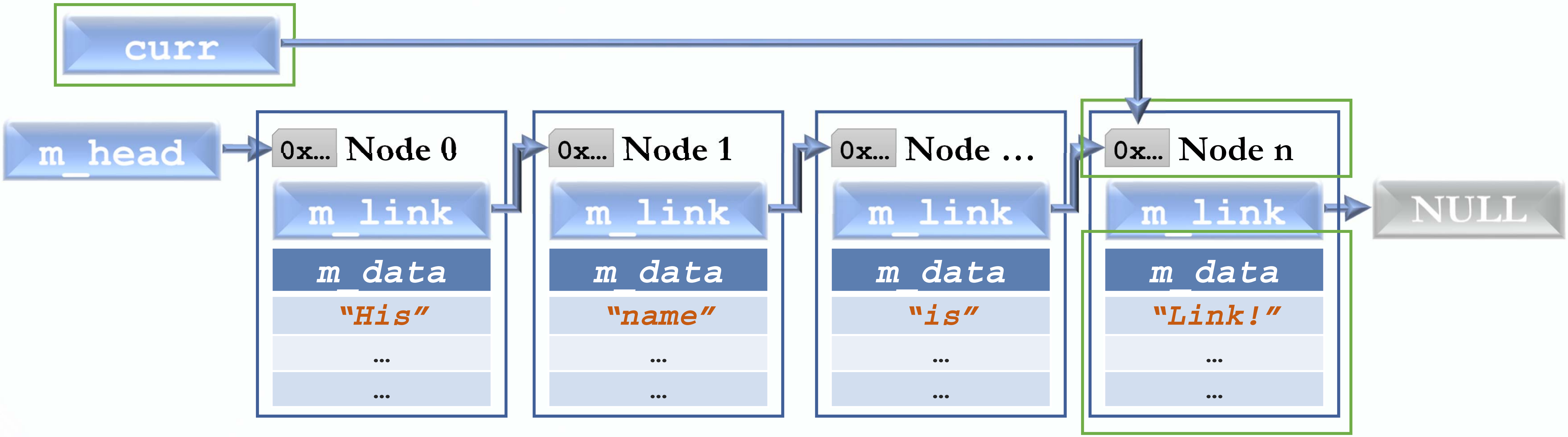
To perform LL Traversal, a control loop is used:




```
for (Node * curr = list.m_head; curr!=NULL; curr = curr->m_link)
```

Linked-List Traversal

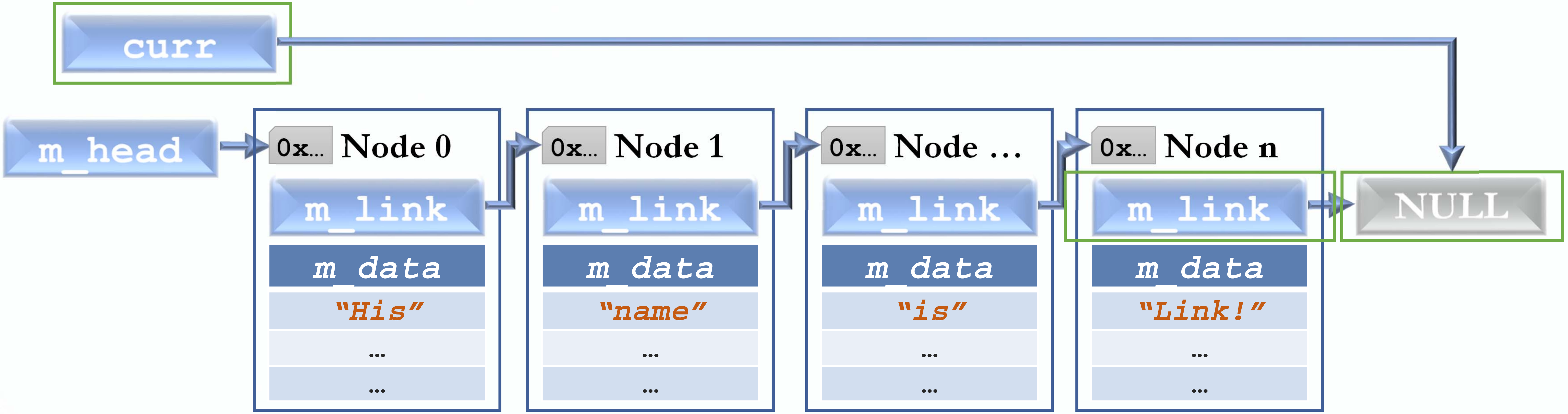
To perform LL Traversal, a control loop is used:




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

Linked-List Traversal

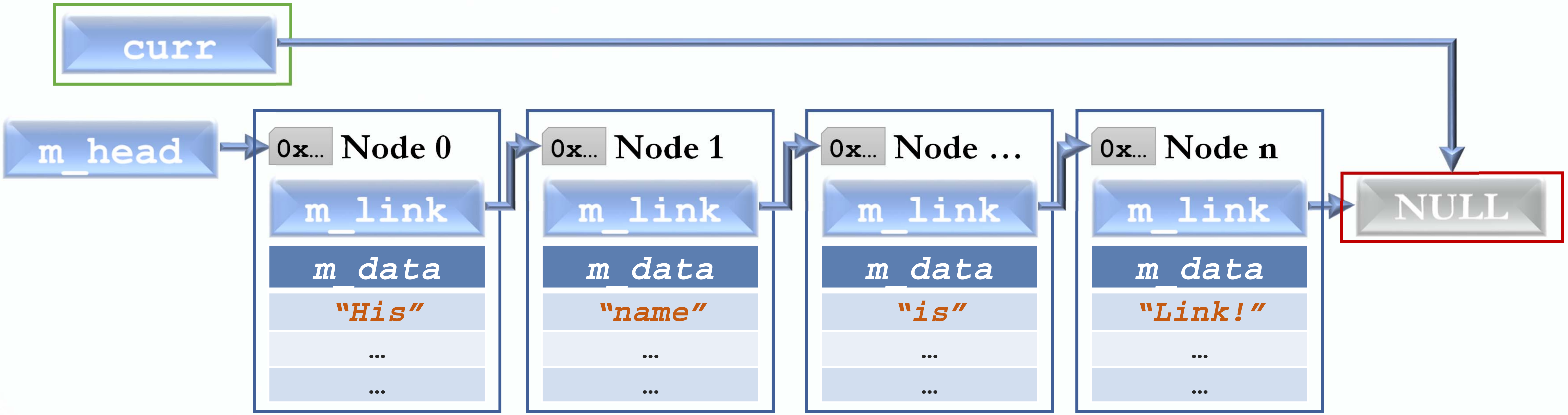
To perform LL Traversal, a control loop is used:




```
for (Node * curr = list.m_head; curr!=NULL; curr = curr->m_link)
```

