Thursday, May 8, 2014

/*MIDTERM 1 REVIEW*/
CS61B Lecture 1

Object-Oriented Programming

Object: A repository of data
Class: Type of object
Method: procedure of function that operates on an object
ex: addItem: adds an item to any list type object

	Object Type	Primitive Types
Contains	reference	values
Defined	class definition	built into Java
Created	"new" keyword	"6", "3.4", "true"
Initialized	constructor	default "zero" value string: empty int: 0
Used	method	operators (+, *)

Inheritance: a class can inherit properties from a base/ parent class
ex: ShoppingList inherits properties from a List class, which has the property of storing a sequence of items
Polymorphism: One method works on several classes, even if the classes need different implementations (of that method)
Object-Oriented: Each object knows its own class and methods

Java

variables: you must declare them and their type
python: x = 1 scheme: (let ((x 1)))
Java:

```
int x;
x = 1;
```

int x does 2 things:
1. allocates memory to store integer, type "int"
2. gives the chunk of memory a variable name "x"

x [ ]

x = 1 ======> x [1]

variables are used to reference object
1. Use one defined by someone else
2. create your own

string myString myString [ ]

constructor: new String()
myString = new String()
SIGNATURE: String()
CALL: new String()
"=" is an assignment, causes myString to reference the string object constructed
myString [ ] ———— -> [empty string object]

Java programs must be compiled before you can run them

Java Program (.java)—>java_compiler—>".class" files—>run JVM on class files
JVM: java virtual machine

Objects and Constructors:

string s; //declare a string variable
s = new String(); //construct empty string, assign it to s
OR
string s = new String(); //both steps combined
s = "Yow!"; //old reference is replaced by new "Yow!"
if nothing else is pointing to old object, garbage collection will clean up the memory spot of the old object
string s2 = s //s2 now points at "Yow!" (same object that s points to)
s2 = new String(s) //Constructor makes copy of "YOW" (same word, diff obj)
s2 now points to a different object than s, though the objects are copies
1. Looks where s points
2. reads string object
3. constructs new string with copy of characters
4. makes s2 reference the new string just created

3 Constructors:

1. new String(): constructs empty string (zero characters)
2. "Yow!": specifies string (quotation construction)
3. new String(s): takes a parameter s, makes copy of object that s references

constructors always have same name as their class, except the quotation constructor

---

Methods

s2 = s.toUppercase();
s2 [ ]——>[YOW!]
string s3 = s2.concat("!!");
s3 = s2 + "!!";

Strings are <u>immutable,</u> once a string is made, only modified copies can be constructed, but you cannot change the original string (contents never change)

I/O classes and Objects

objects in System class for interacting with a user:
System.out is a PrintStream object that outputs to the screen
System.in is an InputStream object that reads from the keyboard
readLine is defined on BufferedReader objects
BufferedReader are constructed with an InputStreamReader
InputStreamReader is constructed with an InputStream such as System.in
(Look in JAVA API libraries— java.io)

InputSream objects reads raw data
InputStreamReader composes raw data into characters
    characters in java are 2 bytes long
BufferedReader compose characters into an entire line of text

EXAMPLE JAVA PROGRAM:

```
import java.io.*;
class SimpleIO {
    public static void main (String[] arg) throws Exception {
        BufferedReader keybd =
            new BufferedReader(new InputeStreamReader(System.in))
        System.out.println(keybd.readLine());
    }
}
```

CS61b Lecture 3

Defining Classes

Fields: variables stored in objects
    also known as INSTANCE VARIABLES
    does not have parentheses after
    ex:
        amanda.age (field/instance variable)
        amanda.introduce() (method call)

