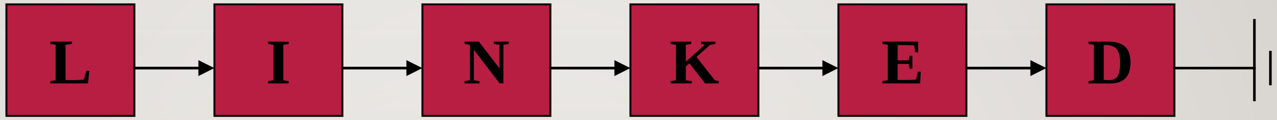



# LINKED LISTS

NATIONAL UNIVERSITY OF TECHNOLOGY (NUTECH)

DR. SAMAN RIAZ

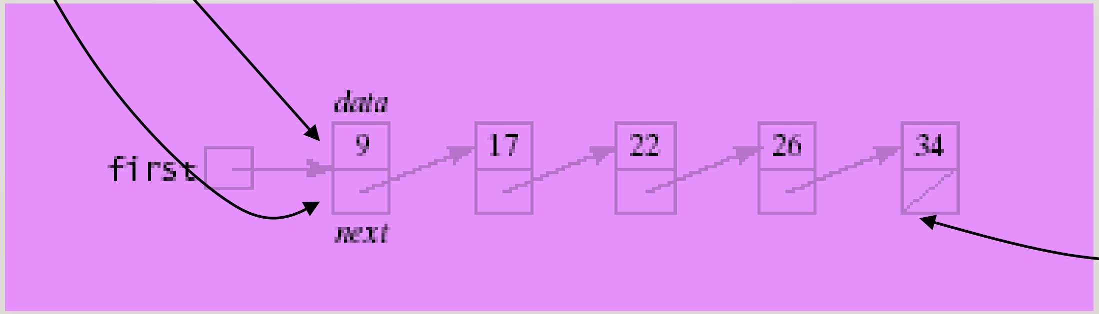
LECTURE # 8

# LINKED LIST USING POINTERS BASED IMPLEMENTATION OF 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)



# IMPLEMENTATION OVERVIEW

# 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`

```
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

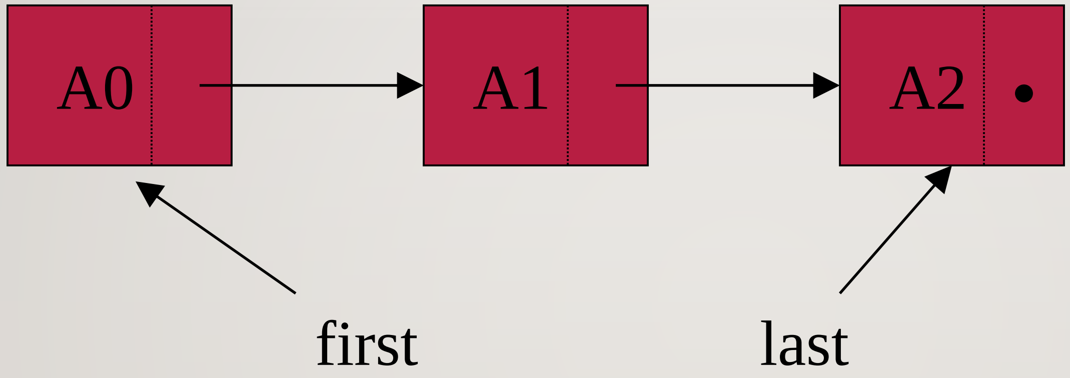
- Node* InsertNode(int index, double x)
  - Insert a node with data equal to x after the index'th elements.
  - If the insertion is successful, return the inserted node. Otherwise, return NULL.

    (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, 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

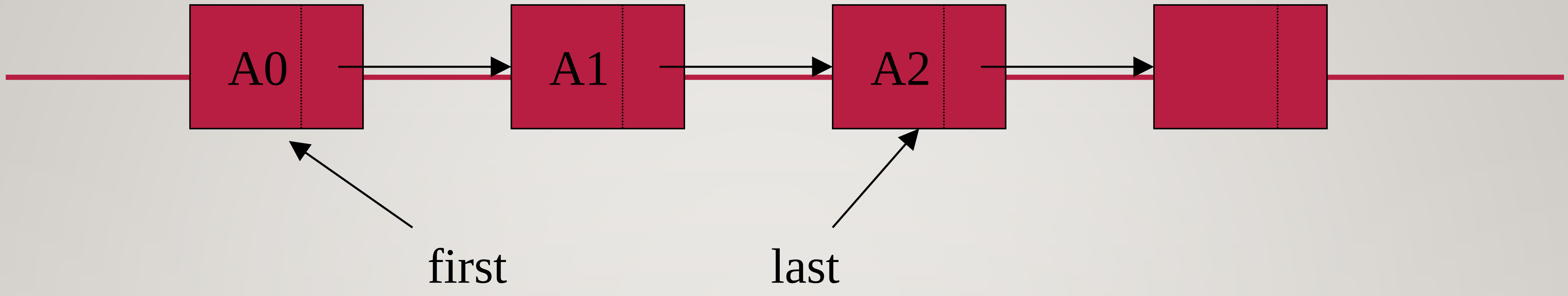



At any point, we can add a new last item **x** by doing this (after locating last element):

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

Steps

1. Locate index'th element