Linked-List Traversal

To perform LL Traversal, a control loop is used:




```
for (Node * curr = list.m_head; curr!=NULL; curr = curr->m_link)
```

Linked-List Traversal

To perform LL Traversal, a control loop is used:

```
int listCount = 0;
cout << "List says:" << endl;

for (Node * curr = list.m_head; cur != NULL; cur = cur->m_link)
{
  ++listCount;  /* a counter can be used to extract the exact size */
  /* data can be "simple" data (int, double, char*) types
  or struct (direct accessing) or class (accessors/helpers needed) */
  cout << curr->m data;
}
```

Output: `"His name is Link!"`

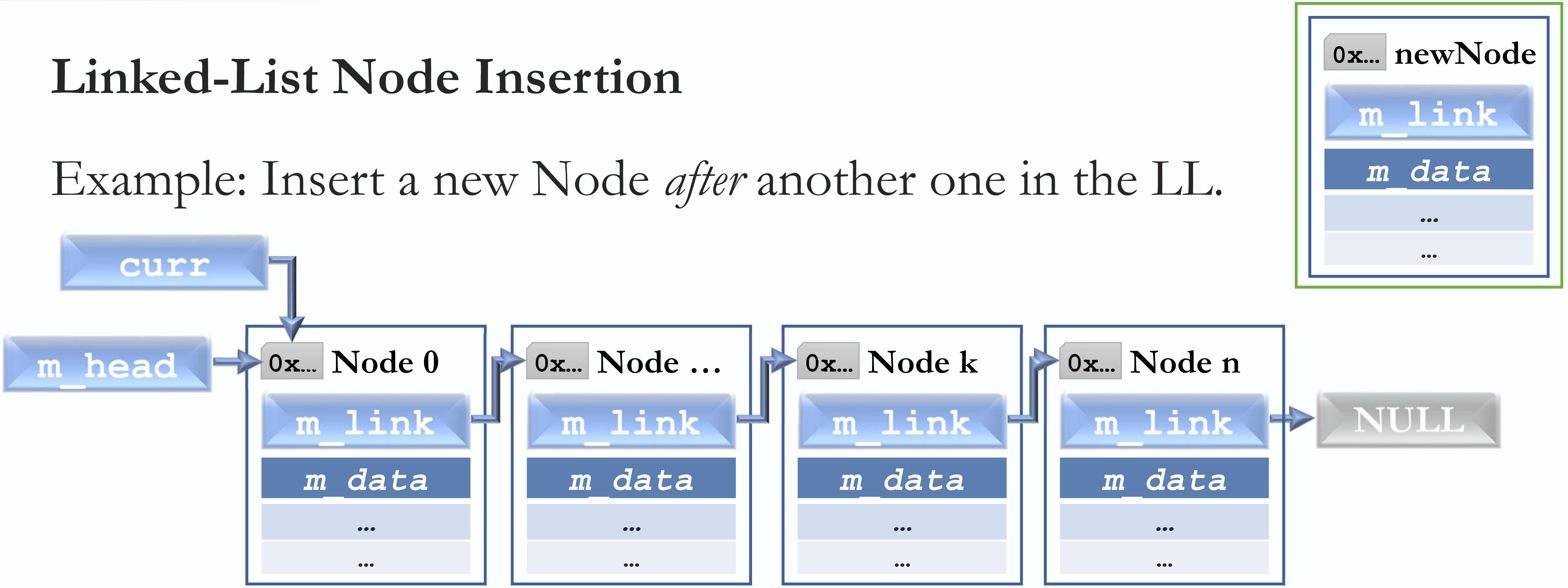
Linked-List Insertion

There are many ways to insert a new Node into a Linked-List:

- As the new "First" element.
- As the new "Last" element.
- Before a given Node (specified by a reference).
- After a given Node.
- Before a given Value.
- After a given Value.

Linked-List Node Insertion

Example: Insert a new Node *after* another one in the LL.

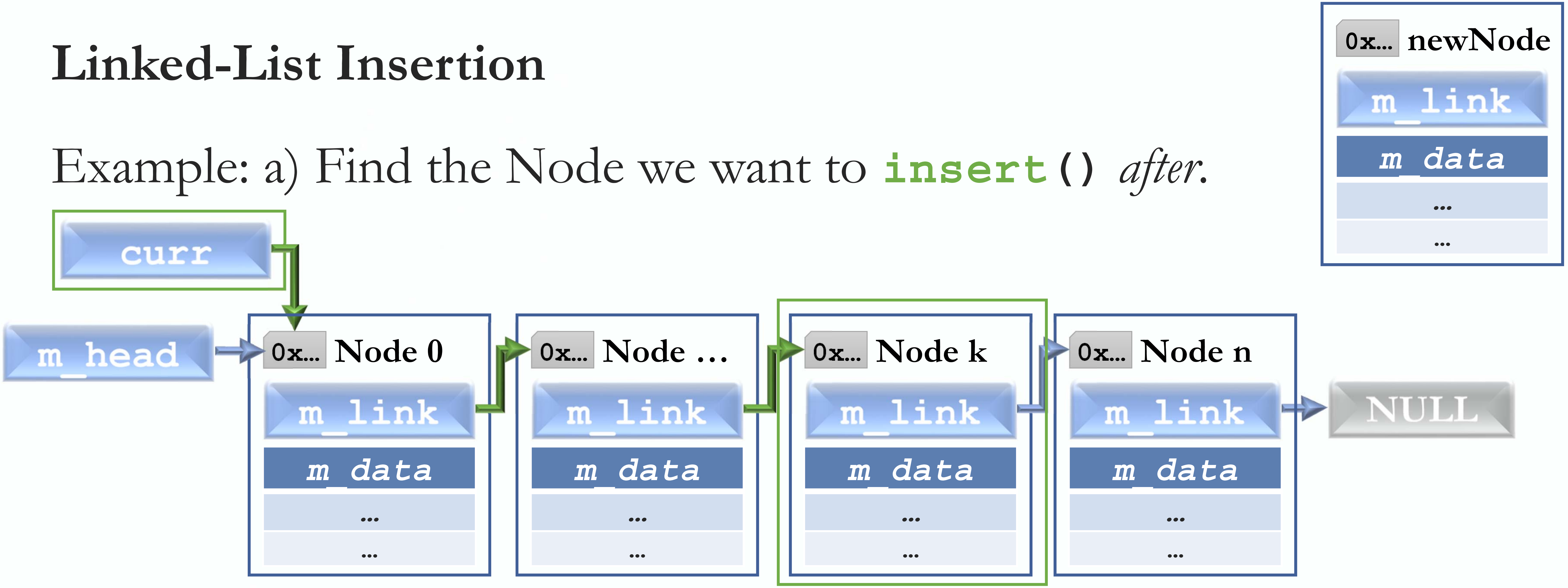


Assume a new'ed Node:

```
Node * newNode_Pt = new Node(…);
```

Linked-List Insertion

Example: a) Find the Node we want to `insert()` *after*.

