ADT: Design & Implementation

Dr. Yingwu Zhu

Outline

- Concept: ADT
- Demonstration of ADT's design and implementation
  - List as an ADT (our focus)
  - Stack & Queue as ADTs (exercises)

Abstract Data Type (ADT)

- ADT = data items + operations on the data
- Design of ADT
  - Determine data members & operations
- Implementation of ADT
  - Storage/data structures to store the data
  - Algorithms for the operations, i.e., how to do it

Example: List

- List as an ADT
- 3 different implementations
  - Static array-based
  - Dynamic array-based
  - Linked-list based

Properties of Lists

- Can have a single element
- Can have no elements → empty!
- There can be lists of lists

- We will look at the list as an abstract data type
  - Homogeneous
  - Finite length
  - Sequential elements

Basic Operations

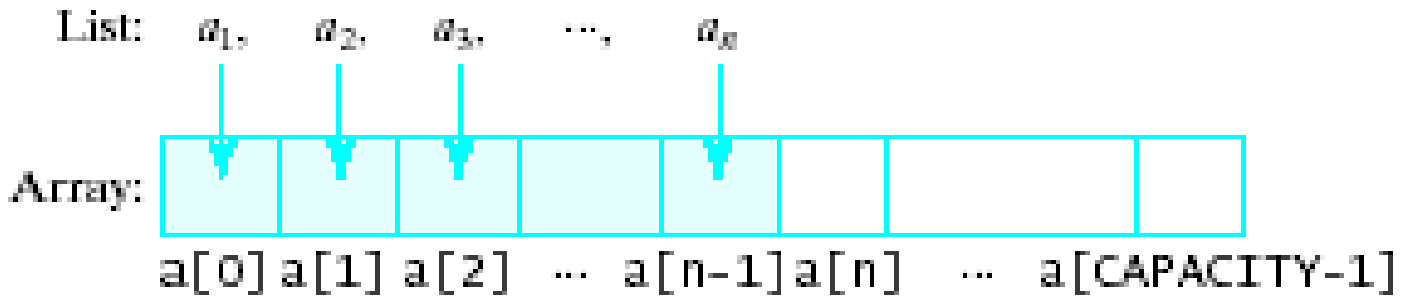
- Construct an empty list
- Determine whether or not empty
- Insert an element into the list
- Delete an element from the list
- Traverse (iterate through) the list to
  - Modify
  - Output
  - Search for a specific value
  - Copy or save
  - Rearrange

Designing a `List` Class

- Should contain at least the following function members
  - Constructor
  - `empty()`
  - `insert()`
  - `delete()`
  - `display()`
- Implementation involves
  - Choosing data structures to store data members
  - Choosing algorithms to implement function members

Impl. 1: Static Array

- Use a static array to hold list elements
- Easy to manipulate arrays for implementing operations
- Natural fit for mapping list elements to array elements



Design

- What data members?
- What operations?

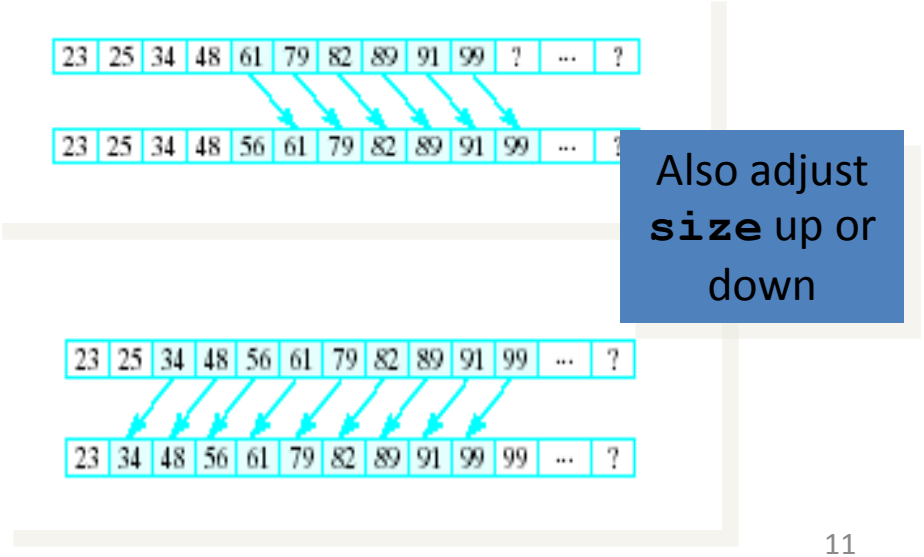
Design

```
const  int CAPACITY = 1000; //for what purpose?
typedef  int  ElementType;   //for what purpose?

class List {
   private:
      int size;   //# of elements
      ElementType array[CAPACITY];
   public:
       …
};
```

