Data Structures
(Linked List)

Lecture#6

Agenda

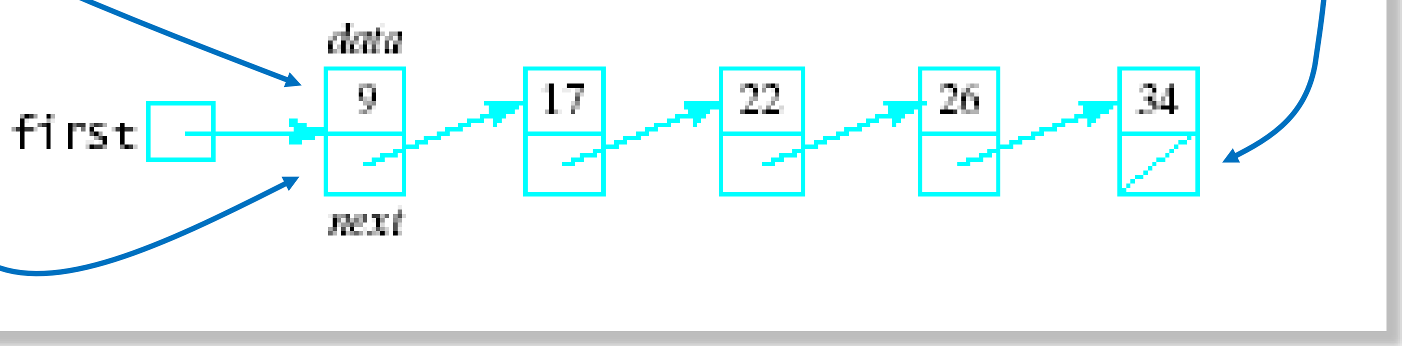
- Linked lists (pointer-based implementation)
  - Insertion (start, middle, and end of the list)
  - Deletion (start, middle, and end of the list)
  - Searching
  - Destroying a list

- Variations of the linked-list

Pointers-Based Implementation of Lists (Linked List)

Linked List

- Linked list nodes composed of two parts
  - Data part
    - ➢ Stores an element of the list
  - Next (pointer) part
    - ➢ Stores link/address/pointer to next element
    - ➢ Stores Null value, when no next element



Simple Linked List Class (1)

- 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 Node
};
```

Simple Linked List Class (2)

- 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`

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

        bool IsEmpty() { return head == NULL; }
        bool Insert(int index, double x);
        int Find(double x);
        int Delete(double x);
        void DisplayList(void);
    private:
        Node* head;
};
```

Simple Linked List Class (3)

Operations of List

- `IsEmpty`: determine whether or not the list is empty
- `Insert`: insert a new node at a particular position
- `Find`: find a node with a given value
- `Delete`: delete a node with a given value
- `DisplayList`: print all the nodes in the list

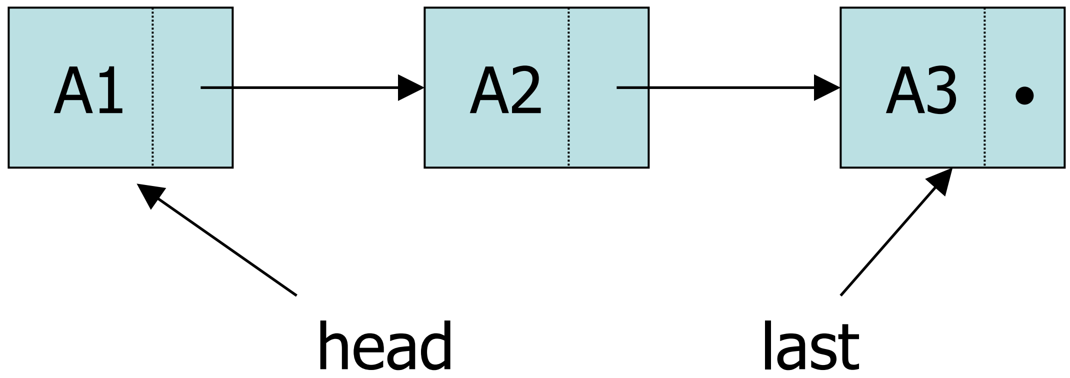
Inserting a New Node

- `bool Insert(int index, double x)`
  - Insert a node with data equal to x at the index elements
  - If the insertion is successful
    - Return `true`
    - Otherwise, return `false`
  - If index is <= 0 or > length of the list, the insertion will fail

- Steps
  1. Locate the node at the position one less than index [list is indexed from 1 to n]
  2. Allocate memory for the new node, copy data into node
  3. Point the new node to its successor (next node)
  4. Point the new node's predecessor (preceding node) to the new node

Insertion After The Last Element (1)

- Suppose `last` points to the last element of the list
  - We can add a new last item x by doing this

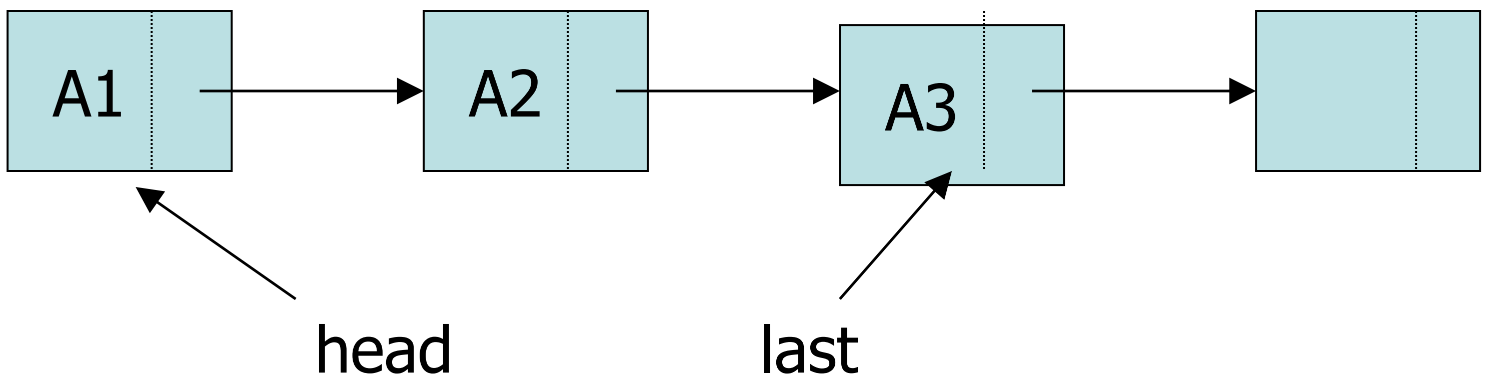



```
last->next = new Node();
last = last->next;
last->data = x;
last->next = null;
```