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

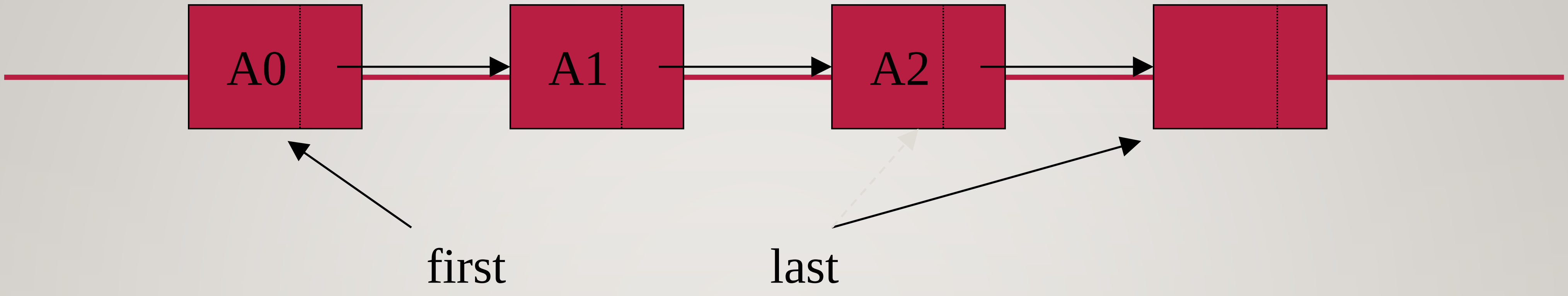



At any point, we can add a new last item **x** by doing this:

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

Steps

1. Locate index'th element
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**

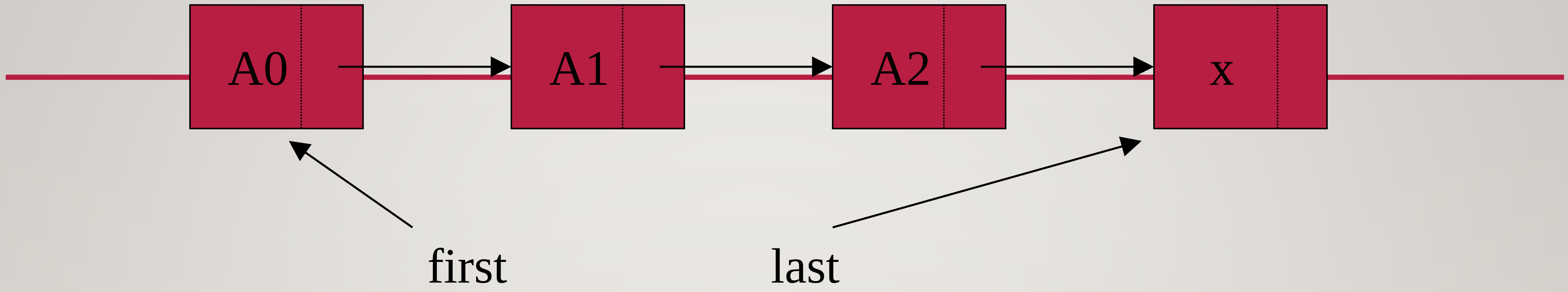



At any point, we can add a new last item **x** by doing this:

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

Steps

1. Locate index'th element
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

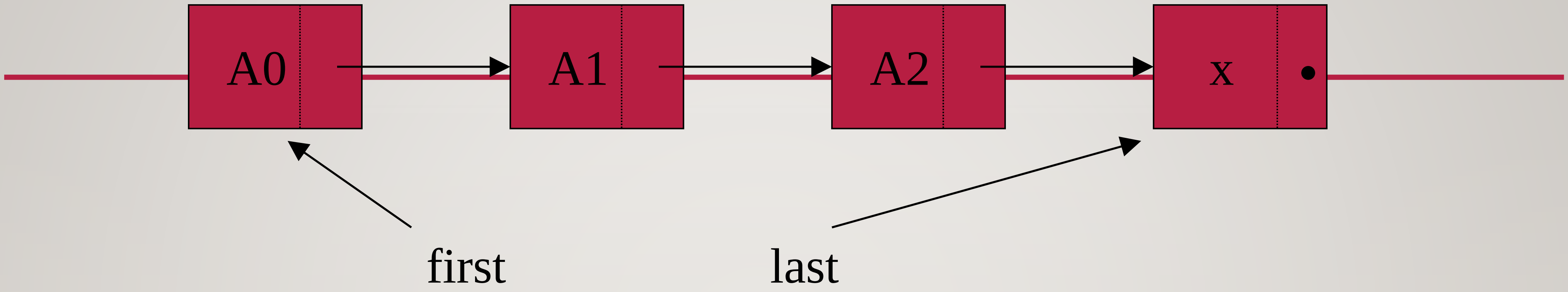



At any point, we can add a new last item **x** by doing this:

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

Steps

1. Locate index'th element
2. Allocate memory for the new node, **copy data into node**
3. Point the new node to its successor (following node)
4. Point the new node's predecessor (preceding node) to the new node



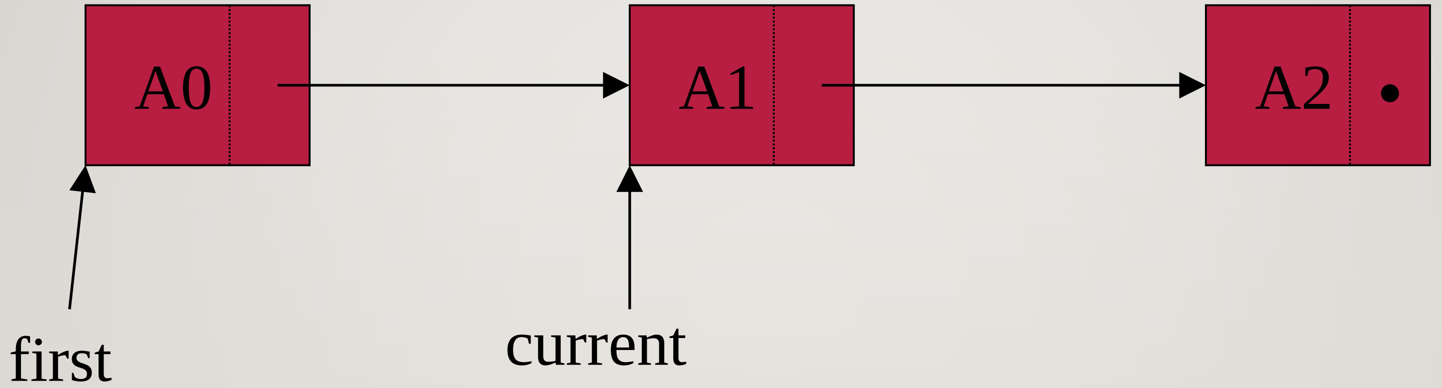

At any point, we can add a new last item **x** by doing this:

```
last->next = new ListNode
last = last->next;
last->data = x;
last->next = null;
```

Steps

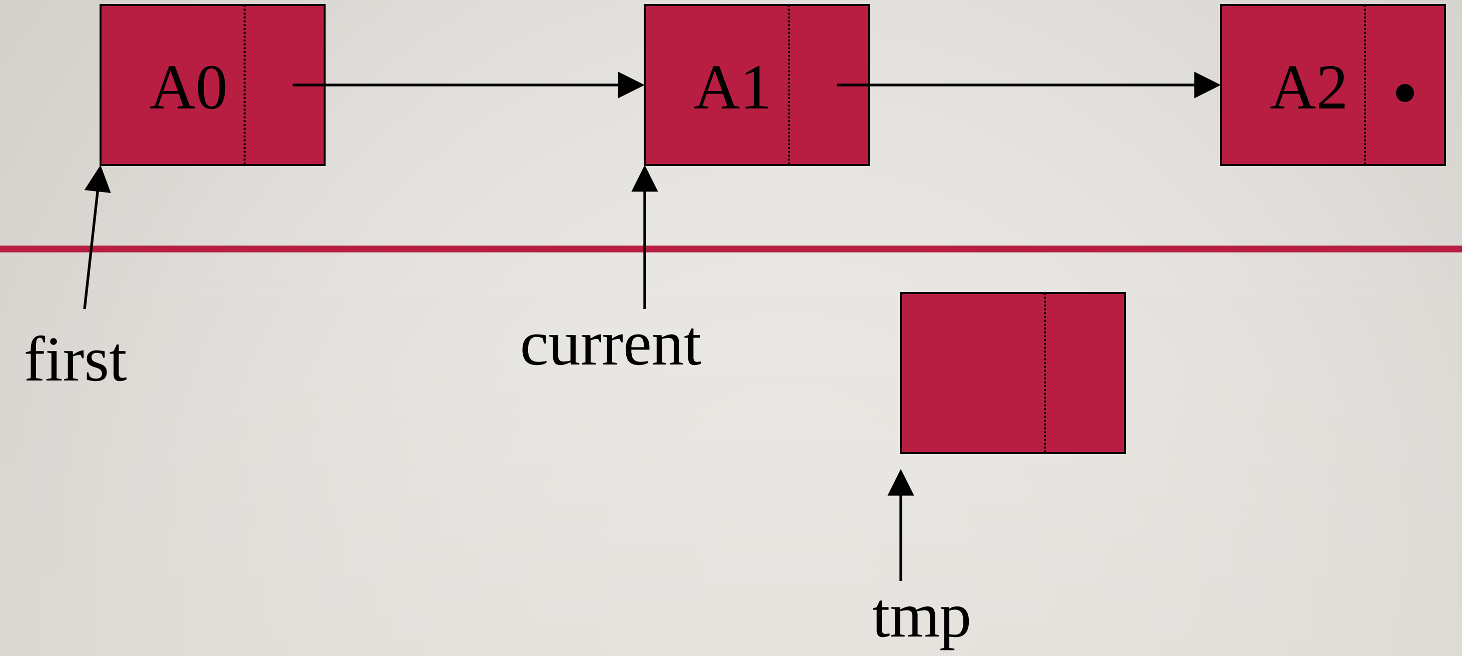
1. Locate index'th element
2. Allocate memory for the new node, copy data into node
3. **Point the new node to its successor (following node)**
4. Point the new node's predecessor (preceding node) to the new node

# INSERTION *AT THE MIDDLE*




At any point, we can add a new item **x** by doing this (after locating the required index using “current” pointer):

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

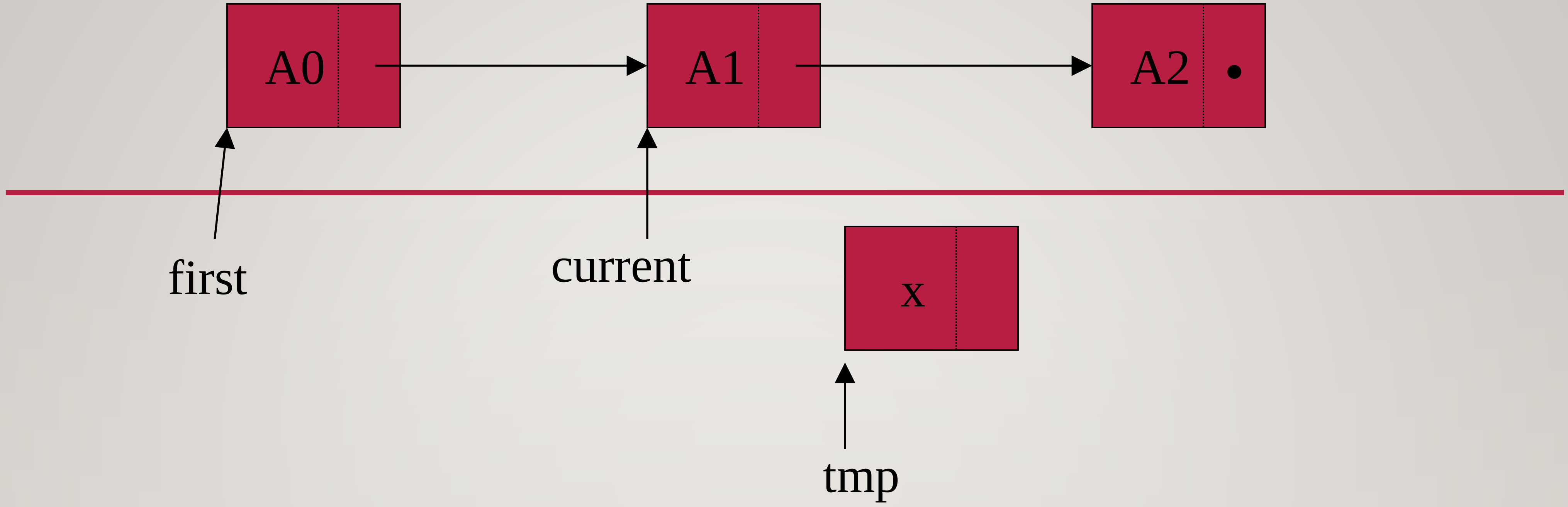



At any point, we can add a new item **x** by doing this:

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