Implementing Operations

- Constructor
  - Static array allocated at compile time. No need to allocate explicitly!
- Empty
  - Check if size == 0
- Traverse
  - Use a loop from 0th element to `size - 1`
- Insert
  - Shift elements to right of insertion point
- Delete
  - Shift elements back



`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

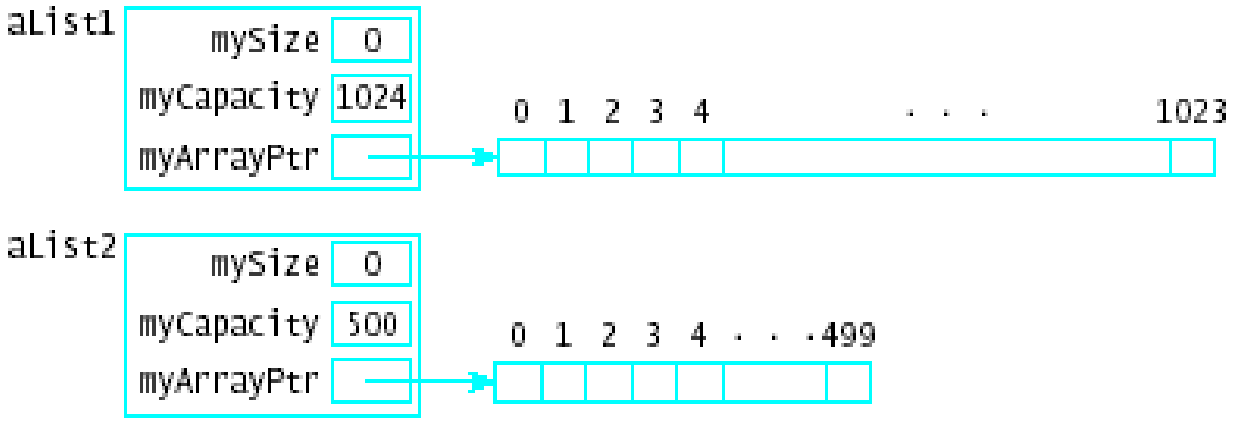
Impl. 2: Using Dynamic Arrays

- Allow List objects to specify varied sizes

Dynamic-Allocation for `List` Class

- Now possible to specify different sized lists

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



Dynamic-Allocation for `List` Class

- Changes required in data members
  - Eliminate const declaration for CAPACITY
  - Add 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()`

Design

```
typedef   int   ElementType;

class List {
   private:
      int mySize;   //# of elements
      int myCapacity;
      ElementType *myArray;
   public:
      List (int cap);
};
```

What needs to be changed?

