

Lists

Chapter 6

Chapter Contents

6.1 List as an ADT

6.2 An Array-Based Implementation of Lists

6.3 An array Based Implementation of Lists with
Dynamic Allocation

6.4 Introduction to Linked Lists

6.5 A Pointer-Based Implementation of Linked Lists in
C++

6.6 An Array-Based Implementation of Linked Lists
optional)

Chapter 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
- (Optional) Study array-based implementation of linked lists

Consider Every Day 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

- Can have a single element
- Can have no elements
- There can be lists of lists
- We will look at the list as an abstract data type
 - Homogeneous
 - Finite length
 - Sequential elements

Basic Operations

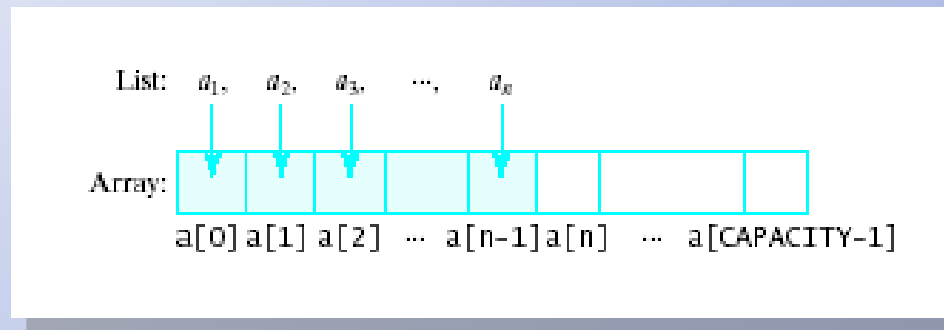
- Construct an empty list
- Determine whether or not empty
- Insert an element into the list
- Delete an element from the list
- Traverse (iterate through) the list to
 - Modify
 - Output
 - Search for a specific value
 - Copy or save
 - Rearrange

Designing a `List` Class

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

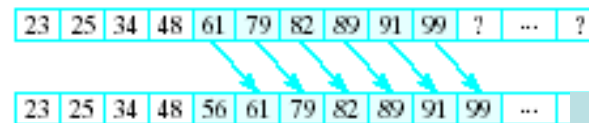
Array-Based Implementation of Lists

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

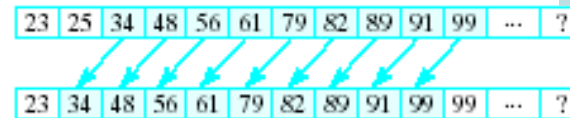


Implementing Operations

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



Also adjust
size up or
down



List Class with Static Array

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

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

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

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 Example

- Declaration file, Fig. [6.1A](#)
 - Note use of `typedef` mechanism outside the class
 - This example good for a list of `int`
- Definition, implementation Fig. [6.1B](#)
 - Note considerable steps required for `insert()` and `erase()` functions
- Program to test the class, Fig [6.1C](#)

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

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()`

Dynamic-Allocation for `List` Class

- Note data changes in [Fig. 6.2A](#), `List.h`
- Note implementation file [Fig. 6.2B](#), `List.cpp`
 - Changes to constructor
 - Addition of other functions to deal with dynamically allocated memory
- Note testing of various features in [Fig. 6.2C](#), the demo program

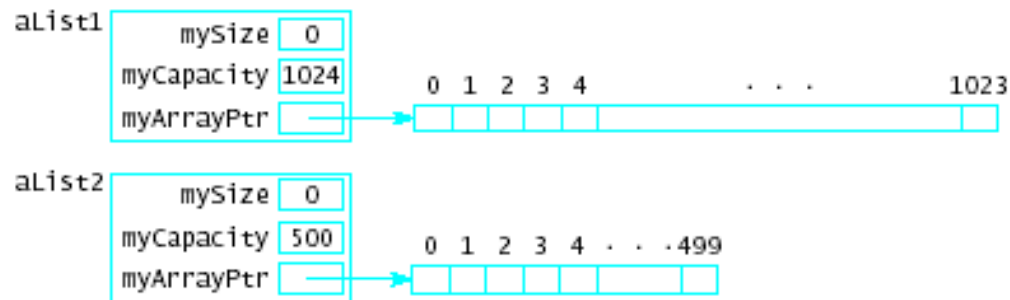
Dynamic-Allocation for `List` Class

- Now possible to specify different sized lists

```
cin >> maxListSize;
```

```
List aList1 (maxListSize);
```

