COMP 2710
Software Construction

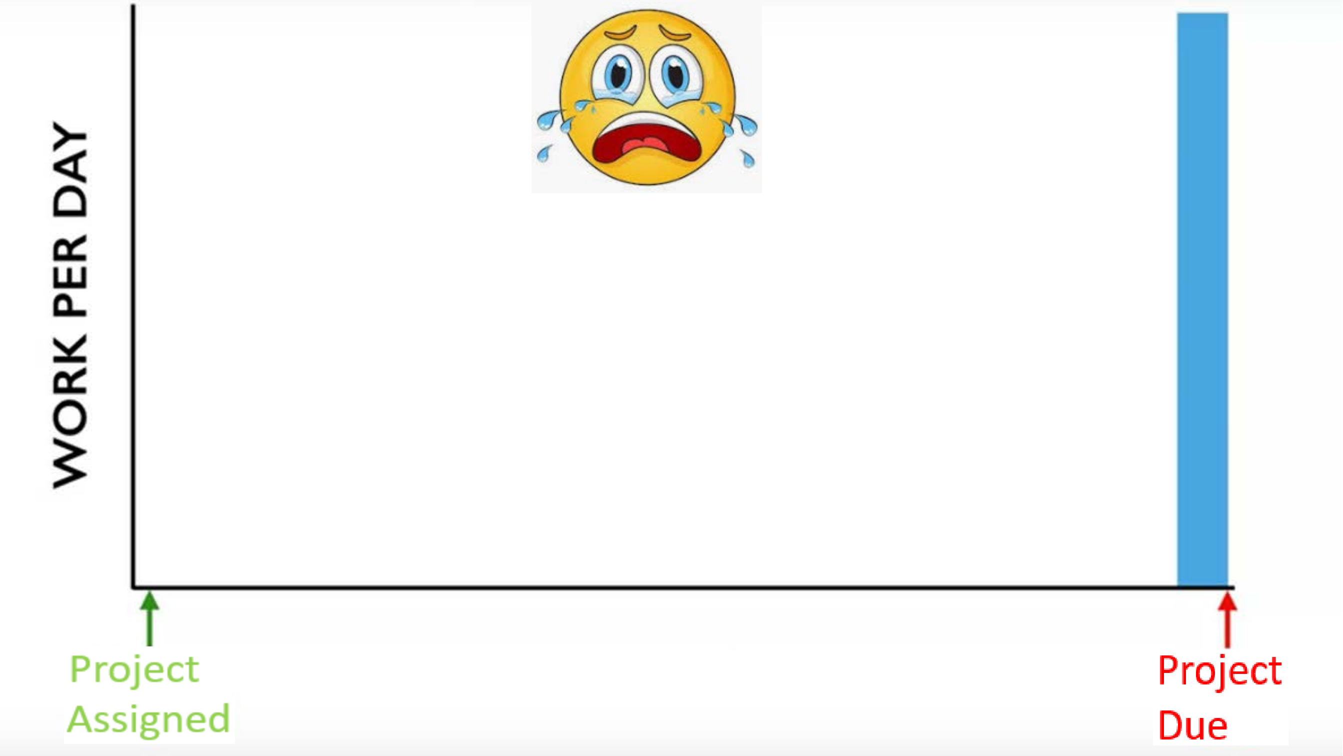
Chapter 4

Singly/Doubly Linked List

AUBURN UNIVERSITY

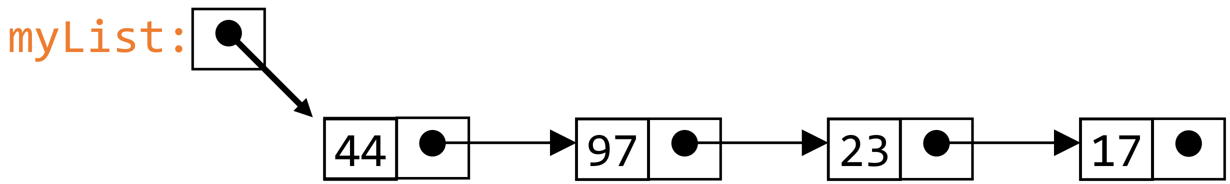
SAMUEL GINN COLLEGE OF ENGINEERING


WORK PER DAY
Project
Assigned
Project
Due


WORK PER DAY
Project
Assigned
Project
Due

Singly Linked List

Creating links in Java



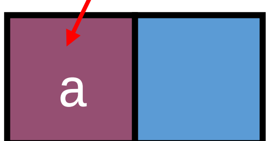

```
class Node {
     int value;
     Node next;

     Node (int v, Node n) { // constructor
          value = v;
          next = n;
     }

}

Node temp = new Node(17, null);

temp = new Node(23, temp);

temp = new Node(97, temp);

Node myList = new Node(44, temp);
```

Pointers and references

- In C and C++ we have "pointers," while in Java we have "references"
  - These are essentially the same thing
    - The difference is that C and C++ allow you to modify pointers in arbitrary ways, and to point to anything
  - In Java, a reference is more of a "black box," or ADT
    - Available operations are:
      - dereference ("follow")
      - copy
      - compare for equality
    - There are constraints on what kind of thing is referenced: for example, a reference to an `array of int` can *only* refer to an `array of int`

Pointer Implementation (Linked List)

- First, define a Node.

