CPE 212 - Fundamentals of Software Engineering

Linked Lists

Outline

- Defining Linked Lists
- Linked Lists Allocation
- Singly-Linked List
- Doubly-Linked List
- Destructor

Reminder:

Project 02 due this Friday by 11:59pm

Linked List

Same basic operations as other lists

- Create
- Insert
- Delete
- IsFull
- IsEmpty
- Implementation of these methods is different

Array-based implementation of Linked Lists

- Each element of the array is a node
- Each node contains a data item and a link to the next item in the list
- Adjacent items in the list may not be adjacent within the array
- Need to know where in the list begins within the array

Array-Based Linked List Example

Head = 2

```
struct NodeType
{
  ItemType component;
  int nextItem;
};
```

	component	nextItem
node[0]	58	-1
node[1]		
node[2]	4	5
node[3]		
node[4]	46	0
node[5]	16	7
node[6]		
node[7]	39	4

Link List Example

```
//*******  LList.h Standard Header Information Here **********

#ifndef LLIST_CLASS_H_
 #define LLIST_CLASS_H_

 const int MAX_LENGTH = 100;       // Maximum number of list items
 typedef int ItemType;     // Data type of each item in list
  struct NodeType           // Node record with data field and next item field
 {
   ItemType value;
   int nextItem;
 };
 class LList           // Unordered Linked List of items
 {
   private:
      int  length;         // Actual number of items in list <<==== ADDED
      int  head;           // Index of first node in list <<==== ADDED
      NodeType  node[MAX_LENGTH];  // List of unsorted data items

   public:
      LList();         // Default constructor creates empty list
      bool IsEmpty() const;        // Returns TRUE if empty, FALSE otherwise
      bool IsFull() const;     // Returns TRUE if full, FALSE otherwise
      int  Length() const;     // Returns length of list
      void Insert(ItemType item);  // Adds item to end of list assuming list is not full
      void Delete(ItemType item);  // Removes first item occurrence from list if not empty
      bool IsPresent(ItemType item);   // Returns TRUE if item in list, otherwise FALSE
 };

#endif  // End of LLIST_CLASS_H_
```

Link List Example

```
//*******  LList.cpp Standard Header Information Here **********

#include “LList.h”

LList::LList()          // Default constructor creates empty list
{
 length = 0;
} // End LList::List()



bool LList::IsEmpty() const     // Returns TRUE if empty, FALSE otherwise
{
 return (length == 0);
} // End LList::IsEmpty()



bool LList::IsFull() const      // Returns TRUE if full, FALSE otherwise
{
 return (length == MAX_LENGTH);
} // End LList::IsFull()



int LList::Length() const       // Returns length of list
{
 return length;
} // End LList::Length()
```

Link List Example

```
//*******  LList.cpp continued from previous slide **********


bool LList::IsPresent(ItemType item)  // Returns TRUE if item in list,
otherwise FALSE
{
 int index = head;    // Start looking at the beginning of the linked list

 while ((index != -1) && (item != data[index])) // Locate item index first
 {
   index = data[index].nextItem; // Follow the link to the next item in list
 }

 return (index != -1);  // index == length => item not found.  Index == -1 =>
item found

} // End LList::IsPresent(…)
```

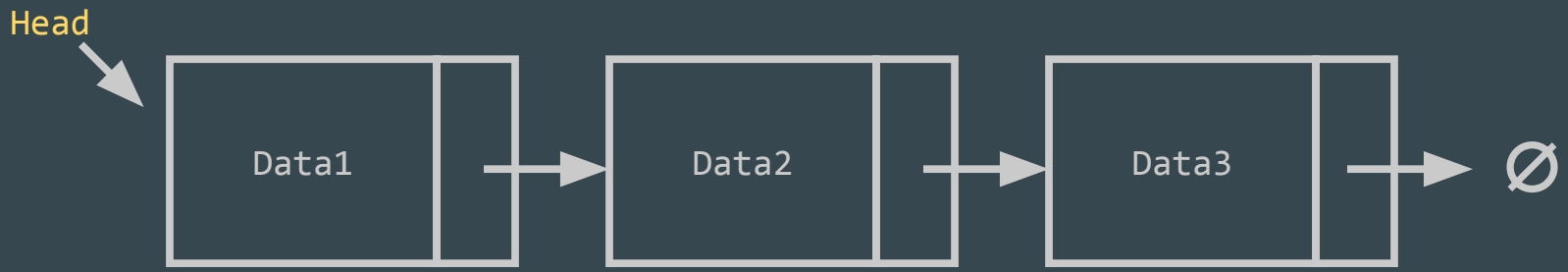
Link List Example

```
//*******  LList.cpp continued from previous slide **********

void LList::Insert(ItemType item)   // Adds item to end of list assuming list is not full
{

// ???
// Must now worry about memory management since items are no longer stored in consecutive
// memory cells
//
// If list is empty, must also set nextItem link of new item to -1 to terminate the list
// properly.
 } // End LList::Insert(…)


void List::Delete(ItemType item)    // Removes first occurrence from list if not empty
{

// ???
// Must now worry about memory management since items are no longer stored in consecutive
// memory cells.

} // End LList::Delete(…)
```

