CS 112 – Introduction to Computing II

Wayne Snyder
Computer Science Department
Boston University

Today

- Introduction to Linked Lists
- Stacks and Queues using Linked Lists

Next Time

- Iterative Algorithms on Linked Lists
- Reading: Notes on Iteration and Linked Lists (on web)

Representing Sequences of Data

The simplest "geometrical" arrangement of data, we have seen, is in a linear sequence or list:

3 1 4 1 5 9 2 6 5 3

and we have see there is a simple representation for a linear sequence in an array:

	0	1	2	3	4	5	6	7	8	9
A:	3	1	4	1	5	9	2	6	5	3

Representing Sequences of Data

Computer Science

	0	1	2	3	4	5	6	7	8	9
A:	3	1	4	1	5	9	2	6	5	3

The **advantages** of an array are:

Simplicity: Easy to define, understand, and use

Efficiency: Compact representation in computer memory, every element can be accessed in the same amount of time ("Random Access") quickly.

The **disadvantages** of an array are basically that it is **inflexible**:

The **size** is fixed and must be **specified in advance**; must be reallocated if resized;

To **insert** or **delete** an element at an arbitrary position, you must move elements over!

Data in Computer Memory

Computer Science

The reason that arrays are so efficient is that basically computer memory (“Random Access Memory”) is a huge array built in hardware; each location has a address (= index of array) and holds numbers:

RAM:

0	2
1	5
2	**13**
3	23
4	-34
5	232
6	2
7	6
8	**3**
9	**10**
10	-78
11	3
12	4
13	5
14	5
15	-1
16	2

Computer instructions say things like:

“Put a 3 in location 8:”

RAM[8] = 3;

“Add the numbers in locations 8 and 9 and put the sum in location 2:”

RAM[2] = RAM[8] + RAM[9]

This is why arrays are so common and so efficient: RAM is just a big array!

Access time = about 10^{-7} secs

Data in Computer Memory (review)

Computer Science

When you create variables in Java (or any programming language), these are "nicknames" or shortcut ways of referring to locations in RAM:

RAM:

These "shortcut" names for primitive types can not change during execution.

	Address	Value
	0	2
	1	5
z:	2	13
	3	23
	4	-34
	5	232
	6	2
	7	6
x:	8	3
y:	9	10
	10	-78
	11	3
	12	4
	13	5
	14	5
	15	-1
	16	2

```
int x;     // same as RAM[8]
int y;     // same as RAM[9]
int z;     // same as RAM[2]

// now the previous computation
// would be

x = 3;
y = 10;
z = x + y;
```

When we draw our diagrams of variables, we are really just giving a shortcut view of RAM without the addresses:

x:	3

Objects/Classes in Computer Memory (review)

BUT Reference Types (arrays, Strings, objects – anything you can use the word `new` to create new instances of) are **references** or **pointers** to their values: **they store the location of the value, not the value itself.**

	0	2
	1	5
z:	2	**13**
	3	23
A:	4	10
	5	232
	6	2
P:	7	14
x:	8	**3**
y:	9	**10**
	10	11
	11	3
	12	4
	13	8
	14	5
	15	-1
	16	2

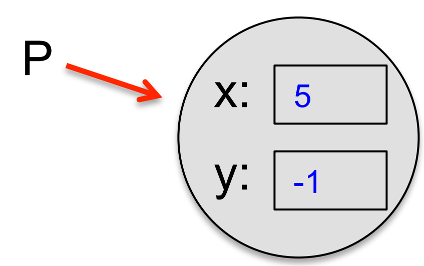
```
int x;
int y;
int z;
```

```
class Point {
    int x;
    int y;
}
```

```
int [] A = { 11, 3, 4 };
Point P = new Point(5, -1);
```

