Data Structure and Algorithms

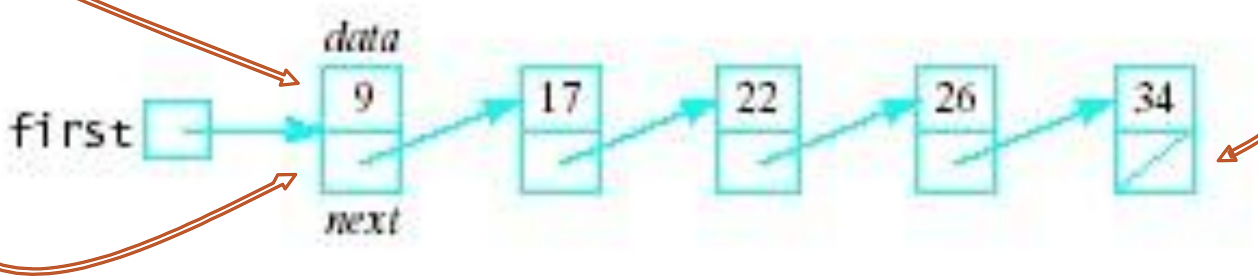
Affefah Qureshi

Department of Computer Science

Iqra University, Islamabad Campus.

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

- 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

- Declare List, which contains
  - head: a pointer to the first node in the list
  - Since the list is empty initially, headis 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

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

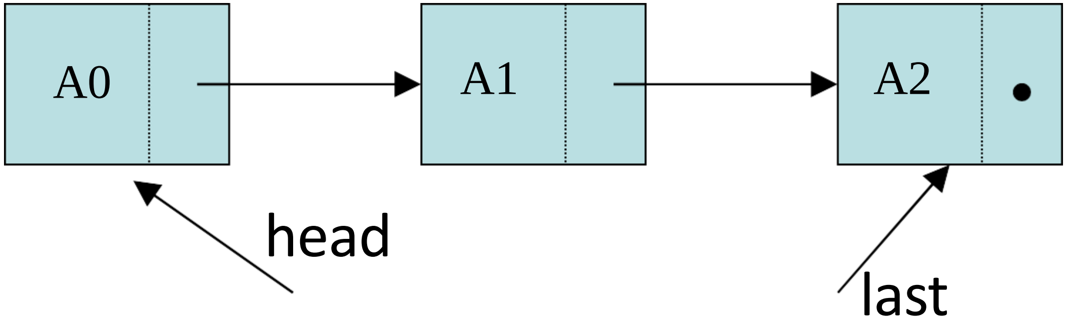
Inserting a New Node

- **`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 or >length of the list, 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

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

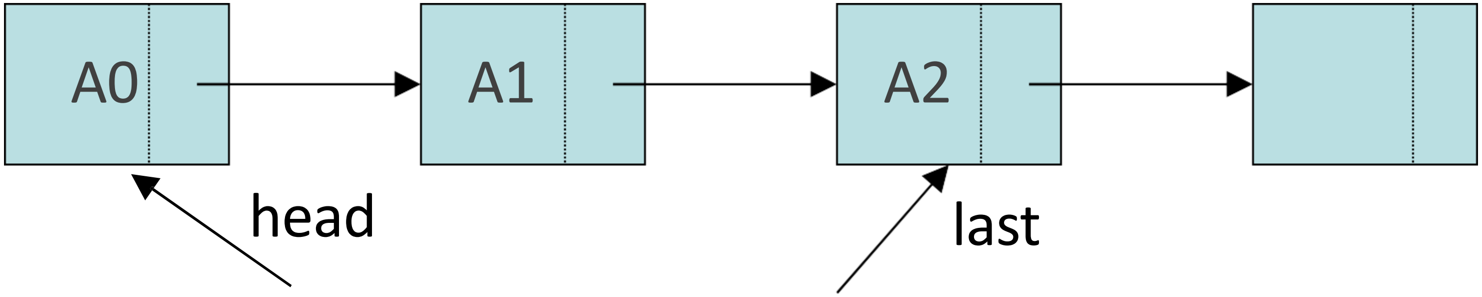


```
last->next = new Node();
last = last->next;
last->data = x;
last->next = null;
```