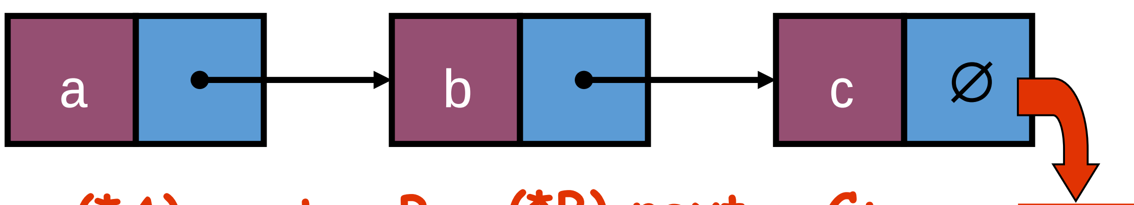



```
struct Node {
    double data;        // data
    Node* next;         // pointer to next
};
```

"self-referentiality"

- Second, linked them together.

Node * A; Node * B; Node * C;



(*A).next = B; (*B).next = C;

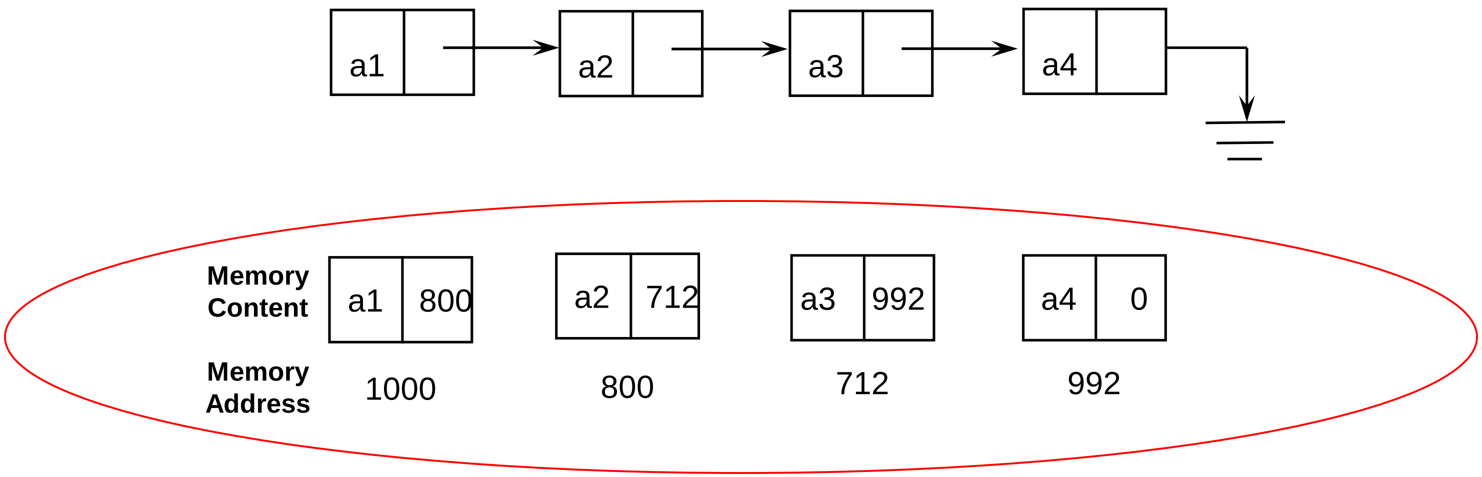
(*C).next = NULL;

