Data Structures
(Linked List)

Lecture#5

Agenda

- Review to the previous lecture
    - ➢Search (Array)
    - ➢Insertion (Array)
    - ➢Deletion (Array)
    - ➢Issues with Arrays
- 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

Array Operations

- **Insertion**
  - Operation of adding another element to an array
  - How many steps in terms of n (number of elements in array)?
    - At the end
    - In the middle
    - In the beginning
  - n steps at maximum (move items to insert at given location)
- **Deletion**
  - Operation of removing one of the elements from an array
  - How many steps in terms of n (number of elements in array)?
    - At the end
    - In the middle
    - In the beginning
  - n steps at maximum (move items back to take place of deleted item)

Array Operations: **Search Algorithms**

- Linear or Sequential Search
  - Best case: constant time
  - Worst case: $O(n)$
  - Easy to implement and understand
  - Does not require any pre-sorted array.
- Binary Search
  - Best case: constant time
  - Worst case: $O(\log(n))$
  - Relatively difficult to implement
  - requires sorted array

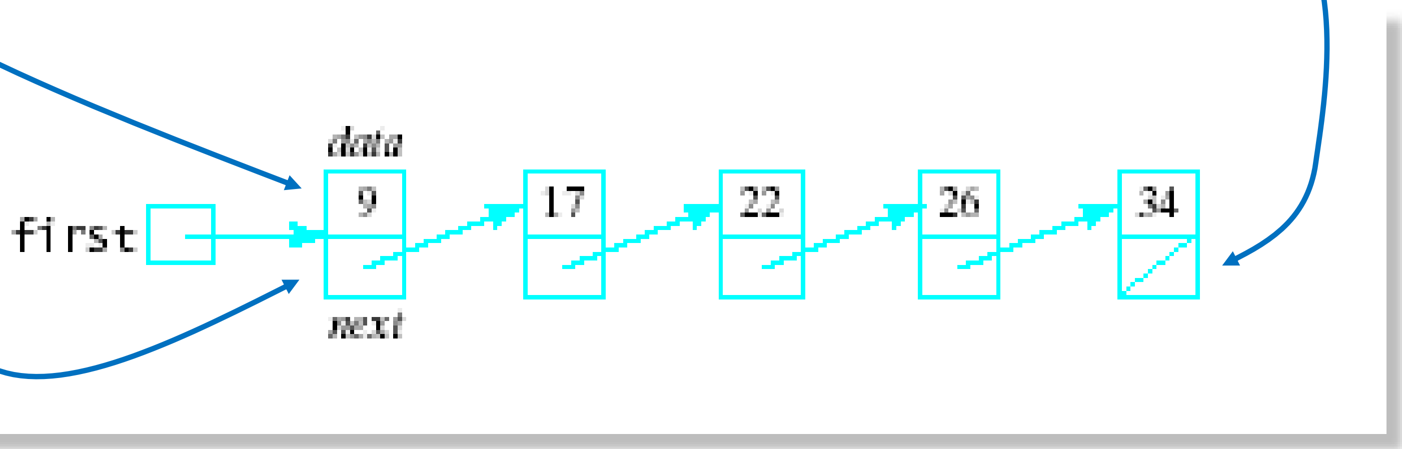
Limitation of Arrays

- An array has a limited number of elements
  - routines inserting a new value have to check that there is room
- Can partially solve this problem by **reallocating** the array as needed (how much memory to add?)
  - adding one element at a time could be costly
  - one approach - double the current size of the array
- A better approach: use a ***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