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



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

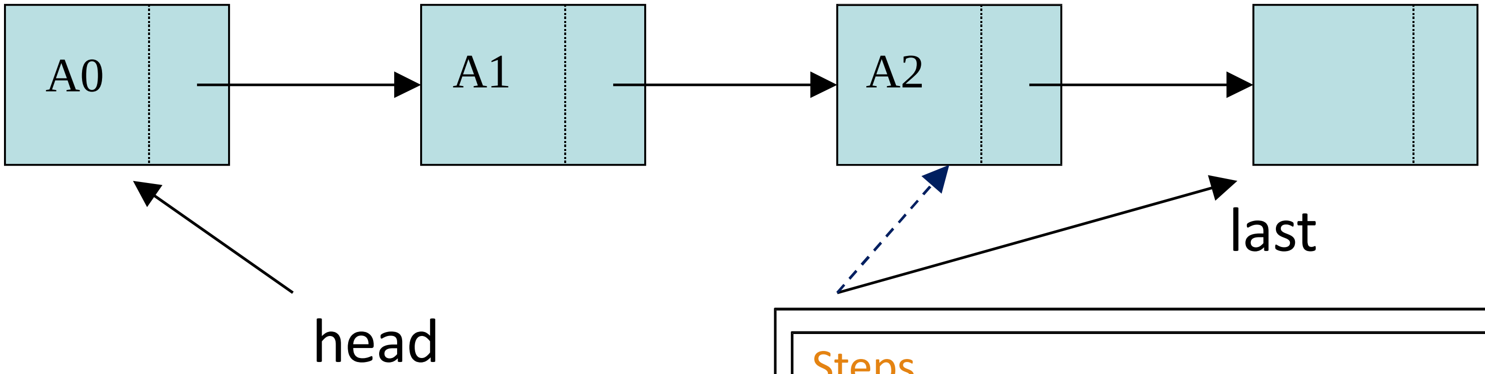
```
last->next = new Node();
last = last->next;
last->data = x;
last->next = null;
```

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

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



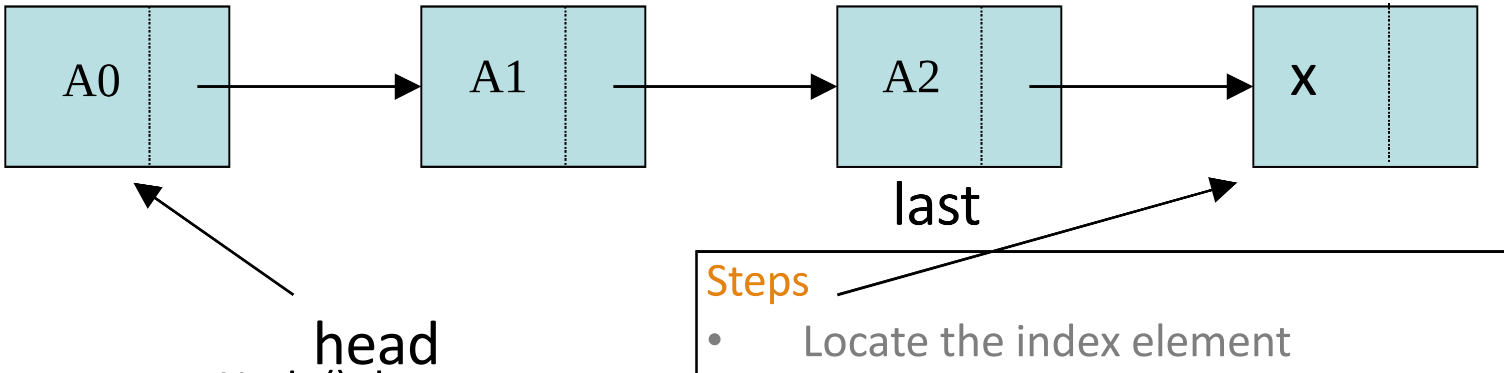
```
last->next = new Node();
last  = last->next;
last->data = x;
last->next = null;
```

