Abstract Data Types

Data types I

- We type data--classify it into various categories--such as int, boolean, String, Applet
  - A **data type** represents a set of possible values, such as {..., -2, -1, 0, 1, 2, ...}, or {true, false}
- By typing our variables, we allow the computer to find some of our errors
  - Some operations only make sense when applied to certain kinds of data--multiplication, searching
- Typing simplifies internal representation
  - A String requires more and different storage than a boolean

Data types II

- A data type is characterized by:
  - a set of *values*
  - a *data representation,* which is common to all these values, and
  - a set of *operations,* which can be applied uniformly to all these values

Primitive types in Java

- Java provides eight primitive types:
  - boolean
  - char, byte, short, int, long
  - float, double
- Each primitive type has
  - a set of values
  - a data representation
  - a set of operations
- These are "set in stone"—there is nothing the programmer can do to change anything about them

Primitive types as data types

Type	Values	Representation	Operations
boolean	true, false	Single byte	&&, ||, !
char, byte, short, int, long	Integers of varying sizes	Two's complement	+, -, *, /, others
float, double	Floating point numbers of varying sizes and precisions	Two's complement with exponent and mantissa	+, -, *, /, others

Classes in Java

- A class is a *data type*
  - The possible *values* of a class are called objects
  - The *data representation* is a reference (pointer) to a block of storage
    - The structure of this block is defined by the fields (both inherited and immediate) of the class
  - The *operations* on the objects are called methods
- Many classes are defined in Java's packages
- You can (and must) define your own, as well

Methods and operators

- An *operator* typically
  - Is written with non-alphabetic characters: `+`, `*`, `++`, `+=`, `&&`, etc.
  - Is written as prefix, infix, or postfix: `-x`, `x+y`, `x++`
  - Has only one or two arguments, or *operands*
- A *method* (or *function*) typically
  - Is written with letters, and its arguments are enclosed in parentheses: `toString()`, `Math.abs(n)`
  - Has any (predetermined) number of arguments

Insertion into a list

- There are many ways you could insert a new node into a list:

- **As the new first element**
- **As the new last element**
- **Before a given node**
- **After a given node**
- **Before a given value**
- **After a given value**
- **Before the n^{th} element**
- **After the n^{th} element**
- **Before the n^{th} from the end**
- **After the n^{th} from the end**
- **In the correct location to keep the list in sorted order**

- Is it a good idea to supply all of these?
- If not, why not?

Cognitive load

- Human minds are limited—you can't remember everything
  - You probably don't even remember all the Java operators for integers
    - What's the difference between `>>` and `>>>` ?
    - What about between `<<` and `<<<` ?
- We want our operators (and methods) to be *useful* and *worth remembering*

Efficiency

- A list is just a sequence of values—it could be implemented by a linked list *or* by an array
  - Inserting as a new first element is efficient for a linked list representation, inefficient for an array
  - Accessing the n^{th} element is efficient for an array representation, inefficient for a linked list
  - Inserting in the n^{th} position is efficient for neither
- Do we want to make it easy for the user to be inefficient?
- Do we want the user to have to know the implementation?

Abstract Data Types

- An Abstract Data Type (ADT) is:
  - a set of *values*
  - a set of *operations,* which can be applied uniformly to all these values
- To *abstract* is to leave out information, keeping (hopefully) the more important parts
  - What part of a Data Type does an ADT leave out?

Data Structures

- Many kinds of data consist of multiple parts, organized (structured) in some way
- A **data structure** is simply some way of organizing a value that consists of multiple parts
  - Hence, an array is a data structure, but an integer is not
- When we talk about data structures, we are talking about the *implementation* of a data type
- If I talk about the possible *values* of, say, complex numbers, and the *operations* I can perform with them, I am talking about them as an ADT
- If I talk about the way the parts ("real" and "imaginary") of a complex number are *stored in memory,* I am talking about a data structure
- An ADT may be implemented in several different ways
  - A complex number might be stored as two separate doubles, or as an array of two doubles, or even in some bizarre way

Data representation in an ADT

- An ADT must obviously have *some* kind of representation for its data
  - The user need not know the representation
  - The user should not be allowed to tamper with the representation
  - Solution: Make all data private
- But what if it's really more convenient for the user to have direct access to the data?
  - Solution: Use *setters* and *getters*

Example of setters and getters

```
class Pair {
    private int first, last;

    public getFirst() { return first; }
    public setFirst(int first) { this.first = first;
      }

    public getLast() { return last; }
    public setLast(int last) { this.last = last; }
}
```

Naming setters and getters

- Setters and getters should be named by:
  - Capitalizing the first letter of the variable (first becomes First), and
  - Prefixing the name with get or set (setFirst)
  - For boolean variables, replace get with is (for example, isRunning)
- This is more than just a convention—if and when you start using JavaBeans, it becomes a *requirement*

What's the point?

