Data Structures and Algorithms

Dr. Salwa Osama

Chapter 2

Lists

Contents

Objectives

- To study List as an ADT
- Build a static-array-based implementation of Lists and note strengths, weaknesses
- Build a dynamic-array-based implementation of Lists, noting strengths and weaknesses
    - See need for destructor, copy constructor, assignment methods
- Take first look at linked lists, note strengths, weaknesses
- Study pointer-based implementation of linked lists

Consider Everyday Lists

- Groceries to be purchased
- Job to-do list
- List of assignments for a course
- Dean's list

- Can you name some others??

Properties of Lists

a list is a collection of things. we recognize that there are certain common properties of the collection in each list:

- Can have a single element
- Can have <u>no</u> elements
- There can be list of lists

We will look at the list as an abstract data type

- Homogeneous
- Finite length
- Sequential arranged elements

Basic Operations

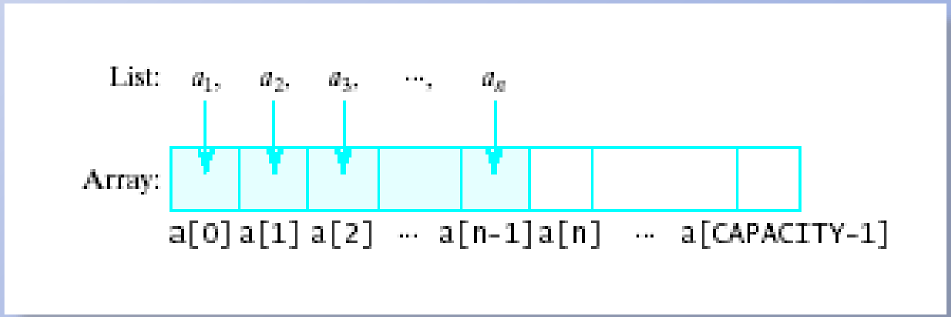
- Construct an empty list
- Determine whether or not empty
- Insert an element into the list
- Delete an element from the list
- Traverse (iterate through) the list to
  - Modify
  - Output
  - Search for a specific value
  - Copy or save
  - Rearrange

Designing a `List` Class

- Should contain at least the following function members
  - Constructor (List)
  - `empty()`
  - `insert()`
  - `erase()`
  - `display()`
- Implementation involves
  - Defining data members
  - Defining function members from design phase

1- Array-Based Implementation of Lists

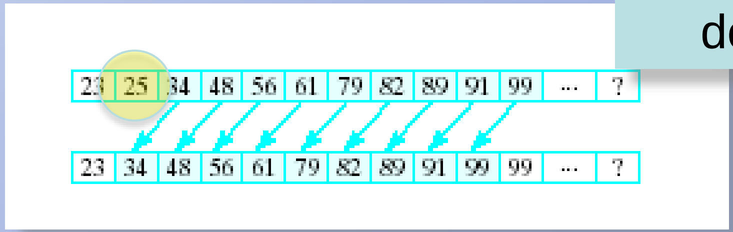
- An array is a viable choice for storing list elements, why?
  - Element are sequential
  - It is a commonly available data type
  - Algorithm development is easy
- Normally sequential orderings of list elements match with array elements



Implementing Operations

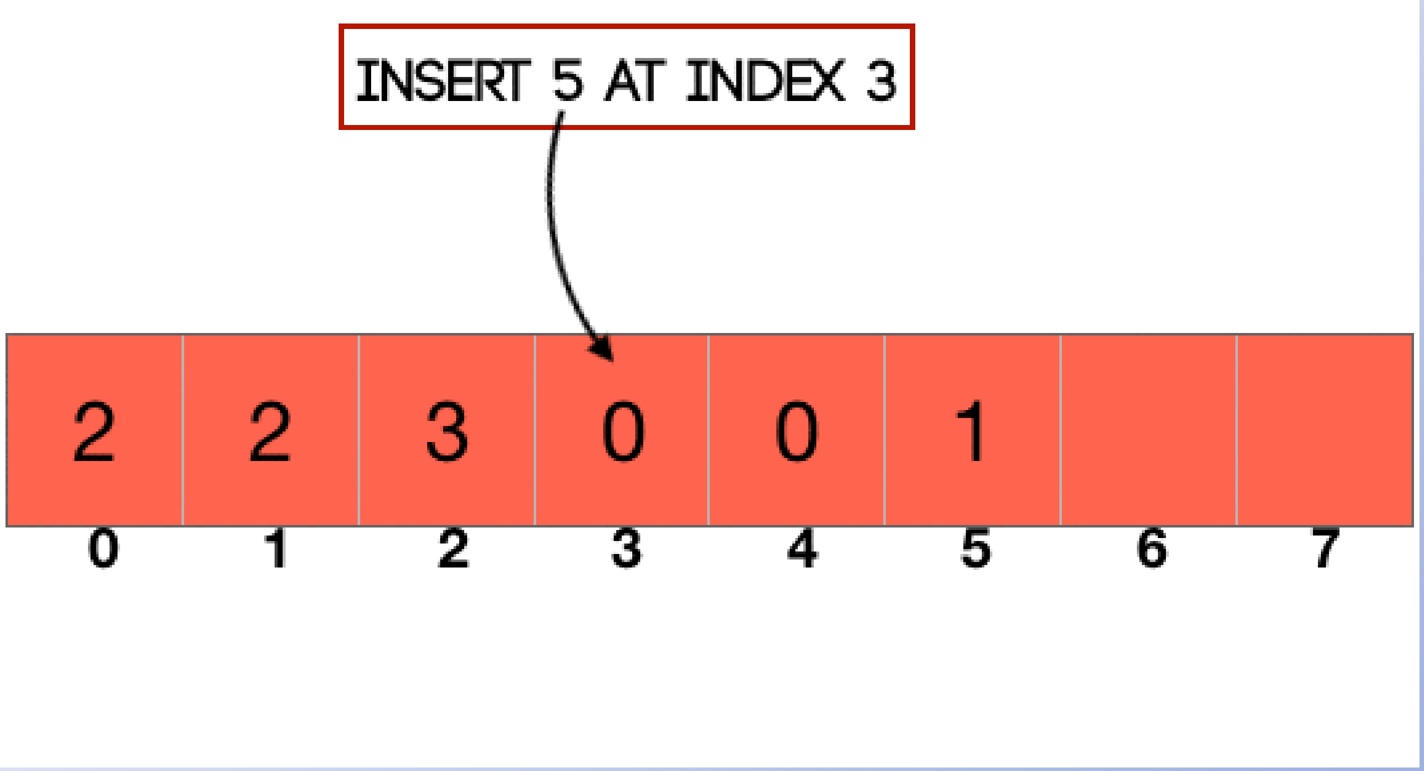
- Constructor
  - Static array allocated at compile time
- Empty
  - Check if size == 0
- Traverse
  - Use a loop from 0th element to `size – 1`
- Insert
  - Shift elements to right of insertion point
- Erase
  - Shift elements back to erase element

Also adjust `size` up or down

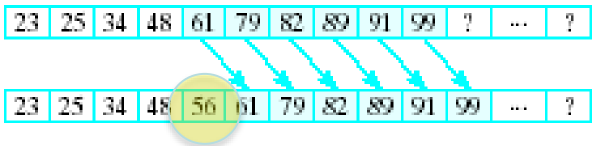


1-Array-Based Implementation of Lists

Insert Algorithm



Insert Algorithm



//--- Insert *item* at position *pos* in a list.