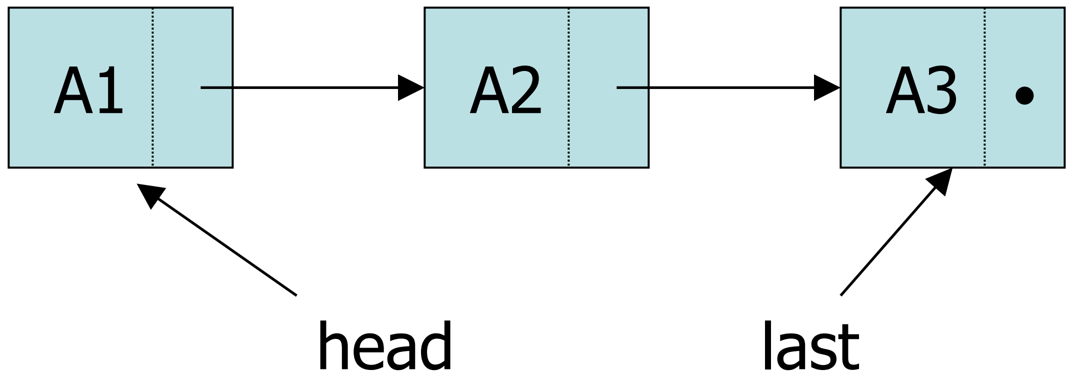
Inserting a New Node

- `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

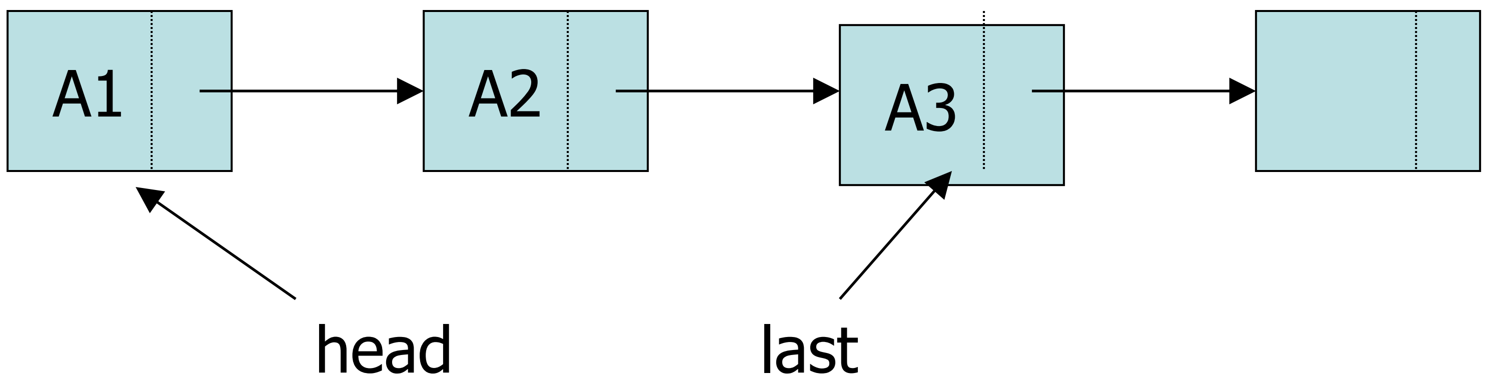



```
last->next = new Node();
last = last->next;
last->data = x;
last->next = null;
```

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

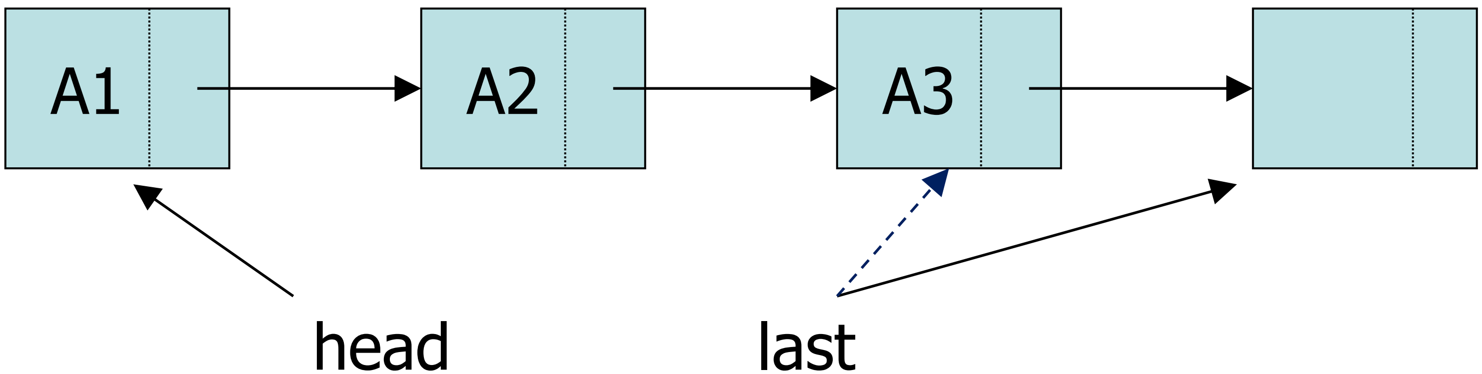



```
last->next = new Node();
last = last->next;
last->data = x;
last->next = null;
```