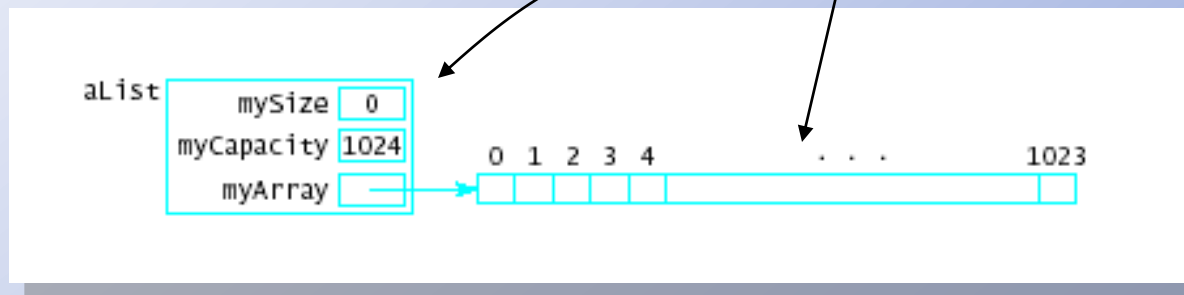
```
List aList2 (500);
```



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

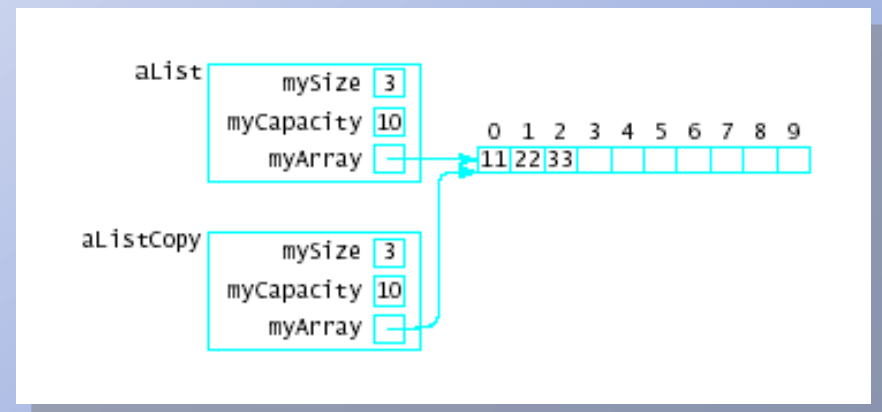


- The destructor reclaims dynamically allocated memory

New Functions Needed

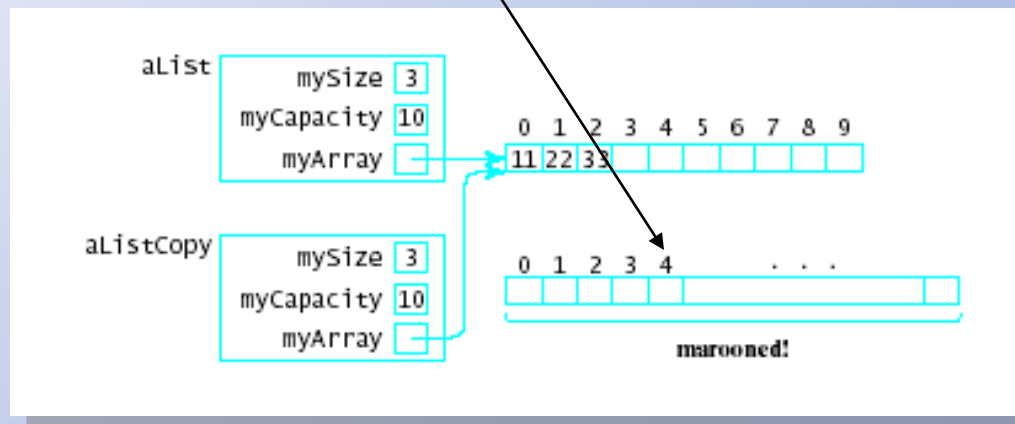
- Copy Constructor – makes a "deep copy" of an object
 - When argument passed as value parameter
 - When function returns a local object
 - When temporary storage of object needed
 - When object initialized by another in a declaration