- Constructor
- Addition of other functions to deal with dynamically allocated memory
  - Destructor
  - Copy constructor (won't be discussed here)
  - Assignment operator (won't be discussed here)

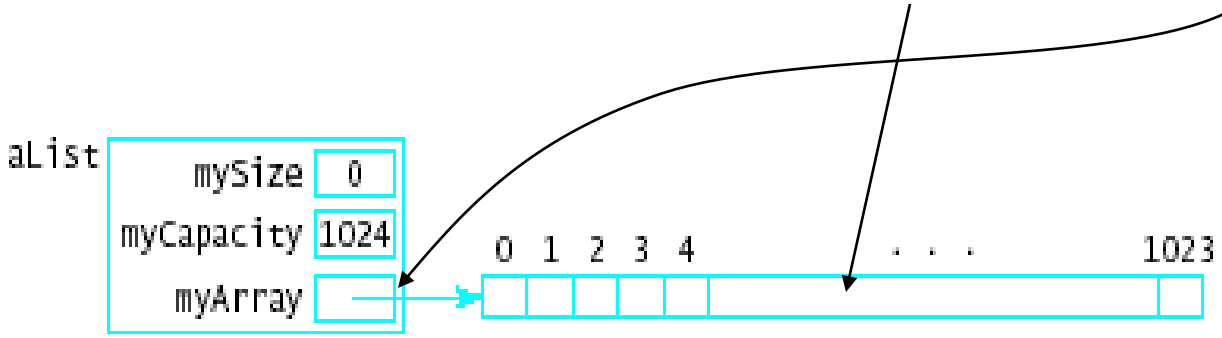
Constructor

- Two tasks
  - Data member initialization
  - Dynamically allocate memory

```
//constructor
List::List(int cap) : mySize(0), myCapacity(cap)  {
    myArray = new  int[cap]; //allocate memory
    assert(myArray != 0); //need #include <cassert>
}
```

Destructor Needed!

- When class object goes out of scope, the pointer to the dynamically allocated memory is reclaimed automatically
- The dynamically allocated memory is <u>not</u>



- The destructor reclaims dynamically allocated memory (the default destructor fails here!)

Destructor

- The default destructor needs to be overrided!
- De-allocate memory you allocated previously; otherwise causes memory leak problem!

```
List::~List() {
    delete [] myArray;
}
```

Problems

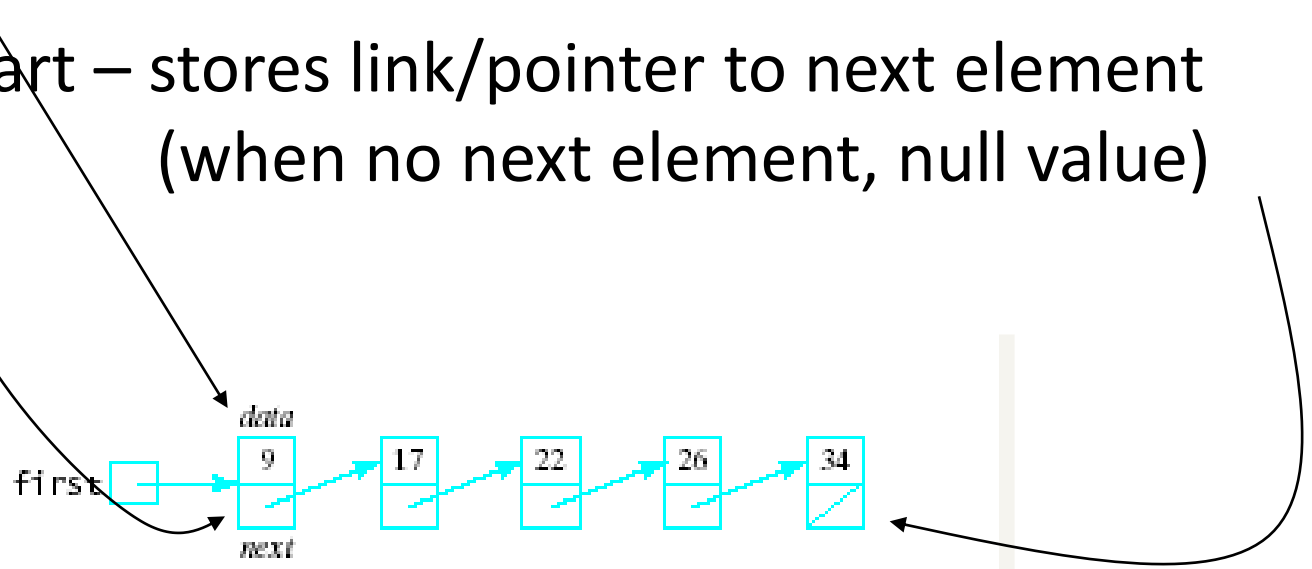
- Array capacity cannot be changed during running time (execution)!
- Insertion & deletion are not efficient!
  - Involve element shifting