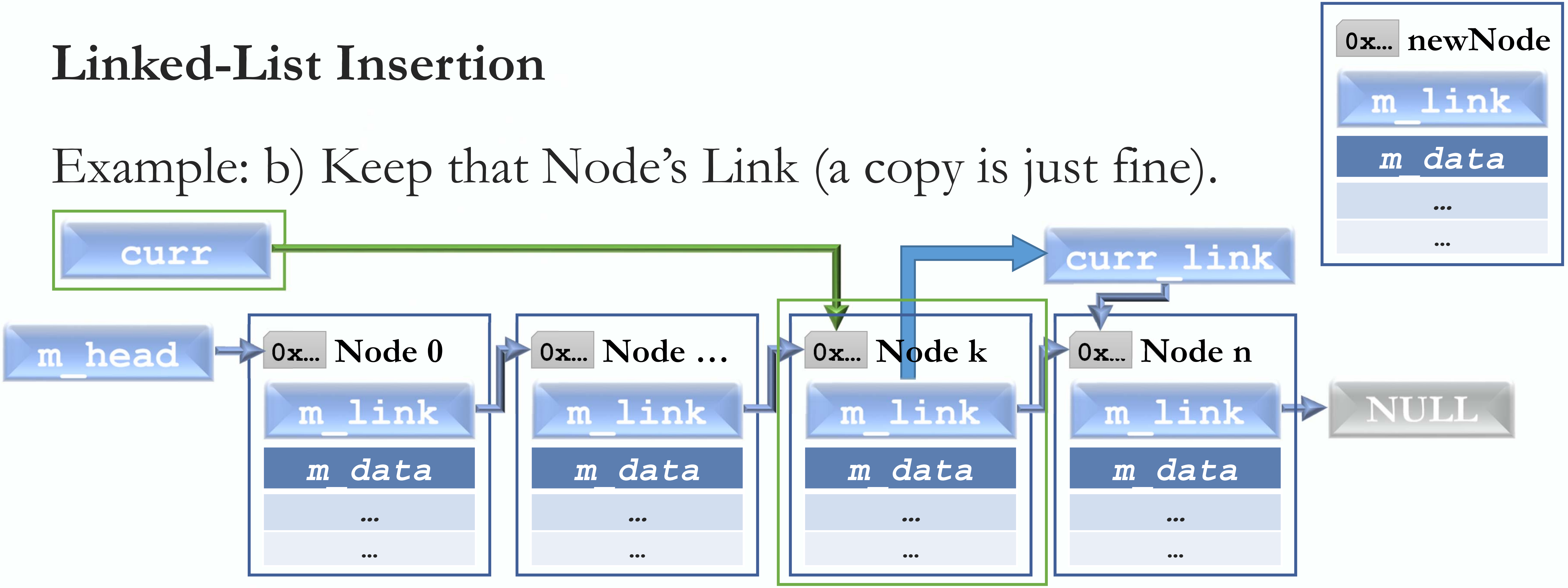



```
for (Node * curr = list.m_head; curr!=NULL; curr = curr->m_link)
  if ( curr->m_data == … ){ //check to find requested node
```

Linked-List(s)

Linked-List Insertion

Example: b) Keep that Node's Link (a copy is just fine).

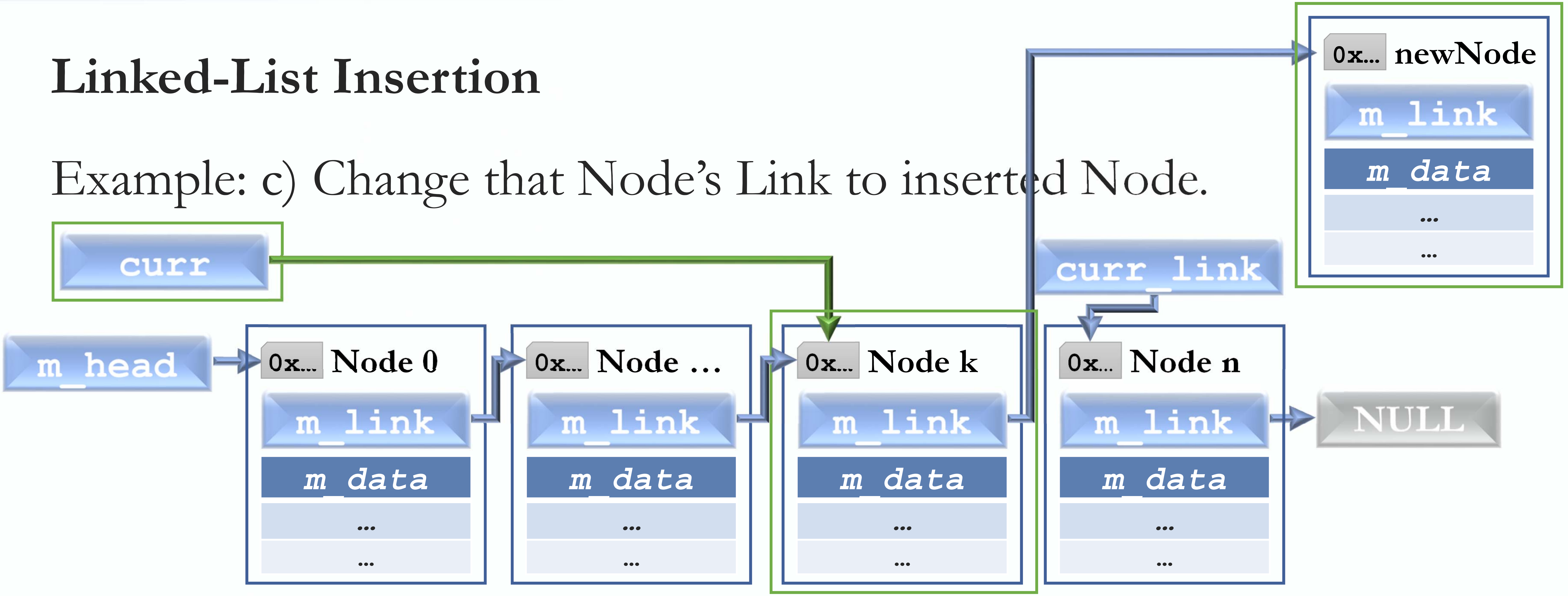



```
for (Node * curr = list.m_head; curr!=NULL; curr = curr->m_link)
  if ( curr->m_data == … ){
    Node * curr_link = curr->m_link; //keep copy of original link
    …
  }
```

Linked-List(s)