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



New Functions Needed

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



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 [Fig. 6.3](#) which exercises constructors and destructor

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 $size - 1$

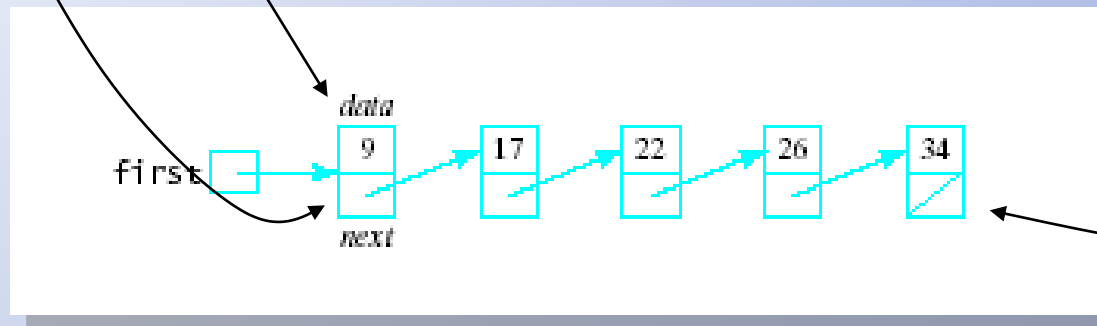
Fix:

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 !!

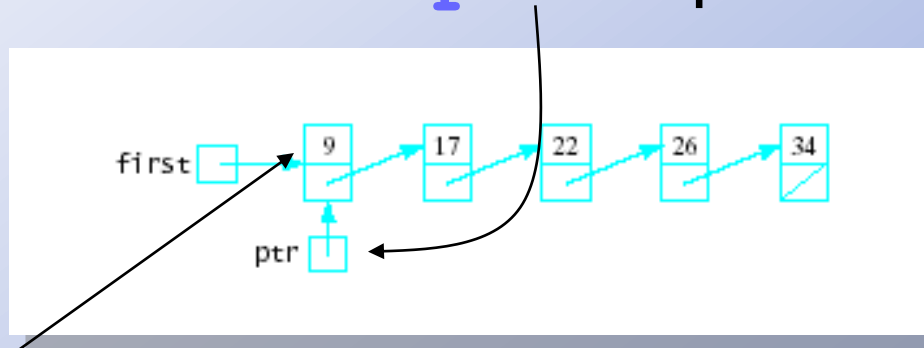
Linked List

- Linked list nodes contain
 - 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