Steps

1. Locate index'th element
2. **Allocate memory for the new node**, copy data into node
3. Point the new node to its successor (following node)
4. Point the new node's predecessor (preceding node) to the new node

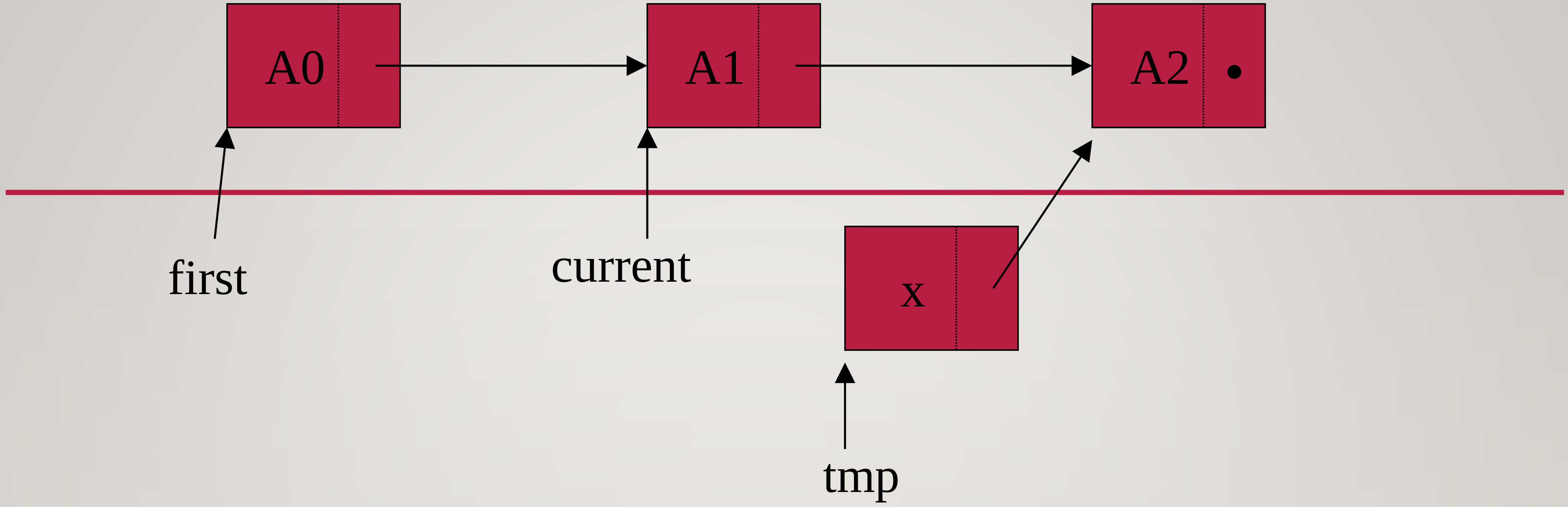



At any point, we can add a new item **x** by doing this:

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

Steps

1. Locate index'th element
2. Allocate memory for the new node, **copy data into node**
3. Point the new node to its successor (following node)
4. Point the new node's predecessor (preceding node) to the new node

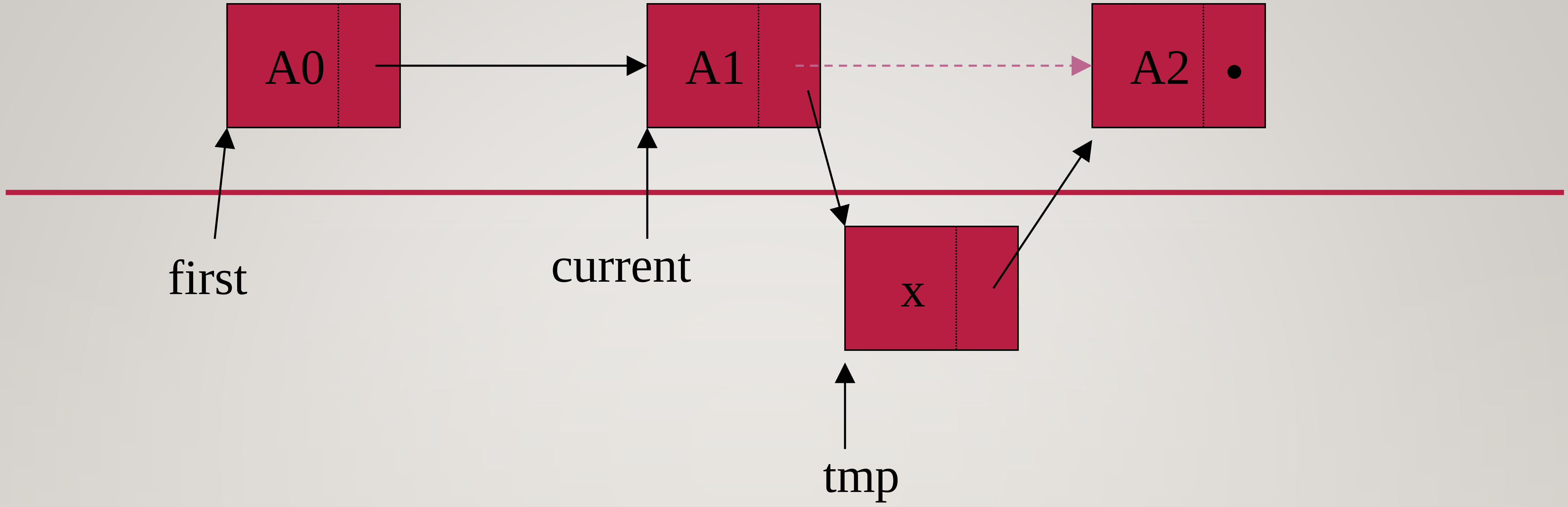



At any point, we can add a new item **x** by doing this:

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

Steps

1. Locate index'th element
2. Allocate memory for the new node, copy data into node
3. **Point the new node to its successor (following node)**
4. Point the new node's predecessor (preceding node) to the new node




At any point, we can add a new last item **x** by doing this:

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

Steps

1. Locate index'th element
2. Allocate memory for the new node, copy data into node
3. Point the new node to its successor (following node)
4. **Point the new node's predecessor (preceding node) 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)

