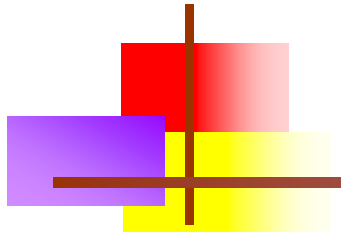
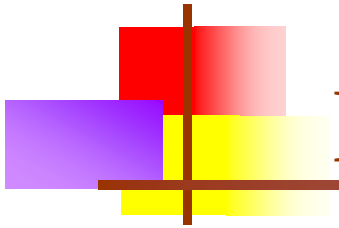
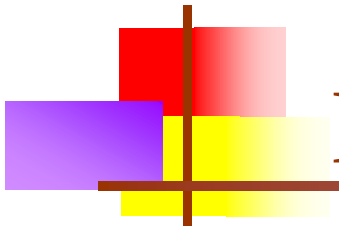


LINKED LISTS



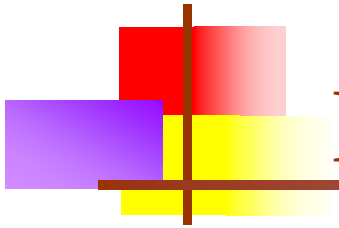
Issues with Arrays

- If the arrays are sorted, it takes O(log n) time to search for an element.
- Insertions into Arrays may take large number of operations, as it may involve creating a hole and shifting number of elements
- Deletions from the Arrays also may take large number of operations. To close the hole a number of elements need to be shifted left.
- Time complexity of insertions and deletions are of order O(n)
- Array size has to be fixed in the beginning.



Handling Lists

- Consider a list of integers
- { 16, 8, 10, 2, 34, 20, 12, 32, 18, 9, 3 }
- It can be thought of as an element 16 followed by another list
- 16 --{ 8, 10, 2, 34, 20, 12, 32, 18, 9, 3 }
- The second list can also be thought of an element 8 followed by a list
- 16--8-- { 10, 2, 34, 20, 12, 32, 18, 9, 3 }
- Thus any list can be thought of as an element followed by a list
- We can come up with a recursive definition of a linked list where each element is linked to the next one



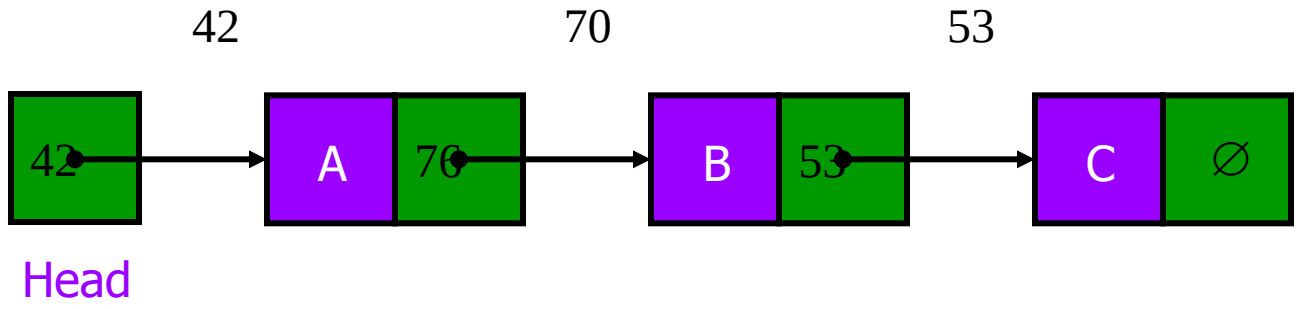
Linked List

- A linked list is a linear data structure where each element is a separate object.
- Each element of such a list needs to comprise of two items - the data and a reference to the next node.
- The object that holds the data and refers to the next element in the list is called a **node**.
- **data** component may actually consist of several fields
- For example in a linked list of students, the data could be student's name and Roll number.