Steps
- Locate the index element
- Allocate memory for the new node
- Copy data into node
- Point the new node to its successor (next node)
- Point the new node's predecessor (preceding node) to the new node

Insertion After The Last Element (2)

- Suppose `last` points to the last element of the list
  - We can add a new last item x by doing this

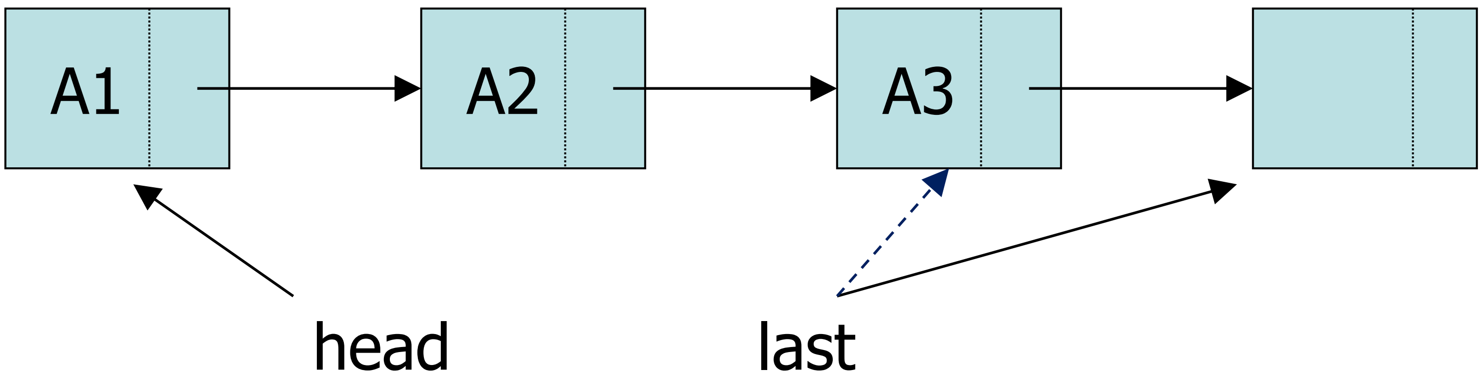



```
last->next = new Node();
last = last->next;
last->data = x;
last->next = null;
```

Steps
- Locate the index element
- Allocate memory for the new node
- Copy data into node
- Point the new node to its successor (next node)
- Point the new node's predecessor (preceding node) to the new node

Insertion After The Last Element (3)

- Suppose `last` points to the last element of the list
  - We can add a new last item x by doing this

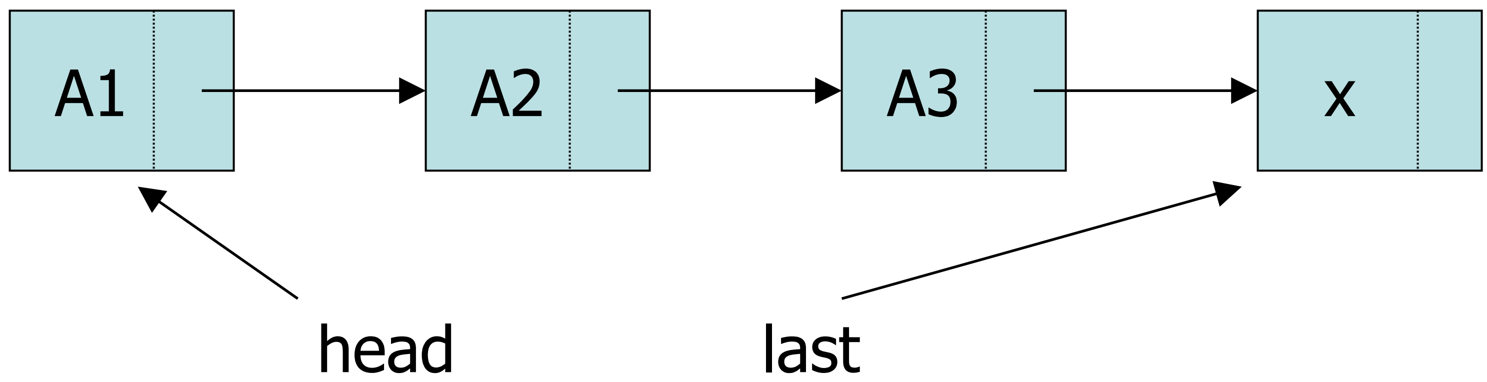



```
last->next = new Node();
last = last->next;
last->data = x;
last->next = null;
```

Steps
- Locate the index element
- Allocate memory for the new node
- Copy data into node
- Point the new node to its successor (next node)
- Point the new node's predecessor (preceding node) to the new node

Insertion After The Last Element (4)

- Suppose `last` points to the last element of the list
  - We can add a new last item x by doing this