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



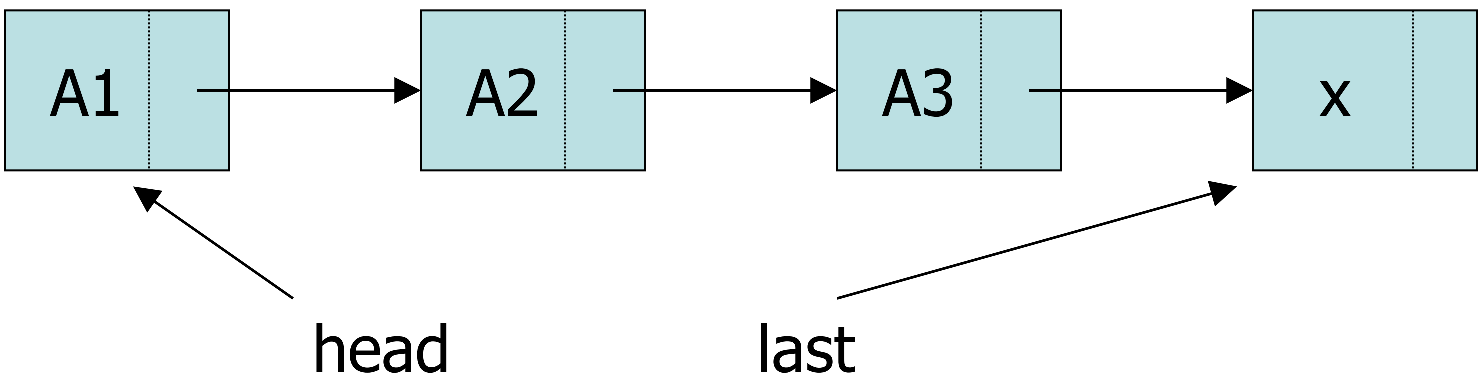

```
last->next = new Node();
last = last->next;
last->data = x;
last->next = null;
```

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



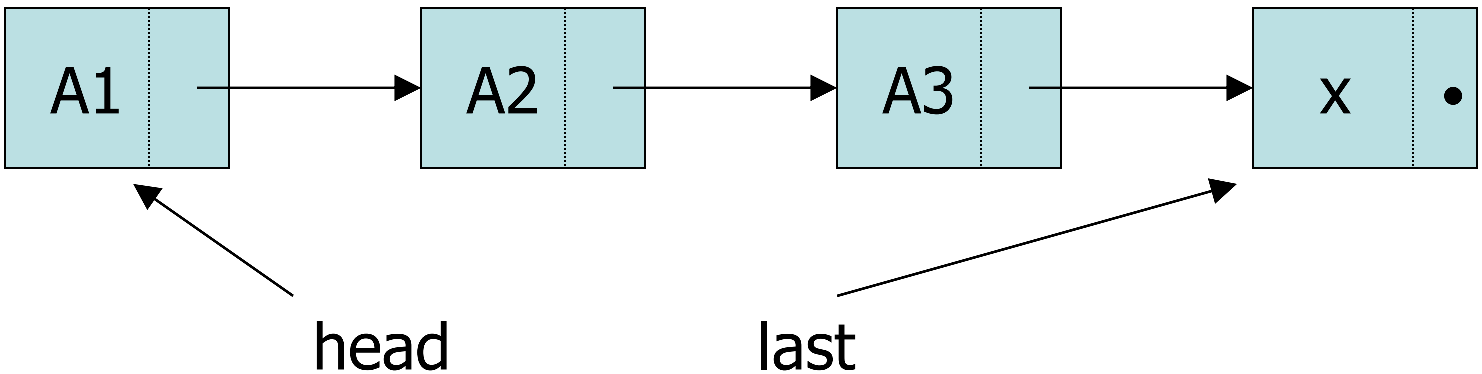

```
last->next = new Node();
last = last->next;
last->data = x;
last->next = null;
```