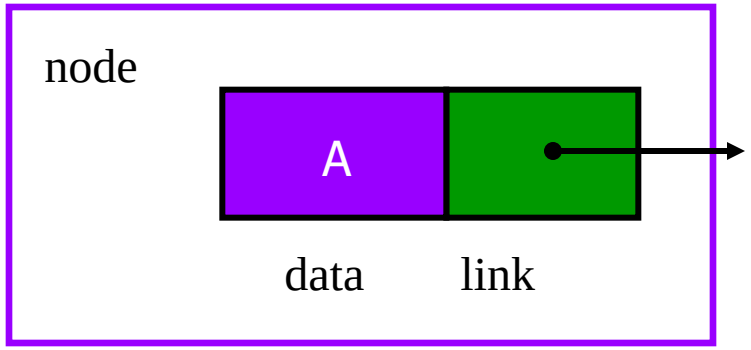
List stored in an array A[ ]

Memory Address	Array Index	List Contents
3200	A[0]	36
3202	A[1]	42
3204	A[2]	20
3206	A[3]	16
3208	A[4]	38
3210	A[5]	40
3212	A[6]	12
3214	A[7]	54
3216	A[8]	82

Linked Lists

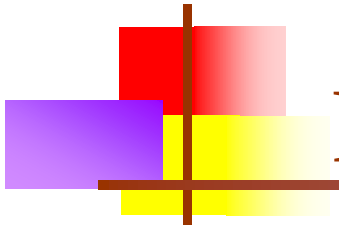


- A *linked list* is a series of connected *nodes*
- Each node contains at least
  - A piece of data (any type)
  - Link to the next node in the list
- *Head*: points to the first node
- Links generated by system
- The last node points to `NULL`



List items stored in a linked list

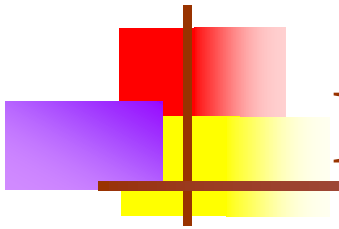
Memory Address	List contents	Next link
2020	36	450
450	42	3600
3600	20	4200
4200	16	4231
4231	38	760
760	40	5555
5555	12	X



Recursive Definition of List Node

```
public class Node {
        public int data;
        public Node link;

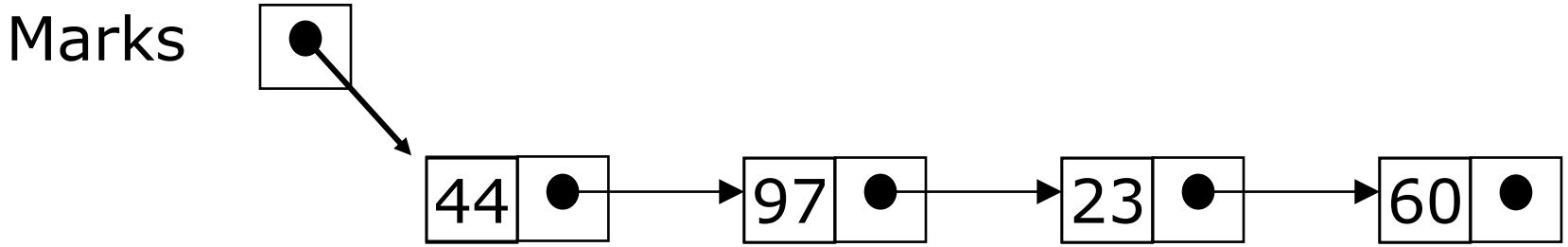
         Node(int n, Node p) {
                data = n;
                link = p;
         }
};
```



Extended Definition

```
public class Node {
        public int data;
        public Node link;

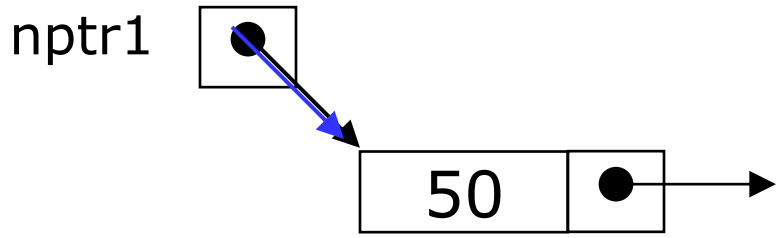
        Node() {                    // a simple node
                item = 0;
                link = null;
        }
         Node(int n) {              // a node with a given value
                data = n;
                link = null;
        }
         Node(int n, Node p) {      // a node with given value and reference
                data = n;
                link = p;
         }
};
```