Linked List Using Dynamic Allocation

Concept

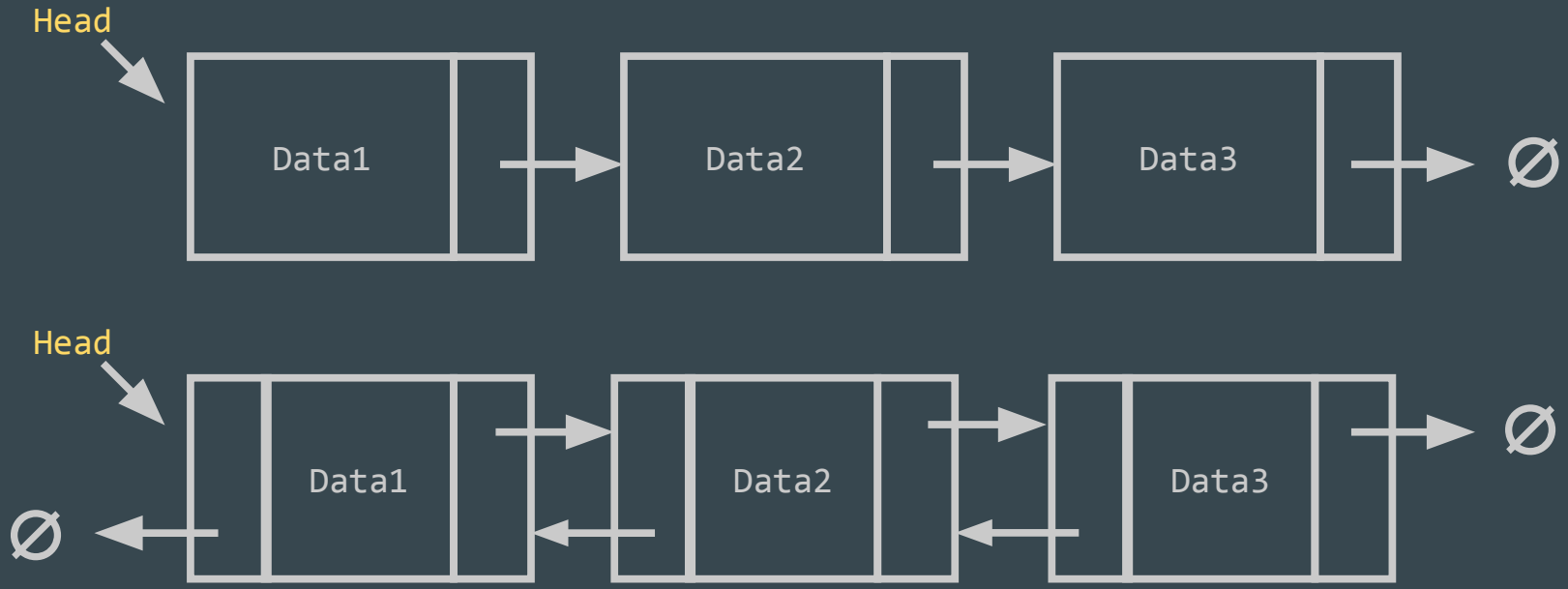
- Dynamically allocate records (“nodes”)
- Each node holds a data item and one or more links to other nodes in the list
- If one knows where the list begins (“head”), one can follow the links to find the rest of the list
- Must remember to mark the last node so that one can tell when the end of list has been reached (NULL is found in the cstddef library)



⌀ is used to denote a pointer whose value is NULL

Why use dynamic allocation?