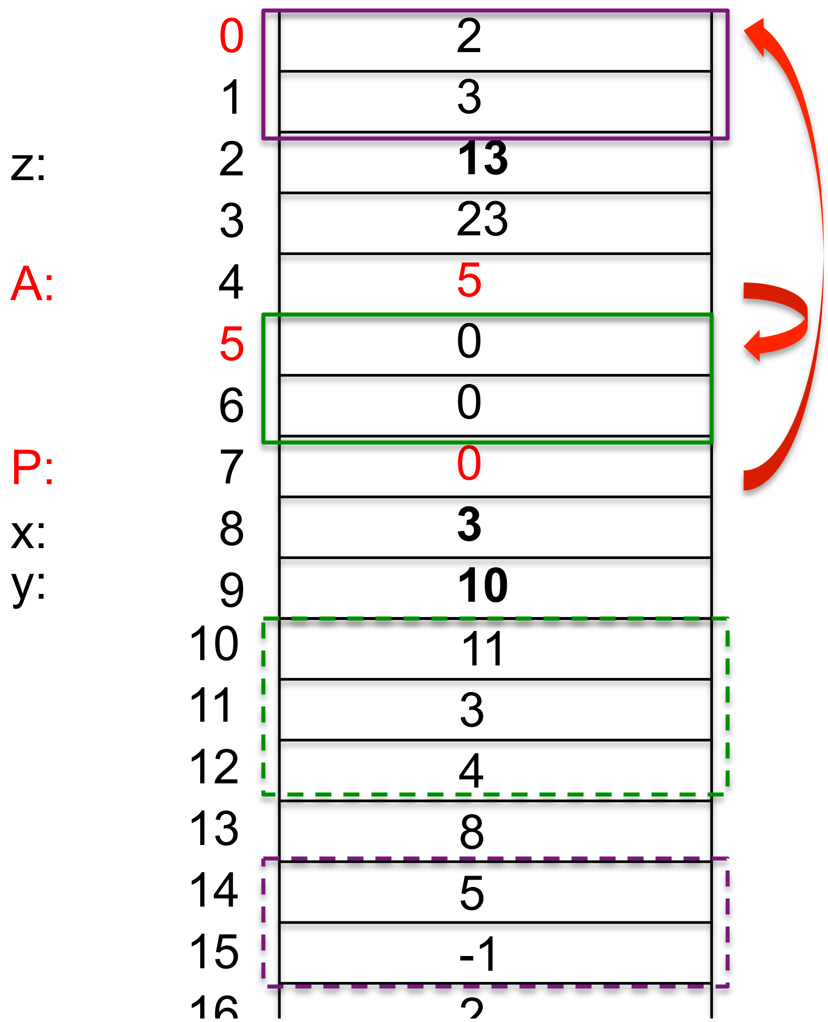
A

A[0]:	11
A[1]:	3
A[2]:	4

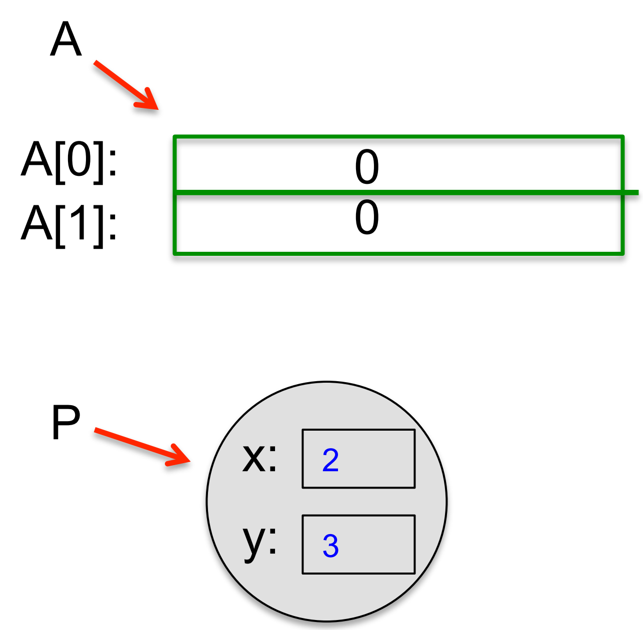


Objects/Classes in Computer Memory (review)

Now we can change the “meaning” of the reference variable by assigning it a new location; in fact, `new` returns the new location, which is stored in the reference variable as its “value.”




```
.....
int [] A = { 11, 3, 4 };
Point P = new Point(5, -1);

A = new int[2];
P = new Point(2,3);
```



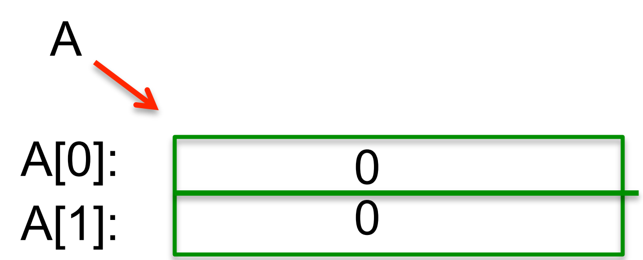
Objects/Classes in Computer Memory (review)

Computer Science

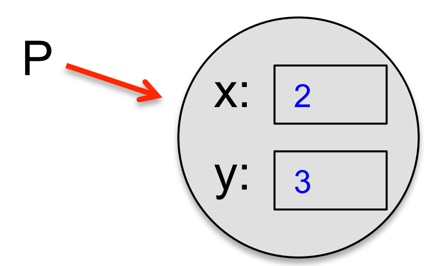
Now we can change the "meaning" of the reference variable by assigning it a new location; in fact, `new` returns the new location, which is stored in the reference variable as its "value."

	0	2
	1	3
z:	2	**13**
	3	23
A:	4	5
	5	0
	6	0
P:	7	0
x:	8	**3**
y:	9	**10**
	10	11
	11	3
	12	4
	13	8
	14	5
	15	-1
	16	2

```
.....
int [] A = { 11, 3, 4 };
Point P = new Point(5, -1);

A = new int[2];
P = new Point(2,3);
```



Old objects are "garbage" and the memory will be reclaimed by the "garbage collector" and reused.

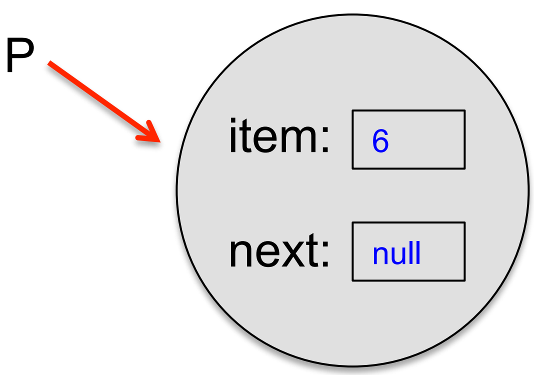


Linked Lists in Computer Memory

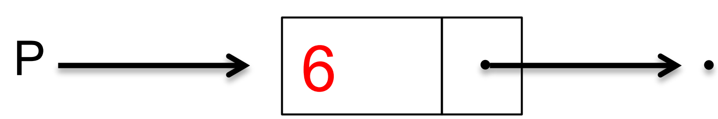
This FREES US from having to store items in contiguous locations, as in an array.
A **linked list** is a way to do this, with a completely different set of tradeoffs from the array representation.

A linked list is a collection of **nodes**, which are just objects which hold a data item and a pointer to another node just like itself:

```
class Node {
    int item;          // data item
    Node next;         // pointer to a Node
}

Node P = new Node();
P.item = 6;
P.next = null;
```



A more compact diagram would be:

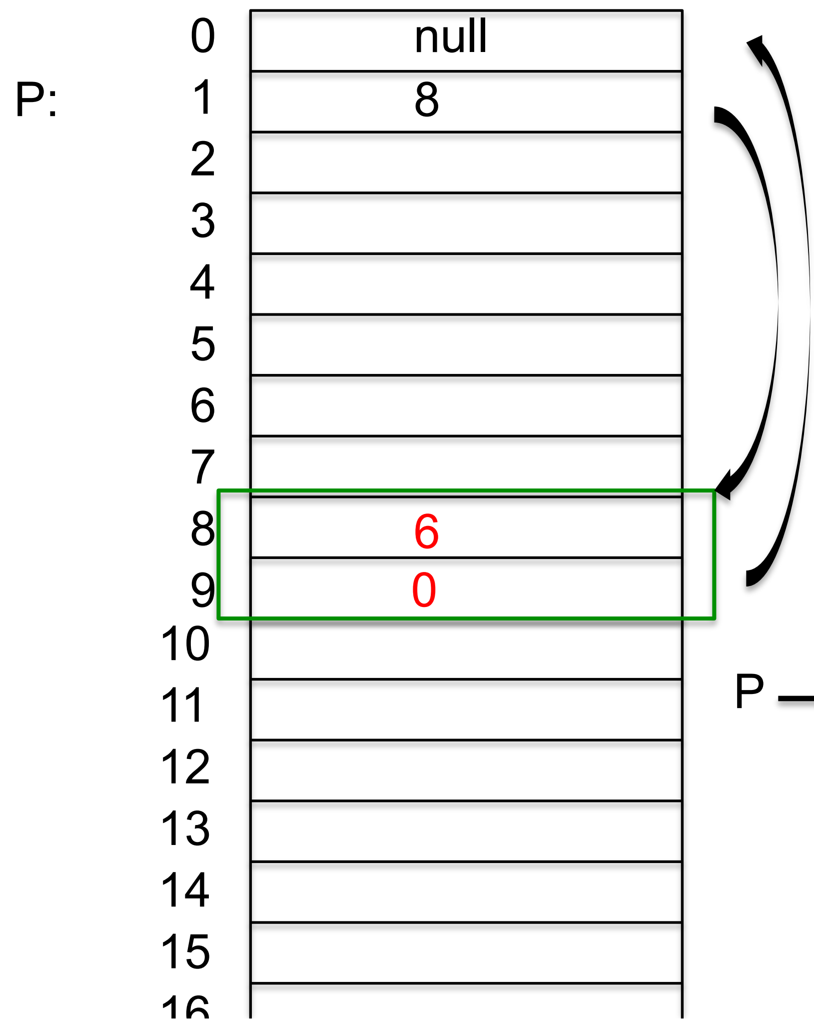


• = null

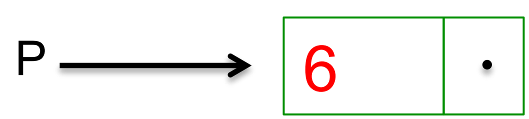
null (actually location 0) is a special location indicating “nothing there”.

Linked Lists in Computer Memory

These nodes are just classes, like any other, and are stored in RAM just as before:




```
class Node {
      int item;          // data item
      Node next;         // pointer to a Node
}

Node P = new Node();
P.item = 6;
P.next = null;
```

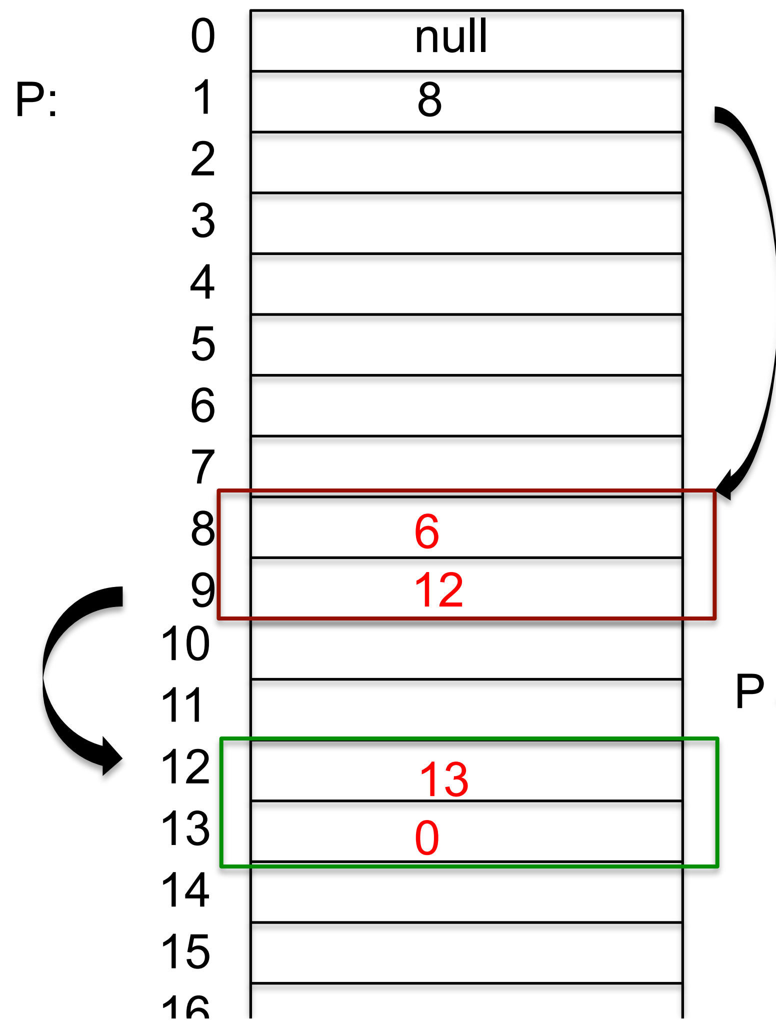


• = 0 = null

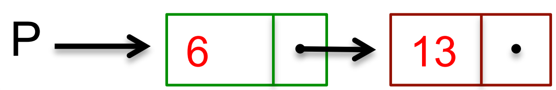
Linked Lists in Computer Memory

Computer Science

However, separating the name from the physical location in memory using references gives us a **huge advantage**, in that now we can create **sequences which do not have to be in contiguous locations:**

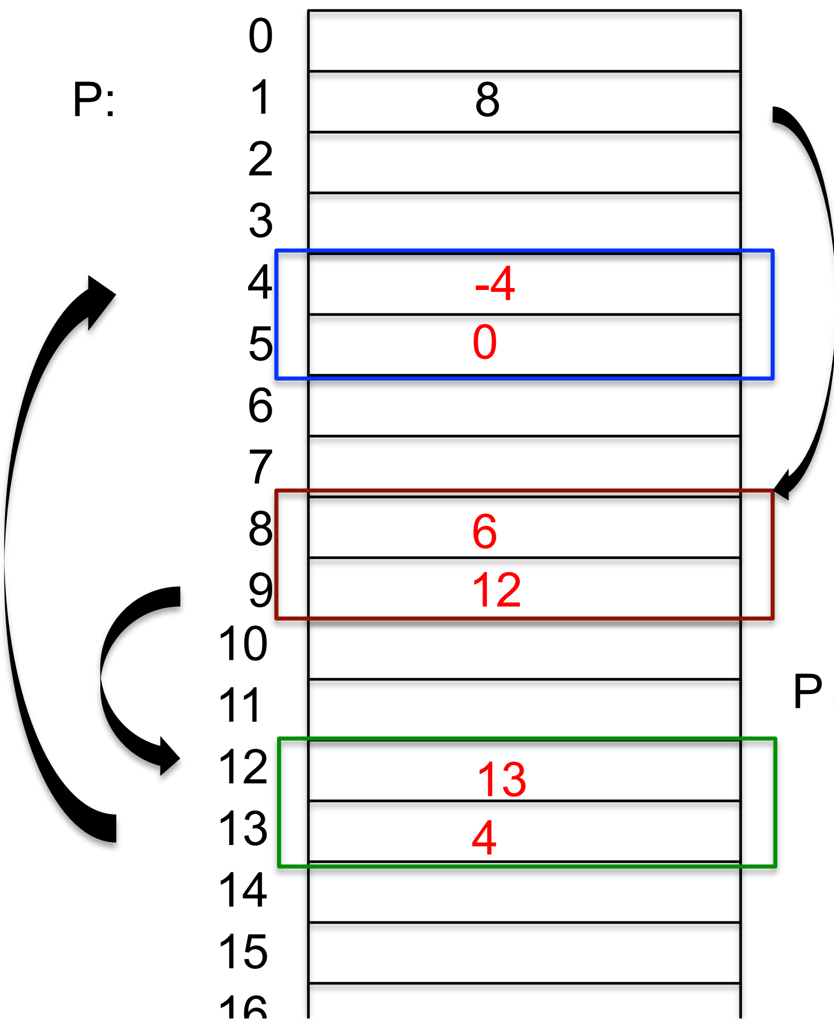



```
class Node {
      int item;            // data item
      Node next;       // pointer to a Node
      public Node(int n, Node p) { // constructor
           item = n;  next = p;
      }
}

Node P = new Node(6, null);
P.next = new Node(13, null);
```