Marks
44
97
23
60

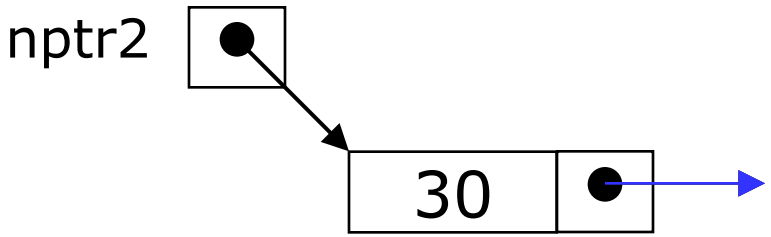
Creating two nodes

- Let us create a node with the statement:

```
Node nptr1 = new Node(50, null);
```



- Similarly create another node `Node nptr2 = new Node(30, null);`



- Let us now form a linked list with first node being the Front node

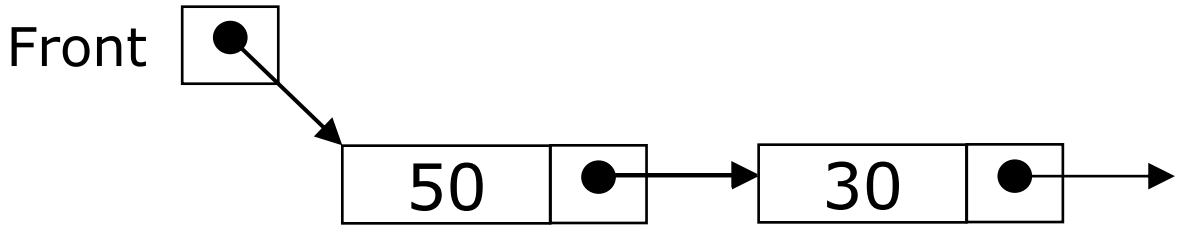
Linked list with first node followed by second node

- Front = nptr1

  Front.setLink (nptr2)

This statement will set the link value of first pointer with nptr2

- Since nptr2 points to second node, they get linked



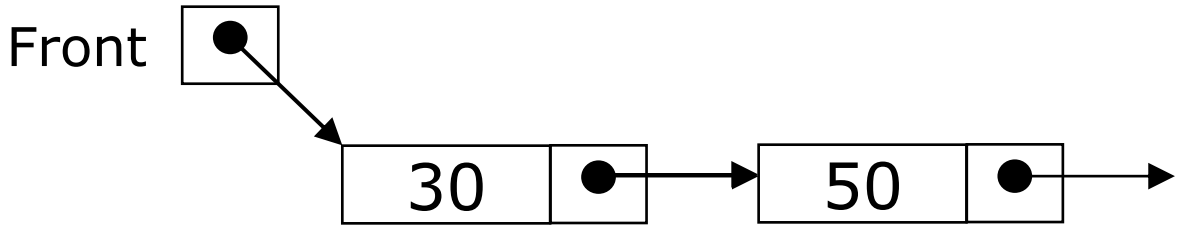
Linked list with second node followed by first node

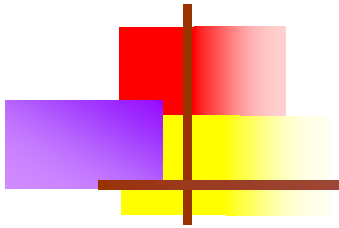
- Front = nptr2

  Front.setLink (nptr1)

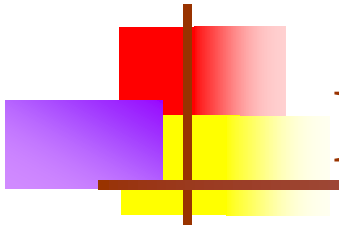
This statement will set the link value of second node to nptr1