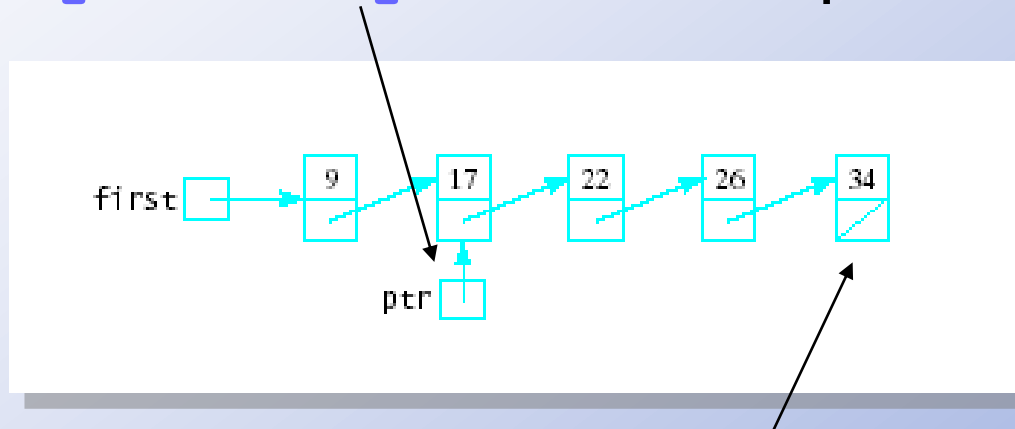


- Process data where `ptr` points

Linked Lists Operations

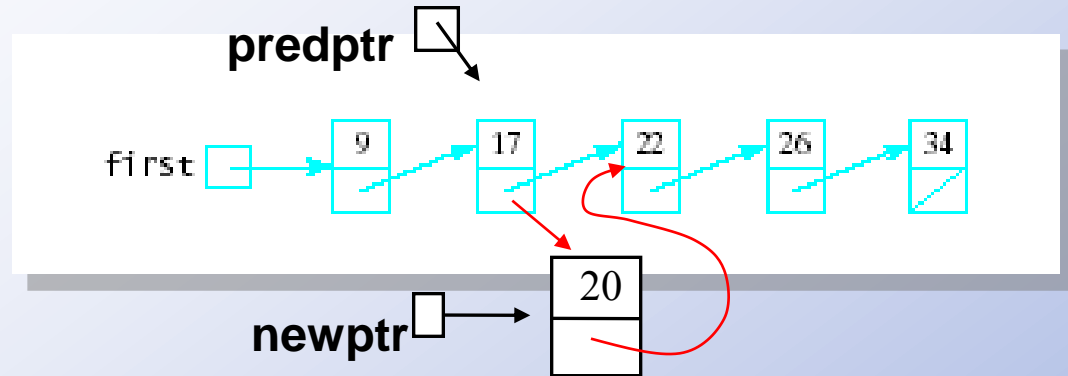
- Traverse (ctd)

- set `ptr = ptr->next`, process `ptr->data`



- Continue until `ptr == null`

Operations: Insertion



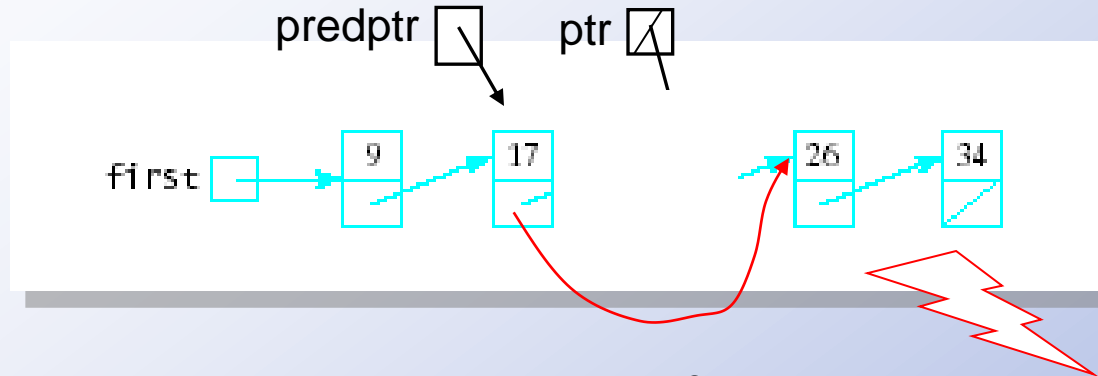
- Insertion
 - To insert 20 after 17
 - Need address of item before point of insertion
 - **predptr** points to the node containing 17
 - Get a new node pointed to by **newptr** and store 20 in it
 - Set the next pointer of this new node equal to the next pointer in its predecessor, thus making it point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

Operations: Insertion

- Note: insertion also works at end of list
 - pointer member of new node set to `null`
- Insertion at the beginning 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 !**

Operations: Deletion



To free
space

- Delete node containing 22 from list.
 - Suppose **ptr** points to the node to be deleted
 - **predptr** points to its predecessor (the 20)
- 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
- Delete 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