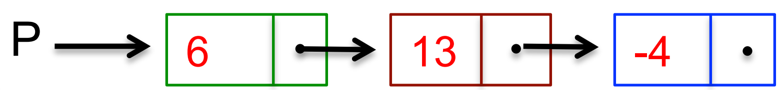


Linked Lists in Computer Memory

These linked lists are very, very common in all kinds of applications!




```
class Node {
     int item;           // data item
     Node next;       // pointer to a Node
     public Node(int n, Node p) { // constructor
          item = n;  next = p;
     }
}

Node P = new Node(6, null);
P.next = new Node(13, null);
P.next.next = new Node(-4, null);
```




```
// Or could write:

Node P = new Node(6, new Node(13, new Node(-4, null)));
```

Linked Lists

Be **SURE** that you understand the different uses of a variable in an assignment statement:

```
int n = 4;
int m = 6;
n = m;
```

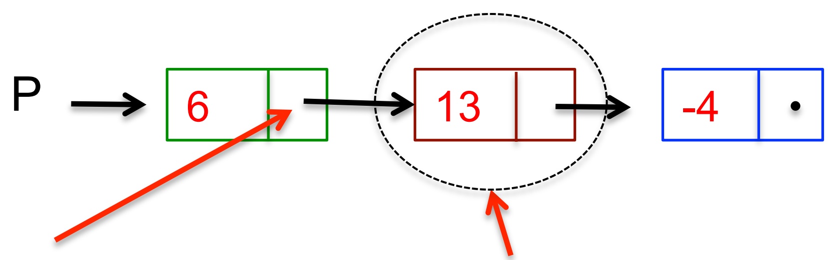
location to store value

the value to b assigned

This explains how to interprete occurrences of the variable "P.next":

`P.next = .....` // This is the field `next` in the object P, which stores a location

`.... = P.next;` // This is the value or meaning or referent of `P.next`, the node it // points to



P.next: this is field next in Node P

This is the Node referred to by P.next

Example:

p.next = p.next.next;

p.next = (p.next).next;

compare:

n = n + 1;

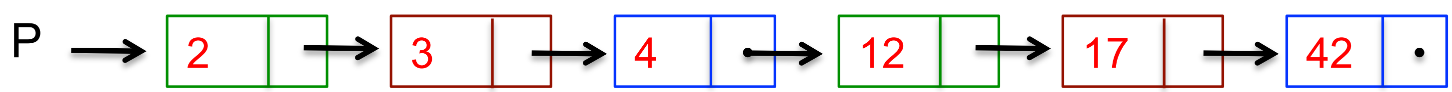
Linked Lists

The **advantage** of a linked lists is its flexibility: it can be any length, you don't have to find a contiguous sequence in memory for whole list, and you can add/delete an element anywhere by just changing some pointers.

The **disadvantages** of a linked list are that (1) each "slot" requires a call to allocate memory, and (2) it must use sequential access:

To find any particular item, you must run through all previous items in the sequence.

For example, you CAN'T USE binary search!



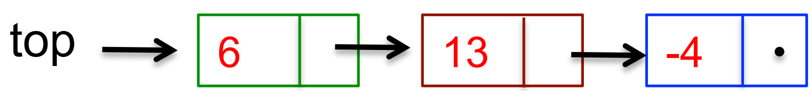
Thus, sorting a linked list is of limited use: at most, it can tell you when to stop when doing linear search (if you search for 5, you can stop when you get to the 12).

Linked Lists: Stacks and Queues

Clearly, linked lists are a great way to implement stacks and queues, since there is no possibility of overflow! The only question is: which direction should the list go?

Remember, you can only “chain along” one direction in a LL (following the arrows), **so removing an element and moving a pointer to the next element must take place at the front of the list** (e.g., pop in a Stack and dequeue in a Queue):

Stack:



6
13
-4

```
Node top = null;

int pop() {
        int tmp = top.item;
        top = top.next;
        return tmp;
}
```

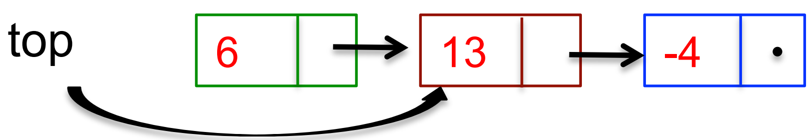
```
void push(int n) {
    top = new Node(n, top);
}

boolean isEmpty() {
    return ( top == null );
}

int size() {
     return length(top);    // counts nodes
}
```

Linked Lists: Stacks and Queues

Clearly, linked lists are a great way to implement stacks and queues, since there is no possibility of overflow! The only question is: which direction should the list go?

Remember, you can only “chain along” one direction in a LL (following the arrows), **so removing an element and moving a pointer to the next element must take place at the front of the list** (e.g., pop in a Stack and dequeue in a Queue):

Stack:



13
-4
------

tmp: 6

```
Node top = null;

int pop() {
        int tmp = top.item;
        top = top.next;
        return tmp;
}
```

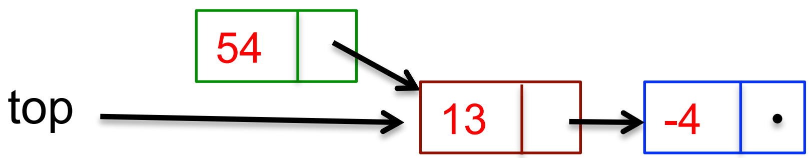
```
void push(int n) {
    top = new Node(n, top);
}

boolean isEmpty() {
    return ( top == null );
}

int size() {
    return length(top);    // counts nodes
}
```

Linked Lists: Stacks and Queues

Clearly, linked lists are a great way to implement stacks and queues, since there is no possibility of overflow! The only question is: which direction should the list go?