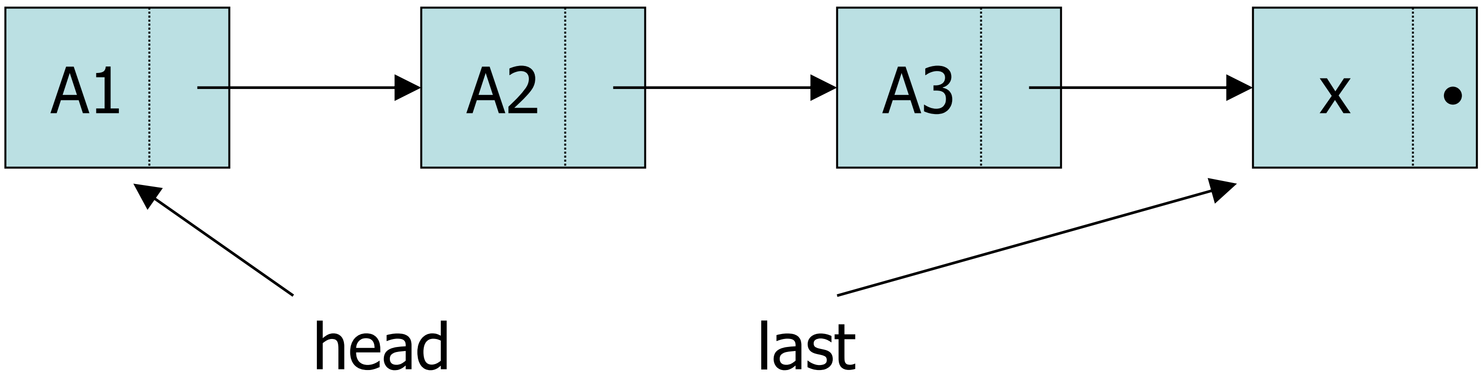

```
last->next = new Node();
last = last->next;
last->data = x;
last->next = null;
```

Steps

- Locate the index element
- Allocate memory for the new node
- Copy data into node
- Point the new node to its successor (next node)
- Point the new node's predecessor (preceding node) to the new node

Insertion After The Last Element (4)

- Suppose `last` points to the last element of the list
  - We can add a new last item x by doing this



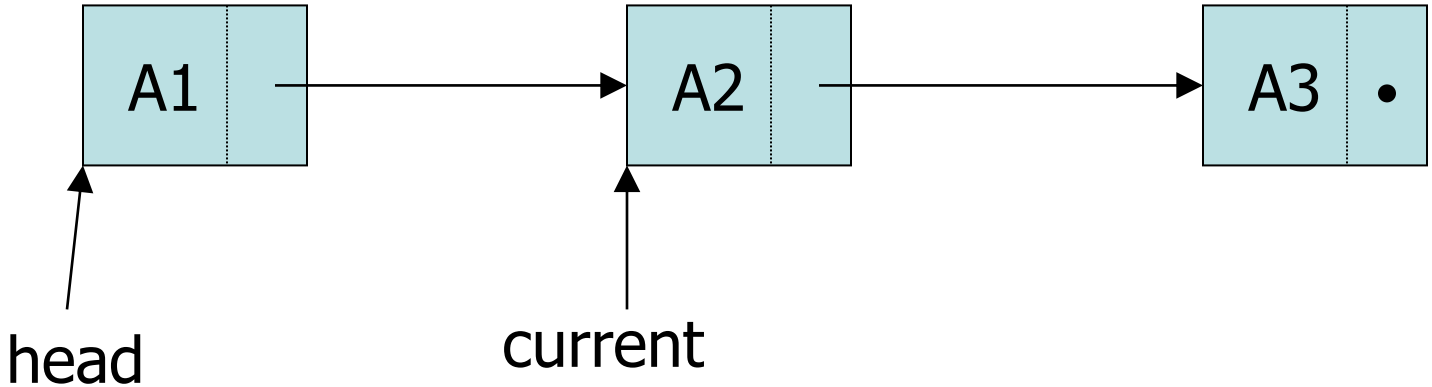

```
last->next = new Node();
last = last->next;
last->data = x;
last->next = null;
```

Steps

- Locate the index element
- Allocate memory for the new node
- Copy data into node
- Point the new node to its successor (next node)
- Point the new node's predecessor (preceding node) to the new node

Insertion At The Middle (1)

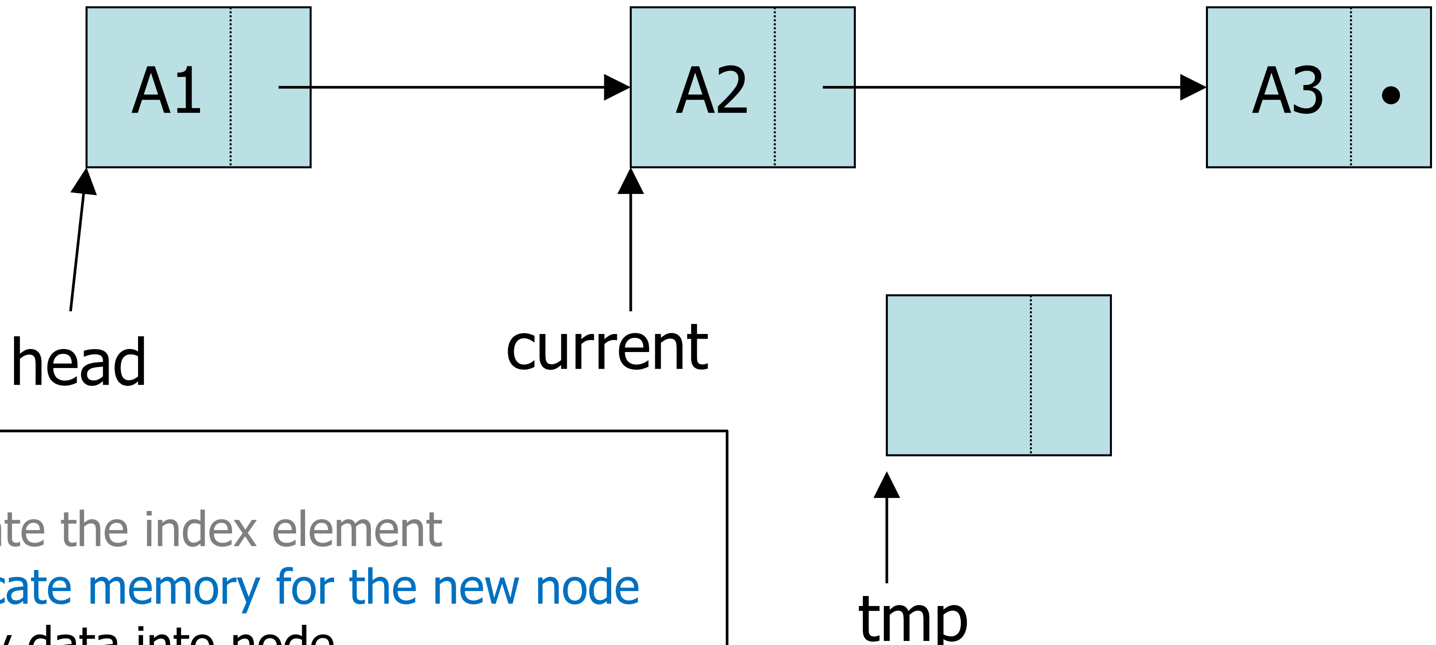
- Suppose `current` points to the middle element of the list
  - We can add a new item x by doing this




```
tmp = new Node();
tmp->data= x;
tmp->next = current->next;
current->next = tmp;
```