CLASS DEFINITION: (with field declarations and method definition)

```
class Human {
    public static int numberOfHumans;
    public int age;
    public String name;
    public void introduce() {
        System.out.println("I'm "+name+" and I'm "+age+" years old.");
    }
    public void copy(Human original) {
        age = original.age;
        name = original.name;
    }
    public Human(String givenName) {
    age = 6;
    name = givenName;
    }
    public Human() {
```

```
            numberOfHumans++;
        }
}
```

method definition knows that NAME refers to amanda.name

Constructing a Human Object:

```
Human amanda = new Human();
amanda.age  = 6;
amanda.name = “Amanda”;
amanda.introduce()
```

Using human.copy() Method: copy method require HUMAN type input

```
amanda.copy(rishi);
```

Class Constructor:

```
public Human(String givenName) {
        age = 6;
        name = givenName;
}
```

Using Constructor:

```
Human amanda = new Human(“Amanda”);
```

When no constructor is defined in a class, Java will automatically create a default constructor that takes no parameters. Does not initialize any fields (null, 0, etc.):

```
public Human() { }
```

Writing a new constructor will make the default constructor unusable; if you want to use it still you will need to explicitly write a new default constructor again (like above). Constructor methods need the NEW keyword in order to work, since they are not called on an object.

THIS keyword

amanda.introduce implicitly passes the object (amanda) as a parameter called “this”

```
public void change(int age) {
        String name = “Tom”;
        this.age = age;
        this.name = name;
}
```

must distinguish the field name this.age (amanda) from the local name “tom”, and the field age this.age from the local parameter age (parameter of change method)
IMPORTANT: you cannot change the value of “this”
this = amanda; ———> compile time error

STATIC keyword

a static field is a single variable charred by a whole class of objects; called a class variable

Lecture 1-29-2014 Number 4-5

Primitives

Byte: 8 bit integer -128 to 127
short : 16 bit
long: 64-bit
double: A 64-bit floating point number 18.3555
float: 32 bit
boolean: true or false
char: a character
int: 32 bit

long x = 43L;
double & float values must have a decimal point
double y = 18.0;
float f = 43.9f;
char c= ‘h’;

Object Types vs Primitive Types

number operators:
-x x*y x+y x/y x-y
integer operators:
x%y (remainder)
x/y truncating division (not true division)

java.lang library:
-Math class
x = Math.abs(y); x = Math.sqrt(y);
-Integer class
int x = Integer.parseInt(“1984”);
double d = Double.parseDouble(“3.14”);
Integers can be assigned to variables of longer types

```
int i = 43;
long l = 43;
l = i
i = l            //Compiler ERROR
i = (int) l;     //Cast will compile correctly
```

Boolean Values

“true” or “false”

a	b	a&&b	allb	!a
f	f	f	f	t
f	t	f	t	t
t	f	f	t	f
t	t	t	t	f

Boolean values can be created by comparison operators:

“==“ “<“ “>” “<=“ “>=“ “!=“

```
boolean x  = 3 == 5;          //x is false
x = 4.5 >= 4.5;          //true
x =  4 != 5 -1;          //false
x = false == (3 == 0);          //true
```

Conditional Statements

```
if (boolValue) {
        statements;
}
```

Boolean value must be in parentheses

```
boolean pass = score >= 75;
if (pass) {
        output(“You pass CS61b”)
} else {
        output(“You are such an unbelievable loser.”)
}
```

if-then-else clauses can be Nested and Daisy chained

find maximum of 3 numbers

```
        if (x>y) {
                if (x > z) {
                        max = x;
                } else {
                        max = z;
                }
        } else if (y > z) {
                max = y;
        } else {
                max = z;
        )
```

Switch Statements

```
switch (month) {
case 2:
        days = 28;
```

```
        break;
case 4:
case 6
case 9
case 11:
        days = 30;
        break;
default:
        days = 31;
        break;
}
```

Return Keyword

return causes a method to end immediately. Control returns to the calling method
Print numbers from 1 to x:

```
public static void oneToX(int x) {
        if (x < 1) {
                return;
        }
        oneToX(x-1)
        System.out.println(x);
}
```

While Loops

```
public static boolean isPrime(int n) {
        int divisor = 2;
        while (divisor < n) {
                if (n % divisor == 0) {
                        return false;
                }
                divisor ++;
        }
        return true;
}
```

“divisor < n” is called the loop condition. The loop will not be executed if the condition is false right away. The code inside the braces created by the loop condition is called the loop body.
Iteration: a pass through the loop body.