- Since nptr1 points to second node, they get linked as shown





Inserting a node in a Linked List

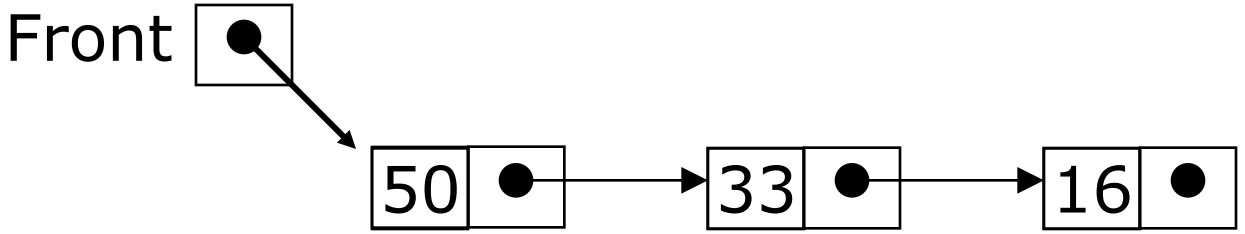


Inserting a node at front of the list

- If the Linked list is initially empty the new node becomes the Front node.
- Otherwise link the previous Front with new node and declare Front to be the inserted node.

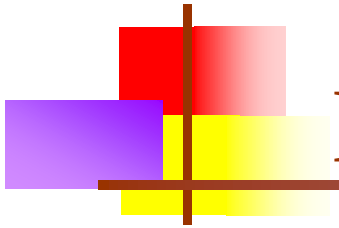
Inserting a node at front of the list

- Suppose we wanted to insert a node containing value 82 at front of the linked list



First form a new node with value 82 (and pointing to null)
Next we link node 82 to Front (node containing 50)

Finally, we declare node 82 to be the new Front node

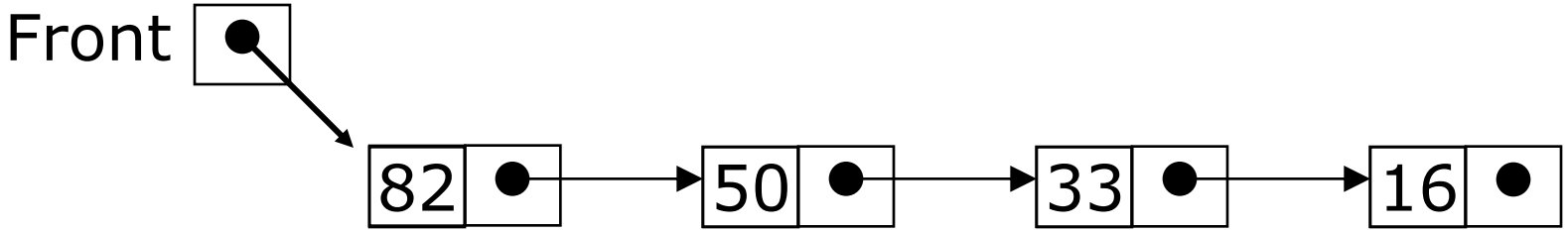


Inserting a node at front of the list

```
Node nptr = new Node (82, null);
if (Front == null)
        Front = nptr;
else
        nptr.setLink(Front);
Front = nptr;
```

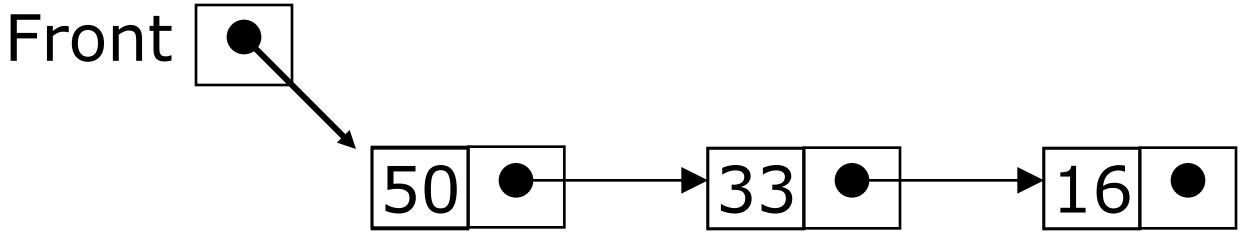
Modified linked list

- The linked list now takes the form:

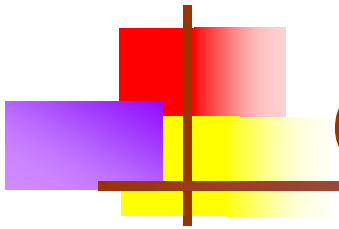


Inserting a node at rear of linked list

- Suppose we wanted to insert a node containing value 82 at the end of the linked list



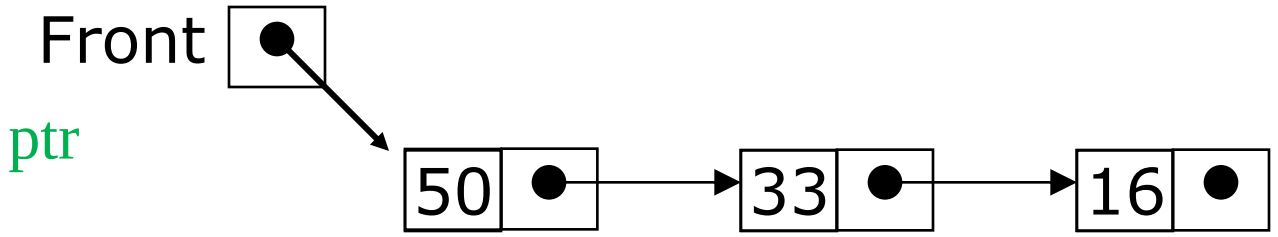
First form a new node with value 82 (and pointing to null)
Next we need to traverse through the list to reach the last node (pointing to null).



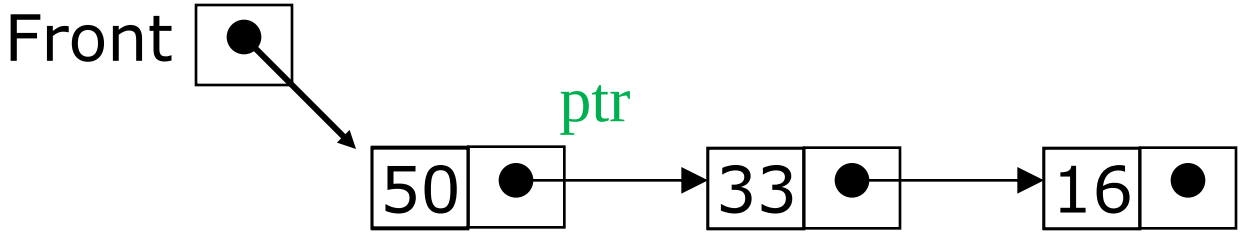
Code to link one node at rear of List

```
Node nptr = new Node (82, null);
ptr = Front;
while (ptr.getLink( )  != null)
{
    ptr = ptr.getLink( );
 }
 ptr.setLink(nptr);
```