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

- Suppose lastpoints 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;
```

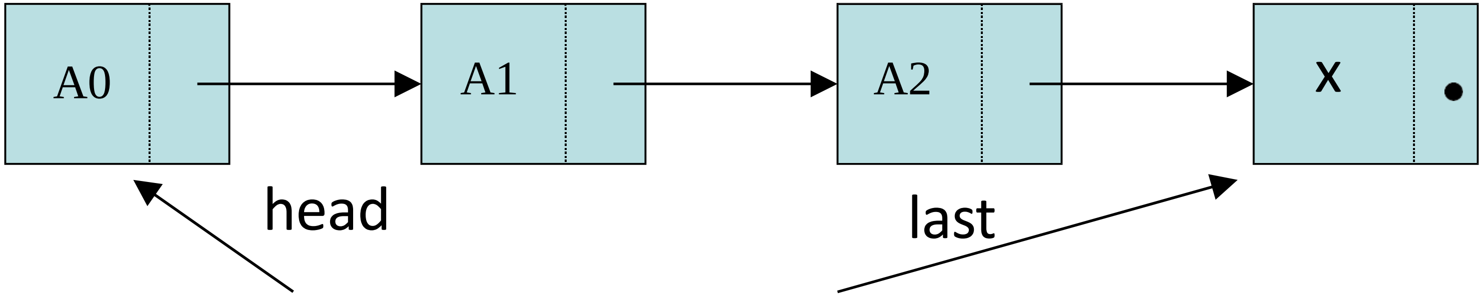
last->next = null;

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

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



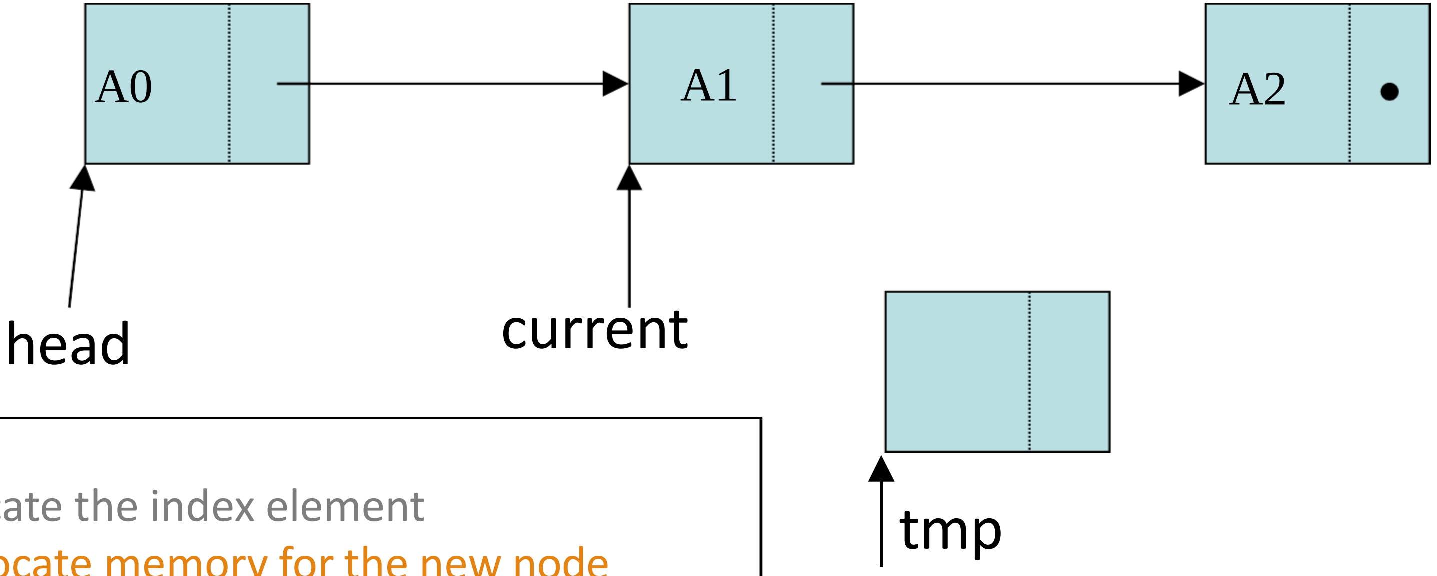
last->next = new Node(); last = last->next;
last->data = x;
last->next = null;

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

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



