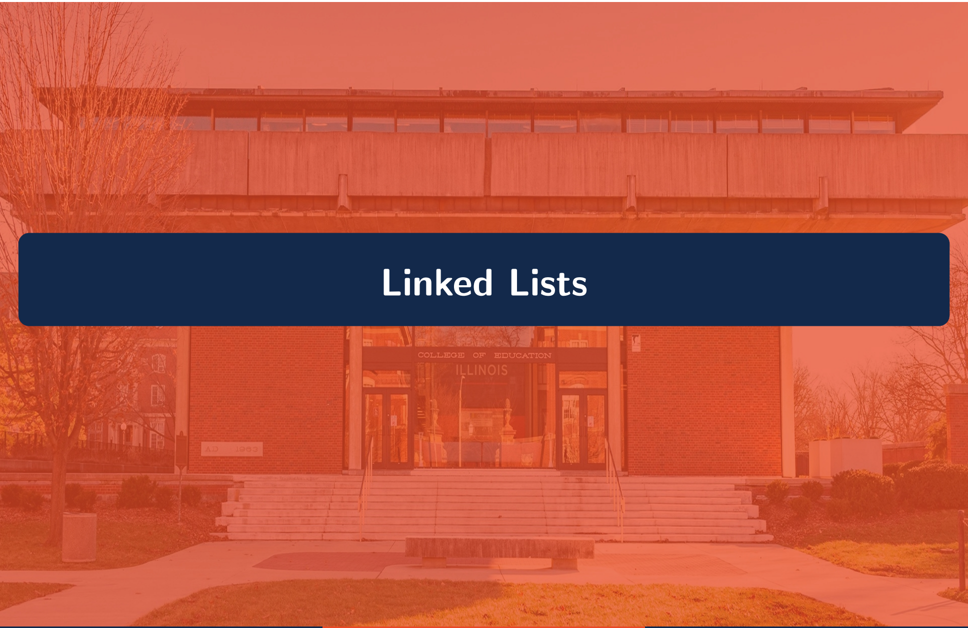
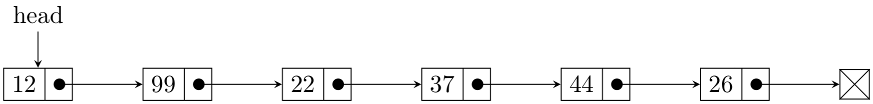


Linked Lists

Objectives

- Understand the implementation of a LinkedList and it's ListNode class.
- Understand the process of implementing the following operations:
  - Displaying the contents of the list.
  - Searching for a given Node
  - Removing Nodes
  - Adding Nodes
- You will extend these concepts in your mini-assignment in order to create a linked list with the following characteristics:
  - Singly linked
  - A head and tail pointers

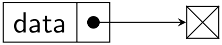


Basic Operations:

- **Add:** Adding operations add at either the front, middle, or end of the list.
- **Remove:** Remove removes a node either front, middle, or end of the list.
- **Search:** Search traverses the list, checks each node to see if it is the one we are looking for, and, if the node is found, returns a reference to it.

Off to the worksheet to implement the ListNode class.

```
class ListNode{
    public int data;
    public ListNode next;

    ListNode(data){
        this.data = data;
        next = null;
    }
}
```



Nested vs Inner Classes

```
class OuterClass{
    class InnerClass{
        // ...
    }
}
```

```
class OuterClass{
    static class NestedClass{
        // ...
    }
}
```

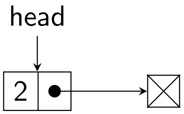
- **Inner Class:** Inner classes have no static modiers and therefore have: 1) access to all attributes and methods of the outer class and 2) cannot be instantiated unless the outerclass has been instantiated.
- **Nested Class:** Nested classes are declared with a `static` modifier and are therefore independent of the outerclass.