```
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;
}
```

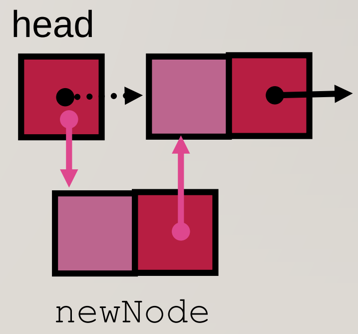
Try to locate index'th node. If it doesn't exist, return NULL.

```
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;
}
```

Create a new node

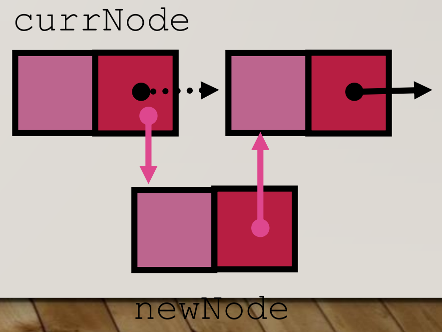
```
Node* List::InsertNode(int index, double x) {
	if (index < 0) return NULL;

	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



```
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 after `currNode`

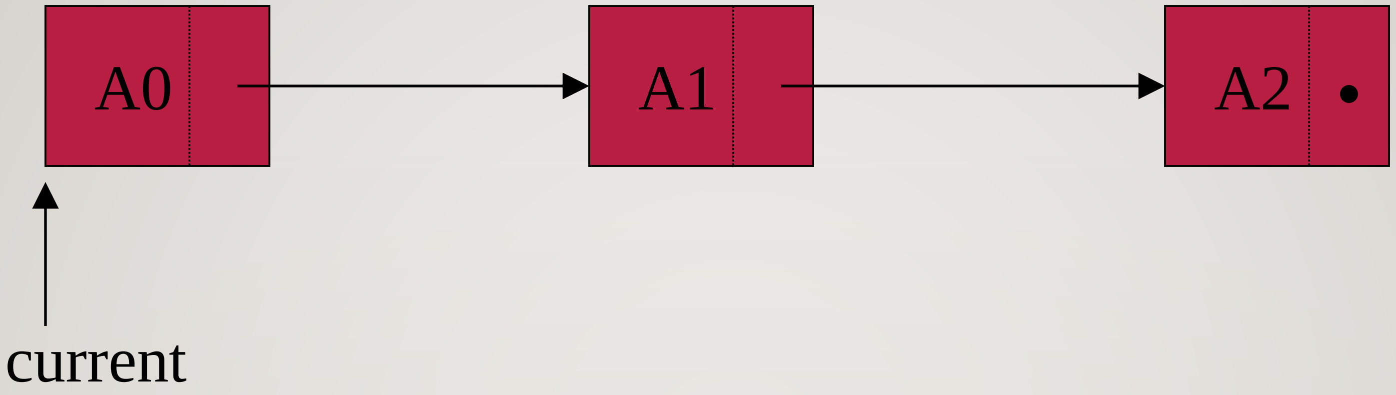


# FINDING A NODE

- `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.

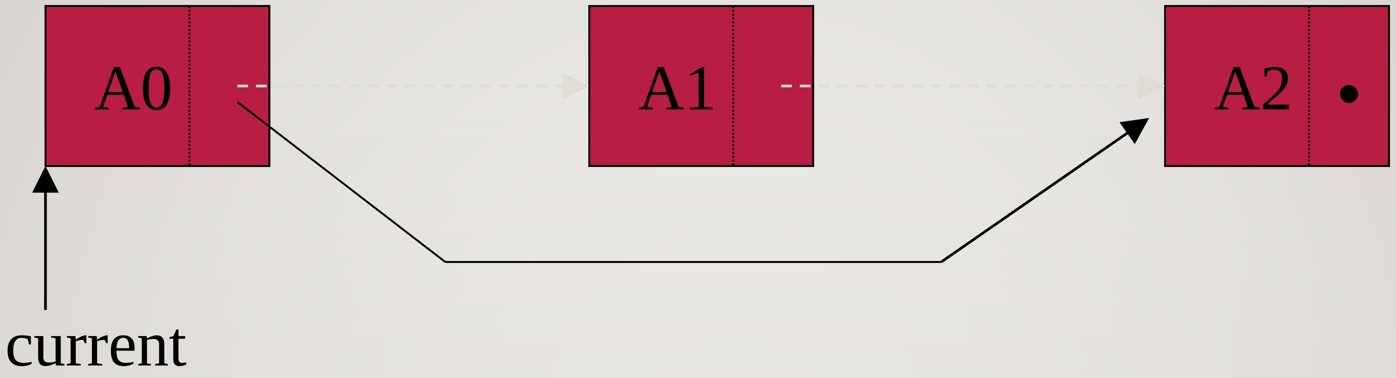
```
int List::FindNode(double x) {
	Node* currNode	=	head;
	int currIndex	=	1;
	while (currNode && currNode->data != x) {
		currNode	=	currNode->next;
		currIndex++;
	}
	if (currNode) return currIndex;
	return 0;
}
```

# DELETING A NODE




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

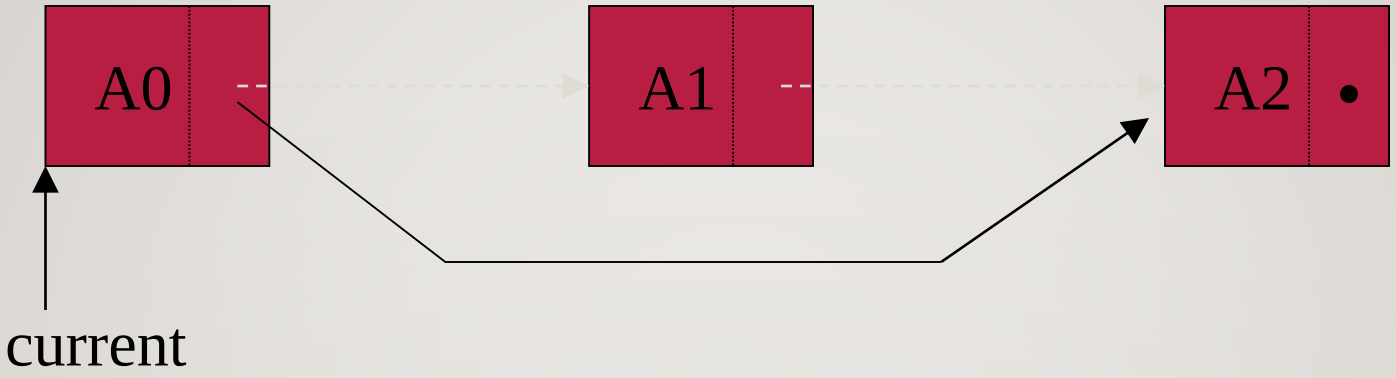
DELETING A NODE




