ITT DUBLIN
Institute of Technology Tallaght
Institiúid Teicneolaíochta Tamhlacht

Linked Lists

Maintaining data in a list

- Any data items can be maintained in a list: it could be a simple list of integers, or a list of characters, or a more complex list, like a list of order-lines where each order-line is a struct like the one here
- We would like to maintain the list by:
  - Adding another item to the beginning or end
  - Inserting an item into the middle (say of a sorted list)
  - Deleting an item from anywhere in the list
- We would also like to be able to
  - Sort the list
  - Search the list for a given item
- And it must be very efficient.
- How can we do this?

```
struct Orderline
{
string name;
int quantity;
};
```

Set up an array?

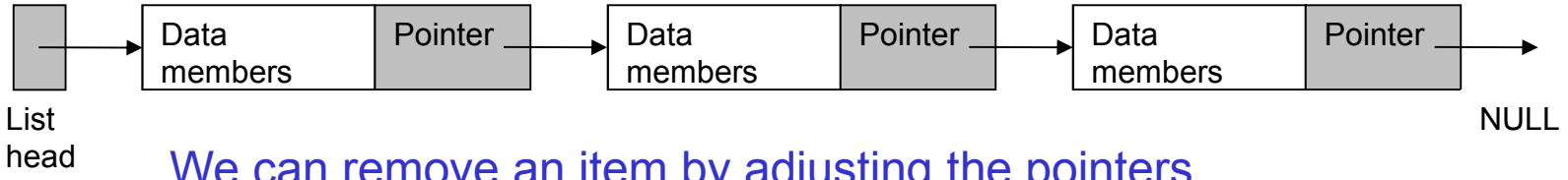
- How can we be sure the array is large enough?
- How will we add another item at the beginning of the array?
- How will we delete an item from the middle of the array?

Use the vector class of the STL?

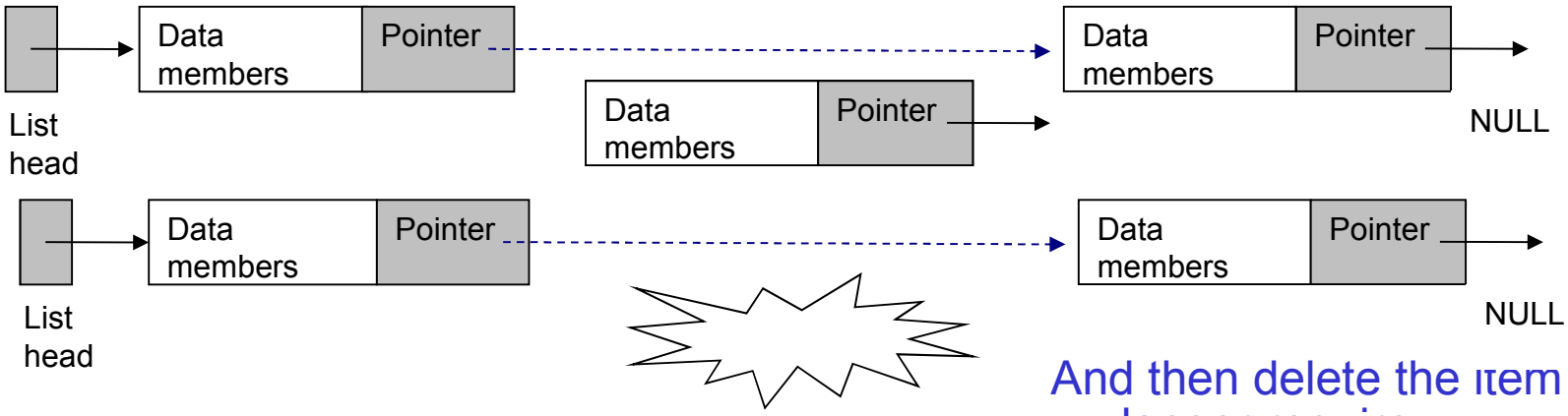
- What is the performance overhead of inserting/deleting an item in the array?
- Even more so if the vector must be resized to accommodate an insertion.