Remember, you can only “chain along” one direction in a LL (following the arrows), **so removing an element and moving a pointer to the next element must take place at the front of the list** (e.g., pop in a Stack and dequeue in a Queue):

Stack:



push(54);

13
-4

```
Node top = null;

int pop() {
        int tmp = top.item;
        top = top.next;
        return tmp;
}
```

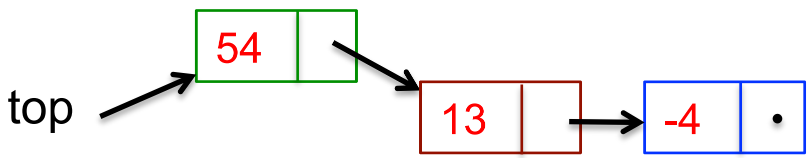
```
void push(int n) {
    top = new Node(n, top);
}

boolean isEmpty() {
    return ( top == null );
}

int size() {
     return length(top);    // counts nodes
}
```

Linked Lists: Stacks and Queues

Clearly, linked lists are a great way to implement stacks and queues, since there is no possibility of overflow! The only question is: which direction should the list go?

Remember, you can only “chain along” one direction in a LL (following the arrows), **so removing an element and moving a pointer to the next element must take place at the front of the list** (e.g., pop in a Stack and dequeue in a Queue):

Stack:



54
13
-4

```
Node top = null;

int pop() {
        int tmp = top.item;
        top = top.next;
        return tmp;
}
```

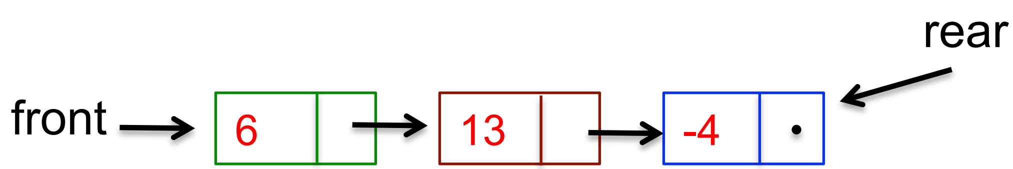
```
void push(int n) {
    top = new Node(n, top);
}

boolean isEmpty() {
    return ( top == null );
}

int size() {
     return length(top);    // counts nodes
}
```

Linked Lists: Stacks and Queues

For a queue, pop is the same as dequeue, and since you can only remove nodes from the front of a linked list (without traversing it), we have to follow this rule:

In a stack arrows go down, in a queue, from front to back….

Queue:




```
int dequeue() {
        int tmp = front.item;
        front= front.next;
        if( front == null )      // Q now empty
            rear = front
        return tmp;
}
```

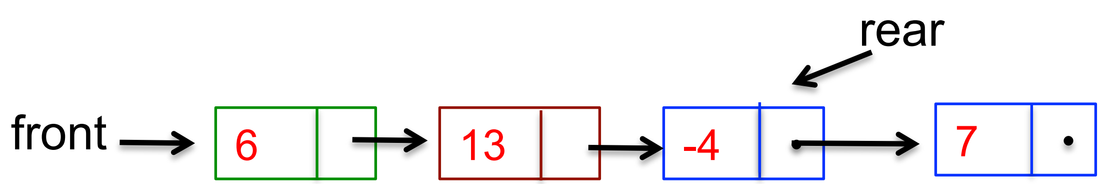
```
void enqueue(int n) {
     if( front == null) {      // Q was empty
        front = rear = new Node( n );
     else {
         rear.next = new Node( n );
         rear = rear.next;
     }
}

boolean isEmpty() {
    return ( front == null );
}

int size() {
     return length(top);    // counts nodes
}
```

Linked Lists: Stacks and Queues

For a queue, pop is the same as dequeue, and we must be able to get to the next nodes, so again, we must make sure we get the arrows in the right direction!

Summary: in a stack arrows go down, in a queue, from front to back….

Queue:




```
int dequeue() {
        int tmp = front.item;
        front= front.next;
        if( front == null )       // Q now empty
            rear = front
        return tmp;
}
```

```
void enqueue(int n) {
     if( front == null) {      // Q was empty
        front = rear = new Node( n );
     else {
         rear.next = new Node( n );
         rear = rear.next;
     }
}

boolean isEmpty() {
    return ( front == null );
}

int size() {
     return length(top);    // counts nodes
}
```

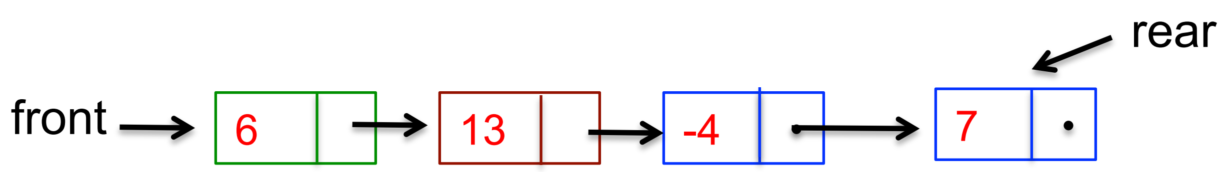
Linked Lists: Stacks and Queues

Computer Science

For a queue, pop is the same as dequeue, and we must be able to get to the next nodes, so again, we must make sure we get the arrows in the right direction!

Summary: in a stack arrows go down, in a queue, from front to rear….

Queue:



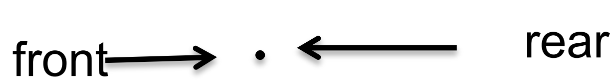

```
int dequeue() {
        int tmp = front.item;
        front= front.next;
        if( front == null )       // Q now empty
            rear = front
        return tmp;
}
```

```
void enqueue(int n) {
     if( front == null) {       // Q was empty
        front = rear = new Node( n );
     else {
         rear.next = new Node( n );
         rear = rear.next;
     }
}

boolean isEmpty() {
    return ( front == null );
}

int size() {
     return length(top);    // counts nodes
}
```

