PURDUE UNIVERSITY

CS18000: Problem Solving and Object-Oriented Programming

Recursion

28 March 2011

Prof. Chris Clifton

COMPUTER SCIENCES

Recursion

- Idea: break a problem down into small, similar sub-problems
  - Write a method to solve first
  - Call that method to solve next
- Sounds similar to a loop
  - Same basic effect
  - But often an easier way to conceptualize the solution

Simple Example: Factorial

- What is n!
  - n*(n-1)*(n-2)*…*1
- Easy enough with a loop
  - int result = 1;
  - for (int i=1; i<=n; i++) result = result*i;
- But think of the problem differently:
  - n! = n * (n-1)!

Recursive Factorial

- n! = n * (n-1)!

```
public static int factorial( int n ) {
    if ( n==1 ) return 1; else
    return n * factorial( n-1 );
}
```

- Key components:
  - Base case – when do we know the answer
  - Recursive case – how do we call to get the answer

Visualizing Recursion: Call Stack

- "Save" what is left to do
- Call (recursive) method

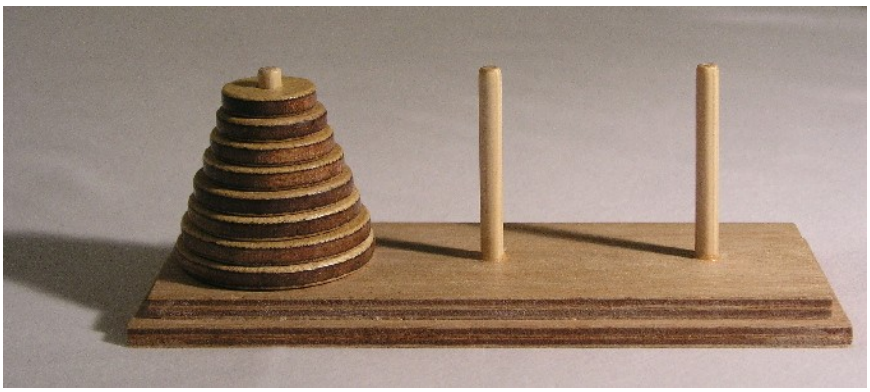
```
public static int factorial( int n ) {
    if ( n==1 ) return 1; else
    return n * factorial( n-1 );
}
```

- factorial(3)
- factorial(3)
  - ~~return 3 * factorial(2)~~ ~~2~~ 6
- factorial(2)
  - ~~return 2 * factorial(1)~~ ~~1~~ 2
- factorial(1)
  - return 1

What is factorial(3); ?

A. 1
B. 2
C. 6
D. 12
E. 24

Tougher Problem: Towers of Hanoi



Source: Wikipedia

Tougher Problem: Towers of Hanoi

- We know how to move the top disk
  - Just move it
- But how to move one underneath?
  1. Move disk above to center
  2. Move disk to right
  3. Move upper disk from center to right

Towers of Hanoi: Solution

Source: Wikipedia

Tougher Problem: Towers of Hanoi

- We know how to move the top disk
  - Just move it
- But how to move one underneath?
  1. Move disk above to center
  2. Move disk to right
  3. Move upper disk from center to right
- Recursive approach
  - call “move” for “move disk above”

Hanoi Example

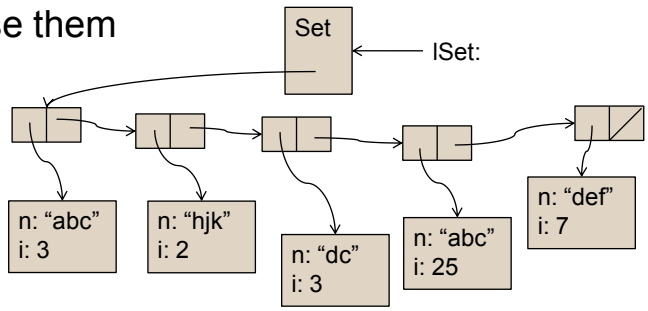
```
public class TowersOfHanoi {
    public static void main(String[] args) {
        move(4, “A”, “C”, “B”);
    }
    static void move(int n, String from, String to, String via) {
        if (n == 1)
            System.out.println(“move disk from tower “ + from + “ to “ + to);
        else {
            move(n-1, from, via, to);
            move(1, from, to, via);
            move (n-1, via, to, from);
        }
    }
}
```


Towers of Hanoi: *Object-Oriented Thinking*

- I am a disk, I'm asked to move to right
  - Ask disk above me to move to center
  - Then I move to right
  - Then ask disk above to move back above me
    - (move to right)

Use with Linked Data Structures

- Node has methods
  - Use them



- If I can do method, do it
- If not, ask "next" *list* to do it

PURDUE UNIVERSITY

CS18000: Problem Solving and Object-Oriented Programming

Search Trees

30 March 2011

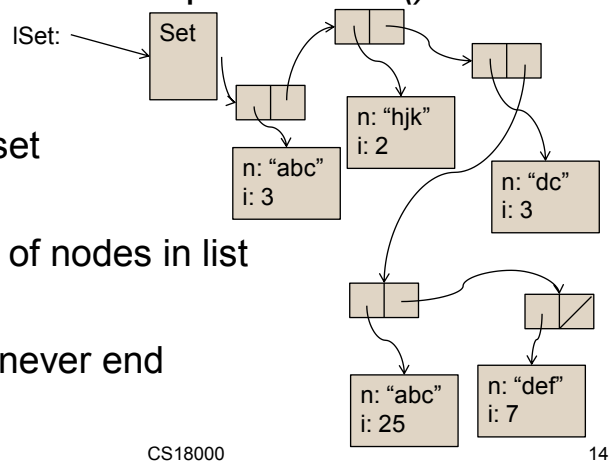
Prof. Chris Clifton

COMPUTER SCIENCES

Linked List Set

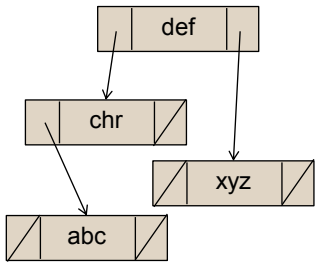
- What is the most steps *contains()* takes?