// First check if there's room in the array

1. If *size* is equal to *capacity*
   Issue an error message and terminate this operation.

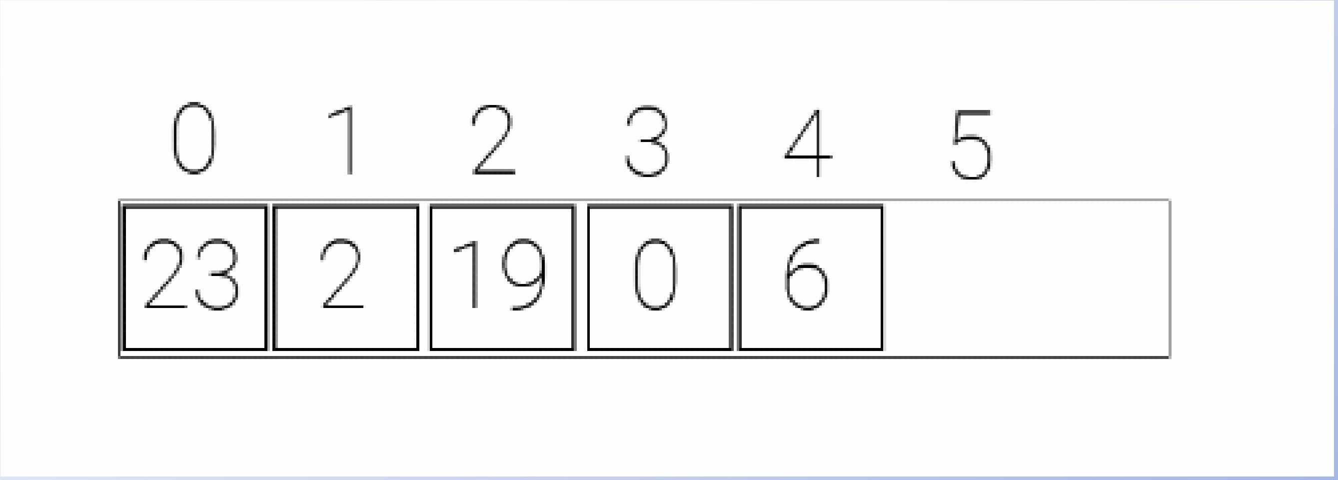
// Next check if the position is legal.

2. If *pos* < 0 or *pos* > *size*
   Signal an illegal insert position and terminate this operation.
   Otherwise:
   // Shift array elements right to make room for *item*
   a. For *i* ranging from *size* down to *pos* + 1:
      array[*i*] = *array*[*i* – 1]
   // Now insert *item* at position *pos* and increase the list size
   b. *array*[*pos*] = *item*
   c. *size*++

What is the worst case: O(*n*)
What is the best case: O(*1*)

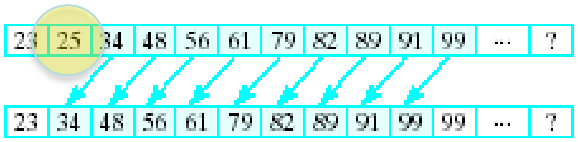
1-Array-Based Implementation of Lists

Erase Algorithm



Erase Algorithm

//--- Delete the element at position *pos* in a list.



// First check that list isn't empty

1. If *size* is 0

    Issue an error message and terminate this operation.

// Next check that *index* is legal

2. If *pos* < 0 or *pos* ≥ *size*

    Issue an error message and terminate this operation.

   Otherwise:

    // Shift array elements left to close the gap

    a. For index *i* ranging from *pos* to *size* – 2:

        array[*i*] = *array*[*i* + 1]

    // Decrease the list size

    b. *size*– –

What is the worst case: **O(*n*)**
What is the best case: O(*1*)

List Class with Static Array

- Must deal with issue of declaration of CAPACITY
- Use **typedef** mechanism

```
typedef Some_Specific_Type ElementType
typedef int ElementType;
ElementType array[CAPACITY];
```

- For specific implementation of our class, we simply fill in desired type for **Some_Specific_Type**

1-Array-Based Implementation of Lists

`List` Class with Static Array

- Can put `typedef` declaration inside or outside of class
  - Inside: must specify `List::ElementType` for reference to the type outside the class
  - Outside: now able to use the `template` mechanism (this will be our choice)
- Also specify the `CAPACITY` as a `const`
  - Also choose to declare outside class

`List` Class with Static Array

- Can put **`typedef`** declaration inside or outside of class

```cpp
class List {
public:
    typedef int ElementType; // Nested typedef
    void print(ElementType value);
};

void List::print(ElementType value) { // ❌ Error: ElementType is not recognized outside
    std::cout << value << std::endl;
}

void List::print(List::ElementType value) { // ✅ Correct way
    std::cout << value << std::endl;
}
```

`List` Class Example

- Declaration file, List.h
  - Note use of `typedef` mechanism outside the class
  - This example good for a list of `int`
- Definition, implementation List.cpp
  - Note considerable steps required for `insert()` and `erase()` functions
- Program to test the class, main.cpp

`List` Class with Static Array- Problems

- Stuck with "one size fits all"
  - Could be wasting space
  - Could run out of space
- Better to have instantiation of specific list specify what the capacity should be
- Thus, we consider creating a `List` class with dynamically-allocated array

2- Dynamic-Allocation for `List` Class

- Changes required in data members:
  - Eliminate const declaration for CAPACITY
  - Add variable data member to store capacity specified by client program
  - Change array data member to a pointer
  - Constructor requires considerable change
- Little or no changes required for:
  - `empty()`
  - `display()`
  - `erase()`
  - `insert()`

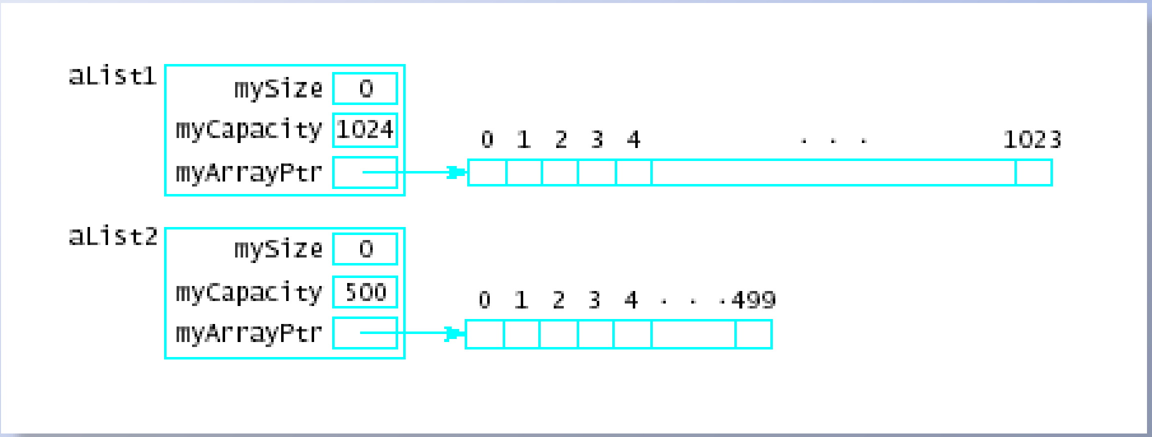
2-Dynamic-Allocation of List Class Example