Traversing the List with ptr

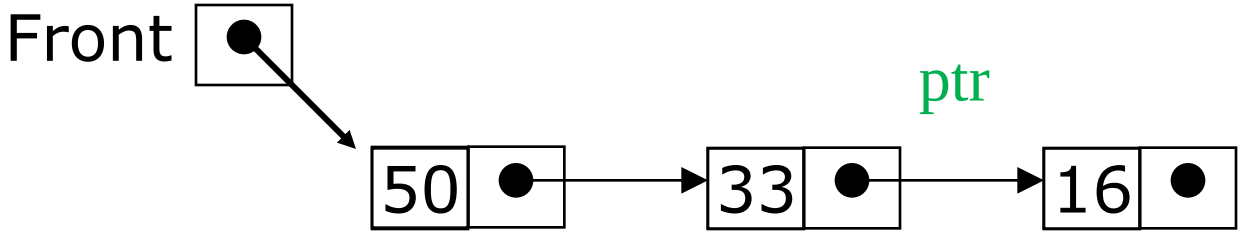


Traversing the List with ptr



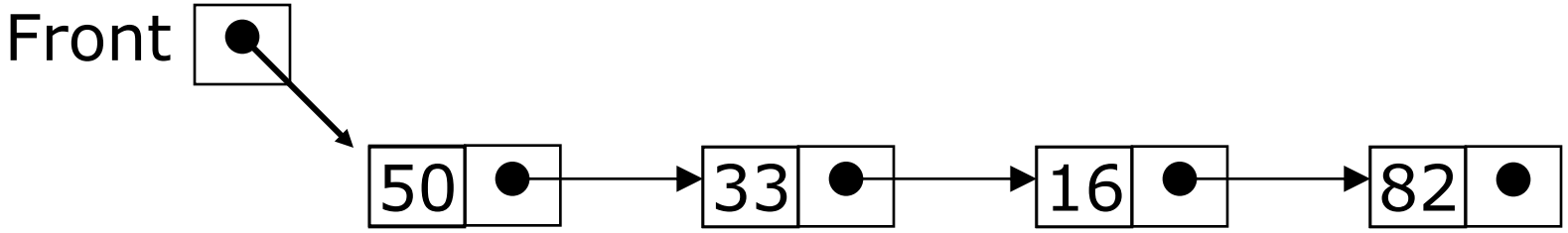
Traversing the List with ptr

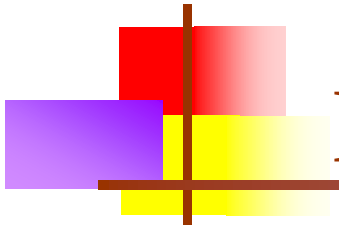
- Now we reach a node which has a null link. So we attach the new node here



Modified linked list

- The new linked list now takes the form:





Inserting a new node in an empty list

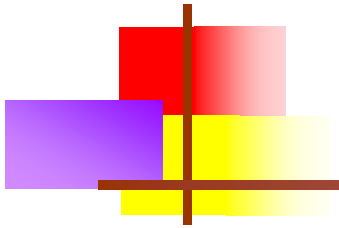
- But if the list was initially empty then the code needs modification.
- If Front was pointing to null, and then set Front to the new Node, as the new linked list will contain just this node

```
Node nptr = new Node (82, null);
if (Front == null)
        Front = nptr;
```

Inserting a new node in any list

- Now we combine both the parts so that insertion a node at rear of the list can be done for any type of linked list.

- ```
  Node nptr = new Node (82, null);
  if (Front == null)
          Front = nptr;
  else
  {
          ptr = Front;
          while (ptr.getLink()  != null)
                  ptr = ptr.getLink();
          ptr.setLink(nptr);
   }
  ```
  ```

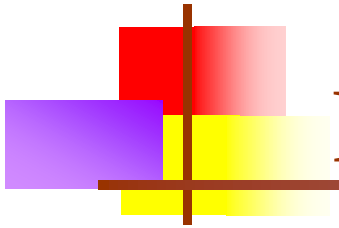


Printing and counting nodes of a List

Print contents of nodes

- Set a running pointer ptr to Front. Traverse through the list, and print contents of each node as long as link is not null.

```
Node ptr = Front;
while (ptr != null)
{
            System.out.print(ptr.getData()+ “  ”);
            ptr = ptr.getLink();
 }
```



Recursion and Linked lists

- Now we employ recursive methods to operate on Linked Lists.
- This is natural as linked lists themselves are recursive data structures.
- Recursive methods operating on linked lists are often simpler to write and easier to understand than their iterative counterparts.
- Later when we study Tree data structure, we shall see that some methods can only be written by using recursion.

Recursively print contents of nodes

- Set a running pointer ptr to Front. Print the data for this node and then recursively call the function with link to current pt.

```
Public  static void  printList (Node ptr )
{
      if (ptr != null)
      {
            System.out.print(ptr.getData()+ “  ”);
            printList( ptr.getLink() );  }
 }
```

Count number of nodes

- Set a running pointer ptr to Front. Start a counter. Traverse through the list, and increment the counter as long as link is not null.