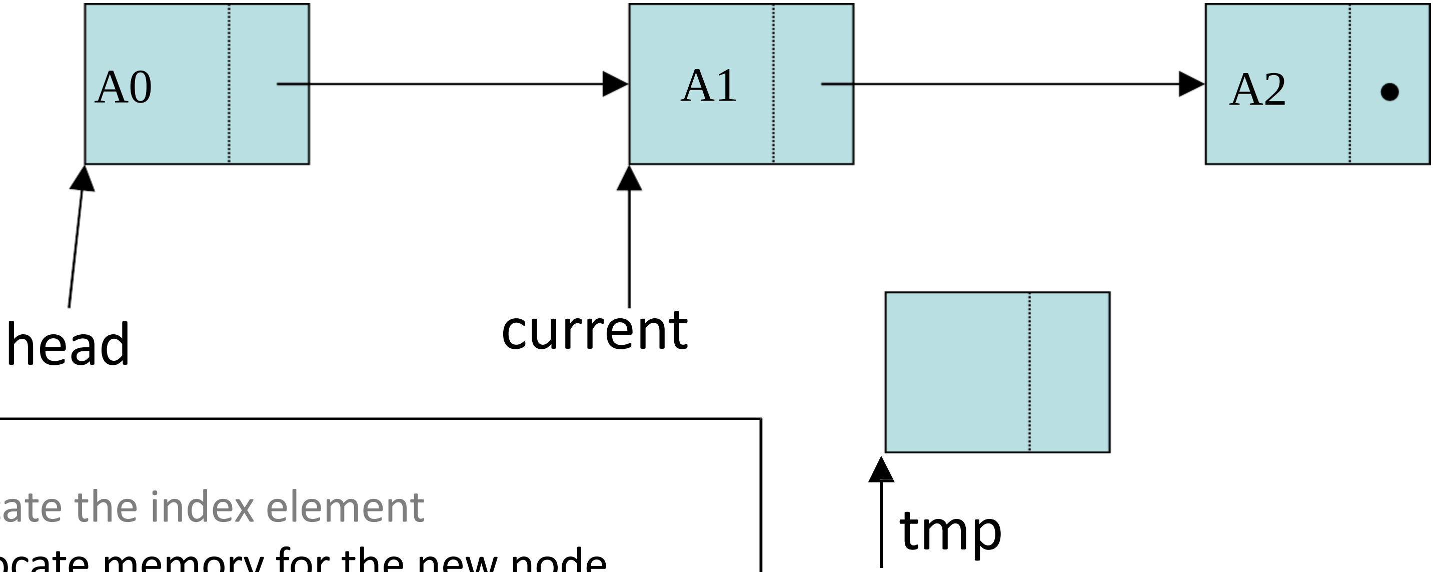
Steps

- Locate the index element
- Allocate memory for the new node
- Copy data into node its successor
- Point the new node to (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

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



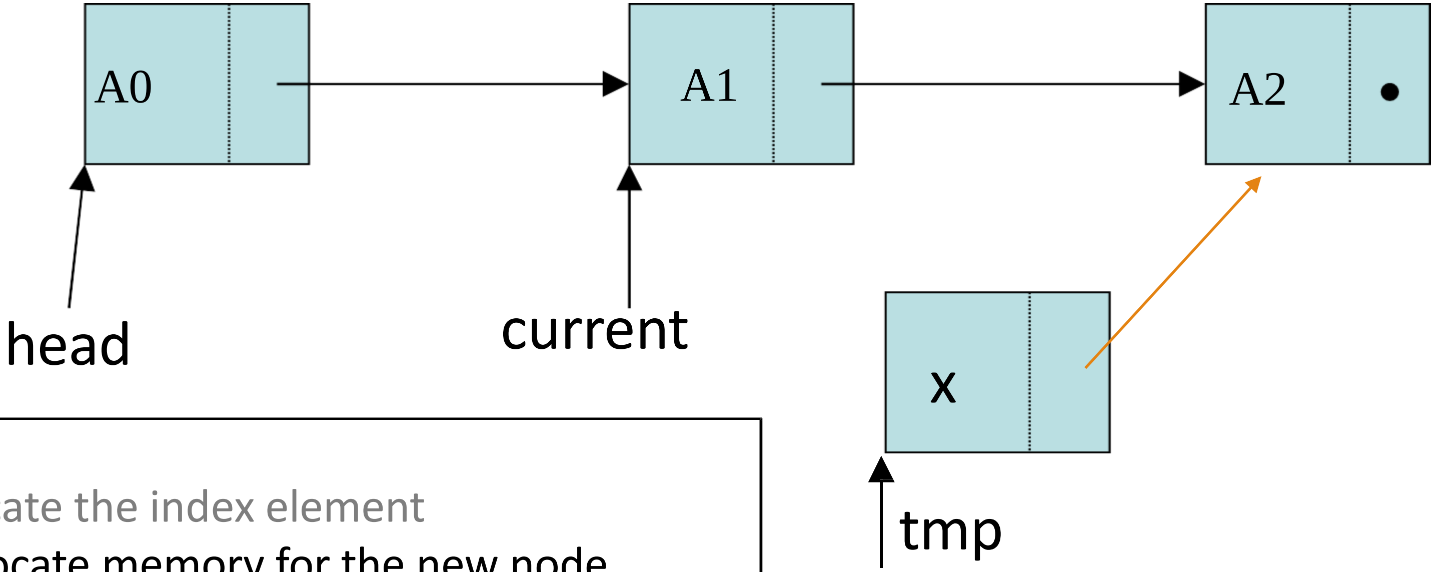
Steps

- Locate the index element
- Allocate memory for the new node
- Copy data into node its successor
- Point the new node to (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

