LINKED LISTS

SUKHBIR TATLA

SUKHBIR.TATLA@SENECACOLLEGE.CA

INTRODUCTION TO THE LINKED LIST

- A linked list is a series of connected *nodes*, where each node is a data structure.
- A linked list can grow or shrink in size as the program runs
- Linked lists and arrays are similar since they both store collections of data.
  - The *array's* features all follow from its strategy of allocating the memory for all its elements in one block of memory.
  - *Linked lists* use an entirely different strategy: linked lists allocate memory for each element separately and only when necessary.

DISADVANTAGES OF ARRAYS

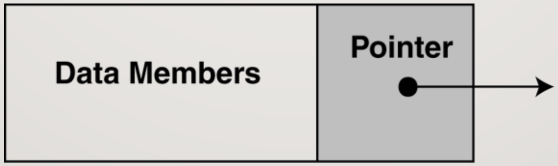
1. **The size of the array is fixed**.
   - In case of **dynamically resizing** the array from size S to 2S, we need 3S units of available memory.
   - Programmers allocate arrays which seem **"large enough**" This strategy has two disadvantages: (a) most of the time there are just 20% or 30% elements in the array and 70% of the space in the array really is wasted. (b) If the program ever needs to process more than the declared size, the code breaks.
2. **Inserting (and deleting)** elements into the middle of the array is potentially expensive because existing elements need to be shifted over to make room.

ADVANTAGES OF LINKED LISTS OVER ARRAYS AND VECTORS

- A linked list can easily grow or shrink in size.
- Insertion and deletion of nodes is quicker with linked lists than with vectors.
- Linked lists are appropriate when the number of data elements to be represented in the data structure at once is unpredictable.
- Linked lists are dynamic, so the length of a list can increase or decrease as necessary.
- Each node does not necessarily follow the previous one physically in the memory.
- Linked lists can be maintained in sorted order by inserting or deleting an element at the proper point in the list.

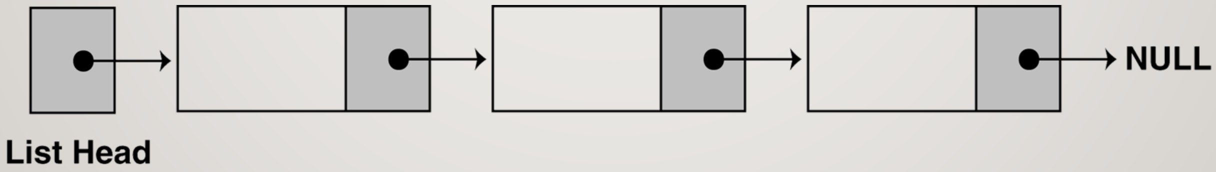
THE COMPOSITION OF A LINKED LIST

- Each node in a linked list contains one or more members that represent data.
- In addition to the data, each node contains a pointer, which can point to another node.



THE COMPOSITION OF A LINKED LIST

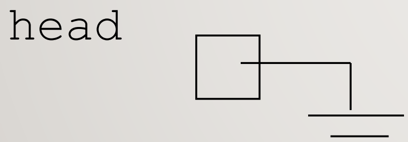
- A linked list is called "linked" because each node in the series has a pointer that points to the next node in the list.



EMPTY LIST

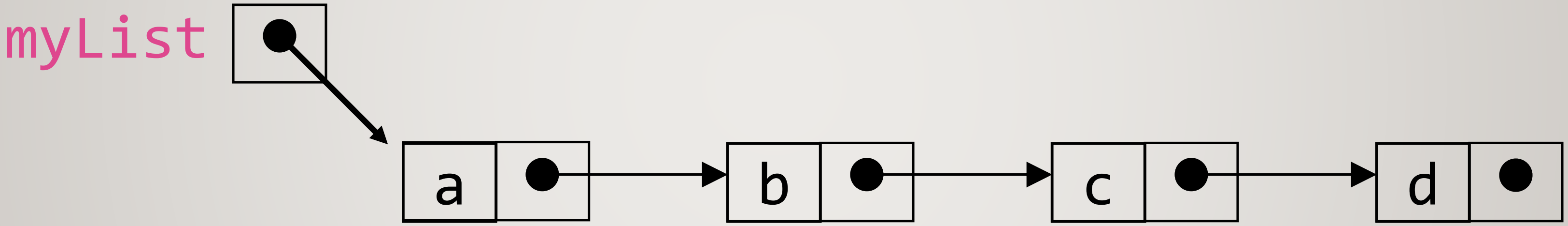
- Empty Linked list is a single pointer having the value of NULL.

```
head = NULL;
```



