

# Data Structures

---

## **8. Linked Lists**

# Roadmap

---

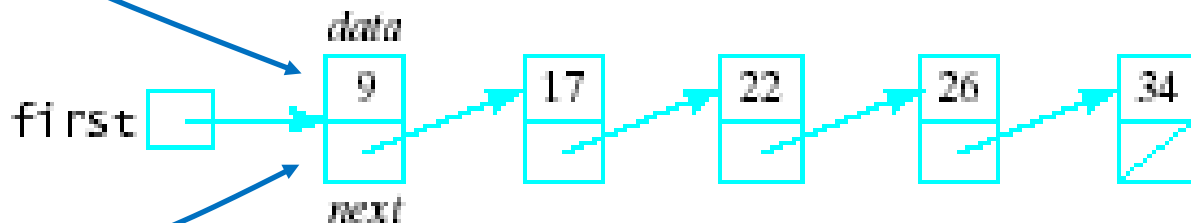
- Introduction to linked lists
- A pointer-based implementation in C++
- Variations of linked lists

---

# Linked List Using Pointers-Based Implementation of Lists

# Linked List

- Linked list nodes composed of two parts
  - Data part
    - Stores an element of the list
  - Next part
    - Stores link/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  
};
```

# 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; }
        Node* InsertNode(int index, double x);
        int FindNode(double x);
        int DeleteNode(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
- `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

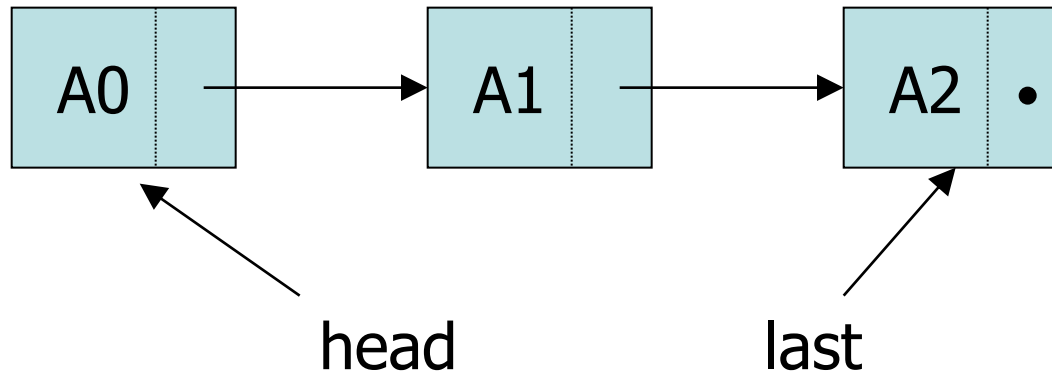
---

- **Node\* InsertNode(int index, double x)**
  - Insert a node with data equal to x after the index elements
  - If the insertion is successful
    - Return the inserted node