- Setters and getters allow you to keep control of your implementation
- For example, you decide to define a Point in a plane by its x-y coordinates:
  - `class Point { public int x; public int y; }`
- Later on, as you gradually add methods to this class, you decide that it's more efficient to represent a point by its angle and distance from the origin, θ and ρ
- Sorry, you can't do that—you'll break too much code that accesses x and y directly
- If you had used setters and getters, you could redefine them to *compute* x and y from θ and ρ

Contracts

- Every ADT should have a contract (or specification) that:
  - Specifies the set of valid values of the ADT
  - Specifies, for each operation of the ADT:
    - Its name
    - Its parameter types
    - Its result type, if any
    - Its observable behavior
  - Does *not* specify:
    - The data representation
    - The algorithms used to implement the operations

Importance of the contract

- A contract is an agreement between two parties; in this case
  - The implementer of the ADT, who is concerned with making the operations correct and efficient
  - The applications programmer, who just wants to *use* the ADT to get a job done
- It doesn't matter if you are *both* of these parties; the contract is *still essential* for good code
- This separation of concerns is essential in any large project

Promise no more than necessary

- For a general API, the implementer should provide as much generality as feasible
- But for a specific program, the class author should provide only what is essential at the moment
  - In Extreme Programming terms, “You ain’t gonna need it!”
  - In fact, XP practice is to *remove* functionality that isn’t currently needed!
  - Your documentation should not expose anything that the application programmer does not need to know
- If you design for generality, it’s easy to add functionality later—but removing it may have serious consequences

Implementing an ADT

- To implement an ADT, you need to choose:
  - a data representation
    - must be able to represent all necessary values of the ADT
    - should be private
  - an algorithm for each of the necessary operations
    - must be consistent with the chosen representation
    - all auxiliary (helper) operations that are not in the contract should be private

- Remember: Once other people (or other classes) are using your class:
  - It's easy to add functionality
  - You can only remove functionality if no one is using it!

Example contract (Javadoc)

- General description of class

```
/**
 * Each value is a die (singular of dice) with n sides,
 * numbered 1 to n, with one face showing.
 */
public class Die
```

- Constructor

```
/**
 * Constructs a die with faces numbered 1 thru numberOfSides.
 */
public Die(int numberOfSides)
```

- Accessor

```
/**
 * Returns the result of the previous roll.
 */
int lastRoll()
```

- Transformer (mutative)

```
/**
 * Returns the result of a new roll of the die.
 */
int roll()
```

Responsibilities

- A class is responsible for its own values
  - It should protect them from careless or malicious users
- Ideally, a class should be written to be *generally* useful
  - The goal is to make the class reusable
  - The class should *not* be responsible for anything specific to the application in which it is used
- In practice, most classes are application-specific
- Java's classes are, on the whole, extremely well designed
  - They weren't written specifically for *your* program
  - Strive to make your classes more like Java's!

Summary

- A Data Type describes values, representations, and operations
- An Abstract Data Type describes values and operations, but *not* representations
  - An ADT should *protect its data* and *keep it valid*
    - All, or nearly all, data should be private
    - Access to data should be via getters and setters
  - An ADT should provide:
    - A *contract*
    - A *necessary and sufficient* set of operations

The ADT List

- ADT List operations
  - Create an empty list
  - Determine whether a list is empty
  - Determine the number of items in a list
  - Add an item at a given position in the list
  - Remove the item at a given position in the list
  - Remove all the items from the list
  - Retrieve (get) the item at a given position in the list
- Items are referenced by their position within the list