Impl. 3: Using Linked-Lists

- To address the two problems faced by array-based solutions

Review: 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)

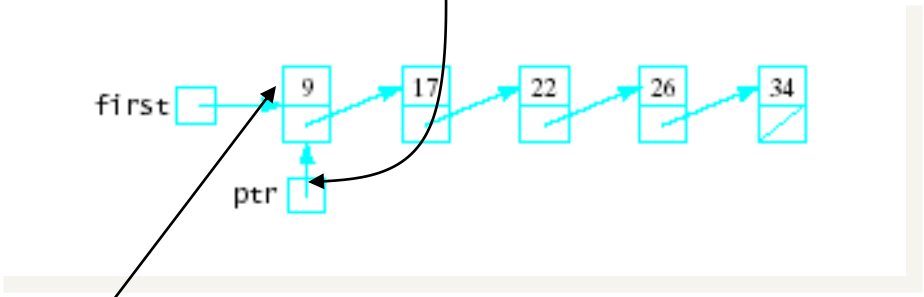


Basic Operations

- Traversal
- Insertion
- Deletion

Traversal

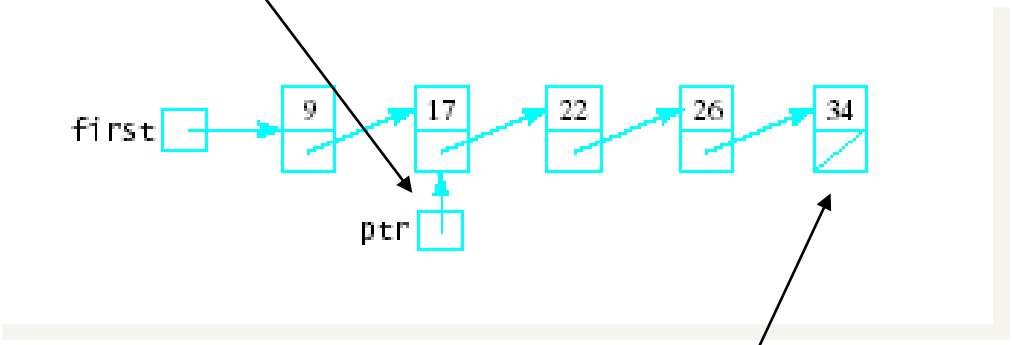
- Initialize a variable `ptr` to point to first node



- Process data where `ptr` points

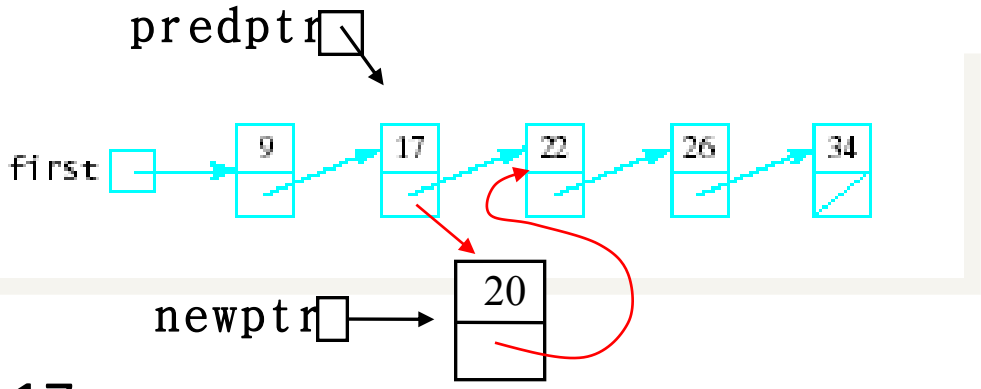
Traversal (cont.)