    - Otherwise, return NULL
  - If index is  $< 0$  the insertion will fail
- **Steps**
  1. Locate the element at the index position
  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



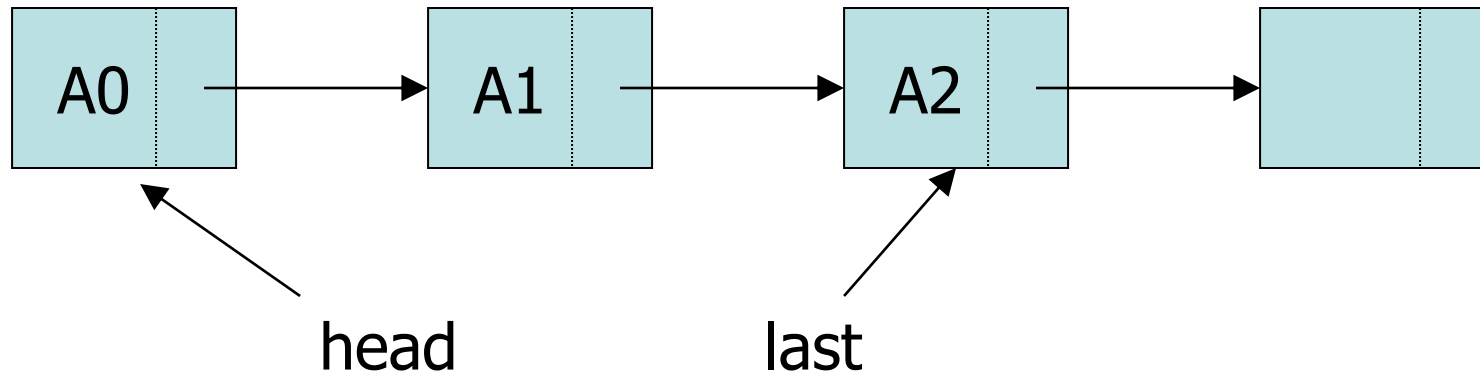
```
last->next = new Node();  
last = last->next;  
last->data = x;  
last->next = null;
```

## 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



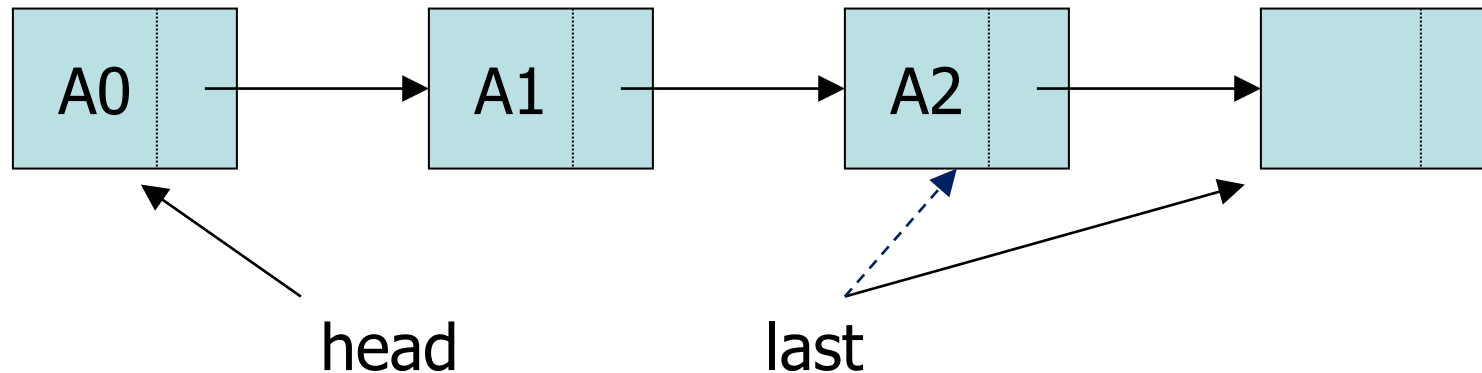
```
last->next = new Node();  
last = last->next;  
last->data = x;  
last->next = null;
```

## 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



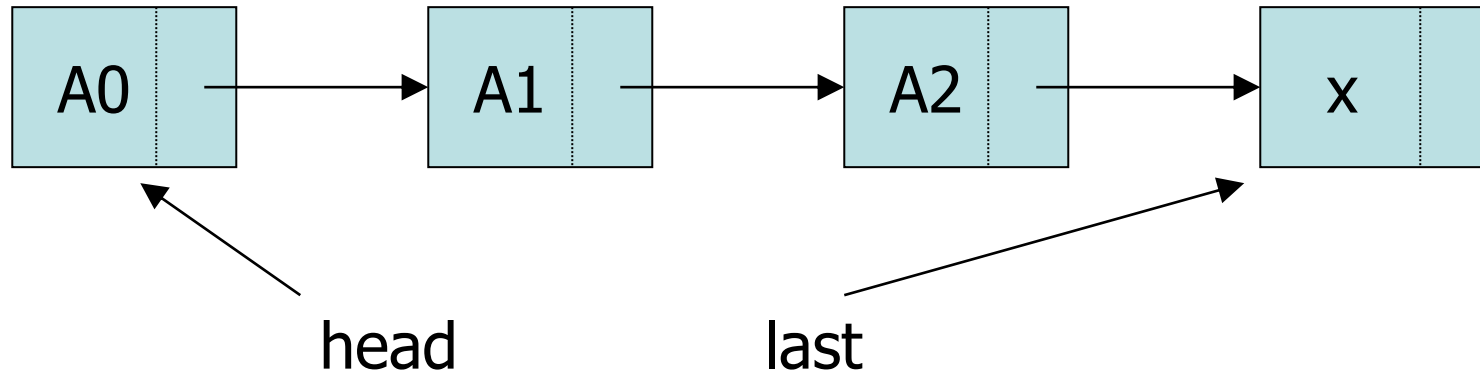
```
last->next = new Node();  
last = last->next;  
last->data = x;  
last->next = null;
```

## 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



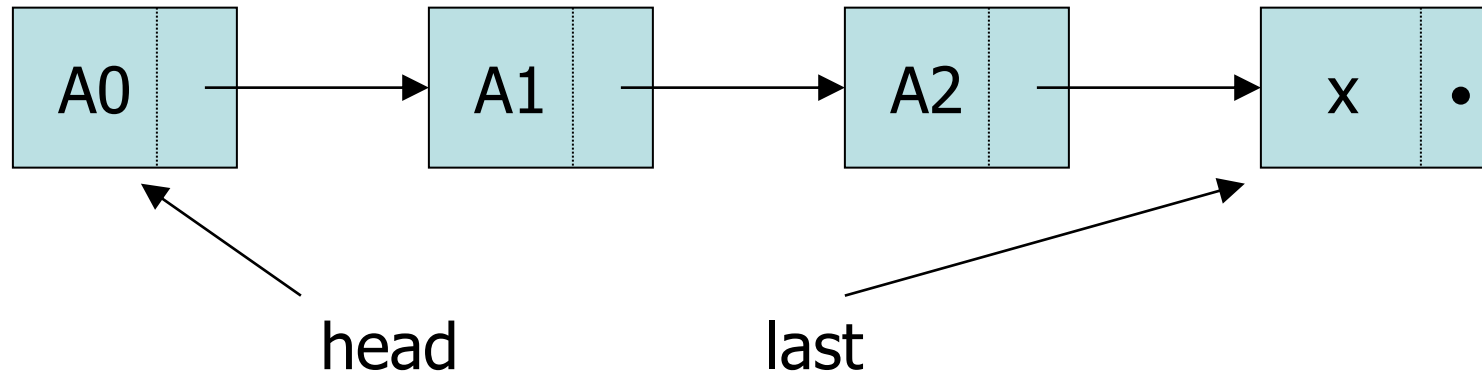
```
last->next = new Node();  
last = last->next;  
last->data = x;  
last->next = null;
```

## 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;
```