The ADT List

- ADT List operations – needed ?
  - Remove all the items from the list

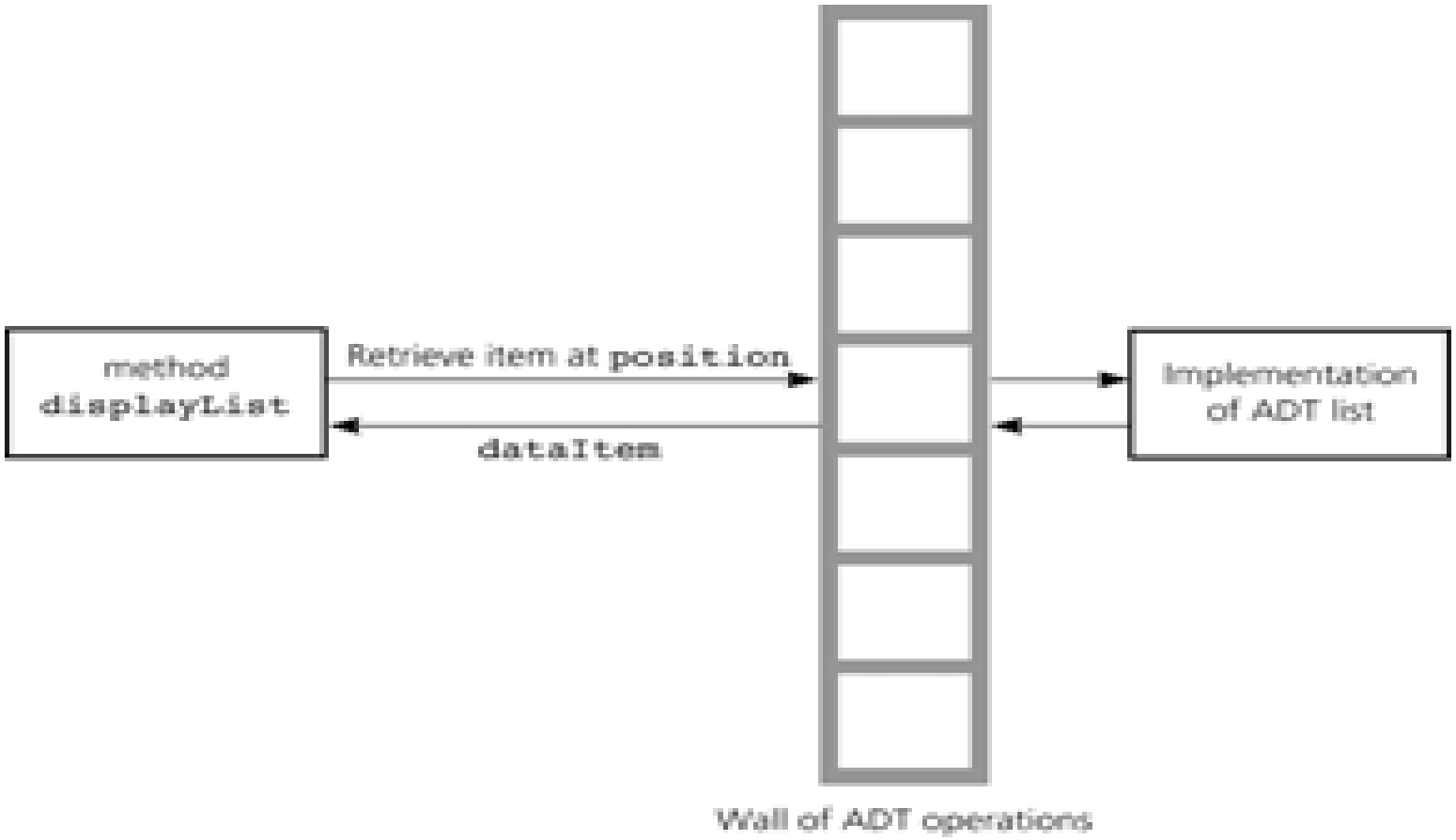
The ADT List

- ADT List operations – all needed ?
  - Remove all the items from the list
    - Determine number of items *n in the list*
    - **for *i=1 to n remove item at position i***

The ADT List

- Specifications of the ADT operations
  - Define the functionality of the ADT list
  - Do not specify how to store the list or how to perform the operations
- ADT operations can be used in an application without the knowledge of how the operations will be implemented

The ADT List



- **The wall between *display List and the implementation of the ADT list***

The ADT Sorted List

- The ADT sorted list
  - Maintains items in sorted order
  - Inserts and deletes items by their values, not their positions

Designing an ADT

- The design of an ADT should evolve naturally during the problem-solving process
- Questions to ask when designing an ADT
  - What data does a problem require?
  - What operations does a problem require?

Implementing ADTs

- Choosing the data structure to represent the ADT's data is a part of implementation
  - Choice of a data structure depends on
    - Details of the ADT's operations
    - Context in which the operations will be used
- Implementation details should be hidden behind a wall of ADT operations
  - A program would only be able to access the data structure using the ADT operations

The List ADT

- A sequence of zero or more elements

$$A_1, A_2, A_3, \ldots A_N$$

- N: length of the list
- A_1: first element
- A_N: last element
- A_i: position i
- If N=0, then empty list
- Linearly ordered
  - A_i precedes A_{i+1}
  - A_i follows A_{i-1}