Singly-Linked vs Doubly-Linked Lists



⌀ is used to denote a pointer whose value is NULL

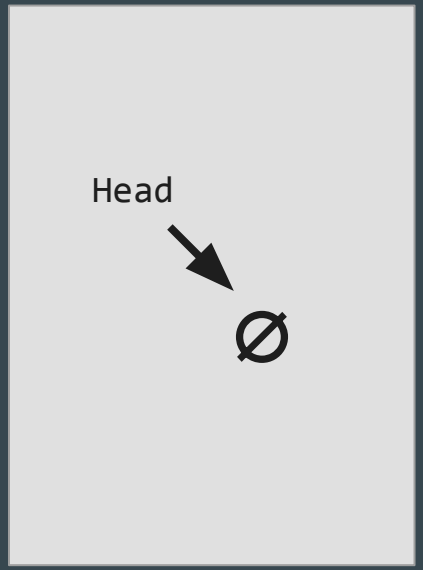
Basic Operations - Create

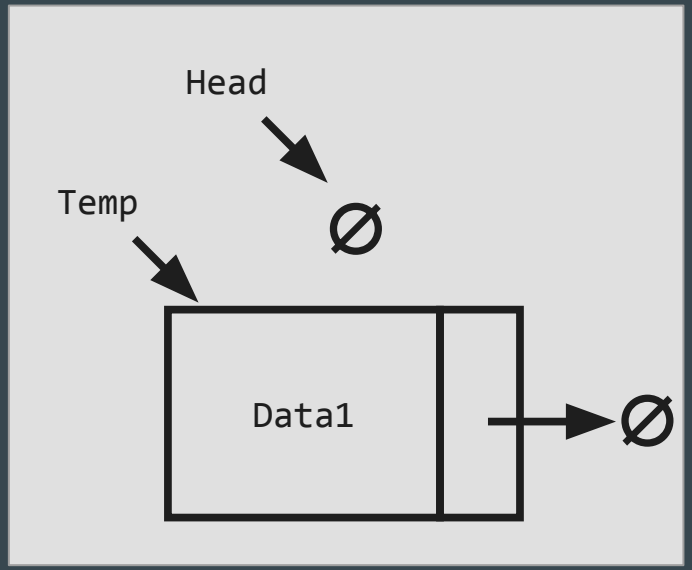
- Constructor creates an instance of the class
- *Head* is set to NULL to indicate that the list is empty

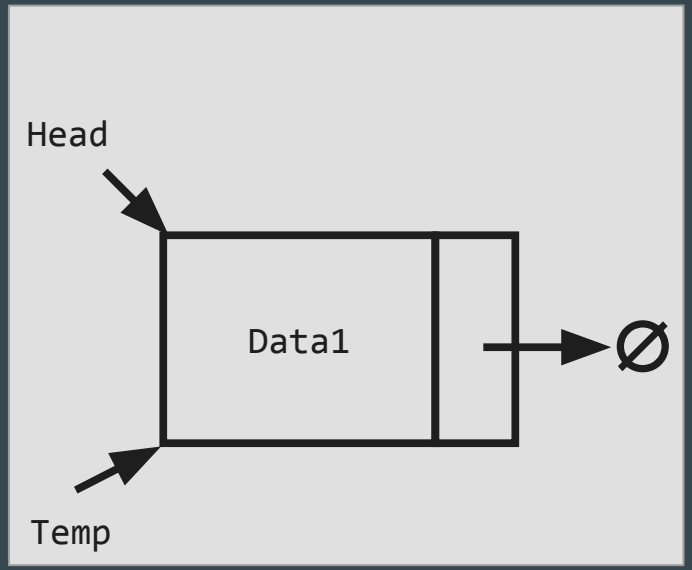
⌀ is used to denote a pointer whose value is NULL