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



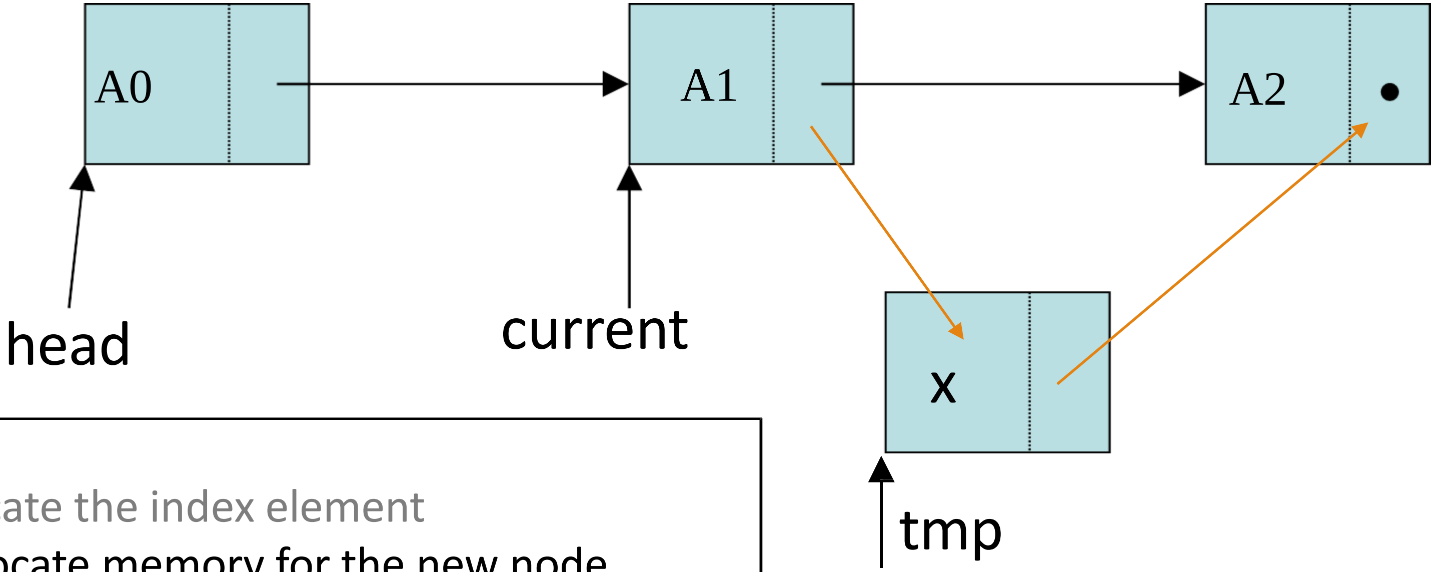
Steps

- Locate the index element
- Allocate memory for the new node
- Copy data into node its successor
- Point the new node to (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

- 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 its successor
- Point the new node to (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

- 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