- Note data changes in, `List.h`
- Note implementation file `List.cpp`
  - Changes to constructor
  - Addition of other functions to deal with dynamically allocated memory
- Note testing of various features in the demo program

2-Dynamic Allocation of List
Dynamic-Allocation for `List` Class

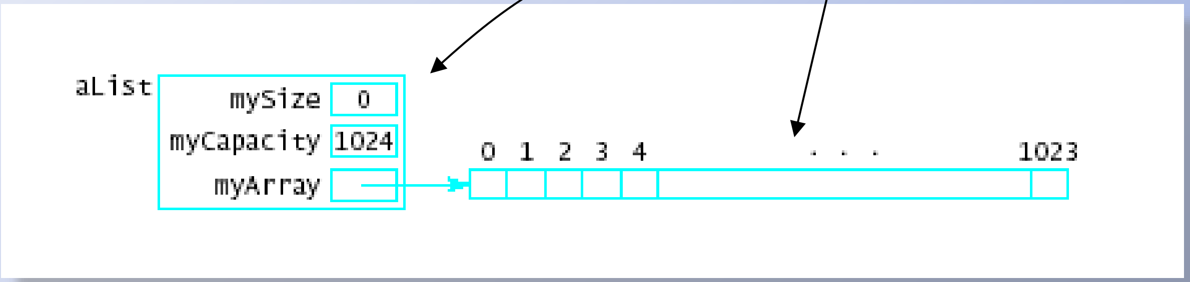
- Now possible to specify different sized lists

```
cin >> maxListSize;
List aList1 (maxListSize);
List aList2 (500);
```



2-Dynamic Allocation of List
New Functions Needed

- Destructor
  - When class object goes out of scope the pointer to the dynamically allocated memory is reclaimed automatically
  - The dynamically allocated memory is not



  - The destructor reclaims dynamically allocated memory

2-Dynamic Allocation of List

Destructor

The Class Destructor

Forms:

~ClassName()

Purpose:

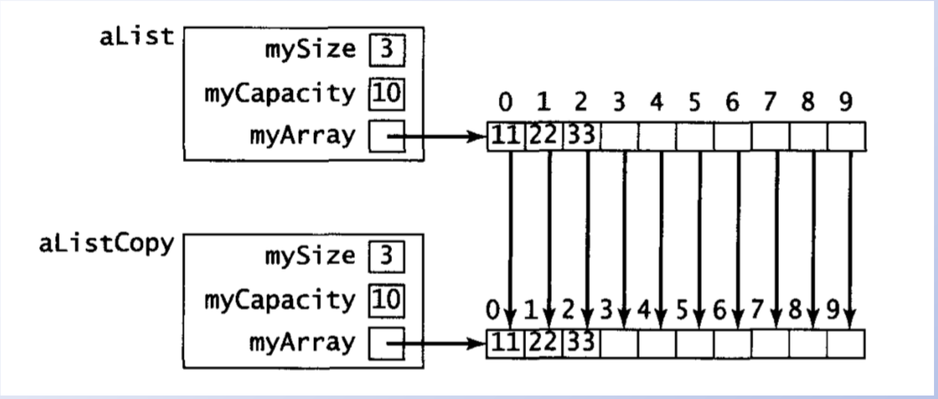
This function is called automatically to reclaim any memory allocated dynamically in an object of type *ClassName* whenever such an object should no longer exist. It will be called first, before deallocation of memory for other items in that object. Note that like a constructor, a destructor has no return type. However, unlike a constructor, a destructor cannot have parameters; thus a class can have only one destructor.

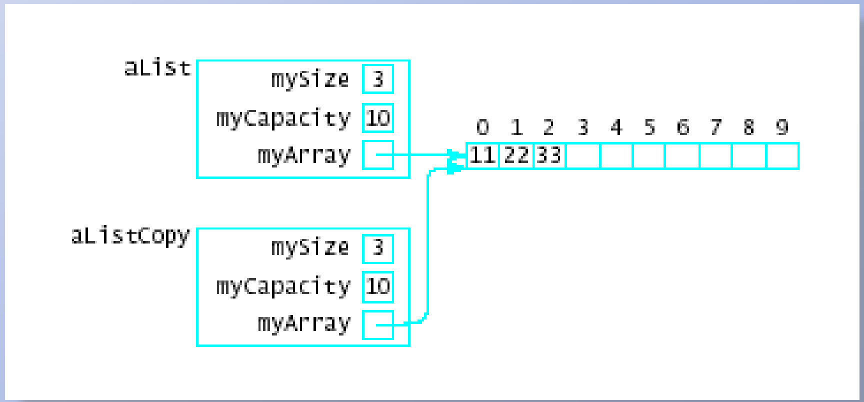
Common situations in which an object's destructor is called include the following:

- At the end of each block in which that object is declared (provided it is not static)[3]
- When execution of a program terminates for a static object
- At the end of a function definition in which that object is a value parameter
- If that object is created by a copy constructor and is no longer needed
- If that object was created using `new` and is destroyed using `delete`
- When some object containing that object as a data member is destroyed

2-Dynamic Allocation of List
New Functions Needed

- Copy Constructor – makes a "deep copy" of an object
  - When argument passed as value parameter
  - When function returns a local object
  - When temporary storage of object needed
  - When object initialized by another in a declaration

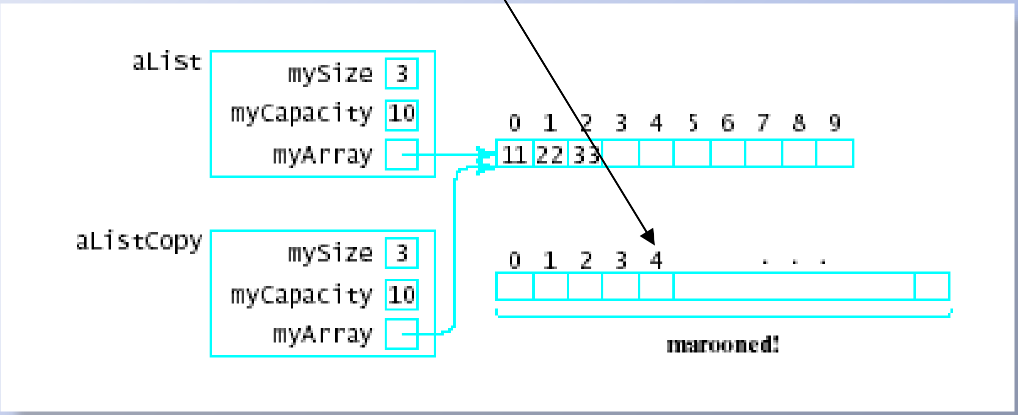




If copy is not made, observe results (aliasing problem, "shallow" copy)

2-Dynamic Allocation of List
New Functions Needed

- **Assignment operator**
  - Default assignment operator makes shallow copy
  - Can cause memory leak, dynamically-allocated memory has nothing pointing to it



2-Dynamic Allocation of List
Notes on Class Design

- If a class allocates memory at run time using the **`new`**, then it should provide …
  - A destructor
  - A copy constructor
  - An assignment operator

- Note code which exercises constructors and destructor

2-Dynamic Allocation of List

Future Improvements to Our `List` Class

- Problem 1: Array used has fixed capacity

  Solution:

  - If larger array needed during program execution
  - Allocate, copy smaller array to the new one

- Problem 2: Class bound to one type at a time

  Solution:

  - Create multiple `List` classes with differing names
  - Use class template

Recall Inefficiency of Array-Implemented List

- `insert()` and `erase()` functions inefficient for dynamic lists
  - Those that change frequently
  - Those with many insertions and deletions

So ...