Insertion At The Middle (1)

- Suppose `current` points to the middle element of the list
  - We can add a new item x by doing this



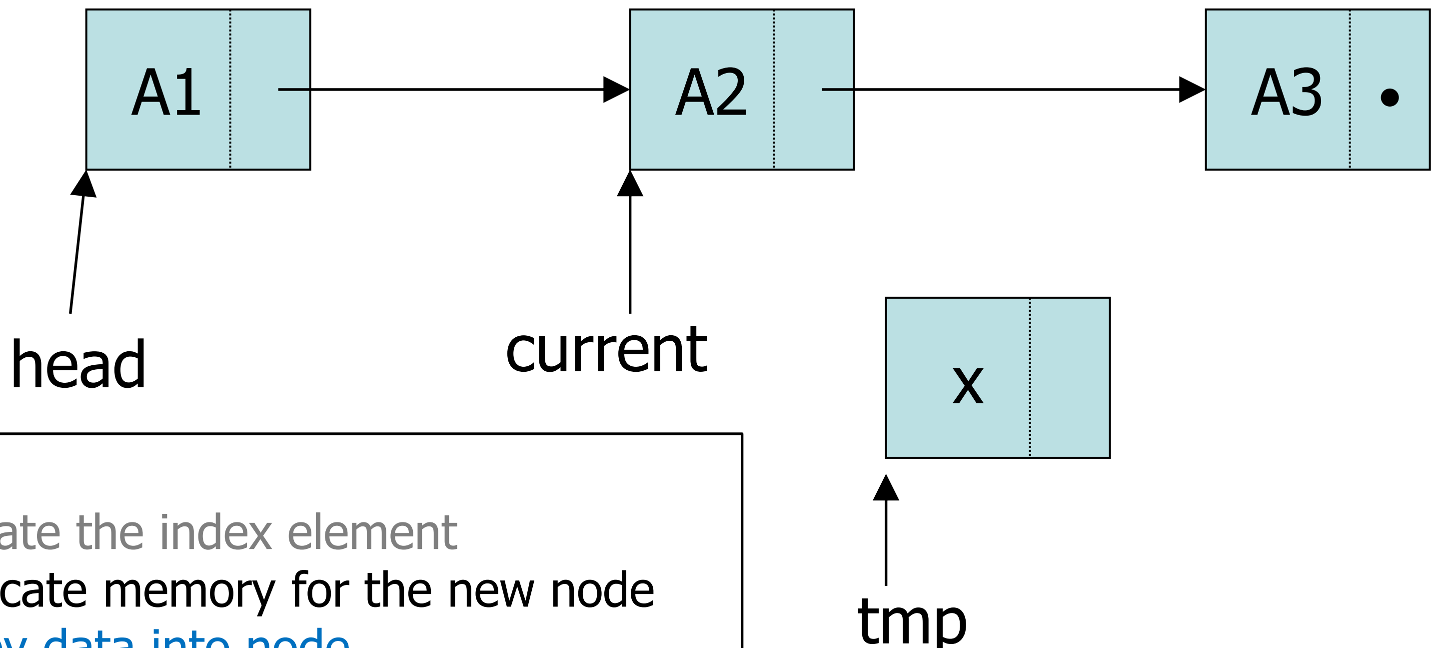
Steps

- Locate the index element
- Allocate memory for the new node
- Copy data into node
- Point the new node to its successor (next node)
- Point the new node's predecessor (preceding node) to the new node

```
tmp = new Node();
tmp->data= x;
tmp->next = current->next;
current->next = tmp;
```

Insertion At The Middle (1)

- Suppose `current` points to the middle element of the list
  - We can add a new item x by doing this