Linked Lists: Stacks and Queues

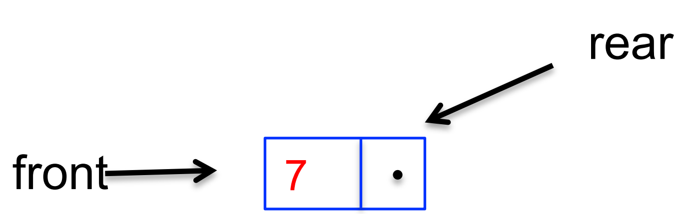
An empty stack is just represented by an empty list

and a queue by an empty list with two pointers:




```
void enqueue(int n) {
    if( front == null) {      // Q was empty
        front = rear = new Node( n );
    else {
        rear.next = new Node( n );
        rear = rear.next;
    }
}
```

enqueue(7);

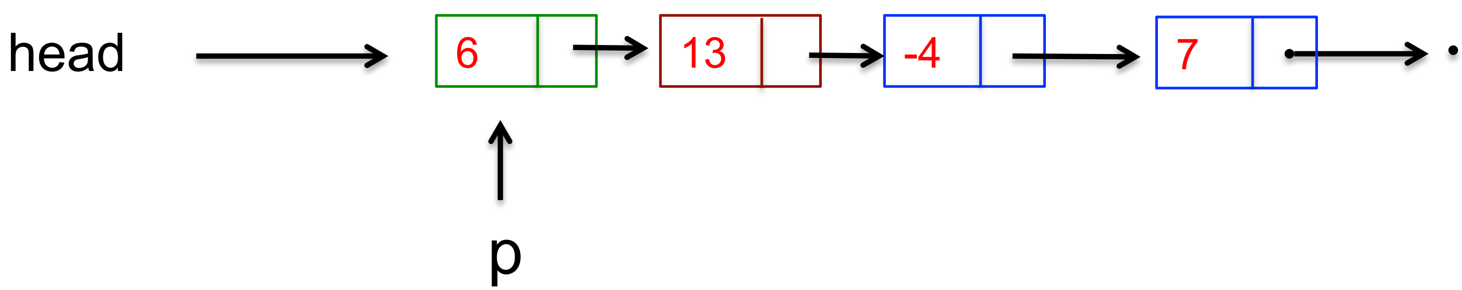


Linked Lists: Iterative Algorithms

The **basic operation** on a linked list, especially using loops, is to send a pointer down the list, “chaining along” to point to every element in turn. We will examine four different algorithms that use this idea: Printing a list, Searching a list, Deleting from a list, and Inserting into a list.

Printing a Linked List

Suppose we already have a linked list as follows:



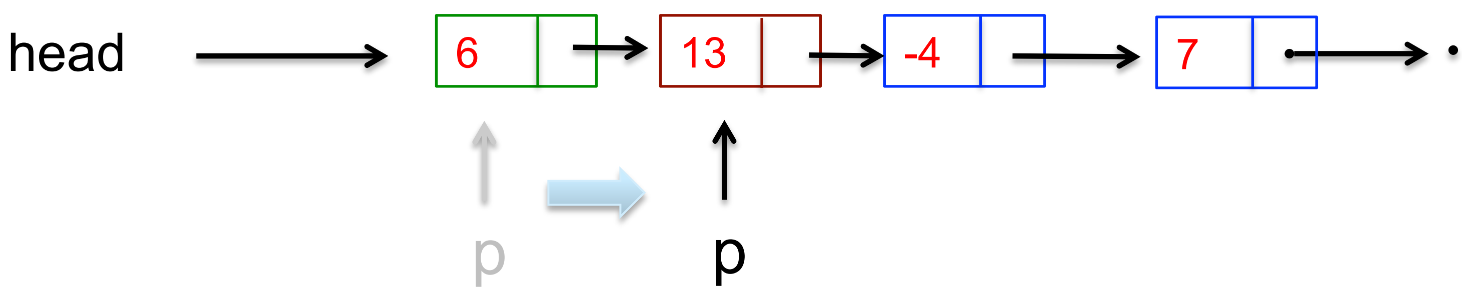
We can move down the list by assigning a pointer p to point to head,

```
Node p = head;
```

The **basic operation** on a linked list, especially using loops, is to send a pointer down the list, "chaining along" to point to every element in turn. We will examine four different algorithms that use this idea: Printing a list, Searching a list, Deleting from a list, and Inserting into a list.

Printing a Linked List

Suppose we already have a linked list as follows:



We can move down the list by assigning a point p to point to head,

```
Node p = head;
```

and then "chain along" using the following characteristic statement:

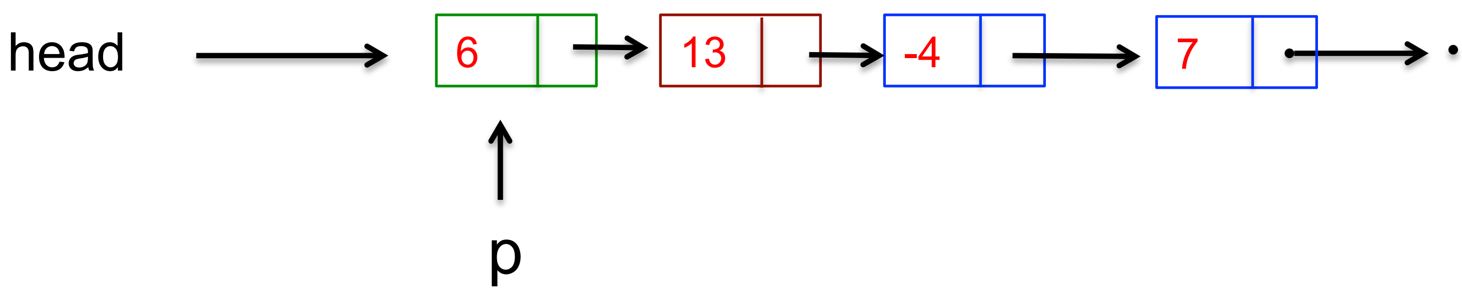
```
p = p.next;
```

Linked Lists: Iterative Algorithms

Computer Science

The **basic operation** on a linked list, especially using loops, is to send a pointer down the list, “chaining along” to point to every element in turn. We will examine four different algorithms that use this idea: Printing a list, Searching a list, Deleting from a list, and Inserting into a list.

Printing a Linked List



Printout:

6

Better to put this in a loop!