```
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

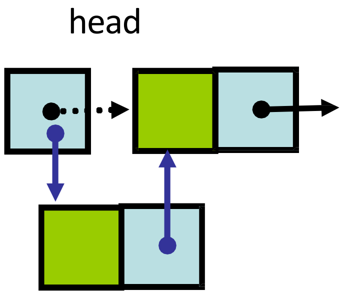
```
Node* List::InsertNode(int  index, double  x) {
        if  (index < 0)
                return  NULL;
        int currIndex =         1;
        Node* currNode =   head;
        while  (currNode && index > currIndex) {
                currNode       = currNode->next;
                currIndex++; }
        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; }
```

Create a new node

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; }
```

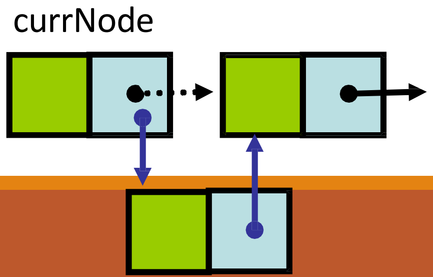
Insert as first element



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; }
```

Insert after currNode



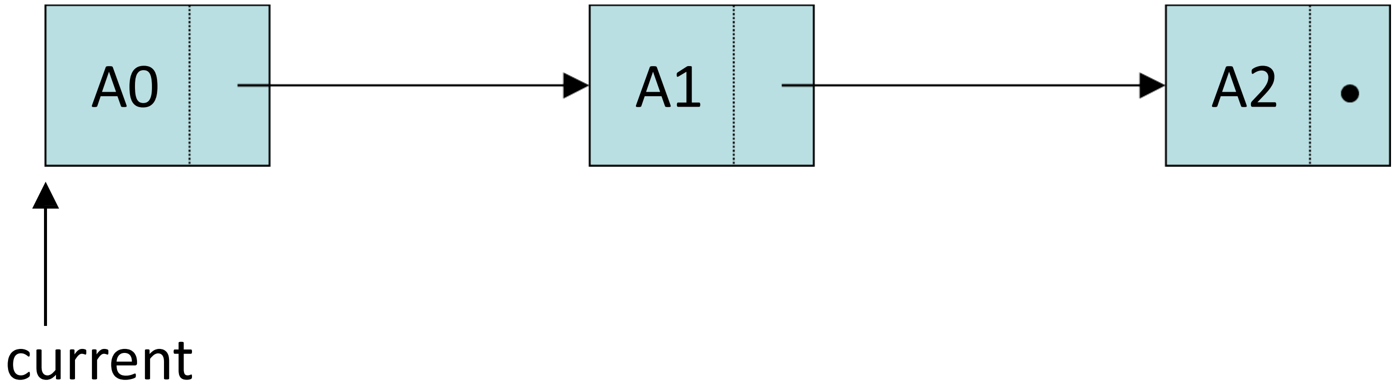
Finding a Node

- **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;
          else
          return  0; }