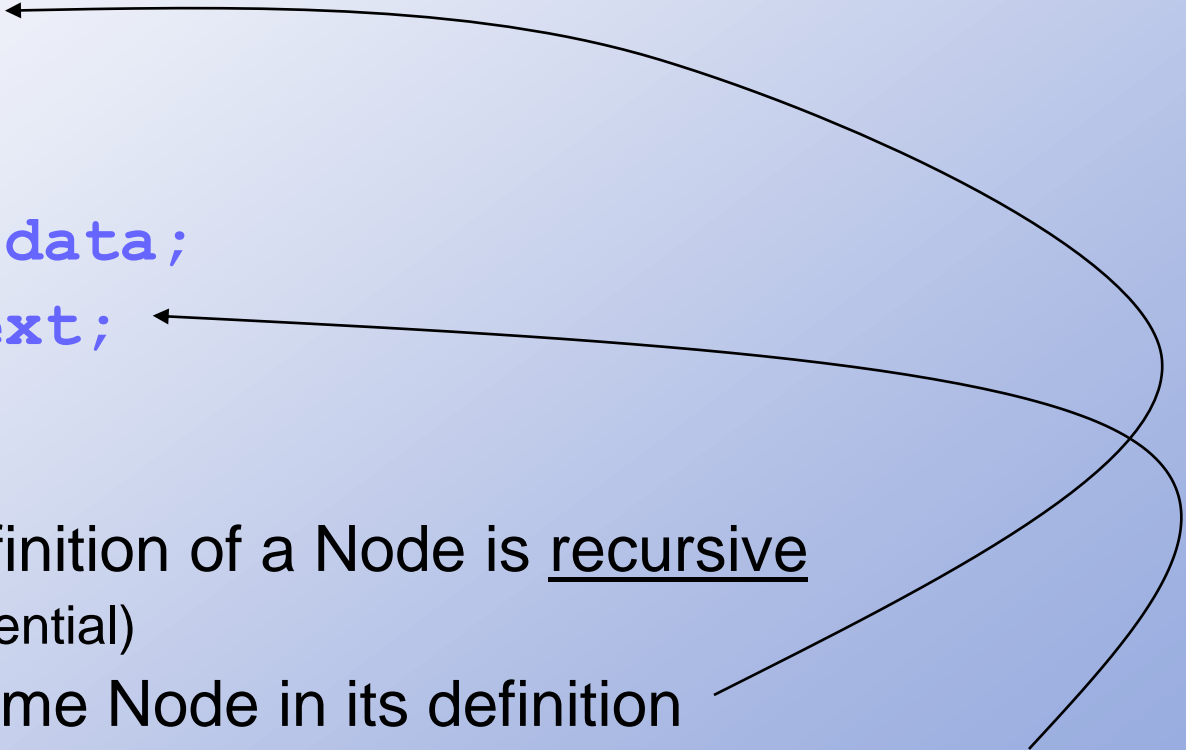
Array	Linked List
<pre>a[size++] = value;</pre>	<p>Get a new node; set data part = value next part = <i>null_value</i></p> <p>If list is empty Set first to point to new node.</p> <p>Else</p> <p>Traverse list to find last node</p> <p>Set next part of last node to point to new node.</p>

This is the inefficient part

Using C++ Pointers and Classes

- To Implement Nodes

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

A diagram consisting of two curved arrows. The first arrow starts from the 'Node * next;' line in the code and points back to the 'class Node' line. The second arrow starts from the 'Node * next;' line and points to the 'Note: The definition of a Node is recursive' bullet point.

- Note: The definition of a Node is recursive
 - (or self-referential)
- It uses the name Node in its definition
- The **next** member is defined as a pointer to a Node

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

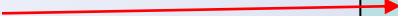
```
(*ptr).data    and    (*ptr).next
```

or

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

Working with Nodes

- Note data members are public



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

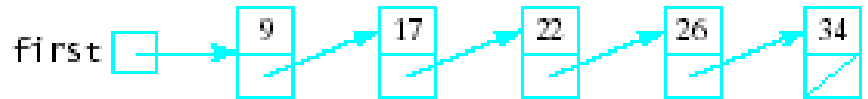
- This class declaration will be placed inside another class declaration for **List**
- The data members **data** and **next** of struct **Node** will be public inside the class
 - will accessible to the member and friend functions
 - will be private outside the class

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

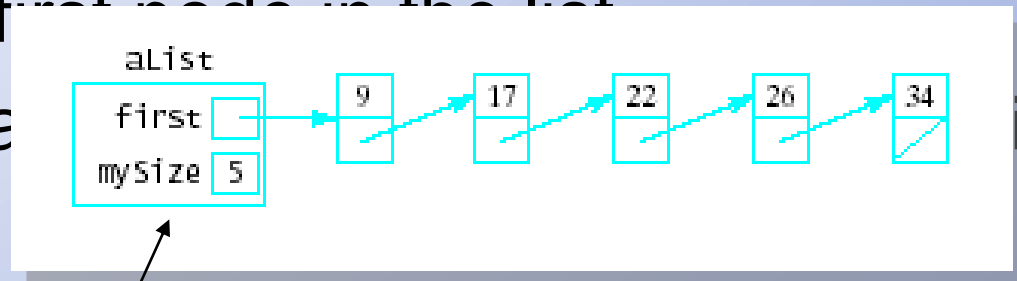
- Each node contains the data and the pointer to the next node in the list