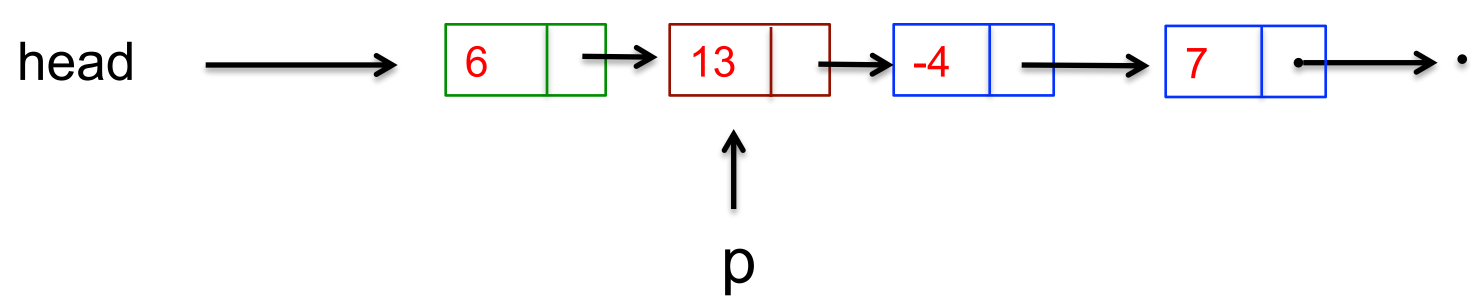
```
Node p;

for( p = head; p != null; p = p.next ) {

     System.out.println( p.item );

}
```

Linked Lists: Iterative Algorithms

Computer Science

The **basic operation** on a linked list, especially using loops, is to send a pointer down the list, “chaining along” to point to every element in turn. We will examine four different algorithms that use this idea: Printing a list, Searching a list, Deleting from a list, and Inserting into a list.

Printing a Linked List



Printout:

6

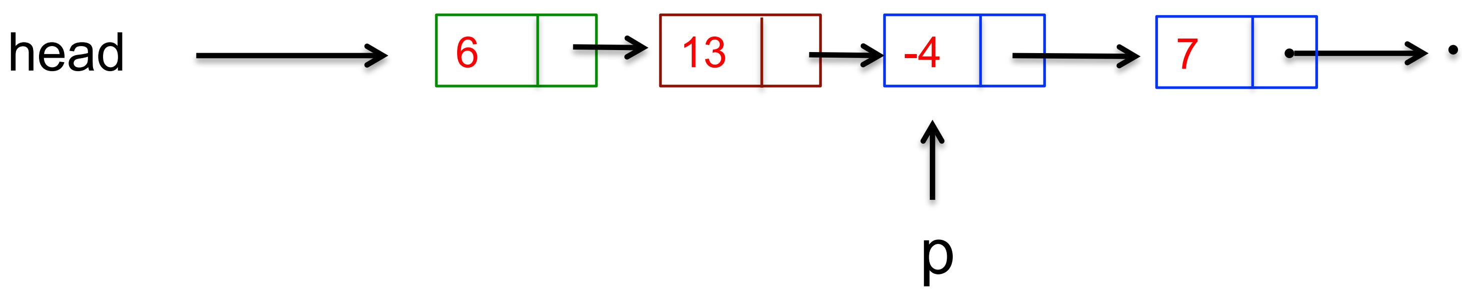
13

```
//  Better to put this in a loop!

Node p;

for( p = head; p != null; p = p.next ) {

     System.out.println( p.item );

}
```

Linked Lists: Iterative Algorithms

The **basic operation** on a linked list, especially using loops, is to send a pointer down the list, "chaining along" to point to every element in turn. We will examine four different algorithms that use this idea: Printing a list, Searching a list, Deleting from a list, and Inserting into a list.

Printing a Linked List



Printout:

```
6
13
-4
```

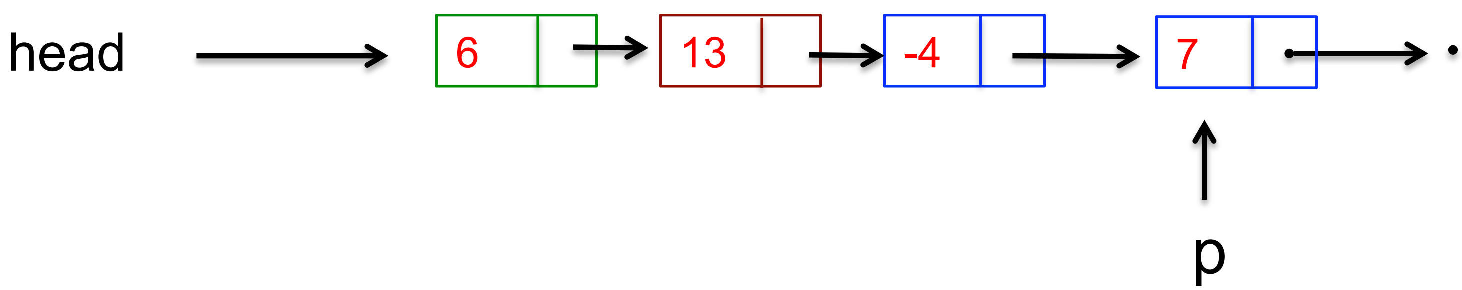
```
//  Better to put this in a loop!

Node p;

for( p = head; p != null; p = p.next ) {

      System.out.println( p.item );

}
```

Linked Lists: Iterative Algorithms

Computer Science

The **basic operation** on a linked list, especially using loops, is to send a pointer down the list, "chaining along" to point to every element in turn. We will examine four different algorithms that use this idea: Printing a list, Searching a list, Deleting from a list, and Inserting into a list.

Printing a Linked List




```
//  Better to put this in a loop!

Node p;

for( p = head; p != null; p = p.next ) {

      System.out.println( p.item );

}
```

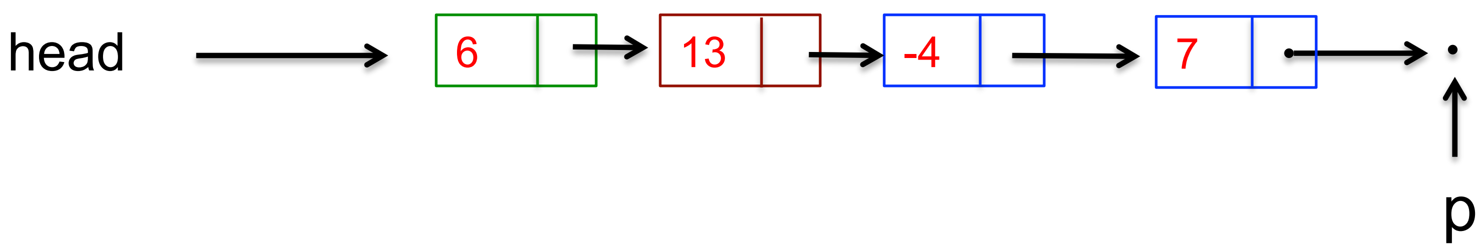
Printout:

```
6

13

-4

7
```

Linked Lists: Iterative Algorithms

The **basic operation** on a linked list, especially using loops, is to send a pointer down the list, "chaining along" to point to every element in turn. We will examine four different algorithms that use this idea: Printing a list, Searching a list, Deleting from a list, and Inserting into a list.

