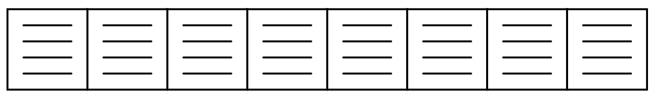
Static vs. Dynamic Data Structures

- Static data structures such as arrays allow
  - fast access to elements
  - expensive to insert/remove elements
  - have fixed, maximum size

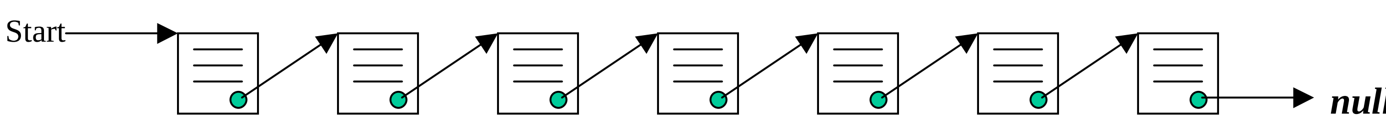
- Dynamic data structures such as linked lists allow
  - fast insertion/deletion of element
  - but slower access to elements
  - have flexible size

Linked Lists

- So far, if we required an unknown number of some kind of data structure, we would define an array with some maximum number of elements.



- This may use a lot of memory needlessly and also makes the data quite hard to manipulate.
- A linked list is a number of structures (i.e nodes) which are connected together using pointers.



- Each structure contains at least one pointer that points to another structure in the linked list.
- The pointer of the final structure in the list points to ***null.***

Creating a linked list

- We may represent a linked list node in Java by creating a class of type node :

```
class Node {
   int data = 0;          // data in each node
   Node next = null;   // pointer to next node in list
}
```

- This type of data structure is considered a recursive data structure since it calls itself.
- We can create nodes in our code in the same way that we create new objects by using the ***new*** operator:

- Example :

```
Node p = null;       // declare a node object
p = new Node();      // create the node
p.data = 100;        // set data in node to 100
p.next = NULL;   // set next pointer to point to null
```

Manipulating the Linked List

- In order to access the elements of the linked list, a pointer to the first structure is all that is required.

- We can add a node to the linked list by using the ***new*** operator and setting the pointer in the previous node to point to the new node.

- We can recognise the last node in the list by the fact that its “next” pointer points to a *null* or else by having an end node pointer as well as a start node pointer.

- To remove a node from the linked list, simply repoint the pointer in the structure which points to node that is to be removed (plus any other relevant pointers).

Example

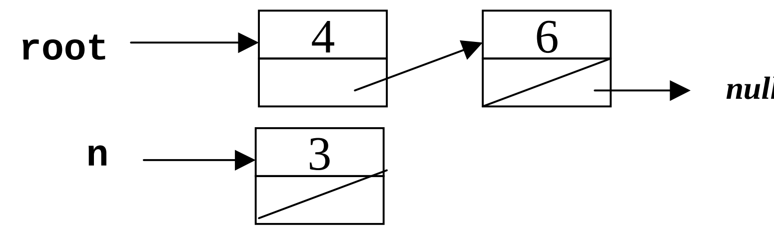
Draw the picture that results from the following code:

```
Node front;
Node temp;

front = new Node();
front.data = 1;

front.next = new Node();
front.next.data = 2;
front.next.next = null;

temp = front;
front = front.next;
```

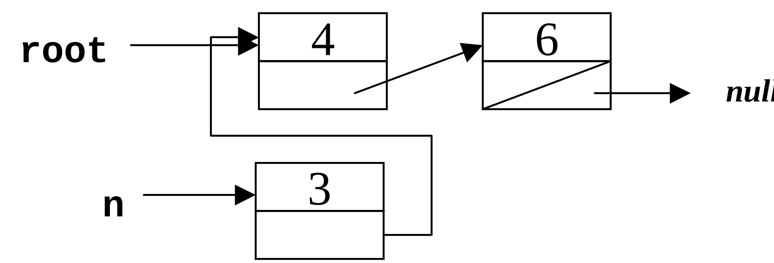
Inserting Nodes to Linked Lists

Inserting an element at the head of a linked list:

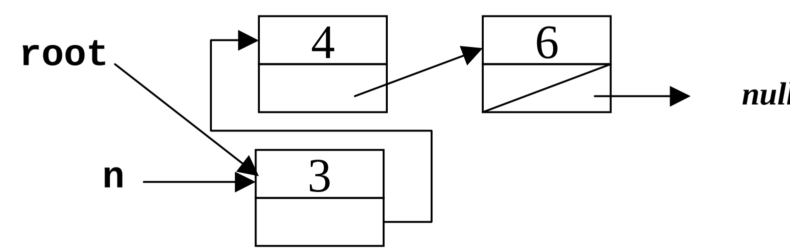
1) Create a new node



2) Set the link of the new node



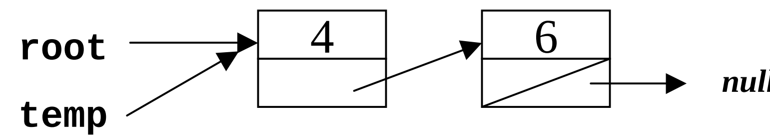
3) Set **`root`** so that it points to the new node



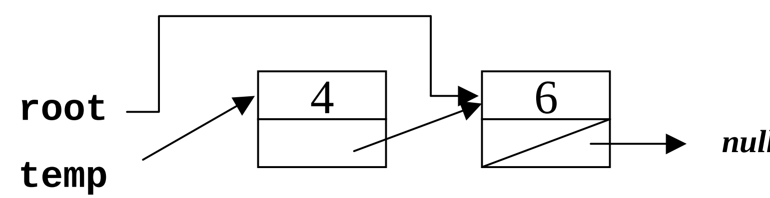
Removing Nodes From Linked Lists

Removing an element from the linked list:

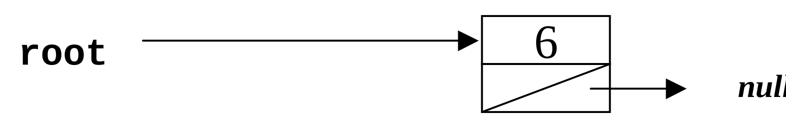
1) Create a temporary pointer



2) Set root to the new head

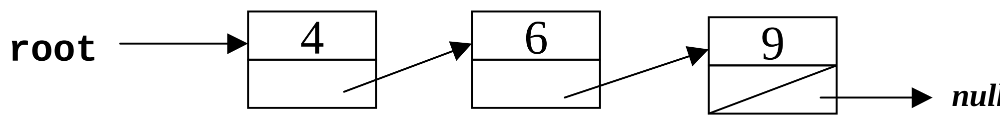


3) Delete old node



Traversing Linked Lists

Traversing a linked list:



Start with **`root`** and 'hop' from node to node until **`next`** points to ***null***:

```
Node c = root;
while (c != null)
{
   PerformOperation(c);
   c = c.next;
}
```

Traversing Linked List Recursively

- Example : Printing data in a Linked List

- What is the base case?
  - The base case : if *null* is reached

- What is the recursive case?
  - The recursive case : printList(head.next);

```
int printList(node head)
{
    if (head == null) {            // base case: list is empty
        return 0;
    else {                         // recursive case
        System.out.print(head.data);
        return printList(head.next);
    }
}
```

Sample Linked List ADT

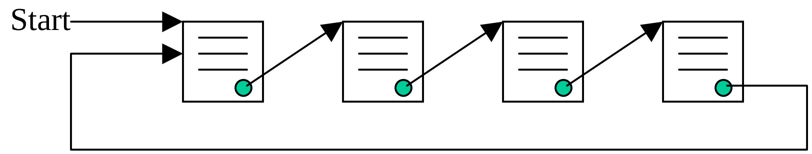
List specification

```
class Node
{
   int data = 0;
   Node next = null;
}
clas LinkedList
{
   Node head = null;

   Node InitList(LinkedList list){ … … … }
   void AddHead(LinkedList list, int elem) {… … …}
   void AddTail(LinkedList list, int elem) {… … …}
   int  size(LinkedList list) {… … …}
   boolean inList(LinkedList l, int elem) {… … …}
   void RemoveHead(LinkedList list) {… … …}
   ... (You can think of other useful functions)
}
```

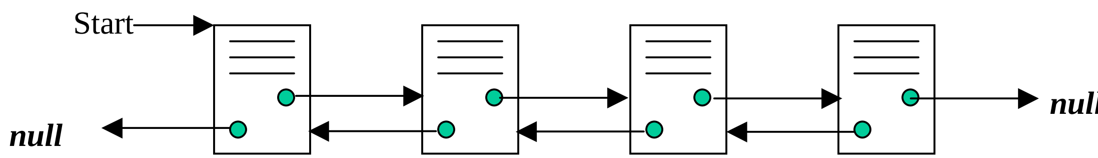
Different Linked Lists

- So far we have only been looking at singly linked lists

- There are 3 other types of linked lists:
  - circular linked lists :



