CS 367 - Introduction to Data Structures
Thursday, September 3, 2015

Instructor

- Jim Skrentny
- skrentny@cs.wisc.edu
- 5379 CS

Today

Waitlisted? More info next week...

Course Intro

- Topics
- Course Information
- Course Work

Java Primitives vs. References Review

Next Time

Read: *Introduction*, start *Lists*

Collections

- Bag Intro
- Abstract Data Types
- designing the Bag ADT - Java interfaces
- using the Bag ADT

Characteristics of Good & Reusable Software

Implementing the Bag ADT

- casting when using Object
- using Java generics for generality

Topics

Survey of Abstract Data Types (ADTs) and Data Structures (DS)

Linear → Hierarchical → Graphical

position oriented → value entered.

Introduction to Algorithms

ADT OPERATIONS

TRAVERSING

SEARCHING / SORTING

HASHING

Introduction to Complexity

TIME / SPACE

BIG "O" NOTATION

COMPARE / CONTRAST ALGO's / CODE

Review of and Build on Java Concepts from CS 302

PRIMITIVES vs. REFERENCES

Exceptions

Interface

ITERATERS

JAVA Collections FRAMEWORK

** We assume that you are proficient at object-oriented programming in Java. If you have not learned object-oriented programming, you should complete CS 302 first. If you have learned object-oriented programming in a language like C++, you should focus time in the next two weeks to learn Java or consider taking or sitting in CS 302.*

Course Information Sources

Course Website

- http://pages.cs.wisc.edu/~cs367-1/
- readings, outlines, homeworks, programs, announcements, policies, etc...

Online Readings (no textbook)

- cover page of lecture outline has reminders of what to be reading
- some things in readings won't be covered in lecture
- some things in lecture aren't in the notes

Lecture & Outlines

- do assigned online readings before lecture
- print and bring outlines to lecture
- readings/outlines posted on the course syllabus page
- outlines are usually posted the morning of or late the night before lecture
- fill in outlines with your notes during lecture

Piazza

- search and ask/answer questions about
  - homeworks and programs
  - course content and logistics
  - exams

Learn@UW

- access to exam and other scores, solutions to homeworks, exam sample questions

CS Account

- access to lab computers with Eclipse and Java
- linux lab computers in 1350 CS
- same user name/password as your 302 account (new? must activate your account)
- access to "in" folder to submit your coursework
- access to homework and program grade reports with scores and feedback

Course Work

Exams (55%)

- Midterm 1 (17%): Tuesday, October 20th, 5:00 pm to 7:00 pm
- Midterm 2 (17%): Tuesday, November 24th, 5:00 pm to 7:00 pm
- Final (21%): Tuesday, December 23th, 7:45 AM to 9:45 AM

Notify us next week if you have a conflict with any of these exams.
I'll say more about this next lecture.

Programming Assignments (25%)

- 5 programs, 5% each
- about two weeks to complete each
- topics needed are typically covered before a program is assigned
- pair programming is allowed Except FIR PROGRAM P1
- submitted electronically into your "in" handin directory Homework PDF
- accepted late only if
- extenuating circumstances are beyond your control
- you notify me at least 3 days prior to due date
- the work can be completed in a few extra days

Homework Assignments (20%)

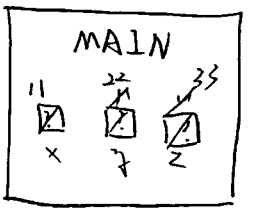
- 10 homeworks, 2% each
- about one week to complete each
- topics needed are often covered during the week the homework is assigned
- no collaboration is allowed
- submitted electronically into your "in" handin directory
- not accepted late

Primitive vs. Reference Types: Assignment

Primitives

```
int x, y, z;
x = 11;
y = x;
z = x;
z = 33;
y = 22;
```

Call stack

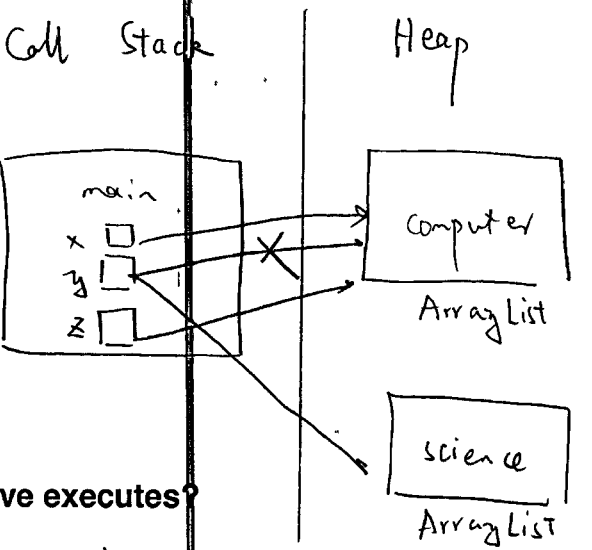


→ What does each variable contain after the code above executes?

A.) x has 11 y has 22 z has 33
B.) x has y has z has
C.) x has y has z has

References

```
ArrayList<String> x, y, z;
x = new ArrayList<String>();
y = x;
z = x;
y = new ArrayList<String>();
z.add("Computer");
y.add("Science");
```



→ What does each ArrayList contain after the code above executes?

A.) x's ArrayList has computer y's ArrayList has Science z's ArrayList has computer

B.) x's ArrayList has y's ArrayList has z's ArrayList has

C.) x's ArrayList has y's ArrayList has z's ArrayList has

→ What do x, y and z contain?

Addresses Reference

Primitive vs. Reference Types: Parameter Passing

Primitives

Given:

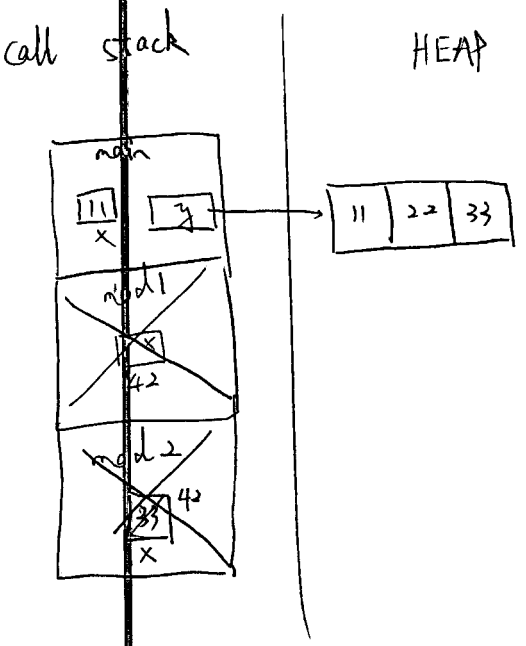
```
void mod1(int x) {
   x = 42;
}
```

Execute code in `main`:

```
int   x = 11;
int[] y = {11, 22, 33};
mod1(x);
mod1(y[2]);
```

→ What does variable `x` and array `y` in `main` contain after the code above executes?

A.) `x` has 11 `y`'s array has 11 22 33.
B.) `x` has `y`'s array has
C.) `x` has `y`'s array has



→ What happens if we call `mod1(y)` in main?

Primitive vs. Reference Types: Parameter Passing

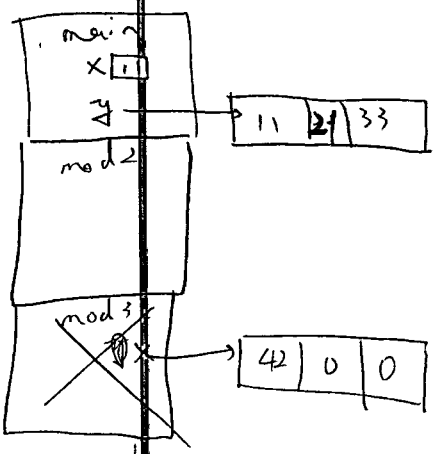
References

Given:

```
void mod2(int[] x) {
   x[0] = 21;
}

void mod3(int[] x) {
   x = new int[x.length];
   x[0] = 42;
}
```

Execute code in `main`:

```
int   x = 11;
int[] y = {11, 22, 33};
mod2(y);
mod3(y);
```



→ What does variable `x` and array `y` in `main` contain after the code above executes?

A.) `x` has `y`'s array has
B.) `x` has `y`'s array has
C.) `x` has `y`'s array has

→ What happens if we call `mod2(x)` in main?

Abstract:

Programmer's Memory Model for Java

Call Stack

ACTIVATION RECORDS FOR EACH CALLED METHOD

Contain Local VARS (Param VARS)

"BIRTH" AT DECLARATION

"DEATH" WHEN FALLS OUT of Scope

Heap

ARRAYS & objects

NEED A REFENCE, VAR, To Access Them.

"BIRTH" AT new ALLOcation.

"BIRTH" WHEN NO LONGER REFERENCE

(GARBAGE COLLECTOR)

Static Data

LITERALS

CLASS VARS (static)

ALLOCATION DOESN'T CHANGE

BIRTH AT PROGRAM's EXECUTION START

DEATH AT ... ... END

CS 367 - Introduction to Data Structures
Tuesday, September 8, 2015

Course Website - http://pages.cs.wisc.edu/~cs367-1/
Signup for Piazza

Waitlisted? Currently just a few seats available. Signup on the sheet past out during lecture.

Last Time

Course Intro
- Topics
- Course Information
- Course Work

Java Primitives vs. References Review

Today

Finish Java Primitives vs. References Review (from last time)
Collections
- Bag Intro
- Abstract Data Types & Data Structures
- designing the Bag ADT - Java interfaces
- using the Bag ADT

Characteristics of Good & Reusable Software
Implementing the Bag ADT using Java `Objects`

Next Time

Read: continue *Lists*
Implementing the Bag ADT
- casting when using Object
- using Java generics for generality

List ADT
- coding the ListADT as a Java interface
- using lists via the ListADT

Collections

→ **What is a *collection*?**

A GROUP OF ITEMS GATHERED INTO A CONTIANER

ITEMS (DATA) - INDIVIDUAL members of the collection

Simple - numbers (primitives)

Composite - student RECORD, ALBUM

(REFERENCES To objects)

CONTAINER (DATA STRUTURE) - STRUTURE used store the collection.

→ **What operations can you do on a collection? Which are the most fundamental?**

ADD / INSERT

REMOVE / DELETE

LOOKUP / SEARCH / GET

Example: Bags

Concept

GENERAL CONTAINER
- IT CAN STORE ANY TYPE OF ITEM
- IT CAN STORE DIFFERENT TYPES OF ITEMS IN THE SAME BAG.
- DUPLICATES ARE OKAY.

UNORDERED CONTAINER

UNORDERED CONTAINER

- THERE'S NO EXPLICIT INTERNAL STRUCTURE

Operations

- ADD & REMOVE ARE FAST (REMOVE A RANDOM ITEM)
- LOOK UP IS SLOW

void add (item)

Item remove ()

boolean isEmpty ()

Problems

→ What problems might occur when doing Bag operations?

REMOVE WHEN BAG IS EMPTY

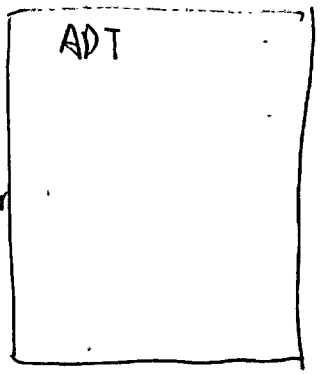
~~ADD WHEN BAG IS FULL~~. WELL EXPAND THE CAPACITY

ADTs - Abstract Data Types 1970s

WHY
APPLICATIONS

LIKELY TO HAVE MANY USES / IF YOUR ADT is general.

WHAT



HOW
IMPLEMENTATIONS

CODE THAT SAYS
- HOW THE COLLECTION is STRUCTURED
- How THE OPERATIO WORK

Array List
LinkedList

ADTs vs. Data Structures

Abstract Data Type (ADT)

SPECIFIES WHAT you can do.

JAVA DOCS - CONCEPTUAL DESCRIPTION
- List of operations

JAVA INTERFACE
is used to code the ADT IN JAVA

Data Structure (DS)

A CONSTRUCT WITHIN THE IMPLEMENTING CLASS
THAT's used & STORE THE COLLECTION of ITEMS.
E.G. ARRAY
CHAIN OF NODES

Example 1: Using a Bag ADT

→ **Write a code fragment**
to put the numbers 0 through 99 into a BagADT named bag.

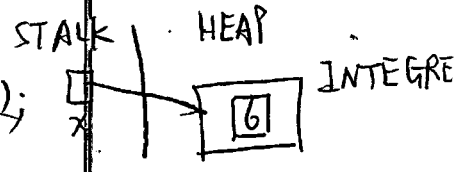
```
BagADT bag = new ...;  //assume the bag has been instantiated for you
```

```
For (int i=0; i<100 ; i++)
    bag.add (i);
```

Java Autoboxing:

JAVA AUTOMATICALLY CONVERTS BETWEEN PRIMITIVES AND THEIR WRAPPED CLASSES:

```
Integer x = 6;
= new Integer (6);
```




```
int y = x;
= (int) x;
```

Designing the Bag ADT

Conceptual Description BAG ADT ArrayList contain various bags

The I

you can add. items

R

E

Public Interface

ADP

REMORE

IS Empty

Coding Issues

Public class ArrayBag {

Public bag }

};

Public ~~bag~~ ~~dd~~ add. }

}.

Public REMORE {

}

}

Example 2: Using a Bag ADT

→ **Complete the printBag method**
so that it prints the contents of the parameter `bag`.
Challenge: Implement your `printBag` method so that it doesn't change the bag's contents.

```
public static void printBag(BagADT bag) {
```

isEmpty? then System.out.p
bag.size()

System.out for (St. i = 1, i<##; i++)

System.out.print(bag.get(i)

While (! bag.isEmpty())
S.o.p(n (bag.remove());

BagADT tmp = new ArrayBag();
while (!bag.isEmpty()) {
Object item = bag.remove();
System.out.println(item);
tmp.add(item);
}
while (!tmp.isEmpty()) {
bag.add(tmp.remove());
}
}

What makes software good?

1. WORKS
2. EASY TO USE / MODIFY
3. REASONABLY EFFICIENT

What makes code reuseable?

1. Documentation
2. modularity
   methods
   classes
   Interfaces
3. GENERALITY

The Bag ADT and Java `Objects`

```
import java.util.*;

public interface BagADT {

    void add(Object item);
    Object remove() throws NoSuchElementException;
    boolean isEmpty();
}
```

ABSTRACT METHODS

→ Why are we using the `Object` class in our `BagADT` interface?

FOR GENERALITY

- IN JAVA AN Object REF CAN refer to any Java object.

* BAG CAN STORE ANY TYPE of Object
* A SINGLE BAG CAN STORE DIFFERENT TYPES of objects

Implementing `BagADT` Using an Array of `Object` References

```
public class ArrayBag implements BagADT {

   //instance variables
   private int N;
   private Object[] items;
   private int numItem;

   //constructor
   public Bag() {
      first = null
      N = 0;
   }

   public ArrayBag() {
      items = new object[100];
      numberITEMS = 0;
   }

   //BagADT methods

   isEmpty public boolean isEmpty() {
      return numItem == 0;
   }

   size

   public void add(Object item) {
      // ADD ITEM TO FIRST FREE ELEMENT;

   // could add other methods specific to the array implementation
}
```

CS 367 Announcements
Thursday, September 10, 2015

Course Website - http://pages.cs.wisc.edu/~cs367-1/
Signup for Piazza

Waitlisted? Some invitations to register were emailed yesterday. A few more likely.

Exam Conflict? Information on how to report your conflict provided next lecture.

Program 1 assigned this morning (Thursday).

Last Time

Collections
- Bag Intro
- Abstract Data Types
- designing the Bag ADT - Java interfaces
- using the Bag ADT

Characteristics of Good & Reusable Software
Implementing the Bag ADT using Java `Objects`

Today

Implementing the Bag ADT
- casting when using Object
- using Java generics for generality

List ADT
- coding the ListADT as a Java interface
- using lists via the ListADT

Next Time

Read: continue *Lists*
Lists
- implementing the ListADT using an array (SimpleArrayList)

Java API Lists
Iterators
- concept
- iterators and the Java API
- using iterators

Recall the Bag ADT

1. Bag ADT Design & Interface (the WHAT):

A Bag is a general unordered container of items where duplicates are allowed.

Javadoc

```
import java.util.*;

public interface BagADT {
   void add(Object item);
   Object remove() throws NoSuchElementException;
   boolean isEmpty();
}
```

} what you can do.

→ Why were we using the `Object` class in our `BagADT` interface?

GENERALITY BAG can store any type as well as mixed types.

2. Bag ADT Implementation (the How):

```
public class ArrayBag implements BagADT {

   private Object[] items;
   private int numItems;

   public ArrayBag() {
      items = new Object[100];
      numItems = 0;
   }

   void add(Object item) { ... }
   Object remove() throws NoSuchElementException { ... }
   boolean isEmpty(){ return numItems == 0; }
}
```

] How is data structured.

] How operations work.

3. Bag ADT Use (the WHY)...

Use - BagADT and Casting

Using a general ADT and its implementation to instantiate a container:

ADT_NAME Object_Name = New IMPL_classname();

ADT_NAME: Java interface
Object_Name: Java Identifier
IMPL_classname(): The java class for the specific implementation we want to use

→ **Write a statement** that makes a Bag ADT container named bag.

BagADT bag = new ArrayBag();

→ Assume Die is a class representing dice and has a zero parameter constructor.
Write a code fragment that adds 6 dice to bag.

```
for (int i = 0; i < 6; i++)
    bag.add(new Die());
```

up casting: specific → general. Die → Object

→ Assume the bag has had items added to it. **Why doesn't the following code compile?**

```
while (!bag.isEmpty()) {
    Die myDie = bag.remove();
    myDie.roll();
}
```

ERROR

remove return type object.
But type Die is needed

Java up casts Automatically

Down casting: General → specific. Object → Die.

```
Object obj = bag.remove();
if (obj instanceof Die) {
    Die myDie = (Die) obj;
    myDie.roll();
}
```

Not Done Automatically

Java Generics - A Better Way to Make a General Bag ADT

What changes are needed to make the interface below generic?

```
import java.util.*;

public interface BagADT <E> {

   void add(Object item);

   Object remove() throws NoSuchElementException;

   boolean isEmpty();

}
```

1. Specify one or more "Type param"
   - List after class/interface Name.
   - put them in <>

2. Replace occurrences of specific type with the type param.

Use - Generic Bag ADT

How do we use a generic interface and its generic implementation to make a container?

ADT_Name <Item.TYPE> object_name = new IMPL_class <ITEMTYPE>();

Specifying the item type and restricting the container. Names to only type type. - No Down Casting is needed.

→ **Write a code fragment** to make one generic Bag ADT container storing `String` objects and another one storing `Die` objects.

```
BagADT <String> Bag1 = new ArrayBag <String>();
BagADT <Die> Bag2 = new ArrayBag <Die>();
```

→ **Write a statement** to add "cs367" into the appropriate container.

```
bag1.add("cs367");
```

→ Can we make a single generic Bag ADT container that can store both `String` and `Die` objects at the same time?

Can.

```
BagADT<Object> mixedBag = ArrayBag <Object>();
```

down-casting

But now down casting is needed

Implementation - Generic BagADT

What changes are needed to make the implementation below generic?

```
public class ArrayBag implements BagADT {

   private Object[] items;

   private int numItems;


   public ArrayBag() {
      items = new Object[100];
      numItems = 0;
   }


   boolean isEmpty(){ return numItems == 0; }

   void add(Object item) { ... }

   Object remove() throws NoSuchElementException { ... }
}
```

1. Specify ONE or more type params which typically matches those used by ADT.
2. REPLACE THE occurances of the specific type with the type params.

* 3. Be CAREFUL WITH CONSTRUCTORS THAT ALLOCATE generic Arrays you must make an array with some existing class (E.g. object) and cast to an array of the type param.

Design - List ADT

Concept GENERAL CONTAINER THAT ALLOWS DUPLICATES
Contains collection
position oriented with zero-based indexing
EXPANDS capacity
maintains Relative ordering

Operations

- add item at end of list
- add item at specified position
- get item at specified position
- remove item at specified position
- check if list contains a specified item
- get size of list (number of items it contains)
- check if list is empty

Issues

Null item – detect then signal with `IllegalArgumentException`

Bad position – detect then signal with `IndexOutOfBoundsException` ✓

Empty list – handle as a bad position

Interface - Generic `ListADT`

— Interface — Generics — Java Docs

```
/**
 * A List is a general container storing a contiguous collection
 * of items, that is position-oriented using zero-based indexing
 * and where duplicates are allowed.
*/
public interface ListADT <E> {

   /**
    * Add item to the end of the List.
    *
    * @param item the item to add
    * @throws IllegalArgumentException if item is null
    */
   void add(E item);

   /**
    * Add item at position pos in the List, moving the items
    * originally in positions pos through size()- 1 one place
    * to the right to make room.
    *
    * @param pos the position at which to add the item
    * @param item the item to add
    * @throws IllegalArgumentException if item is null
    * @throws IndexOutOfBoundsException if pos is less than 0
    * or greater than size()
    */
   void add(int pos, E item);

   /**
    * Return true iff item is in the List (i.e., there is an
    * item x in the List such that x.equals(item))
    *
    * @param item the item to check
    * @return true if item is in the List, false otherwise
    */
   boolean contains(E item);
```

what

Interface - Generic `ListADT` (cont.)

```
    /**
     * Return the number of items in the List.
     *
     * @return the number of items in the List
     */
    int size();

    /**
     * Return true iff the List is empty.
     *
     * @return true if the List is empty, false otherwise
     */
    boolean isEmpty();

    /**
     * Return the item at position pos in the List.
     *
     * @param pos the position of the item to return
     * @return the item at position pos
     * @throws IndexOutOfBoundsException if pos is less than 0
     * or greater than or equal to size()
     */
    E get(int pos);

    /**
     * Remove and return the item at position pos in the List,
     * moving the items originally in positions pos+1 through
     * size() one place to the left to fill in the gap.
     *
     * @param pos the position at which to remove the item
     * @return the item at position pos
     * @throws IndexOutOfBoundsException if pos is less than 0
     * or greater than or equal to size()
     */
    E remove(int pos);
}
```

Use - `ListADT`

→ Assume `myList` is a ListADT. **What does the following code fragment do in general?**

```
for (int i = 0; i < myList.size(); i++) {
   myList.remove(i);
}
```

REASONING CAN BE FAULTY
TRACING CODE IS MORE RELIABLE!

myList a b c d e i 0 size 5
b c d e i 1 4
b d e i 2

Use - `ListADT`

→ Assume `myList` is a ListADT. **Write a code fragment** to reverse the contents of `myList` without using any additional ListADTs or other data structures (e.g., array).

myList FLIP INITIAL

PLIF DESIRED RESULT

```
for (int i=1; i<myList.size(); i++)
    myList.add(0, myList.remove(i));
```

FLIP i=1 size 4

LFIP i=2 size 4

PLFI i=3.

CS 367 Announcements
Tuesday, September 15, 2015

Homework h1 due 10 pm Friday, September 18th → NO HANDIN DIRS YET
Program p1 due 10 pm Sunday, September 27th (get started now!)

We'll discuss assignment submission Thursday. ~ 1266 CS
Assignment questions? Post on Piazza or see a TA during lab consulting hours.

Post a *private* note on Piazza in the "econflicts" folder by this Friday, 9/18, if you have a conflict with any of our exam dates. Specify which exam and an explanation:

- course/exam – include course name/number, time, instructor name and email
- VISA – include the accommodation(s) requested
- other – include concise explanation

Email skrentny@cs.wisc.edu by this Friday, 9/18, if you:

- **Have a VISA** from the McBurney Disability Resource Center.
  Email with subject "CS 367 VISA" and request an appointment.
- **Participate in religious observances** that may interfere with course requirements.
  Email with subject "CS 367 Religious Observance" and include the date and explanation.

Last Time

Course Topics
Implementing the Bag ADT

- casting when using Object
- using Java generics for generality

List ADT Intro

Today

List ADT

- coding the ListADT as a Java interface (from last lecture)
- using lists via the ListADT (from last lecture)
- implementing the ListADT using an array (SimpleArrayList)

Java API Lists
Iterators Intro

Next Time

Read: finish *Lists*, start *Exceptions*
Iterators

- iterators and the Java API
- using iterators
- options for implementing iterators
- making a class iterable

Exceptions Review

- throwing
- catching/handling.

Recall the List ADT

Concept

A List is a general, position-oriented container with zero-based indexing that stores a contiguous collection of items where duplicates are allowed and maintains relative ordering

Operations

```
void add(E item);
void add(int pos, E item);
E get(int pos);
E remove(int pos);
boolean contains(E item);
int size();
boolean isEmpty();
```

Issues

Null item – detect then signal with `IllegalArgumentException`
Bad position – detect then signal with `IndexOutOfBoundsException`
Empty list – handle as a bad position

Implementation - `ListADT` using a Generic Array

```
public class SimpleArrayList<E> implements ListADT<E> {

    private E[] items;       //the items in the List
    private int numItems;    //the # of items in the List

    public SimpleArrayList() {

    }

    //*** required ListADT methods ***

    public void add(E item) { ... }

    public void add(int pos, E item) { ... }
    public E remove(int pos) { ... }
    public E get (int pos) { ... }

    public boolean contains (E item) { ... }

    public int size() { ... }
    public boolean isEmpty() { ... }

    //*** additional optional array list methods ***

}
```

Implementing `contains`

→ **Complete the method below** so that it returns true iff the given `item` is in the list.

```
public boolean contains(E item) {
```

```
for (int i=0; i < Si ... .size(), i++){
{ E temp = ...  item.size[i]}
if temp.equals(item)
return true;
}
return false;
} if (item == null) throw new
                    IllegalArgumentException
for (int i = 0; i < numItems; i++)
    if ( items[i].equals(item))
        return true;)
    return false;
```

Implementing add at end

→ **What problem might occur with the following implementation:**

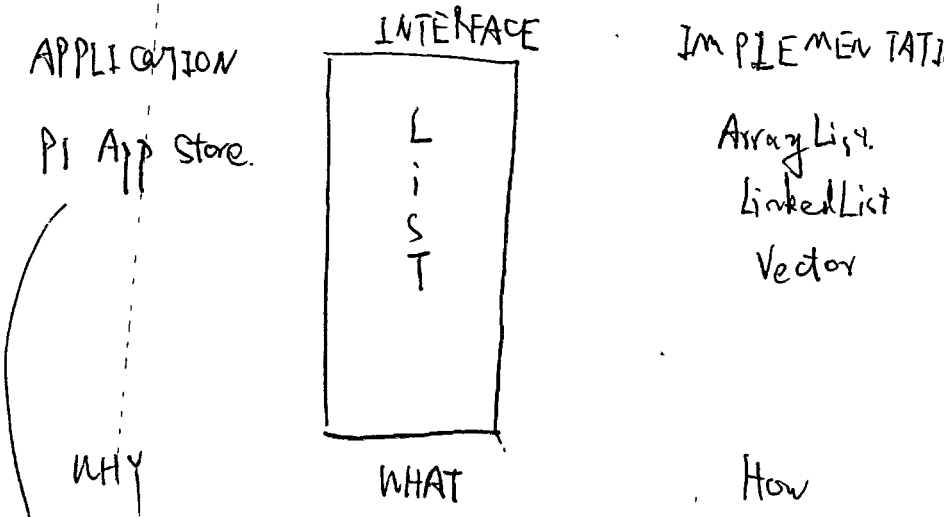
```
public void add(E item) {

   items[numItems] = item;
   numItems++;

}
```

end

if (item == null)
throw new IllegalArgumentException();

if (items.length == numItem
expandArray(); ← Readings

Java API Lists




```
List<String> words = new
INTERFACE ArrayList
                <String>().
                Implementation.
```

Design - Iterators

What are they?

- Objects used to step through A collection of Items
- An abstraction of A pointer
  - stores its position in a particular collection
  - position can change
  - item AT position can be accessed.
- ITS EXTERNAL / separate FROM its container
  - Has its own interface & implementation
  - user can instantiate as many iterators as desired to access the items in the collection

Concept

① Get iterator from a particular container storing the collection we desire to step through.

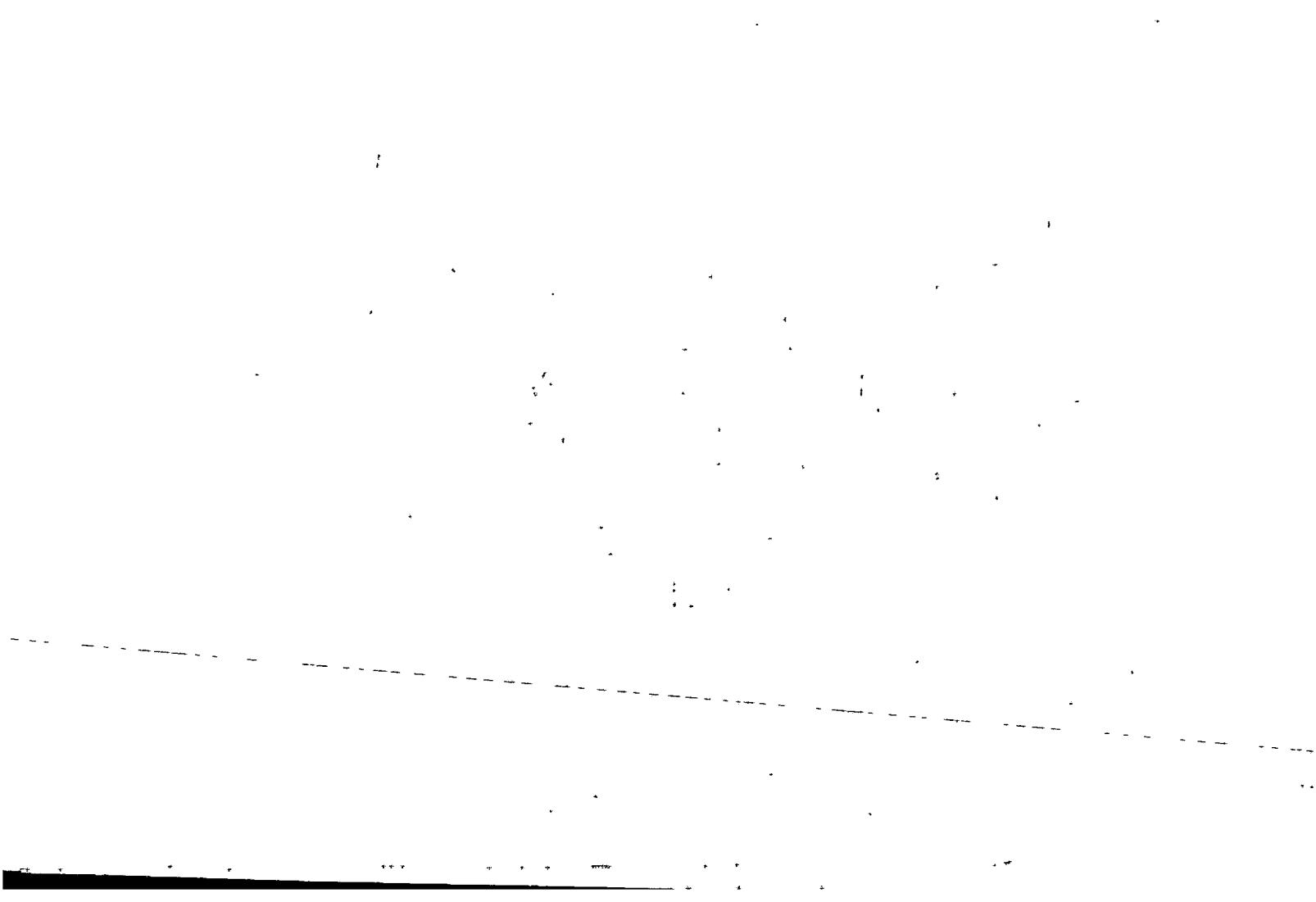
② Use Iterator to access the collection of Items in the container.