Basic Operations - Insert

Case 1: Insert into an empty list



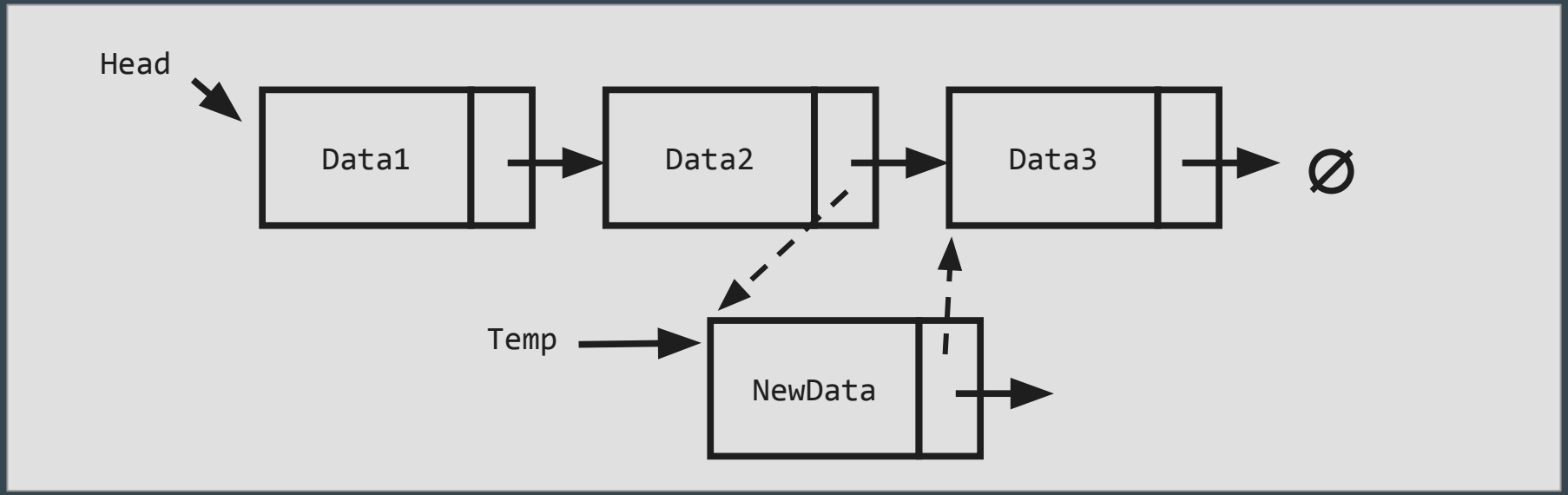




⌀ is used to denote a pointer whose value is NULL

Basic Operations - Insert

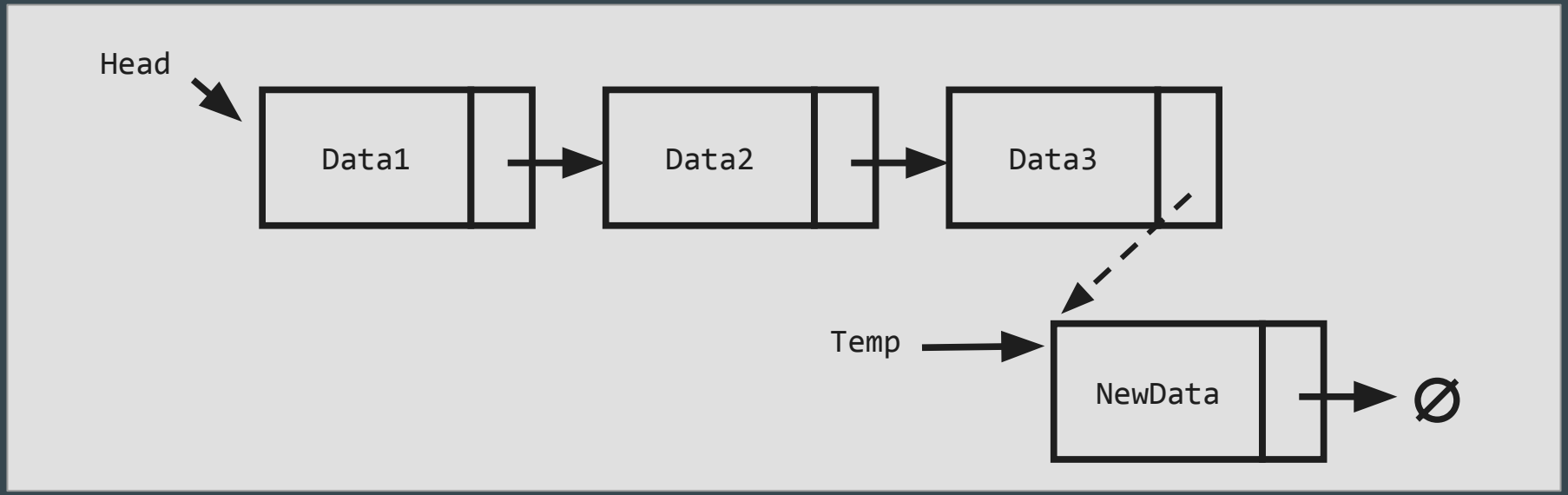
Case 2: Insert into an non-empty list



⌀ is used to denote a pointer whose value is NULL