ANATOMY OF A LINKED LIST

- A linked list consists of:
  - A sequence of nodes



Each node contains a value
and a link (pointer or reference) to some other node

The last node contains a null link

The list may (or may not) have a header

MORE TERMINOLOGY

- A node's successor is the next node in the sequence
  - The last node has no successor
- A node's predecessor is the previous node in the sequence
  - The first node has no predecessor
- A list's length is the number of elements in it
  - A list may be empty (contain no elements)

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`
  - Operations on `List`

```
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;
};
```

LINKED LIST OPERATIONS

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

APPENDING A NODE TO THE LIST

- To append a node to a linked list means to add the node to the end of the list.
- The pseudocode is shown below.
- The C++ code follows.

```
Create a new node.
Store data in the new node.
If there are no nodes in the list
    Make the new node the first node.
Else
    Traverse the List to Find the last node.
    Add the new node to the end of the list.
End If.
```

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* 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

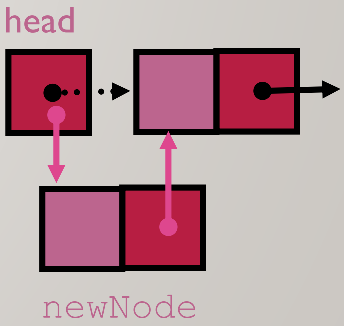
```
Node* List::InsertNode(int index, double x) {
    if (index < 0) return NULL;

    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;
}
```

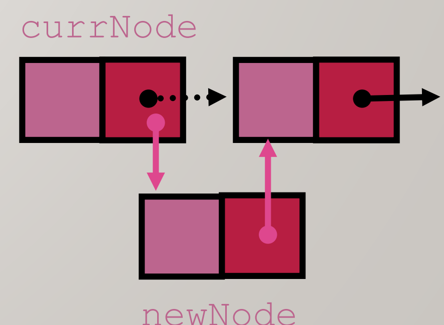
Insert as first element



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;
   return 0;
}
```

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

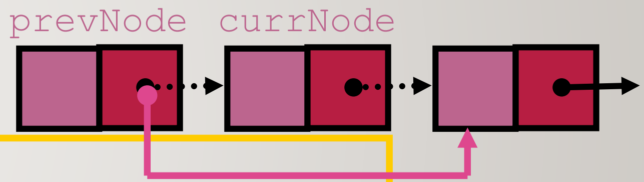
DELETING A NODE

```
int 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

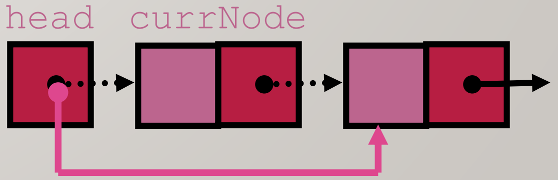
DELETING A NODE

```
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