Operations

① Container class Operation that makes and returns An iterator.

② iterator class Operations that determine

- If Has ANOTHER ITEM.
- Access ITEM AT THE CURRENT position THEN ADVANCES THE ITERATOR TO NEXT POSITION



CS 367 Announcements
Thursday, September 17, 2015

Homework h1 due by 10 pm tomorrow, September 18th
- make sure your file is a pdf
- make sure you use the name h1.pdf
- submit to your `in` handin directory
- remember homeworks are to be done individually
- remember that late work is not accepted

Homework h2 assigned late tomorrow

Program p1 due 10 pm Sunday, September 27th (get started now!)

Assignment questions? Post on Piazza or see a TA during lab consulting hours.

By tomorrow, 9/18:
- Post a private note on Piazza in the "econflicts" folder if you have an exam conflict.
- Email skrentny@cs.wisc.edu if you have a VISA or participate in religious observances that may interfere with course requirements.

Last Time

List ADT
- coding the ListADT as a Java interface
- using lists via the ListADT
- implementing the ListADT using an array (SimpleArrayList)

Java API Lists

Iterators Intro

Today

Iterators
- designing iterators (from last time)
- coding iterator interfaces and the Java API
- using iterators
- making a class iterable
- options for implementing iterators

Submitting homeworks

Next Time

Read: *Exceptions*

Exceptions Review
- throwing
- handling
- execution
- practice with exception handling
- throws and checked vs. unchecked
- defining

Interfaces - Iterators in Java API

`Iterable<T>` interface in `java.lang`

specifies the operation to get an iterator to step through a collection:

(1)

- Iterator<T> iterator()

generic types

"Pointing to"

- RETURNS AN ITERATOR INITIATE TO THE FIRST ITEM IN THE COLLECTION. (OR "END MARK" IF THE Container is empty
- THIS METHOD is implemented in the container class.

`Iterator<E>` interface in `java.util`

(2) specifies the operations that iterators can do:

- boolean hasNext()
  RETURN TRUE IF ITERATOR is pointing at An item
- E next()
  RETURNS A REF TO THE CURRET ITEM AND ADVANCES THE ITERATOR TO NEXT position.
- void remove() //"optional"

How Do we make An operation optional?

THESE METHODS ARE IMPLEMENTED IN THE ITERATOR

Making Array Bags `Iterable`

Approach in Readings - Modify the `BagADT` Interface

```
import java.util.*;
public interface BagADT<E> {
    void add(E item);
    E remove() throws NoSuchElementException;
    boolean isEmpty();
    Iterator<E> iterator();
}
```

IN JAVA THERE'S A SEPERATE INTERFACE NAMED Iterable.

Also Modify the `ArrayBag` Class

```
import java.util.*;
public class ArrayBag<E> implements BagADT<E>, Iterable<E> {

    // *** Data members (fields) ***
    private E[] items;
    private int numItems;
    private static final int INIT_SIZE = 100;

    //*** required BagADT methods ***
    public void add(E item) { ... }
    public E remove() throws NoSuchElementException { ... }
    public boolean isEmpty() { ... }

    public Iterator<E> iterator() {
        return new ArrayBagIterator<E>( );
    }
}
```

we need to implement

need to pass arguments to properly initialize iterator

How would we do this using Java's approach?

INDIRECT use this.

DIRECT use.

Use - Iterators

Suppose `words` is a `SimpleArrayList<String>` that implements the `Iterable` Interface.
→ **Write a code fragment** that gets an iterator, named `itr`, from `words`.

① Iterator <String> itr
= words.iterator();

Suppose `words` is a `SimpleArrayList<String>` and `itr` is an iterator for `words`.
→ **Write a code fragment** that uses `itr` to print each item in `words`.

② while(itr.hasNext()); while (itr.hasNext()) {
S.O.println(itr.next()); Sys.O.p itr.next()
}

→ **Next write a code fragment** that uses `itr` to print the length of each item in `words`.

While (itr.hasNext()),
itr = words.iterator(); S.O.println(itr.next().length());
// Start again;

* These basic iterators have one-TIME use to go through the items in the container, you'll need to ask for another iterator.

Use - Iterators

Assume `SimpleArrayList<String>` implements the `Iterable` Interface.
→ **Challenge: Complete the method** using iterators to determine `list` contains duplicates.

```
public boolean hasDups(SimpleArrayList<String> list) {
    Iterator<String> currItr = list.iterator();
    while (currItr.hasNext()) {
        String currStr = currItr.next();
        Iterator<String> dupItr =
                list.iterator();
        while (currStr != dupItr.
                next())
        while (dupItr.hasNext()){
        if (currStr.equals(dupItr.
                next())){
            return true;
        }
    }
    return false;
}
```

POST
saturday
Afternoon
on PIAZZA
A solution
Remove Sunday
at MIDNIGHT

Implementation - Options for Iterator Classes

Indirect Access

- ITERATOR USES THE CONTAINER'S OPERATIONS TO ACCESS the items in a collection.
- Container class constructs the iterator by passing A Ref. to itself (this)
  * Indirect ACCESS is safer
  - Only works if the container class has sufficient operations so that iterator can throw the collection. vie then to step
- Iterator goes right to container's private instance vars to access the items in the collection.
- Container class constructs the iterator by passing The instnl vars needed by the iterator For Direct Access.
  * DIRECT ACCESS is FASTER!

Direct Access

Implementation - Indirect Access SimpleArrayListIterator Class

```
import java.util.*;
public class SimpleArrayListIterator<E> implements Iterator<E>
    Private SimpleArrayList<E> myList;
    private int curPos;

    public SimpleArrayListIterator( SimpleArrayList<E> list) {
            myList = list;
            curPos = 0;
    }

    public boolean hasNext() {
            return curPos < myList.size();
    }

    public E next() {
        if (curPos >= myList.size())
            throw new NoSuchElementException();
        E result = myList.get(curPos);
          curPos++;
            return result;
    }
    public void remove() {
            throw new UnsupportedPositionException
                ();
    }
}
```

Implementation - Direct Access `ArrayBagIterator` Class

```
import java.util.*;
public class ArrayBagIterator<E> implements Iterator<E> {

    private E[] myItems;
    private int myNumItems;
    private int currPos;

    public ArrayBagIterator( E[] items, int numItems ) {
        myItems = items;
        myNumItems = numItems;
        currPos = 0;
    }

    public boolean hasNext() {
        return currPos < myNumItems;
    }

    public E next() {
        if (currPos >= myNumItems) throw new NoSuchElementException();
        E result = myItem[currPos];
        currPos++;
        return result;
    }

    public void remove() {
       throw new UnsupportedOperationException();
    }
}
```

→ **Could we code this as an indirect access iterator instead?**

No. Array bags don't provide size or a way it means to get items.

CS 367 Announcements
Tuesday, September 22, 2015

Homework h1 due 10 pm Friday, September 25th
Program p1 due 10 pm Sunday, September 27th (time is running out!)

We'll discuss program submission Thursday.
Assignment questions? Post on Piazza or see a TA during lab consulting hours.

Last Time

Iterators

- coding iterator interfaces and the Java API
- using iterators
- making a class iterable
- options for implementing iterators

Today

Iterators

- options for implementing iterators (from last time)

Exceptions Review

- throwing
- handling
- execution
- practice

Next Time

Read: finish *Exceptions*, start *Linked Lists*

Exceptions Review

- throws and checked vs. unchecked
- defining

Chains of Linked Nodes

- Listnode class
- practice

Compiling & Submitting Programs

Exception Throwing – Signaling a Problem

WHEN a problem is detected we (or RTE) CAN signal a problem has occurred by throwing AN Exception.

* when an exception is thrown, NORMAL EXECUTION ENDS, AND SWITCH TO Exception HANDLING EXECUTION;

Java Syntax

```
throw exceptionObject;
```

Example

```
// could ADD TO ArrayBag's remove()
if (numItems <= 0)  <- DETECT
    throw new EmptyBagException(); <- signal
```

Exception Handling – Resolving a Problem

TRY - CATCH STMT:

① try code that might be problematic,
② catch EXCEPTIONS to handle particular problems.

Java Syntax

```
try {
   // try block
   code that might cause an exception to be thrown

} catch (ExceptionType1 identifier1) {
   // catch block
   code to handle exception type 1

} catch (ExceptionType2 identifier2) {
   // catch block
   code to handle exception type 2

}
... more catch blocks

finally {
   // finally block - optional
   code always executed when try block is entered
}
```

Exception HANDLER

optional catch

MULTIPLE CATCHES ARE OKAY

optional

Example

```
// main() code for ArrayBag Tester
try { for(int i=0; i<1000; i++)
        bag1.remove(1);
}
catch (EmptyBagException e){
   System.out.println("Remove Resulted in An EmptyBag Exception");
}
```

Exception Execution

Normal Execution
- Start: top of main() FLOW THROUGH THE CODE IN THE NORMAL MANNER
- Execute: NORMAL CODE, CODE IN TRY BLOCKS, CODE IN ANY FINALLY BLOCKS whose TRY WAS ENTERED.
- Skip: CODE IN CATCH BLOCKS.
- Switch to Exception Handling Execution
  WHEN AN EXCEPTION is thrown.

Exception Handling Execution
- Skip: NORMAL CODE INCLUDING ANY REMAINING CODE IN TRY BLOCK'S THAT WERE ENTERED.
- Execute: DO CODE IN FINALLY BLOCKS whose try was entered.
- Switch back to Normal Execution
  ELSE TERMINATE THE method WHEN THE EXCEPTION is caught.

Searching for a Matching Catch

Locally
IF THE CODE CAUSING THE EXCEPTION is IN a try Block THEN CHECK its CATCHES FOR A MATCH

2. Remotely
IF AN Exception is returned from A METHOD call that is in A try block then check its CATCH FOR A MATCH

ELSE TERMINATE THE METHOD AND THROW THE EXCEPTION TO ITS CALLER;

CHECKING FOR A MATCH DONE FROM TOP To Bottom of CATCHES.

a. Match Found
- switch Back to normal execution.
- execute code in the CATCH BLOCK AND corresponding FINALLY BLOCK (IF ANY)
- CONTINUE EXECUTION After That try-catch STMT

b. No Match Found
- Execute code in corresponding FINALLY BLOCK (If Any)

IF MAIN() THROWS AN Exception to its caller. TERMINATE THE METHODS AND Throw the Exception to RTE, THEN THE RTE TERMINATE THE PROGRAM [CRASH!] AND Display AN Exception message.

ExceptionTester Example

```
public class ExceptionTester {

   public static void main(String[] args) {
      System.out.print("main[");
      try {
         methodA( ); System.out.print("after A,");
         methodE( ); System.out.print("after E,");
      } catch (RedException exc) {
         System.out.print("main-red,");
      } catch (GreenException exc) {
         System.out.print("main-green,");
      } finally {
         System.out.print("main-finally,");
      }
      System.out.println("]main");
   }

   private static void methodA( ) {
      System.out.print("\nA[");
      try {
         methodB( );
         System.out.print("after B,");
      } catch (BlueException exc) {
         System.out.print("A-blue,");
      }
      System.out.println("]A");
   }

   private static void methodB( ) {
      System.out.print("\nB[");
      methodC( );
      System.out.print("after C,");
      try {
         methodD( );
         System.out.print("after D,");
      } catch (YellowException exc) {
         System.out.print("B-yellow,");
         throw new GreenException();
      } catch (RedException exc) {
         System.out.print("B-red,");
      } finally {
         System.out.print("B-finally,");
      }
      System.out.println("]B");
   }
```

ASSUME NO OUTPUT

What is Output When:

1. no exception is thrown

```
main[
A[
B[ after C, after D, B-finally ]B
, after B, ]A
, after A, after E, main-finally ]main
```

2. methodE throws a YellowException?

```
main[
A[
B[ after C, after D, B-finally, ]B
, after B, ]A
, after A, main-finally, CRASH!
```

3. methodC throws a GreenException?

```
main[
A[
B[
```

4. methodD throws a GreenException?

```
main[
A[
B[
```

What is Output When:

5. `methodC` throws a `RedException`?

```
main[
A[
B[
```

6. `methodD` throws a `RedException`?

```
main[
A[
B[
```

7. `methodD` throws a `YellowException`?

```
main[
A[
B[ after c, B-yellow, B-finally, main-green, main-finally,]
```

8. `methodD` throws a `OrangeException`?

```
main[
A[
B[
```

What is Output When:

9. `methodC` throws a `YellowException`?

```
main[
A[
B[
```

10. `methodC` throws a `BlueException`?

```
main[
A[
B[
```

11. `methodE` throws a `RedException`?

```
main[
A[
B[
```

throws clause – Passing the Buck

Checked vs. Unchecked

unchecked: For progs that can & should be avoided with careful programming
eg: NullPointerException, IndexOutOfBoundsException

checked: For progs that cannot be avoided with careful programming
compiler checks that programmer is aware — By looking for 1 or 2 ① catch block ② throws clause

Java Syntax

```
... methodName(parameter list)
      throws ExceptionType1, ExceptionType2, ... {
   ...
}
```

Example

```
public static void main(String[] args) throws IOException { ...
```

Defining a New Exception Class

Checked

```
public class MyException extends Exception {
    // EMPTY
}
```

Unchecked

```
public class MyException extends RuntimeException {
    // EMPTY
}
```

Example

```
public class EmptyBagException extends Exception {

    public EmptyBagException() {
        super();
    }

    public EmptyBagException(String msg) {
        super(msg);
    }

}
```

```
if (...)
  throw new EmptyBagException ("remove caused
                                this exception");
```

CS 367 Announcements
Thursday, September 24, 2015

Homework h2 due by 10 pm tomorrow, September 25th
- make sure your file is a pdf but not pdf scan of written work or pdf of a screen shot
- make sure you use the name h2.pdf
- submit to your `in` handin directory
- remember homeworks are to be done individually
- remember that late work is not accepted

Homework h3 assigned Saturday

Program p1 due 10 pm Sunday, September 27th
- submit java files to your `in` directory
- make sure to name your source files as specified in the submission section
- verify that you've submitted the correct files (ls, more, javac, java)

Program p2 assigned in the next day or 2, pair programming on p2 is allowed

Assignment questions? Post on Piazza or see a TA during lab consulting hours.

Last Time

Iterators
- options for implementing iterators (from last time)

Exceptions Review
- throwing
- handling
- execution
- practice

Today

Exceptions Review
- practice (from last time)
- throws and checked vs. unchecked
- defining

Chains of Linked Nodes
- Listnode class
- practice

Compiling & Submitting Programs

Next Time

Read: continue *Linked Lists*

Chains of Linked Nodes
- practice

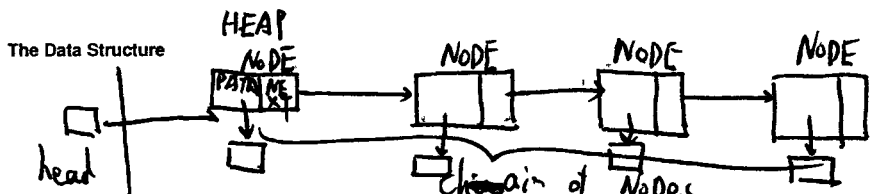
Java Visibility Modifiers

LinkedList Class

LinkedListIterator Class

means it is the last NODE

Chain of Linked Nodes Data Structure



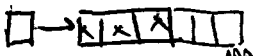
REFs THE FIRST NODE IN CHAIN

ENABLE STORING A COLLECTION of ITEMS

Empty chain. head

Array vs. Chain of Nodes

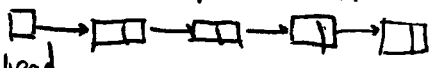
STATIC. D.S



CANNOT ADD ELEMS TO AN ARRAY!

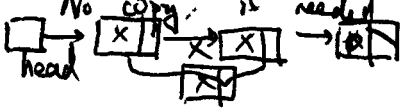
INSTEAD WE make A new LARGER ARRAY AND COPY

DYNAMIC D.S



can add Nodes by just linking them on

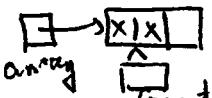
No copying is needed



CAN INSERT NEW NODES just link it in

No shifting needed.

Goal



INSTEAD you must shift items over to make space

Implement ADTs using A chain of NODES as the Data STRUCTURE RATHER THAN AN ARRAY

Listnode Class

```
class Listnode<E> {

  private E data;
  private Listnode<E> next;


  public Listnode(E d) {
    this(d, null);
  }

  public Listnode(E d, Listnode<E> n) {
    data = d;
    next = n;
  }


  public E getData()                  { return data; }

  public Listnode<E> getNext()        { return next; }


  public void setData(E d)            { data = d; }

  public void setNext(Listnode<E> n) { next = n; }
}
```

self data structure

Practice: Using Listnodes

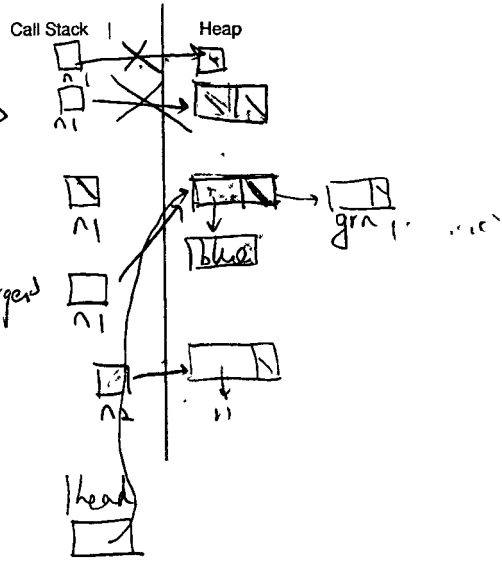
→ **Draw a memory diagram** corresponding to the given code:

assume code is in main()

```
Listnode<String> n1 = null;
```

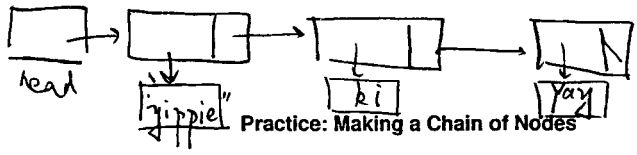
n1 = new Listnode<String>("blue")

Listnode<Integer> n2 = new Listnode<Integer>(11);



→ **Write the code that results in:**

Listnode<String> head = n1;
head.setNext(new Listnode<String>("green"));
head = null;



Practice: Making a Chain of Nodes

→ **Create a chain of `ListNodes` containing the `Strings`** "yippie", "ki", and "yay" (in that order) in as few statements as you can.

```
head = new ListNode<String>("yippie");
head.setNext(new ListNode<String>("ki"));
head.setNext(new ListNode<String>("yay"));
```

1 STMT

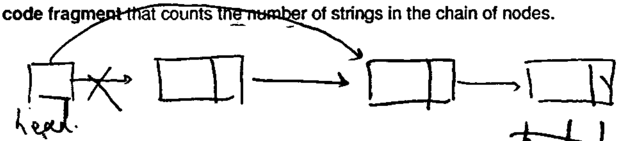
```
ListNode<String> head =
    new ListNode<String>("yippie",
        new ListNode<String>("ki",
            new ListNode<String>("yay")));
```

* THIS APPROACH DOESN'T WORK WELL WITH LONG CHAINS, it's better to implement A List ADT CLASS AND ADD THE NODE ONE-BY-ONE

Practice: Traversing a Chain of Nodes

Assume **head** points to the first node in a chain of **Listnodes** containing **Strings**.

→ **Write a code fragment that counts the number of strings in the chain of nodes.**



COUNT is 3

~~head.length(); X.~~

ListNode head = new List Node <String>

if(headnode.hasnext() count++;)

while (new List h())

count++

```
int count = 0;
while (head != null {
        count ++;
    head =head.getnext();
}
```

* when traversing a chain of nodes use ANOTHER REFERENCE variable. so you don't mess up head and lose the chain

```
ListNode <String> curr =head;
while (curr != null ){
    count++;
    curr = curr. getNext ();
}
```

Practice: Adding a Node at the Chain's End

Assume `head` points to the first node in a chain of `Listnodes` containing `Strings`.

→ **Write a code fragment** that adds a node containing "rear" to the end of the chain of nodes. You may assume the chain has at least one item.

```
ListNode<String> tmp = new ListNode<String>("rear");
tmp.setNext(head.getNext());
head.setNext(tmp);
```

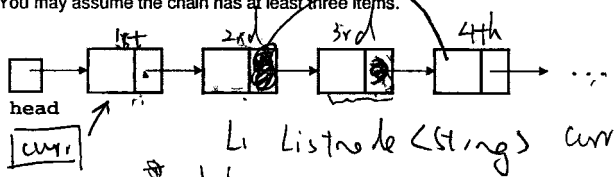
① Traverse To node at END of chain

```
ListNode<String> curr = head;
while (curr.getNext() != null) {
    curr = curr.getNext();
}
curr.setNext(new ListNode<String>("rear"));
```

Practice: Removing a Node from a Chain

Assume `head` points to the first node in a chain of `Listnodes` containing `Strings`.

→ **Write a code fragment** that removes the third item from the chain of nodes.
You may assume the chain has at least three items.



ListNode<String> curr = head

for (int i=0; i<2 i++)
{ curr = curr.getNext();
}

head.getNext().
setNext(

head.getNext().
getNext().
getNext())

curr.setNext(curr.getNext().getNext())

(curr.getNext())(curr.getNext()))

get messy with long chains and it doesn't generalize!

→ **How would you generalize your code** so it removes the Nth item from the chain of nodes?

ListNode<String> tmp = l;

① use A curr reference to traverse to (n-th-1) node
while (tmp.getNext() != n)
tmp = tmp.getNext();

② set the next Field of the $N^{th}-1$ NODE to tmp.setNext(n.getNext()),
be The value in the next field of the nth node.

Practice: Challenge Question

Assume `head` points to the first node in a chain of `Listnodes` containing `Strings`.

→ **Write a code fragment** that reverses the order of the nodes in the chain.

Command Line Java Development

Download

Like Pine in hl
Submit to in
But CAN also
download to your Private directory

pwd prints working directory
ls list directory contents
cd changes directory

Edit

Vim
Emacs.

cat *.java
more java
→ Vim. java

Compile

javac *.java

Run

java Main class Name
NOTE Don't include ".class"

Submit

cp To copy
cp <sourse> <destination>

ls !$

CS 367 Announcements
Tuesday, September 29, 2015

Homework h3 due 10 pm Friday, October 2nd
Program p2 due 10 pm Friday, October 23rd

Assignment questions? Post on Piazza or see a TA during lab consulting hours.

Last Time
Exceptions Review
- practice (from last time)
- throws and checked vs. unchecked
- defining

Chains of Linked Nodes
- Listnode class

Compiling & Submitting Programs

Today
Chains of Linked Nodes
- practice (from last time)

Java Visibility Modifiers
LinkedList Class

Next Time
Read: continue *Linked Lists*
LinkedListIterator Class
Linked List Variations
- tail reference
- header node
- double linking
- circular linking

Java Visibility Modifiers

public `public class ArrayList<E>`

private `private Object[] items`

protected `protected String name`

package `class Listnode<E>`

LinkedList - Implementing ListADT using a Chain of Nodes

```
public class LinkedList<E> implements ListADT<E> {

   private Listnode<E> head;
   private int numItems;



   public LinkedList() {
                head = null;
                numItem = 0.0;
   }

   public void add(E item) {
```


LinkedList (cont.)

```
public class LinkedList<E> implements ListADT<E> {

   private Listnode<E> head;
   private int numItems;

   public LinkedList() { ... }
   public void add(E item) { ... }


   public E get(int pos) {
```

CS 367 Announcements
Thursday, October 1, 2015

Homework h3 due by 10 pm tomorrow, October 2nd
- make sure your file is a pdf but not pdf scan of written work or pdf of a screen shot
- make sure you use the name h3.pdf
- submit to your i-n handin directory
- remember homeworks are to be done individually
- remember that late work is not accepted

Homework h4 assigned Sunday

Program p2 due 10 pm Friday, October 23rd

Last Time

Chains of Linked Nodes
- practice

Today

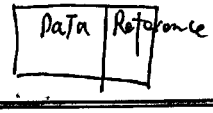
Java Visibility Modifiers (repeats)
LinkedList Class (repeats)
LinkedListIterator Class
Linked List Variations
- tail reference
- header node
- double linking
- circular linking

Next Time

Read: finish *Linked Lists*, start *Complexity*
Linked List Variations (finish)
Complexity
- Intro
- big-O notation
- analyzing algorithms practice
- analyzing Java code
- analyzing Java code practice

Recall Chain of Linked Nodes Data Structure

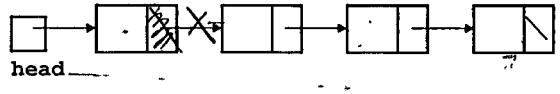
Listnode



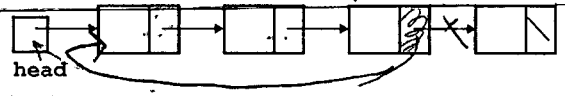
Listnode class

```
class Listnode<E> {

  private E data;
  private Listnode<E> next;

  public Listnode(E d)                 { . . . }
  public Listnode(E d, Listnode<E> n){ . . . }
  public E getData()                   { return data; }
  public Listnode<E> getNext()         { return next; }
  public void setData(E d)             { data = d; }
  public void setNext(Listnode<E> n) { next = n; }
}
```

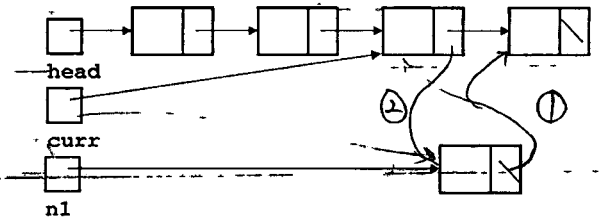
→ **Show how the memory diagrams change** as a result of executing the code beneath each:




```
head.setNext(head.getNext().getNext().getNext().getNext());
```




```
head.getNext().getNext().setNext(head);
```




```
n1.setNext(curr.getNext());
curr.setNext(n1);
```

Java Visibility Modifiers

public — `public class ArrayList<E>`

Accessible to any code

private — `private Object[] items`

Accessible only to code within the same scope (E.G., class)

protected — `protected String name`

Accessible only to code within the same class or any subclasses of that class's package

package — `class Listnode<E>`

↑ MEANS package access

accessible only to code within the same package

(FOLDER, DIRECTORY)

GOAL: HIDE D.S. FROM OTHERS

STEPS: ① CREATE A FOLDER LinkedList

② PUT IN IT:
- Listnode.java
- LinkedList.java
- LinkedListIterator.java

(Package. / Public)

③ User must:
import linked list *

LinkedList - Implementing ListADT using a Chain of Nodes

```
public class LinkedList<E> implements ListADT<E> {

  private Listnode<E> head;
  private int numItems;

  public LinkedList() {
    head = null;
    numItems = 0;
  }

  public void add(E item) {
    // CHECK for invalid item
    if (item == null) throw new IllegalArgumentException();

    // CREATE A NEW LISTNODE
    ListNode<E> newNode = new ListNode<E>(item);

    // SPECIAL CASE: EMPTY
    if (head == null) { head = newNode; }
    else {
      // GENERAL CASE: NOT EMPTY
      // TRAVERSE TO LAST NODE IN CHAIN
      ListNode<E> curr = head;
      while (curr.getNext() != null)
        curr = curr.getNext();
      // LINK IN NEW NODE
      curr.setNext(newNode);
    }
    //COUNT THE NEW ITEM
    numItems++;
  }
```

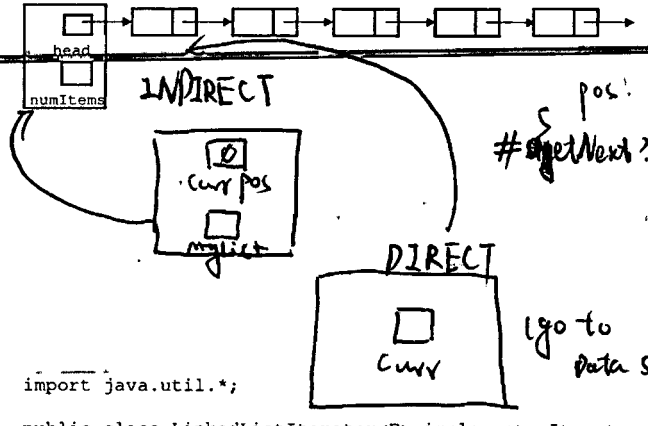
Access works if Listnode & LinkedList in the same package

LinkedList (cont.)

```
public class LinkedList<E> implements ListADT<E> {

  private Listnode<E> head;
  private int numItems;

  public LinkedList() { ... }
  public void add(E item) { ... }

  public E get(int pos) {
    //CHECK FOR AN INVALID POSITION
    if (pos < 0 || pos >= numItems )
        throw new IllegalArgument OutofBoundsException();
    // TRAVERSE TO POSITION
    ListNode<E> curr = head;
    for ( int i = 0; i < pos; i++ )
        curr = curr.getNext();
    //RETURN DATA IN curr
    RETURN curr.getData();
```


Implementing LinkedListIterator

→ Should an indirect or a direct iterator implementation be used with a LinkedList?



see for loop

pos: 0 1 2 3 4 5

getNext? 0+1+2+3+4+5 ...

growing

INDIRECT TRAV
using ge
ALWA
STAR
AT 0

#getNexts
+1+1+1+1+1

CONSTAT
DIRECT
FASTER
SINCE
ITERATO
knows wh
NODE it's a

```
import java.util.*;

public class LinkedListIterator<E> implements Iterator<E> {
  private ListNode<E> curr;

  LinkedListIterator( ListNode<E> head ) {
    curr = head;
  }
  public boolean hasNext() {
    return curr != null;
  }
  public E next() {
    if ( curr == null) throw new NoSuchElementException();
    E item = curr.getData();
    curr = curr.getNext();
    return item;
  }
  public void remove() {
    throw new UnsupportedOperationException();
  }
}
```