We want a new symbol ☺

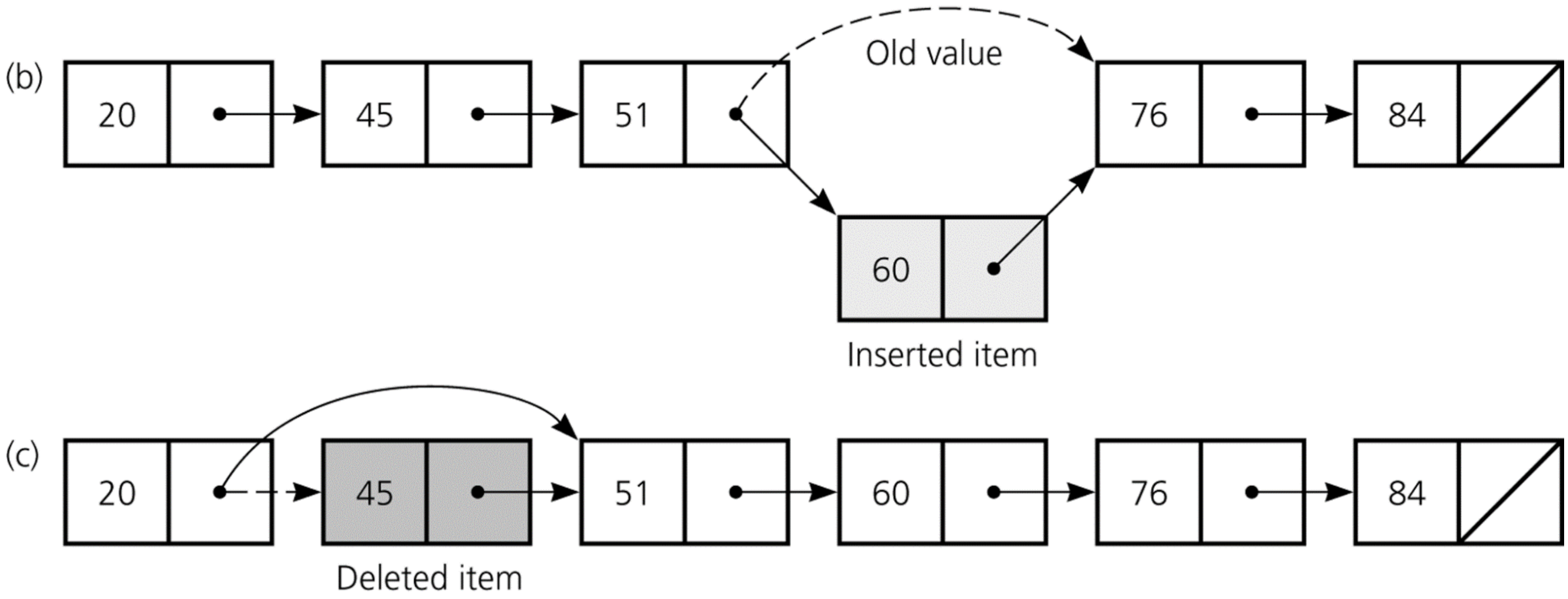
A->next = B; B->next = C;

C->next = NULL;

What does the memory look like?