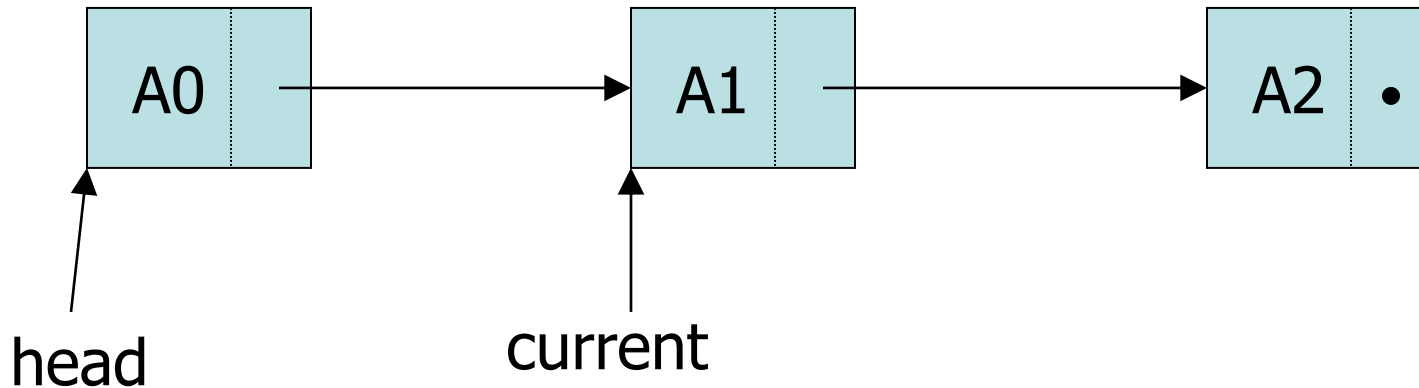
## Steps

- 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)