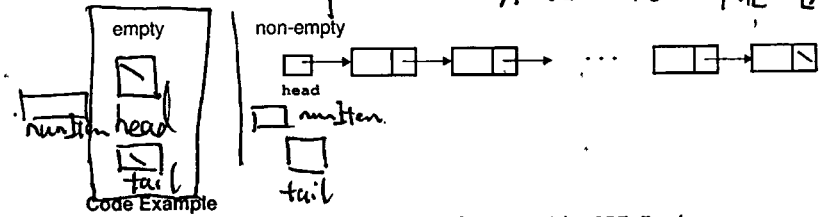
Making `LinkedList` Iterable

```
public class LinkedList<E> implements ListADT<E> , Iterable <E> {

   private Listnode<E> head;
   private int numItems;


   public LinkedList() { ... }
   public void add(E item) { ... }
   public E get(int pos) { ... }
      .
      .
      .

   public Iterator<E> iterator() {
          return new LinkedList Iterator (head);


   }
}
```


Tail Reference

Concept

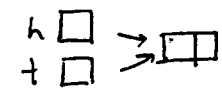
USED for DIRECT ACCESS TO THE LAST NODE



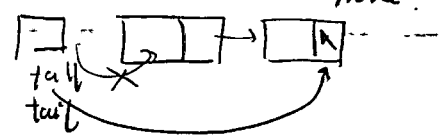
Code Example

```
public class LinkedList<E> implements ListADT<E> {

   private Listnode<E> head;
   Privat Listnode<E> tail;
   private int numItems;

   public LinkedList() {
      head = null;
      Tail = null;
      numItems = 0;
   }

   public void add(E item) {
      if (item == null) throw new IllegalArgumentExceptions();

      Listnode<E> newnode = new Listnode<E>(item);

      //Special Case: empty list
      if (head == null) {
         head = newnode;
            tail = newNode;
      }
      //General Case: non-empty list
      else {
         Listnode<E> curr = head;
         while (curr.getNext() != null)
            curr = curr.getNext();
         curr.setNext(newnode);
      }

      numItems++;
   }
```

1 ITEM



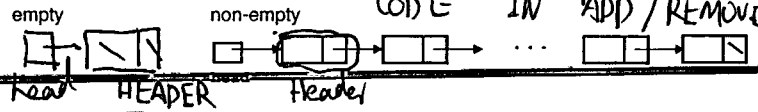
tail.setNext(newNode);
tail = newNode



* must make sure tail is properly updated at needed in other ADD/REMOVE OPERATIONS;

Header Node

Concept Used to ELIMINATE SPECIAL CASE CODE IN ADD / REMOVE



HEADER NODE DOES NOT STORE DATA

Code Example

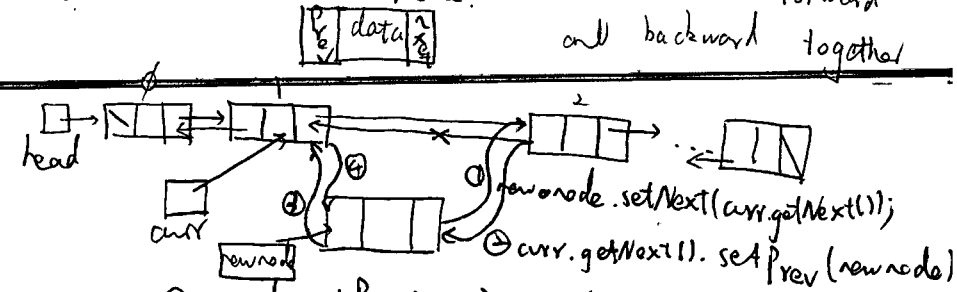
```
public class LinkedList<E> implements ListADT<E> {

   private Listnode<E> head;
   private int numItems;

   public LinkedList() {
      head = null; new Listnode<E>(null); // Make header node
      numItems = 0;
   }

   public void add(E item) {
      if (item == null) throw new IllegalArgumentExceptions();

      Listnode<E> newnode = new Listnode<E>(item);

      //Special Case: empty list
      if (head == null) {
         head = newnode;
      }
      //General Case: non-empty list
      else {
         Listnode<E> curr = head;
         while (curr.getNext() != null)
            curr = curr.getNext();
         curr.setNext(newnode);
      }

      numItems++;
   }
```

Works for all cases

* MAKE SURE YOUR OPERATIONS (e.g. get, contains, skip over the header node) WORK WITH tail REFERENCE

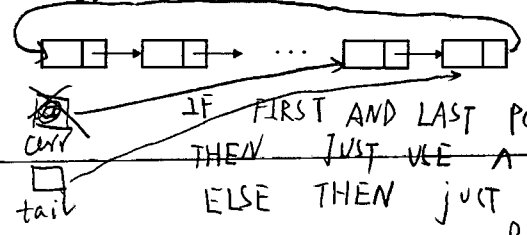
Double and Circular Linking

Doubly-Linked Chains of Nodes Dbl Linknode. Enables Both Forward and backward together



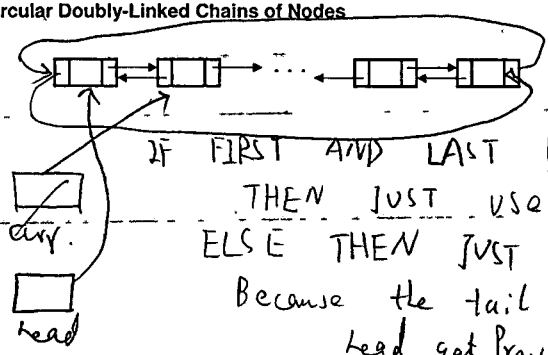
* common to combine this with a tail ref and/or header node.

Circular Singly-Linked Chains of Nodes



IF FIRST AND LAST POSITIONS DON'T MATTER THEN JUST USE A CURR REF
ELSE THEN JUST USE A tail Ref.
Because the head is at tail.getNext()

Circular Doubly-Linked Chains of Nodes



IF FIRST AND LAST POSITIONS DON'T MATTER
THEN JUST USE A curr REF.
ELSE THEN JUST USE A HEAD REF
Because the tail is at head.getPrev();

garage collection.

CS 367 Announcements
Tuesday, October 6, 2015

Homework h4 due 10 pm Friday, October 9th

Program p2 due 10 pm Friday, October 23rd

Last Time

Java Visibility Modifiers (repeats)
LinkedList Class (repeats)
LinkedListIterator Class
Linked List Variations
- tail reference
- header node

Today

Linked List Variations
- double linking
- circular linking

Complexity
- concept
- big-O notation
- analyzing algorithms practice
- analyzing Java code
- analyzing Java code practice

Next Time

Read: finish *Complexity*

Complexity
- significance of scaling
- complexity caveats

Comparing ArrayList vs LinkedList
- shadow array - improving array resizing

Analyzing Algorithm Efficiency

Complexity

- Used to compare DIFF ALGOs/code THAT solved THE Same Prob.
- CHARACTERIZED HOW THE USE of RESOURCES SCALES AS THE PROBLEM SIZE grows.

Resources < TIME ie basic ops/ dominant ops
Space, is memory

Problem size – THE ASPECT of THE Problem when changing Affects the amount of resource used.

If problem size doubles and the # of operations:

STAY the same ⇒ constant TIME complexity

double ⇒ LINEAR T.C.

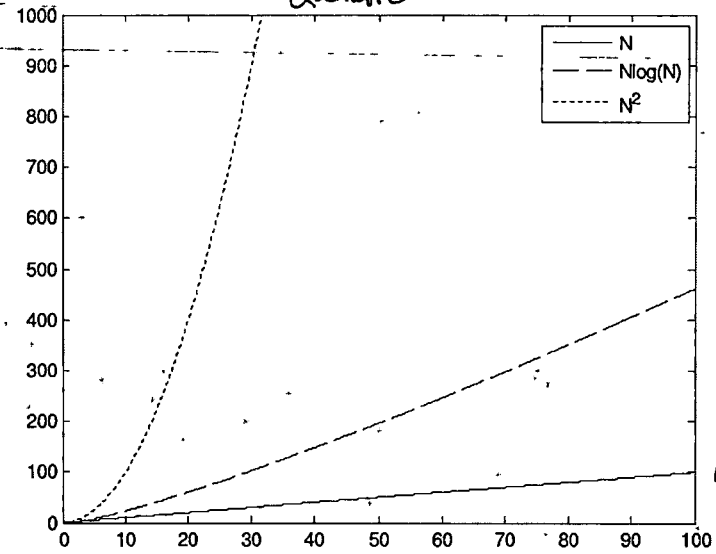
Quadruples ⇒ Quadratic T.C.

* Complexity Classifications

- A FIRST-ORDER Analysis. (where we focus on the most significant differences FIRST)

N vs. Nlog(N) vs. N^2

Quadratic



WORK. resource used.

Quasi Linear

Linear

N Problem size

Complexity analysis is NOT Concered about particular Problem sizes.

- is concerned with How the work/ recources used CHANGES As THE Problem size grows.

* Complexity does affect performance But not vise versa

Big-O Notation

Concept characterize functions by their growth rates

Grow rate funcs: 1, $\log N$, N, $N\log N$, N^2, N^3, N^c, 2^N, $N!$

low → High

constant complexity O(1)
linear complexity O(N)
quadratic complexity $O(N^2)$

> Read as: "order of..."

Simplifying Equations

TIME EQN $T(N) = AN^2 + 3N + 11$ (① ② ③)

① Drop Low order TERMS

② DROP MULTIPLICATIVE CONSTANTS ON HIGH DROP TERMS

Formal Definition

A FUNC $T(N)$ ← ACTUAL RESOURCE consumption FUNCTION

is $O(F(N))$ → growth rate functions

IF FOR SOME CONSTANT C AND SOME NUMBER N_0 such that

$T(N) \leq C * F(N)$ for all $N \geq N_0$

$T(N) = N+2$		$F(N) = N$		CHOOSE	
1	3	1	1		
2	4	2	2	$C=2$	$N_0=1$ OK
3	5	3	3	2	
4	6	4	4	4	
5	7	5	5	6	
6	8	6	6	8	
		7	7	10.	

Prob size must be LARGE enough

* cHoose ORF F THAT is LARGST.

Number Guessing Game

Picker picks a number between 1 and N
Repeat until number is guessed:
  Guesser guesses a number
  Picker answers "correct", "higher", or "lower"

problem size: N for the upper bounds
dominant operation: Guess - characterized the work/TIME

→ **What is the complexity of each algorithm below** that the guesser uses to decide the sequence of numbers to give as guesses? so well count it)

Algorithm 1:
  guess = 1
  repeat
    If guess incorrect, increment guess by 1
  until correct

BEST case:
1 Guess ⇒ O(1) (Secret No. is always 1)
WORST case: (Secret # is Always < N)

* WITH ANALYZING You WANT to DETERMINE IF THERE ARE N Guesses OR circumstances THAT BEST /AVG/ ⇒ O(N)
WORST CASES.

AVG Case (ALL Numbers are equally likely) CALL Numbers Once

$\frac{N}{2}$ Guess = O(N)

Algorithm 2:
  guess = N/2
  step = N/4
  repeat
    If guess is too large, decrease guess by step
    If guess is too small, increase guess by step
    step = step/2 (alternate rounding up/down)
  until correct

Ex: N=100 SECRET is 100

Guess: 50 75 88 94 97 99 100
Step 25 13↑ 6↓ 3 2↑ 1

Best Case:
N = 50
1 Guess ⇒ O(1) $\log_2 N$ ⇒ O(log N)

WORST Case
N=100
$\log_2 100 = 6.6$

Complexity of Java Code

Basic operations O(1) anything that isn't a control statement or a method call

Sequence of statements

MEANS sum

```
statement1;
statement2;
...
statementk;
```

O(1) + O(1) + ... + O(1) = O(K*1) = O(1)

O(N) + O(N²) + ... + O(N) = O(N²)

Left with high order term.

If-else

```
if (cond) {
  //if sequence of statements
}
else {
  //else sequence of statements
}
```

N_{cond}, N_{IF}, N_{ELSE}

$O(N_{cond}) + O(Max(N_{IF}, N_{ELSE}))$ worst case

$\equiv O(Max(N_{cond}, N_{IF}, N_{ELSE}))$ worst case

Complexity of Java Code (cont.)

Basic loops
→ What is the problem size based on?

```
for (i = 0; i < j; i++) {
  //sequence of statements
}
```

loop typically mean multiply. j.

Look for a var in loop whose value is unknown. And if changed affects the number of iterations.

$j * O(N_{STMTS})$

$= O(j * N_{STMTS}) \Rightarrow$ if $O(N_{STMTS})$ is $O(1)$ then $O(j)$

If $O(N_{stmts})$ is $O(j)$ then $O(j^2)$

Nested loops
→ What is the problem size based on? N, M

```
for (i = 0; i < N; i++) {
  for (j = 0; j < M; j++) {
    //sequence of statements
  }
}
```

$O(N * M * N_{STMTS}) =$ IF $N=M$ then $O(N^2 * N_{STEM})$

IF $N=5$ Then $O(M * N_{STEM})$

IF $N=$?

Keep all variables in your analysis in case they are related or discovered to be constants.

Loops with nested method calls (assume problem size based on N)

```
for (i = 0; i < N; i++) {
  f1(i);  //assume O(1)
}
```

$N * O(1)$

$= O(N * 1) = O(N)$

```
for (i = 0; i < N; i++) {
  f2(N);  //assume O(N)
}
```

$N * O(N)$

$= O(N * N) = O(N^2)$

```
for (i = 0; i < N; i++) {
  f3(i);  //assume O(i)
}
```

unroll loop

i: 0, 1, 2, 3, ..., N-1

$\sum_{i=0}^{N-1} f3(i) = 0 + 1 + 2 + \ldots + (N-3) + (N-2) + (N-1)$

$= \frac{N}{2}(N-1) = \frac{N^2}{2} - \frac{N}{2}$

$= O(N^2)$

Practice - Complexity of Java Code

method1
→ What is the problem size based on? A.length = N

```
public void method1(int[] A) {
  for (int i = 0; i < A.length - 1; i++)
    method2(A, i);
}
```

O(N) W. case.

$O(N^2)$

method2

```
public void method2(int[] B, int s) {
  for (int i = s; i < B.length - 1; i++)
    if (B[i] > B[i+1])
      method3(B, i, i+1);
}
```

B.length = N s = i

O(1) O(1) $> O(1)$ Assume worst case

s = ∅.

Then O(N)

method3

```
public void method3(int[] C, int x, int y) {
  int temp = C[x];
  C[x] = A[y];
  C[y] = temp;
}
```

O(1)

C.length = N

x = i

y = i+1

Practice - Complexity of Java Code

method4

→ What is the problem size based on? B.

```
public void method4(int Q) {
   int sum = 0, R = 1000;

   for (int i = Q; i >= 1; i--)
      for (int j = 0; j < R; j++)
         sum += j;
}
```

Q R. $O(Q \cdot R)$ $O(N)$

method5

→ What is the problem size based on? O

```
public void method5(int X) {
   int tmp, arr[];

   arr = new int[X];
   for (int i = 0; i < X; i++)
      arr[i] = X - i;

   for (int i = 0; i < X - 1; i++) {
      for (int j = i; j < X - 2; j++) {
         if (arr[j] > arr[j+1]) {
            tmp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = tmp;
         }
      }
   }
}
```

$O(X)$ X−1 X−2. $O(X^2)$

$O(1)$ $O(N^2)$

$\max(O(N^2), O(1))$

CS 367 Announcements
Thursday, October 8, 2015

Homework h4 due by 10 pm tomorrow, October 9th
- make sure your file is a pdf but not pdf scan of written work or pdf of a screen shot
- make sure you use the name h4.pdf
- submit to your `in` handin directory
- remember homeworks are to be done individually
- remember that late work is not accepted

Homework h5 assigned Sunday

Program p2 due 10 pm Friday, October 23rd

Last Time

Linked List Variations
- double linking
- circular linking

Complexity
- concept
- big-O notation
- analyzing algorithms practice

Today

Complexity
- analyzing Java code (from last time)
- analyzing Java code practice (from last time)
- significance of scaling
- complexity caveats

Comparing ArrayList vs LinkedList
- shadow array - improving array resizing

Next Time

Read: *Stacks and Queues*

Stack ADT
- concept
- array implementations
- chain of nodes implementations

Queue ADT
- concept
- chain of nodes implementations
- array implementations
- circular array

Returning N Papers to N Students

problem size (N) = Number of Students
dominant operation = Looking at name on paper ← account for work/time

→ What is the complexity of each algorithm below?

Algorithm 1:
call out each name,
have student come forward & pick up

best-case:
worst-case: > Can these be distinguished? NO! O(N)

Algorithm 2:
hand pile to first student,
student linearly searches through papers & takes hers/his,
pass pile to next student who does likewise

best-case: PAPER are ordered same as students
worst-case: Opposite ↗ $O(N^2)$
N N-1 ... 0.

Algorithm 3:
sort the papers alphabetically,
hand pile to first student who does binary search,
pass to next student who does likewise

The Significance of Scaling

factorial.

N	N log(N)	N^2	2^N	N!
2	2.0	4	4	2
4	8.0	16	16	24
6	15.5	36	64	720
8	24.0	64	256	40,320
10	33.2	100	1024	3,628,800
15	58.6	225	32,768	1.3×10^{12}
20	86.4	400	1,048,576	2.44×10^{18}
100	664.4	10,000	1.30×10^{30}	9.3×10^{157}
1000	9965.8	1,000,000	1.1×10^{301}	4.0×10^{2567}

Complexity Caveats

Small Problem Size

IF THE PROBLEM size is small, complexity is NOT a good basic for choosing algo's.

Same Complexity

- Now consider complexities of different resources.
- Consider the multiplicative constants and lower order terms.

Comparing `ListADT` Implementations

Space Requirements Dominant unit is one memory reference.

→ Problem size N is? Number of items.

Array: ~~Each element refs data.~~

O(length of the array) — MEMORY used is not necessarily proportional to number of items because we allocate extra space so we don't have to expand frequency.

Singly-Linked List: data + next

Circular Singly-Linked List: same

Doubly-Linked List: prev + data + next

Circular Doubly-Linked List: same.

memory used is proportional to number of items.

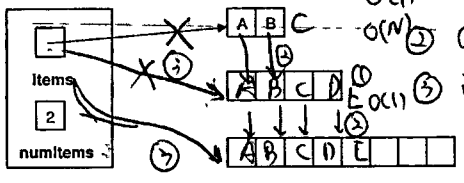
Comparing `ListADT` Implementations

Time Requirements
Problem size N is number of items

	constructor	add (E) "at end"	add (int,E) "at pos"	contains (E)	size	isEmpty	get (int)	remove (int)
Array	O(1)	W.C O(N) Expand	W.C. O(N) SHIFT	W.C O(N)	O(1) using numItems	O(1) using numItems == 0	O(1)	W.C O(N) SHIFT
Singly-Linked List (SLL)		O(1) tail ref	Expand W.C O(N) Traverse (Linking O(1))	linear search			W.C O(N) Traverse	W.C O(N) Traverse (unlinking O(1))
Circular SLL		O(1) with only tail.						
Doubly-LL		O(1) with tail						
CircularDLL	O(1)	O(1) with only head	W.C O(N)	O(N) W.C	O(1)	O(1)	O(N)	O(N)

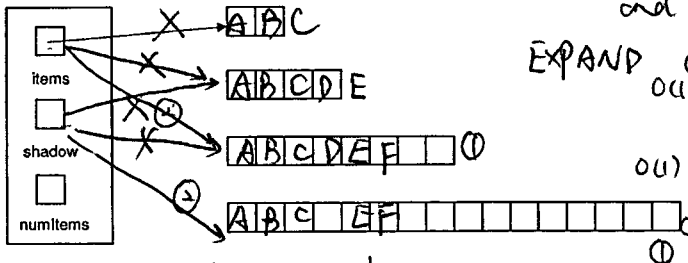
Shadow Array – Improving Array Resizing

"Naïve" Approach



EXPAND:
① CREATE NEW 2×LARGER ARRY O(1)
② COPY ITEMS FROM OLD to new array O(N)
③ items = larger. Array. O(1)

"Shadow Array" Improvement



ADD: add new item to items
and show arrays copy one item if shadow if needed
EXPAND ① CREATE A NEW O(1)
2× LARGER ARRAY
② items = shadow. O(1)
③ shadow = larger array O(1)

NEW expand is O(1) because
the cost of copying is spread
across multiple ADDS.

Comparing `ListADT` Implementations

Ease of Implementation

ALL EASY D.S

Array:	SHIFTING EXPANDING SHIFTING
Singly-Linked List:	
Circular Singly-Linked List:	
Doubly-Linked List:	
Circular Doubly-Linked List:	

GETTING THE LINKS RIGHT CAN BE A CHALLENGE.
* DRAW PICTURES TO CONFIRM YOUR CODE WORKS AS EXPECTED

CS 367 Announcements
Tuesday, October 13, 2015

Midterm Exam 1
- Tuesday, October 20, 5:00 pm
- Lec 1: B10 Ingraham Hall
- Lec 2: 6210 Social Sciences Building
- ~~UWID required~~
- See posted exam information

Homework h5 due 10 pm Friday, October 16th

Program p2 due 10 pm Friday, October 23rd

Last Time

Complexity
- analyzing Java code (from last time)
- analyzing Java code practice (from last time)
- significance of scaling
- complexity caveats

Today

Comparing ArrayList vs LinkedList (from last time)
- shadow array (from last time)

Stack ADT
- concept
- array implementations
- chain of nodes implementations

Queue ADT
- concept
- chain of nodes implementations
- array implementations

Next Time

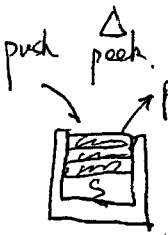
Read: start-*Recursion* (topic not on exam 1)

Queue ADT
- circular array

Recursion
- call stack tracing
- recursion vs. iteration
- concept

Stack ADT

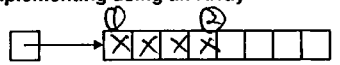
Concept



"FILO"
First in
Last out.

Operations - void push (E item)
E pop()
E peek()
boolean isEmpty
> throws EmptyStackException

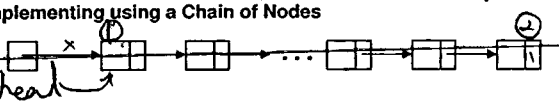
Implementing using an Array



→ Where should the top be located in the array? (3)

① PUSH O(N) SHIFT POP O(N) SHIFT
② PUSH O(1) POP O(1)

Implementing using a Chain of Nodes



→ Where should the top be located in the chain of nodes?

① PUSH O(1) POP O(1)
② PUSH O(N) TRAversal POP O(N) TRAversal.

Complexities

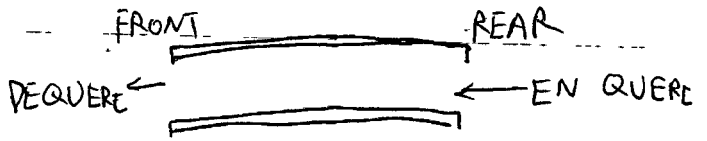
ARRAY - all ops O(1) when top is at rear and using a shadow

chain of nodes - all ops O(1) when top is at head.

* WHEN CHOOSING APPLYING P.S's THE OPERATIONS that matters most are those that change the collection.

Queue ADT

Concept like a checkout line FIFO



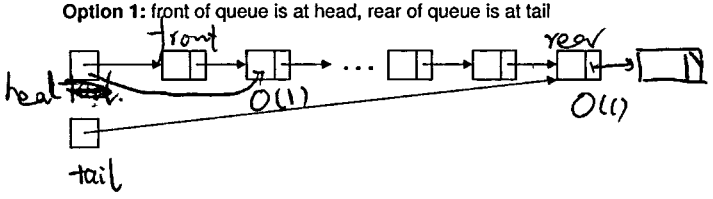
Operations

void enqueue (E item)
E dequeue () throws EmptyQueueException
boolean isEmpty ()

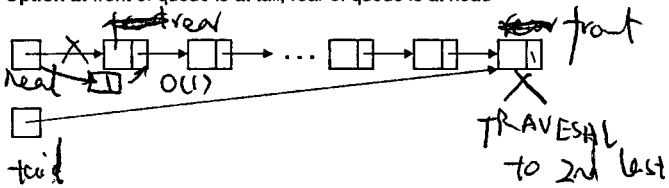
Implementing a using a Chain of Nodes

→ Is one option better than the other? ①

Option 1: front of queue is at head, rear of queue is at tail



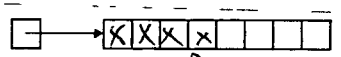
Option 2: front of queue is at tail, rear of queue is at head



Implementing a Queue ADT using an Array

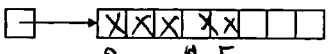
Assume a shadow array is used so that expand is O(1).

Option 1: front of queue is at Index 0, rear of queue is at Index N-1



ENQUEUE O(1)
DEQUEUE O(N) shift

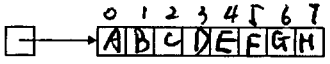
Option 2: front of queue is at Index N-1, rear of queue is at index 0



DEQUEUE O(1)
ENQUEUE O(N) shift

PROBLEM: FIXING either front or rear to index 0 requires one of ops to do shifting

Option 3: front of queue is at First used, rear of queue is at last used



q F R # F R

	4	0	3	
	5		4	ENQE
	6		5	ENQF
5	1			
4	2			DEQ
				DEQ.
5			6	ENQ G
6			7	ENQ H
				ENQ I

ENQ:
INC R
q[R]=item
numItems++

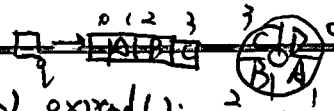
DEQ:
TMP = q[F]
INC F
numItem--
return Tmp

WHEN QUEUE REACH THE END of THE ARRAY, we'll wrap around. and reuse any free elems at the front of the array.

```
int incrementIndex (int i) {
  if (i == q.length-1)
    return 0;
  return i+1;
}
```

Implementing a Queue ADT using Circular Array

Concept REUSE THE FRO ELEMS AT THE THE ARRAY IS FULL



enqueue(item)

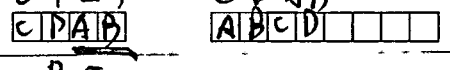
```
if (numItems == q.length) expand();
R = incrementIndex(R);
q[R] = item;
numItem++;
```

dequeue()

```
if (numItems == 0) throw new EmptyQueueException();
TMP = q[F]
F = incrementIndex(F);
numItem--;
return TMP;
```

#	F	R	
0	1	0	← constructor
1		1	ENQ A
2		2	ENQ B
3		3	ENQ C
4		0	ENQ D
			ENQ E
3	2		DEQ
2	3		DEQ
1	0		DEQ
0	1		DEQ

expand()



R F

unrolling the queue

System.arraycopy(q, F, newq, 0, q.length - F);

System.arraycopy(q, 0, newq, q.length-F, F);

Variations from reading that is simpler

F = 0

R = numItem - 1

CS 367 Announcements
Thursday, October 15, 2015

Midterm Exam 1
- Tuesday, October 20, 5:00 pm
- Lec 1: B10 Ingraham Hall
- Lec 2: 6210 Social Sciences Building
- UW ID required
- See posted exam information

Homework h5 due by 10 pm tomorrow, October 16th
- make sure your file is a pdf but not pdf scan of written work or pdf of a screen shot
- make sure you use the name h5.pdf
- submit to your `in` handin directory
- remember homeworks are to be done individually
- remember that late work is not accepted

Program p2 due 10 pm Friday, October 23rd

Last Time

Comparing ArrayList vs LinkedList
- shadow array

Stack ADT
- concept
- array implementations
- chain of nodes implementations

Queue ADT
- concept
- chain of nodes implementations
- array implementations

Today

Queue ADT
- array implementations (from last time)

Recursion
- call stack tracing
- recursion vs. iteration
- concept

Next Time

Exam mechanics

Sample questions solution

Displaying a Singly-Linked Chain of Nodes

Method Call:

```
print(head);
```

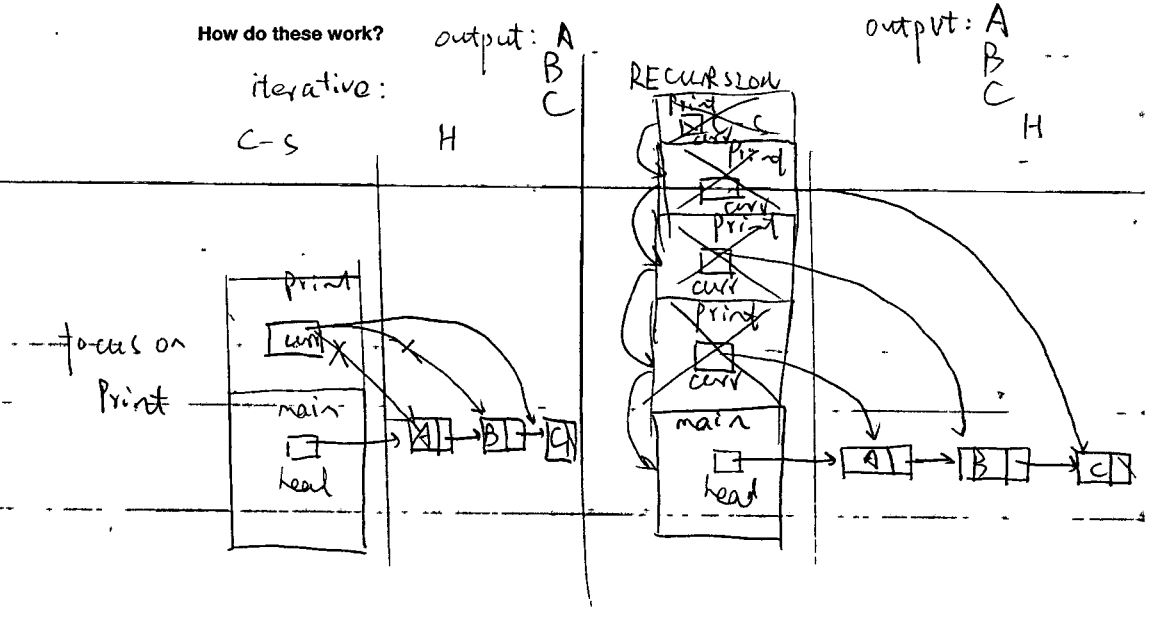
Iterative Implementation:

```
void print (Listnode<String> curr) {
  while (curr != null) {
    System.out.println(curr.getData());
    curr = curr.getNext();
  }
}
```

Recursive Implementation:

```
void print(Listnode<String> curr) {
  if (curr == null) return;        <- base case
  System.out.println(curr.getData());
  print(curr.getNext());           recursive case
}
```

How do these work?