  - doubly linked lists :



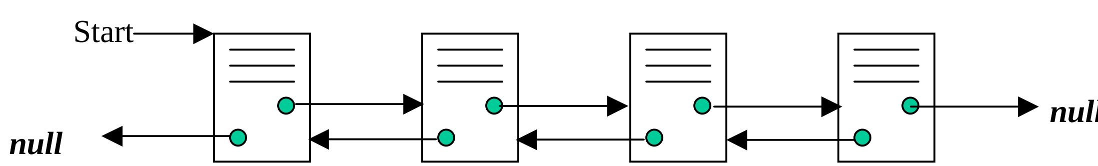
- And of course, circular doubly linked lists

Doubly Linked List

- One shortcoming of singly linked lists is that we can only move forwards through the list

- A doubly linked list is a linked list which also has pointers from each element to the preceding element

- Doubly linked lists make manipulation of lists easier



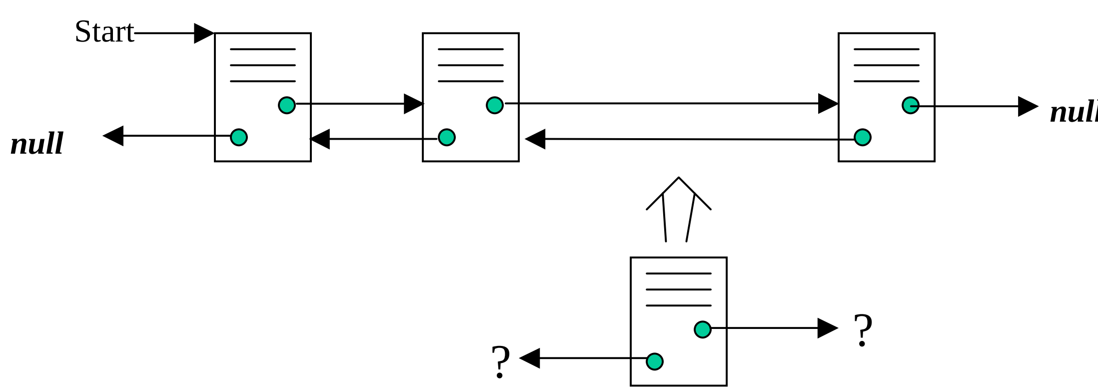
Doubly Linked Lists in Java

- We need only add an extra line to make our list doubly linked.
- A node in a doubly linked list would look something like this:

```
class Node
{
   int data = 0;
   Node next = null;
   Node prev = null;          // note the extra line
}
```

Inserting to a DLL

Inserting to a doubly linked list is the almost the same as inserting to a singly linked list. The only difference is that you have an extra “previous” pointer to take care of as well as the “next” pointer.



A Stack Implementation

In the following case study we are going to rewrite the stack implementation using a linked list.
Remember the stack interface:

```
void InitStack(Stack st)
void Push(Stack st, int elem)
int  Pop(Stack st)
int  Top(Stack st)
```

Stack specification

A Stack Implementation (2)

```
class Node
{
   int data = 0;
   Node next = null;
   Node prev = null;
}
```

Class **`Node`** represents an element in the stack.

```
class Stack
{
   Node Bottom = null;
   Node Top = null;

   << stack methods >>
}
```

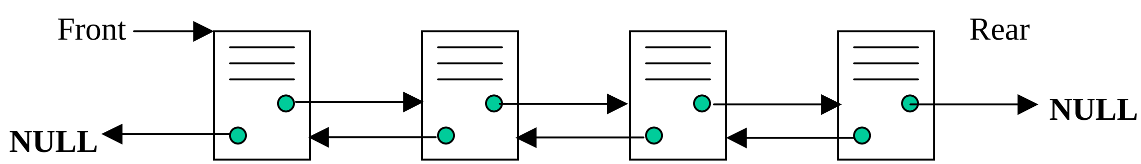
Top points to the top element of the stack. *Bottom* keeps track of the location of the bottom of the stack

A Stack Implementation (3)

```
void Init(Stack st)
{
   st.Top = null;
   st.Bottom = null;
}

void Push(Stack st, int elem)
{
    Node new_node = null;
    new_node = new Node();
    new_node.data = elem;
     // Set up the pointers
    st->Top.next = new_node;
    new_node.next = null;
    new_node.prev = st.Top;
    st.Top = new_node;
}
```

A Queue implementation

A queue can also be easily implemented using a doubly linked list:



I’ll leave this as an exercise for you to try.