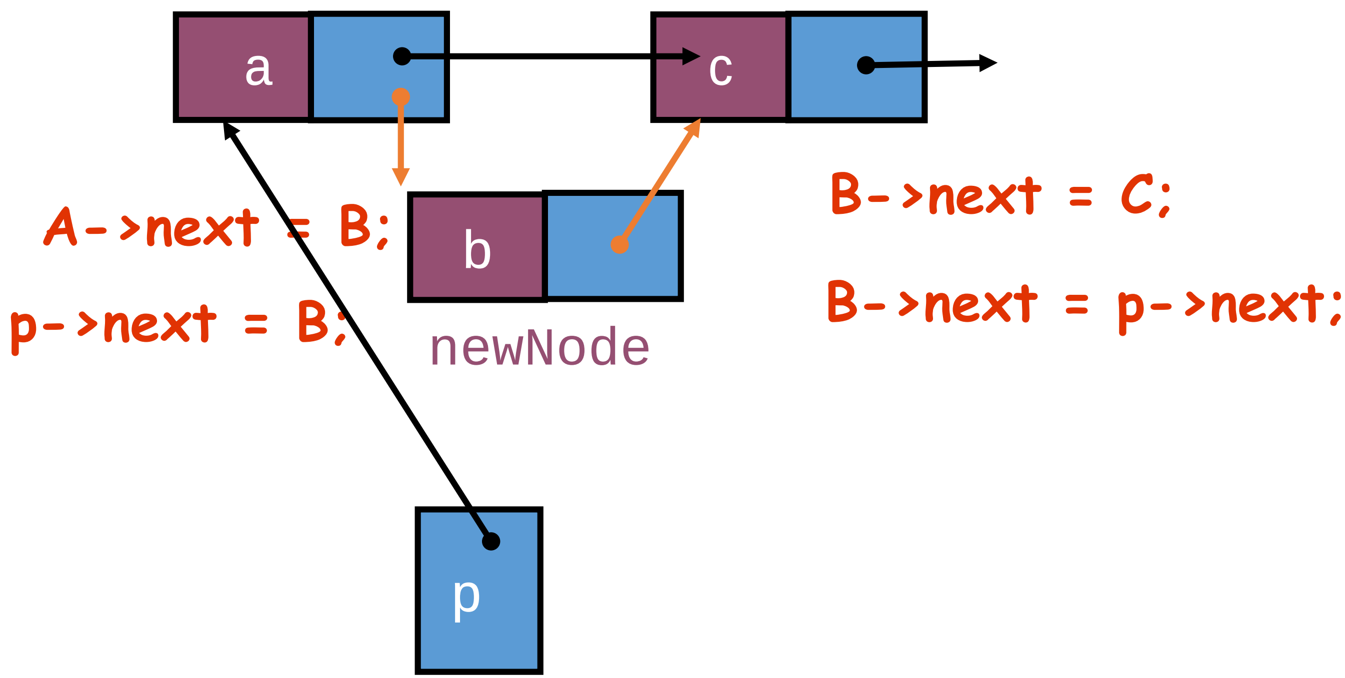


Update linked list



Inserting a node

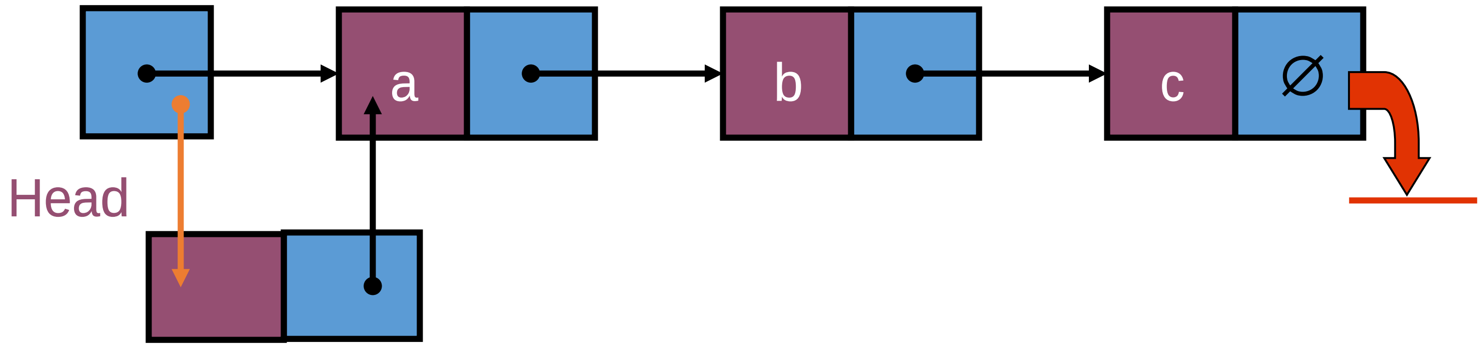
- What if I only want to use a pointer that points to A?



However, what if we want to insert in the front?

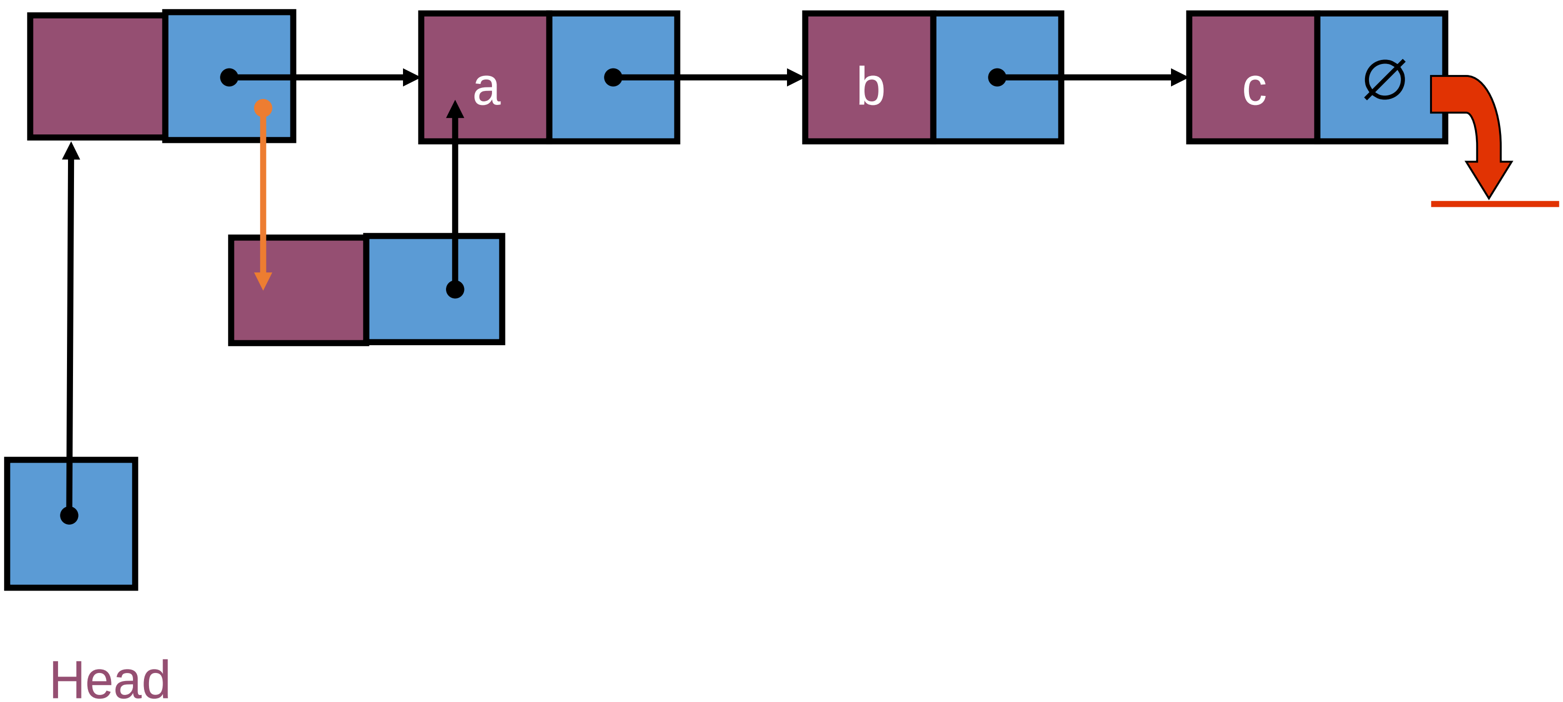
Option One

- Deal with it differently.