Basic Operations - Insert

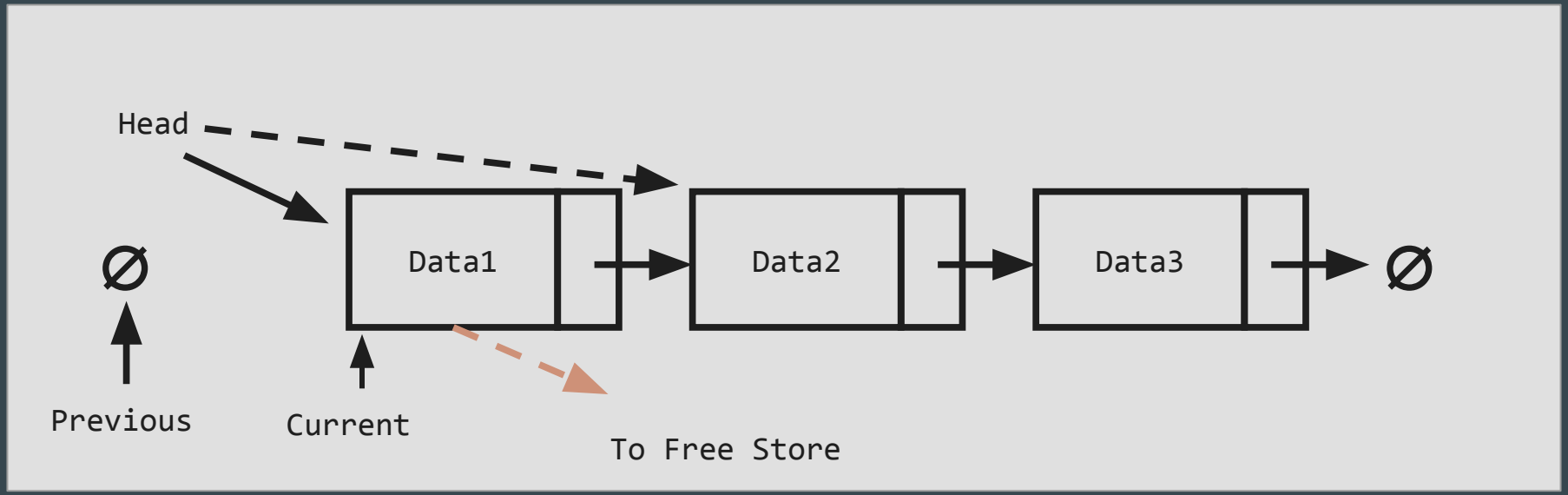
Case 3: Insert into an non-empty list at the end of the list



∅ is used to denote a pointer whose value is NULL

Basic Operations - Delete

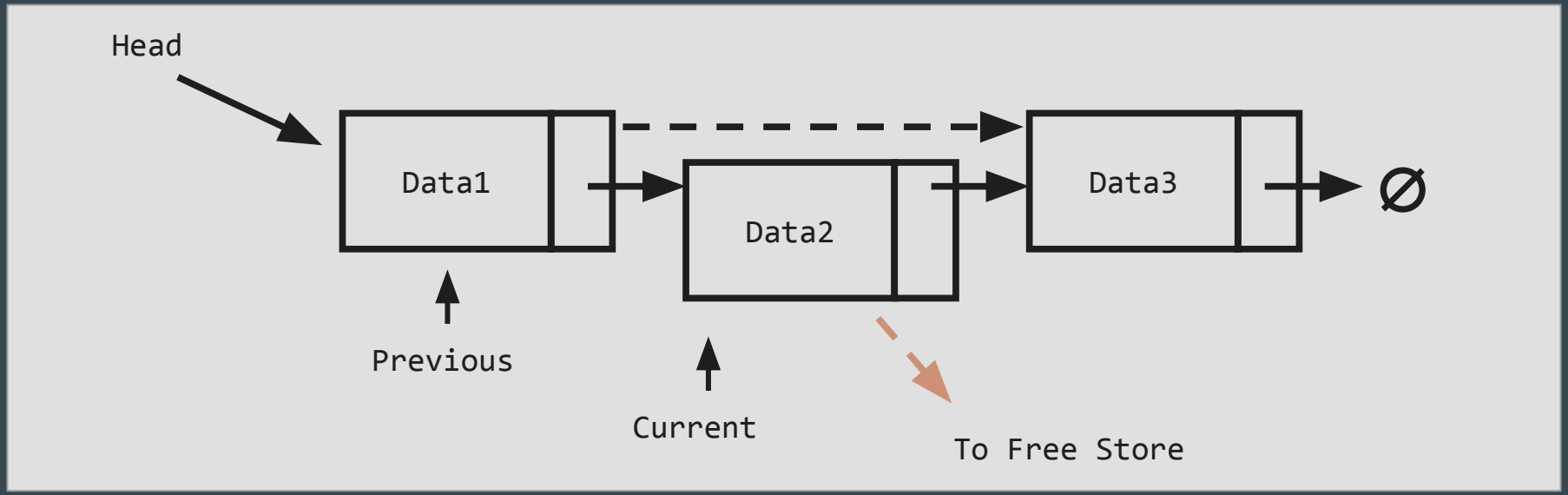
Case 1: Delete the first node in the list