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



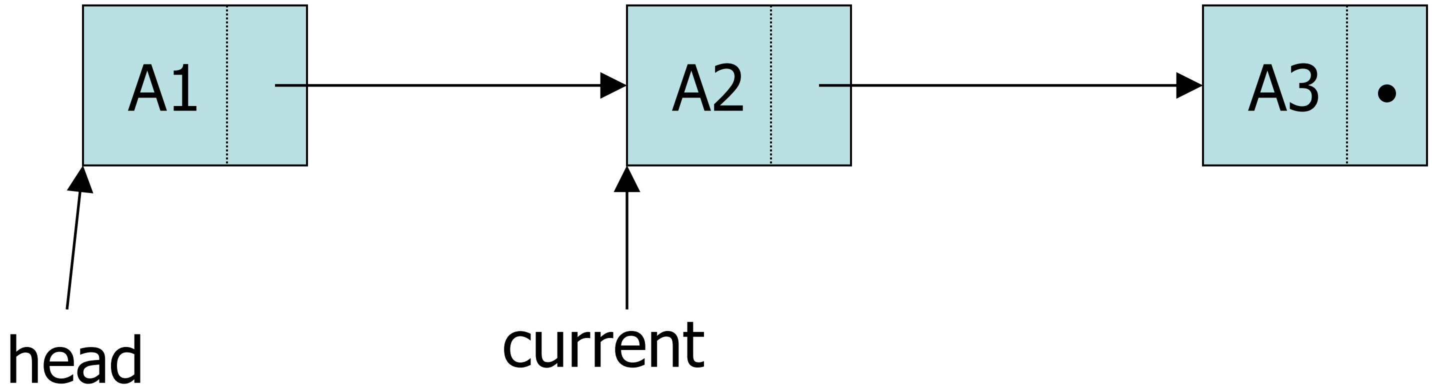

```
last->next = new Node();
last = last->next;
last->data = x;
last->next = null;
```

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)