We look for an alternative implementation.

Linked List

For the array-based implementation:

1. First element is at location 0
2. Successor of item at location *i* is at location `i + 1`
3. End is at location `size - 1`

Fix:

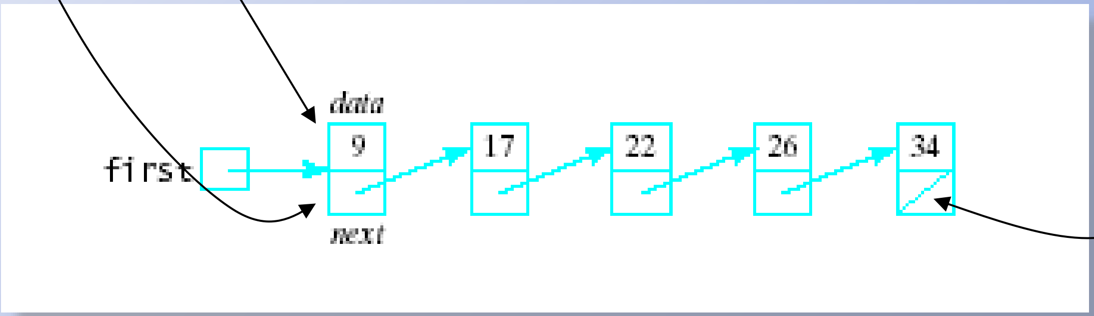
1. Remove requirement that list elements be stored in consecutive location.
2. But then need a "link" that connects each element to its successor

Linked Lists !!

Linked List

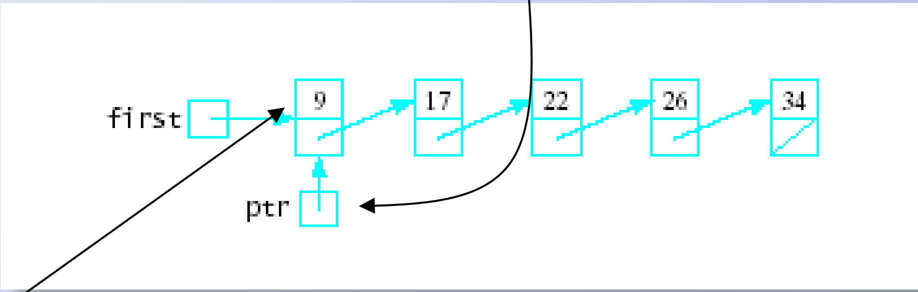
A **linked list** is a sequence of elements called **nodes**, each of which has two parts:

- *Data part* – stores an element of the list
- *Next part* – stores link/pointer to next element (when no next element, null value)



Linked Lists Operations

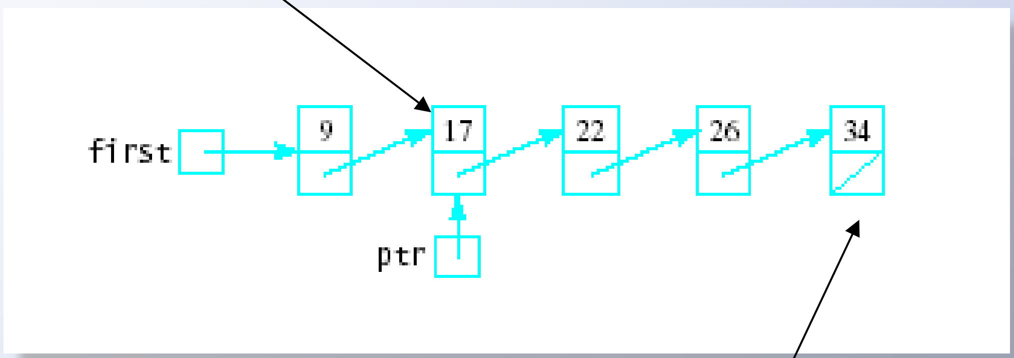
- Construction: `first = null_value;`
- Empty: `first == null_value?`
- Traverse
  - Initialize a variable **`ptr`** to point to first node