⌀ is used to denote a pointer whose value is NULL

Basic Operations - Delete

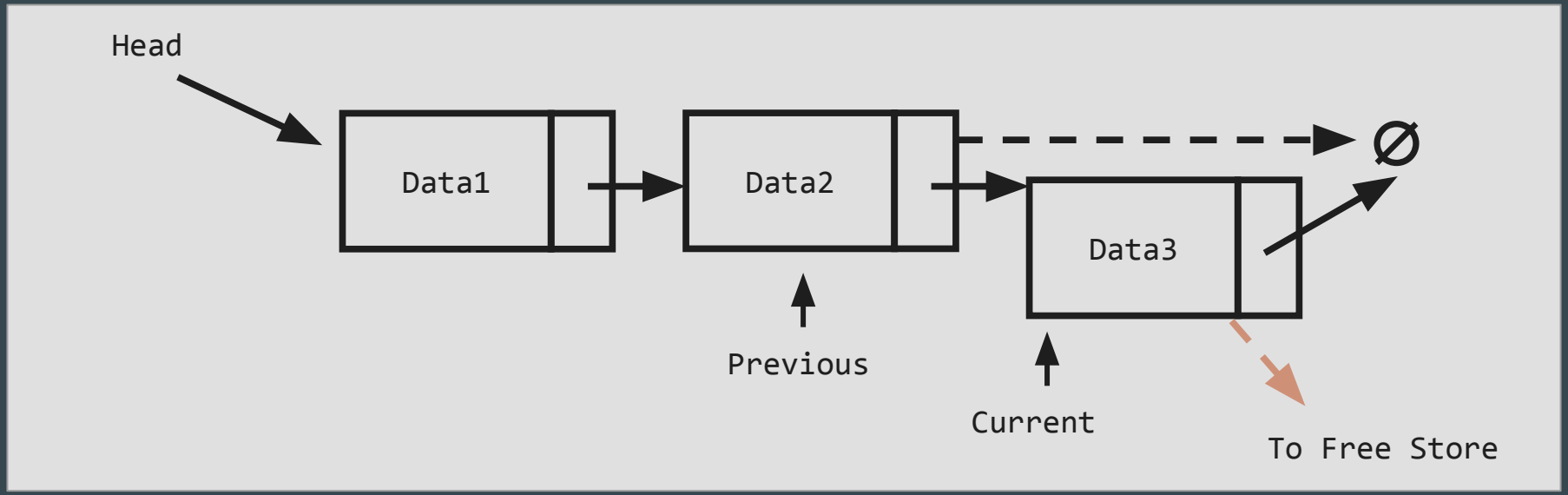
Case 2: Delete a middle node in a list



⌀ is used to denote a pointer whose value is NULL

Basic Operations - Delete

Case 2: Delete the last node in a list



⌀ is used to denote a pointer whose value is NULL

Link List Example LList.h

```
#ifndef LINKED_LIST_LLIST_H
#define LINKED_LIST_LLIST_H

struct NodeType;            // Forward declaration, complete declaration in
LList.cpp

typedef int ItemType;       // Data type of each item in list

class LList {
private:
   NodeType* head;          // Head of linked list

public:
   /**
    * Default constructor
    * @post Empty list created
    */
   LList();

   /**
    * Method to check to see if the list is empty
    * @post Returns TRUE if empty else FALSE
    */
   bool IsEmpty() const;
```

Link List Example LList.h

```
/**
    * Method to print the contents of the list
    * @post The list contents, if any, are printed to the console
    */
   void Print() const;

   /**
    * Method to add an item to the beginning of the list
    * @param item ItemType
    * @pre Item is less than the first item in the list
    * @post Item is first in the list and the list items are in order
    */
   void InsertAsFirst(ItemType item);


   /**
    * Method to insert an item in the correct place in the list
    * @param item ItemType
    * @pre The items in the list are in ascending order
    * @post The new node containing the item is added to the sorted list
    */
   void Insert(ItemType item);
```

Link List Example LList.h

```
/**
    * Method to remove the first item in the list
    * @param item ItemType&
    * @post The first item in the list is removed and the head is now pointing to the
next node
    */
   void RemoveFirst(ItemType& item);

   /**
    * Method to delete an item from the list
    * @param item ItemType
    * @pre Item is in the list
    * @post The first occurrence of the item is no longer in the list and the list
remains sorted
    */
   void Delete(ItemType item);

   /**
    * Destructor
    * @post List is destroyed
    */
   ~LList();
};


#endif //LINKED_LIST_LLIST_H
```