Key Point: If the class needs access to things in the outerclass we use an *inner class*. If it does not and we are encapsulating it as a design choice and the class is otherwise independent we use a *nested class*.

```
class LinkedList{
    static class ListNode{/* ... */}

    ListNode head;
    int size;

    LinkedList(){
        head = null;
        size = 0;
    }

    /* Methods Below */
}
```

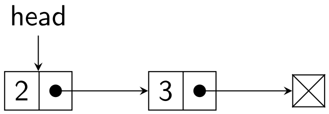
- For your lab we will be setting up ListNodes as *nested classes*.
- Design wise, they don't exist independently of the LinkedList class.
- However, they don't need access to any of the attributes or methods of LinkedList.

Manually Adding Nodes



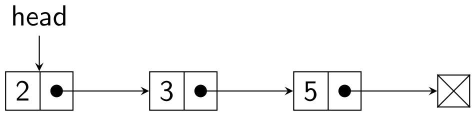
```
ListNode head = new ListNode(2);
```

Manually Adding Nodes



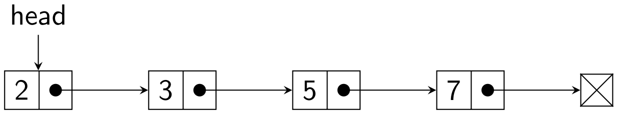
```
ListNode head = new ListNode(2);
head.next = new ListNode(3);
```

Manually Adding Nodes



```
ListNode head = new ListNode(2);
head.next = new ListNode(3);
head.next.next = new ListNode(5);
```

Manually Adding Nodes

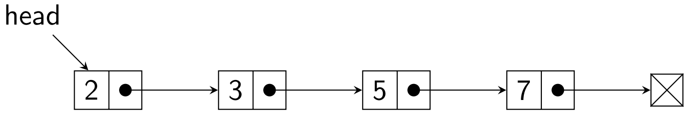


```
ListNode head = new ListNode(2);
head.next = new ListNode(3);
head.next.next = new ListNode(5);
head.next.next.next = new ListNode(7);
```

Displaying a LinkedList

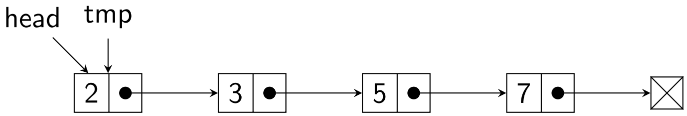
Off to the worksheet to practice this!

Displaying a LinkedList



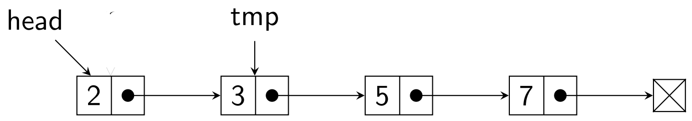
- We make a copy of the node referenced by head
  - ListNode tmp = head;
- We advance that reference by iteratively doing tmp = tmp.next.
- Print the data in the node referenced by tmp.
- We stop once tmp == null.

Displaying a LinkedList



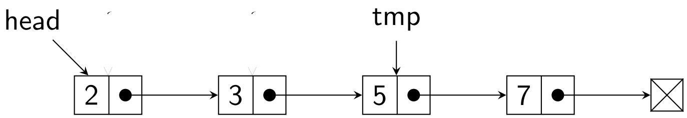
- We make a copy of the node referenced by head
    - ListNode tmp = head;
- We advance that reference by iteratively doing tmp = tmp.next.
- Print the data in the node referenced by tmp.
- We stop once tmp == null.

Displaying a LinkedList



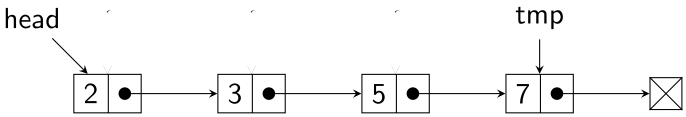
- We make a copy of the node referenced by head
  - ListNode tmp = head;
- We advance that reference by iteratively doing tmp = tmp.next.
- Print the data in the node referenced by tmp.
- We stop once tmp == null.

Displaying a LinkedList



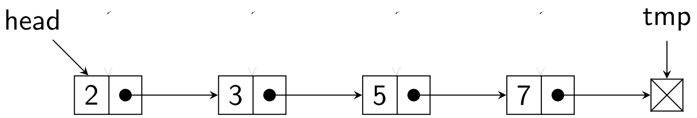
- We make a copy of the node referenced by head
    - ListNode tmp = head;
- We advance that reference by iteratively doing tmp = tmp.next.
- Print the data in the node referenced by tmp.
- We stop once tmp == null.