```
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;
    }
}
```

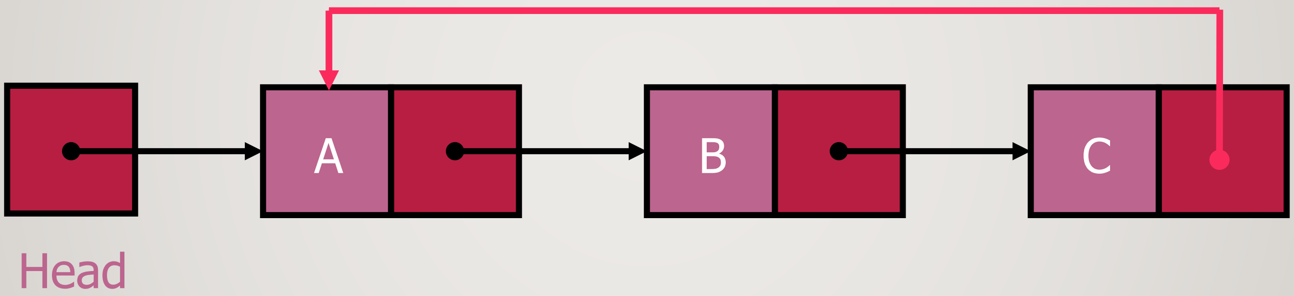
PRACTICE

- REALITY CHECK (Week 10)

- Question #1 (Word problem)

VARIATIONS OF LINKED LISTS

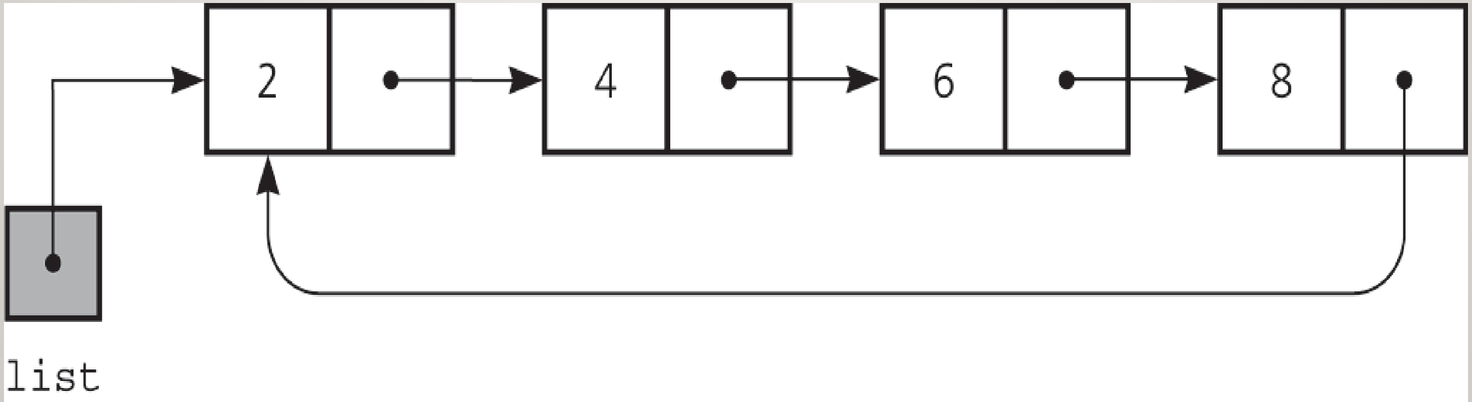
- *Circular linked lists*
  - **The last node points to the first node of the list**



  - **How do we know when we have finished traversing the list? (Tip: check if the pointer of the current node is equal to the head.)**

CIRCULAR LINKED LISTS

- Last node references the first node