Both REPEAT ONE-MORE STMTS
UNTIL some condition is met

Recursion vs. Iteration

Recursion is like iteration:

Iteration	Recursion
loop stopping condition	base case.
loop body	Recursive case

Recursion is NOT like iteration:

- Each loop iteration use the same Activation record
  parameter & local variables
  each recursive call use a new A.R.
  params & local variables.
- A loop with a bad stopping condition Result in an infinite loop
  A recursive method with a bad base case
  results in "infinite" recursion.
  Activa results in a program crashing due to call stack overflow

Rules for Recursion

1. every recursive method must have at least one base case (explicit or implicit)
2. Every recursive call must make progress towards a base case.

Recursion

What is it?

- is a divide & conquer strategy for solving problems that divides a problem into smaller & smaller problems of the same ~~kind~~ kind until the program is so small that it's solution is obvious or known
- A method is recursive if it calls itself either direct (ex in print) or in direct.
  METHOD A → METHOD B → METH C

Why use it?

Not used because it's faster
Not used because less memory is needed
Because it can often solve complex problems in a simple and concise manner.

How do you write the code to display a singly-linked chain of nodes in reverse order?

recursive call — modification flipping print: s.o.pln(—).

Factorials: n!

Consider the factorial of n (assume n >=0):

n! = n x (n-1) x (n-2) x (n-3) x . . . x 2 x 1

6! = 6 x 5 x 4 x 3 x 2 x 1 = 720

6×5!

5×4!

Method Call:

```
factorial(6);
```

Iterative Implementation:

```
int factorial(int n) {
   int result = 1;
   for (; n > 1; n--)
      result = result * n;
   return result;
}
```

Recursive Definition: $n! = n * (n-1)!$ if $n > 0$

$n = 0$

1

→ Complete the Recursive Implementation:

```
int factorial(int n) {
```

CS 367 Announcements
Thursday, October 22, 2015

Program p2 due 10 pm tomorrow, Friday, October 23rd
- submit java files to your `in` directory
- make sure to name your source files as specified in the submission section
- do not submit as a project/package/folder
- verify that you've submitted the correct files (ls, more, javac, java)
- partners? only ONE submits source but BOTH submit README.txt

Program p3 assigned This evening

Last Time
- exam mechanics
- sample questions

h6 Assigned Monday

Today

Recursion
- writing recursive code
- practice writing recursive code
- complexity of recursive methods
- practice analyzing complexity

Next Time

Read: finish *Recursion*, *Search*

Recursion
- more practice writing/analyzing
- execution tree tracing

Searching

Exams Returned

Recall Recursion

Recursion solves a problem by breaking it down into

smaller and smaller problems of the same kind.

until the problem is so small that it has a known/obvious **solution**

→ Why use recursion?

simpler/concise code.

→ How do you tell that a method is recursive?

calls itself directly

Rules:
1. Every recursive method must have at least one base case (implicit or explicit).
2. Every recursive method call must make progress towards a base case.

Constructing Recursive Code

→ Write a recursive method that computes n^m that is, it computes `double` n raised to an `int` power m?

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

recursive definition:

$$n^m = n * n^{m-1} \quad \text{IF } m > 0$$
$$= 1 \quad \text{IF } m = 0$$
$$= 1/n^{-m} \quad \text{IF } m < 0$$

recursive implementation:

```
double power (double n, int m) {
  if (m == 0) return 1;
  if (m > 0) return n * power(n, m-1);
    return 1/power(n, -m);
}
```

Key Questions:

1. How can you solve the problem in terms of smaller problems of the same kind.
2. What instances of the problem can serve as base cases?
3. How does the problem size decrease with each recursive call?
4. As the prob size decrease will A base case be reached?

Practice – ListADT

→ Write a recursive method that displays the values in a (non-null) list of strings.

1. To display A List, Print item in curr position, And then display the remaining List.
2. A List with no (remaining) items displays nothing
3. As curr advances dow the list, The remaining List decrease in size by 1
4. Eventualy there will be no remaining List

```
void display(ListADT<String> list) {
  if (list.isEmpty()) return;
  S.o.p (list.remove(0));
  display(list);
}

void display(ListADT<String> list, int curr)
  if (curr >= list.size()) return;
  S.o.pln(list.get(curr));
  display(list, curr+1);
}

void display(...)
  if (curr < list.size) {
    S.o.pln(...);
    display( );
  }
}
```

Destructive

Explicit base

Implicit base

Practice – Array

→ Write a recursive method that counts the number of even values in an (non-null) array filled with integers.

1. To count evens, check if current element's value is even, if so, add 1 to count evens of the remaining array otherwise Add 0.
2. An Array with No (REMAINING) ELEMs has 0 evens
3. As the curr element advances the remaining array decreases in size by one.
4. Eventually there will be No remaining array to be continued.

```
int evenCount(int[] array) {
    if (curr >= array.length) return 0;
    if (array[curr] % 2 == 0)
        count++;
    evenCount(array, curr+1);
}
return evenCount(array, 0);
```

```
int evenCount(int[] array, int curr) {
    if (curr >= array.length) return 0;
    if (array[curr] % 2 == 0)
        return 1 + evenCount(array, curr+1);
    return 0 + evenCount(array, curr+1);
}
```

recursive companion method.

* Sometimes A companion method is needed to allow addIt's params to be passed in the recursive method.

Analyzing Complexity of Recursive Methods

Options:
1. Informal Reasoning
2. recurrence equations → need to determining what aspect of the problem controls the problem size.

Steps

1. write equations
   base case(s) form: $T(\text{prob size of the base case}) = GRF$ for work of base case
   recursive case form: $T(N) = GRF$ for work of rec case excluding rec. calls $+ T(\text{PROB size of the rec call})$
2. expand the equations in a table
   - Look for a pattern between N & T(N)
   - Guess a solution based on the pattern
3. verify the guessed soln by substituting it back into the recurrence EQN
4. Do complexity simplification on the verified solution.

Practice – Complexity of Recursive evenCount

Problem size N is Number of elements in the arrays

1. Equations

$T(0) = 1$

$T(N) = 1 + T(N-1)$

2. Table

N	T(N)
0	1
1	$1 + T(0) = 1+1 = 2$
2	$1 + T(2-1) = 1 + T(1) = 3$
3	$1 + T(3) = 1 + T(2) = 1+3 = 4$
⋮	⋮
k	k+1

3. Verify

Solve +

$T(N) = 1 + T(N-1)$

$k+1 \quad k+1$

$N+1 = 1 + (N-1) + 1$

$= 2 + N - 1 = N+1$

4. Complexity

$O(N+1) = O(N)$

Towers of Hanoi

Algorithm

```
solveTowers(count, src, dest, spare) {
```

IF (count==1) Move Disk from SRC to dest

ELSE Solve Towers (count-1, SRC, SPARE, dest)

Solve Towers (1 , SRC, dest, spare)

solve towers (count-1, spare, dest, SRC)

Complexity

Problem size N is Number of disks

1. Equations

$T(1) = 1$

$T(N) = 1 + T(N-1) + T(1) + T(N-1)$

$= 2 * 1 + 2 * T(N-1)$

2. Table

N	T(N)
1	1
2	$1 + 2*(T(N-1)) = 3$
3	$1 + 2*(2) = 7$
4	$1 + 2*(3) = 15$
⋮	⋮
K	$2^K - 1$

3 Verify

$T(N) = 1 + 2\,T(N-1)$

$2^N - 1 \quad 2^{N-1} - 1$

$2^N - 1 = 1 + 2(2^{N-1} - 1) = 1 + 2^N - 2$

$= 2^N - 1$

4. Complexity $O(2^N - 1) \quad O(2^N)$

CS 367 Announcements
Tuesday, October 27, 2015

Homework h6 due 10 pm, Friday, October 30th

Program p3 due 10 pm, Sunday, November 8th

Last Time

Recursion
- practice writing recursive code
- complexity of recursive methods
- practice analyzing complexity

Today

Recursion
- more practice writing/analyzing recursion
- execution tree tracing

Searching

Exams Returned

Next Time

Read: *Trees*

Categorizing ADTs

Tree Terms

General Trees
- implementing
- determining tree height

Binary Trees
- implementing

Tree Traversals

Practice – Strings

→ **Write a recursive method that determines if a string is a palindrome.**

Examples:
- eye
- mom
- radar
- racecar
- Rise to vote, sir!
- Never odd or even!
- A nut for a jar of tuna.
- Campus Motto: Bottoms up, Mac.
- Ed, I saw Harpo Marx ram Oprah W aside!
- Doc note: I dissent. A fast never prevents a fatness. I diet on cod.

Assumptions: non-null input string, all spaces and punctuation removed, all lower-case

Useful string methods:
- char charAt(int index)
- int length()
- String substring(int begin, int one_past_last)

if charAt(0) == charAt

curr.>=

if (s.length()=0

substring(curr+1,

① A string is a palindrome if the first and last chars ARE THE SAME AND THE REMAINING STRING is A PALINDROME

② AN EMPTY STRING OR A STRING with one chars ARE Both PALINDROME

③
```
boolean isPal (String s) {
  if(s.length()==0 || s.length()== 1) return true;
  return s.charAt(0) == s.charAt(s.length()-1)
  isPal(s.substring(1, s.length()-1));
```

Analyzing Recursive `isPalindrome`

Problem size N is length of the string

1. Equations

$T(0) = 1$

$T(N) = N$

$T(0) = 1$

$T(1) = 1$

$T(N) = 1 + T(N-2)$

2. Table

N	T(N)
0	1
1	1
2	2
3	2
4	3
5	3
6	4
7	4
K	K/2 + 1 (larger)

3. Verify

$T(N) = 1 + T(N-2)$ $\quad$ $(N) = 1 + T(N-2)$

$K/2+1 = K/2+1$ $\quad$ ~~$T(N+1) = 1 + T(N-1)$~~ $\quad$ $T(N+2) = 1 + T(N)$

$N/2+1 \stackrel{?}{=} 1 + \frac{N-2}{2} + 1 = 1 + T(1<) = 1 +$

$= \frac{N-2}{2} + 2 = \frac{N+2}{2} = \frac{N}{2} + 1$ $=$

4. Complexity

$O(\frac{N}{2}+1) = O(N)$

Picking Lottery Numbers

What are your odds of winning the lottery? It depends on the number of possible combinations given how many numbers you have to pick and over what range:

Supercash - choose 6 out of 39 numbers (range 1 – 39)
Megabucks - choose 6 out of 49 numbers (range 1 – 49)

order doesn't matter
duplicates aren't allowed.

N Choose K: How many combinations of K things can you make from N things?

Recursive Definition:

① c(n,k) = $C(n-1, k-1) + C(n-1, k)$

count of combinations excludes fav #

count of combination includes FAV #

② $C(n,k) = 1$ IF $K = n$

1 IF $K = 0$.

0 IF $K > n$.

③ range n is always decreasing by 1

Pick K is either staying same or decreasing by 1

④ $C(n-1, k-1)$ reaches $k = \emptyset$

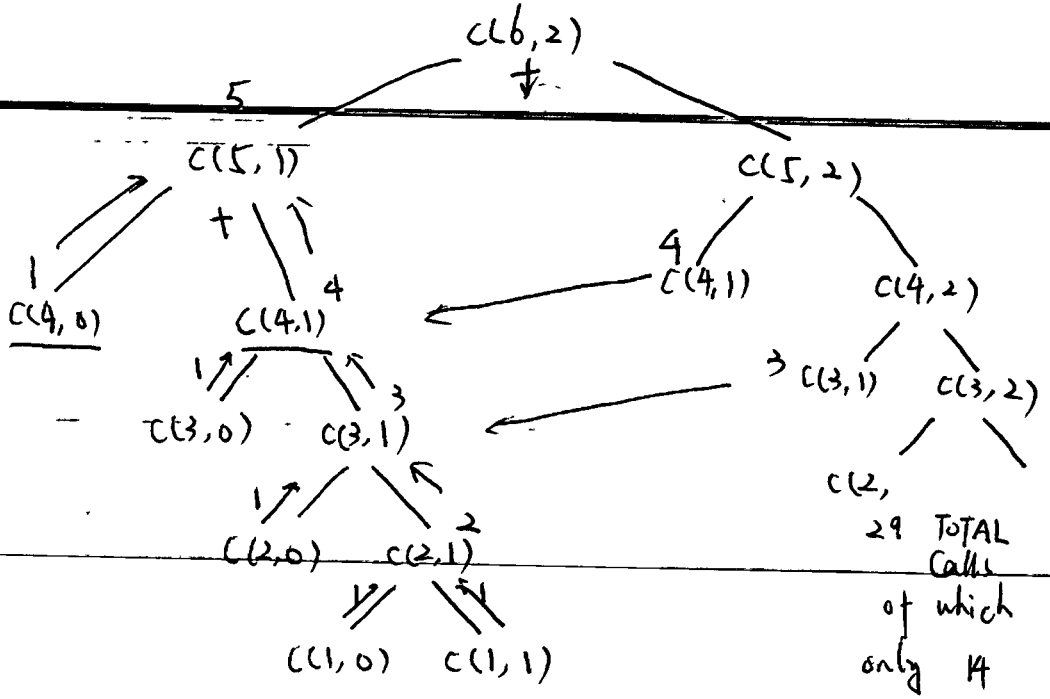
$C(n-1, k)$ reaches $n = k$

→ **Implement the c(n,k) method.**

```
int C(int n, int k) {
  if (k == n || k == 0) return 1;
    if (k > n) return 0;
    return C(n-1, k-1) + C(n-1, k)
}
```

sol'n $\frac{N!}{k!(N-k)!}$

Execution Tree Tracing of c(n,k)



29 TOTAL Calls of which only 14 are unique.

* used to diagram the execution of recursive code that does multiple recursive calls in recursive case.
* CAN REVEAL INEFFICIENCIES IN YOUR ALGO, that can then be addressed in a number of ways
  for example, add base case
  C(n,k) = n if k = 1

Searching

N is size of the List

Linear Search: O(N)

search one by one.
use on unsorted list

```
Linear search (L, pos, X)
IF (pos >= L.size()) return false;
IF ( X EQUALS (L.get(pos)) RETURN TRU
RETURN LINSEARCH (L, pos+1, X)
```

Binary Search: DIVIDE & CONQUE

REQUIRES A SORTED LIST

O(logN) by half

```
BINSEARCH( L, FIRST, LAST, X)
IF (FIRST > LAST) RETURN FALSE
CENTER = (first + last) / 2.
IF (x EQUALS (L.get (CENTER)))
    RETURN TRUE;
IF (x < L.get (CENTER))
    RETURN BINSEARCH (L, FIRST, CENTER-1,
ELSE                                    X)
    RETURN BINSEARCH (L, FIRST, CENTER+
                LAST, X)
```

CS 367 Announcements
Thursday, October 29, 2015

Homework h6 due by 10 pm tomorrow, October 30th
- make sure your file is a pdf but not pdf scan of written work or pdf of a screen shot
- make sure you use the name h6.pdf
- submit to your `in` handin directory
- remember homeworks are to be done individually
- remember that late work is not accepted

Program p3 due 10 pm, Sunday, November 8th

Last Time

Recursion
- more practice writing/analyzing recursion
- execution tree tracing

Searching

Exams Returned

Today

Categorizing ADTs Part 1

Tree Terms

General Trees
- implementing
- determining tree height

Binary Trees
- implementing

Tree Traversals

Next Time

Read: start *Binary Search Trees*

Categorizing ADTs Part 2

Comparable Interface

Binary Search Tree (BST)
- BSTnodes
- BST class
- implementing print

CS Options/Courses

Categorizing ADTs Part 1

BASED ON THEIR LAYOUT

- LINEAR

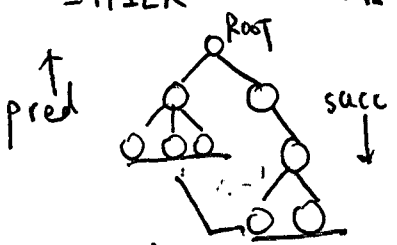
0-0-0-0

NEXT-PREV RELATIONSHIP

predecessor except for first

successor except for last

- HIERARCHICAL

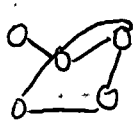


Parent/child relationship

1. pred except for ROOT

1. or more successors except for leaves

- GRAPHICAL

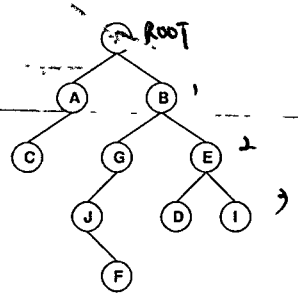


PAIRWISE Relationship

∅ or more pred's

∅ or more succ's

Tree Terminology



1. Which is the **root**? H
2. How many **leaves** are there? 4
3. How many nodes are in the right **branch/subtree** of B? 3
4. Which is the **parent** of G? B
5. How many **children** does E have (**degree** of E)? 2
6. Which is the **sibling** of E? G
7. How many **descendants** does B have? 6
8. What are the **ancestors** of C? A, H
9. What is the **length** of the **path** from B to D? 3 number of items in the path
10. What is the **height** of the tree? 5
11. What is the **depth/level** of J? 4

General Tree

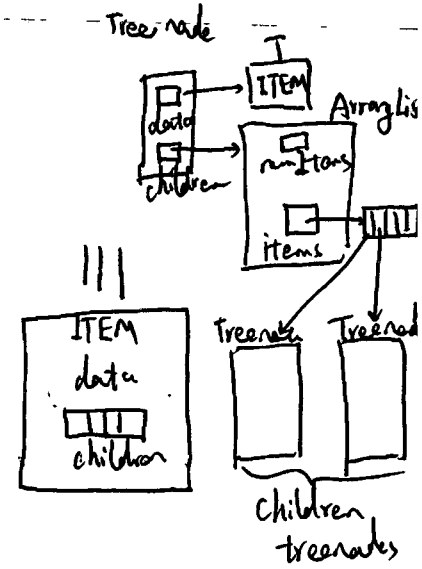
Each node can have an arbitrary number of children

The Tree Node Class:

```
class Treenode<T> {
   private T data;
   private ListADT<Treenode<T>> children;
   ...
```

getData setData getChildren setChildren

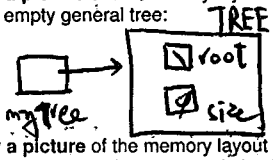
→ **Draw a picture** of the memory layout of a Treenode (assume an ArrayList is used for the ListADT):



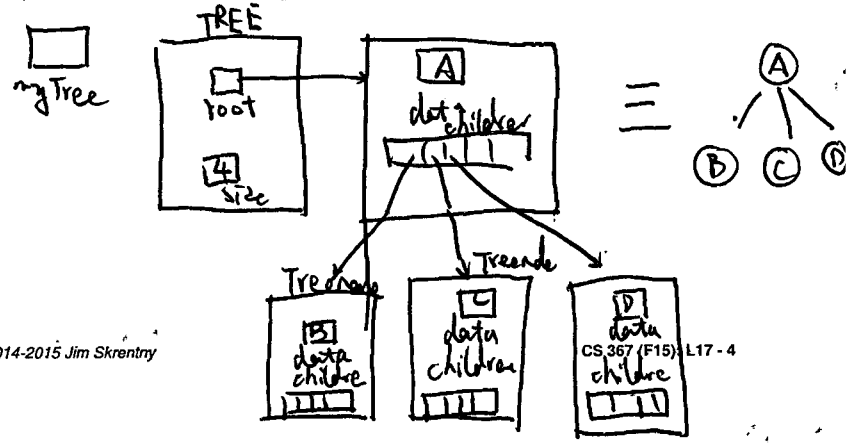
The Tree Class:

```
public class Tree<T> {
   private Treenode<T> root;
   private int size;

   public Tree() {
      root = null;
      size = 0;
   }
   ...
```

→ **Draw a picture** of the memory layout of an empty general tree:



→ **Draw a picture** of the memory layout of a general tree with a root node having 3 children:



Determining Height of a General Tree

Recall the height of a tree is the length of a path from the root to the deepest leaf.

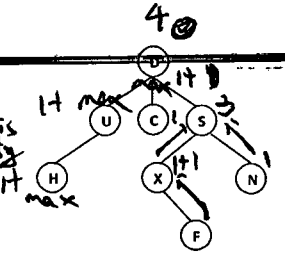
→ **Write a recursive definition** for the height of a general tree.

Height (t) = 0 IF t is null

= 1 if t is leaf

= 1 + max (height of children)

= 1 + max (



→ **Complete the recursive height method** based on the recursive definition. Assume the method is added to a Tree class having a root instance variable.

```
public int height() {   return height(root); }
private int height (Treenode<T> t) {
    if (t == null) return 0;
    if (t.getChildren().isEmpty()) return true 1;
    int maxHT = 0;
    Iterator <Treenode<T>> itr = t.getChildren
    while (itr.hasNext()) {                  Iterator();
        int childHT = height (itr.next());
        if (childHT > maxHT) maxHT = childHT
    } return 1 + maxHT;
}
```

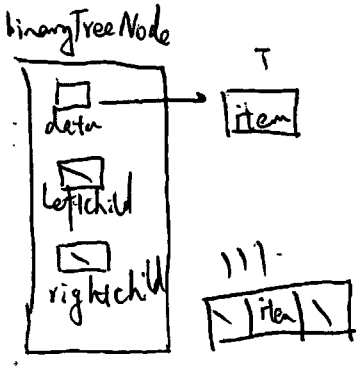

Binary Tree

- EACH NODE HAS AT MOST 2 CHILDREN

The Tree Node Class:

```
class BinaryTreenode<T> {
    private T data;
    private BinaryTreenode<T> leftChild;
    private BinaryTreenode<T> rightChild;

    public BinaryTreenode(T info) {
        data = info;
        leftChild  = null;
        rightChild = null;
    }
    ...
```

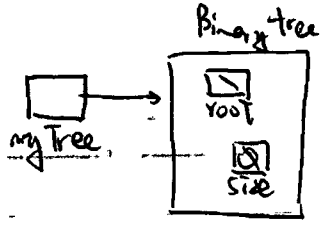
→ **Draw a picture** of the memory layout of a BinaryTreenode:



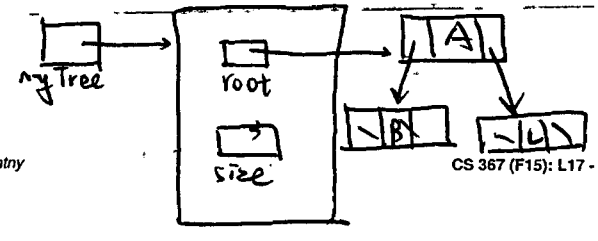
The Tree Class:

```
public class BinaryTree<T> {
    private BinaryTreenode<T> root;
    private int size;

    public BinaryTree() {
        root = null;
        size = 0;
    }
    ...
```

→ **Draw a picture** of the memory layout of an empty binary tree:



→ **Draw a picture** of the memory layout of a binary tree with a root node having 2 children:



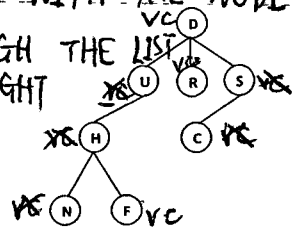
Tree Traversals

Goal: visit every node in the tree exactly once

VISIT MEANS TO DO SOMETHING WITH THE NODE DATA (eg output)

TRAVERING MEANS STEP THROUGH THE LIST OF CHILDREN FROM LEFT TO RIGHT

V = VISIT
C = TRAVERSE CHILDREN
L = TRAVERSE LEFT CHILD
R = TRAVERSE RIGHT CHILD



Level-order D U R S H C N F

TOP TO BOTTOM
LEFT TO RIGHT ON EACH LEVEL

General Tree | Binary Tree

Pre-order VC D U H N F R S C

VLR

Post-order C V N F H U R S C D

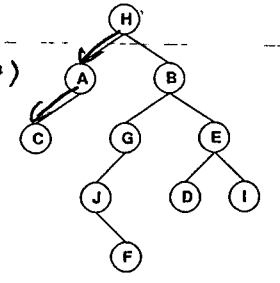
LVR

In-order NOT POSSIBLE

* USE TREE DIAGRAM FOR AN EXECUTION TREE TRACE

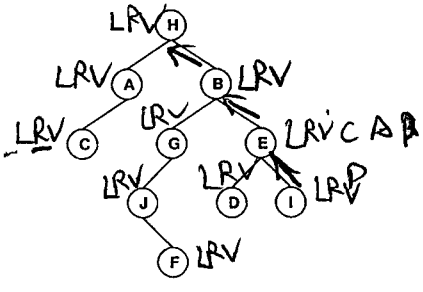
Practice - Tree Traversals

→ **List the nodes** using a pre-order traversal. VLR



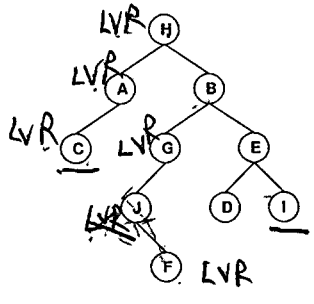
HAC

→ **List the nodes** using a post-order traversal. LRV



LRVCABFJG LEBH

→ **List the nodes** using an in-order traversal.



FARTHEST ON LEFT

CAHJFG BDEI

FARTHEST ON RIGHT

CS 367 Announcements
Tuesday, November 3, 2015

Homework h7 due 10 pm, Friday, November 6th

Program p3 due 10 pm, Sunday, November 8th

Last Time
- Categorizing ADTs Part 1
- Tree Terms
- General Trees
  - implementing
  - determining tree height
- Binary Trees
  - implementing

Today
- Finish Traversals (last lecture)
- Categorizing ADTs Part 2
- Comparable Interface
- Binary Search Tree (BST)
  - BSTnodes
  - BST class
  - implementing print
- CS Options/Courses

Next Time
- Read: finish *Binary Search Trees*
- Binary Search Tree (BST)
  - implementing lookup, insert, delete
  - complexities of BST methods
- Balanced Search Trees
- Classifying Binary Trees

Categorizing ADTs Part 2

BASED ON HOW OPERATIONS ARE DONE

POSITION ORIENTED? OPERATIONS occur ~~AT A SPECIFIED POSITION IN THE ADT~~

List ADT, Stack ADT, Queue ADT

VALUE ORIENTED: operations occur at a position in the ADT THAT's Based on A KEY VALUE IN THE ITEM.

SortedList ADT, MAP ADT

Comparable Interface

* USE TO DETERMINE THE RELATIVE ORDERING of ITEMS
- In java.lang package

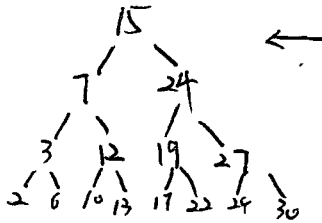
- Specifies one method.
  public int compareTo (T other)
- USE. a.compareTo(b)
  RETURNS ϕ IF a == b (ARE THE SAME)
  $< \phi$ IF a < b (comes before b)
  $> \phi$ IF a > b (comes after)
- Implementation should be compatible with equals()
  a.compareTo(b) == ϕ
  THEN a.equals(b) == TRUE
  Except when
  a.compareTo(null) THROWS NullPointerException
  ⇒ a.equals(null) == TRUE

Binary Search Tree (BST)

Goal

- VALUE ORIENTED
  DUPLICATE KEYS ARE NOT ALLOWED
- FAST LOOKUP, INSERT, REMOVE OPERATIONS
- combine speed of BINARY SEARCH on an ARRAY OF SORTED VALUES WITH THE SPEED OF LINKING & UNLINKING IN A CHAIN of Nodes

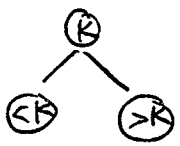
Example 2 3 6 7 10 12 13 15 17 19 22 24 26 27 30



← THIS IS IDEAL SHAPE FOR BST STORING THESE VALS

BUT BST DOESN'T GUARANTEE THIS SHAPE

Ordering Constraint

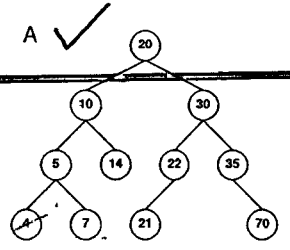


BST REQUIRES
① BINARY NODES
② FOR EACH NODE N HAVING A KEY VALUE K
- L < K for every key L in the left subtree of K.
- → R > K for every key R in the Right subtree of K

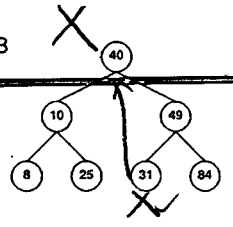
Practice - Identifying Binary Search Trees

→ Identify which trees below are valid BSTs.