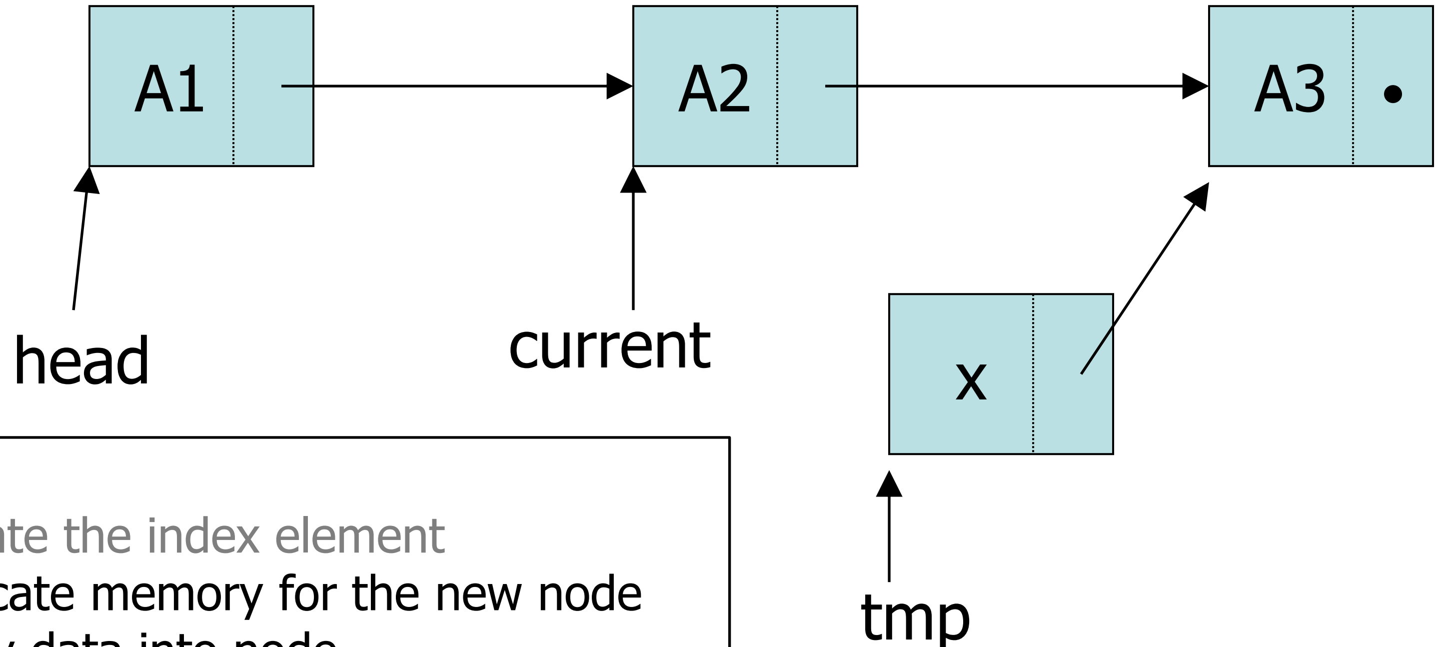
Steps

- Locate the index element
- Allocate memory for the new node
- Copy data into node
- Point the new node to its successor (next node)
- Point the new node's predecessor (preceding node) to the new node

```
tmp = new Node();
tmp->data= x;
tmp->next = current->next;
current->next = tmp;
```

Insertion At The Middle (1)

- Suppose `current` points to the middle element of the list
  - We can add a new item x by doing this

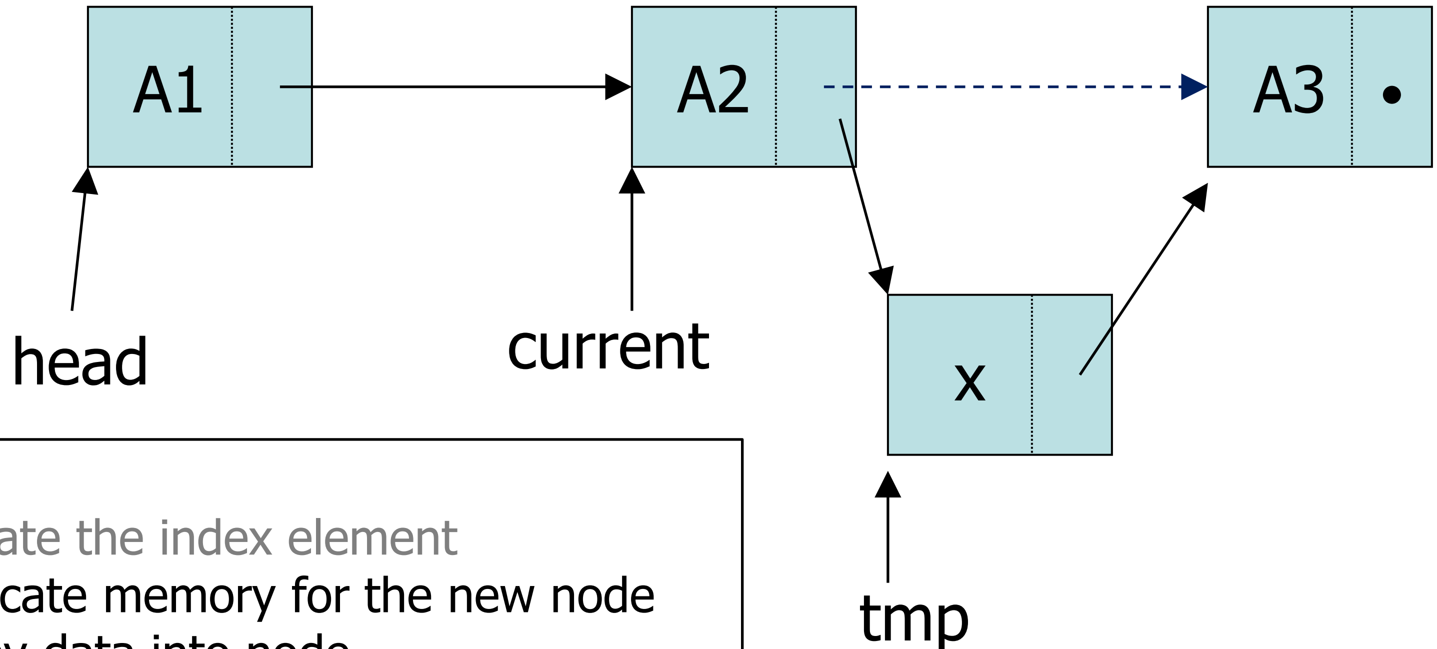


Steps
- Locate the index element
- Allocate memory for the new node
- Copy data into node
- Point the new node to its successor (next node)
- Point the new node's predecessor (preceding node) to the new node

```
tmp = new Node();
tmp->data= x;
tmp->next = current->next;
current->next = tmp;
```

Insertion At The Middle (1)

- Suppose `current` points to the middle element of the list
  - We can add a new item x by doing this