For Loop

```
for (initialize; condition; next) {
        statements;
}

EQUIVALENT TO WHILE LOOP
initialize;
```

```
while (condition) {
        statements;
        next;
}


public static boolean isPrimeFor (int n) {
        for (int divisor = 2; divisor<n; divisor++) {
                if (n % divisor == 0) {
                        return false;
                }
        }
        return true;
}
```

Arrays

an object storing a numbered list of variables
Each is a primitive type or reference

```
char[] c;
c = new char[4];
c[0] = 'b';
c[3] = 'e'
c[4] = 's';       ///RUNTIME ERROR
```

Field "c.length"
        cannot assign a value to length (compile time error)

Primes

Algorithm: Sieve of Eratosthenes
public static void printPrimes(int n)

Multi-dimensional Arrays

2D array: array of references to arrays

```
public static int[][] pascalTriangle(int n) {
        int[][] pt = new int[n][]
                for (int i = 0; i < n; i++) {
                        pt[i] = new int[i+1];
                        pt[i][0]=1;
                        for (int j = 1; j<i;j++) {
                                pt[i][j]=pt[i-1][j-1]+pt[i-1][j];
                        }
                        pt[i][i]= 1;
                }
                return pt;
        }
```

CS61b 6

Automatic Array Construction

```
int[][] table = new int[x][y];
```

table [ ]———>[x “rows”]——>[y “columns”]
x arrays of y ints, all in one array called table
Initializers:

```
Human[] b = {amand, rishi, new Human(“Paolo”)};
```

automatically constructs a 3 Human array

```
int[][] c = {{7,3,2},{x},{8,5,0,0},{y+z, 3}};
```

creates non rectangular array
CANNOT construct array with {} for a variable already declared

```
d = {1,2,3};    ///COMPILER ERROR
```

Multi-dimensional array declaration:

```
int[] a, b, c;    //all reference arrays
int a[], b, c[][];          //a is 1D; c is 2D; b is not array
int[] a, b[];     //a is 1D; b is 2D
```

Array of Objects

- when you construct array of objects, java does not construct objects
- Objects are null until initialized

Main() parameters

You can use command line arguments in terminal when running the main method for a class. The parameter for main() is a String[] array of args

```
class Echo {
        public static void main(String[] args)
                for (int i = 0; i < args.length; i++) {
                        System.out.println(args[i]);
                }
        }
}
```

unix prompt% java Echo kneel and worship Java
kneel
and
worship Java

Do Loop

do always executes loop body at least once

```
do {
        s = keybd.readline()
        process (s);
} while (s.length()>0);
```

useful for when you need to check a condition after a loop has gone through and initialized some variables

Break and Continue statements

break exits the inner most loop or “switch” enclosing the “break”

“TIME-AND-A-HALF” loop

```
s = keybd.readLine();
while (s.length() > 0) {
        process(s);
        s = keybd.readLine();
        }
OR
while (true) {
        s = keybd.readLine();
        if (s.length() == 0) {
                break;
        }
        process(s);
}
```

CONTINUE

1. only applies to loops
2. jumps to end of loop body but another iteration can happen if loop condition

for loop incrementer will still execute after a continue jump

Constants

“final” keyword used for values that can never change (a constant)

array x, x.length is a final field.

