CS 112 – Introduction to Computing II

Wayne Snyder
Computer Science Department
Boston University

Today

- Introduction to Linked Lists

Next Time

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

Computer Science

Representing Sequences of Data

Computer Science

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

The mathematical term for this is a **sequence**, and it has various notations:

< 3, 1, 4, 1, 5, 9, 2, 6, 5, 3 > // most usual

[3, 1, 4, 1, 5, 9, 2, 6, 5, 3]

(3, 1, 4, 1, 5, 9, 2, 6, 5, 3) // also called a tuple

The most important thing to get straight is that each element in a **sequence** has a **fixed position** (first, last, 3th) and order matters! A sequence can have **duplicates**!

But DON"T confuse this with the notation for a **set**!

{ 3, 1, 4, 5, 9, 2, 6 } is same as {5, 9, 2, 6, 3, 1, 4 }

A **set** has **no order** and **no duplicates!**

2

Representing Sequences of Data

Computer Science

The simplest and most efficient representation of a sequence in a computer is, of course, an array:

```
int [] A = { 3, 1, 4, 1, 5, 9, 2, 6, 5, 3 };        // just to confuse you, this is a sequence, not a set!
```

OR:

```
int [] A = new int[10];    A[0] = 3;   A[1] = 1;   A[2] = 4; .....    A[9] = 3;
```

Produces the following structure we've been using since the first week of CS 112:

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

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!

4

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 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

5

Data in Computer Memory

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.

	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

6

Objects/Classes in Computer Memory

Computer Science

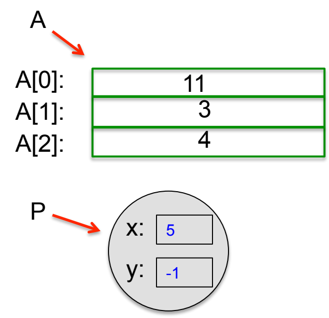
BUT Reference Types (arrays, Strings, objects – anything you can use the word `new` to create new instances of) are **references** or **pointers** to their values: **they store the location of the value, not the value itself.**

	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);
```



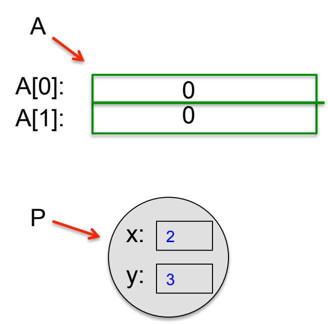
Objects/Classes in Computer Memory

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);
```



Objects/Classes in Computer Memory

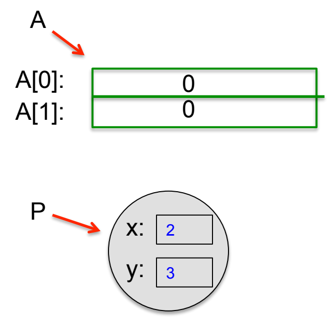
Computer Science

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

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

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

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



9

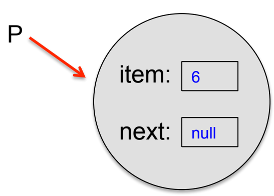
Linked Lists in Computer Memory

Computer Science

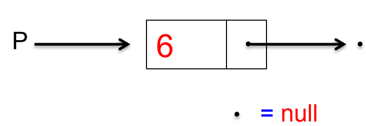
This FREES US from having to store items in contiguous locations, as in an array.
A **linked list** is a way to do this, with a completely different set of tradeoffs from the array representation.

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

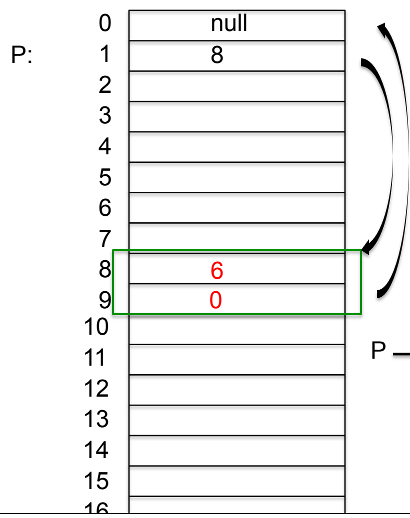
null (actually location 0) is a special location indicating "nothing there".