```

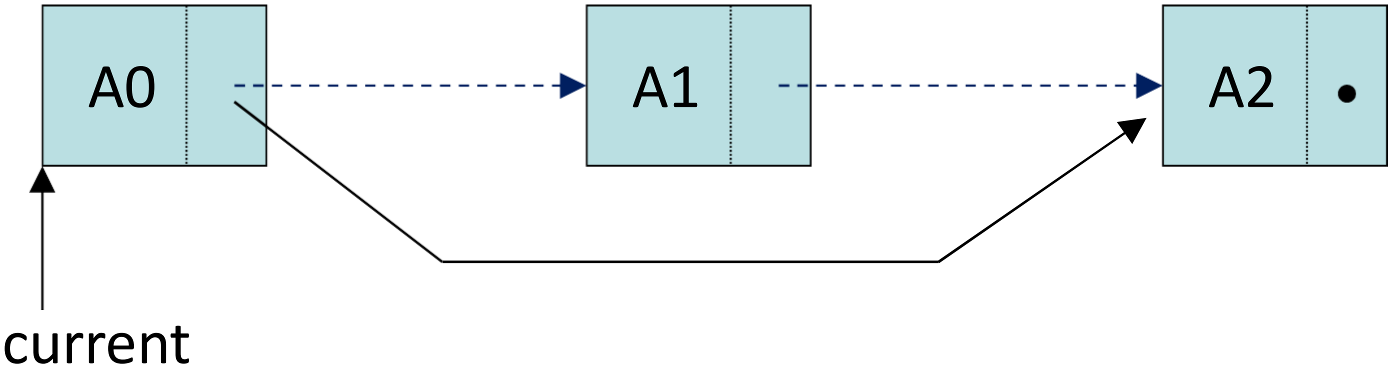
Deleting a Node – Example

- Deleting item A1 from the list



Deleting a Node – Example

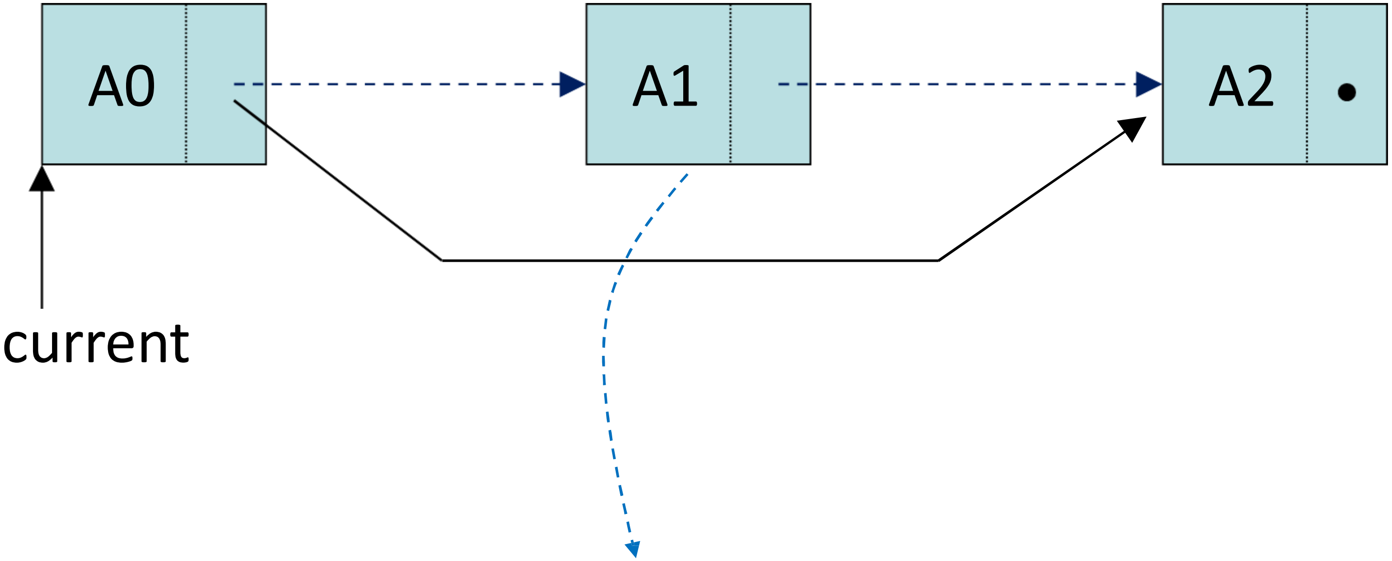
- Deleting item A1 from the list



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

Deleting a Node – Example

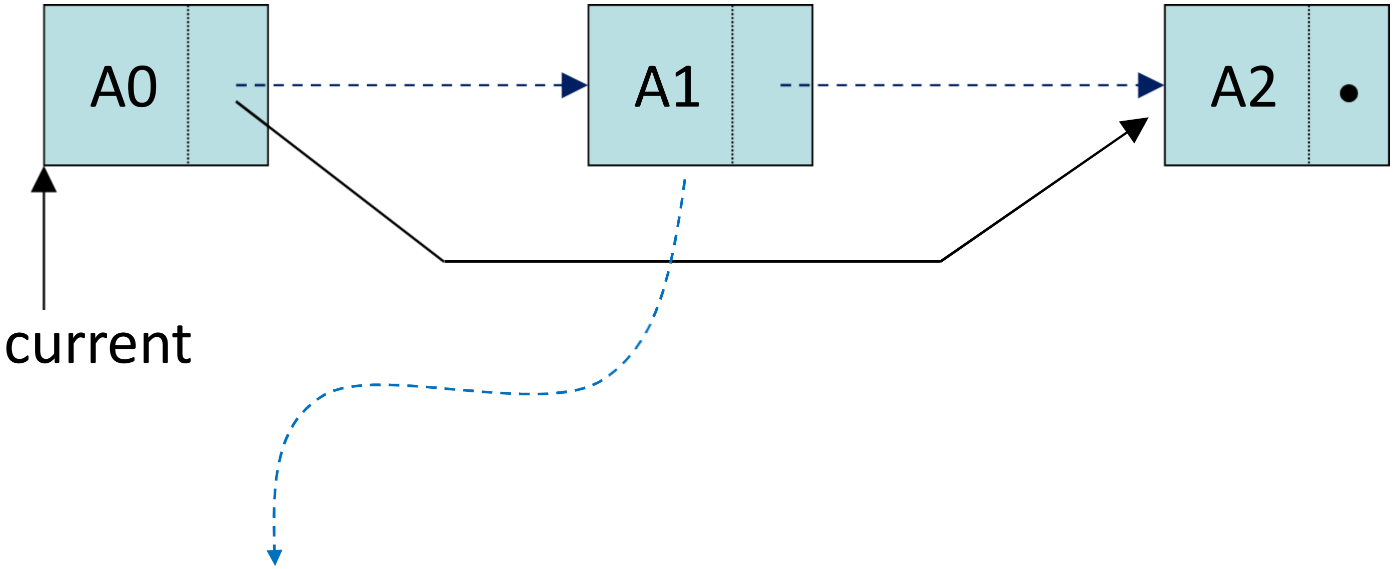
- Deleting item A1 from the list



current->next = **current->next**->next;

Deleting a Node – Example

- 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

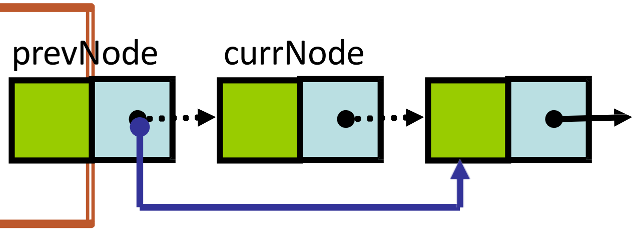
Deleting a Node – Implementation

```
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.

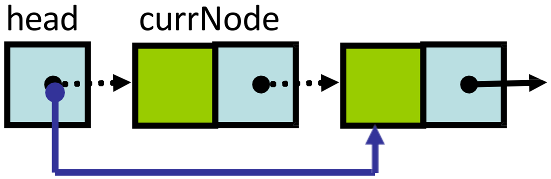
Deleting a Node – Implementation

```
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

```
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;
	Node* nextNode = NULL;
	while (currNode != NULL)
	{
		nextNode	=	currNode->next;
		delete currNode; // destroy the current node
		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
list.DisplayList(); // print all the elements


    return 0;
}
```

Output:

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

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
        list.DisplayList(); // print all the elements
        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); // 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
 list.DisplayList(); // print all the elements
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; }
```

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
```

To do !!

- Search the element in the list
- Find the length of the list
- Reverse each element of list 123 → 321
- Separate the even and odd element of list

Any Question So Far?