- The last node contains a null pointer

- As a variation **first** may

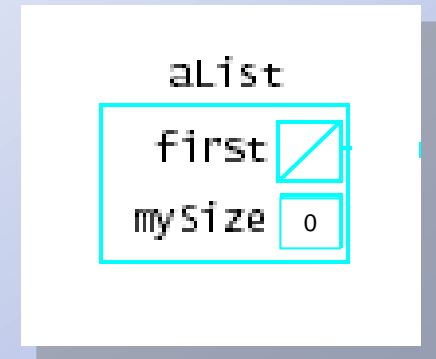
- be a structure

- also contain a count of the elements in the list

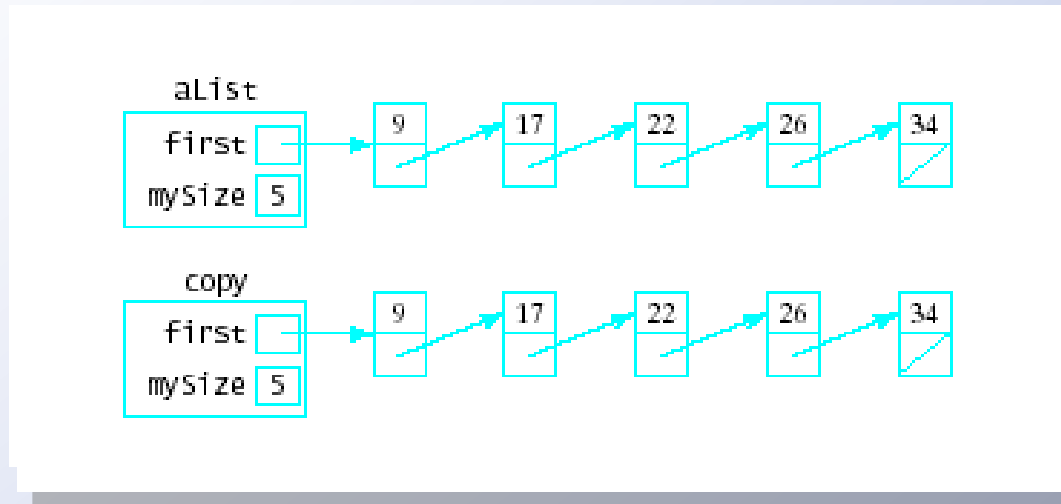


Function Members for Linked-List Implementation

- 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 Implementation



- 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

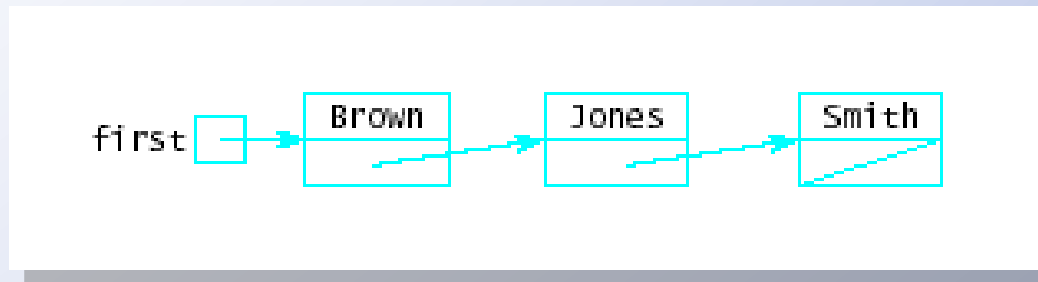
Array-Based Implementation of Linked Lists

- Node structure

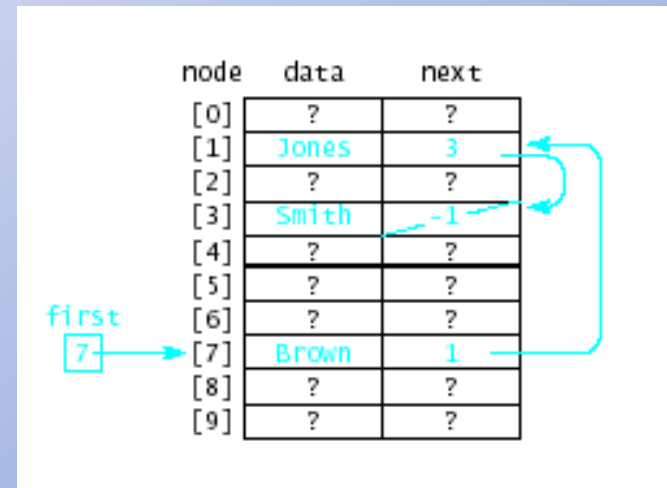
```
struct NodeType
{
    DataType data;
    int next;
};
const int NULL_VALUE = -1;
// Storage Pool
const int NUMNODES = 2048;
NodeType node [NUMNODES];
int free;
```


