

1999 APS105S Lecture 23: Linked Lists II

News

- Friday: quicksort (not in book)
- next week: complexity (difficult topic, but sort of in book)

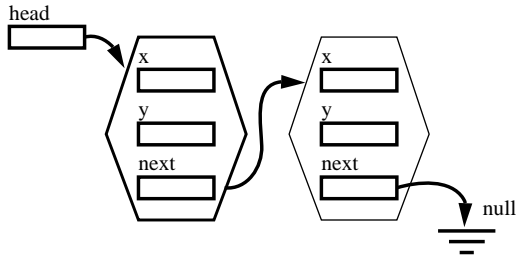
Last Day

- introduction to linked lists
- inserting into the 'head' of the list

Today

- more linked list operations
- simple complexity analysis

Linked List Data Structures (Chapter 13)



```
class PointElem {
    public int x, y;
    public PointElem next;
}
```

note: any class can have reference to an object in it

- the 'next' pointer is *self-referential* (own class)

Linked List Class

The methods are all underlined, for easier reading.

The *bold italic* code maintains the tail pointer. It is only needed by the `appendFast()` method. Ignore this part of the code on your first reading.

```
public class LinkedList
{
    private PointElem head;
    private PointList tail;

    public LinkedList()
    {
        head = null;
        tail = null;
    }

    public PointElem head()
    {
        return head; // hide so it can't change
    }
}
```

Linked List Class

usually we like to 'separate' the ideas:

- of the linked list
- of the scribble program
- benefits
 - organizes our ideas
 - easier to understand our programs
 - easier to make correct programs

proper use:

- make a linked list class
- use this class in ScribblePaint4

identify operations we can do to a linked list, write methods

- create a new linked list
- add an element
 - at the beginning, `insert()`
 - at the end, `append()`
 - before some other element, `insertInOrder()`
- find an element, `find()`
- print the list, `print()`
- delete an element, `delete()`
- reverse the elements, `reverse()`

proper program design

- can use linked lists in many places
 - ScribblePaint applet
 - reverseNumbers() program
 - unique integers program
- like to reuse some parts of linked list code

```
public void insert(PointElem newelem)
{
    if( tail == null )
        tail = head;
    newelem.next = head;
    head = newelem;
}
```

- how many steps does `insert()` take?
- no loops, we'll say it takes 1 step

```
public void append(PointElem newelem)
{
    if( head == null ) {
        insert( newelem );
    } else {
        PointElem curr = head;
        while( curr.next != null ) {
            curr = curr.next;
        }
        curr.next = newelem;
        newelem.next = null; // make sure
        tail = newelem;
    }
}
```

- how many steps does this take?
 - N steps (worst case)
- can we do better?
 - what if we knew where the last item was?
 - call it 'tail'
 - see `appendFast()` later on...

```

public void insertInOrder(PointElem newelem)
{
    if( head == null || head.x >= newelem.x ) {
        insert( newelem ); // change head
    } else {
        PointElem curr = head;

        while( curr.next != null
            && curr.next.x < newelem.x ) {
            curr = curr.next;
        }
        newelem.next = curr.next;
        curr.next    = newelem; // join to list

        // is newelem the new tail?
        if( newelem.next == null )
            tail = newelem;
    }
}

```

- how many steps does insertInOrder() take?
 - based on # steps in the while() loop
 - best case: ONE STEP (adds right away at head)
 - assume WORST CASE
 - ie, must add at the end all the time
 - say that it takes N steps, where N is the size of the list

```

public void appendFast( PointElem newelem )
{
    tail.next    = newelem;
    newelem.next = null; // make sure
    tail = newelem;
}

```

- appendFast() takes 1 step (worst case!)
 - much better than N steps that append() takes!!!!

```

// example to show walking the entire list
public void print()
{
    PointElem item = head;
    while( item != null ) {
        System.out.println(item.x + "," + item.y);
        item = item.next;
    }
}

```

- always takes N steps

```

public PointElem find( int value )
{
    PointElem curr = head;
    while( curr != null ) {
        if( curr.x == value )
            return curr;
        curr = curr.next;
    }
    return null;
}

```

- takes N steps in the WORST CASE

```

// etc. for delete(), ...

```

```

} // end of class LinkedList

```

ScribblePaint4 Program

```

import java.applet.*;
import java.awt.*;

public class ScribblePaint4 extends Applet {
    private LinkedList list;

    public void init() {
        list = new LinkedList();
    }

    public boolean mouseDown( Event e,
                             int x, int y ) {
        PointElem e = PointElem(x,y);
        list = list.append( e ); //correct order
        repaint();
        return true;
    }

    public void paint( Graphics g ) {
        // still have to 'walk' the list
        // the same way as before.
        PointElem curr = list.head();
        while( curr.next != null ) {
            g.drawLine( curr.x, curr.y,
                       curr.next.x, curr.next.y );
            curr = curr.next;
        }
    }
}

// PointList program still specific (x,y)
// will show you how to make a generic
// linked list in a later lecture

```

Simple Linked List Program

Unique Integers

- we only need to store 1 integer
 - store it in the 'x' field of PointElem
 - we won't use the 'y' field
 - normally, we'd make a customized PointElem object for each type of LinkedList we have to create

```

public class UniqueIntegers
{
    public static void main( String[] args )
    {
        LinkedList list = new LinkedList();

        while( true ) {
            int number = Stdin.getInt();
            if( number < 0 )
                break;

            // look for number in list
            if( list.find( number ) == null ) {
                PointElem e;
                e = new PointElem(number,0);
                list.append( e );
            }
        }

        list.print();
    }
}

```