```
Node ptr = Front;
int count = 0;
while (ptr != null)
{
            count++;
            ptr = ptr.getLink();
 }
  System.out.println("nodes ="+ count);
```

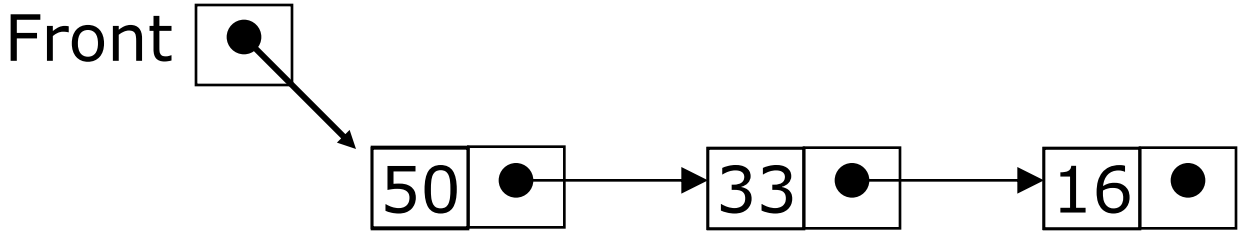
Recursively count number of nodes

- If pointer is not null, add **1** plus nodes in the remaining list (list starting with the next node. When the list reaches the end, ptr reaches null, which adds zero to the count.