Linked-List Insertion

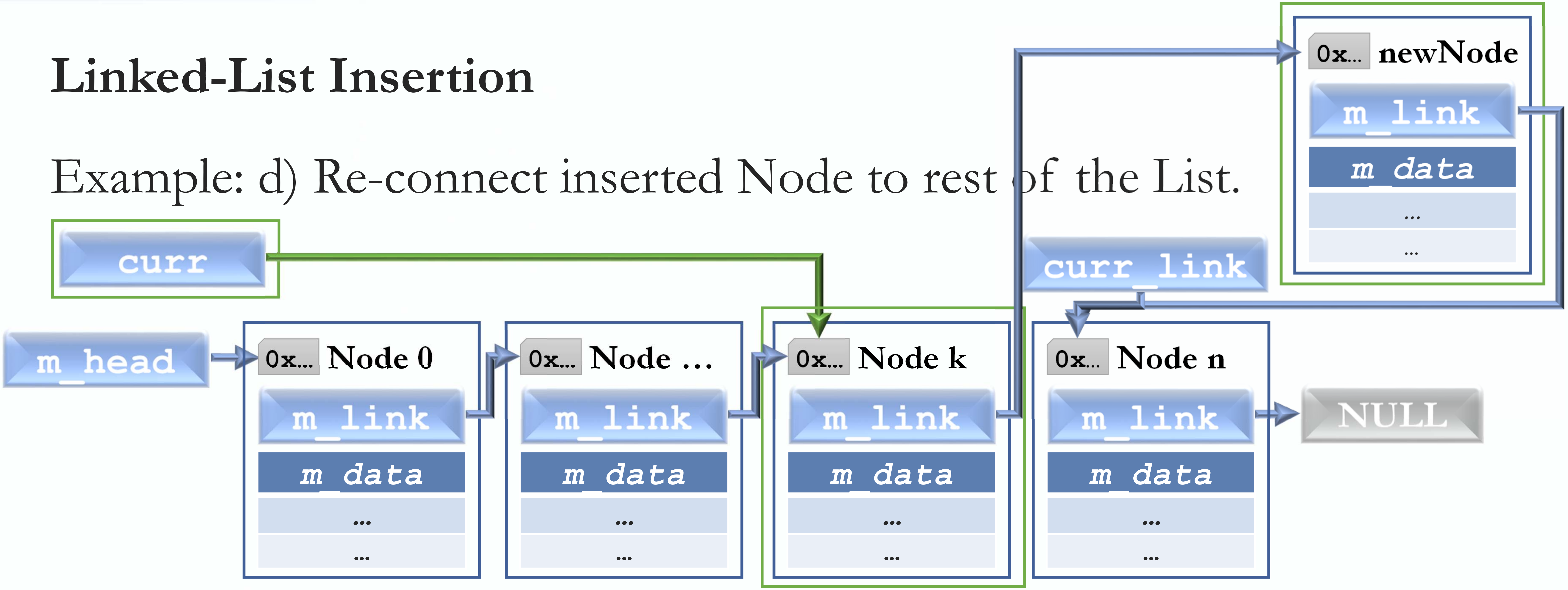
Example: c) Change that Node's Link to inserted Node.




```
for (Node * curr = list.m_head; curr!=NULL; curr = curr->m_link)
  if ( curr->m_data == … ){
    Node * curr_link = curr->m_link;
    curr->m_link = newNode_Pt; //point link to newly inserted node
    …
  }
```

Linked-List Insertion

Example: d) Re-connect inserted Node to rest of the List.



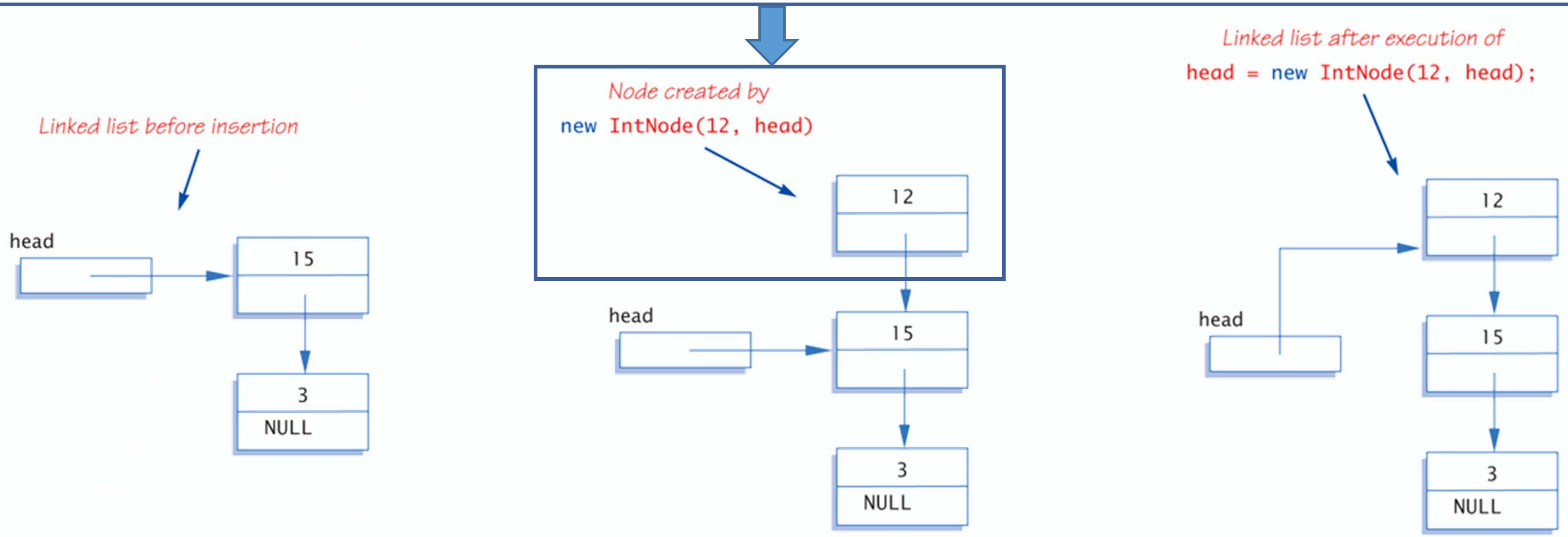

```
for (Node * curr = list.m_head; curr!=NULL; curr = curr->m_link)
  if ( curr->m_data == … ){
    Node * curr_link = curr->m_link;
    curr->m_link = newNode_Pt;
    newNode_Pt->m_link = curr_link; //re-connect list
  }
```

Linked-List Insertion

Further Examples: Insert Node to the Head of the LL.