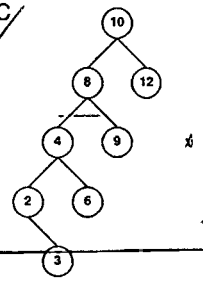
A ✓



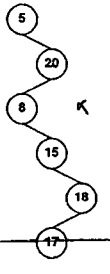
B ✗



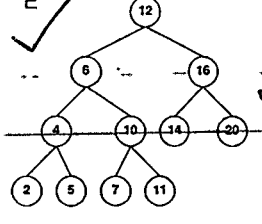
C ✓



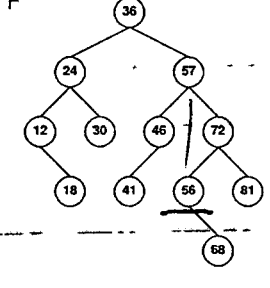
D ✓



E ✓



F ✗



BSTnodes

SEE READINGS FOR MODIFICATIONS TO BSTnode so that it also stores AN associated value

→ Draw a picture of the memory layout of a TreeNode:

PACKAGE

```
class BSTnode<K> {

  private K key;
  private BSTnode<K> left, right;

  public BSTnode(K key, BSTnode<K> left, BSTnode<K> right) {
    this.key = key;
    this.left = left;
    this.right = right;
  }

  public K getKey() { return key; }
  public BSTnode<K> getLeft() { return left; }
  public BSTnode<K> getRight() { return right; }

  public void setKey(K newK) { key = newK; }
  public void setLeft(BSTnode<K> newL) { left = newL; }
  public void setRight(BSTnode<K> newR) { right = newR; }
}
```

BST Class

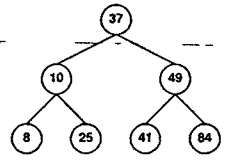
import java.io.*;

Restricts K type to only comparable classes.

```
public class BST<K extends Comparable<K>> {

  private BSTnode<K> root;

  public BST() { root = null; }

  public void insert(K key)
              throws DuplicateException {
                  root = insert(root, key);
  }

  public void delete(K key) {
              root = delete(root, key);
  }

  public boolean lookup(K key) {
              return lookup(root, key)
  }

  public void print(PrintStream p) {
              print(root, p);
  }

  //add helpers ...
    PRIVATE companion methods THAT RECURSIVELY
        DO THE OPERATION
}
```


Implementing print

→ **Write a recursive definition** to print a binary tree.



n... reference to tree node

if n is null, return] BASE case

print(n's left subtree) L
output n's key value) V
print(n's right subtree) R] REC case

8 10 25 37 41 49 84

→ **Complete the recursive print method based** on the recursive definition.

```
public void print(PrintStream p) {
  print(root, p);
}

private void print(BSTnode<K> n, PrintStream p) {
    if n == null return;
    if (n == null) return;
    print(n.getLeft(), p)
    p.println(n.getKey());
    print(n.getRight());
}
```

8 25 10 37 49 41 84

CS Options

CS Certificate

6 Courses
- Programming – CS 302
- Data Structures – CS 367
- 2 Courses >=400 level
- 2 Other CS Courses

CS Major

Basic CS
- Discrete Math – CS 240
- Programming + Data Structures – CS 302, CS 367
- Basic Systems – (CS 252), CS 352, CS 354

Math
- Calculus – MA 221, MA 222
- 2 Beyond Calc – MA 331/431 (probability), MA 340 (linear algebra)

Group A Theory
- Algorithms – CS 577

Group B Hardware/Software
- OS – CS 537

Group C Applications
- AI – CS 540

Group D Electives
- 2 CS Courses >=400 level

CS Double Major

- Must complete major requirements
- Easy for Computer Engineering Majors

CS Courses

Take Next

- CS 240 Introduction to Discrete Mathematics
- (CS 252) Introduction to Computer Engineering (prereq for CS 352)
- CS 352 Digital Systems Fundamentals
- CS 354 Machine Organization and Basic Systems (prereq for many group B)
- (CS 368) Learning a New Programming Language (C++ for CS 537)

>= 400 can take after CS 367

- CS 407 Foundations of Mobile Systems (spring, popular)
- CS 540 Introduction to Artificial Intelligence
- CS 570 Human Computer Interaction (spring)

>= 400 can take after CS 367 + Math

- CS 412 Introduction to Numerical Methods – MA 222 + MA 234 or CS 240
- CS 435 Introduction to Cryptography – MA 320 or MA 340
- CS 525 Linear Programming Methods – MA 320 or MA 340 or MA 443
- CS 533 Image Processing – MA 320 or MA 340 (fall)
- CS 559 Computer Graphics – MA 320 or MA 340
- CS 576 Introduction to Bioinformatics – MA 222 (fall)
- CS 577 Introduction to Algorithms – CS 240

CS 367 Announcements
Thursday, November 5, 2015

Homework h7 due by 10 pm tomorrow, November 6th
- make sure your file is a pdf but not pdf scan of written work or pdf of a screen shot
- make sure you use the name h6.pdf
- submit to your `in` handin directory
- remember homeworks are to be done individually
- remember that late work is not accepted

Homework h8 assigned 11/10

Program p3 due 10 pm tomorrow, Sunday, November 8th
- submit java files to your `in` directory
- make sure to name your source files as specified in the submission section
- do not submit as a project/package/folder
- verify that you've submitted the correct files (ls, more, javac, java)
- partners? only ONE submits source but BOTH submit README.txt

Program p4 assigned Monday 11/11

Last Time

Finish Traversals
Categorizing ADTs Part 2
Comparable Interface
Binary Search Tree (BST)
- BSTnodes
- BST class
- implementing print

CS Options/Courses

Today

Binary Search Tree (BST)
- implementing print (from last time)

Binary Search Tree (BST)
- implementing lookup, insert, delete
- complexities of BST methods

Balanced Search Trees

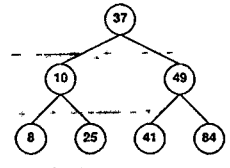
Next Time

~~Read: Red Black Trees~~
Classifying Binary Trees
Red Black Trees
- tree properties
- print, lookup
- insert
- cascaded fixing

Implementing `lookup`

Pseudo-Code Algorithm

```
private boolean lookup(BSTnode<K> n, K key) {
```



IF N is null return false // not found

IF N's key equals key return true // found

IF key < N's key return lookup(N's left subtree's key);

else return lookup(N's Right subtree's key);

This recursive approach to searching down the Tree is used in INSERT & DELETE

* Lookup could be implemented using A loop

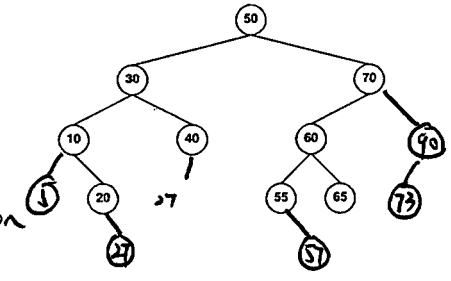
Implementing `insert`

High-Level Algorithm

```
private BSTnode<K> insert(BSTnode<K> n, K key)
                          throws DuplicateException {
```

SEARCH DOWN TREE down in lookup
BST
IF n's key EQUALS key throw duplicateException
WHEN WE GET TO THE END of THE TREE
where lookup would expect to find key
will insert A NEW LEAF NODE -
CONTAINING key

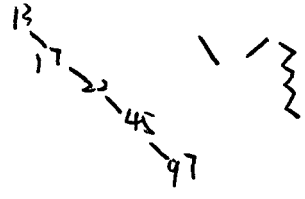
Practice - Inserting into a BST



→ Insert 5, 27, 90, 73, 57 into the tree above.

→ What can you conclude about the shape of a BST when values are inserted in sorted order?

13, 17, 22, 45, 97



→ Will you get that shape only if values are inserted in sorted order?

NO * THE SHAPE OF A BST DEPENDS ON THE SEQUENCE OF INSERTS AND DELETES

11, 55, 22, 99, 33

Implementing `delete`

High-Level Algorithm

```
private BSTnode<K> delete(BSTnode<K> n, K key) {
```

SEARCH DOWN THE TREE AS DONE IN LOOKUP

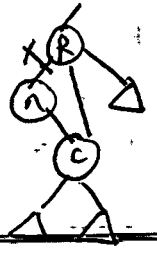
IF n is null RETURN null // NOT FOUND

IF n's key EQUALS KEY,

CASE 1: n HAS NO CHILDREN

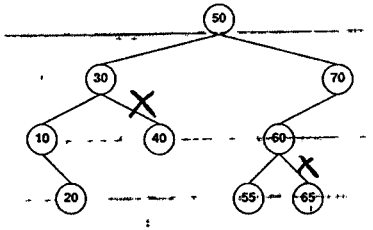
DELETE n by setting the appropriate child of n's PARENT P to null

case 2: n has 1 child

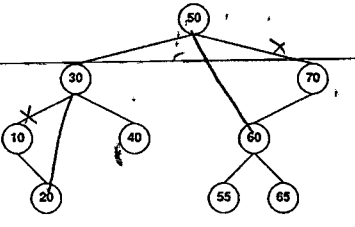


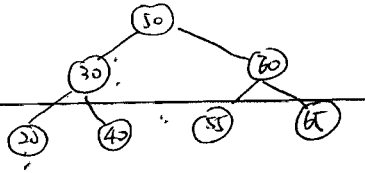
DELETE n by setting the appropriate child of n's PARENT P to n's child

Practice - Deleting from a BST

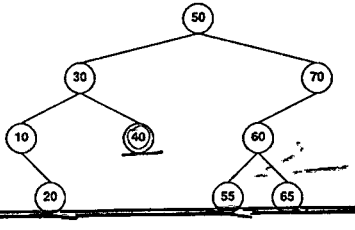


→ Delete 40 and 65 from the tree above.





→ Delete 10 and 70 from the tree above and redraw the tree.



→ How do you delete 50 or 30 from the tree above?

Implementing `delete` (cont.)

Case 3: n has 2 children

NOT so easy because PARENT / ROOT CAN'T HOLD ONTO BOTH CHILD SUBTREES

SOLN: FIND A REPLACEMENT VALUE V in either n's left or Right subtrees.

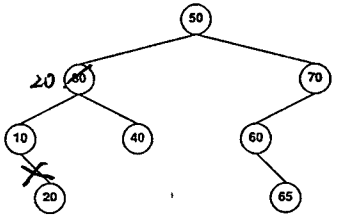
- Copy V into n's key
- Recursively DELETE V in n's subtree.

2. Replacements work inorder predecessor
   - Largest value in left subtree
   - step into left subtree then step as far right as possible

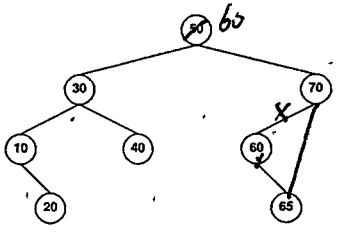
INORDER SUCCESSOR

- Smallest value in right subtree
- step into right subtree then steps as FAR LEFT AS possible

Practice - Deleting from a BST



→ Delete 30 from the tree above using the INORDER PRED



→ Delete 50 from the tree above using the INORDER SUCC

* DOESN'T MATTER WHICH is used
* DELETING THE NODE WHERE THE REPLACEM

Complexities of BST Methods

Problem size: N = Number of nodes = number of keys

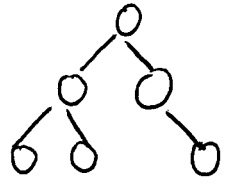
print: O(N)

lookup:

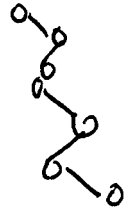
insert: O(H) WHERE H is THE BST's HEIGHT

delete:

O(log N)
Good balanced shape
"BUSHY"



O(N)
BAD LINEAR SHAPE
"STACKY"

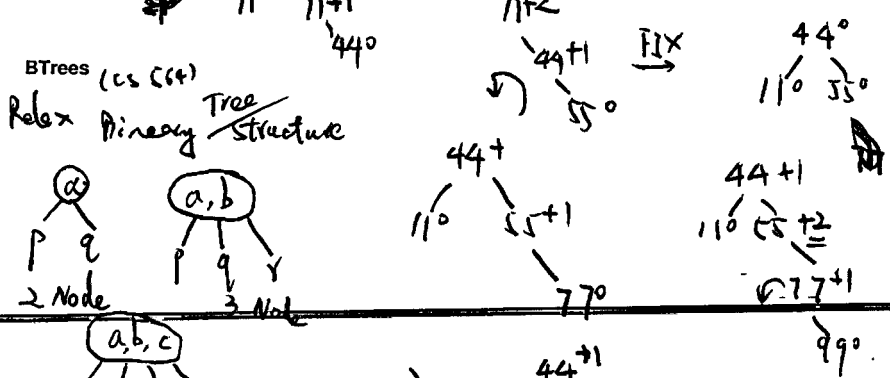


Balanced Search Trees

Goal: KEEP HEIGHT O(log N) where N is # of nodes
so lookup, insert, delete, are fast O(log N)

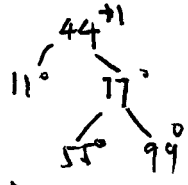
Idea:
MAKE INSERT AND DELETE RESTURCTURE THE TREE WHEN ITS shape goes out of balance
- detect imbalance
- Fix imbalance

AVL
HEIGHT BALANCED
KEEP A BALANCE VALUE IN EACH NODE -1, 0, +1
DETECT: WHEN BALANCE VALUE ±2
FIX: ONE TECH CALLED ROTATION



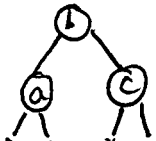
BTrees (CS 564)
Relax Binary Tree structure

FOR 2-3-4 TREE DETECT: USEING INSERT, Look up



FIX. SPLIT 4 NODES

SPLIT



NOW THESE NODES CAN GROW TO 3 NODES TO Accomodate the inserted item.

BTREE of ORDER 3 4 NODES
AKA 2-3 TREE
USES ONLY 2 NODES & 3 NODES

BTREE of order 4
AKA 2-3-4 TREE USES 2, 3 & 4 NODES

CS 367 Announcements
Tuesday, November 10, 2015

Homework h8 due 10 pm, Friday, November 13th

Program p4 due 10 pm, Sunday, November 29th

Last Time

Binary Search Tree (BST)
- implementing print
- implementing lookup, insert, delete
- complexities of BST methods

Balanced Search Trees

Today

Balanced Search Trees (from last time)
Classifying Binary Trees
Red-Black Trees
- tree properties
- print, lookup
- insert

Next Time

Read: *Priority Queues*
Red-Black Trees
- cascaded fixing
- complexity

Priority Queue ADT
- concept
- operations
- implementation options
- Heap Data Structure

Classifying Binary Trees

$N+1 = 2^H$

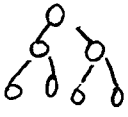
$H = \log_2(N+1)$

$H\ O(\log_2(N+1)) = H\ O(\log N)$

HEIGHT is H

Nodes $2^H - 1$

$N = 2^H - 1$



Full "NO MISSING NODES" ALL LEAVES ARE AT THE SAME DEPTH

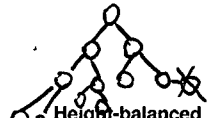
ALL NON-leaves (INTERABLE) NODES MUST HAVE 2 CHILDREN

Complete (PRIORITY QUEUES)

FULL TO DEPTH H-1

~~DEPTH H is filled with nods from left to right~~

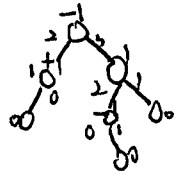
FULL TREE IS complete



Height-balanced (AVL)

FOR EACH NODE THE DIFFERENCE in height of its left & right subtree is at most 1

FULL & complete trees are also height balanced.



Balanced (RED - BLACK)

~~A TREE HAVING A HEIGHT of O(log N)~~

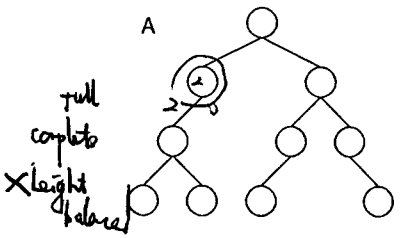
where N is the # of nodes

FULL, Complete, HEIGHT, BALANCED ARE ALL Balanced.

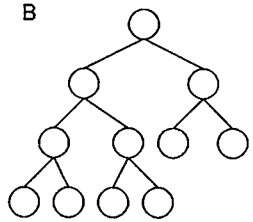
Practice - Classifying Binary Trees

→ Identify which trees below are full, complete and/or height balanced.

A

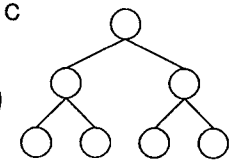


B

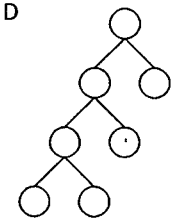


C

FULL
(C) (HB)



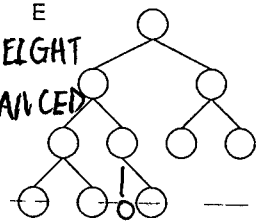
D



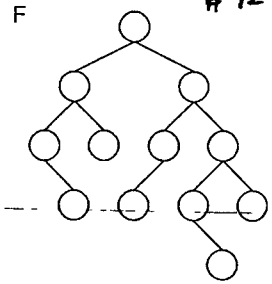
TENDING TO BE STACKY (IMBALANCED)

E

HEIGHT BALANCED



F



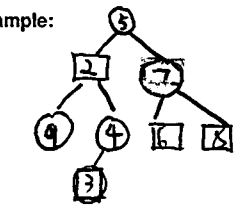
12

N=12
H=5
HB not B
if N is large enough
HB ⇒ B

Red-Black Trees (RBT)

RBT: BST THAT's modified to keep A balanced shape (height O(log N))

Example:



○ = BLACK
□ = RED

Red-Black Tree Properties

root property	ROOT NODE MUST BE BLACK
red property	RED NODES MUST HAVE BLACK CHILDREN
black property	EVERY PATH FROM THE ROOT TO A LEAF HAS THE SAME NUMBER OF BLACK NODES

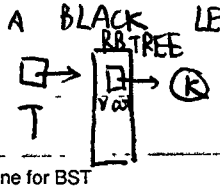
Red-Black Tree Operations

print	same as BST
lookup	same as BST
insert	① lookup ② insert ③ rebalance similar to BST
delete	① lookup ② insert ③ rebalance similar to BST

Inserting into a Red-Black Tree

Goal: insert key value K into red-black tree T and MAINTAIN RBT PROPERTIES

If T is Empty ADD A BLACK LEAVE NODE CONTAINING k



EXCEPT FOR ROOT ALL NEW NODES ADDED ARE RED ~~RED~~ LEAF NODES

If T is Non-Empty

- step down tree as done for BST
- add a leaf node containing K as done for BST, and COLOR IT RED!
- RESTORE RBT PROPERTIES IF NEEDED

→ **Which of the properties might be violated as a result of inserting a red leaf node?**

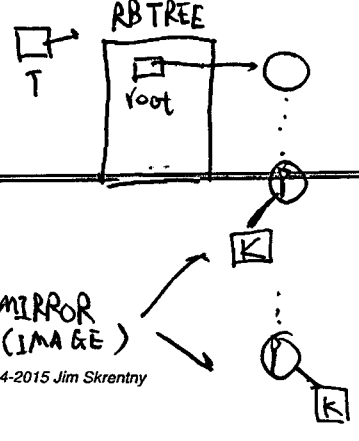
root property A NON-EMPTY TREE ALREADY has a black root

black property ADDING A RED NODE DOESN'T AFFECT THE NUMBER of BLACK NODES

red property (KEY) ADDING A RED NODE WILL VIOLATE THE RED property if the parent is RED

* USE A RPV

Non-Empty Case 1: K's parent P is black

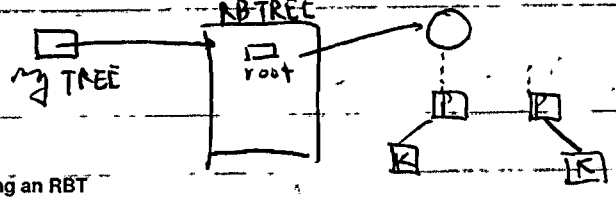


NO RPV SO NO FIX IS NEEDED

INSERT is DONE!

Non-Empty Case 2

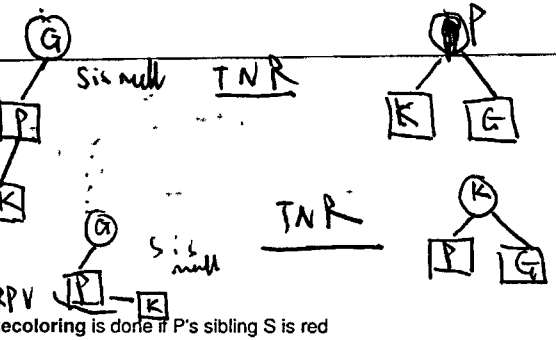
Non-Empty Case 2: K's parent P is red



DETECT RPV WHICH REQUIRE A FIX

Fixing an RBT

Tri-Node Restructuring is done if P's sibling S is null



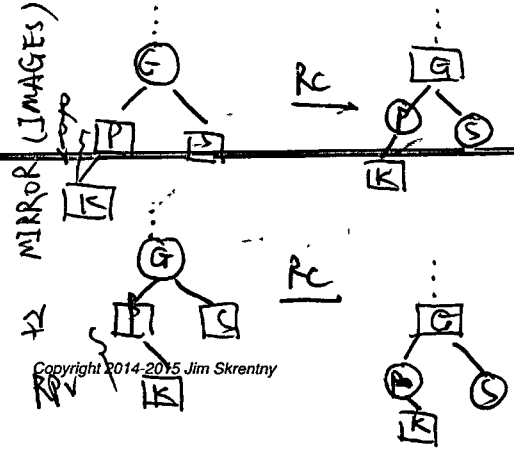
+ 2 MIRROR IMAGES

① MIDDLE VALUE BECOMES BLACK PARENT

② SMALLEST VALUE BECOMES RED LEFT CHILD

③ LARGEST VALUE BECOMES RED RIGHT CHILD

Recoloring is done if P's sibling S is red

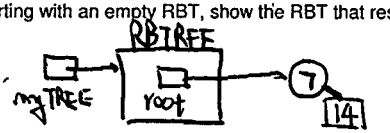


① CHANGE P & S TO BLACK

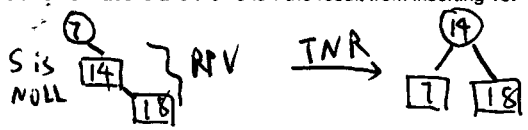
② IF G is ROOT, DONE INSERTING OTHERWISE CHANGE G TO RED

Practice

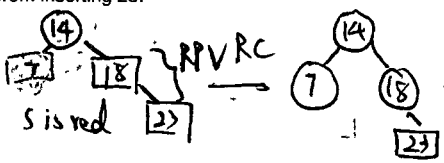
→ 1. Starting with an empty RBT, show the RBT that results from inserting 7 and 14.



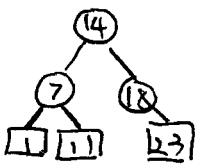
→ 2. Redraw the tree from above and then show the result from inserting 18.



→ 3. Redraw the tree from above and then show the result from inserting 23.



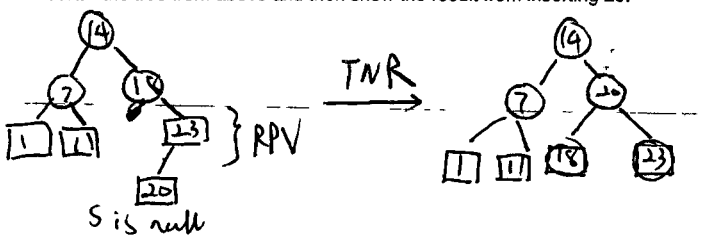
→ 4. Redraw the tree from above and then show the result from inserting 1 and 11.



Some inserts require NO FIXING

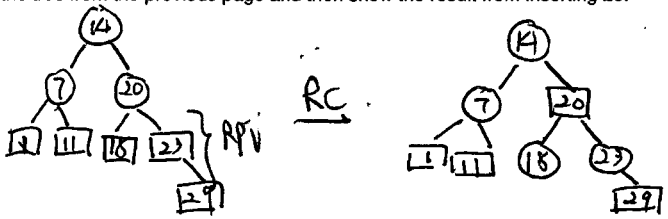
G is Root so No color change

→ 5. Redraw the tree from above and then show the result from inserting 20.

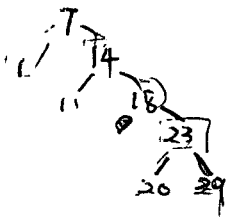


More Practice!

→ 6. Redraw the tree from the previous page and then show the result from inserting 29.



→ 7. Insert the same list of values into an empty BST: 7, 14, 18, 23, 1, 11, 20, 29

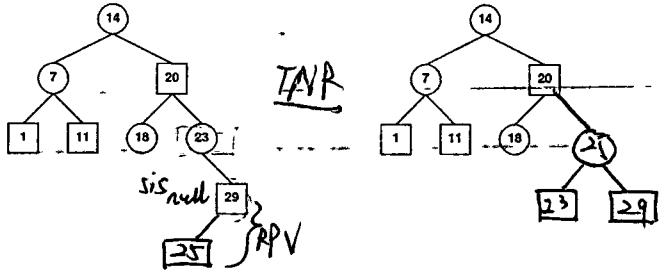


→ What does this demonstrate about the differences between a BST and RBT?

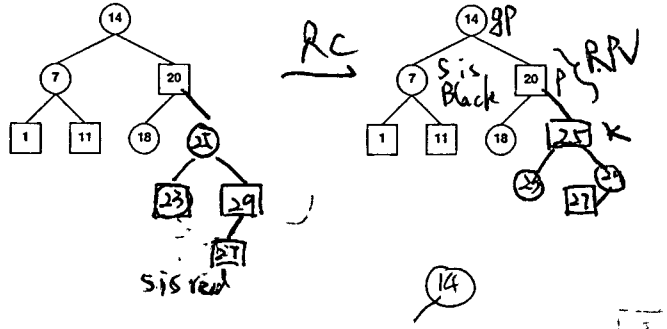
RBT maintains balance

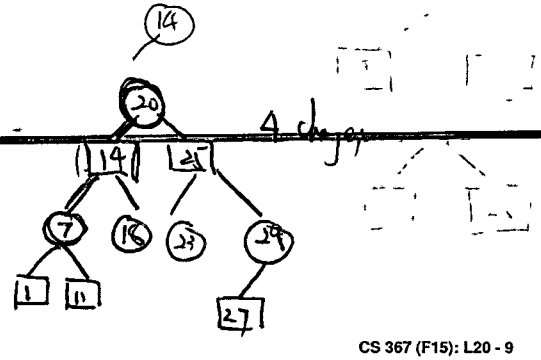
More Practice?

→ 8. Show the result from inserting 25 in the RBT below.



→ 9. Redraw the tree from above and then show the result from inserting 27.





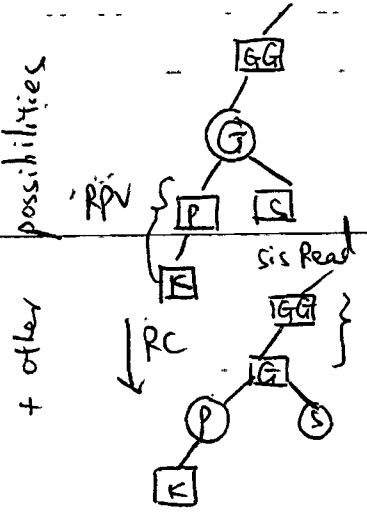
Cascading FIXES

FIXING RBT UPDATED

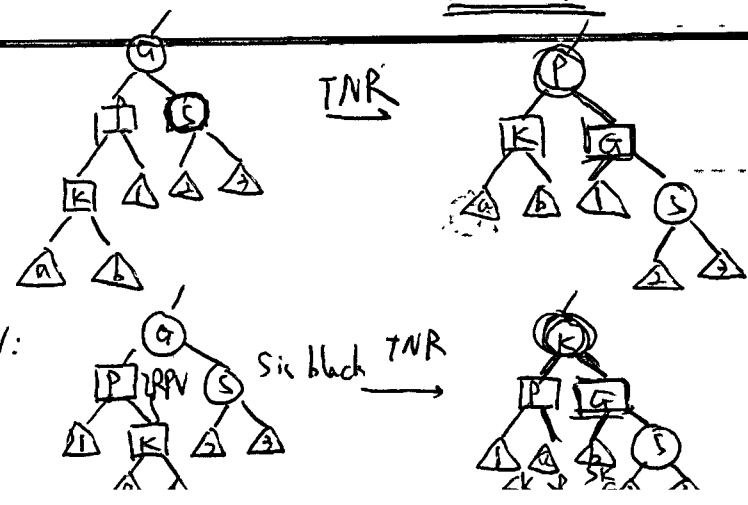
REcolouring is done if P's sibling S is Red

1. CHANGE P & S to Black
2. IF G is ROOT Done!
otherwise change G to red.
if GG is black Done!
ELSE

RPV G & GG which well fix recursively starting at G



TRI-NODE REstructuring is done if P's sibling S is null or Black



RBT complexity

PRINT same as BST O(N)

Lookup same code as BST

BUT W.C. O(logN) since RBT is Balanced

insert ① insert red ~~leave~~ Leave node in W.C is O(logN)

Guarantee Height of RBT

② Restore RBT Properties in W.C recoloring casades Back to root is O(logN)

overall

①	+	②
O(logN)		O(logN)

O(2logN)

O(logN)

CS 367 Announcements
Thursday, November 12, 2015

Homework h8 due 10 pm, Friday, November 13th

Program p4 due 10 pm, Sunday, November 29th

Last Time

Balanced Search Trees
Classifying Binary Trees
Red-Black Trees
- tree properties
- print, lookup
- insert

Today

Red-Black Trees (from last time)
- cascaded fixing
- complexity

Priority Queue ADT
- concept
- operations
- implementation options

Heap Data Structure

Next Time

Read: start *Hashing*
Heap Data Structure
- insert
- removeMax

Hashing
- terminology
- designing a good hash function

Priority Queue ADT

Priorities used to store items ranked by their importance
- each item stores A number FOR its priority
- Duplicate PRIORITIES ARE Allowed
- HIGHEST PRIORITY can be either the smallest or the largest number.

Concept

P.Q is AN ADT where items are removed in order of their priorities.

goal: FAST ACCESS O(1) TO HIGHEST PRIORITY

Operations

void insert (Comparable item)
Comparable getMax()
Comparable removeMax()

Options for Implementing a Priority Queue ADT

data structure	`insert`	`removeMax`
unordered array	O(1) AT REAR w/ shadow	O(N) W.C. OR LINEAR SEARCH ~~O(N) SHIFT~~
ordered array	O(N) W.C. " O(logN) SEARCH +O(N) SHIFT	O(1) MAX PRIORITY AT REAR of ARRAY
unordered chain of nodes	O(1) INSERT @ HEAD	O(N) W.C LINEAR SEARCH
ordered chain of nodes	O(N) W.C " O(N) linear search +O(1) Linking	O(1) MAX PRIORITY @ HEAD
HEAP	O(log N)	O(log N)

Implementing a Priority Queue ADT using a Heap

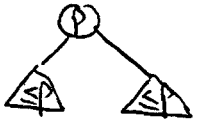
Heap

min heap smallest value is highest priority
max heap largest value is .....

Shape Constraint

① FULL FROM complete binary tree ROOT TO ~~END~~ LAST LEVEL
② LAST LEVEL FILLED FROM LEFT TO RIGHT

Ordering Constraint (max)



TOP ↓ Bottom

For every node i
N's priority P is
≥ the priorities
of N's
Descendants

Implementing Heaps

Max Heap Example:

	56	42	37	38	14	12	26	29	16	8

→ Draw the corresponding binary tree:

CS 367 Announcements
Tuesday, November 17, 2015

Midterm Exam 2
- Tuesday, November 24th, 5:00 pm
- Exam information posted
- Sample questions on Learn@UW
- UW IDs are required

h4, h5, h6 graded

Homework h9 due 10 pm, Friday, November 20th

Program p4 due 10 pm, Sunday, November 29th

Last Time

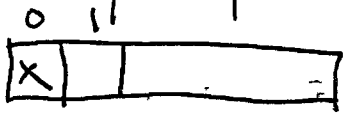
Red-Black Trees
- cascaded fixing
- complexity

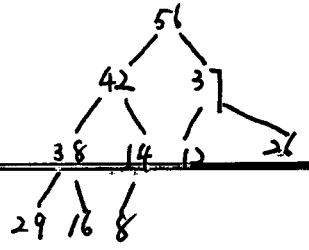
Priority Queue ADT
- concept
- operations
- implementation options

Heap Data Structure

Implementing Heaps
Root is at index 1 (NOT USING ELEM @ Index 0)
For Each Node N at Index i
N's LEFT CHILD is at index 2*i
N's RIGHT 2*i+1
N's parent is at i/2 INT DIV

Max Heap Example:





Today

Heap Data Structure
- insert
- removeMax

Hashing
- terminology
- designing a good hash function

Next Time

Read: finish *Hashing*

Hashing
- choosing table size
- expanding a hash table
- handling collisions

Inserting into a Max Heap

Algorithm ① PUT New Item in next tree element O(1)
② RESTORE HEAP ordering constraint
REHEAPIFY by SWAPPING new item with its smaller parent.