Lecture 7-8

List

Array-based lists:

- fast access to each item
- difficult to insert items at beginning or middle; takes time proportional to length of list

***FINISH IMPLEMENTATION AFTER LECTURE NOTES UPLOADED

```
public class List {
        int[] a;
        int lastItem;

        public List() {
                a = new int[10];
                lastItem = -1;
        }
        public void insertItem(int newItem, int location) {
                int i;
                if (lastItem+1 ==a.length) {
                        int[] b = new int[2*a.length];
```

```
for (i =0;          ;          ) {
}
```

Linked List

Recursive Data type
Made up of nodes. Each node has:
-an item (“car”)
-a reference to next node in list (“cdr”)

```
public class ListNode {
        //FIELDS
        public int item;
        public ListNode next;

        //CONSTRUCTORS
        public ListNode(int i, ListNode n) {
                item = i;
                next = n;
        }f
        public ListNode(int i) {
                this(i, null);
        }

        //METHODS
        public void insertAfter(int item) {
                next = new ListNode(item, next);  //////old next reference is given to new node
}


ListNode l1 = new ListNode, l2 = new ListNode(), l3 = new ListNode();
l1.item= 7;
l2.item=0;
l3.item=6;
l1.next = l2;
l2.next = l3;
l3.next = null;
OR
ListNode l1 = new ListNode(7, new ListNode(0, new ListNode(6)));

l2.insertAfter(3);
```

Inserting items into linked lists takes constant time, if you have a reference to a previous node (may not have constructed with references)
Linked Lists can keep growing until memory runs out.
Take O(n) time to find nth element in linked list

Lists of Objects

```
public class SListNode {
```

```
        public Object item;                    //any object, no primitives
        public SListNode next;
```

Insert new item at beginning:

```
        x = new SListNode("soap", x);
```

Null

```
public class SList {
        private SListNode head;
        private int size;
        public SList() {
                head = null;
                size = 0;
        }
        public void inserFront(Object item) {
                head = new SListNode(item, head);
                size++;
        }
}
```

Public and Private

private method or field: invisible and inaccessible to other classes

- prevent data corrupting by other classes
- objects of the same class can access private variables from other objects
- improve implementation without causing other classes to fail

Software Engineering

Interface: how other classes interact with a class

- prototypes for public methods, public fields
- plus description of their behaviors

Abstract Date type: a class with a well defined interface, but implementation details are hidden

Invariant: a fact about a data structure that is always true

- ex: date abstract data type always stores a valid date
- invariants are enforced by allowing access only through method calls

Not all classes are ADTS. some classes just store data, without invariants (fields can be public)

SLIST ADT

slist enforces 2 invariants:

1. size is always correct
2. list is never circularly linked

these invariants are enforced because only SList methods can change the list

1. Slist fields are private (head and size)
2. no method of SList returns an SListNode

Doubly Linked Lists

Slist inserting and deleting at front of list is easy, but end of list takes a long time

```
class DListNode {
        Object item;
        DListNode next;
        DListNode prev;
}
class Dlist {
        private DlistNode head;
        privated DListNode tail;
        long size;
}
```

Insert and delete items at both ends in constant running time

Remove the tail node (at least 2 items in Dlist):

```
tail.prev.next = null
tail = tail.prev
```

Sentinel: a special node that does not represent an item

- item is null
- previous points to tail
- next is head
- sentinel circularly links lists

DList invariants (with sentinel):

1. for any DList d, d.head != null
2. for any DListNode x, x.prev !=null
3. for any DListNode x, x.next != null
4. for any DListNode x, if x.next == y, then y.prev == x
5. if x.prev = y, then y.next == x
6. size field is correct, does not include sentinel, accessible from sentinel with next

Empty DList: Sentinel's prev and next fields point to itself

```
public void removeBack() {
        if (head.prev != head) {
                head.prev = head.prev.prev;
                head.prev.bext = head;
                size- -;
```

Lecture 2-10 Number 9

Reading Notes: pp. 77, 235–239, 258–265, 663

Java is pass-by-value, meaning that methods copy the value of arguments passed

Project 1 is due February 22

The Stack ad the Heap

The heap stores all object, including all arrays and all class variables
The stack stores all local variables, including parameters.
When the JVM calls a method, it puts the call in a stack frame (aka activation record) on top of the stack, with all its local variables and parameters stored
JVM keeps an internal stack of stack frames. The top of the stack is whatever is executing at that instant in the program.