Maintaining data in a list: some solutions

- It would be possible to write all the functions we needed to manipulate an array of these data items.`
  - It would be quite complex to write them
  - It would involve a lot of moving data around
    - For example, to make a space in the array for a new entry, we would have to move all the later items down one place in the array, And make sure the array had some empty space at the end to do this
  - Or use the vector class which does all this for us!
    - Less effort for us, but still the same performance overhead
- A simpler solution is to use a linked data structure, where each data item points to the next one

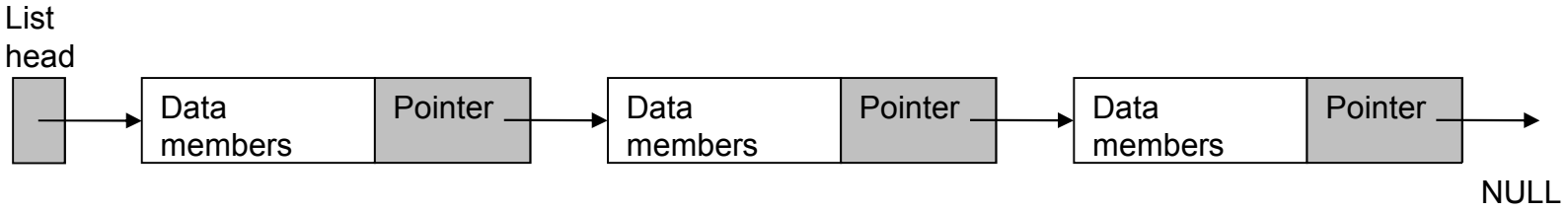


We can remove an item by adjusting the pointers

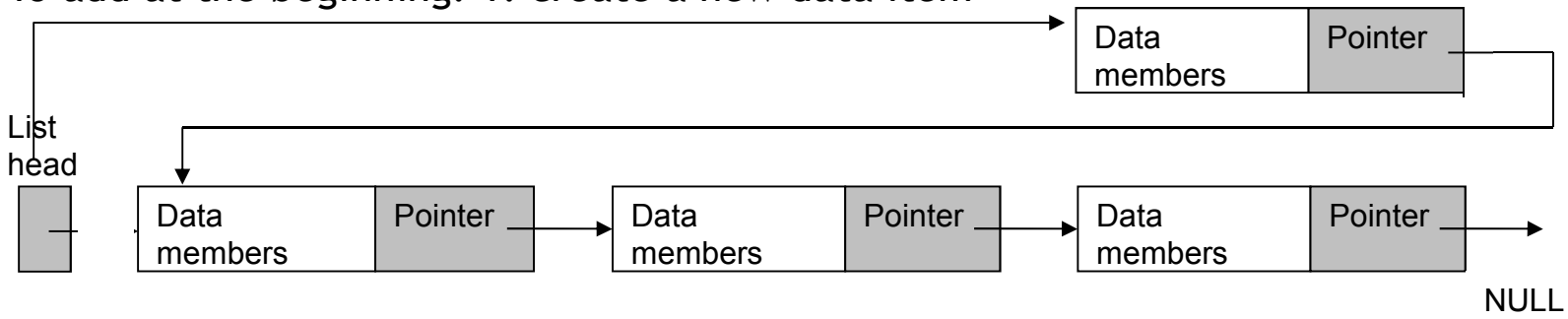


And then delete the item we no longer require

Adding a data item to a linked list



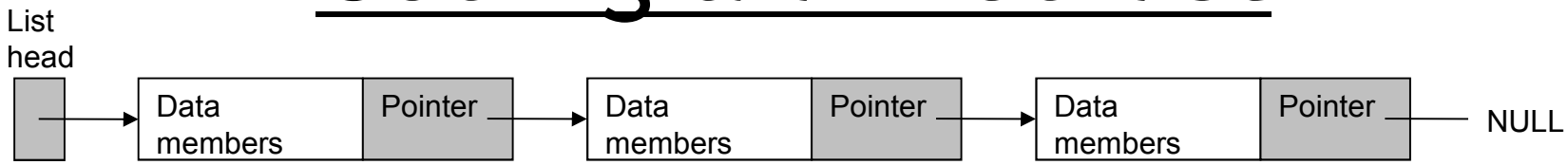
To add at the beginning: 1. create a new data item



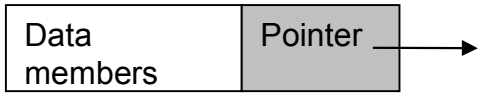
2. Set its pointer to the same value as the list head pointer
3. Move the list head pointer to point to it

To add somewhere in the middle would involve manipulating a few more pointers!