Given the following max heap:

	64	52	35	46	~~57~~ 36	15	34	12	23	14	~~X~~ 17	

O(log(N))

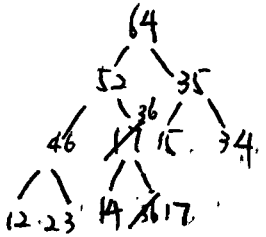
→ Show the heap after inserting 36:

	64	52	35	46	36	~~15~~ 57	34	12	23	14	17	~~57~~ 15

35

→ Show the heap after inserting 57:

	64	52	57	46	36	35	34	12	23	14	17	15



Inserting into a Max Heap (cont.)

PriorityQueue Class Instance Variables:

```
private Comparable[] items;
private int nextLoc;
```

Pseudo-code

```
public void insert(Comparable data) {
    IF (data == null) THROW Exception.
    //1.
    IF (ARRAY IS FULL) EXPAND — O(1)
      items [nextLoc] = data;
       nextLoc ++;

    //2.
      int child = nextLoc -1;
       boolean done = false;
      while (! done) {
         int parent = child /2;
         if (parent == 0) done = true;
         else if (items[child].compareTo(items[parent])
                                    <= 0)
              done = true;
         else { swap child AND parent items
              child = parent;
         }
      }
}
```

O(1)

O(log(N))

Complexity O(log N)

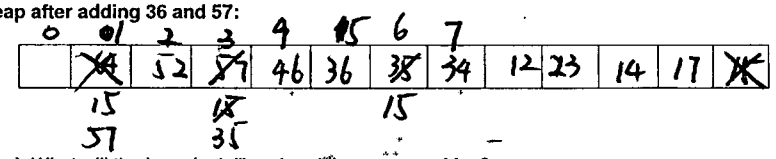
Removing from a Max Heap

Algorithm

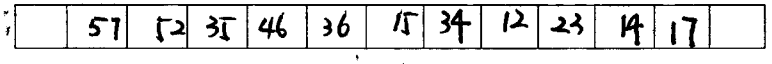
O(1) [① REMOVE ROOT ITEM BY replacing it with the last item in the array.

O(log N) [② RESTORE THE HEAP ORDERING Constraint reheapify: BY SWAPPING Down with largest child

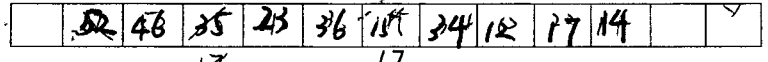
Heap after adding 36 and 57:



→ What will the heap look like after doing a removeMax?



→ What will the heap look like after doing another removeMax?



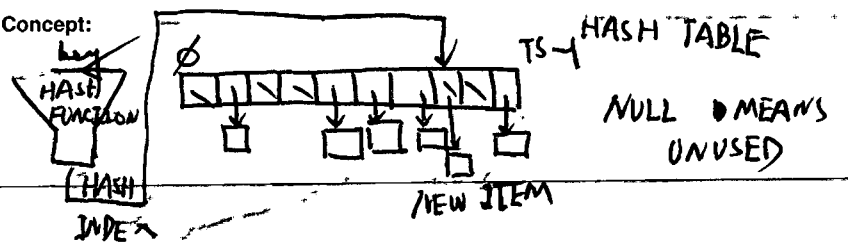
Complexity

Hashing

Goal: DO FASTER THAN O(log N) TIME complexity for lookup, insert, remove on a value oriented collection.

TRY TO ACHIEVE O(1) FOR THESE OPS

Concept:



hash table ARRAY THAT STORES THE COLLECTION of items

table size (TS) LENGTH of the array

load factor (LF) = NUMBER of ITEMS / TS

key INFO IN ITEM THAT DETERMINES WHERE IT GOES IN THE TABLE used to compute the hash index

hash function CONVERTS THE KEY INTO A HASH INDEX

Ideal Hashing

Assume

- store 150 students records
- table is an array of student records
- null is sentinel value meaning element is unused
- key is student id number in format: zipcode + 3 digit number

53706000, 53706001, 53706002, ... 53706148, 53706149

→ What would be a good hash function to use on the id number?

```
int hash(K key) {
    return key - 53700000;
}
```

Perfect Hash Function:

```
void insert(K key, D data) {table[hash(key)] = data;}
D lookup(K key) {return table[hash(key)];}
void delete(K key) {table[hash(key)] = null;}
```

The UW uses 10 digit ID numbers: 9012345789 9012345432 9023456789

→ Is a perfect hash function possible for these id numbers?

Yes, use the trivial hash function hash index is the ID number. But this requires a huge table and wastes lots of space.

Collision: WHEN THE HASH FUNCTION RETURN THE SAME HASH index for diff keys

Key Issues:

- DESIGNING the HASH FUNCTION
- CHOOSING THE TABLE SIZE
- HANDLING COLLISIONS

Designing a Hash Function

Good Hash Functions:

① MUST BE DETERMINISTIC
- KEY MUST ALWAYS GENERATE SAME HASH INDEX.

② Achieve uniformity
- keys ARE distributed evenly Across TABLE

③ FAST/EASY To Compute. use only PARTS of key THAT distinguish items from each other.

④ MINIMIZE collisions

Java Hash Function Steps:

1. GENERATE A HASH CODE item.hashcode(); CONVERTS ITEM's key INTO AN INTEGER;
2. Compress hash code into a valid hash index for the table. Hashcode % TS
   0 to (TS - 1)
   * Beware of negatives Abs (hash code)
   % is slow
   Do BIT SHIFT TO DIVIDE By Power of TWO and to use TS that is a power of two

CS 367 Announcements
Thursday, November 19, 2015

Midterm Exam 2
- Tuesday, November 24th, 5:00 pm
- Lec 1: B10 Ingraham Hall
- Lec 2: 6210 Social Sciences Building
- Exam information posted
- Sample questions on Learn@UW
- UW IDs are required

Homework h9 due 10 pm, tomorrow, November 20th

Program p4 due 10 pm, Sunday, November 29th

Last Time

Heap Data Structure
- insert
- removeMax

Hashing
- terminology
- designing a good hash function

Today

Hashing
- designing a good hash function (from last time)
- choosing table size
- expanding a hash table
- handling collisions

} Not on Exam 2

Next Time
- exam 2 Q&A

Techniques for Generating Hash Codes

Extraction Breaking key into parts and then using just these parts that distinguish ITEM

~~80~~ 123,456,789

Weighting EMPHASIZING some parts over others

123 * 3 + 456 * 30 + 789 * 100

Folding combing parts of - using addition & Bitwise ops (Exclusive OR)

Handling Non-Integer Keys

Strings

$C_0 * 31^{N-1} + C_1 * 31^{N-2} + \cdots + C_{N-2} * 31^1 + C_{N-1} * 31^0$

C_i is ASCII value FOR char At Pos i in A string of length N

For fixed sized string PRECompute values of 31^i

For variable sized string

$(((C_0 * 31) + C_1) * 31 + C_2) * 31 \ldots$

complexity: O(1) WRT size of the collection.

O(N) WRT THE length of the string.

Doubles

64 bit IEEE FLOATING POINT

↓

32 bit unsigned Integer

IDEA: EXTRACT UPPER AND LOWER 32 Bits fold together using Java's BITWISE "Exclusive-OR"

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Ex: 8 BIT Double → 4 Bit Integer

0110 1011

0000 0110 PATTERN For key

SHIFT TO RIGHT By 4 BITS

1101 xOR result

Hash code

Java long bits = Double.doubleToLongBits(key);

Choosing the Table Size

Table Size and Collisions

Assume 100 items with random keys in the range 0 – 9999 are being stored in a hashtable.
Also assume the hash function is simply %tablesize.

→ How likely would a collision occur if the table had 10000 elements? 1000? 100?

* As LOAD FACTOR INCREASES SO DO Collisions.

CHOOSE A TABLE SIZE THAT LEAVES SOME FREE SPACE (LOAD FACTOR .7 TO .8)

Table Size and Distribution

Assume 50 items are stored in a hashtable.
Also assume the hashCode function returns multiples of some value x.
For example, if x = 20 then hashCode returns 20, 40, 60, 80, 100, ...

→ How likely would a collision occur if the table had 60 elements? 50? 37?

THIS CAN RESULT IN POOR DISTRIBUTION

ONLY m BUCKETS WILL BE USED

m = TS / GREATEST COMMON FACTOR (X, TS)

3 = 60/20 5 = 50/10

37 = 37/1

TO AVOID THIS PROBLEM, USE A PRIME TABLE SIZE

BACKUP: PICK TABLE SIZE THAT IS NOT EVENLY DIVIBILE BY 2-19

By 4 BITS 0110 1101

int hashCode = (int)(bits ^ (bits >> 32))

Resizing the Hash Table

- RESIZE WHEN TABLE GETS "TOO FULL" IF $\geq .7$

- NAIVE APPROACH ① make new table twice as BIG
  Doesn't work since TS of rehashing affects the hash index
  ② Copy items from old to new table.

Steps

1. double table size to NEAREST PRIME. BACK UP:
   (2 * OLD TS) + 1
   ALLOCATE NEW TABLE
2. REHASH ITEMS FROM THE OLD TABLE INTO THE NEW TABLE

Complexity: costly $O(N)$ where N is size of collection.

hash(key) = key.hashcode() % TS

TS	hash(63)	hashINDEX
19	63%19	6
37	63%37	26

IF possible, carefully select the initial TS to avoid rehashing

Collision Handling using Open Addressing

EACH ELEMENT IN THE TABLE STORE ONLY ONE ELEMENT

IF collision search FOR AN unused element ELSE WHERE IN THE TABLE

IDEA: use a "PROBE SEQUENCE" (PS) with wrap around search making sure that look up insert & delete use same PS.

Linear Probing

STEP size is 1

PS: $H_K, H_{K+1}, H_{K+2} \ldots$ where $H_K = hash(key)$

RESULTS IN CLUSTED of used elements which DUE to long PS

0	1	2	3	4	5	6	7	8	9	10
440	166	266	124	263			337	359		351

DROP the effeciency

key
166
359
263

hash(key) = key % TS

H_K	H_{K+1}	H_{K+2}
7	8	
9/10	11 → 0	1

Quadratic Probing

BETTER PS: $H_K, H_{K+1}, H_{K+2^2}, H_{K+3^2} \ldots$

BETTER since it reduces clusters

Double Hashing

PS: $H_K, H_K + 1*step, H_K + 2*step, H_K + 3*step$.

where step = hash2(key)

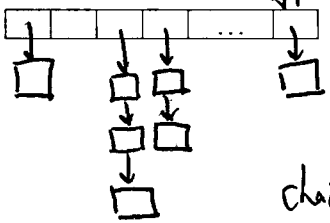
PREFERED FOR OPEN Addressing

Collision Handling using Buckets

EACH ELEMENT CAN STORE MORE THAN - ONE ITEM
- It's A Bucket

if collision: just throw colliosed item in the bucket.

"Chained" Buckets



Typically leave Buckets unsorted.

+ EASY TO IMPLEMENT
+ Buckets are dynamically sized (Grow/ shrink As needed)

chain of listnodes

Insert O(1) AT ITEM AT HEAD

Array Buckets

Typically The Bucket size of 3 works well

lookup/Delete are O(1) ave case,

Tree Buckets

BALANCED SEARCH TREES

Overly complicated buckets. More given A FEW items

For reasonable Ts LAND Good Hash FUNC. ARE O(N) FOR W.C.

Java API Support for Hashing

`hashCode` method

- method of `Object` class
- returns an `int`
- default hash code is BAD - computed from object's memory address

Guidelines for overriding `hashCode`:

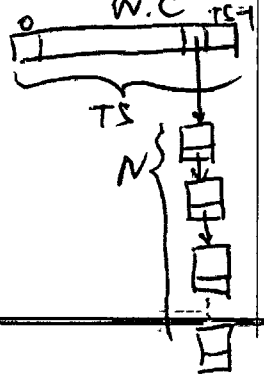
* ~~REMEMBER~~ REMEMBER THAT IT MUST BE DETERMINISTIC
* IF Your item class overrides equals() then it should also override hashcode().

`Hashtable<K,V>` and `HashMap<K,V>` class

- in `java.util` package
- implement `Map<K,V>` interface
- K TYPE PARAM FOR THE KEY (such as name)
  V FOR THE ASSOCIATED VALUE (such as city info)

  operations:
  - V get(K key) Lookup
  - V put(K key, V value) INSERT
  - boolean remove(K key) V remove(K key, V value)
- constructors allow you to set
  initial capacity (default = 16 for HashMap, 11 for HashTable)
  load factor (default = 0.75)
- handles collisions with chained buckets
- HashMap only: ALLOWS null FOR BOTH KEYS & VALUES
- Hashtable only: is synchronized.

TreeMap vs HashMap

	TreeMap	HashMap
underlying D.S	RBT	HASHTABLE WITH CHAINED BUCKETS
Complexity of Basic ops	O(log N)	O(1) AVG-CASE O(N) Worst Case
ITERATING OVER THE KEYS	ASCENDING ORDER	NO PARTICULAR ORDER
complexity of iterating over values	O(N) TREE TRAVERIAL	O(TS+N) W.C 
IF HASHING is so FAST why DON't always USE IT FOR STORING A KEY VALUE ORIENTED COLLECTION.		

CS 367 Announcements
Tuesday, December 1, 2015

Program p5 due 10 pm, Tuesday, December 15th

Last Time
- exam mechanics
- sample questions

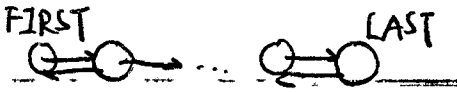
Today
Finish Hashing (prior lecture)
ADTs/Data Structures Revisited
Graphs
- terminology
- implementation issues
- edge representations
- traversals

Return Exam 2

Next Time
Read: continue *Graphs*
- traversals
- applications of BFS/DFS
- more terminology
- topological ordering

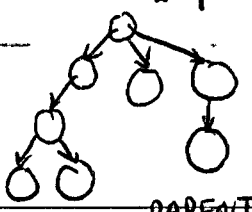
ADTs/Data Structures

Linear



- predecessors: at most 1 PREV
- successors: at most 1 NEXT

Hierarchical

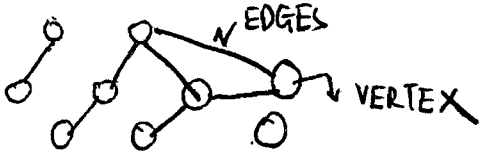


TOP ROOT

"BOTTOM" LEAVES

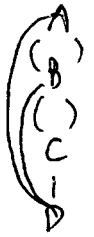
- predecessors: at most 1 PARENT
- successors: 0 or more - general tree, at most two - binary tree CHILDREN

Graphical



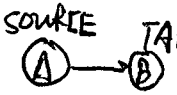
- predecessors: ANY NUMBER
- successors:

* REPRESENT PAIRWISE RELATIONSHIPS/ BETWEEN ITEMS IN THE COLLECTION PROCESSES
  EDGE INDICATES SOME RELATIONSHIP EXISTS.
* NO CLEAR FIRST/LAST TOP/BOTTOM
  SO WE NEED TO SPECIFY WHERE TO START/ STOP
  AND BE ABLE TO ACCESS ANY PARTICULAR VERTEX EFFICIENTLY



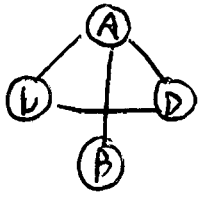
MUST HAVE 0 EXACTLY
ϕ OR 2 VERTEXES OF
ODD DEGREE

Graph Terminology



A PRED B
B SUCC A

UNDIRECT



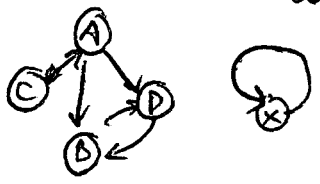
"GRAGH"

SELF EDGE

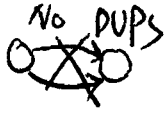
NO DUPS

MULTIGRAPH

DIRECTED



NO DUPS



DEGREE: NUMBER of EDGES FOR A SPECIFIED VERTEX

IN DEGREE: NUMBER of incoming edges

OUT DEGREE: NUMER of outgoing edges

degree: indegree + outdegree

PATH: SEQUENCE OF CONNECTED VERTEXS.

C, A, B ACYCLIC

C, A, D, C CYCLIC

PATH: SEQUENVE OF VERTEXS IN THE DIRECTION OF THEIR CONNECTIONS

A → D → B acyclic.

D → B → D cyclic

Both: ORDER: NUMBER OF VERTEXES

SIZE: NUMBER OF EDGES

Implementing Graphs

Graph ADT Ops

CONSTRUCTOR (ININTIALLY EMPTY)

INSERT VERTEX, INSERT EDGE, (BETWEEN EXISTING VERTEXS)

DELETE VERTEX (& ITS EDGES), DELETE EDGE.

LOOKUP VERTEX (MIGHT RETURN ASSOC. DATA)

LOOKUP EDGE, is Empty. DEGREE, SIZE, ORDER

+ OTHERS.

Graph Class

```
public class graph<T> {
private List<Graph node<T>> nodes;
```

↳ ArrayList, LinkedList

MAP → TreeMAP, HashMAP (DUP ok)

SET → TreeSet, HashSET (NO Dups)

Graphnode Class

```
class Graph node<T> {
private T data;
```

How DO WE STORE EDGES?

Using Edge Representations

→ Write the code to be added to a `Graph` class that computes the degree of a given node in an undirected graph.

1. Adjacency list:

```
public int degree( Graphnode<T> n) {
    return n.getEdges().size();
}
```

2. Adjacency matrix:

```
public int degree( Graphnode<T> n) {
    int i = MAP n's key to its AM INDEX.
    int degree = 0;
    for (int s = 0; s < #Nodes; s++)
        if (edges[i][s]) degree++;
    return degree;
}
```


Comparison of Edge Representations

Ease of Implementation

Both EASY — AM REQUIRES MAPING KEY -> INDE

Space (memory)

AM $O(N^2)$

AL $O(N)$ AVG CASE — SPARSE GRAPH
$O(N^2)$ WORST CASE COMPLETE GRAPH

Time (complexity of ops)

DEPENDS ON THE OPERATIONS

node's degree?

AM $O(N)$

AL $O(1)$

edge exit between two given nodes?

AM CHECK AM[A][B] $O(1)$

AL SEARCH of $O(N)$ W.C

Representing Edges

Adjacency Matrix

STORES EDGES IN A 2D ARRAY

① → ② ≡ (i, j) ≡ AM[i][j]

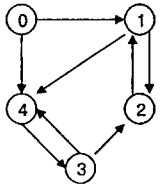
NEED TO MAP NODE'S KEY TO ITS AM INDEX

ADD TO GRAPH class

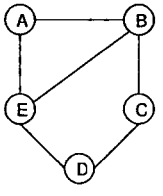
private boolean [][] edges;

Given the following graphs:

Graph 1



Graph 2



→ **Show the adjacency matrix representation of the edges for each of the graphs:**

Graph 1

TARGET TO

SOURCE FROM

	0	1	2	3	4
0		T			T
1			T		T
2	T				T
3			T		T
4				T	

Graph 2

	A	B	C	D	E
A		T			T
B	T		T		T
C		T		T	
D			T		T
E	T	T		T	

UNDIRECTED MATRIX mirror image.

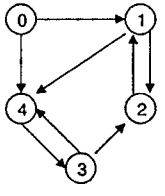
Representing Edges

Adjacency Lists

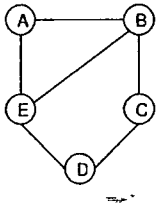
- Stores a list of successor in each Graphnode
- Add to Graphnode class
  private List<Graphnode<T>> edges;

Given the following graphs:

Graph 1



Graph 2



→ **Show an adjacency list representation of the edges for each of the graphs:**

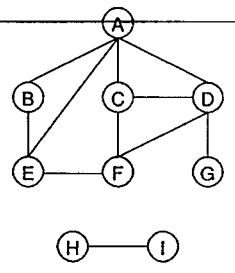
	Graph 1		Graph 2
0:	1, 4	A:	B, E
1:	2, 4	B:	A, C, E
2:	1	C:	B, D
3:	2, 4	D:	C, E
4:	3	E:	A, B, D

Searches and Traversals

Search LOOK THROUGH A COLLECTION STOPPING AT THE FIRST ITEM THAT MATCHES THE SEARCH CRITERIA

Traversal VISIT EACH ITEM IN THE COLLECTION EXACTLY ONCE.

GRAPH: SPECIFY START VERTEX AND VISIT THOSE VERTEXES THAT ARE REACHABLE.



connected compoient

→ What is the length of the longest path starting at A?

VISITING EACH VERTEXES AT MOST ONCE.

PROB. WHAT DO YOU DO TO AVOID CIRCLE?

SOLN: MARK VERTEXES AS THEY'RE VICITED

☆ 367 CONVENTION

PICK UNVICITED SUCCESSORS IN INCREASING NUMRIC ORDER OR ALPHABETIAC ORDER.

CS 367 Announcements
Thursday, December 3, 2015

Homework h10 assigned 12/6

Program p5 due 10 pm, Tuesday, December 15th

Last Time
- Finish Hashing
- ADTs/Data Structures Revisited
- Graphs
  - terminology
  - implementation issues
  - edge representations
- Return Exam 2

Today
- Graphs
  - traversals
  - applications of BFS/DFS
  - more terminology

Next Time
- Read: finish *Graphs*, start *Sorting*
  - topological ordering
  - Dijkstra's algorithm
- Sorting Intro
- Basic Sorts
  - bubble sort
  - insertion sort
  - selection sort

Depth-First Search (DFS)

- ASSUME ALL VERTEXES INITIALLY MARKED "UNVISITED"
- RELIES ON A STACK, WE'LL USE THE CALL STACK WITH RECURSION

Algorithm

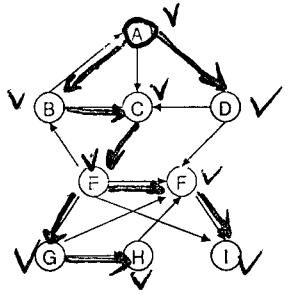
DFS(V)
MARK V AS "VISITED"
FOR EACH UNVISITED SUCCESSOR S THAT IS ADJACENT TO V
DFS(S)

↓ 367 CONVENTION

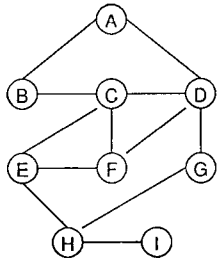
* EQUIVALENT TO PREORDER TRAVERSAL ON TREES.

DFS Practice

Graph 1



Graph 2



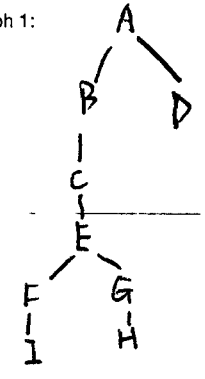
→ Give the order that vertexes are visited for depth-first search (DFS) starting at A

Graph 1: A, B, C, E, F, I BACK TRACK G, H, BACK TRACK BACK TRACK DONE

Graph 2: A B C D F E H G I

→ Give the DFS spanning tree starting at A

Graph 1:



Graph 2:

Breadth-First Search (BFS)

- ASSUME ALL VERTEXES INITIALLY MARKED "UNVISITED."
- RELIES ON A QUEUE.

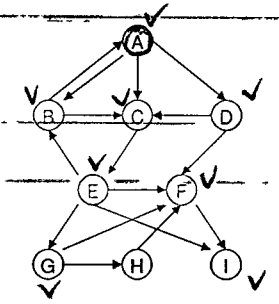
Algorithm

```
BFS(v)
  q = new Queue();
  MARK v AS VISITED
  q.enqueue(v)
  while(!q.isEmpty())
    c = q.dequeue()
    FOR EACH UNVISITED SUCCESSOR
      s ADJACENT TO c,
        MARK s AS VISITED
        q.enqueue(s);
```

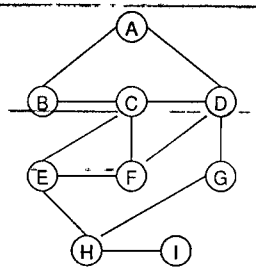
* Equivalent to A LEVEL ORDER TRAVERSAL ON TREES

BFS Practice

Graph 1



Graph 2



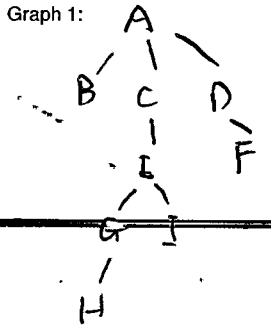
→ Give the order that vertexes are visited for breadth-first search (BFS) starting at A.

Graph 1: FRONT REAR ~~A B C D E F G H I~~ A, B, C, D, E, F, G, I, H

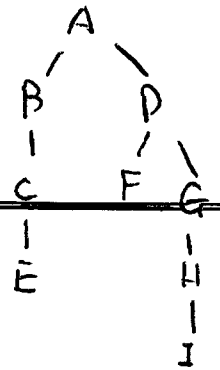
Graph 2: A B D C F G E H I

Give the BFS spanning tree starting at A.

Graph 1:



Graph 2:



Applications of DFS/BFS

- IS THE GRAPH CONNECTED? STARTING AT A GIVEN
- WHAT VERTEXES ARE REACH ABLE FROM START? VERTEX

Path Detection

- IS THERE A PATH FROM START TO X?
- WHAT IS A PATH FROM START TO X?
  IDEA: DO BFS / DFS MODIFIED TO KEEP A PREDECESSOR LIST THAT IS USED TO RECONSTRUCT THE PATH (TODAY)
- WHAT IS THE SHORTEST PATH FROM START TO X? IDEA UNWEIGHTED GRAPH: BFS WILL DETERMINE THE FEWEST EDGES PATH WITH.
  IDEA: WEIGHED GRAPH: DIJKSTRA'S ALGO WILL DETERMIN THE PATH WITH LOWEST TOTAL EDGE WEIGHTS

Cycle Detection

- IS THERE A CYCLE FROM START TO START
- IDEA: DO DFS/BFS MODIFIED TO DETECT IF START IS REACHABLE FROM SOME OTHER VERTEX IN GRAPH. (EXCLUDING SIMPLE CIRCLES
  - BACK AND FORTH AS IN (A)—(B) A, B, @A

A, B, A

WE WANT A LOOP (3 OR MORE VERTEXES)

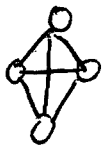
- IS THERE A CIRCLE ANYWHERE IN THE GRAPH? SEE READING FOR INFO ON HOW TO USE 3 MARKS. (ADD AN "IN PROGRESS" MARK)

More Graph Terminology

WEIGHED GRAPH: EDGE ARE ASSIGNED A VALUE REPRESENTING A COST, TIME, DISTANCE, FORCE, ...

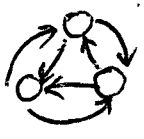
NETWORK: WEIGHTED @ DIGRAPH WHERE EDGE WEIGHTS ARE NON-NEGATING

Complete: GRAPH: AN EDGE EXISTS BETWEEN EVERY PAIR OF VETEXES.



N = # of vertexes

EDGES = $\frac{N(N-1)}{2}$



#EDGES = $N(N-1)$

CONNECTED GRAPH:

UNDIRECTED: A PATH EXISTS BETWEEN EVERY PAIR OF VERTEXS

DIRECTED: WEAKLY: A PATH EXISTS BETWEEN EVERY PAIR OF VERTEXES regardless of ~~edge~~ EDGE DIRECTION.

STRONGLY: RESPECTING EDGE DIRECTIONS.

TREE DIRECTED ACYCLIC GRAPH

CS 367 Announcements
Tuesday, December 8, 2015

Homework h10 due 10 pm, Friday, December 11th

Program p5 due 10 pm, Tuesday, December 15th

Last Time

Graphs
- more terminology
- traversals
- applications of BFS/DFS

Today

Graphs
- applications of BFS/DFS (from last time)
- topological ordering
- Dijkstra's algorithm

Sorting Intro

Basic Sorts
- bubble sort
- insertion sort
- selection sort

Next Time

Read: continue *Sorting*

Finish Basic Sorts

Better Sorts
- heap sort
- merge sort
- quick sort

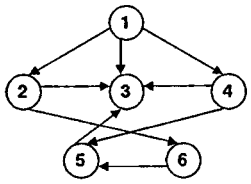
Topological Ordering

" SORTING

" NUMBERING

IDEA: COME UP WITH A LIST OF VERTEXES SUCH THAT EACH VERTEX IN THE LIST COMES BEFORE ITS SUCCESSORS.