WHEN A METHOD FINISHES, ITS STACK FRAME IS ERASED. You must store internal values as an object if you want it saved.

Method "Thread.dumpStack()" useful for debugging (ie finding illegal parameter passing)

Parameter Passing

Like in scheme, Java passes all parameters by value: meaning that arguments are copied into the formal parameters, originals are not changed. You get a copy in the stack frame. Changing it won't change the original. See IntBox{}

When the parameter is a reference, the reference is copied, but the object is shared.

Binary Search (and Recursive Stack Frames)

When a method calls itself, JVM creates multiple stack frames, but only the top stack frame is accessible.
Binary Search searches a sorted array. ONLY WORKS ON SORTED ARRAY
Recursive base cases:

1. search element is the middle element, return index
2. subarray of length zero: return Failure

```
public static final int FAILURE = -1;
private static int bsearch(int[] i, int left, int right, int findMe) {
        if (left > right) {
                return FAILURE;
        }
        int mid = (left + right)/2;
        if (findMe == i[mid]) {
                return mid;
        } else if (findMe < i[mid]) {
                return bsearch(i, left, mid - 1, findMe);
        } else {
                return bsearch(i, mid+1, right, findMe);
        }
}
public static int bsearch(int[] i, int findMe) {
```

```
        bsearch(i, 0, i.length -1, findMe);
}
```

WRAPPER FUNCTION: a method that calls another method that makes it easier to use that second method with a simpler interface. bsearch(int[], int) is a wrapper method of bsearch(int[], int, int, int)

Takes log2n bsearch calls to reduce the problem down to one base case.
Logarithms: 4.1.2 and 4.1.7 of GT

Java has enough stack space for a few thousand stack frames.

Lecture 10 and 11

Inheritance

```
public class TaiList extends SList {
//“head” and “size” inherited from SList
private SListNode tail;
}
```

TaiList is a subclass of SList. SList is the superclass of TailList.
A subclass can modify a superclass in 3 ways:
1. It can declare new fields.
2. Declare new methods.
3. override old methods with new implementations

```
public void insertEnd(Object obj){
        ///solution to lab3
    SListNode node = new SListNode(obj);
    if (head == null) {
      head = node;
      tail = node;
    } else {
      tail.next = node;
      tail = node;
      }
    size++;
}
```

isEmpty(), length(), nth(), toString() are inherited from SList.

Inheritance and Constructors

Java executes TaiList constructor, which calls SList() constructor before it executes any code.

```
public TailList() {
        // SList() sets size = 0, head = null
        tail = null;
```

```
}
public TailList(int x) {
        super(x);      ///Must be first statement in constructor
        tail = null;
}
```

Invoking Overridden Methods

```
public void insertFront(Object obj) {
        super.insertFront(obj);
        if (size == 1) {
                tail = head;
        }
}
```

"Protected" Keyword

protected variables let subclasses see superclass variables. A proceed field is visible to declaring class and all its subclasses, but no others. (Private fields are not visible to subclasses). Protected fields are READ and WRITE for subclasses.

```
public class SList{
        protected SListNode head;
        protected int size;
}
```

Class Hierarchies

Object:

1. Primitives: string, etc.
2. User Defined Classes: Worker
    - i. Proletariat
        - a. Student
        - b. TA
    - ii. Bourgeoisie
        - a. Professor

Dynamic Method Lookup

Every TailList is an SList

```
SList s = new TailList();          //works fine
TailList t = new SList();          //COMPILE TIME ERROR
```

Static type: type of a variable

Dynamic type: the class of the object that the variable references.

When we invoked overridden method, Java calls method for object's dynamic type, regardless of static type.

```
SList s = new TailList();
s.insertEnd(obj);      ///Calls TailList.insertEnd()
s = new SList();
s.insertEnd(obj);      ////Calls SList.insertEnd()
```

Why dynamic method lookup matters?