A Node parametrized `ctor` is assumed that initializes members m_data and m_link:

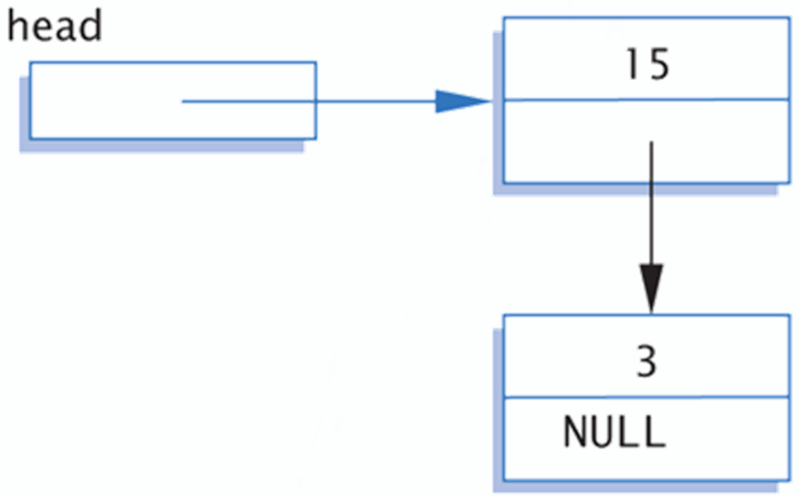
`new IntNode(12, list.m_head);` Creates new node that links to original LL Head.



Linked-List Insertion

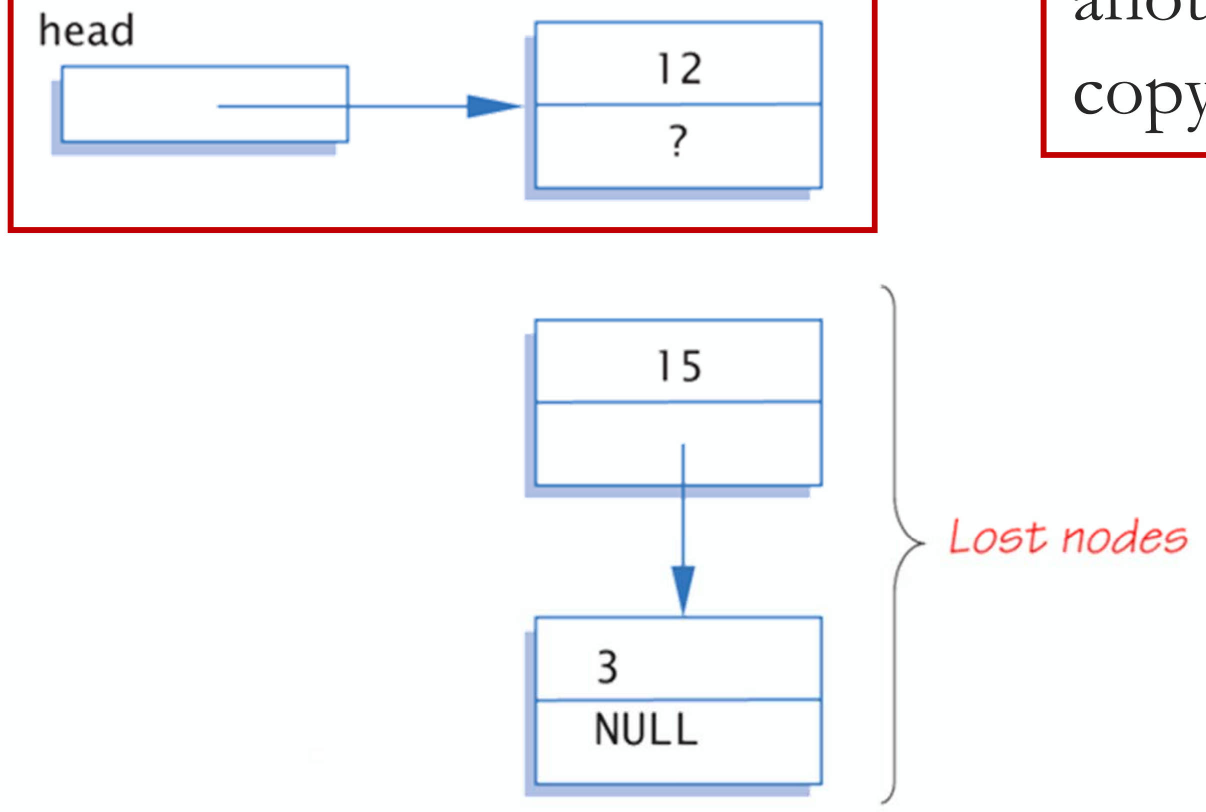
Further Examples: Lost Nodes.