Steps
- Locate the index element
- Allocate memory for the new node
- Copy data into node
- Point the new node to its successor (next node)
- Point the new node's predecessor (preceding node) to the new node

```
tmp = new Node();
tmp->data= x;
tmp->next = current->next;
current->next = tmp;
```

Inserting a New Node (2)

- Possible cases of `Insert`
  1. Insert into an empty list
  2. Insert at front
  3. Insert at back
  4. Insert in middle

- 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 (3)

```
bool List::Insert(int index, double x) {
        if (index <= 0) return false;

        int currIndex   =  2;
        Node* currNode  =  head;
        while (currNode && index > currIndex) {
                currNode = currNode->next;
                currIndex++;
        }
        if (index > 1 && currNode == NULL) return false;

        Node* newNode =  new Node;
        newNode->data =  x;
        if (index == 1) {
                newNode->next  =  head;
                head           =  newNode;
        }
        else {
                newNode->next   =  currNode->next;
                currNode->next  =  newNode;
        }
        return true;
}
```

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

Inserting a New Node (3)

```
bool List::Insert(int index, double x) {
        if (index <= 0) return false;

        int currIndex   =  2;
        Node* currNode  =  head;
        while (currNode && index > currIndex) {
                currNode = currNode->next;
                currIndex++;
        }
        if (index > 1 && currNode == NULL) return false;

        Node* newNode =  new Node;
        newNode->data =  x;
        if (index == 1) {
                newNode->next  =  head;
                head           =  newNode;
        }
        else {
                newNode->next   =  currNode->next;
                currNode->next  =  newNode;
        }
        return true;
}
```

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

Create a new node

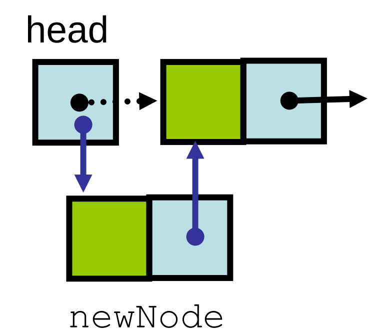
Inserting a New Node (3)

```
bool List::Insert(int index, double x) {
        if (index <= 0) return false;

        int currIndex   =  2;
        Node* currNode  =  head;
        while (currNode && index > currIndex) {
                currNode  = currNode->next;
                currIndex++;
        }
        if (index > 1 && currNode == NULL) return false;

        Node* newNode =  new Node;
        newNode->data =  x;
        if (index == 1) {
                newNode->next   =   head;
                head             =   newNode;
        }
        else {
                newNode->next    =  currNode->next;
                currNode->next   =  newNode;
        }
        return true;
}
```

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

Create a new node

Insert as first element



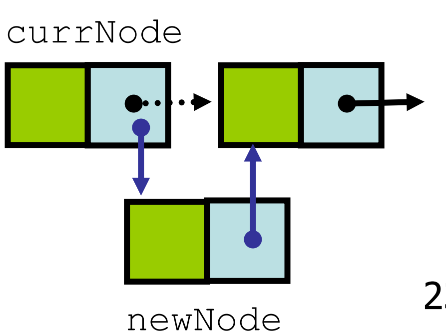
Inserting a New Node (3)

```
bool List::Insert(int index, double x) {
	if (index <= 0) return false;

	int currIndex	=  2;
	Node* currNode	=  head;
	while (currNode && index > currIndex) {
		currNode  = currNode->next;
		currIndex++;
	}
	if (index > 1 && currNode == NULL) return false;

	Node* newNode =  new Node;
	newNode->data =  x;
	if (index == 1) {
		newNode->next  =  head;
		head           =  newNode;
	}
	else {
		newNode->next   =  currNode->next;
		currNode->next  =  newNode;
	}
	return true;
}
```

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

Create a new node

Insert after `currNode`



A Quick Home Work

- Create a linked list of five node

- Dry run the following cases
  1. Add a new node at the beginning i.e., at index 1
  2. Add a new node at the end (i.e., 7^{th} index due to insertion in step 1)
  3. Add node at index 2
  4. Add new node at index 3
  5. Add a new node at index 5
  6. Try Adding a new node index 17