---

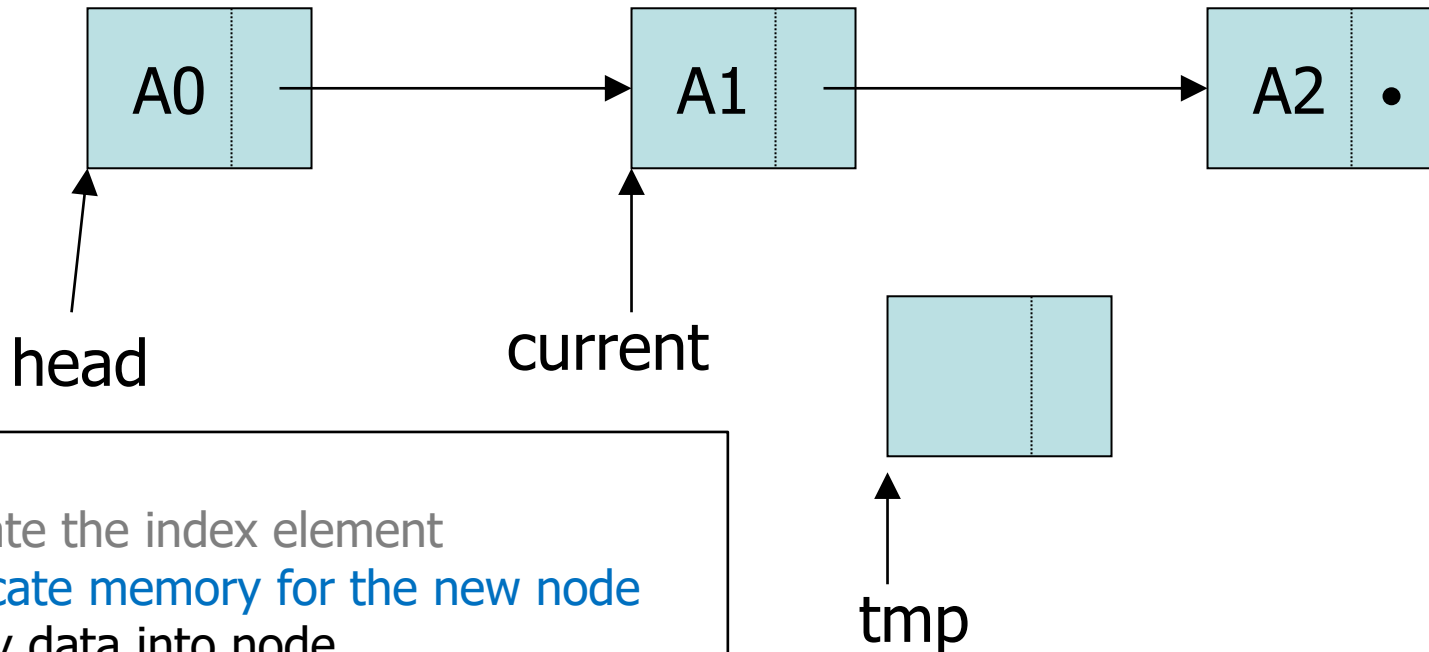
- Suppose **current** points to the middle element of the list
  - We can add a new last item x by doing this



```
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 last item `x` by doing this



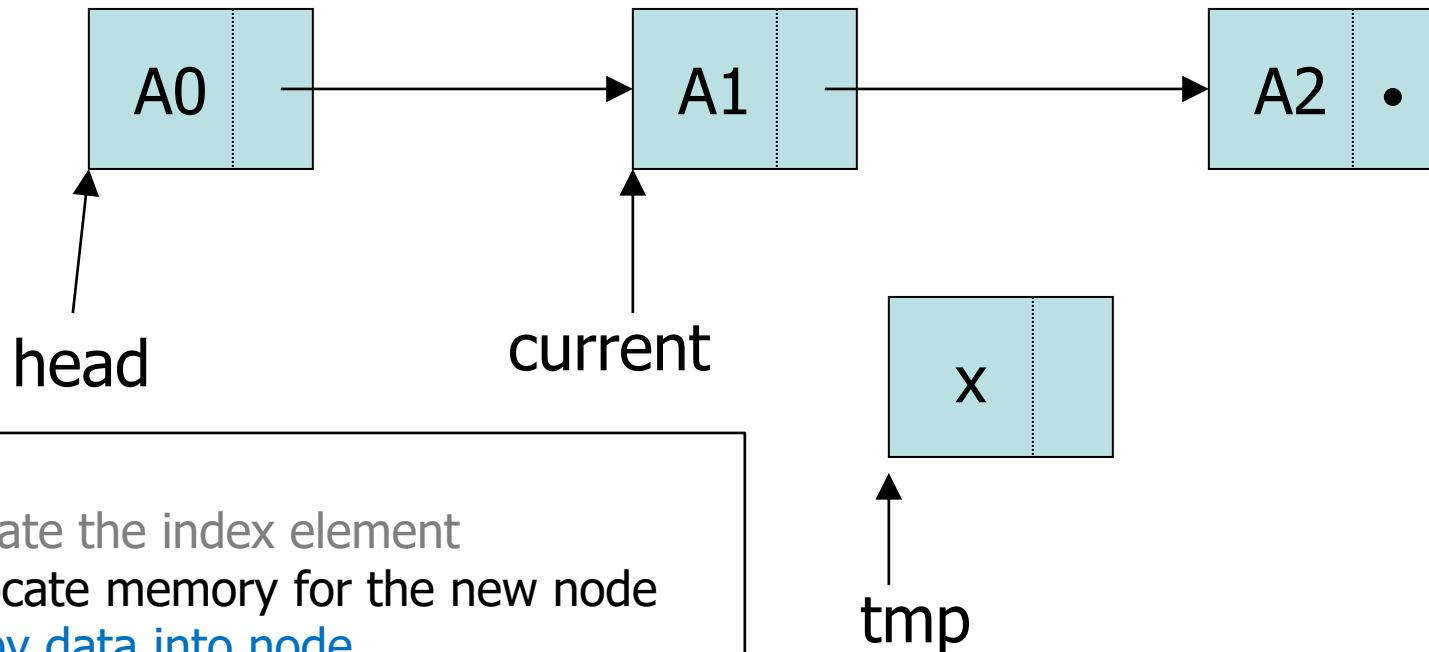