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



- Continue until `ptr == null`

Insertion



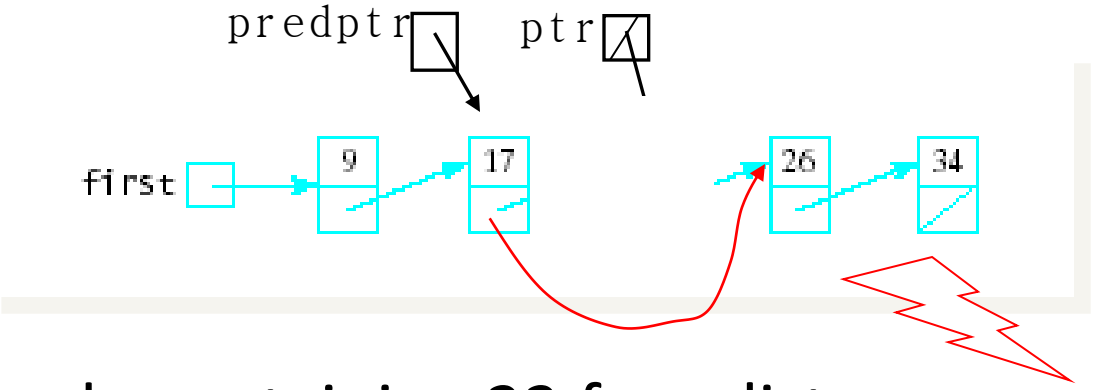
- Insertion
  - To insert 20 after 17
  - Need address of item before point of insertion
  - **`predptr`** points to the node containing 17
  - Create a new node pointed to by **`newptr`** and store 20 in it
  - Set the next pointer of this new node equal to the next pointer in its predecessor, thus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

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



- Delete 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.

1 Lesson in Pointers

- Before operating on an pointer, first check if it is null!

```
if (ptr)  {
    //do sth. meaningful
}
```

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 (but not here!)
- 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

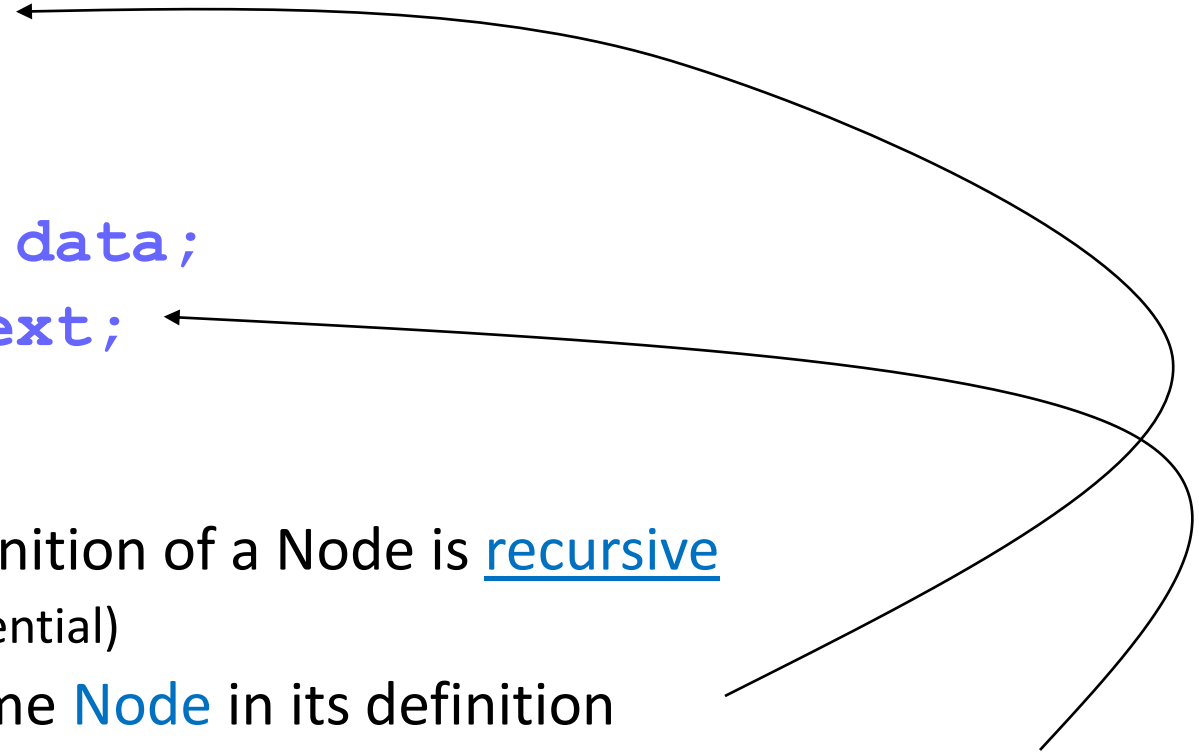
Array	Linked List
`a[size++] = value;`	Get a new node; set data part = value next part = *null_value* If list is empty Set first to point to new node. Else Traverse list to find last node Set next part of last node to point to new node.