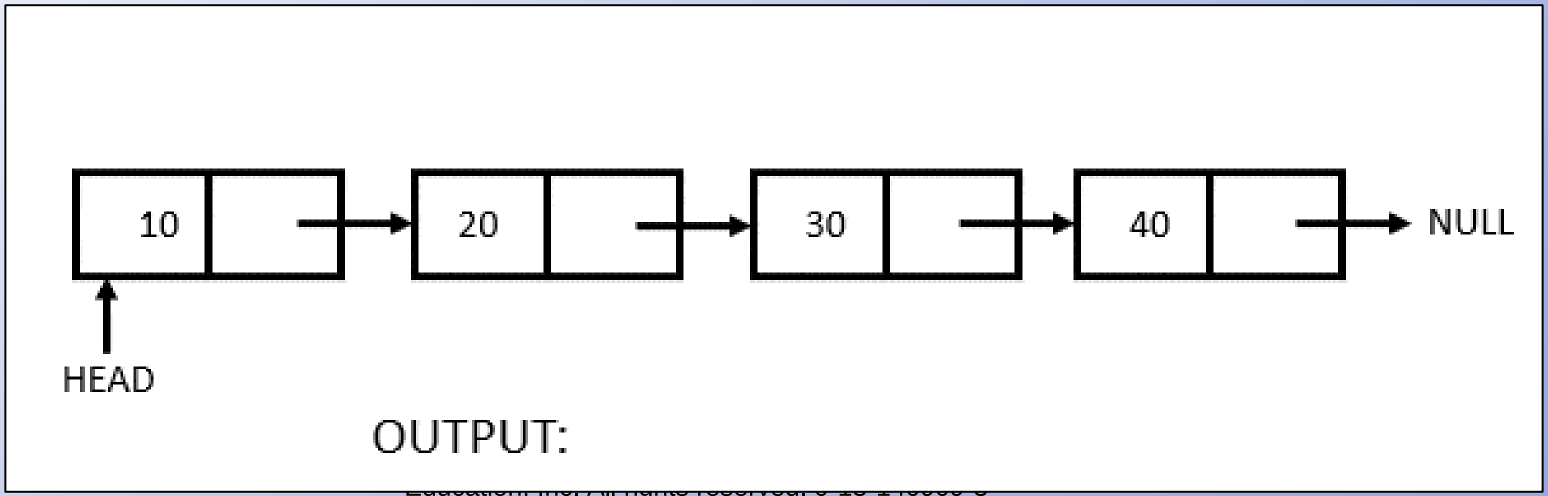
  - Process data where **`ptr`** points

Linked Lists Operations

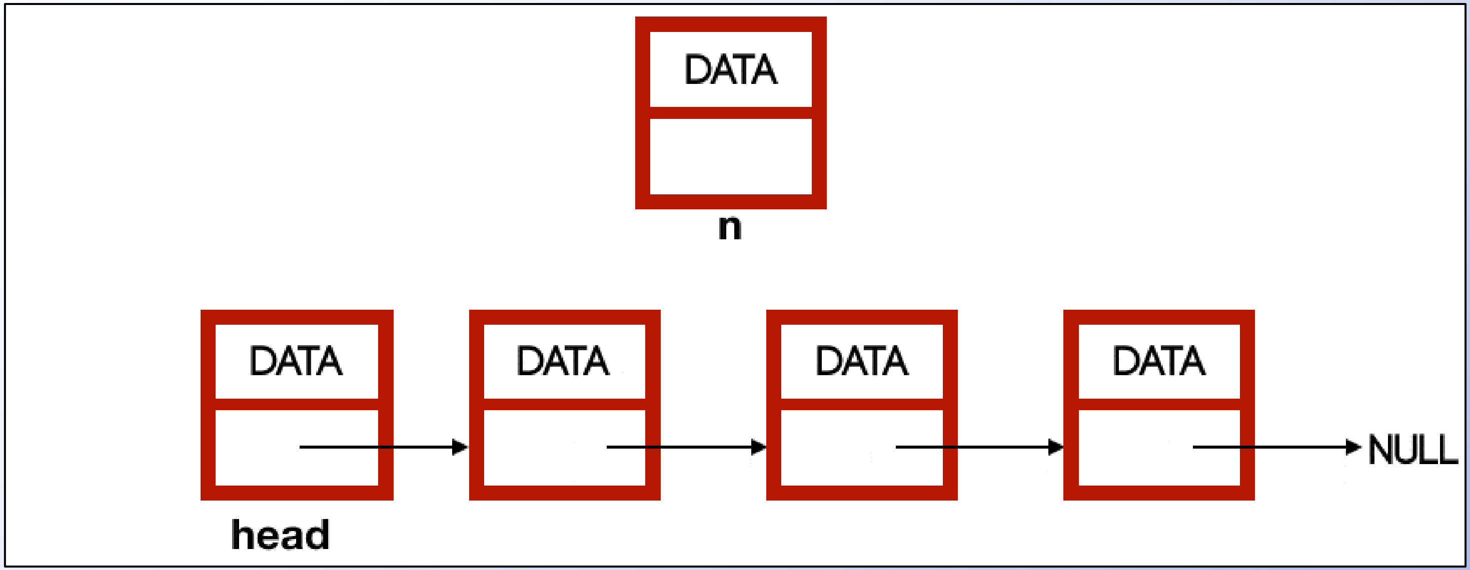
- **Traverse** (ctd)
  - set `ptr = ptr->next`, process `ptr->data`

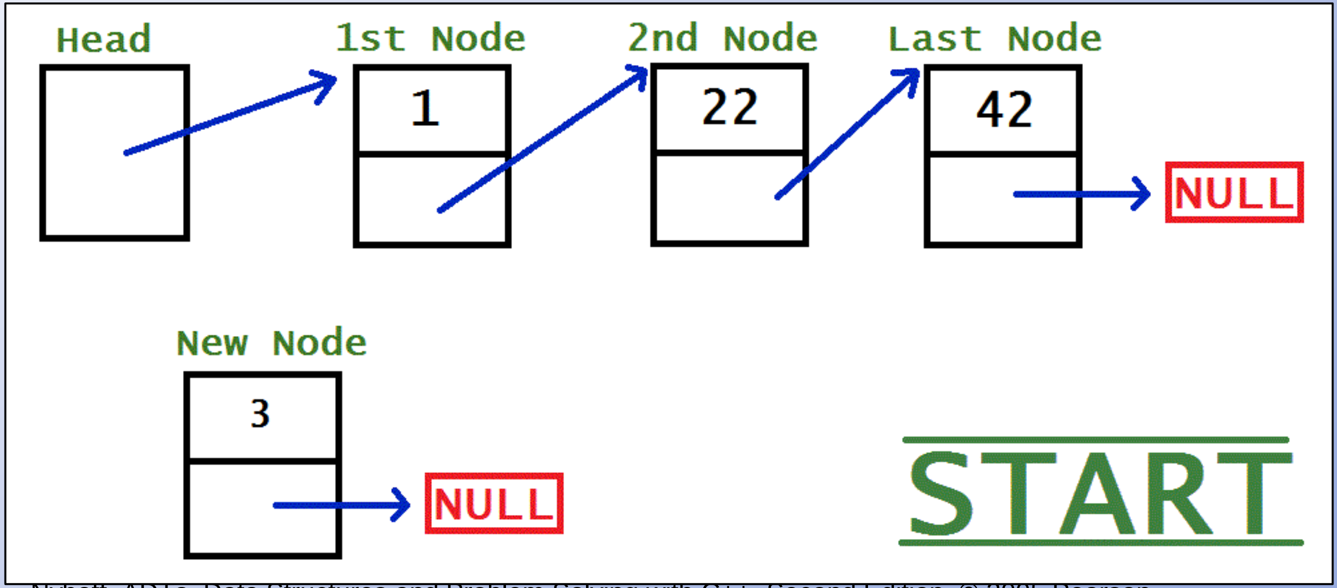


  - Continue until `ptr == null`

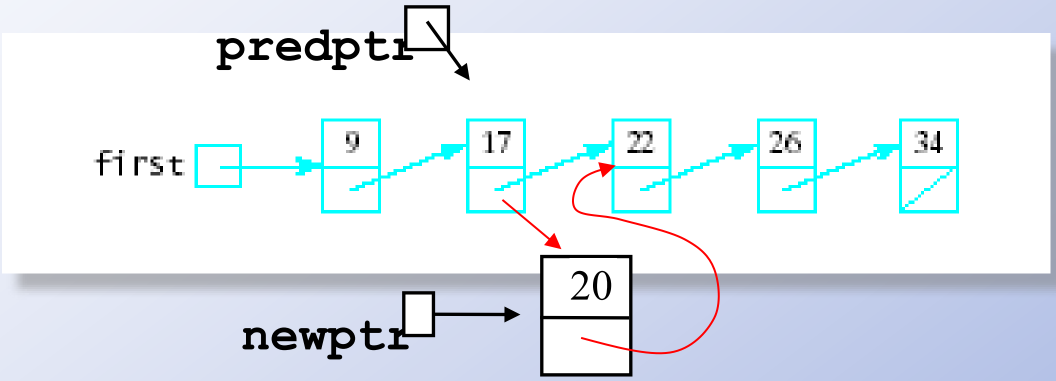


Linked Lists- insert (Graphically)