## Steps

- 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 last item `x` by doing this



## Steps

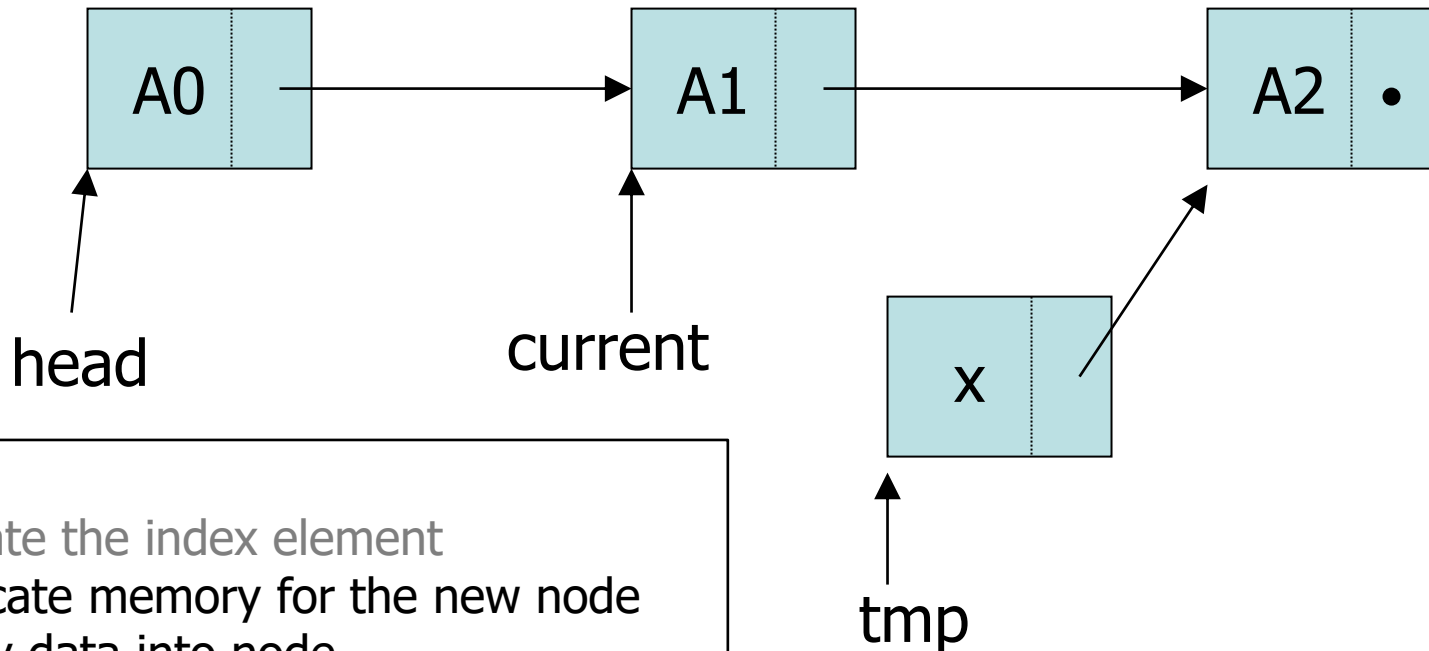
- Locate the index element
- Allocate memory for the new node
- Copy data into node
- Point the new node to its successor (next node)
- Point the new node's predecessor (preceding node) to the new node