→ **Arrange the vertices below in a list such that when the edges are added none of the arrows point to the left:**



NOT VALID

1 2 4 6 5 3 is VALID

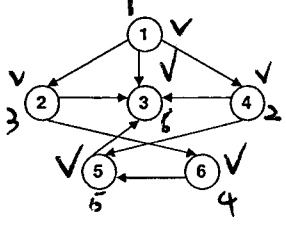
Topological Ordering Algorithm

Iterative Algorithm (READING USE RECURSION, WE'LL USE A STACK)

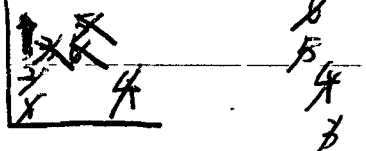
NUM# = NUMBER OF VERTEXES IN GRAPH
ST = new STACK
MARK ALL VERTEXES AS UNVISITED
FOR EACH VERTEX WITH NO PREDECESSORS
  MARK V AS VISITED
  ST. PUSH(V)
WHILE (! ST. ISEMPTY ())
  V. = ST. peek()
  IF ALL SUCCESSORS V MARKED VISITED
    ST. POP()
    GIVE V value of Num
    Num--
  ELSE
    SELECT ONE OF V's UNVISITED SUCCESSORS
    MARK V AS VISITED
    ST. PUSH(V)

USE 367 CONNECTION

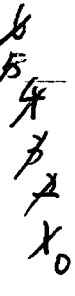
Example



ST



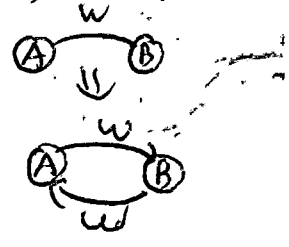
NUM



1 4 2 6 5 3

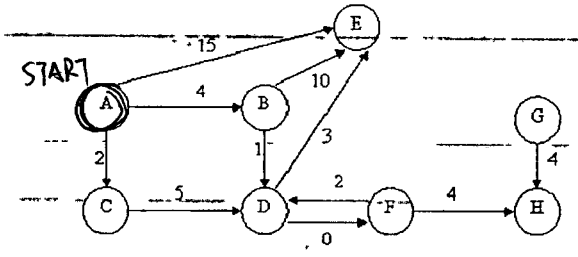
Dijkstra's Algorithm

- FINDS SHORTEST PATH (SMALLEST TOTAL EDGE WEIGHTS) IN A NETWORK.
- EDGE WEIGHTS MUST BE NON-NEGATIVE
- .... CAN BE UNBOUNDED
- MUST SPECIFY A SIMPLE START VERTEX.
- CAN BE USED WITH AN UNDIRECTED GRAPH BY CONVERTING EDGES




```
for each vertex V
   initialize V's visited mark to false
   initialize V's total weight (tw) to infinity
   initialize V's predecessor to null
set start_vertex's total weight (tw) to 0

create new priority queue pq
put (0, start_vertex) on pq

while !pq.isEmpty()
   (V's tw, V) = pq.removeMin()
   set V's visited mark to true

   for each unvisited successor S adjacent to V
      if S's total weight can be reduced
         update S's total weight to: V's tw + edge weight from V to S
         change S's predecessor to: V
         put (S's tw, S) on pq (or just update S's tw if already on pq)
```

Dijkstra's Practice



Iteration	Priority Queue (MIN … MAX)
0	0,A
1	2,C 4,B 15,E
2	4,B 7,D 15,E
3	5,D 14,E
4	5,F 8,E
5	8,E 9,H
6	9,H
7	

Vertex	Visited	Total Weight (tw)	Predecessor
A	T	0	\
B	T	4	A
C	T	2	A
D	T	5	B
E	T	8	D
F	T	5	D
H	T	9	F
G	F	∞	\

UNDERLINED MEANS ADDED DURING THAT ITERATION

Reconstruct shortest path from A to F

TRICK: START AT DESTINATION

F D B A COST 5

PATH A, B, D, F

Sorting

Problem ARRANGE A CONNECTION of items into some prescribed order. We'll do increasing numerical order

Solution Comparison sort compares pairs of items to determine their relative ordering

in java use comparable item with compareTo()

Complexity we'll use < > =

Best comparison sorts are O(NlogN)

worst case time complexity where N is number of items. 2 Dominant ops < comparison data move (swap)

In-Place Sorts

just use one array containing the collection to store & sort

Basic In-Place Comparison Sorts

IDEA. ARRAY is divided into sorted & unsorted parts.

Each pass through the array moves one item from unsorted to sorted.

Sorting the entire array requires At most N-1 passes.

Bubble Sort

Idea Each pass through the unsorted part "Bubbles" The next smallest item to the back of the sorted part.

Psuedocode

```
int passes = A.length-1;
boolean swapDone = true
for (int i = 0; i < passes && swapDone; i++) {   <- pass
    swapDone = false
  for (int j = A.length-1; j > i; j--) {
    if (A[j] < A[j-1]) {
      swap(A[j], A[j-1]);
      swapDone = true;
    }
  }
}
```

Bubbling next smallest through unsorted

Analysis MODIFIED code

	best case	worst case
kind of array	SORTED	REVERSED SORTED
# comparisons	$O(N)$	$O(N^2)$
# swaps	ϕ	$O(N^2)$
total	$O(N)$	$O(N^2) \Rightarrow O(N^2)$

Insertion Sort

Idea START WITH FIRST ITEM IN SORTED
EACH PASS INSERT NEXT UNSORTED ITEM INTO SORTED ITEM
FINDING THE INSERT LOCATION
NAIVE: LINEAR INSERTION
BETTER: BINARY INSERTION

Psuedocode (linear insertion)

```
for (int i = 1; i < A.length; i++) {          <- pass
  int temp = A[i];

  int j;
  for (j = i-1; j >= 0 && A[j] > temp; j--)
    A[j+1] = A[j];

  A[j+1] = temp;
}
```

INSERTs

Analysis NAIVE

	best case	worst case
kind of array	SORTED	REVERSE SORTED
# comparisons	$O(N)$	$O(N^2)$
# shifts	ϕ (NOT counting Assign to TEMP)	$O(N^2)$
total	$O(N)$	$O(N^2) \Rightarrow O(N^2)$

Selection Sort

Idea

- START THE ENTIRE UNSORTED
- EACH PASS SELECT SMALLEST FROM UNSORTED AND SWAP IT WITH THE FIRST UNSORTED.

Psuedocode

```
int passes = A.length-1;
for (int i = 0; i < passes; i++) {
   int minIndex = i;

   for (int j = i+1; j < A.length; j++) {
      if (A[j] < A[minIndex])
         minIndex = j;
   }

   swap(A[minIndex],A[i]);
}
```

- PASS
- SELECTION
- PLACE

Analysis

	best case	worst case
kind of array	SORT	NOT REVERSE SORTED N, 1, 2, 3 ··· N-1
# comparisons	$O(N^2)$	$O(N^2)$
# swaps	$\emptyset$ (EXCLUDE SELF SWAPS)	$O(N)$
total	$O(N^2)$	$O(N^2)$

CS 367 Announcements
Thursday, December 10, 2015

Final Exam
- Wednesday, December 23rd, 7:45 to 9:45 am (morning)
- Exam information posted
- Sample questions on Learn@UW
- UW IDs are required

Homework h10 due by 10 pm tomorrow, Friday, December 11th
- make sure your file is a pdf but not pdf scan of written work or pdf of a screen shot
- make sure you use the name h10.pdf
- submit to your `in` handin directory
- remember homeworks are to be done individually
- remember that late work is not accepted

Program p5 due 10 pm, Tuesday, December 15th

Last Time

Graphs
- applications of BFS/DFS
- topological ordering
- Dijkstra's algorithm

Today

Sorting Intro

Basic Sorts
- bubble sort
- insertion sort
- selection sort

Better Sorts
- heap sort
- merge sort
- quick sort

Next Time

Read: finish *Sorting*

~~Finish Better Sorts~~

Stable Sorts

Sorting in Java

Heap Sort

Idea (NAIVE) ① INSERT EACH ITEM FROM ORIGINAL UNSORT E ARRAY INTO A MINHEAP

② REMOVING EACH ITEM FROM MINHEAP FILLING TH original Array from left to right.

Analysis

TIME

~~① N * O(log N) = O(N log N)~~

① $N * O(\log N) = O(N \log N)$

SPACE

NAIVE: REQUIRE 2N memory (ORG, ARRAY & HEAP ARRAY) $O(2N \log N)$

$O(N \log N)$ always

BETTER (IN PLACE) too much memory

N MEMORY

HOW: use maxheap

① REHEAPIFY FOR N-1 ITEMS

② REMOVE MAX FILLING ARRAY FROM RIGHT TO LEFT

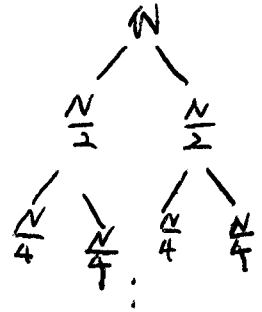
Merge Sort

Idea DIVIDE & CONQUER RECURSIVE ALGO

① DIVIDE THE ARRAY IN HALF AND RECURSIVELY CALL MERGE SORT ON EACH HALF

② MERGE THE TWO SORTED HALVES INTO 1 SORTED LIST

Analysis



BASE CASE: SUBARRAY HAVE 1 ITEM WHICH MEANS THEY'RE SORTED SO THEN MERGE

① EACH LEVEL * # OF LEVELS (H)

1 * N	O(N)
2 * N/2	O(N)
4 * N/4	O(N)
⋮	⋮
	O(N)

* log(N)

SPACE: NAIVE: REQUIRE 2N memory

BUT IN PLACE ALGO NEEDING N memory only

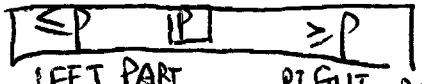
O(N) * O(logN) ⇒ O(NlogN)

Quick Sort

Idea DIVIDE & CONQUER RECURSIVE ALGO

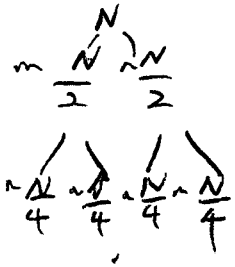
IN-Place ALGO N memory

① select a value from Array (called the pivot) AND PARTITION THE ARRAY INTO:



Analysis ② RECURSIVELY CALL QUICKSORT ON LEFT & RIGHT PARTS

BEST



PARTITION * NUMBER @ EACH LEVEL

1 * N	O(N)
2 * N/2	= O(N)
4 * N/4	= O(N)
⋮	⋮

* O(logN)

O(N) * O(logN)

⇒ O(NlogN)

BEST (AVG, BST)

Choosing a Good Pivot

BAD - PIVOT IS A[FIRST]

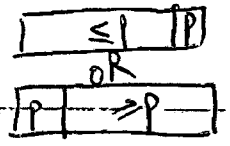
IF ARRAY IS SORTED / REVERSE SORTED

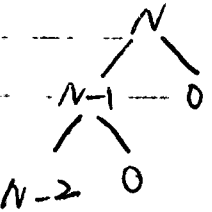
THEN THE PIVOT WILL BE SMALLEST OR LARGEST VALUE RESULTING IN EVERYTHING GOING INTO 1 PART.

Good: - "MEdian of 3" PICK THE MIDDLE VALUE of A[FIRST], A[MID], A[LAST]

WORST CASE

BAD PIVOT PARTITION RESULTS IN ONLY ONE PART AS IN:





$N+(N-1)+(N-2)+\cdots$

$O(N^2)$ (W.C)

Partitioning the Array

[INITIAL PARTITIONING]

6 | 5 9 3

2 5 4 3 7 6 2 8 2 4 1

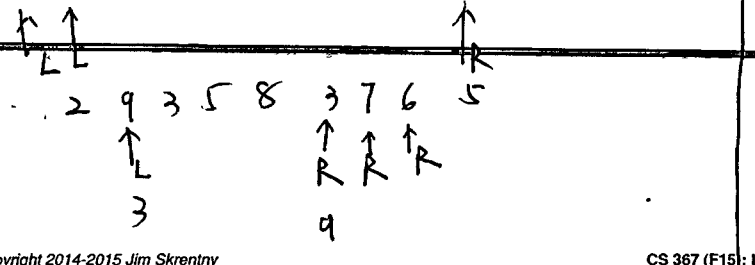
PARTITION THESE VALUES

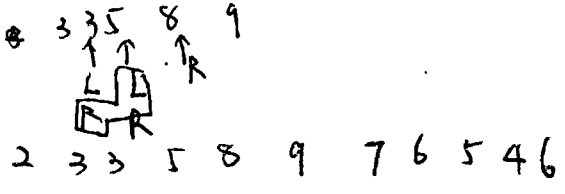
PIVOT SELECTION

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

SMALLEST

PIVOT LARGEST

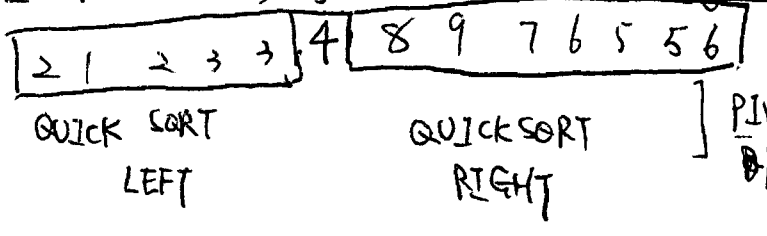




PARTITIONING UNKNOWN VALUES

LEFT (L) INCS TO VALUE > P OR TILL CROSSES R

RIGHT (R) DECS TO VALUE ≤ P OR TILL CROSSES L



QUICK SORT LEFT

QUICKSORT RIGHT

PIVOT PLACEMENT

CS 367 Announcements
Tuesday, December 14, 2015

Final Exam

- Wednesday, December 23rd, 7:45 to 9:45 am (morning)
- Lec 1: B10 Ingraham Hall
- Lec 2: 145 Birge Hall
- UW ID REQUIRED
- Bring #2 pencils
- Exam information posted
- Sample questions on Learn@UW -SOLUTION 12/21

Program p5 due 10 pm, Tonight, December 15th

Verify your scores are correctly entered on Learn@UW

Last Time

Sorting Intro

Basic Sorts

- bubble sort
- insertion sort
- selection sort

Better Sorts

- heap sort
- merge sort
- quick sort

Today

Finish Better Sorts (from last lecture)
Stable Sorts
Sorting in Java
Course Overview Sheets
Final Exam Info
Evaluations – Skrentny, CS 367, lecture

REVIEW OLD EXAMS

Stable Sorts

→ **What do you notice about the sorting of the following three lists of names?**

UNSORTED	SORTED BY FIRST NAME	STABLE SORTED BY LAST NAME
Jane Jetson	Barney Rubble	Stewie Griffin
Elroy Jetson	Elroy Jetson	Elroy Jetson
Homer Simpson →	George Jetson	George Jetson
Marge Simpson	Homer Simpson	Jane Jetson
Stewie Griffin	Jane Jetson	Judy Jetson
Judy Jetson	Judy Jetson	Barney Rubble
George Jetson	Marge Simpson	Homer Simpson
Barney Rubble	Stewie Griffin	Marge Simpson

* USE ON COMPOSITE ITEMS (HAVE MULTIPLE KEYS)

- STABLE SORTS PRESERVE THE RELATIVE ORDERING FOR DUPLICATE KEYS.

STABLE	UNSTABLE
BUBBLE	SELECTION
INSERTION	HEAP
MERGE	

? QUICKSORT ?

Sorting in Java API

In `java.util`

`Collections.sort(List)`

USES A MODIFIED MERGE SORT.
SINCE OBJECTS IN THE COLLECTION ARE LIKELY TO BE COMPOSITES.

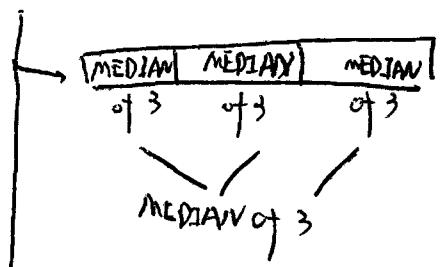
`Arrays.sort(array_to_sort)`

OVERLOADED

IF ARRAY OF REFERENCES

IF ARRAY OF PRIMITIVES

TUNED QUICKSORT



IF SUB ARRAY IS SMALL THEN INSERTION SORT IS USED.

Abstract Data Types (ADTs) and Data Structures (DS)

ADT L1P4
DS L6P6

Layout of Collection

- Linear

`List`	array, `SimpleArrayList`, shadow array chain of nodes, `Listnode`, `SimpleLinkedList` `tail`, `header`, doubly linked, circularly
`linked`	
`Stack`	
`Queue`	
`Deque`	circular array

- hierarchical

	general tree, `Treenode`
	binary tree, `BinaryTreenode`
	binary search tree, `BSTnode`
	balanced search tree
	red-black tree
`PriorityQueue`	heap

- graphical

`Graph`	`Graphnode`
	adjacency matrix
	adjacency list

Orientation of Operations

- position oriented - operations occur at a specified position

  list, stack (top), queue (front/rear), deque ("double ended")

- value oriented - operations occur at position determined by item's key value

	sorted list
	search trees
`Map`	hash table

- hybrid?

`PriorityQueue`	heap
	hash table

Algorithms

Operations on ADTs/data structures

insert, lookup, remove

Recursion

vs. iteration
rules, guiding questions
call stack trace
execution tree trace

Traversing

list			
tree	level	pre/in/post	
graph	DFS (stack)	BFS (queue)	spanning trees

Searching

linear O(N)
binary O(logN)

Hashing

hash function: hash code (extracting, weighting, folding) → hash index (compressing)
table size: prime size, load factor, rehashing
collisions: open addressing, buckets

Graphs

topological ordering
Dijkstra's (priority queue)

Sorting

basic $O(N^2)$: bubble, insertion, selection

better O(NlogN): heap, merge, quick

stable sorts

Complexity

Complexity

1, logN, N, NlogN, N^2, N^3, 2^N, N!

time: abstract, dominant ops

space: memory

worst/average/best-case

big-O

Determining Complexity

informal
- constant
- linear
- quadratic

code
- loops
- method calls

time equation
- simplify

recurrence equations
- base T( ) =
- recursive T(N) = + T ( )
- equations → table, guess solution → verify → complexity

Caveats

small problem size
same complexity

Java Concepts

Primitives vs. References

Command-line Arguments

Exceptions

`throw`
`try/catch/finally`
`throws` (checked vs unchecked)
defining

Programming for Generality

`Object`
generics

Interfaces

`Comparable, compareTo`
ADTs

Iterators

```
Iterable: iterator()
Iterator: hasNext(), next()
```

indirect
direct

Package Visibility

Java Collections Framework

```
Iterable<T>, Iterator<E>
List<T>: ArrayList<T>, LinkedList<T>
Vector<E>, Stack<E>
Hashtable<K,V>
Map<K,V>: TreeMap<K,V>, HashMap<K,V>
Set<E>: TreeSet<E>, HashSet<E>
```

Computer Sciences 367
Midterm Exam 1, 17%

Tuesday 10/20, 2015

PRINT last name (family): Jiang first (given): Han

CIRCLE lecture: Lecture 1 (1-2:15) Lecture 2 (2:30-3:45)

I agree that:
- I'll keep my answers covered during the exam so they may not be viewed by other students.
- I won't use any notes, devices of any kind, or the help from other students.
- I won't copy any part of this exam or share it with any students that haven't taken the exam.

SIGN: Han Jiang
Violation of this agreement results in automatic course failure, which is reported to the Dean's Office.

(1) **SCAN IN** your UW student ID and **TAKE** a scantron form.
(2) On the identification side of the scantron form using a #2 pencil:
WRITE your name and fill in the bubbles.
WRITE your UW student ID number and fill in the bubbles.
WRITE your lecture number under "A" in the *Special Codes* section and fill in the bubble.
WRITE exam version *1* under "B" in the *Special Codes* section and fill in the bubble.
(3) Fill in the information at the top of this page.
(4) Turn off and put away ALL electronic devices before the exam begins.
(5) Check that there is a total of 9 pages in this exam.
(6) When done, **SCAN OUT** your UW student ID and **TURN IN** your exam and scantron form.

This exam is intended to take 90 minutes, but **we'll give you 2 hours.** Answer questions by filling in your choice using a #2 pencil on the scantron form. Only answers on your scantron form matter. Marks in this exam don't count. Unless otherwise specified, assume the classes, ADTs and interfaces are those discussed in lecture and in the readings. We can't provide hints but if you need an exam question clarified or feel that there is an error, please bring this to our attention. **If needed, corrections are written on the board.**

Parts	Number of Questions	Question Format	Possible Points
I	21	Dual Choice	21
II	15	Multiple Choice	45
		Total	66

Part I Dual Choice *[21 questions, 1 point each, 21 total points]*

For questions 1 through 21, choose your answer after reading both choices.
Mark the corresponding letter on your answer sheet.

1.) What is a benefit of declaring `myList` in this manner:

```
ListADT<String> myList = new SimpleArrayList<String>();
```

A. Any class implementing the `ListADT` can be substituted for `SimpleArrayList`.
B. Any methods implemented by `SimpleArrayList` can be used by `myList`.

2.) An ADT specifies:

A. How something is structured and how its operations are implemented.
B. What something is and what operations it can do.

3.) A **Stack** ADT has a A.) FIFO B.) LIFO behavior.

4.) If each iteration of an algorithm reduces the problem size by one half
then the time **complexity** of that algorithm is A.) linear. B.) logarithmic.

For the next 2 questions, consider the following **try-catch statement** shell:

```
try {
   //LOCATION 1
} catch (SomeException e) {
   //LOCATION 2,
} finally {
   //LOCATION 3
}
```

5.) If any exception is thrown at `LOCATION 1`
then the code at `LOCATION 3` A.) does execute. B.) doesn't execute.

6.) If code at `LOCATION 2` executes
then the code at `LOCATION 3` A.) does execute. B.) doesn't execute.

7.) Java's **package** access is used to hide:

A. An `ArrayList`'s array data structure.
B. A `LinkedList`'s `Listnode` data structure.

8.) Which one describes what an **`Iterator`**'s `next()` method does?

A. It advances to the next item and then returns a reference to that item.
B. It returns a reference to the current item and then advances to the next item.

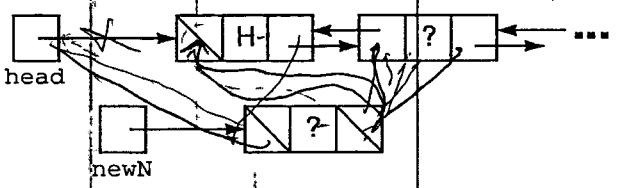
9.) **Java generics** require the type of item that is stored in a collection to be specified when:

A. the generic class is used.
B. the generic class is implemented.

10.) Which one of the following is correct?

A. A collection class can be `Iterable`.
B. An iterator class can be `Iterable`.

11.) If the node labelled H is a header node, which one correctly links the node referenced by `newN` at the front of the list (i.e., at position 0)?



A.
```
newN.setNext(head);
head = newN;
head.getNext().setPrev(newN);
```

B.
```
newN.setNext(head.getNext());
head.setNext(newN);
head.getNext().getNext().setPrev(newN);
newN.setPrev(head);
```

For the next 3 questions, you are given the following **time equations** for three methods:

`method1` is $T(N) = 3N^2 - 7$; $O(N^2)$
`method2` is $T(N) = N^2 - 5N$ $O(N^2)$
`method3` is $T(N) = 30N + 11$ $O(N)$

For each of the following, Mark A for if the statement is true and B if it is false.

A 12.) `method1` has quadratic time complexity.

B 13.) `method3` is always the fastest method.

A 14.) The time complexity of `method1` followed by `method2` followed by `method3` is $O(N^2)$.

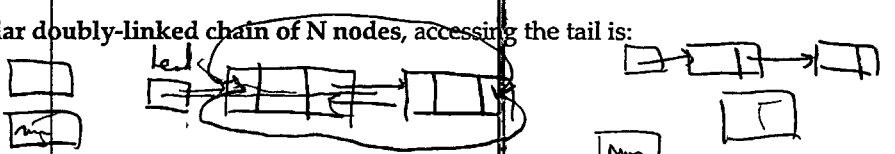
A 15.) A **shadow array** improves the worst-case time complexity for `SimpleArrayList` add operations by spreading the cost of copying required to expand the array across multiple add operations. Mark A for true or B for false.

B 16.) If `ListADT` is a Java interface then the following A.) is B.) isn't valid Java code:

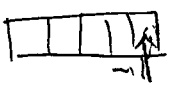
```
ListADT<String> myList = new ListADT<String>();
```

A 17.) With only a head reference to a **circular doubly-linked chain of N nodes**, accessing the tail is:

A. O(1) B. O(N)



For the next 3 questions, consider implementing an **indirect access** reverse iterator, which steps backwards through the items in a `ListADT`, i.e., from the last item to the first. What is the best you can achieve for the `prev` method's worst-case time complexity given `ListADT` implementations listed below?

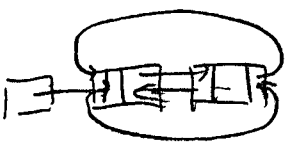


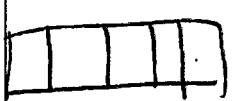
A 18.) An `ArrayList` that uses an array A.) O(1) B.) O(N)

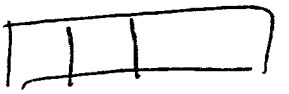
B 19.) A `LinkedList` that uses a singly-linked chain of nodes A.) O(1) B.) O(N)

B 20.) A `LinkedList` that uses a circular, doubly-linked chain of nodes A.) O(1) B.) O(N)

B 21.) Implementing a Stack ADT with an array would best be done by putting the top of the stack at:
A.) index 0 B.) index N-1 where N is the number of items in the stack







Part II Multiple Choice *[15 questions, 3 points each, 45 total points]*

For questions 22 through 36, choose the one *best* answer after reading all of the choices. Mark the corresponding letter on your answer sheet.

22.) Consider the following code fragment using a `Stack<E>` S and a `Queue<E>` Q:

```
Queue<String> Q = new Queue<String>();
for (int i = 0; i < 4; i++) {
   Q.enqueue(S.pop());
}
while (!Q.empty()) {
   S.push(Q.dequeue());
}
```

Assume S initially contains the `Strings` (listed from top to bottom): "toy" "jim" "cat" "bob" "mop". What would `Stack` S contain after the code fragment executes?

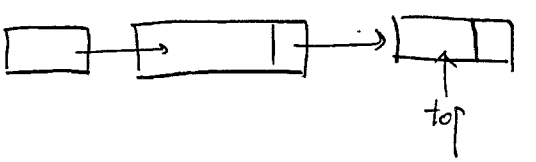
top bottom

A. "bob" "cat" "jim" "toy" "mop"
B. "bob" "jim" "cat" "toy" "mop"
C. "mop" "jim" "cat" "bob" "toy"
D. "mop" "toy" "jim" "cat" "bob"
E. "toy" "jim" "cat" "mop"

23.) Assume `System.out.println` is O(1). What is the worst-case **complexity** of the following code fragment assuming the size of the list is N items:

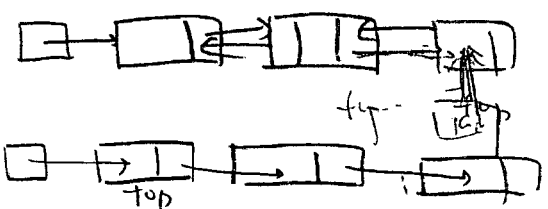
```
if (list.contains(x))
   System.out.println(x);
else
   System.out.println("not found");
```

A. O(1)
B. O(logN)
C. O(N)
D. $O(N^2)$
E. $O(N^3)$



24.) Consider the following proposed designs for implementing a **stack** (assume non-circular chains of nodes):

I. A singly-linked chain of nodes without a last-node reference and the top of the stack is at the rear.
II. A doubly-linked chain of nodes without a last-node reference, and the top of the stack is at the rear.
III. A singly-linked chain of nodes without a last-node reference, and the top of the stack is at the front.



For stacks with a large number of items, which one of the following statements is *false*?

A. Design *II* uses the most memory.
B. Design *III* uses the least memory and has the fastest `push` and `pop` methods.
C. Adding a last-node reference to design *I* would make its `push` method much faster.
D. Adding a last-node reference to design *I* would make its `pop` method much faster.
E. Adding a last-node reference to design *II* would make its `pop` method much faster.

The questions on this page are based on the following code that is *partially* shown below:

```
public class Exceptions {
   public static void main(String[] args){
      try {
         //LINE A - see questions below
         methodX();
      } catch (FileNotFoundException e) {
         System.out.print("Error 1,");
      } catch (IndexOutOfBoundsException e) {
         System.out.print("Error 2,");
      }
      System.out.print("main done,");
   }

   public static void methodX() {
      //LINE B - see questions below
      try {
         methodY();
      } catch (IndexOutOfBoundsException e) {
         System.out.print("Error 3,");
      }
      System.out.print("methodX done,");
   }

   public static void methodY() {
      try {
         //LINE C - see questions below
      } catch (ArithmeticException e) {
         System.out.print("Error 4,");
      }
      System.out.print("methodY done,");
   }
}
```

25.) What is output if only **LINE B** is replaced with code that throws an `IndexOutOfBoundsException`?

A

A. `Error 2,main done,`
B. `Error 3,methodX done,main done,`
C. `methodX done,Error 2,main done,`
D. `methodY done,Error 3,methodX done,main done,`
E. The program crashes and message about the exception is displayed.

D

26.) What is output if only **LINE C** is replaced with code that throws an `ArithmeticException`?

A. `Error 4,`
B. `Error 4,methodX done,main done,`
C. `methodY done,methodX done,main done,`
D. `Error 4,methodY done,methodX done,main done,`
E. The program crashes and message about the exception is displayed.

C

27.) If **LINE C** is replaced with code that throws the *checked exception*, `FileNotFoundException`, which methods would require a `throws` clause be added to their method header for this exception?

A. only `main`
B. only `methodY`
C. only `methodY` and `methodX`
D. only `methodY` and `main`
E. All three methods - `methodY`, `methodX` and `main`

28.) Assume the following code is in the same class:

```
private List myList;

//adds the items in list L to myList
public void addList( List L ) {
   for (Iterator itr = L.iterator(); itr.hasNext(); ) {
      myList.add(0, itr.next());
   }
}
```

If `myList` contains (A, B, C) and `L` contains (X, Y, Z), which one of the following shows, in the correct order, what `myList` contains after `addList` is called.

E

A. A, B, C, X, Y, Z
B. A, B, C, Z, Y, X
C. X, A, Y, B, Z, C
D. X, Y, Z, A, B, C
E. Z, Y, X, A, B, C

A 29.) Assume for the question above, that the iterator constructor, `hasNext` and `next` methods all run in constant time. If `myList` is a `LinkedList` containing *N* items, and `L` contains *M* items, what is the time complexity of method `addList`?

A. O(*M*)
B. O(*N*)
C. O(*N* + *M*)
D. O(*N* * *M*)
E. O(max(*N*, *M*))

30.) Consider the following method:

```
public static boolean mystery (Queue<Object> Q, Object ob) {

   boolean result = false;

   for (int i = Q.size(); i > 0; i--) {
      Object tmp = Q.dequeue();
      if (!result && tmp.equals(ob)) {
         result = true;
      }
      Q.enqueue(tmp);
   }
   return result;
}
```

B Which one of the following *best* describes what `mystery` does?

A. It returns true iff the queue does *not* contain `ob` but it does *not* restore the queue to its original state.
B. It returns true iff the queue contains at least one occurrence of `ob` and it restores the queue to its original state.
C. It returns true iff the queue contains at least one occurrence of `ob` but it does *not* restore the queue to its original state.
D. It returns true iff the queue contains more than one occurrence of `ob` and it restores the queue to its original state.
E. It returns true iff the queue contains more than one occurrence of `ob` but it does *not* restore the queue to its original state.

31.) Assume a class has instance variables `head` and `tail` referencing a doubly-linked chain of nodes. Also assume the chain will have an odd number of nodes. Consider adding a method that determines if the data in the chain of nodes are the same forwards as backwards. For example, the chain of three items: "go", "badgers", "go" would be true, but "bucky", "badgers", "go" would not. The chain "one", "two", "three", "two", "one" would be true, but "one", "two", "three", "six", "one" would not. Which one of the following implementations of the method is the *best* choice for setting `mirrored` to true iff the chain is as described above?