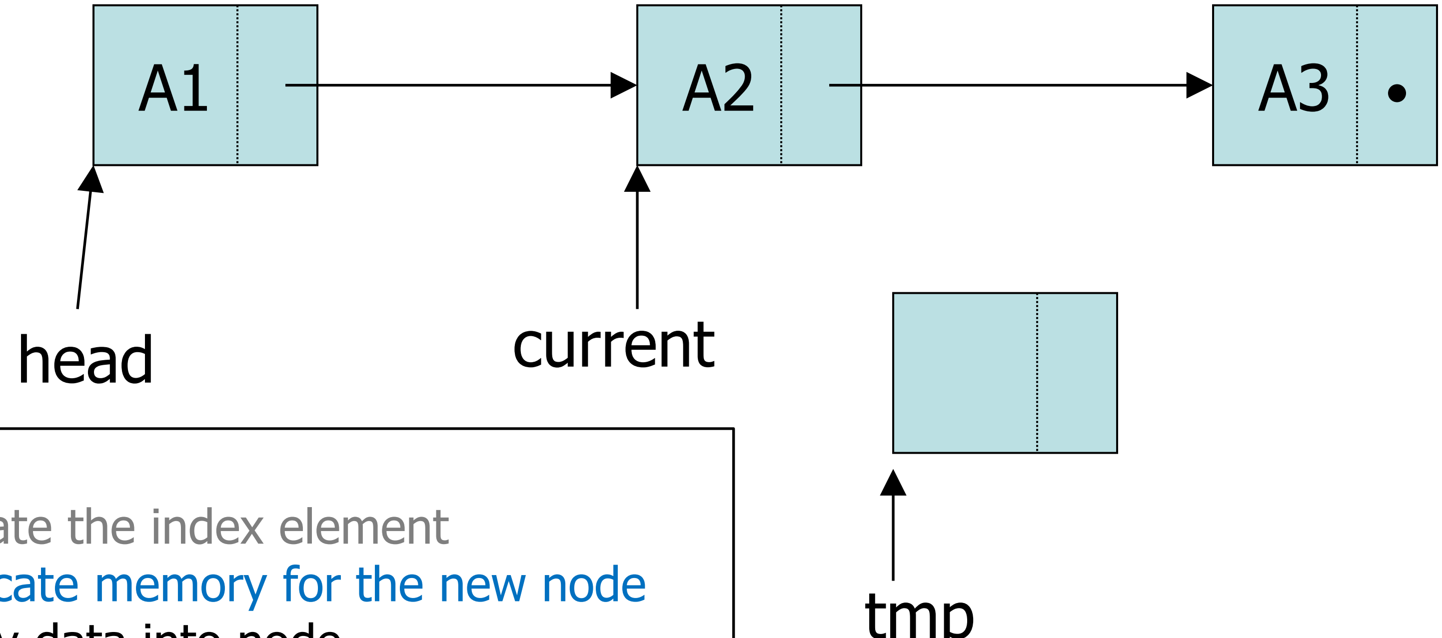
- Suppose `current` points to the middle element of the list
  - We can add a new item x by doing this




```
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



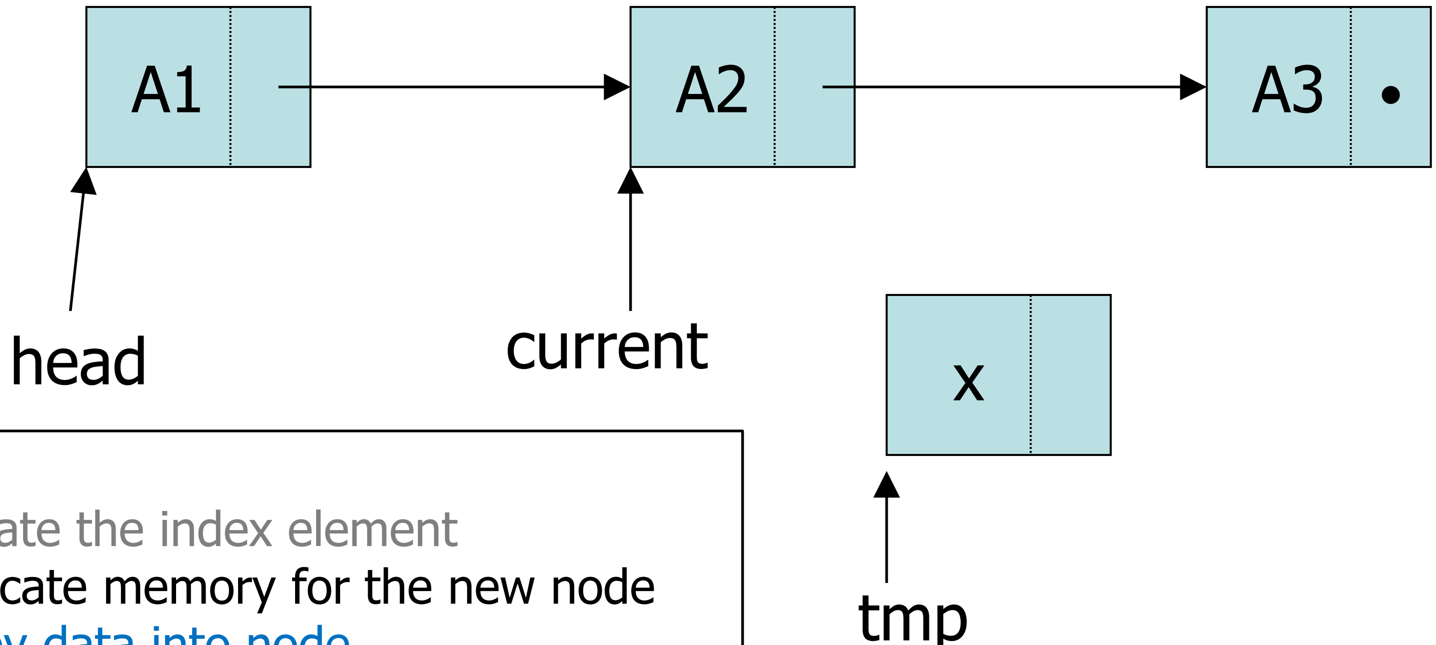
Steps