Operations

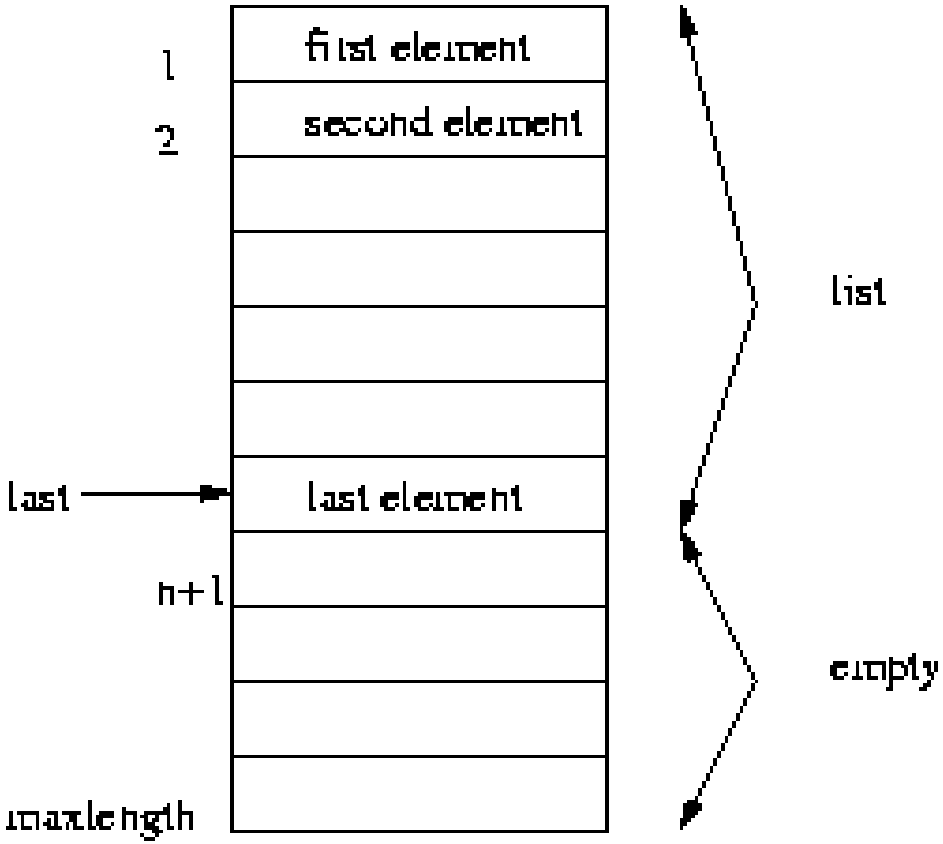
- printList: print the list
- makeEmpty: create an empty list
- find: locate the position of an object in a list
  - list: 34,12, 52, 16, 12
  - find(52) $\rightarrow$ 3
- insert: insert an object to a list
  - insert(x,3) $\rightarrow$ 34, 12, 52, x, 16, 12
- remove: delete an element from the list
  - remove(52) $\rightarrow$ 34, 12, x, 16, 12
- findKth: retrieve the element at a certain position

Implementation of an ADT

- Choose a **data structure** to represent the ADT
  - E.g. arrays, etc.
- Each operation associated with the ADT is implemented by one or more subroutines
- Two standard implementations for the list ADT
  - Array-based
  - Linked list

Array Implementation

- Elements are stored in contiguous array positions



Array Implementation...

- Requires an estimate of the maximum size of the list
  - ➢ waste space
- printList and find: linear
- findKth: constant
- insert and delete: slow
  - e.g. insert at position 0 (making a new element)
    - requires first pushing the entire array down one spot to make room
  - e.g. delete at position 0
    - requires shifting all the elements in the list up one
  - On average, half of the lists needs to be moved for either operation

Pointer Implementation (Linked List)

- Ensure that the list is not stored contiguously
  - use a linked list
  - a series of structures that are not necessarily adjacent in memory