Link List Example LList.cpp

```cpp
#include <cstddef>
#include <iostream>
#include "LList.h"

using namespace std;

struct NodeType {
   ItemType value;
   NodeType* nextNode;
};

LList::LList() {
   head = NULL;
}

LList::~LList() {
   ItemType someItem;

   while ( !IsEmpty() ) {
       RemoveFirst(someItem);
   }
}
```

Link List Example LList.cpp

```
bool LList::IsEmpty() const {
   return (head == NULL);
}

void LList::Print() const {
   cout << "Printing the list" << endl;
   NodeType* currentNodePtr = head;

   while (currentNodePtr != NULL) {
       cout << currentNodePtr->value << endl;
       currentNodePtr = currentNodePtr->nextNode;
   }
}

void LList::InsertAsFirst(ItemType item) {
   NodeType* tempPtr = new NodeType;

   tempPtr->value = item;
   tempPtr->nextNode = head;
   head = tempPtr;
}
```

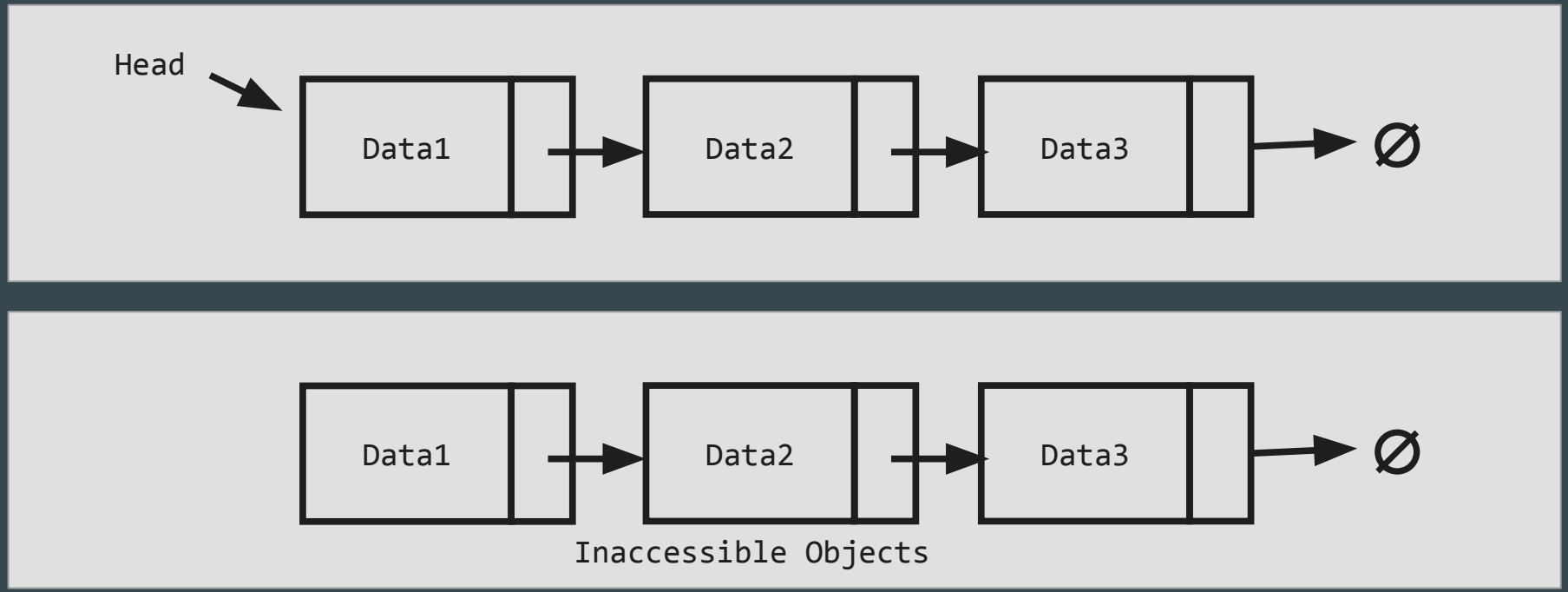
Link List Example
LList.cpp

```
void LList::RemoveFirst(ItemType &item) {
   NodeType* tempPtr = head;
   item = head->value;
   head = head->nextNode;
   delete tempPtr;
}

void LList::Insert(ItemType item)
{
 // Create node and initialize

 // Scan list to locate insertion point  <<=== Think about the details which go here

 // Insert the new node            Hint: look at the pictures

}


void LList::Delete(ItemType item)
{
 // If item first node in list
 // then delete first node

 // otherwise, look for item in rest of list  <<=== Think about the details which go
here
 // and delete it               Hint: look at the pictures
}
```