A. Size of set
B. 1
C. Number of nodes in list
D. 27
E. It might never end



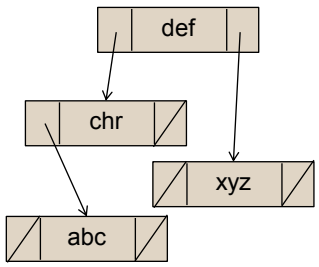
Linked Data Structure: Search Tree

- Like to have faster search
- Solution: Search tree
  - Linked data structure
  - Multiple out links
  - Choose one!



Search tree: boolean contains(T item)

- Check if this.item.equals(item)
  - If yes, return true
- Is item < this.item?
  - return left.contains(item)
- else
  - return right.contains(item)



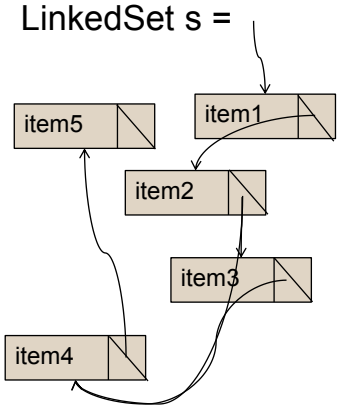
What is the most steps contains() takes?

A. No base case
B. No recursive case
C. Height of tree
D. Might never end

More on Linked Data Structures

- Difficulties with Dynamic Data Structures:
  - Removing self
- Solution 1: Violate Encapsulation
  - Operate on your child
  - Move child to self
- Solution 2: Extra class
  - Linked list contains first linked node

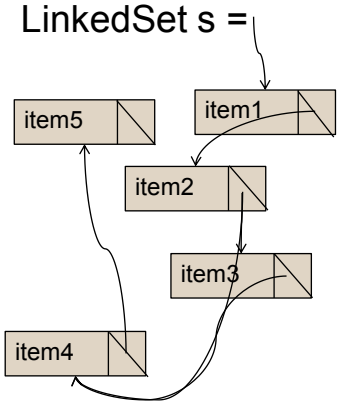
- LinkedSet s =



Exceptions as a Programming Technique

- Difficulties with Dynamic Data Structures:
  - Removing self
  - Pointers to head of structure
- Solution 3: Exceptions

- LinkedSet s =



Idea: Inform Parent

- Tell parent to "eliminate" us
  - Parent has called us with recursive call
  - Return new head
    - void remove(item);
    - Should it be LinkedSet<E> remove(E item); ?
    - Does this make sense if nothing changed?
    - *What if we want to return (for example) the removed item?*
- Solution: throw exception
  - Exception has information parent needs to know

Exception that *Carries Information*

```
class NewHead extends Exception {
    LinkedList<E> head;
    E removedItem = null;
    NewHead ( LinkedList<E> head ) {
        this.head = head;
    }
    NewHead ( LinkedList<E> head, E item ) {
        removedItem = item;
        this.head = head;
    }
}
```


Linked List

```
public class LinkedList<E> {
    private E current;
    private LinkedList<E> next =
        null;

    public class NewHead extends
        Exception {
        private LinkedList<E> head;
        public NewHead(LinkedList<E>
            head) {
            this.head = head;
        }
        public LinkedList<E> newHead()
            {
            return head;
        }
    }
```

```
public LinkedList ( E item ) {
    current = item;
}

private LinkedList (E item,
    LinkedList<E> next) {
    current = item;
    this.next = next;
}

public E getItem() {
    return current;
}

public void addAfter(E item) {
    next = new LinkedList<E>(item, next);
}
```


Linked List

```
public class LinkedList<E> {
    private E current;
    private LinkedList<E> next =
        null;

    public class NewHead extends
        Exception {
        private LinkedList<E> head;
        public NewHead(LinkedList<E>
            head) {
            this.head = head;
        }
        public LinkedList<E> newHead()
            {
            return head;
        }
    }
```

```
public void append( E item ) {
    if ( next == null ) addAfter(item);
    else next.append(item);
}

public void addBefore(E item) throws
    NewHead {
    LinkedList<E> newList = new
        LinkedList<E>(item, this);
    throw new NewHead(newList);
}

public boolean contains(E item) {
    if ( item.equals(current) ) return true;
    else if (next == null) return false;
    else return next.contains(item);
}
```

Linked List

```
public class LinkedList<E> {
    private E current;
    private LinkedList<E> next =
        null;

    public class NewHead extends
        Exception {
        private LinkedList<E> head;
        public NewHead(LinkedList<E>
            head) {
            this.head = head;
        }
        public LinkedList<E> newHead()
            {
            return head;
        }
    }
```

```
public void remove(E item)
    throws NewHead {
    if (item.equals(current)) //
        Eliminate ourself
        throw new NewHead(next);
    else if ( next != null ) {
        try { next.remove(item); }
        catch (NewHead e) {
            next = e.newHead();
        }
    }
}
```