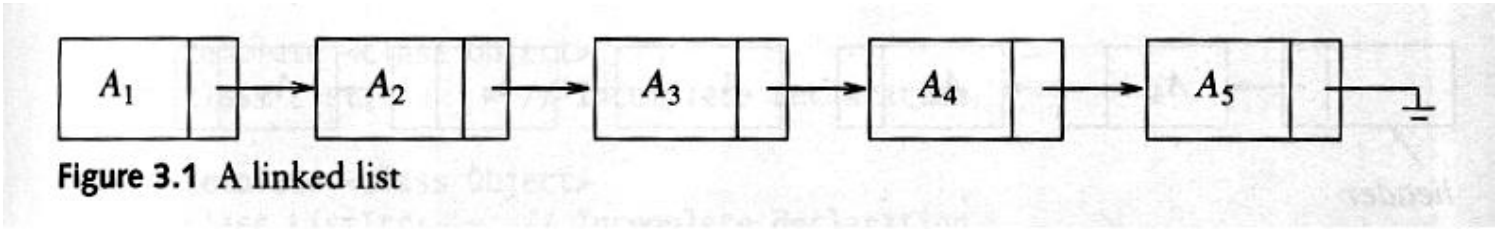
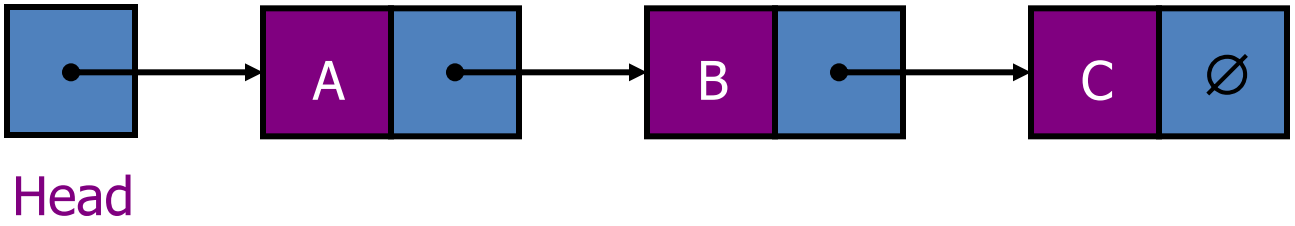


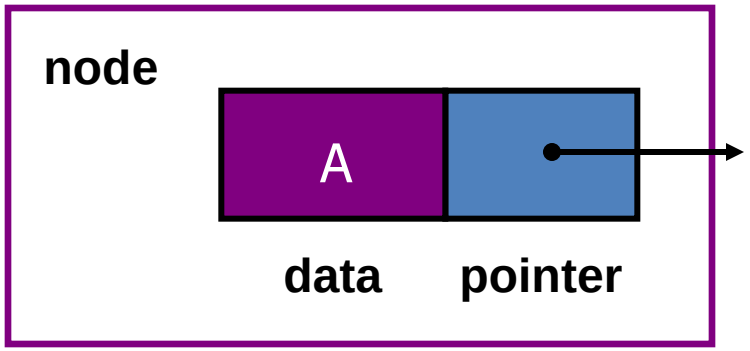
Figure 3.1 A linked list

- Each node contains the element and a pointer to a structure containing its successor
  - the last cell's next link points to NULL
- Compared to the array implementation,
  - ✓ the pointer implementation uses only as much space as is needed for the elements currently on the list
  - ✗ but requires space for the pointers in each cell

Linked Lists



- A *linked list* is a series of connected *nodes*
- Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list
- *Head*: pointer to the first node
- The last node points to `NULL`



A Simple Linked List Class

- We use two classes: **`Node`** and **`List`**
- Declare `Node` class for the nodes
  - `data`: `double`-type data in this example
  - `next`: a pointer to the next node in the list

```
class Node {
public:
      double      data;       // data
      Node*       next;       // pointer to next
};
```

A Simple Linked List Class

- Declare `List`, which contains
  - `head`: a pointer to the first node in the list.
    Since the list is empty initially, `head` is set to `NULL`
  - Operations on `List`

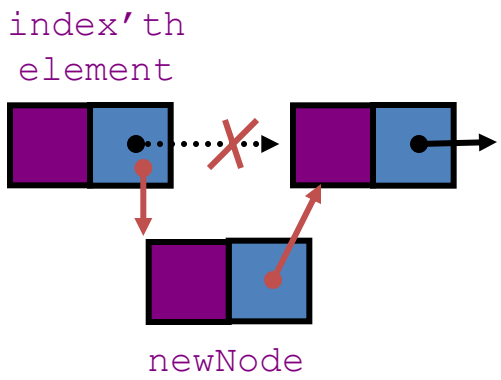
```
class List {
public:
        List(void) { head = NULL; }        // constructor
        ~List(void);                       // destructor

        bool IsEmpty() { return head == NULL; }
        Node* InsertNode(int index, double x);
        int FindNode(double x);
        int DeleteNode(double x);
        void DisplayList(void);
private:
        Node* head;
};
```

A Simple Linked List Class

- Operations of `List`
  - `IsEmpty`: determine whether or not the list is empty
  - `InsertNode`: insert a new node at a particular position
  - `FindNode`: find a node with a given value
  - `DeleteNode`: delete a node with a given value
  - `DisplayList`: print all the nodes in the list

Inserting a new node

- `Node* InsertNode(int index, double x)`
  - Insert a node with data equal to `x` after the `index`'th elements. (i.e., when `index = 0`, insert the node as the first element;
    when `index = 1`, insert the node after the first element, and so on)
  - If the insertion is successful, return the inserted node.
    Otherwise, return `NULL`.
    (If `index` is < 0 or > length of the list, the insertion will fail.)
- Steps
  1. Locate `index`'th element
  2. Allocate memory for the new node
  3. Point the new node to its successor
  4. Point the new node's predecessor to the new node