10

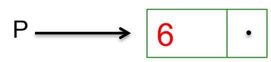
Linked Lists in Computer Memory

Computer Science

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;
```



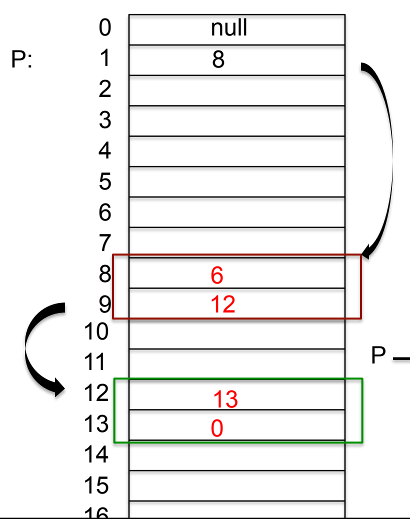
• = 0 = null

11

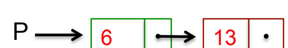
Linked Lists in Computer Memory

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);
```

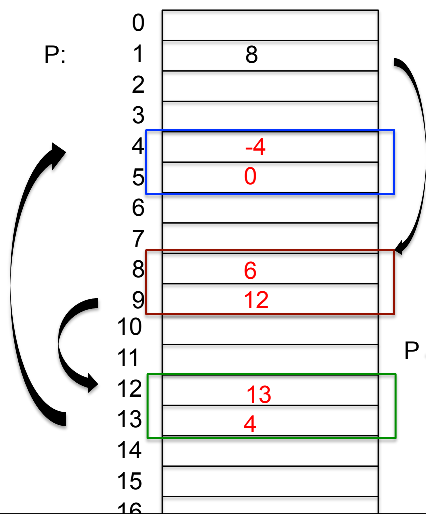


12

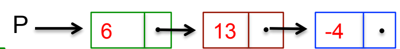
Linked Lists in Computer Memory

Computer Science

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)));
```

13

Linked Lists Review

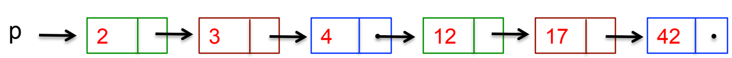
Computer Science

The **advantage** of a linked list 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. Minor advantage: generic types have no issues!

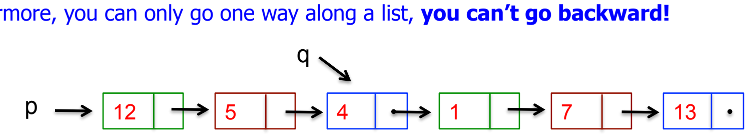
The **disadvantage** of a linked list is that it must use sequential access in one direction only:

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

(For example, you CAN'T USE binary search!)



Furthermore, you can only go one way along a list, **you can't go backward!**



If you have a pointer to the node containing 4, how to get to previous node?? No way!

14

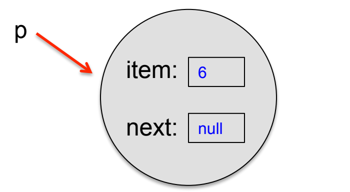
Linked Lists in Java

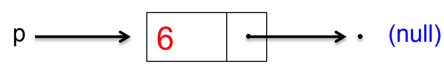
Computer Science

A linked list is a linear sequence of **nodes**, which are just objects which hold a data item and a pointer to another node just like itself; each node points to the next in the sequence.

```
public class Node {
   public int item;
   public Node next;