- Every node has a successor
- No node in a circular linked list contains *NULL*

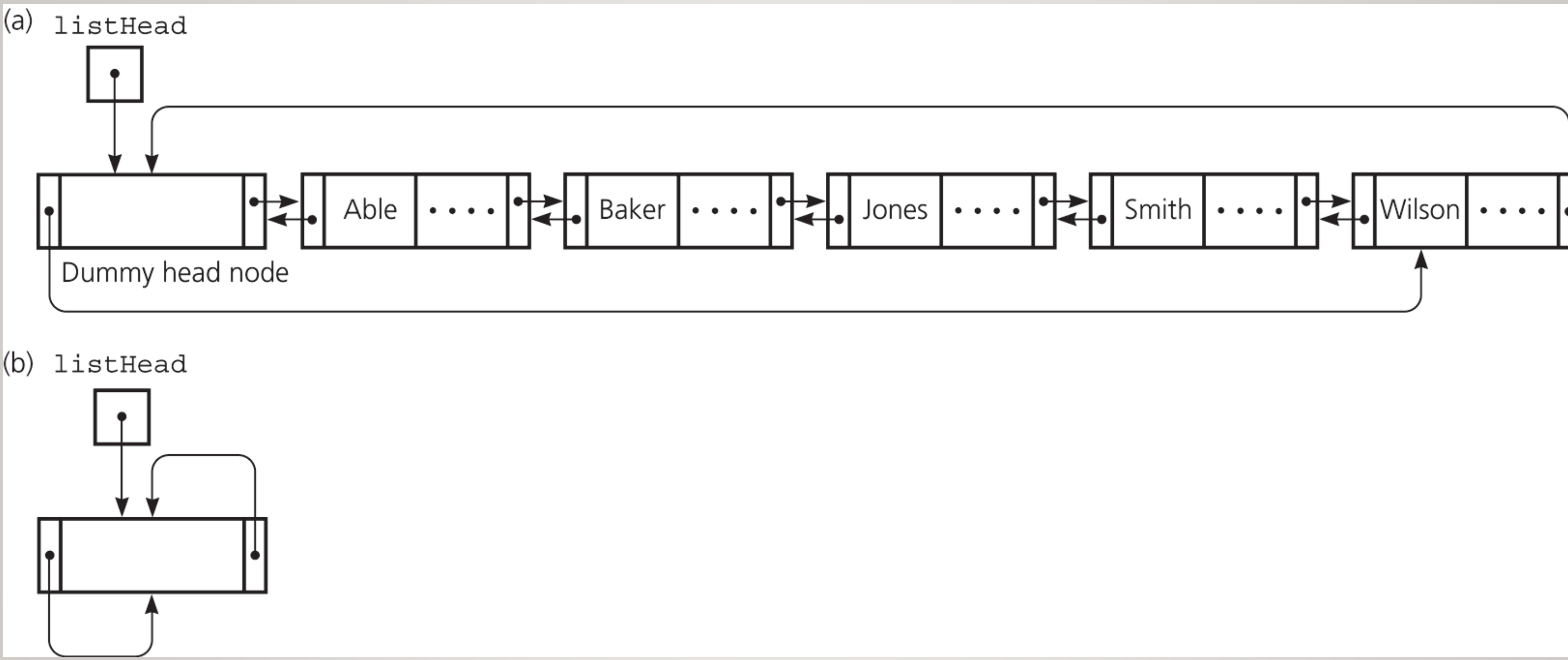


A circular linked list

CIRCULAR DOUBLY LINKED LISTS

- Circular doubly linked list
  - `prev` pointer of the dummy head node points to the last node
  - `next` reference of the last node points to the dummy head node
  - No special cases for insertions and deletions

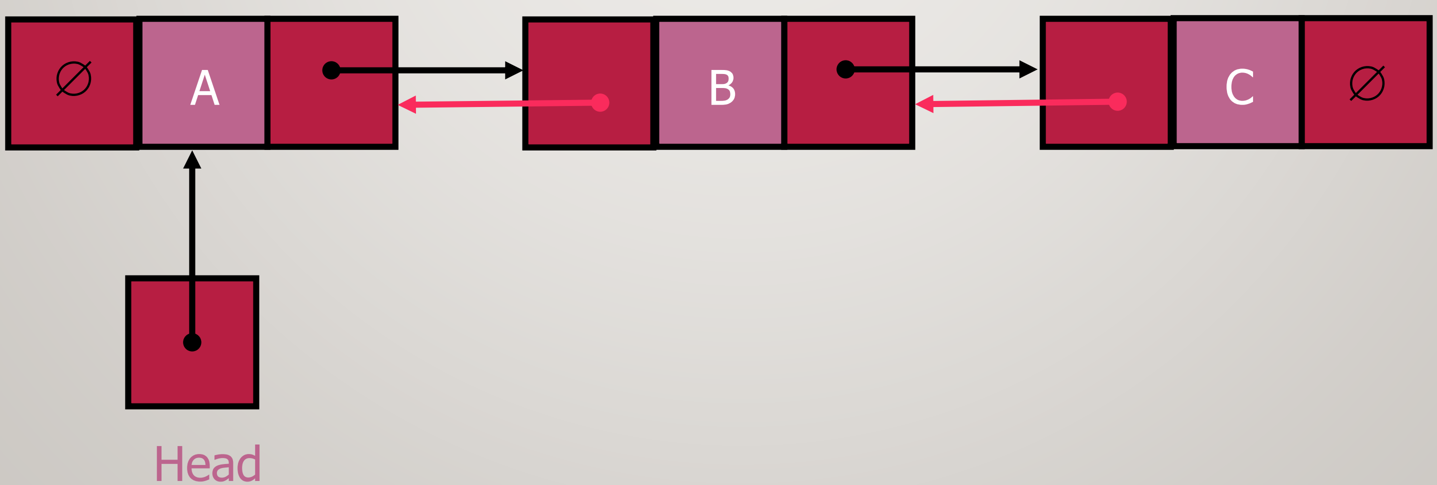
CIRCULAR DOUBLY LINKED LISTS



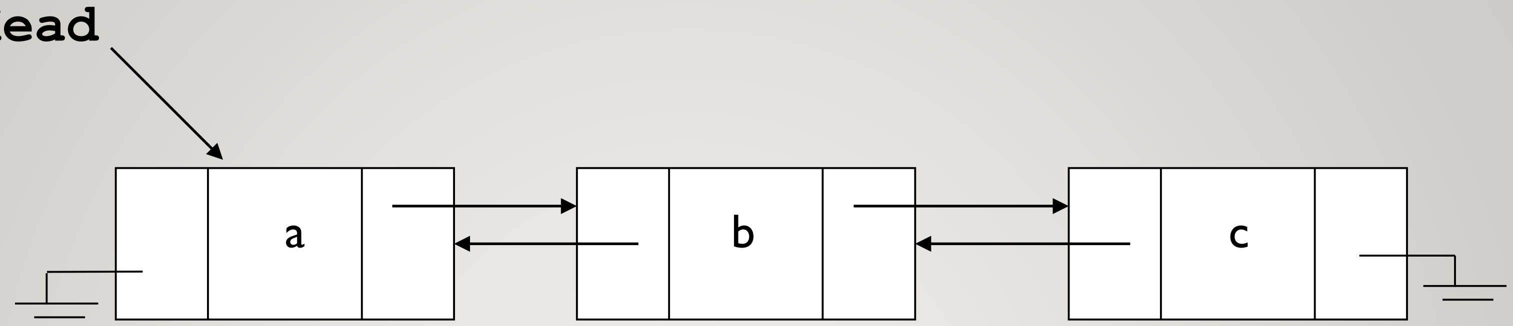