Operations: Insertion



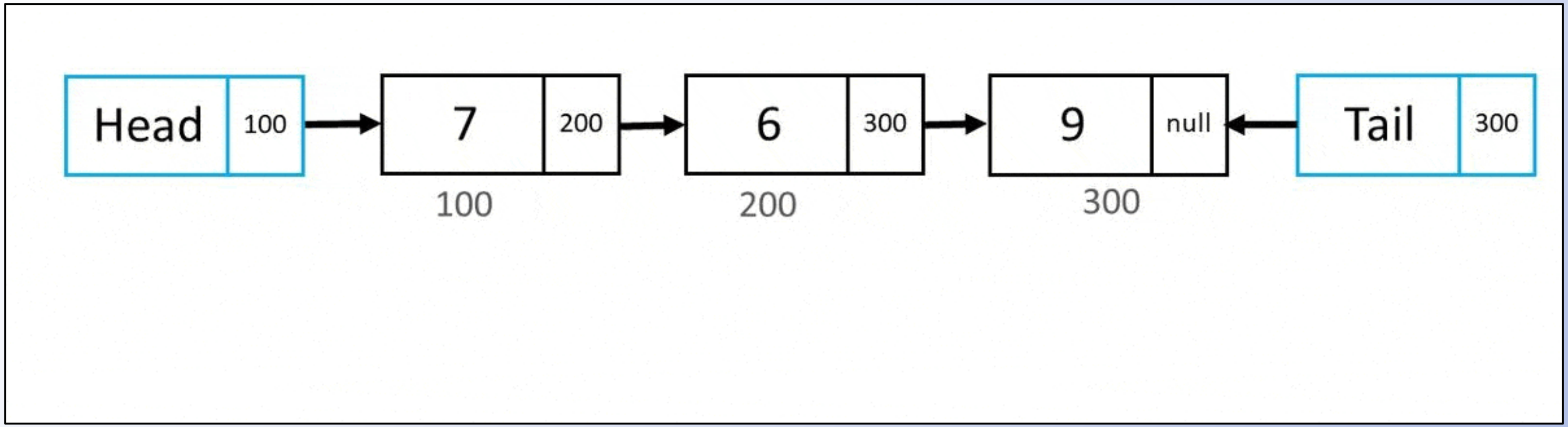
- To insert 20 after 17
- Need address of item before point of insertion
- **predptr** points to the node containing 17
- Get new node pointed to by **newptr** and store 20 in it
- Set the next pointer of this new node to the next pointer in its predecessor, thus making it point to its successor.
- Reset the next pointer of its predecessor to point to this new node

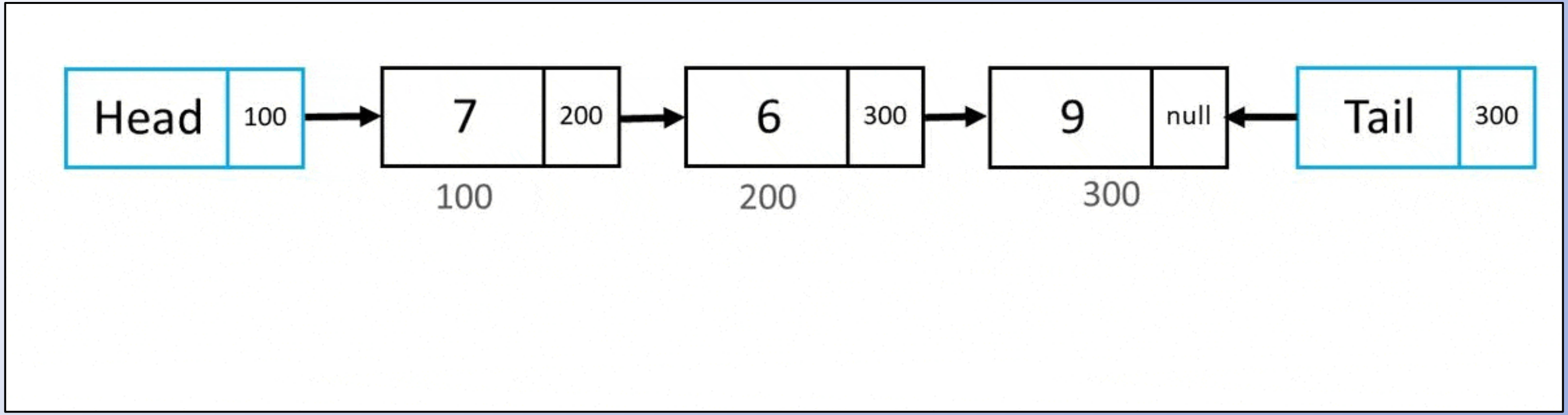
Operations: Insertion

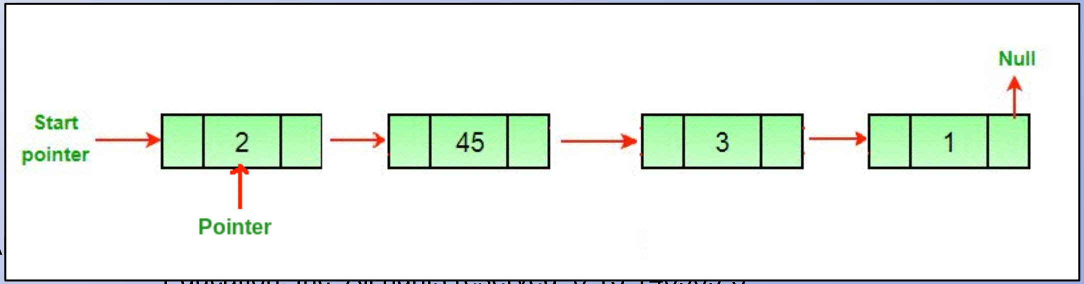
- Note: insertion also works at <u>end</u> of list
  - pointer member of new node set to **`null`**
- Insertion at the <u>beginning</u> of the list
  - **`predptr`** must be set to **`first`**
  - pointer member of **`newptr`** set to that value
  - **`first`** set to value of **`newptr`**

☒ Note: In all cases, **no shifting of list elements is required !**

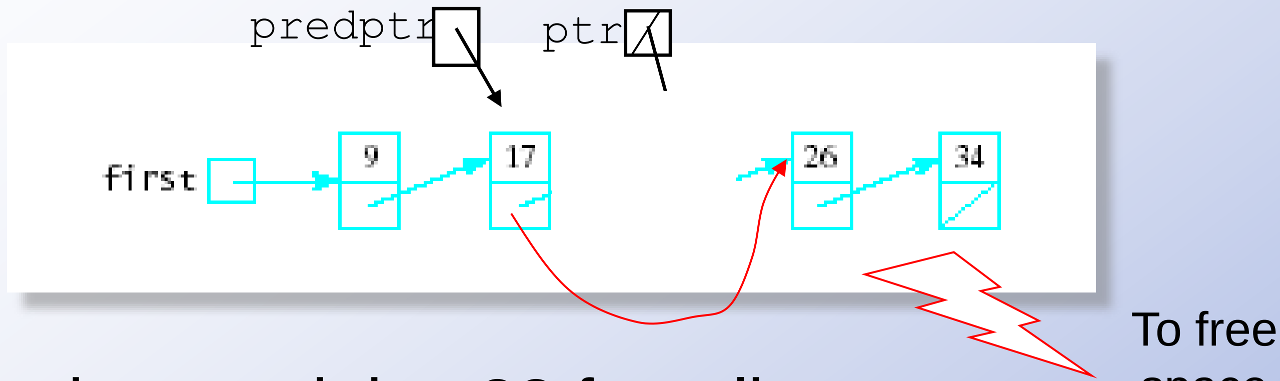
Linked Lists- erase (Graphically)







Operations: Erase



- Erase node containing 22 from list.
  - Suppose **ptr** points to the node to be deleted
  - **predptr** points to its predecessor (the 17)
- Do a bypass operation:
  - Set the next pointer in the predecessor to point to the successor of the node to be deleted
  - Deallocate the node being deleted.