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

**Memory leak!**

# DELETING A NODE




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

# 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

.

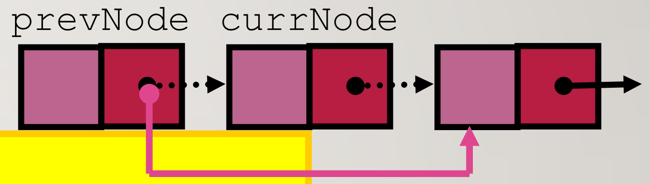
- Steps
  - Find the desirable node (similar to `FindNode`)
  - Set the pointer of the predecessor of the found node to the successor of the found node
  - Release the memory occupied by the found node

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

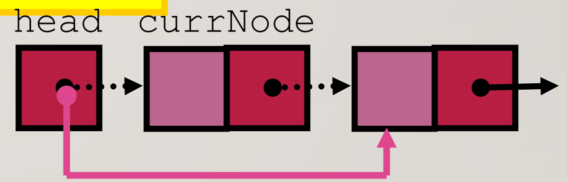
```
int List::DeleteNode(double x) {
    Node* prevNode =    NULL;
    Node* currNode =    head;
    int currIndex  =    1;
    while (currNode && currNode->data != x) {
        prevNode   =    currNode;
        currNode   =    currNode->next;
        currIndex++;
    }
    if (currNode) {
        if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

```
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;
}
```



```
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
```