```
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 last item `x` by doing this



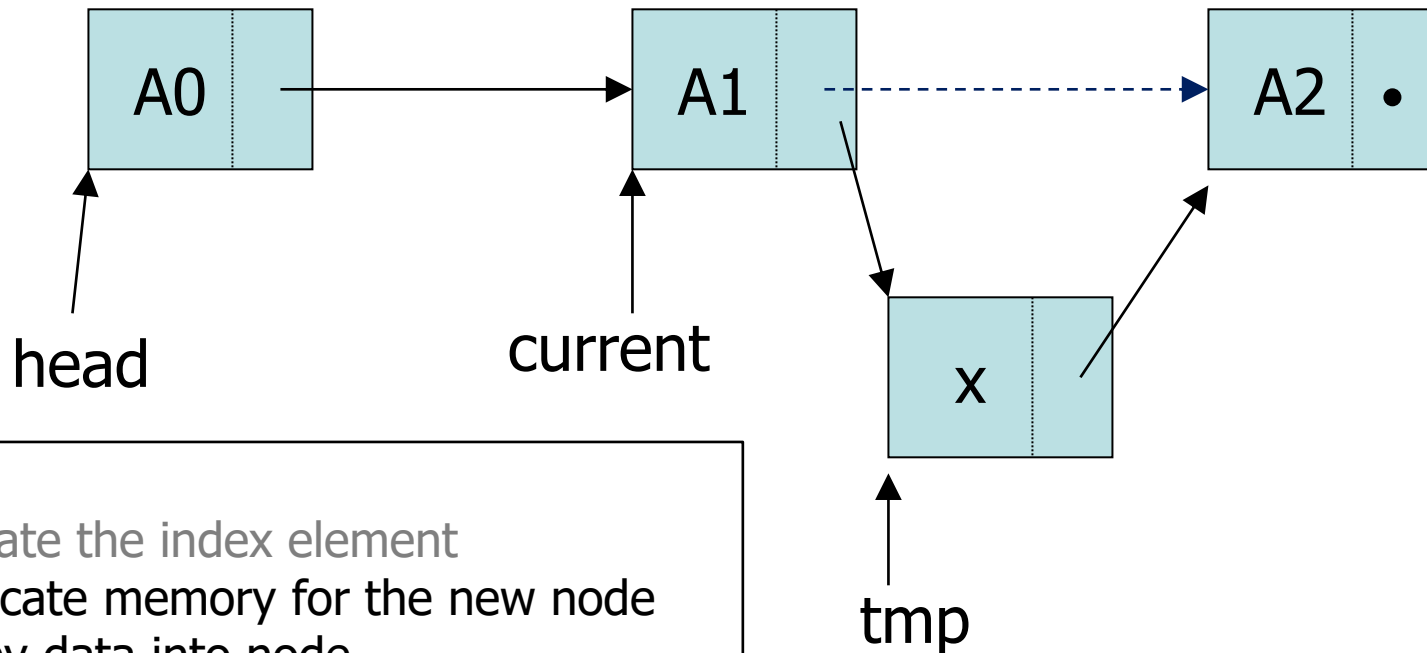
## Steps

- 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 last 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 `InsertNode`
  1. Insert into an empty list
  2. Insert in 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)

---

```
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;
```

Try to locate index'th node.  
If it doesn't exist, 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 (3)

```
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;
```

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

```
}
```

# Inserting a New Node (3)

```
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;
```

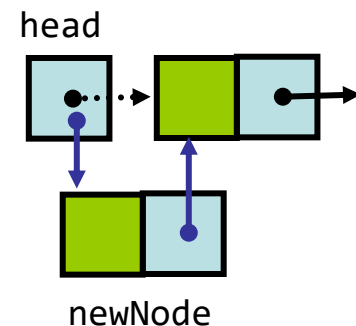
Try to locate index'th node.  
If it doesn't exist, return  
NULL

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

Create a new node

```
    else {  
        newNode->next = currNode->next;  
        currNode->next = newNode;  
    }  
    return newNode;
```

Insert as first element



# Inserting a New Node (3)

```
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;
```

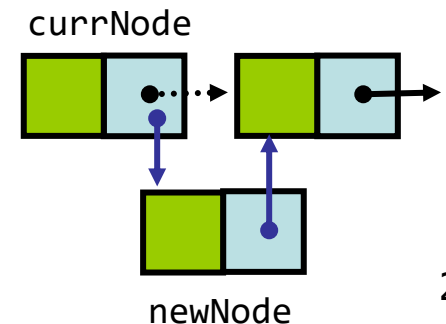
```
}
```

Link Lists

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

Create a new node

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

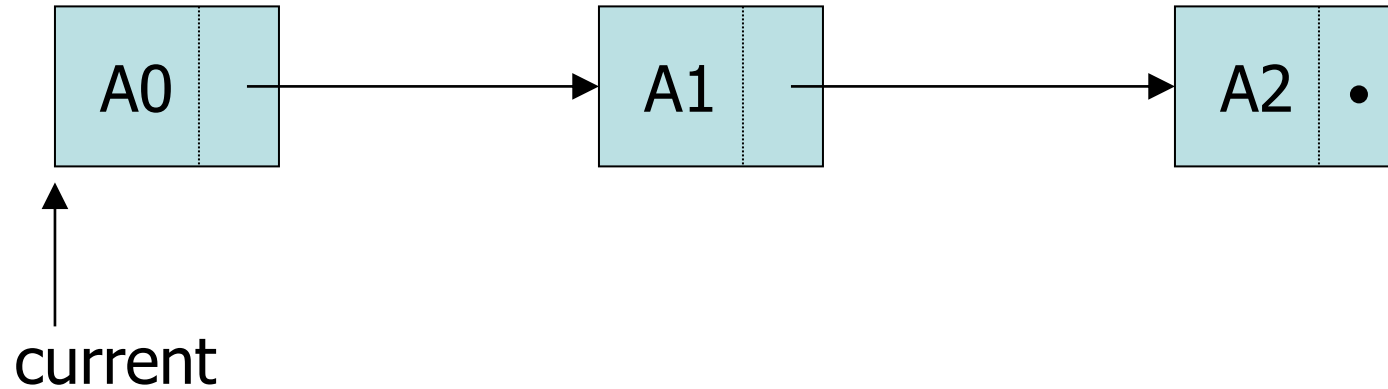
```
int List::FindNode(double x) {  
    Node* currNode = head;  
    int currIndex = 1;  
    while (currNode && currNode->data != x) {  
        currNode = currNode->next;  
        currIndex++;  
    }  
    if (currNode) return currIndex;  
  
    return 0;  
}
```