Linked Lists - Advantages

- Access any item as long as external link to first item maintained
- Insert new item without shifting
- Erase existing item without shifting
- Can expand/contract as necessary

Linked Lists - Disadvantages

- Overhead of links:
  - used only internally, pure overhead
- If dynamic, must provide
  - destructor
  - copy constructor
- No longer have direct access to each element of the list
  - Many sorting algorithms need direct access
  - Binary search needs direct access
- Access of n^{th} item now less efficient
  - must go through first element, and then second, and then third, etc.

Linked Lists - Disadvantages

- List-processing algorithms that require fast access to each element cannot be done as efficiently with linked lists.
- Consider adding an element at the end of the list

Array	Linked List
`a[size++] = value;`	Get a new node; set data part = value next part = *null_value* If list is empty Set first to point to new node. Else Traverse list to find last node Set next part of last node to point to new node.

This is the inefficient part

Using C++ Pointers and Classes

- To Implement Nodes

```
class Node
  {
   public:
   DataType data;
   Node * next;
  };
```

- Note: The definition of a Node is <u>recursive</u>
  - (or self-referential)
- It uses the name Node in its definition
- The `next` member is defined as a pointer to a Node
- Data members of class *Node* is public because the declaration of class *Node* will be inside the *List* class

Working with Nodes

- Declaring pointers

```
Node * ptr;
```
or
```
typedef Node * NodePointer;
    NodePointer ptr;
```

- Allocate and deallocate

```
ptr = new Node;          delete ptr;
```

- Access the data and next part of node

`(*ptr).data` and `(*ptr).next`

or

`ptr->data` and `ptr->next`

Working with Nodes

- Note data members are public

```
class Node
{    public:
     DataType data;
     Node * next;          };
```

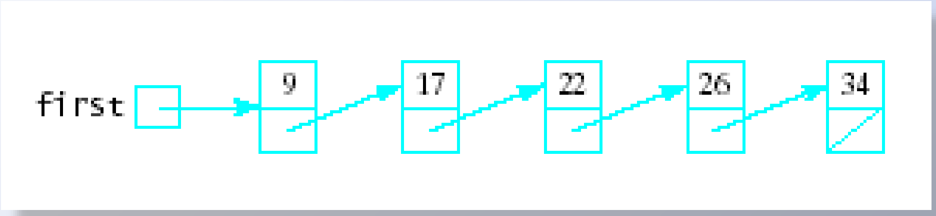
- This class declaration will be placed inside another class declaration for **`List`**
- The data members **`data`** and **`next`** of class **`Node`** will be public inside the class
  - Will be accessible to the member and friend class or functions
  - Will be private outside the class

Class `List`

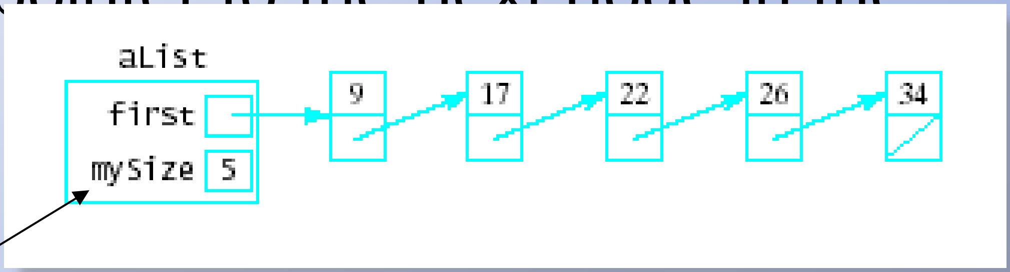
```
typedef int ElementType;
class List
{
 private:
  class Node
  {
  public:
   ElementType data;
   Node * next;
  };
  typedef Node * NodePointer;
   . . .
```

- `data` is public inside class `Node`
- class `Node` is private inside `List`

Data Members for Linked-List Implementation



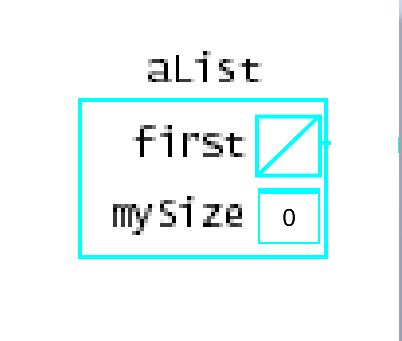
- A linked list will be characterized by:
  - A pointer to the first node in the list.
  - Each node contains a pointer to the next node in the list
  - The last node contains



- As a variation `first` may
  - be a structure
  - also contain a count of the elements in the list

Function Members for Linked-List Implementation

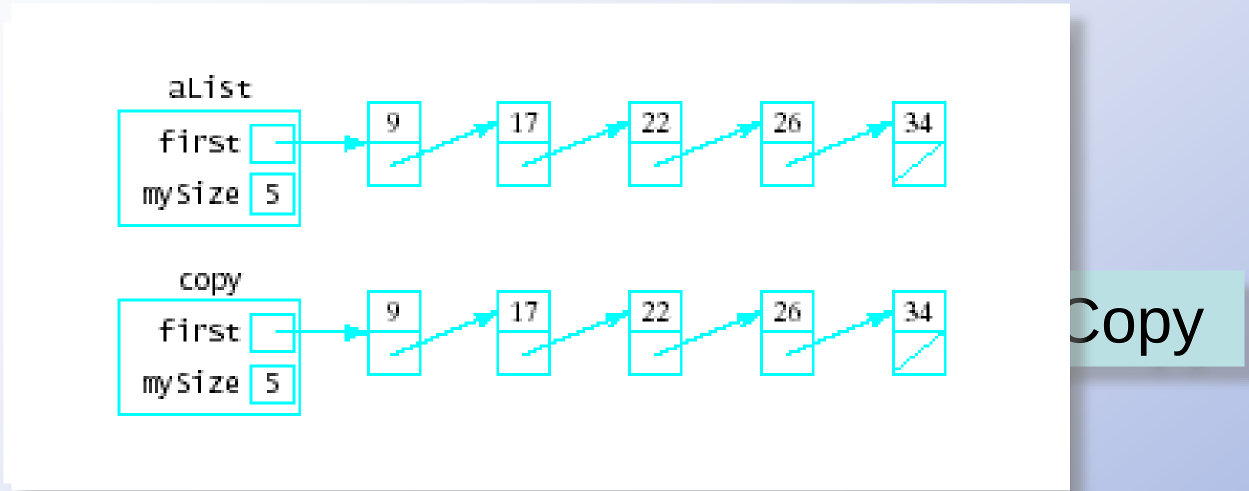
- Constructor
  - Make **`first`** a null pointer and
  - set **`mySize`** to 0