Linked list before insertion



Situation after executing

```
head = new IntNode;
head->setData(theData);
```

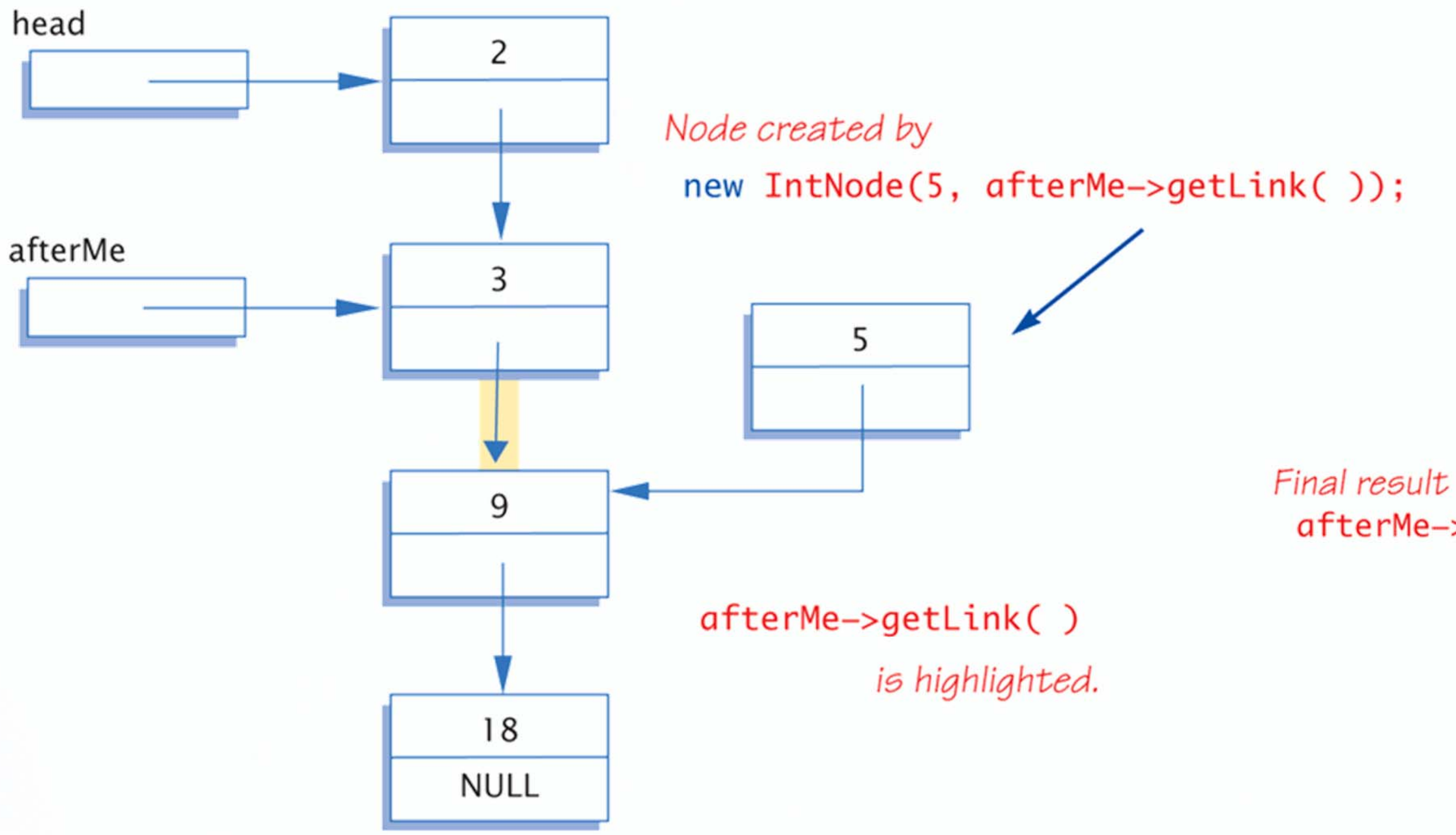


Wrong: Directly works on LL Head, effectively repointing it to another Node without keeping a copy of connectivity.

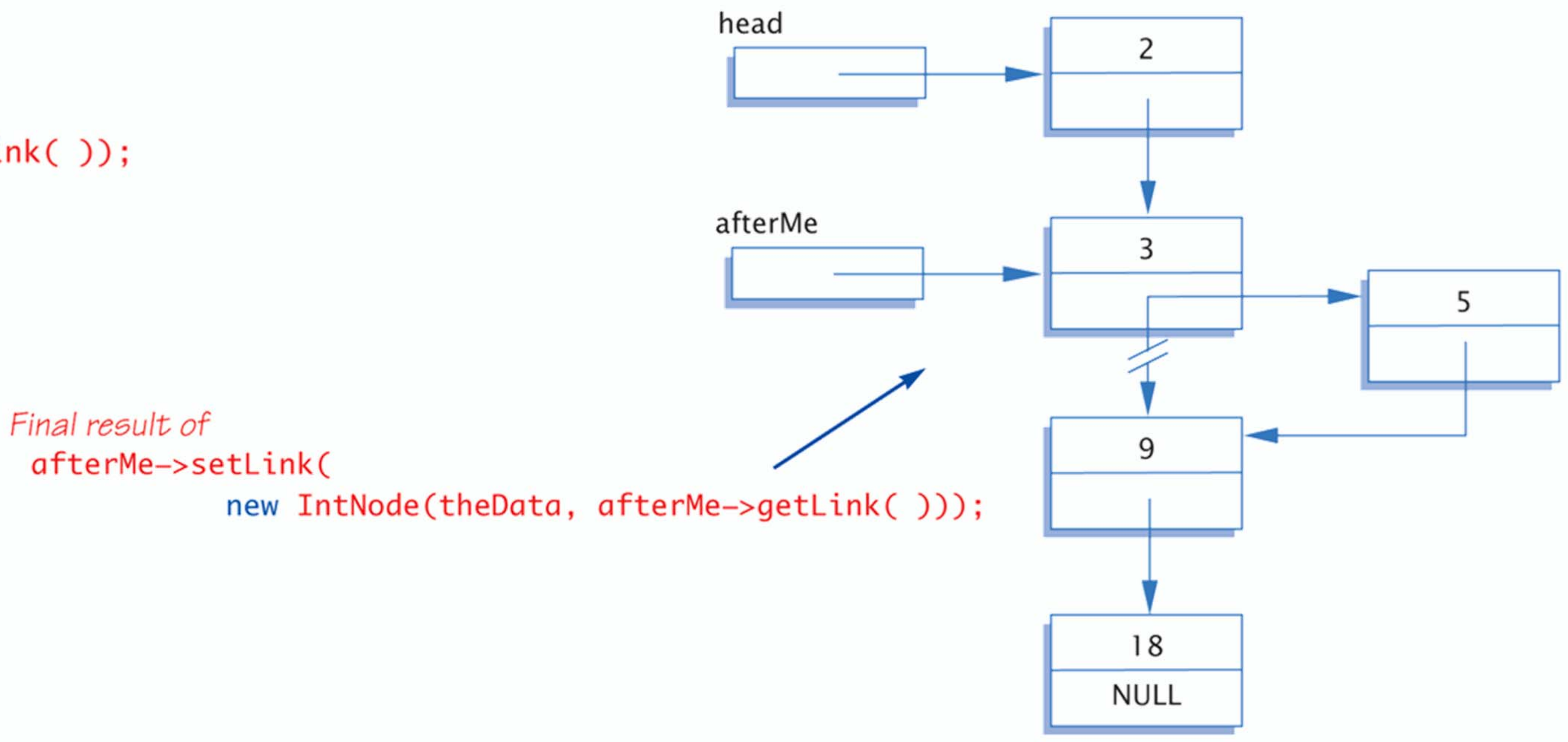
Linked-List Insertion

Further Examples: Insertion in middle of the LL.

State a) :



State b) :



Linked-List Insertion

To perform LL insertion, control switches are used:

```
Data newData(…);
Node * newNode_Pt = new Node(newData, NULL);
…
Node * curr=list.getHead();
if (curr == NULL){  //check if empty list
 list.setHead(m_head) = newNode_Pt;  //set as first element
 newNode_Pt->setLink(NULL);     //(redundant)
}
else{  //traverse list
 for (Node* curr=list.getHead(); curr!=NULL; curr=curr->getLink()){
   if ( curr->getData() … ){
     newNode_Pt->setLink( curr->getLink() );  //connect to original
     curr->setLink( newNode_Pt->getLink() );  //connect to new
   }
 }
}
```

Not really necessary.
Why?

```
class Node {
 public:
 Node();
 Node(const Data & d,
      Node * l=NULL);
 ~Node();
 private:
 Data m_data;
 Node * m_link;
};
```

Linked-List Node Deletion

In order to delete a Node from a Linked-List, the Link of its *predecessor* has to change.