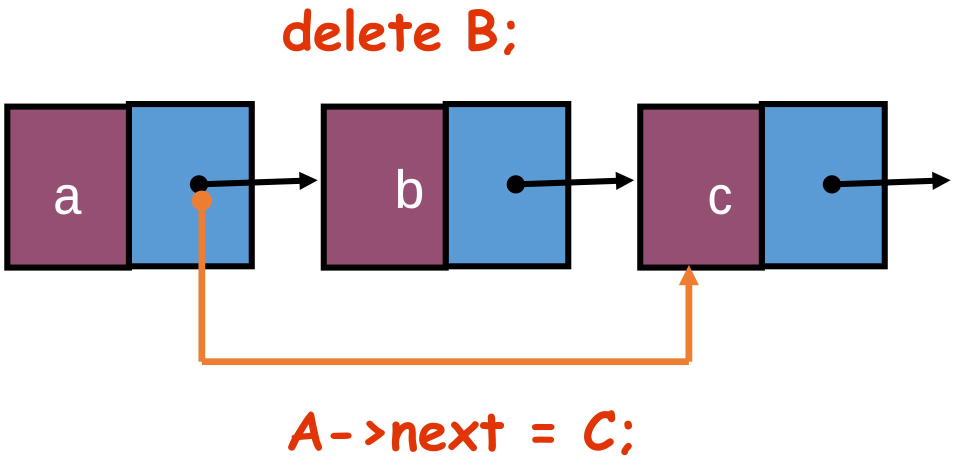


Option Two

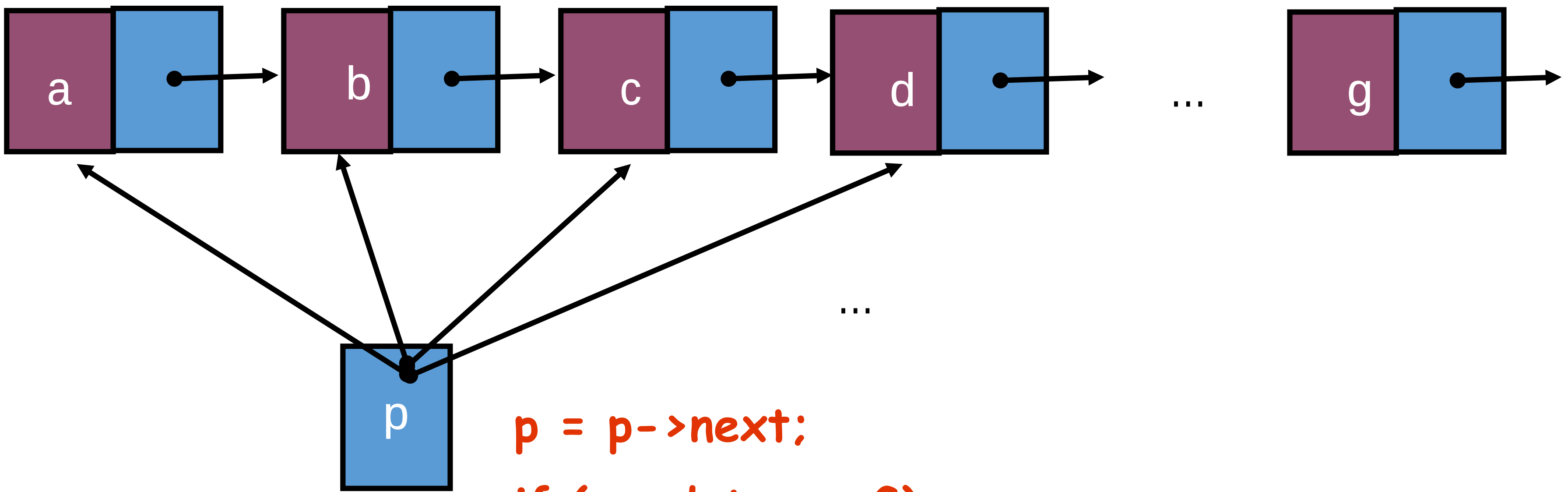
- Add a faked node. ☺ It is always there (even for an empty linked list)



Deleting a Node



Finding a node.




```
p = p->next;
if (p->data == G)
    ...
```

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)

Inserting a new node

```
Node* 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

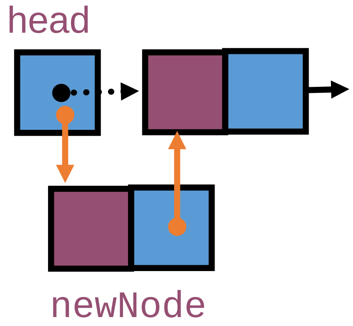
```
Node* InsertNode(int index, double x) {
    if (index < 0) return NULL;