Coding a linked list



- We call each of the structures, which holds the data and a pointer to the next one, a *List node*



The list node

- To fully define a list, all we need to know is where it starts, that is the value in the *list head pointer*
- If we define the list as a class, we can add the methods which we want to call to manipulate the list
- We have defined a new Abstract Data Type (ADT)
- Of course, we still have to define the `ListNode` datatype for this code to make sense

```
//an Abstract Data Type
class LinkedList
{
  public:
    LinkedList();
    ~Linked List();
    void insert(Data);
    void delete (int pos, Data);
    void displayList();
  private:
    ListNode *head;
};
```

Defining the ListNode

In this example, the type of the data stored in the `ListNode` is `Data` (could be an `int`, or a string, or an `OrderLine`, or ..)

```
class ListNode {
public:
   ListNode(Data);
private:
   Data theData;
   ListNode *next;
};
typedef ListNode* ListNodePtr;
```

the typedef is optional, it means we can refer to `ListNodePtr` instead of remembering the *notation all the time.

```
class LinkedList
{
  public:
    LinkedList();
    ~Linked List();
    void addNode(Data);
    void deleteNode(int pos, Data);
    void displayList();
  private:
    ListNode* head;
};
```

Now all we have to do is code the constructors for all our classes, and the methods for the LinkedList class

Exercise: re-write using the `ListNodePtr` type instead of `ListNode*`

The Linked List Constructor

```
class LinkedList
{
  public:
      LinkedList();
      ~Linked List();
      void addNode(Data);
      void deleteNode(int, Data);
      void displayList();
  private:
      ListNode* head;
};
```

```
LinkedList::LinkedList()
   :head(NULL) //an empty list to start
{}
```

For the rest of the methods, we need to think about some pseudocode, and look at some diagrams. Once we have figured out what is happening, the code will be quite simple.

Adding a node to the list

- The list class in the STL
  - yes – there is one!
- has the following methods to add to the list:
  - `push_back(Data & )` //adds to the end of the list
  - `push_front(Data & )` //adds to the beginning of the list
  - `insert (int pos, Data &)`
    - actually that last one is not quite as the STL specifies – in the STL it has an *iterator* as the first argument to show where the next item is to be inserted
  - Notice that names match up with the ones for the vector class!
  - Our `addNode` method will add to the beginning of the list, like `push_front`, because that is the easiest thing to do!

LinkedList::addNode

```
// Create a new Node "newNode" and store the data in it.
   ListNode *newNode = new ListNode(nodeData);

// set pointer to point to the present first node on the list
   newNode->next = head

// set this node to be the head of the list
   head = newNode;
```

- Notice we are assuming that the `LinkedList` class method can access the data member '`next`' of the `ListNode`. We need to make a change to the `ListNode` class to enable this.

Granting ‘friend’ status to another class

```
class ListNode {
friend class List;
public:
   ListNode(Data);
private:
   Data theData;
   ListNode *next;
};
```

- All the methods of the `List` class now have access to the private data in the `ListNode` class
- Friendship must be granted, it cannot be demanded
- Notice that the friend declaration is inside the class, but access specifiers (public/private) are irrelevant.
  - we normally put it at the very top.

LinkedList::displayList()

- Starting with the head pointer, traverse the list and print out the details of each item.

We will assume that:

- either the data stored in a node is a primitive type
- or we have defined how to print objects of type Data (ie overloaded the << operator for this type)

```
LinkedList::displayList()
{
  ListNodePtr tempptr = head;
  while (tempptr  != NULL)
  {
    cout << tempptr->theData;
    tempptr = tempptr->next;
  }
}
```

LinkedList::appendNode(Data)

- We could add a method to our list to append to the list
- appending a node to a list means adding it to the end of the list, so this is like the push_back method in the STL
- The top level pseudo code for this is shown below:

```
Create a new node.
Store data in the new node.
If (there are no nodes in the list)
    Make the new node the first node.
Else
   Traverse the List to Find the last node.
   Add the new node to the end of the list.
End Else
```

(We might need to expand out the pseudo code for the operations shown underlined)

```
void LinkedList::appendNode(Data nodeData)
{
   // Create a new Node "newNode" and store the data in it.
    ListNode *newNode = new ListNode(nodeData);
   .
   ListNode* nodePtr; // Create a temporary pointer to traverse the list.

// If (there are no nodes in the list) , Make the new node the first node
   if (!head) // if head is null - i.e. an empty list
    head = newNode;
   else
   {
    // else traverse the list to find the last node
    for (nodePtr = head; nodePtr->next; nodePtr =
                        nodePtr-> next);
    // and add the new node to the end of the list.
    nodePtr->next = newNode;
   }
}
```

; at the end means this is an empty for loop!

exercise: what is happening in the for loop above? How do we know the node ptr is pointing to the last node?