Array-Based Implementation of Linked Lists

- Given a list with names



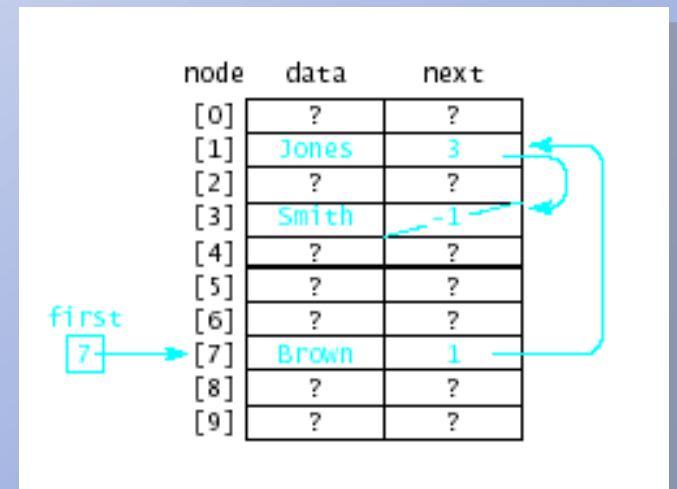
- Implementation would look like this



Array-Based Implementation of Linked Lists

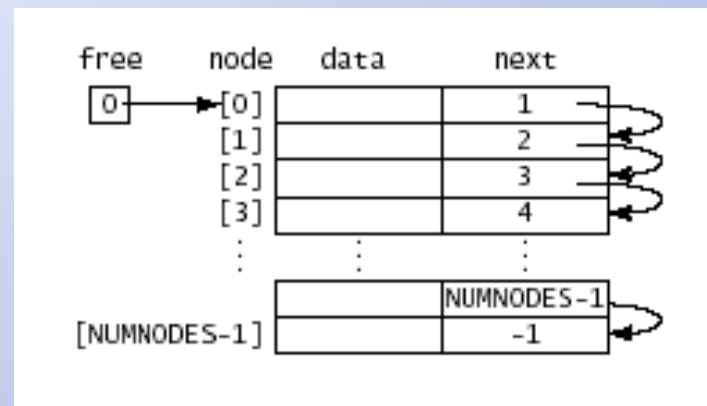
- To traverse

```
ptr = first;  
while (ptr != NULL_VALUE)  
{  
    // process data at node[ptr].data  
    ptr = node[ptr].next;  
}
```



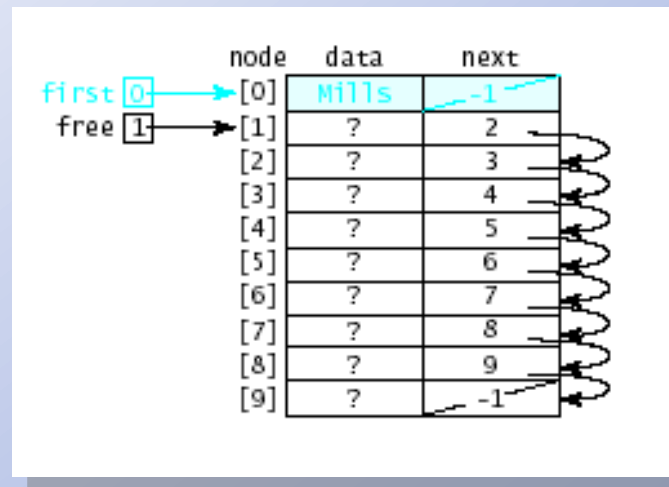
Organizing Storage Pool

- In the array
 - Some locations are storing nodes of the list
 - Others are free nodes, available for new data
- Could initially link all nodes as free



Organizing Storage Pool

- Then use nodes as required for adds and inserts
 - Variable free points to beginning of linked nodes of storage pool



Organizing Storage Pool

- Links to actual list and storage pool maintained as new data nodes are added and deleted