Displaying a LinkedList



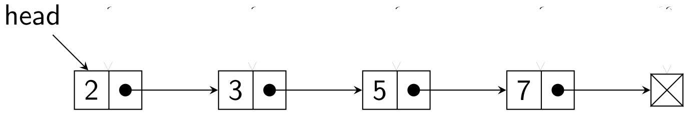
- We make a copy of the node referenced by head
    - ListNode tmp = head;
- We advance that reference by iteratively doing tmp = tmp.next.
- Print the data in the node referenced by tmp.
- We stop once tmp == null.

Displaying a LinkedList



- We make a copy of the node referenced by head
  - ListNode tmp = head;
- We advance that reference by iteratively doing tmp = tmp.next.
- Print the data in the node referenced by tmp.
- We stop once tmp == null.

Displaying a LinkedList



- We make a copy of the node referenced by head
  - ListNode tmp = head;
- We advance that reference by iteratively doing tmp = tmp.next.
- Print the data in the node referenced by tmp.
- We stop once tmp == null.

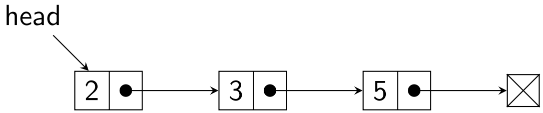
Worksheet: Displaying a LinkedList

Now we will implement this in the worksheet.

Adding nodes to a LinkedList

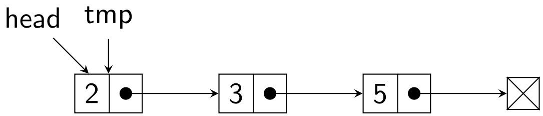
- **Case 0:** If the list is empty head=newNode.
- There are three other cases for adding a node to a linked list if the list is not empty.
  - **Case 1:** Adding to the end of the list.
  - **Case 2:** Adding to the front of a list.
  - **Case 3:** Adding in the middle of the list.
- We'll cover all cases at a highlevel
- We'll cover case 1 in the worksheet as it's similar to displaying.

Case 1: Adding a Node to the End of a LinkedList



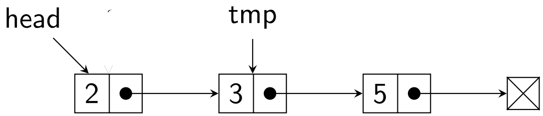
- Find the end of the list by advancing tmp until tmp.next == null.
- Once we find that create a new node and tmp.next = newNode

Case 1: Adding a Node to the End of a LinkedList



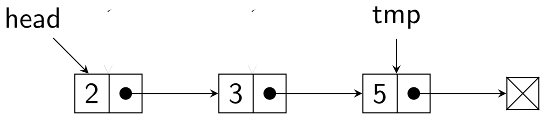
- Find the end of the list by advancing tmp until tmp.next == null.
- Once we find that create a new node and tmp.next = newNode

Case 1: Adding a Node to the End of a LinkedList



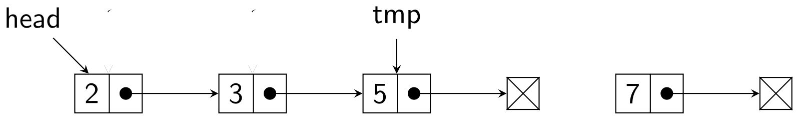
- Find the end of the list by advancing `tmp` until `tmp.next == null`.
- Once we find that create a new node and `tmp.next = newNode`

Case 1: Adding a Node to the End of a LinkedList



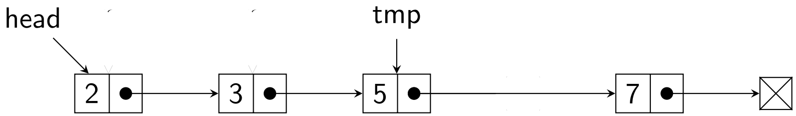
- Find the end of the list by advancing `tmp` until `tmp.next == null`.
- Once we find that create a new node and `tmp.next = newNode`