# Deleting a Node – Example (1)

---

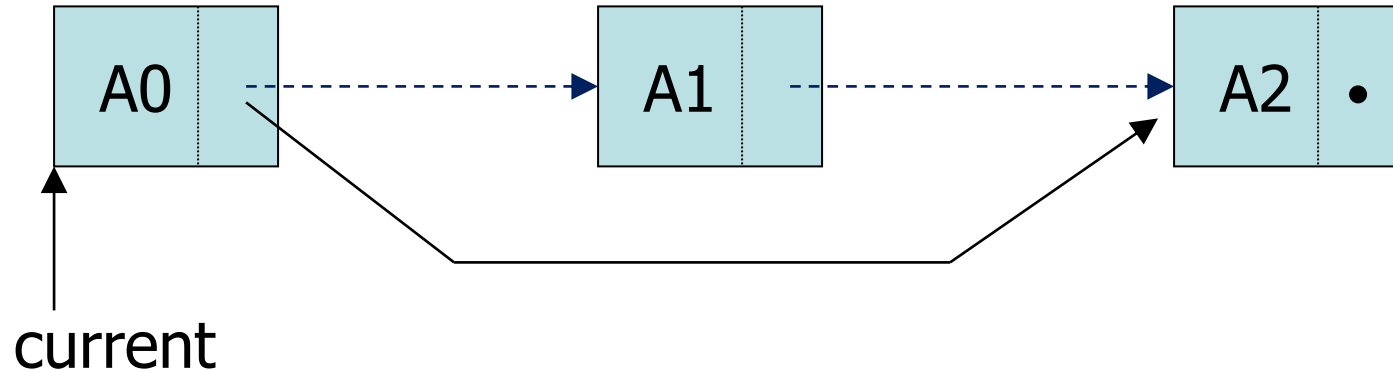
- Deleting item A1 from the list



## Deleting a Node – Example (2)

---

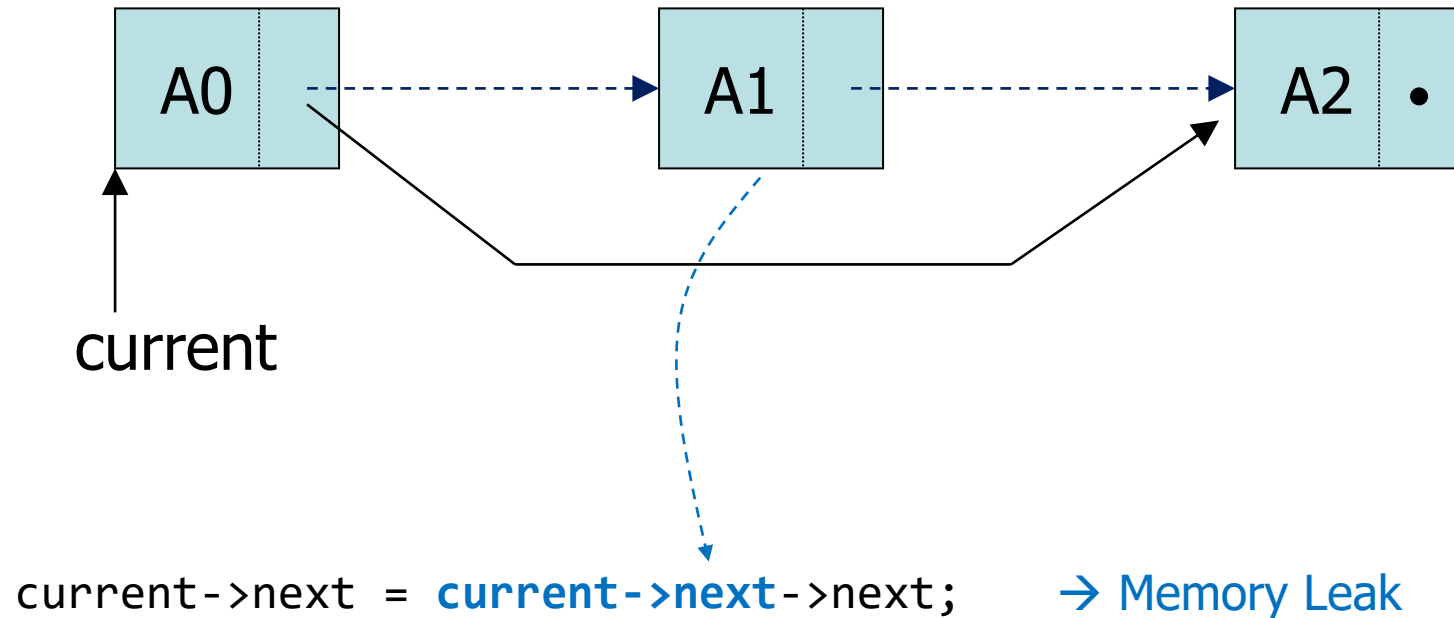
- Deleting item A1 from the list



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

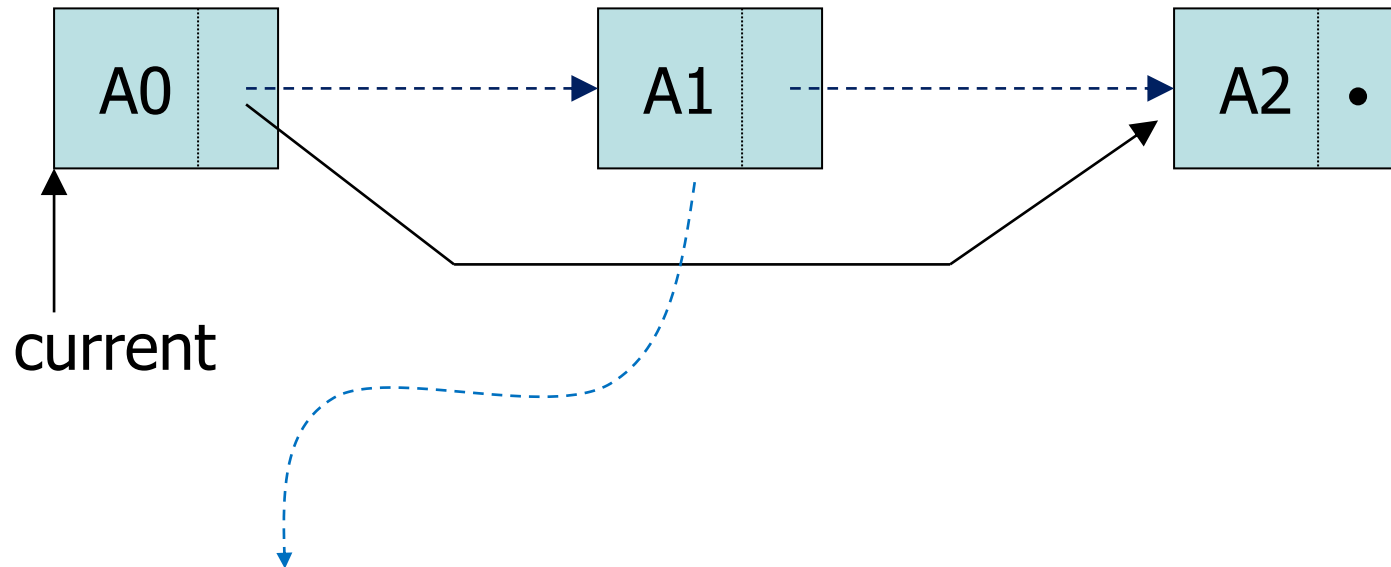
## Deleting a Node – Example (3)

- Deleting item A1 from the list



## Deleting a Node – Example (4)

- Deleting item A1 from the list



```
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)
  - 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 – Implementation (1)

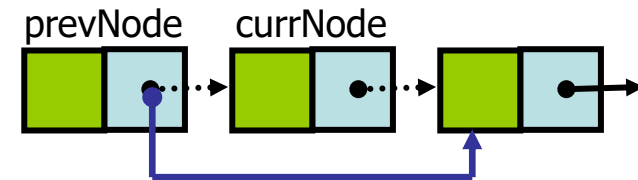
---

```
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 node with its value equal to x.

# Deleting a Node – Implementation (2)

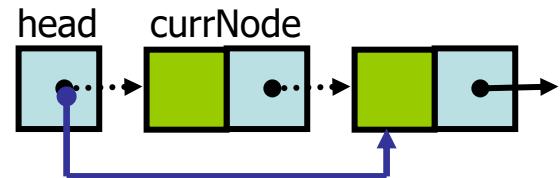
```
int List::DeleteNode(double x) {  
    Node* prevNode    = NULL;  
    Node* currNode     = head;  
    int currIndex      = 1;  
    while (currNode && currNode->data != x) {  
        prevNode      = currNode;  
        currNode       = currNode->next;  
        currIndex++;  
    }  
    if (currNode)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



# Deleting a Node – Implementation (3)

```
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;  
}
```

Link Lists





# 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;  
    }  
    head = NULL;  
}
```

# Using List (1)

```
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();

    return 0;
}
```

Output:

6

7

5

Number of nodes in the list: 3

## Using List (2)

```
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;

    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.InsertNode(0, 7.0);
    list.InsertNode(1, 5.0);
    list.InsertNode(-1, 5.0);
    list.InsertNode(0, 6.0);
    list.InsertNode(8, 4.0);
    // print all the elements
    list.DisplayList();
    if(list.FindNode(5.0) > 0)
    else
    if(list.FindNode(4.5) > 0)
    else
    list.DeleteNode(7.0);
    list.DisplayList();
    return 0;
}
```

```
// successful
// successful
// unsuccessful
// successful
// unsuccessful
```

```
cout << "5.0 found" << endl;
cout << "5.0 not found" << endl;
cout << "4.5 found" << endl;
cout << "4.5 not found" << endl;
```

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?

---