    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* 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;
}
```

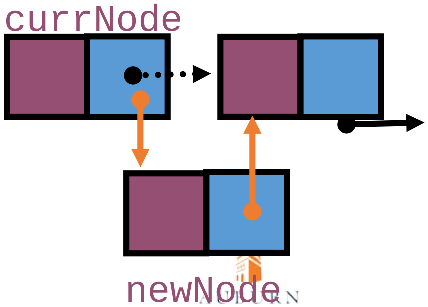
Insert as first element



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



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

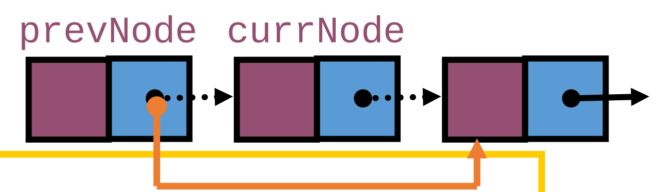
Deleting a node

```
int 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

Deleting a node

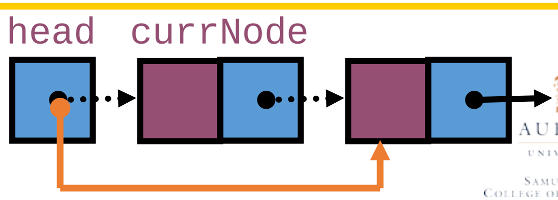
```
int DeleteNode(double x) {
    Node* prevNode =   NULL;
    Node* currNode =   head;
    int currIndex  =   1;
    while (currNode && currNode->data != x) {
        prevNode    =   currNode;
        currNode    =   currNode->next;
        currIndex++;
    }
    if (currNode) {
        if (prevNode)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



Since the space for the node was allocated by “new”

Deleting a node

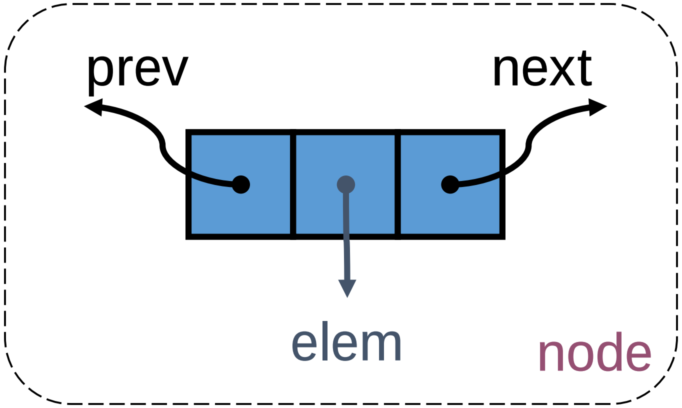
```
int DeleteNode(double x) {
    Node* prevNode =   NULL;
    Node* currNode =   head;
    int currIndex  =   1;
    while (currNode && currNode->data != x) {
        prevNode    =   currNode;
        currNode    =   currNode->next;
        currIndex++;
    }
    if (currNode) {
        if (prevNode)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

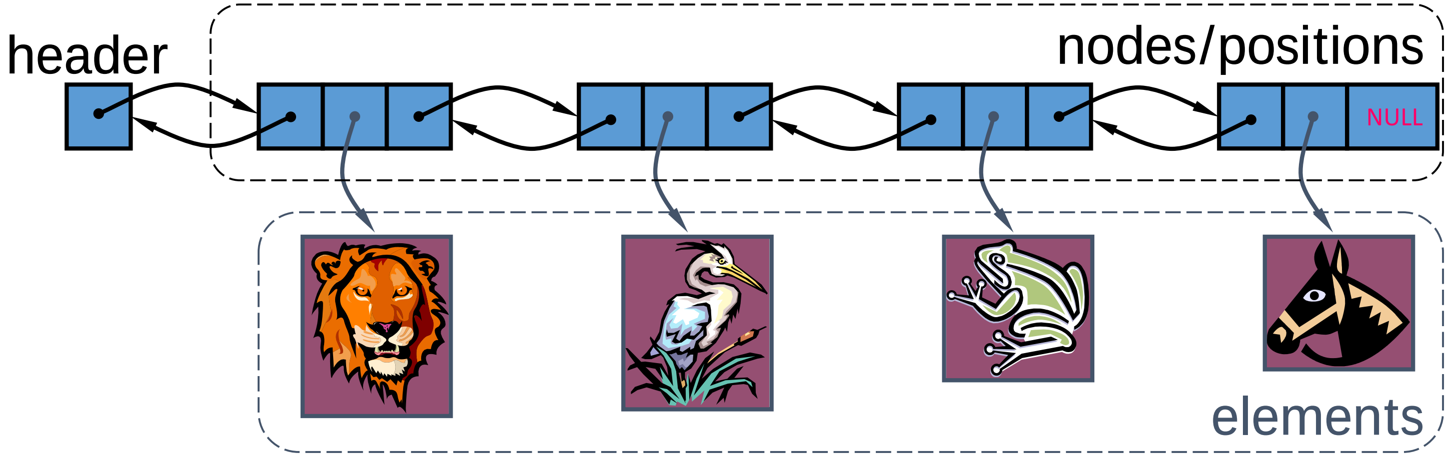


Doubly Linked List

Doubly Linked List

- A doubly linked list provides a natural implementation of the List ADT
- Nodes implement Position and store:
  - **element**
  - **link to the previous node**
  - **link to the next node**