Case 1: Adding a Node to the End of a LinkedList



- Find the end of the list by advancing tmp until tmp.next == null.
- Once we find that create a new node and tmp.next = newNode

Case 1: Adding a Node to the End of a LinkedList



- Find the end of the list by advancing tmp until tmp.next == null.
- Once we find that create a new node and tmp.next = newNode

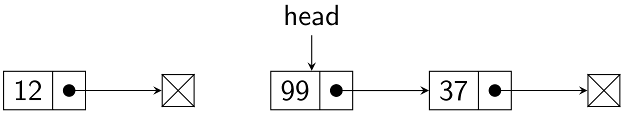
Case 1: Adding a Node to the End of a LinkedList Psuedocode

```
Procedure AddToEnd(Data)

    /* Step 1: Get our head copy */
    Tmp = Head

    /* Step 2: Go through the list and find the thing */
    While(Tmp.Next != Null)
        Tmp = Tmp.Next
    EndWhile

    Tmp.Next = new ListNode(Data)
    Size++
EndProcedure
```

Worksheet: Adding a to the end of the list

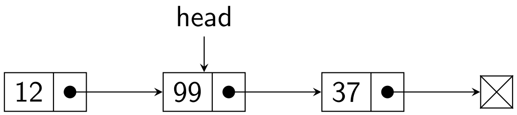
Now lets implement this method in a simplfied linked list class in the worksheet.

Case 2: Adding to the Front



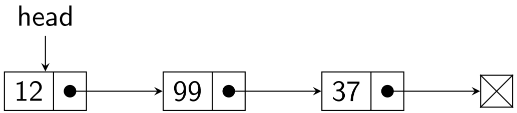
1. Instantiate a new ListNode

Case 2: Adding to the Front



1. Instantiate a new ListNode
2. Add the node to the front of the list by setting the new node's next reference to the head

Case 2: Adding to the Front

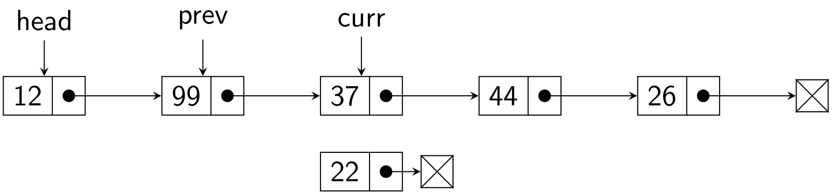


1. Instantiate a new ListNode
2. Add the node to the front of the list by setting the next reference
3. Update head attribute to reference the new node.

Case 2: Adding a Node to the Front of a LinkedList Psuedocode

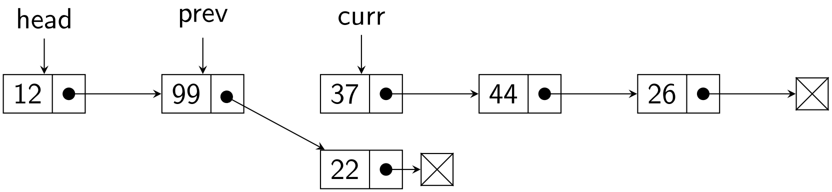
```
Procedure AddToFront(Data)
    /* Step 1: Check if our List is empty */
    If(Head is Null)
        Head = new ListNode(Data)
    Else
        NewNode = new ListNode(Data)

        /*Set the soon to be old head to the new one's next */
        NewNode.Next = Head

        /* Update the head attribute to be the new front */
        Head = NewNode
    EndIf
    Size++
EndProcedure
```

Case 3: Adding to the Middle



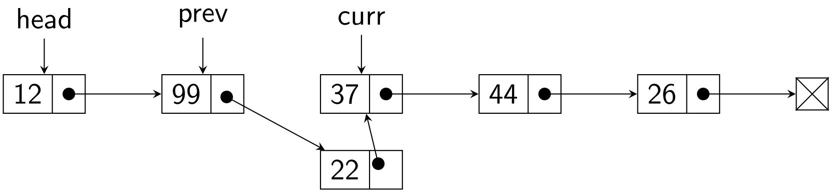
1. We have two pointers one that is similar to tmp from the previous example (curr) and one that follows it (prev).
2. Advance curr and prev until we find the position at which we want to insert.
3. Create a new node.