- 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

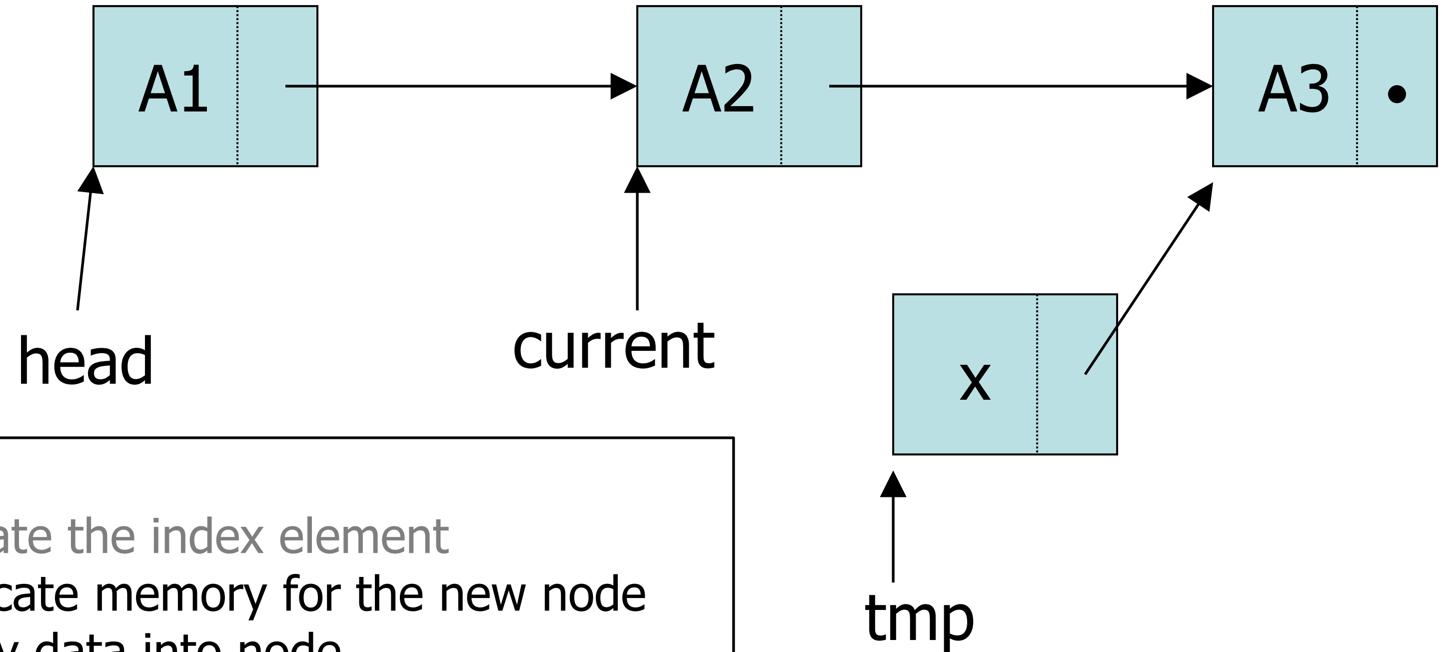


Steps
- Locate the index element
- Allocate memory for the new node
- Copy data into node
- Point the new node to its successor (next node)
- Point the new node's predecessor (preceding node) to the new node