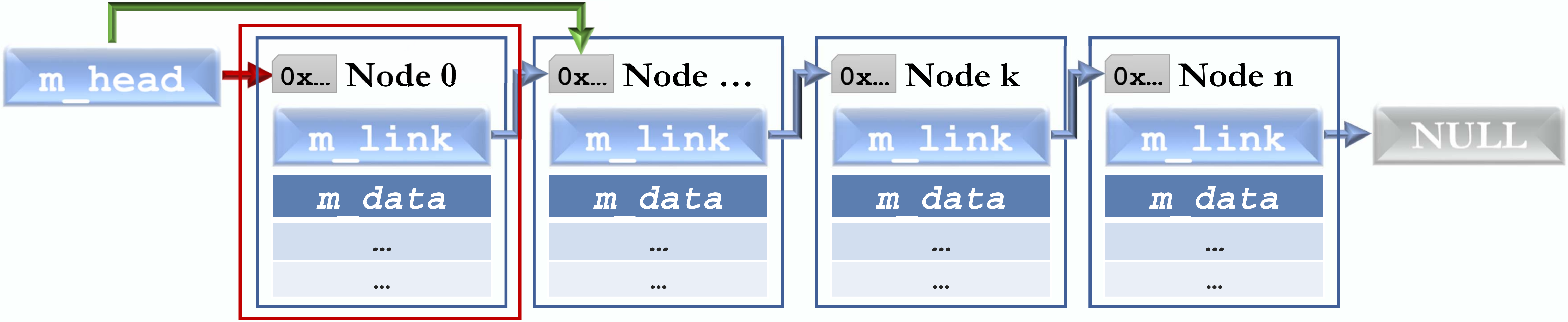
Slightly tricky:

- Link(s) are Pointers that only point forward (to "Next" Node in the Linked-List), so *predecessor* Node cannot be directly resolved.
- Deleting the "First" Node in a Linked-List is also a special case, as the Node's *predecessor* is the LL Header.

Linked-List Node Deletion

To remove the "First" element, change the Head of the LL.

➢ But also be careful to actually **delete** the original element.

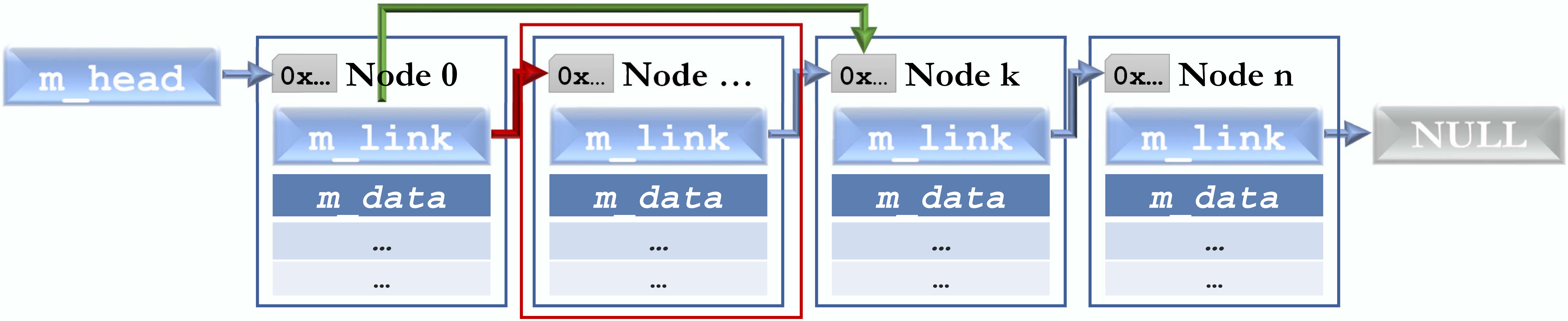



```
Node * head_link = list.m_head;   //make copy of original head
list.m_head = list.m_head->m_link;   //point head to next of original head
delete head_link;   //delete original head
```

Linked-List Node Deletion

To remove some other element, change the Link of its *predecessor*.

- But also be careful to actually `delete` the original element.

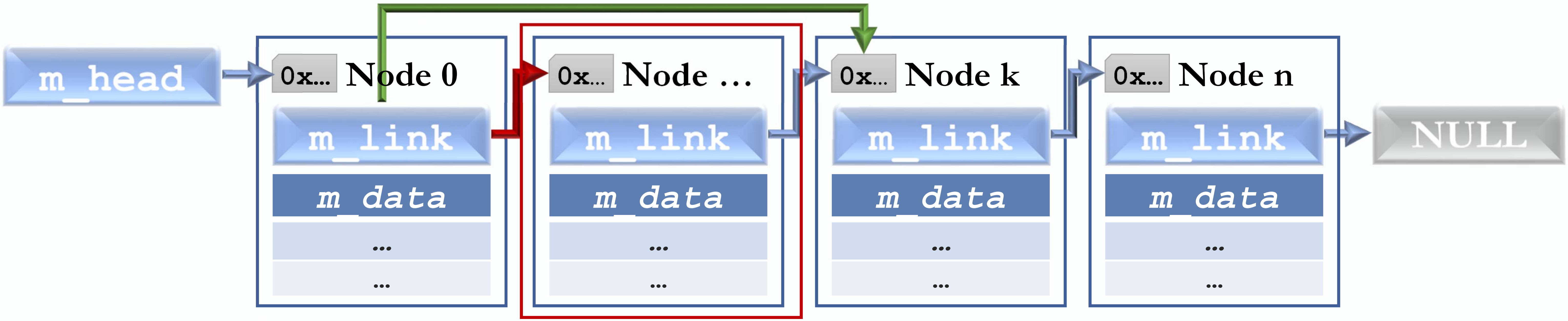



```
for (Node * curr = list.m_head; curr!=NULL; curr = curr->m_link)
  /* search node to be removed on a predecessor basis:
     curr node will be the predecessor node of the one to be deleted */
  if ( curr->m_link && curr->m_link->m_data == … )
   // first check that next node is not NULL and then check its data
```

Linked-List Node Deletion

To remove some other element, change the Link of its *predecessor*.

- ➢ But also be careful to actually `delete` the original element.