Case 3: Adding to the Middle



1. We have two pointers one that is similar to tmp from the previous example (curr) and one that follows it (prev).
2. Advance curr and prev until we find the position at which we want to insert.
3. Create a new node.
4. Set the prev.next to point to the new node.

Case 3: Adding to the Middle



1. We have two pointers one that is similar to tmp from the previous example (curr) and one that follows it (prev).
2. Advance curr and prev until we find the position at which we want to insert.
3. Create a new node.
4. Set the prev.next to point to the new node.
5. Set the new node's next pointer equal to curr.

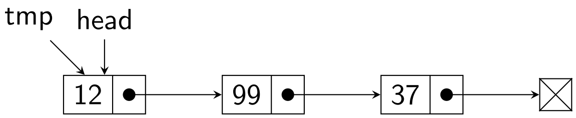
Case 3: Adding a Node to the Middle Psuedocode

```
Procedure AddToMiddle(Index, Data)
    If(Index not Valid) Return/Error

    Prev = Null
    Curr = Head

    For(1 to Index)
        Prev = Curr
        Curr = Curr.Next
    EndFor

    NewNode = new ListNode(Data)
    Prev.Next = NewNode
    NewNode.Next = Curr
    Size++
EndProcedure
```

Removing nodes from a LinkedList

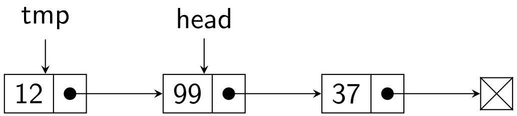
- **Case 0:** Again, if the list is empty we need to check if there is anything to remove.
- We have the same three cases when removing a node to a linked list:
  - **Case 1:** Adding to the front of a list.
  - **Case 2:** Adding to the end of the list.
  - **Case 3:** Adding in the middle of the list.
- We'll cover all cases at a highlevel
- We'll cover case 1 in the worksheet as it's similar to displaying.

Case 1: Remove from Front



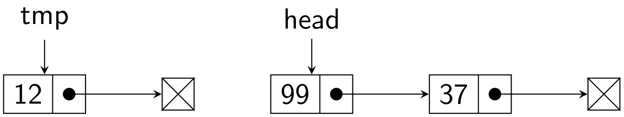
1. Make a tmpreference to the same node referenced by the head.

Case 1: Remove from Front



1. Make a tmp reference to the same node referenced by the head.
2. Advance the head.

Case 1: Remove from Front



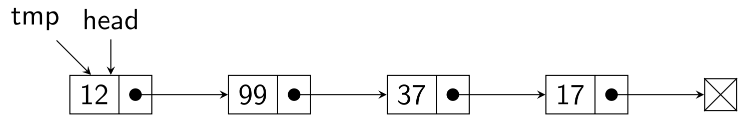
1. Make a tmp variable that references the same node referenced by the head.
2. Advance the head.
3. Set tmp.next equal to null.

Case 1: Remove from Front Psuedocode

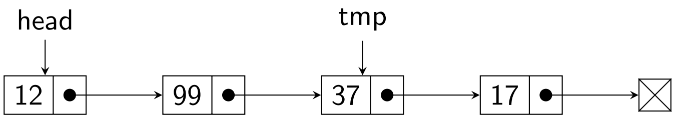
```
Procedure RemoveFromFront()
     If(Head is Null)
          Return/Error
     EndIf
     Tmp = Head
     Head = Head.Next
     Tmp.Next = Null
     Size--
EndProcedure
```

Case 2: Remove from Back

Before:

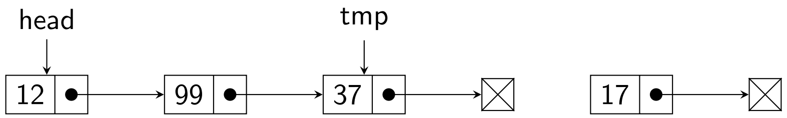


After:



1. Find the second to last node in the list by iterating until either:
    1. tmp.next.next == null
    2. you've iterated size - 2 times

Case 2: Remove from Back



1. Find the second to last node in the list.
2. Set it's reference to the next node equal to `null`.

Case 2: Remove from end

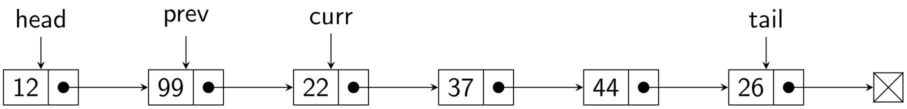
There are three subcases to consider:

1. The list is empty (i.e., head == null) so we can't remove.
   1. **Solution:** Return or throw an exception.
2. There is only one element in the list (i.e., head.next == null or size == 1)
   1. **Solution:** Just set the head equal to null.
3. There are more than two elements.
   1. **Solution:** We just went over that

Case 2: Remove from Back Psuedocode

```
Procedure RemoveFromBack(i)
    If(Head is Null)
        Return/Error
    ElseIf(Head.Next is Null)
        Head = Null
    Else
        Tmp = Head
        While(Tmp.Next.Next != Null)
            Tmp = Tmp.Next
        EndWhile
        Tmp.Next = Null
    EndIf
    Size−−
EndProcedure
```

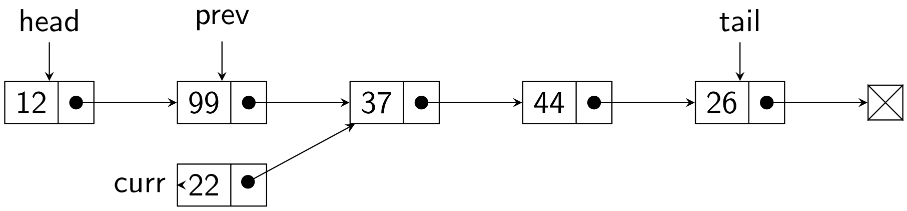
Case 3: Remove from Middle



Step 1: Search for and find a reference to the node you want to remove and the node that precedes it.

```
/* Step 1: Start at the beginning */
Prev = Null
Curr = Head

/* Walk through the list */
While(We havent found our thing/spot)
    Prev = Curr
    Curr = Curr.Next
EndWhile
```

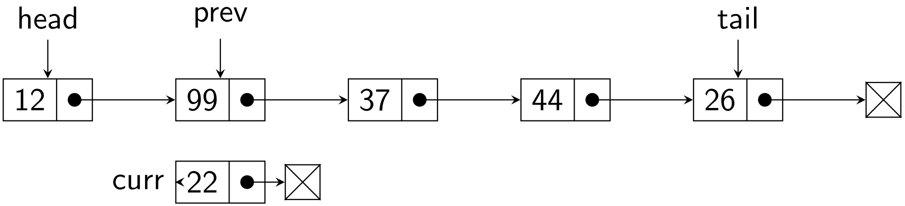
Case 3: Remove from Middle



Step 2: Update previous node's next node to be the current nodes next node.

```
/* Remove the element */
Prev.Next = Curr.Next // <—— Here
Curr.Next = Null
```

Case 2: Remove from Middle



Step 3: Null out the current nodes next node to fully remove it from the list

```
/* Remove the element */
Prev.Next = Curr.Next
Curr.Next = Null // <—— Here
```

Case 3: Remove from Middle Psuedocode

```
Procedure RemoveFromMiddle(i)
    /* Check if it's valid */
    If(i < 1 || i > size - 2)
        Return/Error
    EndIf

    Prev = Null
    Tmp = Head

    For(0 to i)
        Prev = Tmp
        Tmp = Tmp.Next
    EndFor

    Prev.Next = Tmp.Next
    Tmp.Next = Null
EndProcedure
```

Generic Linked List Template

```
public class GList<E>{

    public class GListNode<E>{
        /* Attributes */
    }

    public GList(){

    }


    public void addNode(E data){

    }

    public void displayList(){


    }
}
```

Worksheet: Generic Linked List Template

Off again to the worksheet to fill in the template we just saw!