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:

```
A:
```

```
0 1 2 3 4 5 6 7 8 9
```

```
3 1 4 1 5 9 2 6 5 3
```
Representing Sequences of Data

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

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 an address (= index of array) and holds numbers:

<table>
<thead>
<tr>
<th>RAM:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>-34</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>232</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>-78</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>

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:"


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

Access time = about $10^{-7}$ secs
When you create variables in Java (or any programming language), these are “nicknames” or shortcut ways of referring to locations in RAM:

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

```
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:
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.

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

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

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

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

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

A[0]: 0  
A[1]: 0

.....

P
x: 2  
y: 3
```
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);  

A[0]:  
A[1]:  
P:  
```

Old objects are “garbage” and the memory will be reclaimed by the “garbage collector” and reused.
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:

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

P → 6 → null
Linked Lists in Computer Memory

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

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

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

P: null

6

• = 0 = null
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**:

```java
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!

```java
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:

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

This explains how to interpret 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

Example:

```java
p.next = p.next.next;
p.next = (p.next).next;
```

compare:

```java
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).
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):

```
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
}
```
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):

```java
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
}
```
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:

Node top = null;
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:

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
}
```
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.

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

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

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

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

```java
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
}
```
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….

```
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
}
```

Queue:

```
```

```
An empty stack is just represented by an empty list

```
  top  •
```

and a queue by an empty list with two pointers:

```
front  •  ←  rear
```

```java
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);

```
front  7  •
```

```
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,

\[ \text{Node } p = \text{head}; \]
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:

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

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;
```
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;

```java
for (p = head; p != null; p = p.next) {
    System.out.println(p.item);
}
```

Printout:

6
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

```java
Node p;
for( p = head; p != null; p = p.next ) {
    System.out.println( p.item );
}
```

Printout:

```
6
13
```

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

```java
Node p;
for( p = head; p != null; p = p.next ) {
    System.out.println( p.item );
}
```

Printout: 6 13 -4 7
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

```java
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**

```java
// 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
```
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

```java
Node p;
for( p = head; p != null; p = p.next ) {
    System.out.println( p.item );
}
```

Note:

The for/while condition

\[ p \neq null \]

**guards** the dereferences

\[ p.item \quad \text{and} \quad p.next \]

so no `NullPointerException`! 30
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**

```java
// Called like this:
printList( head );
```

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

```java
// 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**

```java
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:
```java
if (member(13, head))
    System.out.println("Found");
```
Linked Lists: Iterative Algorithms

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

**Method for deleting a node from a linked list**

```java
void delete(int n) {
    if(head == null) {  // case 1: list is empty, do nothing
        return;
    }
    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:

dele(23);