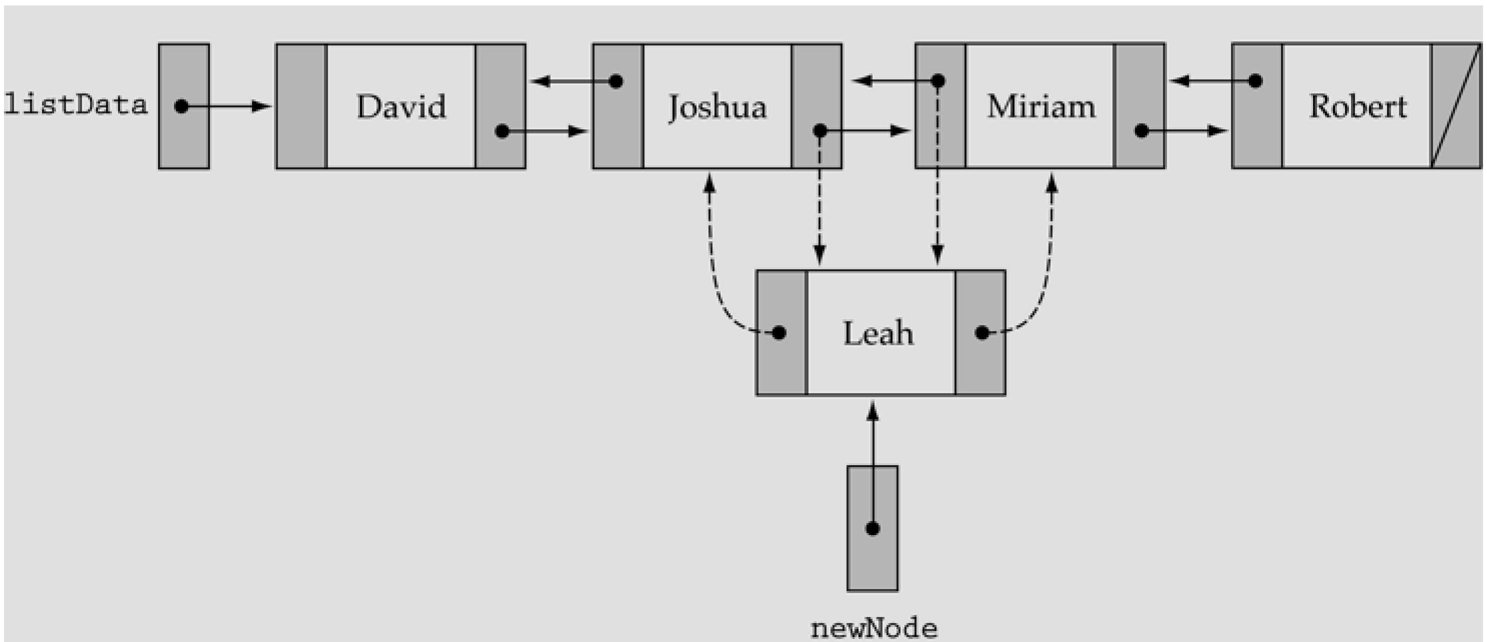
Advantages over Singly-linked Lists

- Quick update operations:

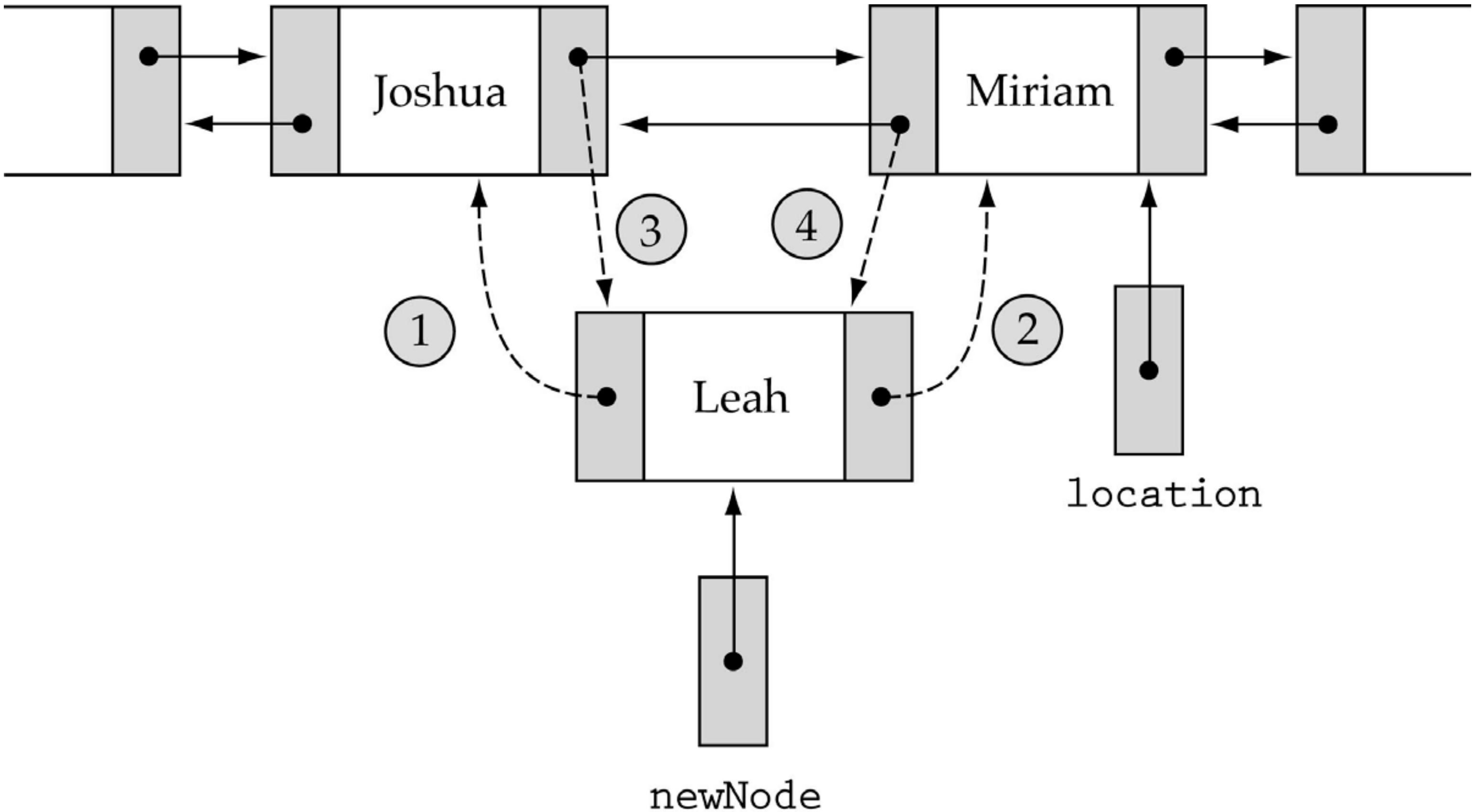
  such as: insertions, deletions at *both* ends (head and tail), and also at the middle of the list.

- A node in a doubly-linked list store two references:
  - A ***next*** link; that points to the next node in the list, and
  - A ***prev*** link; that points to the previous node in the list.

Insertion

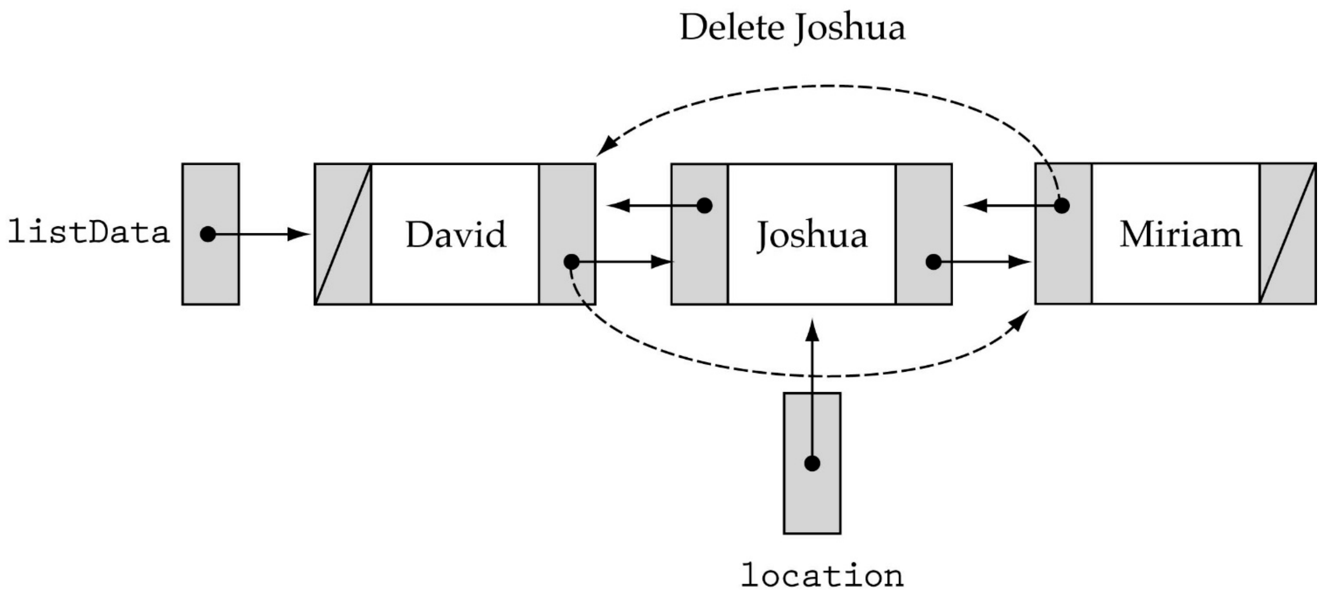


Insertion implementation



1. newNode->back = location->back; 3. location->back->next=newNode;
2. newNode->next = location 4. location->back = newNode;

Deletion



- Be very careful about the end cases!!