# RECURSIVE LINKED LIST OPERATIONS

- Recursion may be used in some operations on linked lists.
- We will look at functions that:
  - Count the number of nodes in a list, and
  - Display the value of the list nodes in reverse order.

# COUNTING THE NODES IN THE LIST

```
int List::countNodes(Node *nodePtr)
{
    if (nodePtr != NULL)
        return 1 + countNodes(nodePtr->next);
    else
        return 0;
}
```

The base case for the function is `nodePtr` being equal to NULL.

# COUNTING THE NODES IN THE LIST

The function's recursive logic can be expressed as:

*If the current node has a value*
*Return 1 + the number of the remaining nodes.*
*Else*
*Return 0.*
*end If.*

# DISPLAYING THE LIST NODES IN REVERSE ORDER

```
void List::showReverse(Node *nodePtr)
{
    if (nodePtr != NULL)
    {
        showReverse(nodePtr->next);
        cout << nodePtr->value << " ";
    }
}
```

The base case for the function is `nodePtr` being equal to NULL.

- Some mathematical problems are designed to be solved recursively. One example is the calculation of *Fibonacci numbers,* which are the following sequence:

**0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, …**

# SOLVING RECURSIVELY DEFINED PROBLEMS

- The Fibonacci series can be defined as:

$F_0 = 0,$

$F_1 = 1,$

$F_N = F_{N-1} + F_{N-2}$ for $N \geq 2$.

# SOLVING RECURSIVELY DEFINED PROBLEMS

- A recursive C++ function to calculate the nth number in the Fibonacci series is shown below .

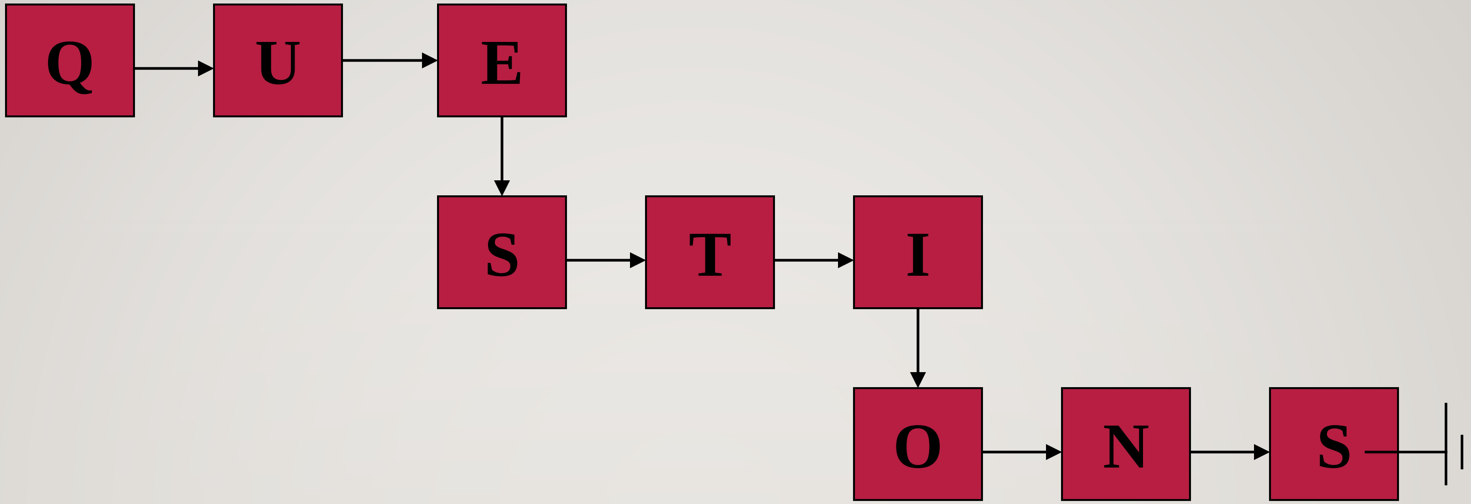
```
int fib(int n)
{
    if (n <= 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}
```

```
// This programs demonstrates a recursive function
// that calculates Fibonacci numbers.

#include <iostream.h>

// Function prototype
int fib(int);

void main(void)
{
    cout << "The first 10 Fibonacci numbers are:\n";
    for (int x = 0; x < 10; x++)
        cout << fib(x) << " ";
    cout << endl;
}
```

```
//*******************************************
// Function fib. Accepts an int argument  *
// in n. This function returns the nth    *
// Fibonacci number.                      *
//*******************************************

int fib(int n)
{
    if (n <= 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fib(n - 1) + fib(n - 2);
}
```

# Program Output

```
The first 10 Fibonacci numbers are:
0 1 1 2 3 5 8 13 21 34
```


Q
U
E
S
T
I
O
N
S