Inserting a new node

- Possible cases of `InsertNode`
  1. Insert into an empty list
  2. Insert in front
  3. Insert at back
  4. Insert in middle
- But, in fact, only need to handle two cases
  - Insert as the first node (Case 1 and Case 2)
  - Insert in the middle or at the end of the list (Case 3 and Case 4)

Inserting a new node

```
Node* List::InsertNode(int index, double x) {
        if (index < 0) return NULL;

        int currIndex  =       1;
        Node* currNode =       head;
        while (currNode && index > currIndex) {
                currNode       =       currNode->next;
                currIndex++;
        }
        if (index > 0 && currNode == NULL) return NULL;

        Node* newNode  =       new     Node;
        newNode->data  =       x;
        if (index == 0) {
                newNode->next  =       head;
                head           =       newNode;
        }
        else {
                newNode->next  =       currNode->next;
                currNode->next =       newNode;
        }
        return newNode;
}
```

Try to locate `index`'th node. If it doesn't exist, return `NULL`.

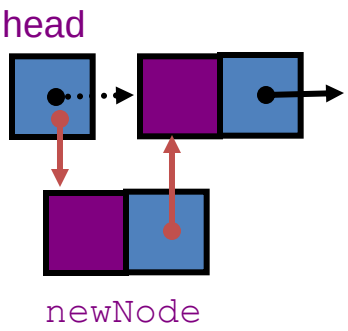
Inserting a new node

```
Node* List::InsertNode(int index, double x) {
        if (index < 0) return NULL;

        int currIndex  =       1;
        Node* currNode =       head;
        while (currNode && index > currIndex) {
                currNode       =       currNode->next;
                currIndex++;
        }
        if (index > 0 && currNode == NULL) return NULL;

        Node* newNode  =       new     Node;
        newNode->data  =       x;
        if (index == 0) {
                newNode->next  =       head;
                head           =       newNode;
        }
        else {
                newNode->next  =       currNode->next;
                currNode->next =       newNode;
        }
        return newNode;
}
```

Inserting a new node

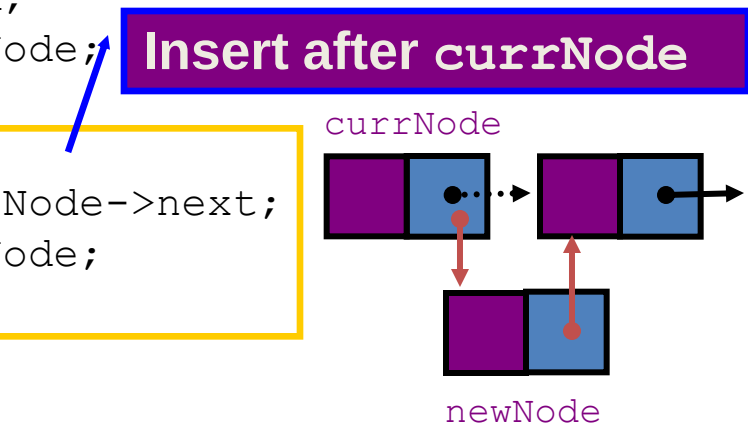
```
Node* List::InsertNode(int index, double x) {
        if (index < 0) return NULL;

        int currIndex  =       1;
        Node* currNode =       head;
        while (currNode && index > currIndex) {
                currNode       =       currNode->next;
                currIndex++;
        }
        if (index > 0 && currNode == NULL) return NULL;

        Node* newNode  =       new     Node;
        newNode->data  =       x;
        if (index == 0) {
                newNode->next  =       head;
                head           =       newNode;
        }
        else {
                newNode->next  =       currNode->next;
                currNode->next =       newNode;
        }
        return newNode;
}
```

Insert as first element



Inserting a new node

```
Node* List::InsertNode(int index, double x) {
        if (index < 0) return NULL;

        int currIndex  =       1;
        Node* currNode =       head;
        while (currNode && index > currIndex) {
                currNode       =       currNode->next;
                currIndex++;
        }
        if (index > 0 && currNode == NULL) return NULL;

        Node* newNode  =       new     Node;
        newNode->data  =       x;
        if (index == 0) {
                newNode->next  =       head;
                head           =       newNode;
        }
        else {
                newNode->next  =       currNode->next;
                currNode->next =       newNode;
        }
        return newNode;
}
```



Finding a node

- `int FindNode(double x)`
  - Search for a node with the value equal to `x` in the list.
  - If such a node is found, return its position. Otherwise, return 0.

```
int List::FindNode(double x) {
      Node* currNode    =     head;
      int currIndex     =     1;
      while (currNode && currNode->data != x) {
            currNode    =     currNode->next;
            currIndex++;
      }
      if (currNode) return currIndex;
      return 0;
}
```