Printing a Linked List




```
//  Better to put this in a loop!

Node p;

for( p = head; p != null; p = p.next ) {

     System.out.println( p.item );

}
```

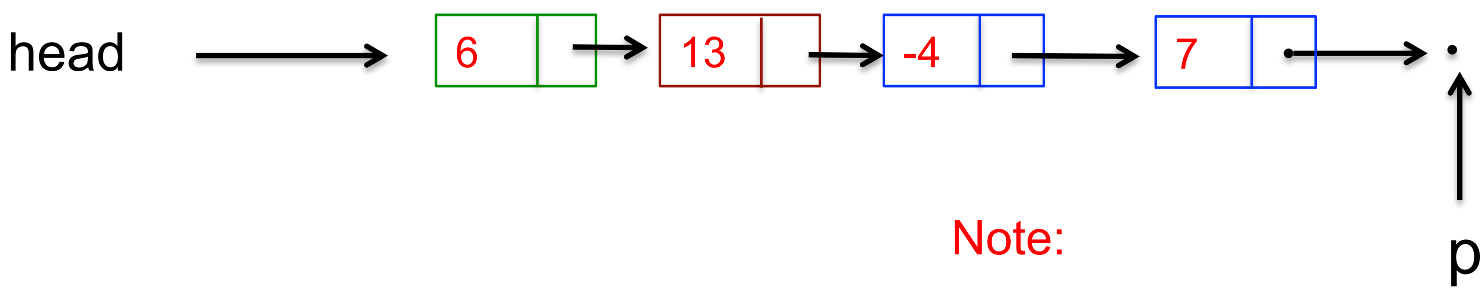
Printout:

```
6

13

-4

7
```

Linked Lists: Iterative Algorithms

The **basic operation** on a linked list, especially using loops, is to send a pointer down the list, "chaining along" to point to every element in turn. We will examine four different algorithms that use this idea: Printing a list, Searching a list, Deleting from a list, and Inserting into a list.

Printing a Linked List




```
//  Better to put this in a loop!

Node p;

for( p = head; p != null; p = p.next ) {

     System.out.println( p.item );

}
```

Note:

The for/while condition

p != null

guards the dereferences

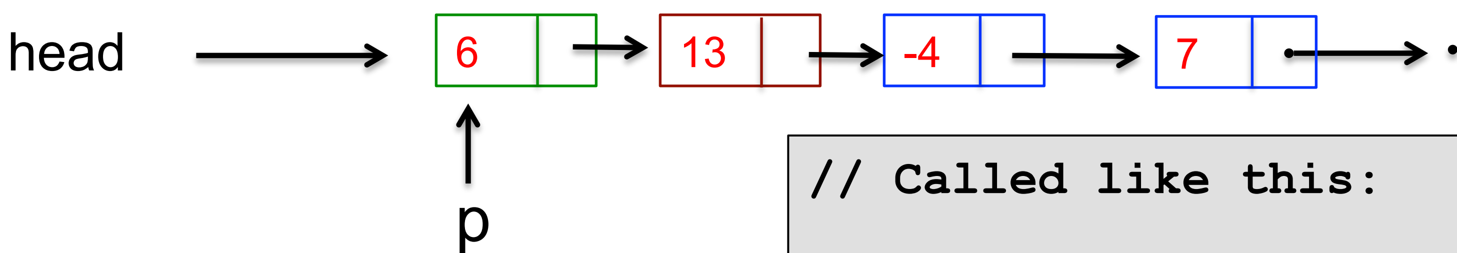
p.item and p.next

so no NullPointerException!

Linked Lists: Iterative Algorithms

The **basic operation** on a linked list, especially using loops, is to send a pointer down the list, “chaining along” to point to every element in turn. We will examine four different algorithms that use this idea: Printing a list, Searching a list, Deleting from a list, and Inserting into a list.

Method for Printing a Linked List

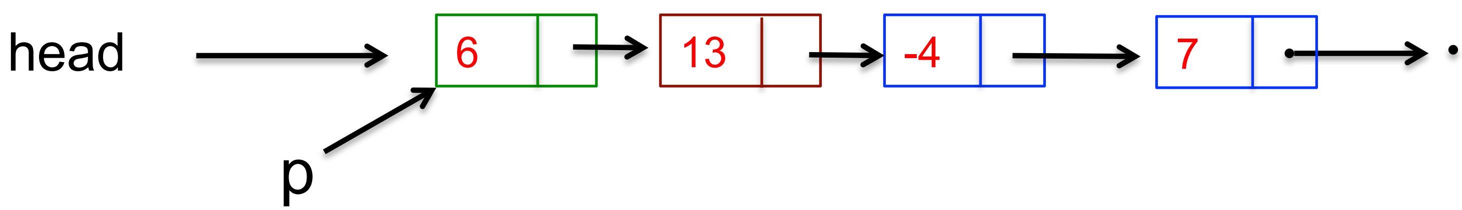



```
// Called like this:

printList( head );

 6 -> 13 -> -4 -> 7 -> .
```

```
//  Print out a Linked List

void printList( Node h ) {

      for( Node p = h;  p != null;  p = p.next  )
           System.out.print( p.item + " -> " );

      System.out.println(".");
}
```

The **basic operation** on a linked list, especially using loops, is to send a pointer down the list, "chaining along" to point to every element in turn. We will examine four different algorithms that use this idea: Printing a list, Searching a list, Deleting from a list, and Inserting into a list.

Method for Searching a Linked List




```
boolean member( int k, Node h ) {

      for( Node p = h; p != null; p = p.next ) {
           if( p.item == k ) return true;
      }

      return false;
}
```

```
// Called like this:

if ( member( 13, head ) )
    System.out.println("Found");
```

For deleting and inserting, we will need to do the examples on the board….

Method for deleting a node from a linked list

```
void delete(int n ) {
  if(head == null)          // case 1: list is empty, do nothing
    ;
  else if(head.item == n)   // case 2: n occurs in first node
    head = head.next;       //    skip around first node

  else {                    // case 3: find the node before n
    for(Node p = head.next, q = head; p!=null ; q=p, p=p.next ) {
      if( p.item == n ) {
         q.next = p.next;
         return;  // delete this line to remove all instances
      }
    }
  }
}

// You would call the method like this:

    delete(23);
```