- Destructor
  - Nodes are dynamically allocated by **`new`**
  - Default destructor will not specify the **`delete`**
  - All the nodes from that point on would be "marooned memory"
  - A destructor must be explicitly implemented to do the **`delete`**

Function Members for Linked-List

Copy constructor

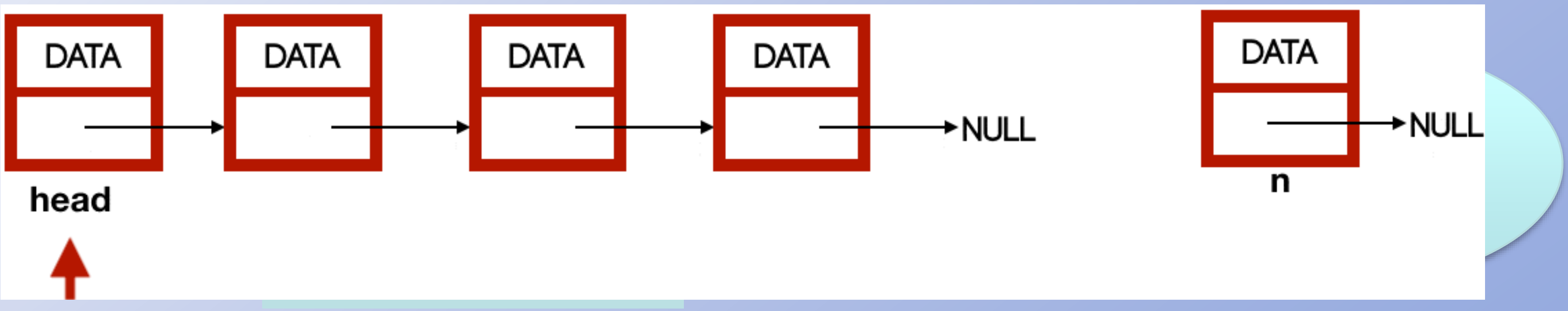


- Copy constructor for deep copy
  - By default, when a copy is made of a **`List`** object, it only gets the head pointer
  - Copy constructor will make a new linked list of nodes to which **`copy`** will point

Types of Linked-List

Singly Linked-List

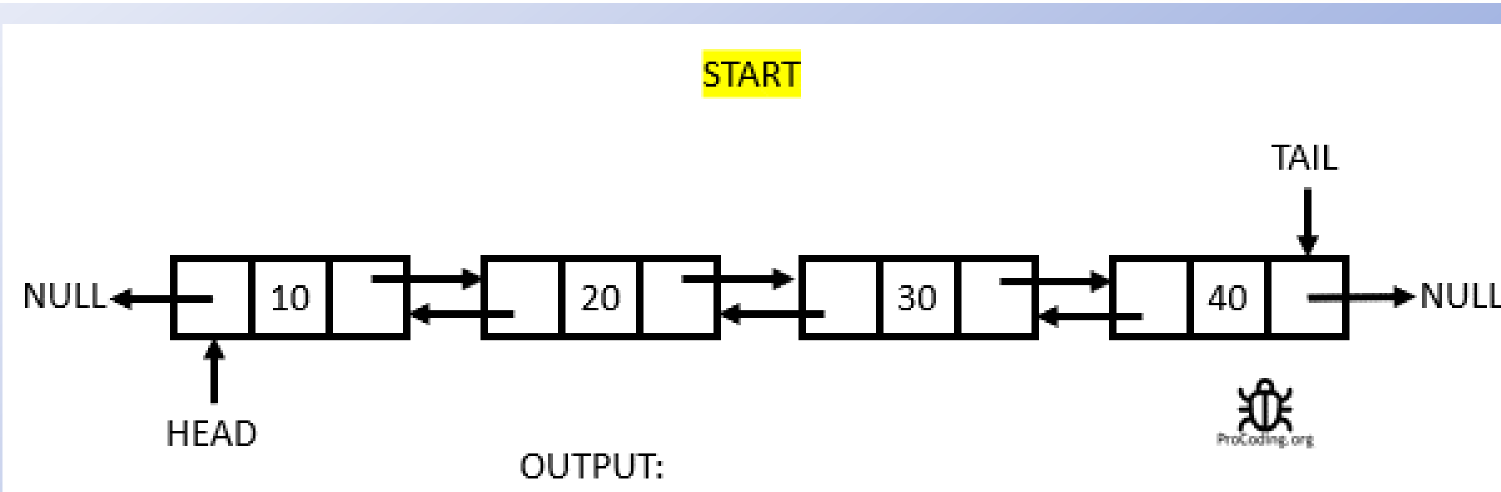
It is the most common. Each node has data and a pointer to the next node.



Types of Linked-List

Doubly Linked-List

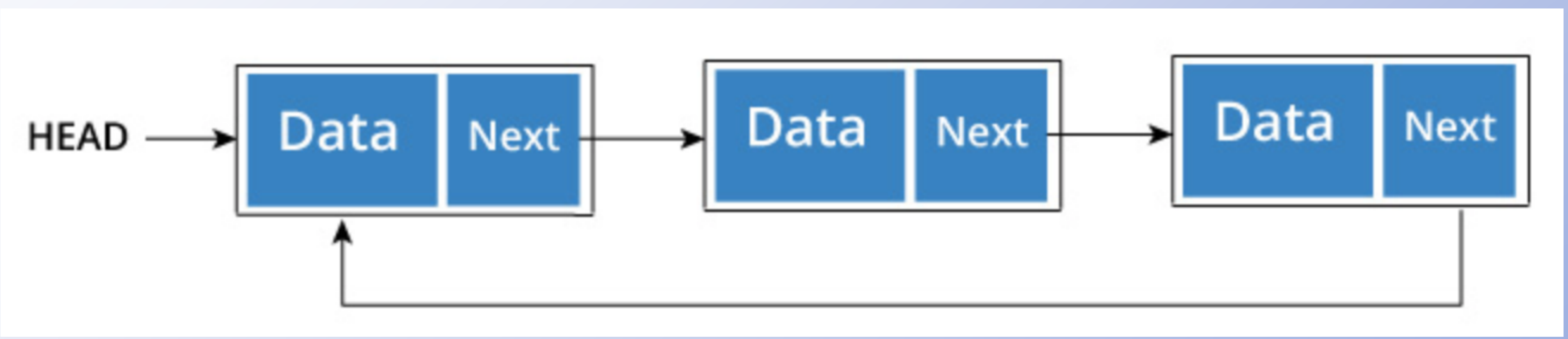
We add a pointer to the previous node in a doubly linked list. Thus, we can go in either direction: forward or backward.



Types of Linked-List

Circular Linked-List

A circular linked list is a variation of linked list in which the last element is linked to the first element. This forms a circular loop.



A circu ... ked.

- for s ... tem.
- In d ... n.