Deleting a node

- `int DeleteNode(double x)`
  - Delete a node with the value equal to `x` from the list.
  - If such a node is found, return its position. Otherwise, return 0.
- Steps
  - Find the desirable node (similar to `FindNode`)
  - Release the memory occupied by the found node
  - Set the pointer of the predecessor of the found node to the successor of the found node
- Like `InsertNode`, there are two special cases
  - Delete first node
  - Delete the node in middle or at the end of the list

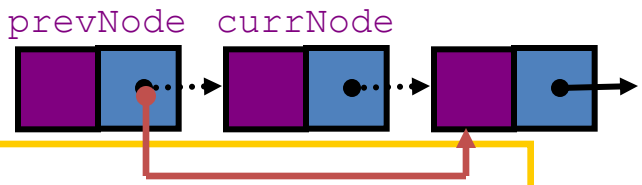
Deleting a node

```
int List::DeleteNode(double x) {
        Node* prevNode =        NULL;
        Node* currNode =        head;
        int currIndex  =        1;
        while (currNode && currNode->data != x) {
                prevNode        =       currNode;
                currNode        =       currNode->next;
                currIndex++;
        }
        if (currNode) {
                if (prevNode) {
                        prevNode->next =        currNode->next;
                        delete currNode;
                }
                else {
                        head            =       currNode->next;
                        delete currNode;
                }
                return currIndex;
        }
        return 0;
}
```

Try to find the node with its value equal to `x`

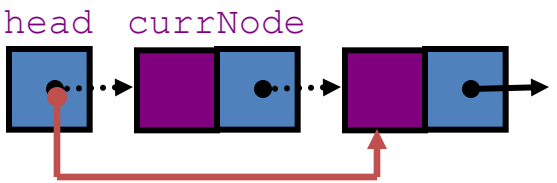
Deleting a node

```
int List::DeleteNode(double x) {
        Node* prevNode =        NULL;
        Node* currNode =        head;
        int currIndex  =        1;
        while (currNode && currNode->data != x) {
                prevNode        =       currNode;
                currNode        =       currNode->next;
                currIndex++;
        }
        if (currNode) {
                if (prevNode) {
                        prevNode->next =        currNode->next;
                        delete currNode;
                }
                else {
                        head            =       currNode->next;
                        delete currNode;
                }
                return currIndex;
        }
        return 0;
}
```



Deleting a node

```
int List::DeleteNode(double x) {
        Node* prevNode =        NULL;
        Node* currNode =        head;
        int currIndex  =        1;
        while (currNode && currNode->data != x) {
                prevNode        =       currNode;
                currNode        =       currNode->next;
                currIndex++;
        }
        if (currNode) {
                if (prevNode) {
                        prevNode->next =        currNode->next;
                        delete currNode;
                }
                else {
                        head            =       currNode->next;
                        delete currNode;
                }
                return currIndex;
        }
        return 0;
}
```



Printing all the elements

- `void DisplayList(void)`
  - Print the data of all the elements
  - Print the number of the nodes in the list

```
void List::DisplayList()
{
   int num           =      0;
   Node* currNode    =      head;
   while (currNode != NULL){
       cout << currNode->data << endl;
       currNode      =      currNode->next;
       num++;
   }
   cout << "Number of nodes in the list: " << num << endl;
}
```

Destroying the list

- `~List(void)`
  - Use the destructor to release all the memory used by the list.
  - Step through the list and delete each node one by one.

```
List::~List(void) {
    Node* currNode = head, *nextNode = NULL;
    while (currNode != NULL)
    {
        nextNode    =     currNode->next;
        // destroy the current node
        delete currNode;
        currNode    =     nextNode;
    }
}
```