   // constructors
   public Node() {
      item = 0;
      next = null;
   }

   public Node(int n) {
       item = n;
       next = null;
   }

   public Node(int n, Node p) {
          item = n;
          next = p;
   }
};
```

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





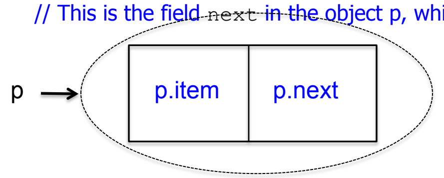
15

Linked Lists in Java

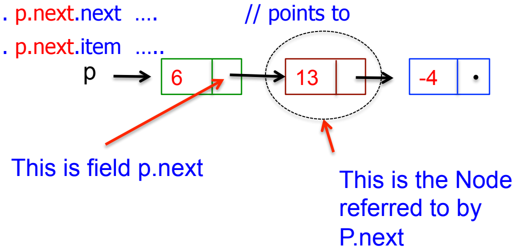
Computer Science

Be **SURE** that you understand the different uses of p.item, p.next , etc. in assignment statements:

p.item = ... // This is the field `item` in the object p, which stores a data item
p.next = ... // This is the field ~~`next` in the object~~ p, which stores a pointer



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



Example:

Node p = ...
Node q = ...

p = q;

Both p and q point to node pointed to by q

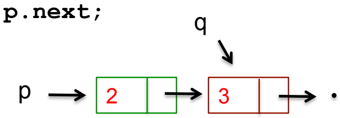
16

Linked Lists in Java

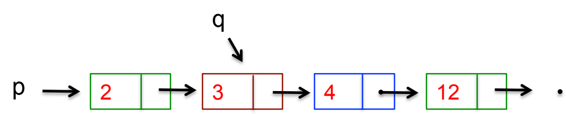
Computer Science

```
Node p = new Node( 2 );
```


```
p.next = new Node( 3, null );
Node q = p.next;
```




```
q.next = new Node( 4, new Node( 12 ) );
```



17

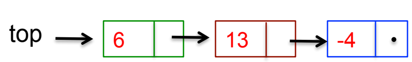
Linked List Basics Concluded: Stacks and Queues

Computer Science

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() {
       .......
}
```

```
void push(int n) {
    ......
}

boolean isEmpty() {
    .......
}

int size() {
    ..........
}
```

18

Linked List Basics Concluded: Stacks and Queues

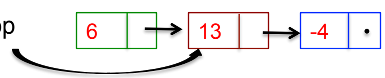
Computer Science

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:

top



13
-4
------

tmp: 6

Node top = null;

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

```
void push(int n) {
    ……
}

boolean isEmpty() {
    …….
}

int size() {
    …….
}
```

19

Linked List Basics Concluded: Stacks and Queues

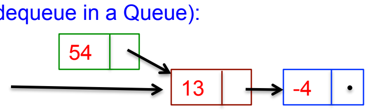
Computer Science

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:

top



push(54);

13
-4

Node top = null;

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

```
void push(int n) {
    ……..
}

boolean isEmpty() {
    ……..
}

int size() {
    ……..
}
```

20

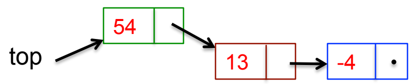
Linked List Basics Concluded: Stacks and Queues

Computer Science

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) {
    ……..
}

boolean isEmpty() {
    ……..
}

int size() {
    ……..
}
```

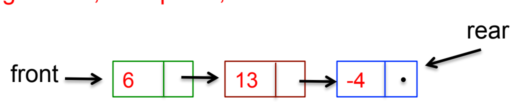
21

Linked List Basics Concluded: Stacks and Queues

Computer Science

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() {
        ……..
}
```

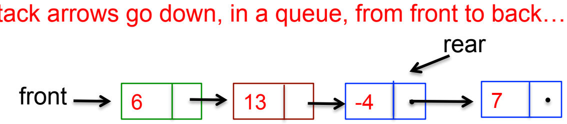
```
void enqueue(int n) {
    ……..
}

boolean isEmpty() {
    ……..
}

int size() {
    ……..
}
```

22

Linked List Basics Concluded: 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:



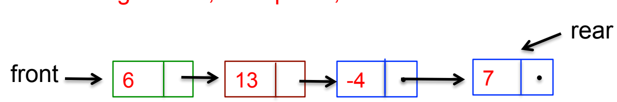

```
int dequeue() {
       ……..
}
```

```
void enqueue(int n) {
    ……..
}

boolean isEmpty() {
    ……..
}

int size() {
     ……..
}
```


Linked List Basics Concluded: 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 rear….

Queue:




```
int dequeue() {
       ……..
}
```

```
void enqueue(int n) {
    ……..
}

boolean isEmpty() {
    ……..
}

int size() {
     ……..
}
```

Linked List Basics Concluded: Stacks and Queues

Computer Science

An empty stack is just represented by an empty list

top → •

------

and a queue by an empty list with two pointers:

```
void enqueue(int n) {
   ........
}
```

front → • ← rear

enqueue(7);

front → | 7 | • | ← rear

25