LinkedList::deletNode(Data)

- Deleting the node with the data shown will require us to 'traverse the list' until we find a node holding that data, then adjust the pointers and delete the node. The pseudo code for this is shown below:
- Note how we keep tabs all the time on 2 nodes with our pointers. *Why?*

```
If (data to be deleted matches the first node in the list)
    assign the head pointer to a temporary pointer
    Update the head pointer to point to the next node
    Delete the first node (still accessible through the temporary pointer)
else
    Traverse the list with 2 pointers until the leading pointer points to the node
    holding the data, or the leading pointer is NULL
End if

If the leading pointer is NULL (we didn't find the data in the list)
    return
Else
     Update pointer in the node pointed to by the trailing pointer to bypass
                                the node pointed to by the leading pointer
     delete the node pointed to by the leading pointer
```

```
void LinkedList::deletNode(Data toDelete) {
   ListNode *leadPtr = head, *trailPtr = NULL;
```

If (data to be deleted matches the first node in the list)

Update the head pointer to point to the next node and Delete the first node

```
   if (head->theData == toDelete) {
        head = head->next;
        delete leadPtr;
   }
```

Danger point here – did we check that the list wasn't empty? – if head is NULL, then 'head->theData' will cause a crash!

Else Traverse the list *with 2 pointers* until the leading pointer points to the node holding the data, or the leading pointer is NULL

```
   else {
        while (leadPtr != NULL && leadPtr->theData != toDelete) {
            trailPtr = leadPtr;
            leadPtr = leadPtr->next;
        }
   }
```

If the leading pointer is NULL (we didn't find the data in the list) return

```
   if (leadPtr == NULL)
        return;
```

Else Update pointer in the node pointed to by the trailing pointer to bypass the node pointed to by the leading pointer, and delete the node pointed to by the leading pointer

```
   else {
        trailPtr->next = leadPtr->next;
        delete leadPtr;
   }
}
```

Exercise: add code for the destructor of the List, LinkedList::~LinkedList()

While list is not empty

```
while (head) {
```

set a temporary pointer to point to the first item on the list

```
tempPtr = head;
```

move the head pointer to the next node on the list (or null if there are no more nodes).

```
head = head->next;
```

delete the node pointed to by the temp ptr

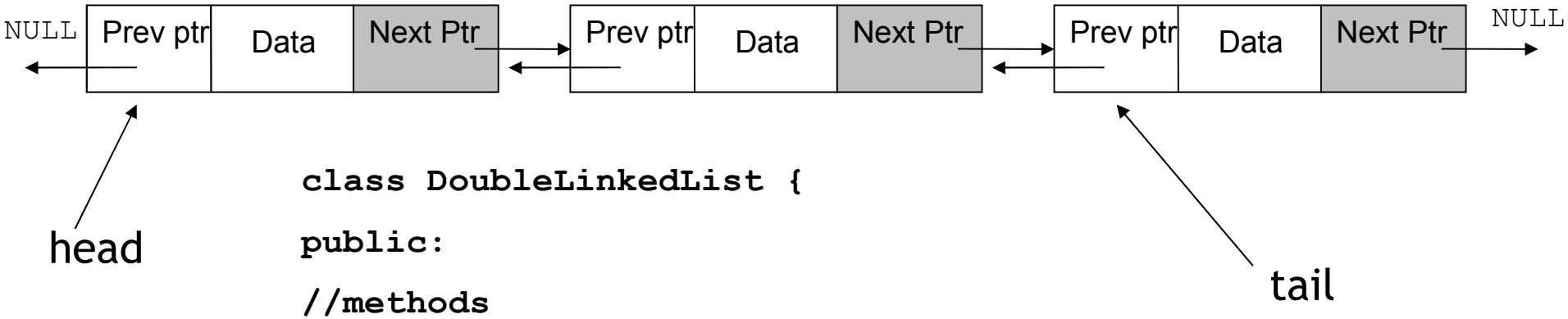
```
delete tempPtr;
```

End while

```
}
```

A doubly linked list

- For some operations at the end of the list, we always traverse the whole list to find the end.
- This would be much easier if a pointer was maintained to the end of the list as well as the head.
- There are many alternative ways to implement a linked list, one which also uses a tail (lastPtr) as well as a head (firstPtr) is the doubly-linked list.
- In this implementation, each node contains 2 pointers, one forwards and one backwards, so it is very easy to move both up and down the list.