```
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



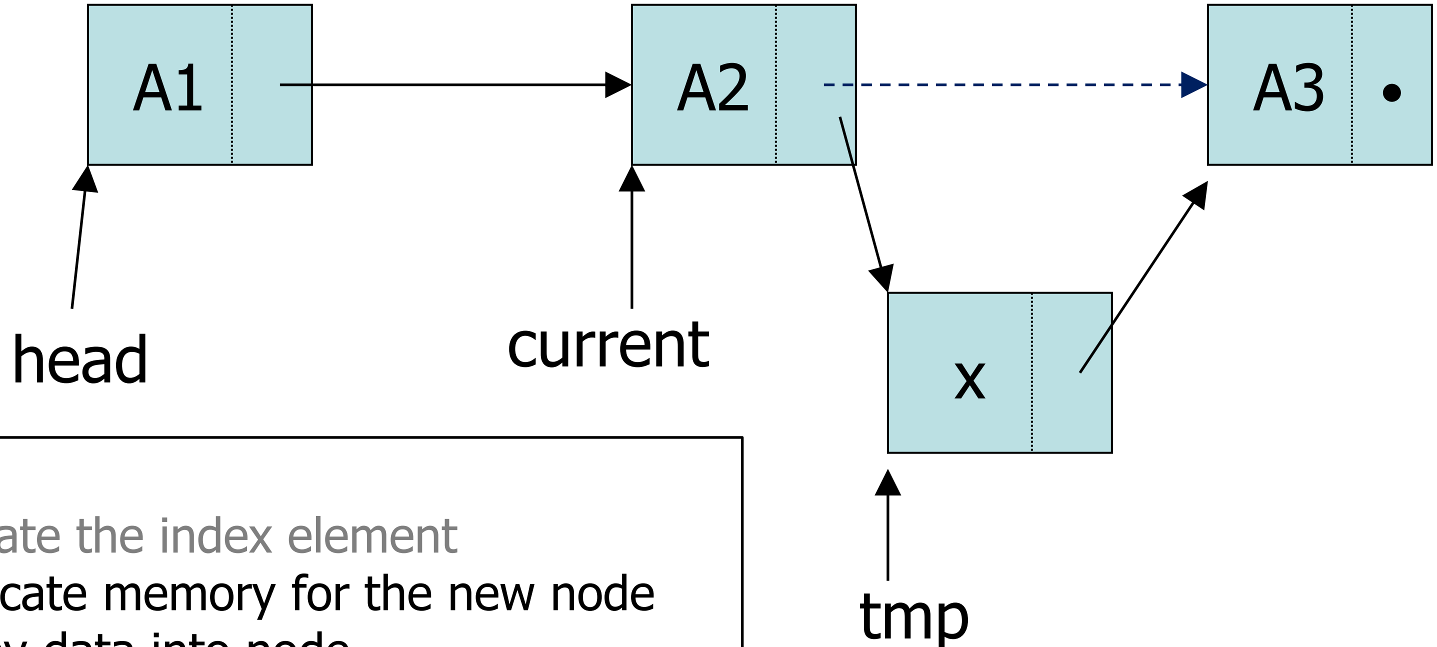
Steps

- 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

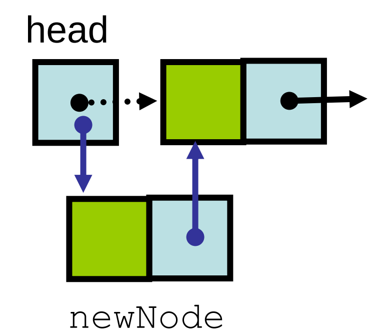
Inserting a New Node (3)

```
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



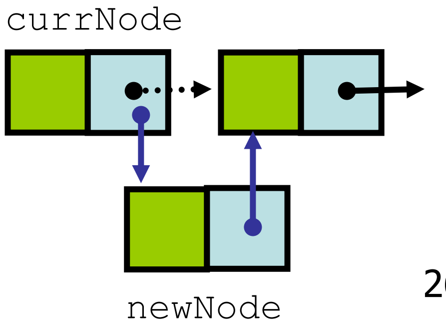
Inserting a New Node (3)

```
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;
}
```

Any Question So Far?