Using `List`

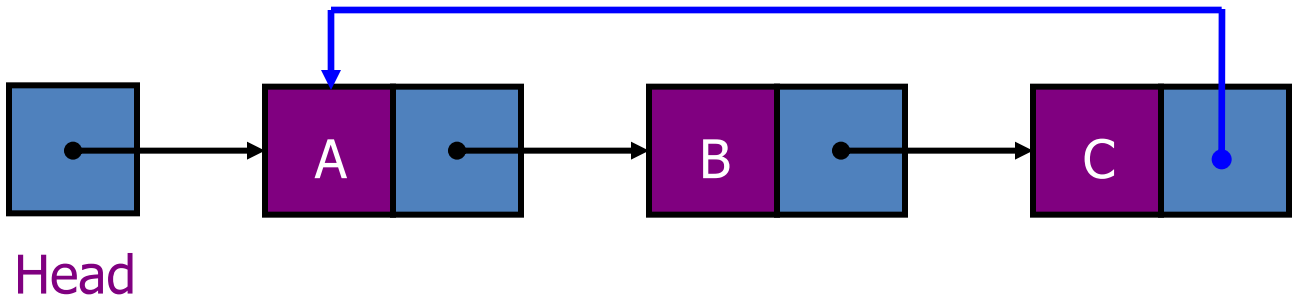
```
int main(void)
{
        List list;
        list.InsertNode(0, 7.0);   // successful
        list.InsertNode(1, 5.0);   // successful
        list.InsertNode(-1, 5.0);  // unsuccessful
        list.InsertNode(0, 6.0);   // successful
        list.InsertNode(8, 4.0);   // unsuccessful
        // print all the elements
        list.DisplayList();
        if(list.FindNode(5.0) > 0) cout << "5.0 found" << endl;
        else                       cout << "5.0 not found" << endl;
        if(list.FindNode(4.5) > 0) cout << "4.5 found" << endl;
        else                       cout << "4.5 not found" << endl;
        list.DeleteNode(7.0);
        list.DisplayList();
        return 0;
}
```

result

```
6
7
5
Number of nodes in the list: 3
5.0 found
4.5 not found
6
5
Number of nodes in the list: 2
```

Variations of Linked Lists

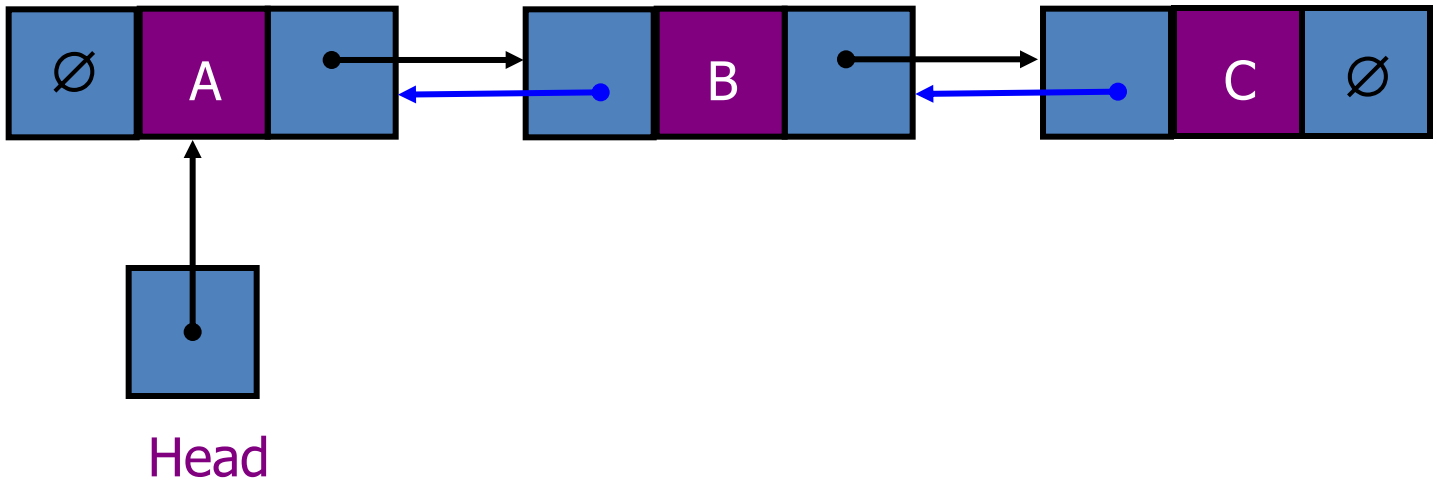
- *Circular linked lists*
  - The last node points to the first node of the list



  - How do we know when we have finished traversing the list? (Tip: check if the pointer of the current node is equal to the head.)

Variations of Linked Lists

- *Doubly linked lists*
  - Each node points to not only successor but the predecessor
  - There are two `NULL`: at the first and last nodes in the list
  - Advantage: given a node, it is easy to visit its predecessor. Convenient to traverse lists backwards



Array versus Linked Lists

- Linked lists are more complex to code and manage than arrays, but they have some distinct advantages.
  - **Dynamic**: a linked list can easily grow and shrink in size.
    - We don't need to know how many nodes will be in the list. They are created in memory as needed.
    - In contrast, the size of a C++ array is fixed at compilation time.
  - **Easy and fast insertions and deletions**
    - To insert or delete an element in an array, we need to copy to temporary variables to make room for new elements or close the gap caused by deleted elements.
    - With a linked list, no need to move other nodes. Only need to reset some pointers.