```
Public  static int countList (Node ptr )
{
      if (ptr == null)
          return 0;
      else
           return 1 + countList (ptr.getLink( ));
}
```

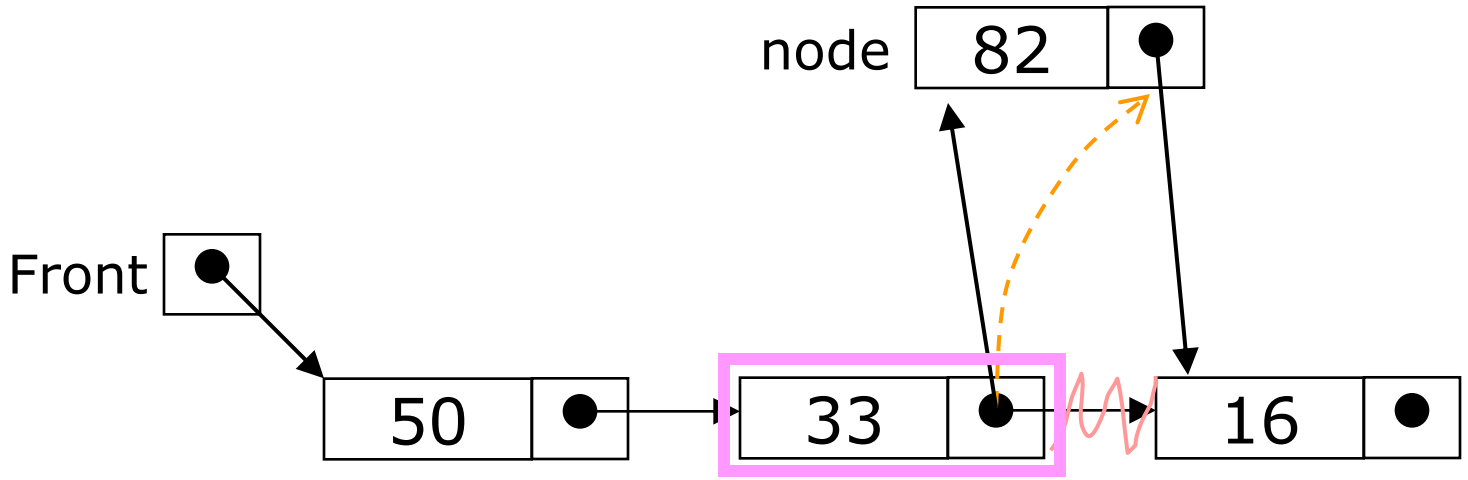
Inserting a node at a specific position in the List

- Assume node 50 is at position 1. Suppose we wanted to insert a node containing value 82 at position 3, that means after node containing 33.
- This would involve breaking the list and setting two links to new values.



- First form a new node with value 82 (and pointing to null).
- Reach position 2. Get link of 33 and use it to set link of 82, so that 82 and 16 get linked. Next set link of 33 to address of new node 82.

Inserting after (animation)



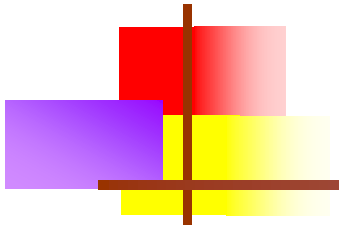
Find the node you want to insert after

First, copy the link from the node that's already in the list

Then, change the link in the node that's already in the list

Inserting a node at a specific position in the List

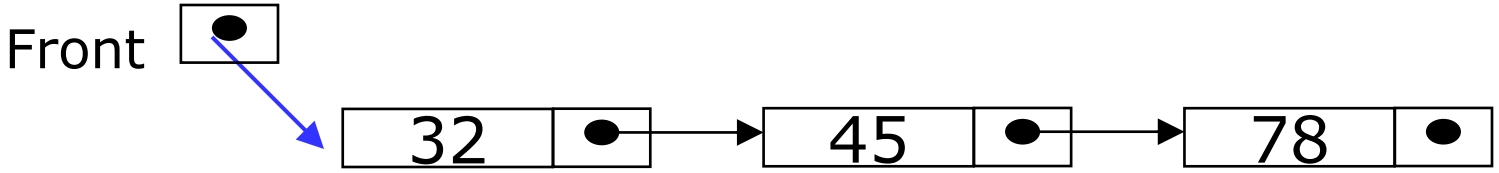
- Position of first node is assumed 1.
-
```
public void insertAtPos(int val , int pos)      {
        Node nptr = new Node(val, null);
        Node ptr = front;
        pos = pos - 1 ;
        for (int i = 1; i < size; i++)          {
            if (i == pos)                   {
                Node tmp = ptr.getLink() ;
                ptr.setLink(nptr);
                nptr.setLink(tmp);
                break;
            }
            ptr = ptr.getLink();
        }
```

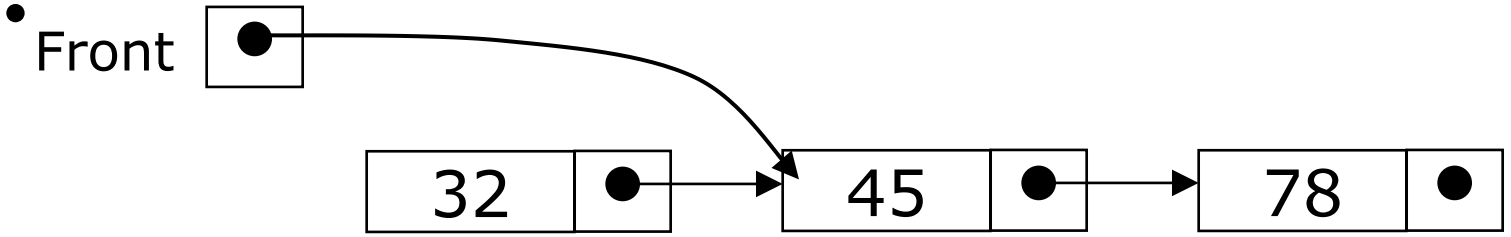


Deletion in Linked Lists

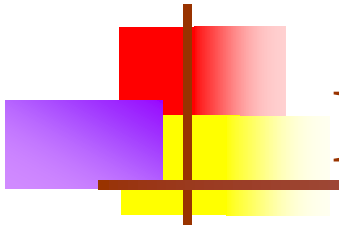
Deleting the first node

- To delete the first element, change the link in the header





- Node ptr = Front.getLink();
  Front = ptr;



Deleting the last node

- Set pointer prev (previous node) to first node of the list.
- Set pointer cur (current node) to second node of the list.
- Traverse the list till cur reaches the last node
- set link for prev to null, so that last node is not reachable.

```
Node prev = Front;
Node cur = Front.getLink();
while ( cur.getLink() != null) {
        prev = cur;
        cur = cur.getLink();
 }
 prev.setLink(null);
```

Add code to handle single node in the list.