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 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

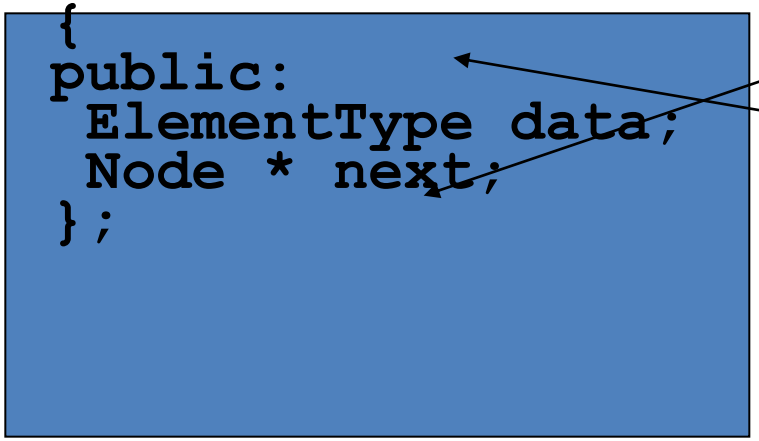
```
class Node
{   public:
    DataType data;
    Node * next;          };
```

- This class declaration will be placed inside another class declaration for **List (private section), p296**

Implementing List with Linked-Lists

```
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**

Design & Impl.

- Add data members
- Add member functions

Class List

```
#ifndef  _LIST_H
#define  _LIST_H
typedef  int ElementType;
class List {
   private:
        class Node {
             public:
                 ElementType data;
                 Node* next;
        };
        typedef Node*   NodePointer;
   private:
       NodePointer   first;
   public:
        List();
        ~List();
        void insert(ElementType x);
        void delete(ElementType x);
};
#endif
```

Implementing

- Constructor
- Destructor
- Operation: insert()
- Operation: delete()

Exercises

- Implement Stack and Queue classes with different implementations.

Stack

- Design and Implement a Stack class
- 3 options
  - Static array
  - Dynamic array
  - Linked list

Stack.h

```
typedef int DataType;
class Stack {
    public:
        Stack();
        Stack(const Stack& org);
        void push(const DataType& v);
        void pop();
        DataType top() const;
        ~Stack();
    private:
        class Node {
            public:
                DataType data;
                Node* next;
                Node(DataType v, Node* p) : data(v), next(0) { }
        };
        typedef Node* NodePtr;

        NodePtr myTop;
};
```

Queue

- Design and Implement a Queue class
- 3 options
  - Static array
  - Dynamic array
  - Linked list

Queue.h

```
typedef int DataType;
class Queue {
    public:
      //constructor
      //… member functions
    private:
       class Node {
          public:
             DataType data;
             Node* next;
             Node(DataType v, Node* p) : data(v), next(0) { }
       };
       typedef Node* NodePtr;
       NodePtr myFront, myback;
};
```