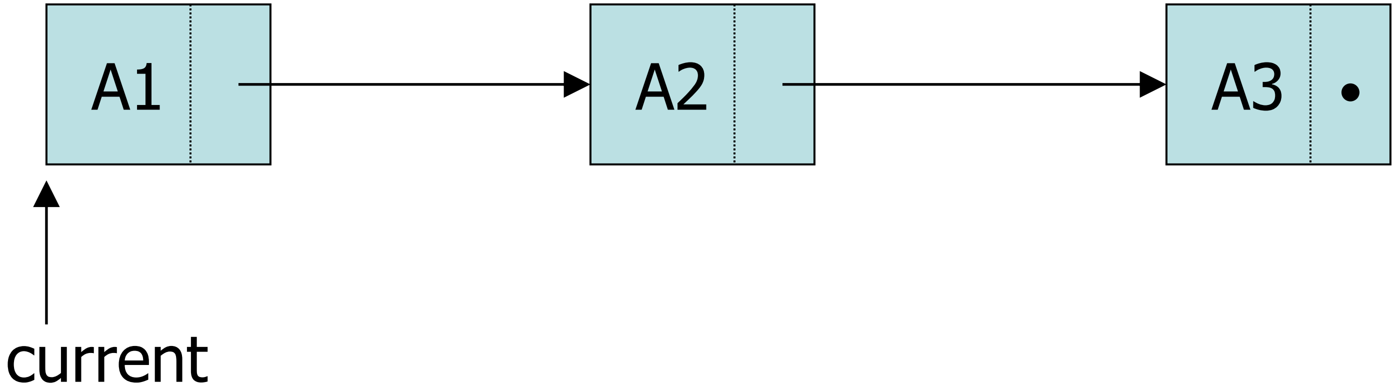
Finding a Node

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

        return 0;
}
```

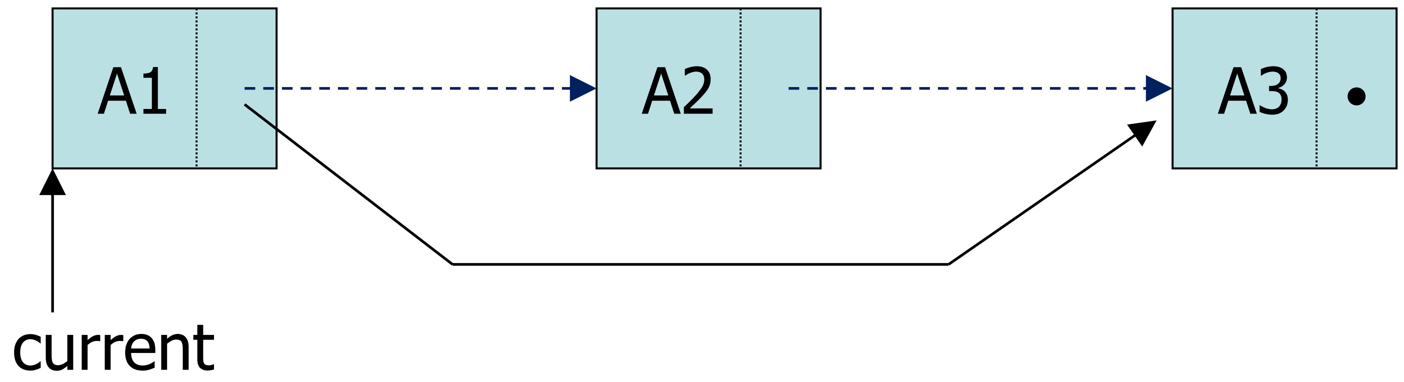
Deleting a Node – Example (1)

- Deleting item A2 from the list



Deleting a Node – Example (2)

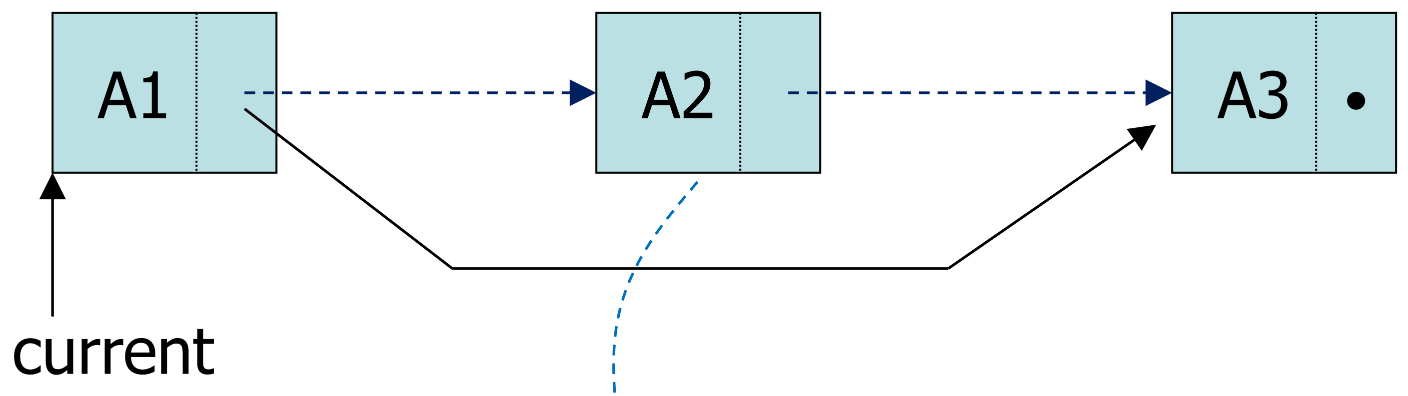
- Deleting item A2 from the list




```
current->next = current->next->next;
```

Deleting a Node – Example (3)

- Deleting item A2 from the list

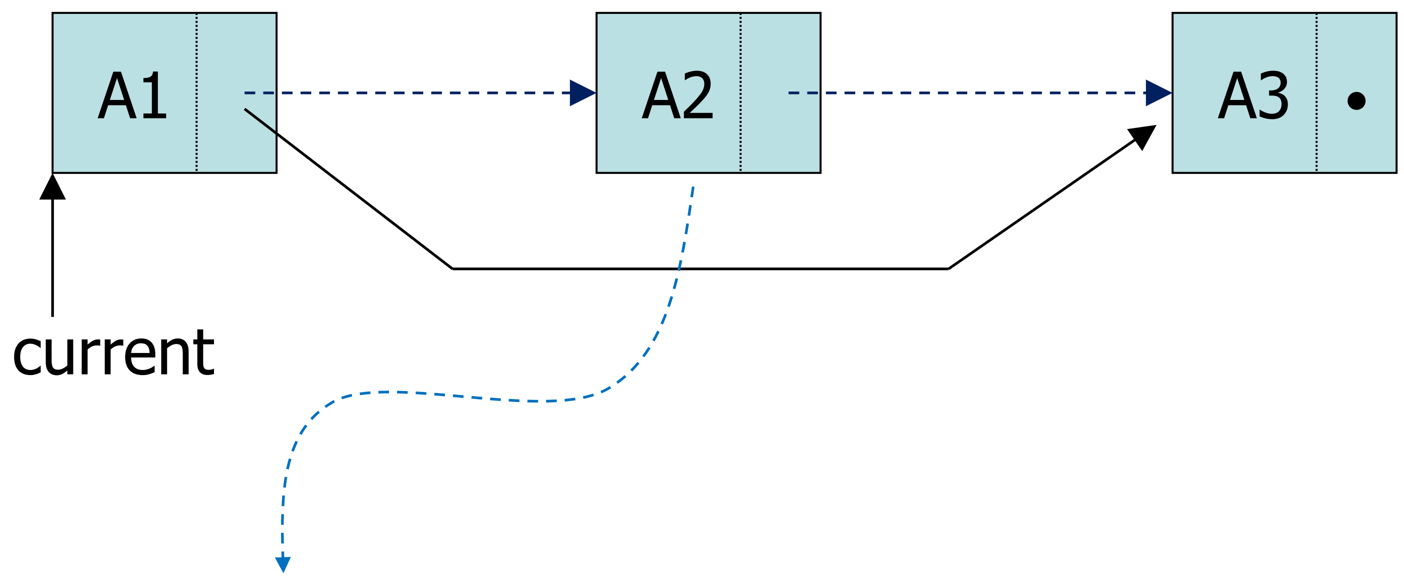