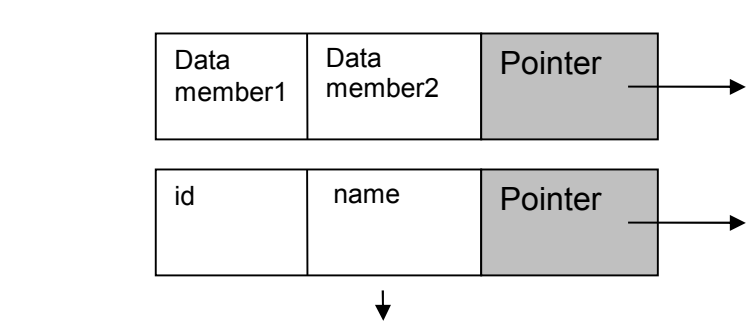


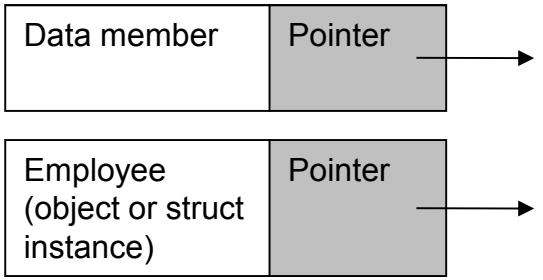

```
class DoubleLinkedList {
public:
//methods
private:
ListNode *head;
ListNode *tail;
};
```

Linked Lists and Memory Management

- The linked list class contains one private data element, a pointer to the head of the list (assuming you are not also using a pointer to the end of the list)
- If that pointer is deleted, or re-assigned, then there is no way to access the elements of the list!
- The list will include public methods to manipulate the items on the list,
- It will also have a constructor to set up a list. These assign a value to the head pointer (and the tail pointer if used) – probably NULL, since nothing has been add to the list yet.
  - *The constructor does not need to allocate any memory*
- Whenever an item is added to the list, a node is created dynamically using `new`.
- Whenever a node is deleted from the list, the node must be deleted to free the memory which was allocated for it when it was constructed.
- When the list itself is deleted, then the memory allocated to every node must be freed.
  - do this by traversing the list and deleting each node in turn
  - But be careful to make sure you have a way of getting to the 'next' node before a node is deleted!

- The examples we have looked at have been linked lists storing one piece of data (type Data) at each list position.
- But a linked list may contain more than one value at each position in the list, or may contain an object of a class (e.g. an Orderline class). It is the **definition of the Node class** which determines what the linked list is holding.






```
class ListNode {
public:
  ListNode(int = 0, string = "", next = 0)
private:
  int id;
  string name;
  ListNode *next;
};
```

```
class ListNode {
public:
  ListNode(int = 0, string = "", next =
0)
private:
  Employee emp;
  ListNode *next;
};
```

Code example

Employee class declares the EmpNode class to be a friend, so that the constructor in the node can easily set the values for id and name

```
class Employee
{
  friend class EmpNode;
public:
  Employee();
private:
  int id;
  string name;
};
```

```
class EmpNode
{
  friend class listOfEmp;
 public:
    EmpNode{int id, string name};
 private:
    Employee emp;
    Node *next;
};
```

EmpNode class declares the List OfEmp class to be a friend, so that the methods in the list can easily access the emp and the next pointer

```
class ListOfEmp
{
public:
  ListOfEmp();
  ~ListOfEmp();
  bool findEmp(int id, Employee &Emp) const;
  bool insertEmp(int id, string name);
private:
  EmpNode *head;
};
```

Implementing the constructor in the EmpNode

```
class EmpNode
{
  friend class listOfEmp;
 public:
    EmpNode{int id, string name};
 private:
    Employee emp;
    EmpNode *next;
};
```

This is the EmpNode class we defined

```
EmpNode::EmpNode{int id, string
name}
{
     emp.id = id;
     emp.name = name;
     next = NULL;
}
```

Question: why did we not use `emp->id` and `emp->next`?

Answer: because the `emp` data member here is an `Employee`, *not a pointer to an* *`Employee`*

We can directly update the id and the name in the `Employee` object that is a data member of the `EmpNode`, because `Employee` class declared `EmpNode` as a friend' otherwise we would have to supply public 'set' methods in the `Employee` class, and call these.

Implementing some methods of the EmpList

```
bool EmpList::insertEmp(int id, string
   name)
{
   EmpNode *newNode = new EmpNode(id,
   name);
   if (!newNode) return false; //failure
   newNode->next = head;
   head = newNode;
   return true; //success
}
```

This code would all look exactly the same if Employee had been a structure instead of a class

```
bool EmpList::findEmp(int id, const Employee &emp) const
{
  EmpNode *currentNode = head;
  while (currentNode != 0)
  {
    if (currentNode->emp.id == id)
    {
      emp = currentNode->emp;
      return true;
    }
    currentNode = currentNode->next;
  }
  return false;
}
```