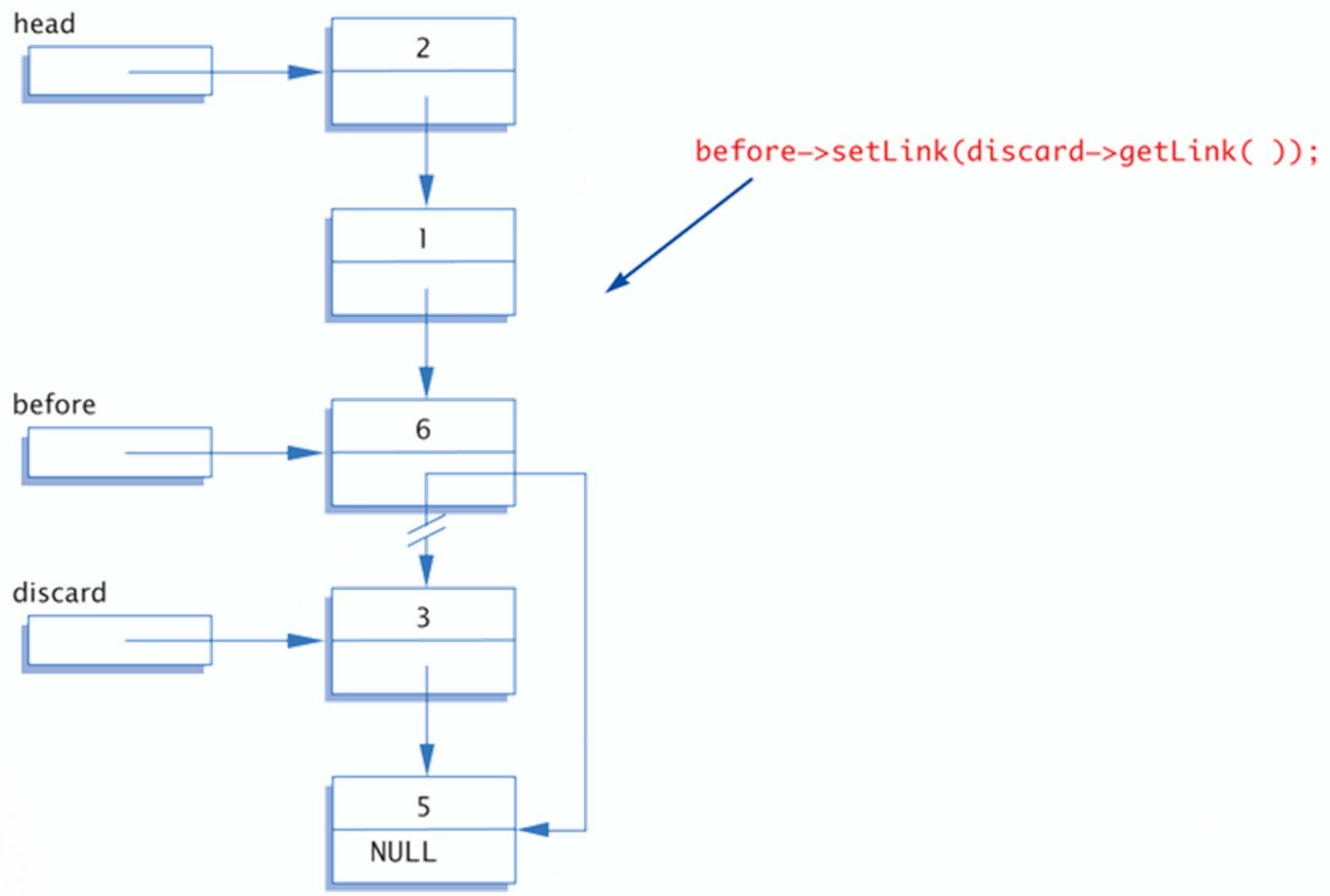



```
for (Node * curr = list.m_head; curr!=NULL; curr = curr->m_link)
  if ( curr->m_link && curr->m_link->m_data == … ){
    Node* curr_link = curr->m_link; //make copy of predecessor's next
    curr->m_link = curr->m_link->m_link; //point predecessor two nodes down
    delete curr_link; //delete predecessor's next
  }
```

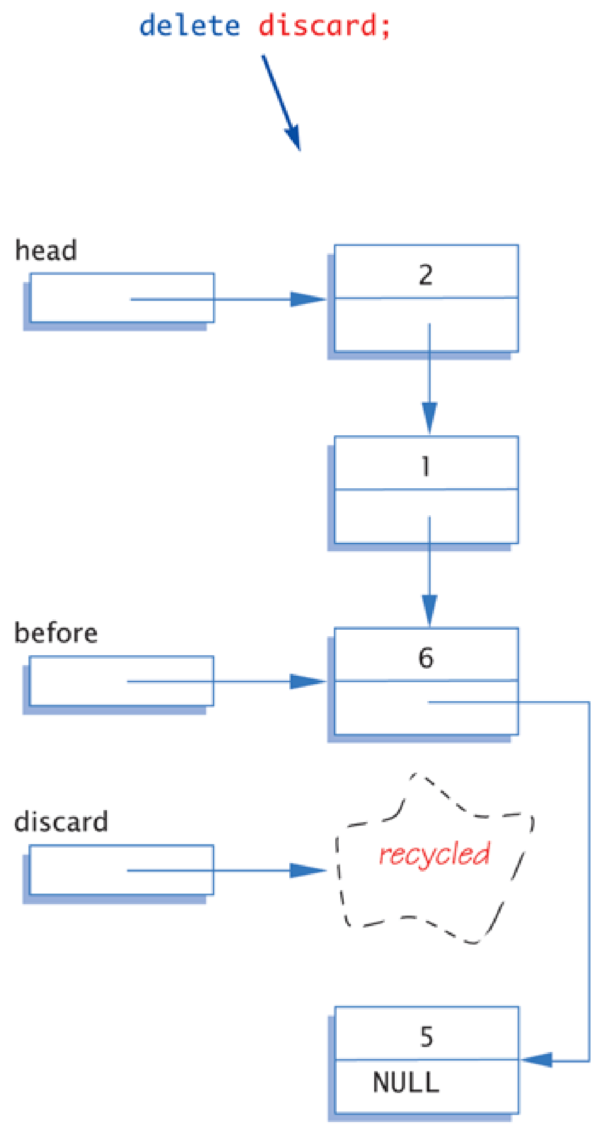
Linked-List Deletion

Further Examples:

State a) :



State b) :



Linked-List Node Search

Implementation involves a function with two arguments:

```
Node * search(Node * head, const Data & target);
```

Precondition:

- Pointer **head** points to Head of LL. Pointer in last node of LL is set to **NULL**. If list is empty, head is NULL

Returns:

- On success, a pointer to the 1st Node found to contain the target (equality checking determined by the specifications of the ***Data*** data type.
- On failure (not found), a **NULL** pointer.

(Simple traversal of the LL is performed, similarly to array traversal)

Linked-List Node Search

Implementation involves a function with two arguments:

```
Node * search(Node * head, const Data & target){
    if (!head){  //empty list
        return NULL; //not found
    }
    Node * curr = head;  //initialize list traversal
    while (curr->getLink()){  //check for last element
        if (curr->getData() == target){  //found
            return curr;
        }
        curr = curr->getLink();  //traverse list
    }
    return NULL;  //not found
}
```

Linked-List Overall

To perform LL insertion, control switches are used:

```
class LinkedList {
  public:
    LinkedList();
    LinkedList(size_t count, const Data & value);
    LinkedList(const LinkedList & other);
   ~LinkedList();
    Data * find(const Data & searchNode);
    void insert(const Data & insertNode);
    void erase(const Data & removeNode);
  private:
    Data * m_head;
    size_t m_size;  //tentative
};
```

How should this be implemented ?

```
class Node {
 public:
 Node();
 Node(const Data & d,
      Node * l=NULL);
 ~Node();
 private:
 Data m_data;
 Node * m_link;
};
```

CS-202

Time for Questions !