```
current->next = current->next->next;
```

→ Memory Leak

Deleting a Node – Example (4)

- Deleting item **A2** from the list




```
Node *deletedNode = current->next;
current->next = current->next->next;
delete deletedNode;
```

Deleting a Node

- **`int Delete(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 `Find`)
  - Set the pointer of the predecessor of the found node to the successor of the found node
  - Release the memory occupied by the found node

- Like `Insert`, there are two special cases
  - Delete first node
  - Delete the node in middle or at the end of the list

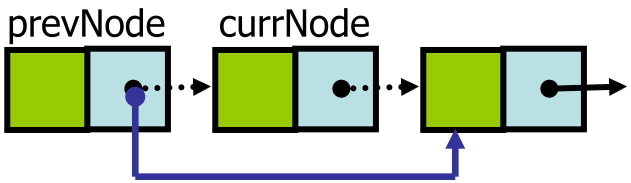
Deleting a Node – Implementation (1)

```
int List::Delete(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 node with its value equal to x.

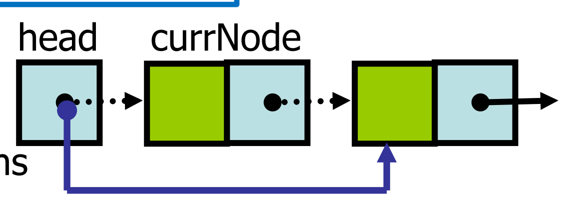
Deleting a Node – Implementation (2)

```
int List::Delete(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;
}
```



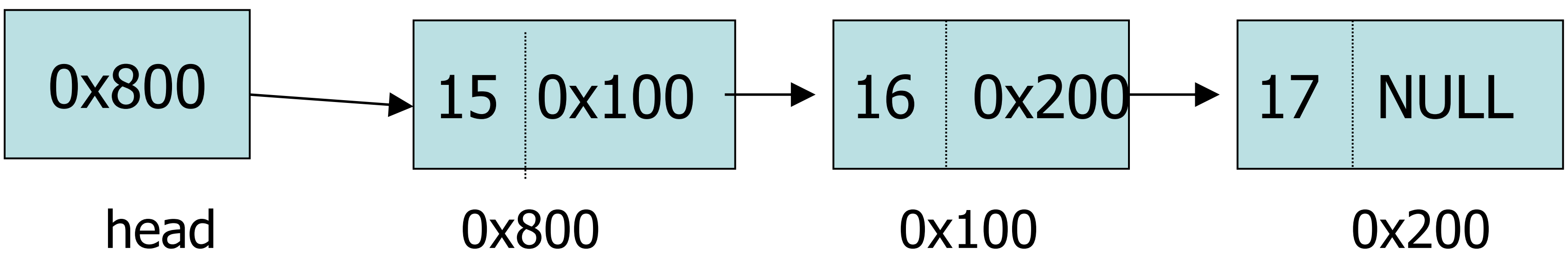
Deleting a Node – Implementation (3)

```
int List::Delete(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;
}
```



Quick Participation-2

1. Assuming that your list is empty try deleting a node with data=15.5 (i.e., call *delete(15.5)*)
2. For the following disjoint tasks assume the list be:



   i. Try deleting node with data=15
   ii. Try deleting node with data=16
   iii. Try deleting node with data=17

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;
   Node* nextNode = NULL;
   while (currNode != NULL)
   {
      nextNode  =  currNode->next;
      delete currNode; // destroy the current node
      currNode  =  nextNode;
   }
}
```

Using List (1)

```
int main(void)
{
   List list;
   list.Insert(1, 7.0);       // successful
   list.Insert(2, 5.0);       // successful
   list.Insert(-1, 5.0);      // unsuccessful
   list.Insert(1, 6.0);       // successful
   list.Insert(8, 4.0);       // unsuccessful
   // print all the elements
   list.DisplayList();

   return 0;
}
```

Output:
6
7
5
Number of nodes in the list: 3

Using List (2)

```
int main(void)
{
   List list;
   list.Insert(1, 7.0);        // successful
   list.Insert(2, 5.0);        // successful
   list.Insert(-1, 5.0);       // unsuccessful
   list.Insert(1, 6.0);        // successful
   list.Insert(8, 4.0);        // unsuccessful
   // print all the elements
   list.DisplayList();
   if(list.Find(5.0) > 0)      cout << "5.0 found" << endl;
   else                        cout << "5.0 not found" << endl;
   if(list.Find(4.5) > 0) cout << "4.5 found" << endl;
   else                        cout << "4.5 not found" << endl;


   return 0;
}
```

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

Using List

```
int main(void)
{
    List list;
    list.Insert(1, 7.0);        // successful
    list.Insert(2, 5.0);        // successful
    list.Insert(-1, 5.0);       // unsuccessful
    list.Insert(1, 6.0);        // successful
    list.Insert(8, 4.0);        // unsuccessful
    // print all the elements
    list.DisplayList();
    if(list.Find(5.0) > 0)      cout << "5.0 found" << endl;
    else                        cout << "5.0 not found" << endl;
    if(list.Find(4.5) > 0) cout << "4.5 found" << endl;
    else                        cout << "4.5 not found" << endl;
    list.Delete(7.0);
    list.DisplayList();
    return 0;
}
```

Output:
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

Any Question So Far?