Linked List Length

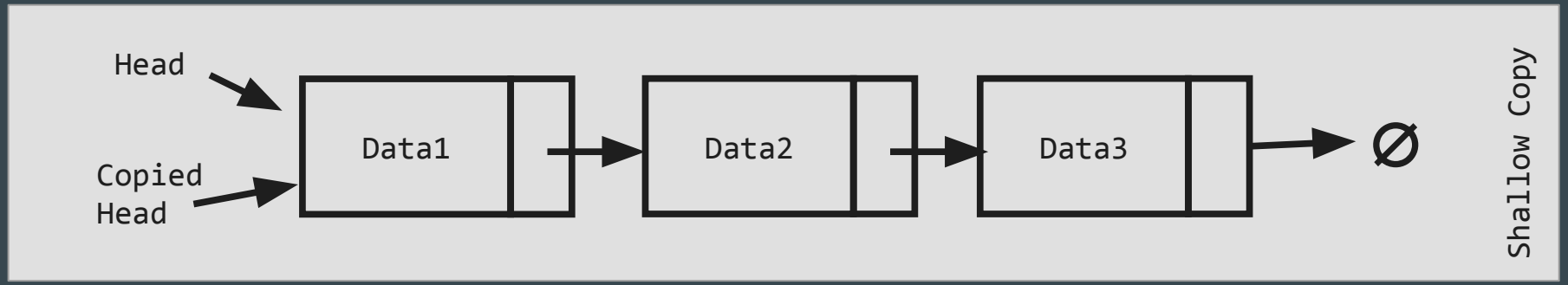
- Option 1
  - Write a method which scans the list to count the total number of items on demand
- Option 2
  - Add an additional private counter attribute called length which is initialized by the constructor and updated by the transformers as nodes are added and removed

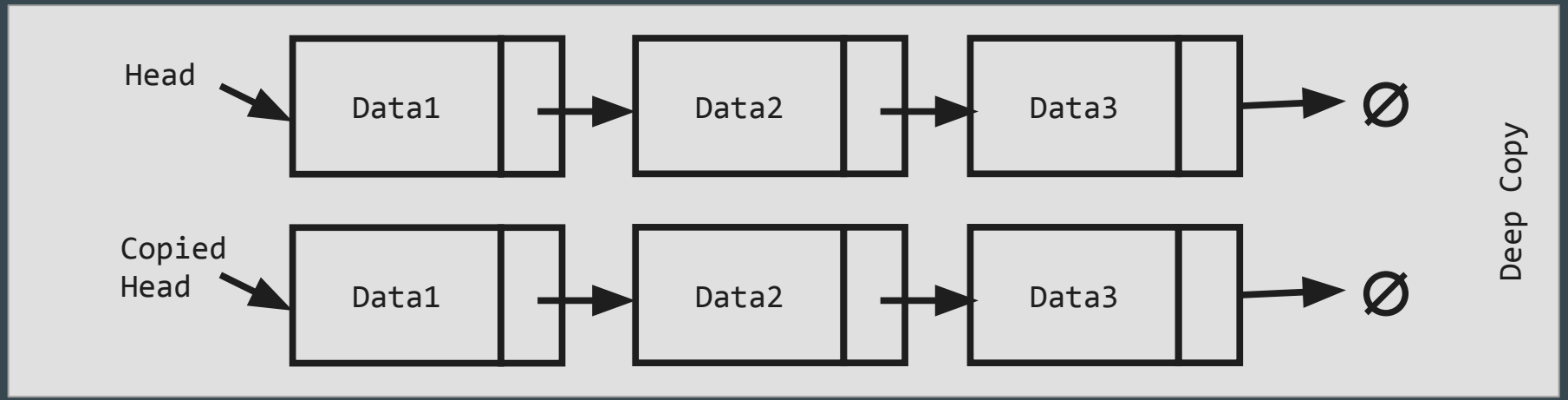
Basic Operations - Destructor



⌀ is used to denote a pointer whose value is NULL

Basic Operations - Copy





Array vs Linked List

Operation	Array Lists	Linked Lists
Sizing/Resizing	Inefficient	Efficient
Search	Cost Bounded	Cost Variable
Insert*	Inefficient	Efficient
Delete*	Inefficient	Efficient
Random Access	Efficient	Inefficient