(a) A circular doubly linked list with a dummy head node
(b) An empty list with a dummy head node

VARIATIONS OF LINKED LISTS

- ***Doubly linked lists***
  - Each node points to not only successor but the predecessor
  - There are two `NULL`: at the first and last nodes in the list
  - Advantage: given a node, it is easy to visit its predecessor. Convenient to traverse lists backwards



DOUBLY LINKED LISTS



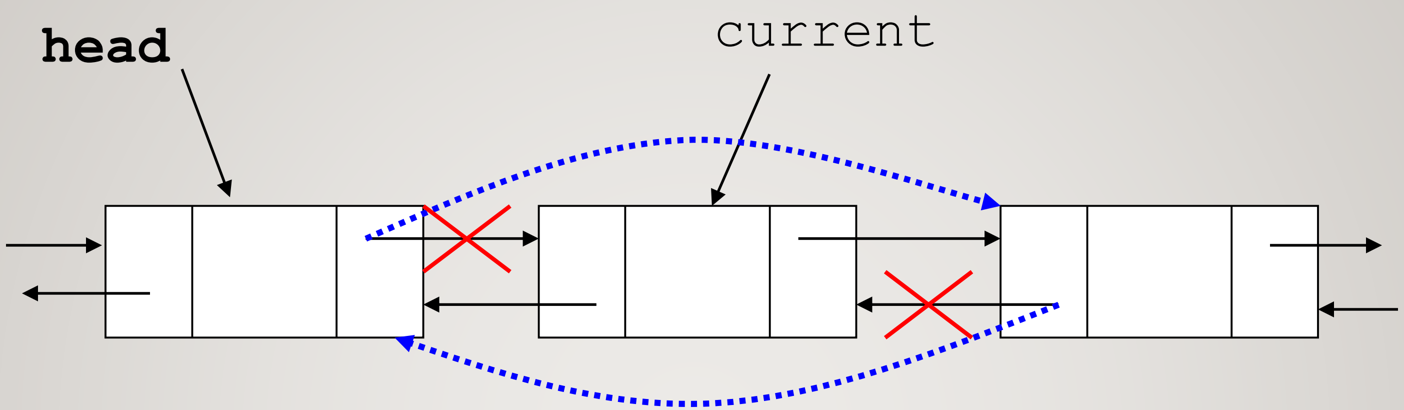
Advantages:

- Convenient to traverse the list backwards.