A.
```
boolean mirrored = false;
while (head != tail) {
   if ( head.getData().equals(tail.getData()) ) mirrored = true;
   head = head.getNext();
   tail = tail.getPrev();
}
```

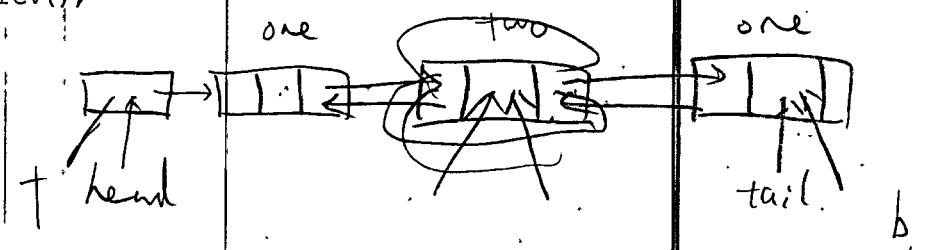
B.
```
boolean mirrored = true;
while (head != tail) {
   if ( !head.getData().equals(tail.getData()) ) mirrored = false;
   head = head.getNext();
   tail = tail.getPrev();
}
```

C.
```
boolean mirrored = true;
while (head != tail && mirrored) {
   if ( !head.getData().equals(tail.getData()) ) mirrored = false;
   head = head.getNext();
   tail = tail.getPrev();
}
```

D.
```
boolean mirrored = false;
DblListnode<String> f = head;
DblListnode<String> b = tail;
while (f != b) {
   if ( f.getData().equals(b.getData()) ) mirrored = true;
   f = f.getNext();
   b = b.getPrev();
}
```

E.
```
boolean mirrored = true;
DblListnode<String> f = head;
DblListnode<String> b = tail;
while (f != b && mirrored) {
   if ( !f.getData().equals(b.getData()) ) mirrored = false;
   f = f.getNext();
   b = b.getPrev();
}
```

E



b

32.) Suppose you are using a **circular array** to implement a Queue. After enqueueing in the following order "A", "B", "C", and "D", the array looks like:

0	1	2	3	4	5
"A"	"B"	"C"	"D"	E	F

Which of the sequences of Queue operations below would result in the following array:

0	1	2	3	4	5
"G"	"H"	"C"	"D"	"E"	"F"

Sequence 1:
enqueue "E"
dequeue
enqueue "F"
dequeue
enqueue "G"
enqueue "H"

Sequence 2:
dequeue
dequeue
enqueue "E"
enqueue "F"
enqueue "G"
enqueue "H"

Sequence 3:
enqueue "E"
enqueue "F"
enqueue "G"
enqueue "H"
dequeue
dequeue

A. *Sequence 1* only
B. *Sequence 2* only
C. *Sequences 1* and 2 only
D. *Sequences 2* and 3 only
E. *Sequences 1, 2* and 3

33.) Consider the following *partial* implementations of two classes:

```
public class AList<E> {

   private ANode<E> items;

   public AList<E> () {
      items = new ANode<E>(null);
   }
   //...
}

public class ANode<E> {

   private E       data;
   private ANode<E> next, prev;

   public ANode (E item) {
      data = item;
      next = null;
      prev = null;
   }
   //...
}
```

Which one *best* describes the **data structure** being used by the class `AList`?

A. a singly-linked chain of nodes with a header node
B. a doubly-linked chain of nodes with a header node
C. a basic doubly-linked chain of nodes (i.e., without any additional features)
D. a singly-linked chain of nodes with a header node and a reference to the last node
E. a doubly-linked chain of nodes with a header node and a reference to the last node

34.) Consider the following code fragment that is to be used in a method of a `LinkedList` class where `items` references the header node.

```
Listnode tmp = items.getNext(); //skip past the header node
while (!tmp.getData().equals(ob)) {
   tmp = tmp.getNext();
}
```

Assume this code fragment has already executed on a **chain of `Listnodes`**. Which one of the following statements is false?

A. If `items` references a chain of `Listnodes` that does not contain `ob`, then the code fragment results in a `NullPointerException` being thrown.
B. If `items` references a singly-linked chain of `Listnodes` that contains `ob`, then `tmp` can be used to link in constant time a new node after the one containing `ob`.
C. If `items` references a doubly-linked chain of `Listnodes` that contains `ob`, then `tmp` can be used to link in constant time a new node before the one containing `ob`.
D. If `items` references a singly-linked chain of `Listnodes` that contains `ob`, then `tmp` can be used to unlink in constant time the node containing `ob`.
E. If `items` references a doubly-linked chain of `Listnodes` that contains `ob`, then `tmp` can be used to unlink in constant time the node containing `ob`.

35.) Consider the three code segments below, each of which sets `ArrayList L` to contain just the item initially in position zero. Which of the following code fragments are O(N), where N is the size of `ArrayList L`?

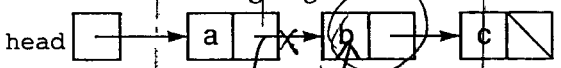
Fragment I
```
Object ob = L.get(0);
L = new ArrayList<Object>();
L.add(ob);
```

Fragment II
```
int p = L.size() - 1;
while (p > 0) {
   L.remove( 1 );
   p--;
}
```

Fragment III
```
int p = L.size() - 1;
while (p > 0) {
   L.remove( p );
   p--;
}
```

A. Fragment II only
B. Fragment III only
C. Fragments I and II
D. Fragments II and III
E. Fragments I, II and III

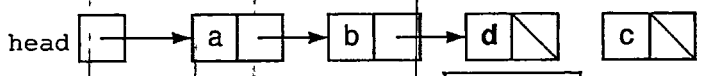
36.) Consider the following diagram of a chain of **`Listnodes`**:

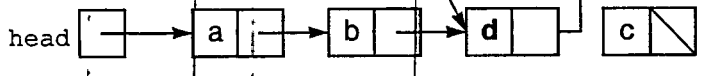


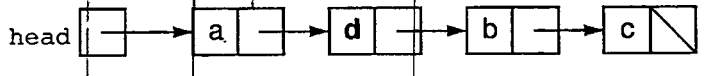
If the following code was executed:

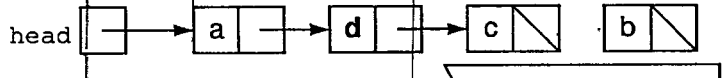
```
head.setNext(new Listnode("d", head.getNext()));
```

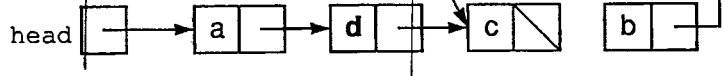
Which one of the following *best* diagrams the resulting chain of `Listnodes`?

A.


B.


C.


D.


E.


October 26, 2015
FILE: cp601

The Office of Testing and Evaluation Services
Room 373, 1025 W Johnson ST Madison, WI 53706 (608)262-5863
Section = 2

Job #: cp601
Customer #: 266

INDIVIDUALIZED LISTING OF CHOSEN VS. CORRECT RESPONSE

JIANG HAN

Special Codes: 21

Student ID: 9070414850 Your Total Score: 63.00 Maximum Possible Score: 66.00

Subscores: #1: 18.00 #2: 45.00

ITEM:	KEYS:	YOURS:
1	A	B
2	B	
3	B	
4	B	
5	A	
6	A	
7	B	A
8	B	
9	A	
10	A	
11	B	A
12	A	
13	B	
14	A	
15	A	
16	B	
17	A	
18	A	
19	B	
20	B	
21	B	
22	A	
23	C	
24	D	
25	A	
26	D	
27	C	
28	E	
29	A	
30	B	
31	E	
32	C	
33	B	
34	D	
35	B	
36	C	

* INDICATES A MULTIPLE RESPONSE X INDICATES THE ITEM WAS OMITTED

Computer Sciences 367
Midterm Exam 2, 17%

Tuesday 11/24, 2015

PRINT last name (family): JIANG, first (given): HAN

CIRCLE lecture: **Lecture 1** (1-2:15) **Lecture 2** (2:30-3:45)

I agree that:
- I'll keep my answers covered during the exam so they may not be viewed by other students.
- I won't use any notes, devices of any kind, or the help from other students.
- I won't copy any part of this exam or share it with any students that haven't taken the exam.

SIGN: HAN JIANG
Violation of this agreement results in automatic course failure, which is reported to the Dean's Office.

(1) **SCAN IN** your UW student ID and **TAKE** a scantron form.
(2) In the identification area on the scantron form using a #2 pencil:
WRITE your name and fill in the bubbles.
WRITE your UW student ID number and fill in the bubbles.
WRITE your lecture number under "A" in the *Special Codes* section and fill in the bubble.
WRITE exam version 2 under "B" in the *Special Codes* section and fill in the bubble.
(3) Fill in the information at the top of this page.
(4) Turn off and put away ALL electronic devices before the exam begins.
(5) Check that there is a total of 9 pages in this exam.
(6) When done, **SCAN OUT** your UW student ID and **TURN IN** your exam and scantron form.

This exam is intended to take 90 minutes, but **we'll give you 2 hours.** Answer questions by filling in your choice using a #2 pencil on the scantron form. Only answers on your scantron form matter. Marks in this exam don't count. Unless otherwise specified, assume the classes, ADTs and interfaces are those discussed in lecture and in the readings. We can't provide hints but if you need an exam question clarified or feel that there is an error, please bring this to our attention. **If needed, corrections are written on the board.**

Parts	Number of Questions	Question Format	Possible Points
I	21	Dual Choice	21
II	15	Multiple Choice	45
		Total	66

Part I Dual Choice *[21 questions, 1 point each, 21 total points]*

For questions 1 through 21, choose your answer after reading both choices.
Mark the corresponding letter on your answer sheet.

1.) A **binary search tree** having N nodes has a *height* in the range of $\log_2(N+1)$ to N inclusive.
Mark A for true and B for false.

2.) The main flaw of a **binary search tree** is:

A. the shape of the tree depends on the sequence of inserts and deletes.
B. the height of the tree depends on the number of children of the nodes to be deleted.

3.) If an insert into a **red-black tree** results in a *recoloring* then

A. no additional fixing to rebalance the tree is required.
B. it may be possible for additional fixing to be required to rebalance the tree.

4.) If the same values is inserted into **red-black trees** but in different orders it is possible for the resulting trees to be identical. Mark A for true and B for false.

5.) The root property of a **red-black tree** requires that the root node must

A. be black.
B. have black children.

6.) When inserting a new node into a **red-black tree**, its parent's A.) sibling B.) color determines how to rebalance the tree.

The next 3 questions are based on the following methods, which are added to a `SimpleLinkedList` class:

```
public int sumList() { return sumList(root); }
private int sumList(ListNode<E> curr) {
   if (curr == null) return 0;
   return curr.getData() + sumList(curr.getNext());
}
```

7.) This code illustrates using A.) a *companion* method. B.) *indirect* recursion.

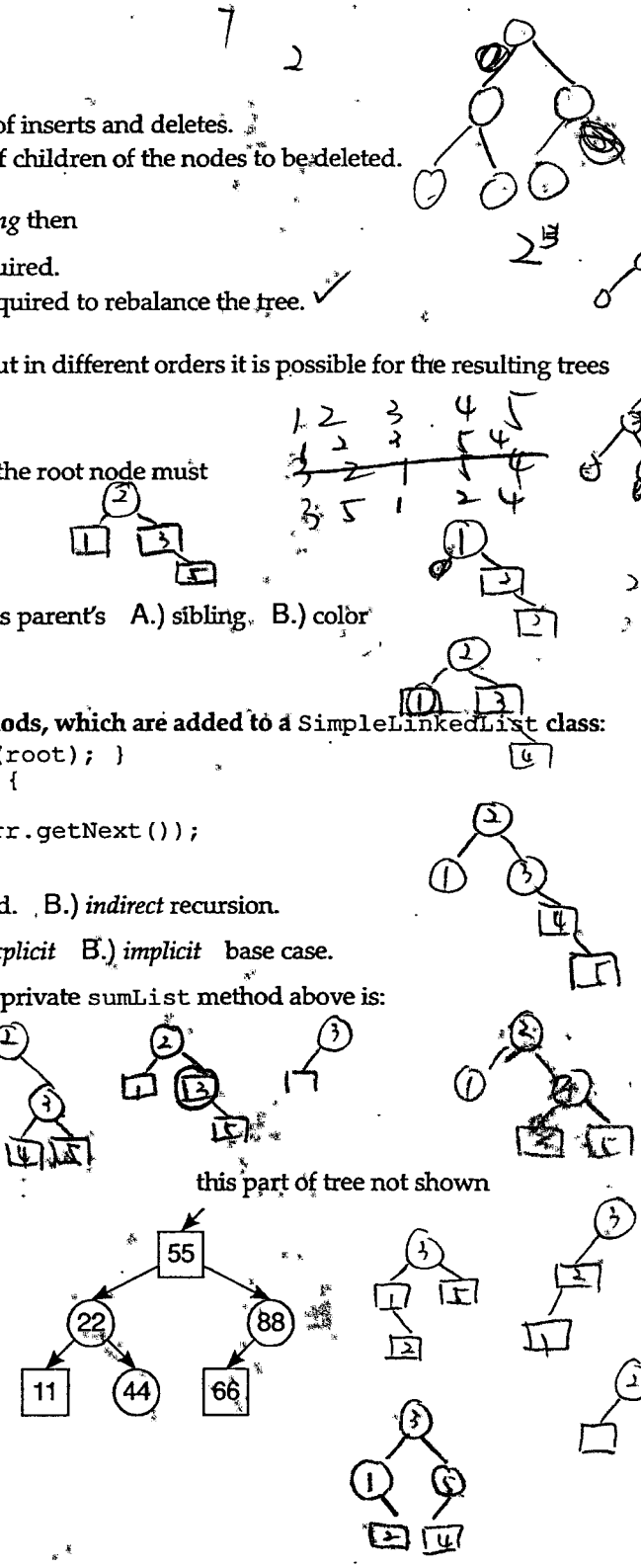
8.) The private `sumList` method above has an A.) *explicit* B.) *implicit* base case.

9.) The equation corresponding to the base case in the private `sumList` method above is:

A. T(1) = 0
B. T(0) = 1

10.) The tree partially shown on the right (parent of 55 and nodes connect to that parent are not shown) would not be a valid **red-black tree** (where circles are black nodes and squares are red) because it would violate the:

A. black property.
B. red property.



11.) If the same sequence of values is inserted into a **binary search tree** and also a **red-black tree**, then an inorder traversal on both trees visits the values in A.) a different B.) the same order.

12.) When deleting a node in a **binary search tree**, its A.) *inorder predecessor* B.) *inorder successor* will not have a right child.

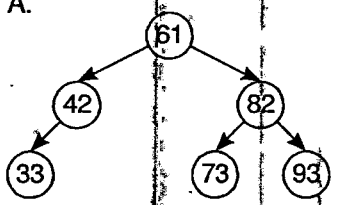
13.) An **AVL tree** requires rebalancing if the *balance value* for any node becomes A.) -1 B.) -2

14.) It is faster to find the second largest value of a collection with no duplicates if it is stored in:

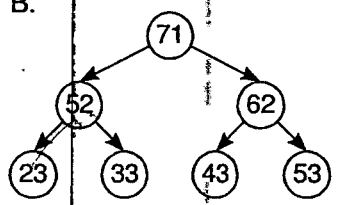
A. a red-black tree implemented using a hierarchy of tree nodes.
B. a priority queue implemented using a max-heap.

15.) Which one of the following is a *complete* **binary tree**:

A.



B.



16.) When the item at the top of a **heap** is removed the last item in the array is used as a replacement to maintain the heap's A.) *ordering constraint.* B.) *shape constraint.*

17.) If all interior (non-leaf) nodes have 2 children then all *leaf* nodes in that tree must be at the same *level.* Mark A for true and B for false.

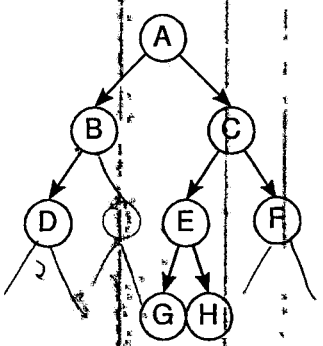
18.) Which one of the following lists of values in an array (starting at index 1) could represent a **max heap**?

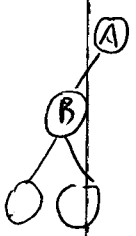
A. 99, 88, 44, 77, 33, 11, 66, 55 B. 99, 55, 77, 44, 33, 66, 11, 33

19.) A recursive method that doesn't make progress toward a base case will result in A.) infinite execution. B.) a program crash.

20.) If N values are removed from a **priority queue** and inserted into a **binary search tree** then that tree's height will be A.) O(logN) B.) O(N)

21.) The **binary tree** below A.) is B.) is *not* *height balanced.*





Part II Multiple Choice *[15 questions, 3 points each, 45 total points]*

For questions 22 through 36, choose the one *best* answer after reading all of the choices. Mark the corresponding letter on your answer sheet.

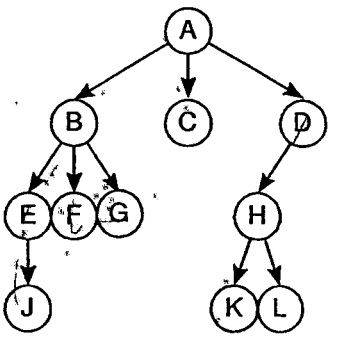
22.) Consider the following *incomplete* method:

```
public static boolean checkMinHeap( ArrayList<Comparable> al ) {
   for (int i = al.size()-1; CONDITION1; i--) {
      Comparable c = al.get(i);
      Comparable p = al.get(EXPRESSION);
      if (CONDITION2) return false;
   }
   return true;
}
```

Which one of the following replacements for CONDITION1, EXPRESSION and CONDITION2 could be used to complete the method so that it returns true if and only if the ArrayList is organized as a min-heap?

	CONDITION1	EXPRESSION	CONDITION2
A.	i > 0	i/2	c.compareTo(p) > 0
B.	i > 0	i/2 + 1	c.compareTo(p) > 0
C.	i > 1	i/2	c.compareTo(p) < 0
D.	i > 1	i/2 + 1	c.compareTo(p) < 0
E.	i > 1	i/2	c.compareTo(p) > 0

23.) Consider the general tree on the right with character labels. Which one of the following lists the order that nodes are visited for *preorder* traversal?



A. A, B, C, D, E, F, G, H, J, K, L
B. A, B, E, J, F, G, C, D, H, K, L
C. A, B, E, J, F, G, C, K, L, H, D
D. J, E, F, G, B, C, K, L, H, D, A
E. L, K, H, D, C, G, F, J, E, B, A

24.) Suppose that the recurrence equations for a recursive method are:

$T(1) = 1$
$T(N) = N + T(N - 1)$

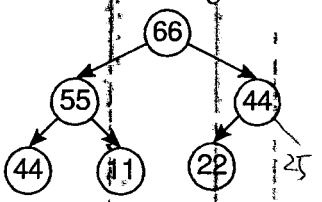
Which one of the following is the complexity in big-O notation for the method?

A. O(log N)
B. O(N)
C. O(2N -1)
D. O(N^2)
E. O(2^N)

25.) Consider inserting N integers in increasing order into both a **min heap** (MH) and a **binary search tree** (BST). Which one of the following statements about the resulting trees is true?

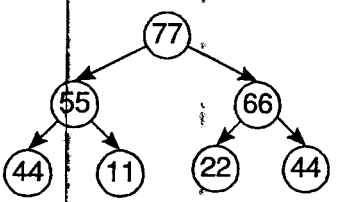
A. Both trees will have the same height.
B. Finding the largest value will take constant time for both trees.
C. Finding the smallest value will take constant time for both trees.
D. The MH and the BST will both have a bad shape (few branches).
E. The MH and the BST will both have a good shape (lots of branches).

26.) Consider the following **max-heap** that stores integers:

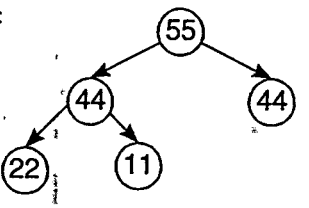


Which one of the following statements about operations on this max-heap is *false*?

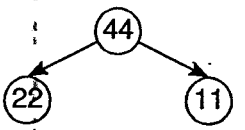
A. Inserting 25 results in an array of integers in this order: 66, 55, 44, 44, 11, 22, 25
B. Inserting 77 results in this tree:



C. A single remove operation results in an array of integers in this order: 55, 44, 44, 22, 11
D. A single remove operation results in this tree:



E. Two remove operations result in this tree:



27.) If the following lists of values were each inserted into an empty **binary search tree**:

i. 22, 11, 7, 33, 9, 37, 3, 27
ii. 22, 11, 33, 37, 27, 9, 7, 3
iii. 22, 33, 27, 11, 7, 9, 37, 3

Which would result in exactly the same tree (i.e., same shape and same order of values)?

A. none of the resulting trees are the same
B. *i* and *ii* result in the same tree
C. *i* and *iii* result in the same tree
D. *ii* and *iii* result in the same tree
E. all of the resulting trees are the same

28.) Consider the following *incomplete* implementation of a recursive method that operates on a **general tree**:

```
public static <T> int mostKids ( Treenode<T> n ) {

   if (CONDITION) return 0;

   int most = n.getKids().size();

   Iterator<Treenode<T>> itr = n.getKids().iterator();
   while (itr.hasNext()) {
      int tmp = mostKids(itr.next());
      if (tmp OPERATOR most) most = tmp;
   }
   return RETURN;
}
```

Which one of the following replacements for `CONDITION`, `OPERATOR` and `RETURN` can be used to correctly complete this method so that it returns the number of children for the node in the tree that has the most children, and ~~0 if the tree is empty?~~

	CONDITION	OPERATOR	RETURN
A.	`n == 0`	`>=`	`most + 1`
B.	`n == null`	`>`	`most`
C.	`n == null`	`<`	`most + 1`
D.	`n.getKids() == null`	`>`	`most + 1`
E.	`n.getKids() == null`	`<`	`most`

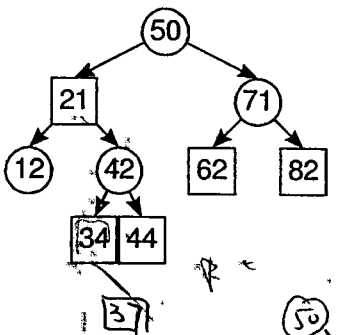
29.) Consider the following *incorrect* implementation for **binary search** on an array of `Comparable` objects:

```
public int binSearch (Comparable[] A, Comparable x, int first, int last)
{
   int mid = (first + last) / 2;
   if (x.compareTo(A[mid]) == 0) return mid;
   if (x.compareTo(A[mid])  < 0) return binSearch(A, x, first, mid-1);
   return binSearch(A, x, mid+1, last);
}
```

Which one of the following identifies what is wrong with this code?

A. The recursive cases skip the item at index `mid`.
B. The calculation for `mid` returns a result of type `double`.
C. The recursive cases are swapped so that the wrong half of the array is searched.
D. The recursive code is missing the base case for when the search item isn't found.
E. The last recursive case requires a test to determine if `x` comes after the `A[mid]`.

30.) Consider the **red-black tree** on the right storing integers (where circles are black nodes and squares are red). Which of the following occur(s) when 37 is inserted into this tree?



i. initially a red leaf node is added
ii. the tree is fixed by recoloring
iii. the tree is fixed by tri-node restructuring

A. *i* only
B. *ii* only
C. *i* and *ii* only
D. *ii* and *iii* only
E. *i*, *ii*, and *iii*

31.) Consider the following interesting **recursive definition**:

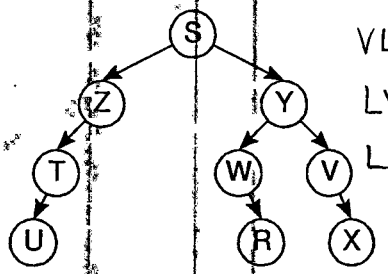
D

Function:	Return Value:	Condition:
ackermann(m, n) = {	n + 1	if m = 0
	ackermann(m -1, 1)	if n = 0
	ackermann(m -1, ackermann(m, n -1))	otherwise

Which statement below is *false*?

A. `ackermann(0,1)` returns 2
B. `ackermann(0,3)` returns 4
C. `ackermann(1,0)` returns 2
D. `ackermann(1,1)` returns 3
E. `ackermann(2,0)` returns 3

32.) Consider the following **binary tree** with character labels:



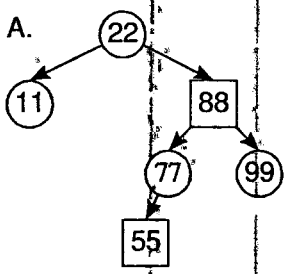
C Which one of the following lists is *not* one of the four orderly traversals (i.e., pre/in/post/level order)?

A. S, Z, Y, T, W, V, U, R, X
B. S, Z, T, U, Y, W, R, V, X
C. U, T, Z, S, R, W, X, V, Y
D. U, T, Z, S, W, R, Y, V, X
E. U, T, Z, R, W, X, V, Y, S

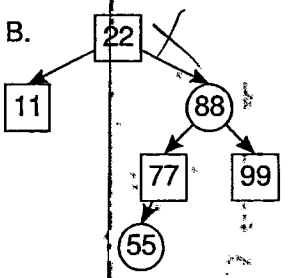
33.) Which one shows the correct **red-black tree** (where circles are black nodes and squares are red) that results from inserting in this order 11, 99, 22, 88, 77, 55?

A

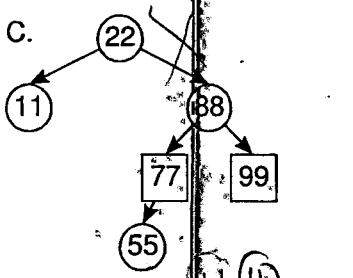
A.



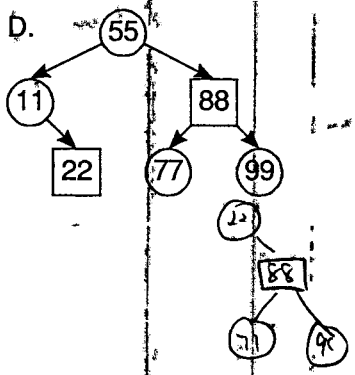
B.



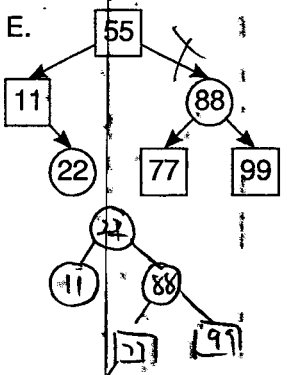
C.



D.



E.

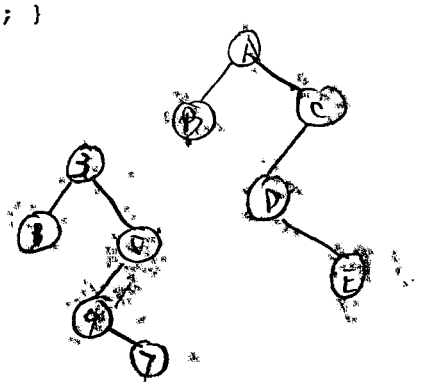


34.) Assume general trees are implemented using a `Treenode` class that includes the following:

```
private int data;
private List<Treenode> children;
private int getData() { return data; }
private List<Treenode> getChildren() { return children; }
```

and that the following class is used to represent general trees and it contains two mystery methods:

```
public class Tree {
   private Treenode root;

   public Treenode mystery() { return mystery(root); }

   private Treenode mystery( Treenode n ) {

      if (n.getChildren().size() == 0) return n;

      Treenode nd = n;
      Iterator itr = n.getChildren().iterator();
      while (itr.hasNext()) {
         Treenode t = mystery(itr.next());
         if (nd.getData() < t.getData()) nd = t;
      }
      return nd;
   }
}
```



Which of the following *best* describes what the method `mystery` does assuming the tree is not empty?

A. Returns the node with the most children.
B. Returns the node at the deepest level in the tree.
C. Returns the node with the most leaves in the tree.
D. Returns the node with the smallest data value in the tree.
E. Returns the node with the largest data value in the tree.

35.) Consider the following method:

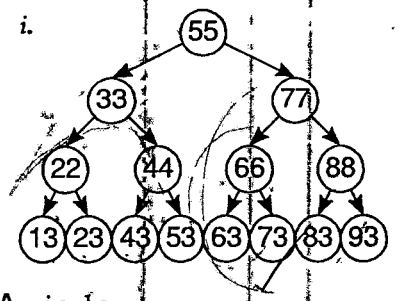
```
public static boolean mystery( Listnode<Integer> n ) {
   if (n == null) throw new IllegalArgumentException();
   if (n.getNext() == null) return true;
   if (n.getData() < n.getNext().getData())
      return mystery(n.getNext());
   else return false;
}
```

Which one of the following *best* describes what `mystery` does?

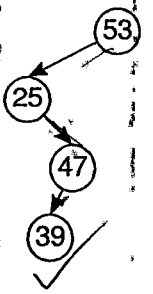
A. It returns true iff the items in a chain of nodes are in increasing order, and it correctly checks all of the items.
B. It returns true iff the items in a chain of nodes are in increasing order, but there is a bug since it doesn't correctly check the last item.
C. It returns true iff the items in a chain of nodes are in decreasing order, and it correctly checks all of the items.
D. It returns true iff the items in a chain of nodes are in decreasing order, but there is a bug since doesn't correctly check the last item.
E. It results in a NullPointerException being thrown but only if the chain of nodes has an odd number of items.

36.) Which of the following represents a **binary search tree**:

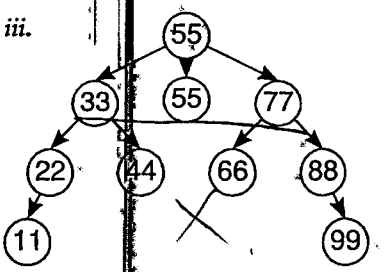
i.



ii.



iii.



A. *i* only
B. *i* and *ii* only
C. *i* and *iii* only
D. *ii* and *iii* only
E. *i, ii* and *iii*

B

November 25, 2015
FILE: cp154

The Office of Testing and Evaluation Services
Room 373, 1025 W Johnson ST Madison, WI 53706 (608)262-5863
Section = 2

Job #: cp154
Customer #: 266

INDIVIDUALIZED LISTING OF CHOSEN VS. CORRECT RESPONSE

JIANG HAN

Special Codes: 22

Student ID: 9070414850 Your Total Score: 63.00 Maximum Possible Score: 66.00
Subscores: #1: 18.00 #2: 45.00

ITEM:	KEYS:	YOURS:
1	A	
2	A	
3	B	
4	A	B
5	A	
6	A	
7	A	
8	A	B
9	B	
10	A	
11	B	
12	A	
13	B	
14	B	
15	B	
16	B	
17	B	A
18	B	
19	B	
20	B	
21	A	
22	C	
23	B	
24	D	
25	C	
26	E	
27	C	
28	B	
29	D	
30	E	
31	D	
32	C	
33	A	
34	E	
35	A	
36	B	

* INDICATES A MULTIPLE RESPONSE X INDICATES THE ITEM WAS OMITTED