- Simplifies insertion and deletion because you no longer have to refer to the previous node.

Disadvantage:

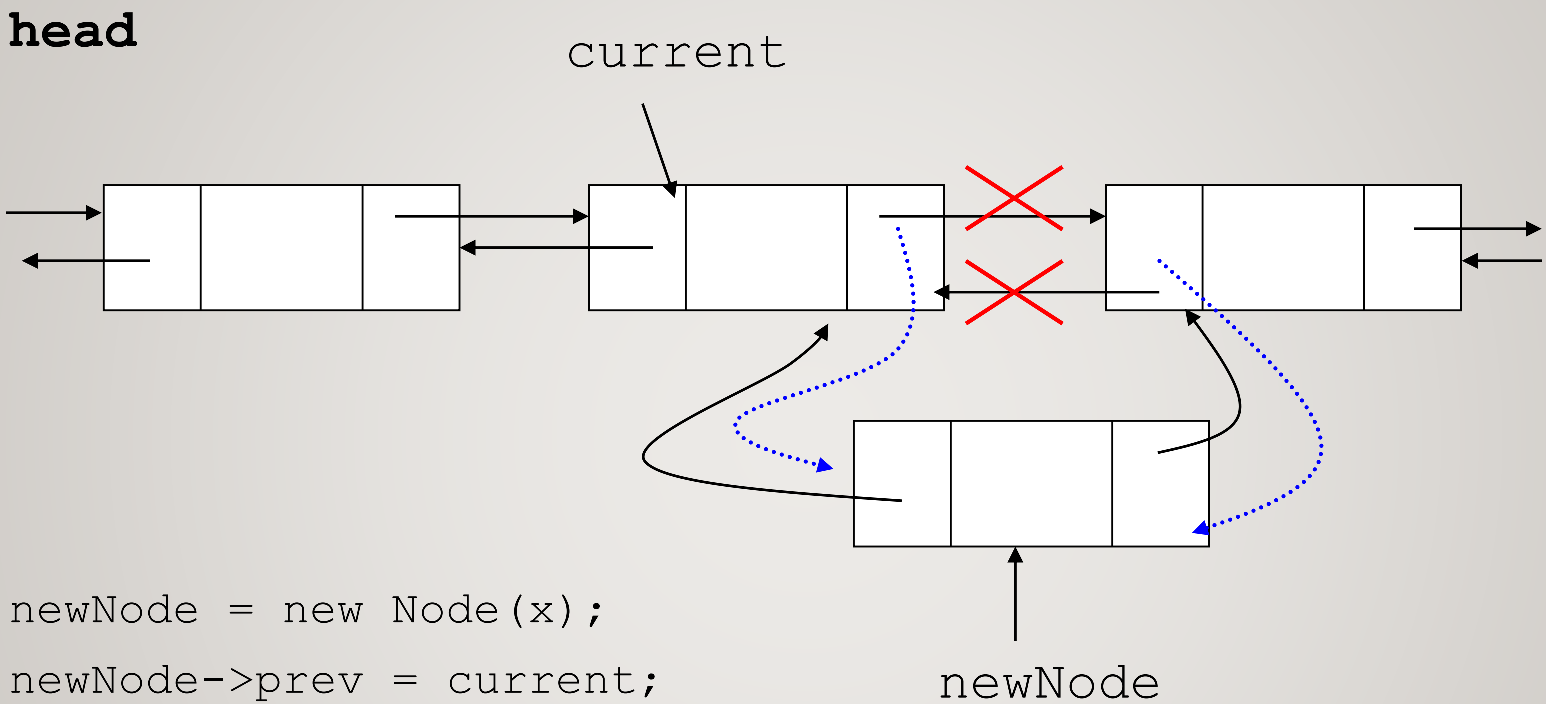
- Increase in space requirements.

DELETION




```
oldNode = current;

oldNode->prev->next = oldNode->next;

oldNode->next->prev = oldNode->prev;

delete oldNode;

current = head;
```

INSERTION




```
newNode = new Node(x);
newNode->prev = current;
newNode->next = current->next;
newNode->prev->next = newNode;
newNode->next->prev = newNode;
current = newNode;
```

THANK YOU!

Any Questions Please?