Method that sorts an SList using only SList method calls now sorts TailList TailLists too.

Subtleties of Inheritance

1. New method in TailList - eatTail()

```
TailList t = new TailList();
t.eatTail();
Sist s = new TailList();
s.eatTail();              //Compiler Error
```

Cannot compile because not every SList has an “eatTail()” method. Java can’t use dynamic method lookup on s.

2.

```
SList s;
TailList t = new TailList();
s = t;
t = s;            ///Compile time error. SList cannot be stored in the subclass TailList
t= (TailList) s;          //Cast will work
s = new SList();
t = (TailList) s;         ///RUNTIME ERROR: ClassCastException

int x = t.nth(1).intValue();    ///Compile-time error. Not ever object has an intValue()
int y =((integer)  t.nth(1)).intValue();        //casts nth object as an integer.
```

3. “instanceof” operator. Operators are in lower case. This operator tells you whether the object is of a specific class

```
if (s instanceof TailList) {
        t = (tailList) s;
}
```

false is s is null or doesn’t reference a tail list. true if s is a reference to a tai list or a subclass of tail list.

Equals()

Every class has an equals() method. It is a method of the object class, so unless the equals method is user defined, it will be inherited from Object.

Default: inherit Object.equals()
r1.equals(r2) /// same as r1==r2
runtime error if r1 is null

Many classes override equals() to compare content.

```
int i1 = 7;
int i2 = 7
(i1 == i2)              ///FALSE
i1.equals(i2)           ///returns TRUE
i3 =  i2;
(i3 == i2)              ///True
i3.equals(i2)           //True
```

Four degrees of equality:

1. Reference equality, == (do they reference the same object)
2. Shallow structural equality: two objects are .equals() when all of their fields are ==
3. Deep structural equality: two objects are .equals() if their fields are .equals()
4. Logical Equality. Ex:
    a. “Set” objects are .equals if they contain the same elements (even in different order)
    b. Fractions 1/3 and 2/6 are .equals()

equals() may test any of these 3 levels.

Deep Structural Equality for SLists (Iterative Algorithm)

Signature must be the same as Object.equals(). Don’t make parameter SList!

```
public class SList {
        public boolean equals (Object other) {
                if (!(other instanceof Slist)){
                        return false;
                }
                SList o = (SList) other;
                if (size != o.size) {
                        return false;
                }
                SListNode n1 = head;
                SListNode n2 = o.head;
                while (n1 != null){
                        if (!n1.item.equals(n2.item)) {
                                return false;
                        }
                        n1 = n1.next;
                        n2 = n2.next;
                }
                return true;
        }
}
```

Only works when SList invariants are met.

For Each Loops

Iterates through array elements.

```
int[] array = {7,12,3,8,4,9}
for (int i : array) {
        System.out.print(i+” “);
```

i takes on the values of the array

Testing

1. Modular Testing- test each method separately.
    Test drivers and stubs:
    a. Test driver calls the code, check results. Must be inside the class where the methods are declared. This way, test driver can check private methods.
    b. Stubs: bits of code that are called by the code being tested.

2. Integration testing: testing all components together.
    -Define interfaces well
    - Learn to use a debugger
3. Result Verification
    a. Data structure integrity checkers.
        Inspects a data structure and verifies that all invariants are satisfied
    b. Algorithm result checkers (i.e. sort verification)

assertion: code that tests an invariant or a result. Boolean value

```
assert x==3;
assert list.size ==list.countLength():
    “wrong SList size: “+list.size;
```

To turn on assertions:
    java -ea
Turn off: (better speed)
    java -da
    (countLength is never called)

Lecture 12 & 13

Abstract Classes

```
public abstract class List{
    protected int size;
    public int length() {
        return size;
    }
    public abstract void insertFront(Object item);
}
```

abstract method lacks an implementation.

```
public class SList extends List {
    //inherits “size”
    protected SListNode head;
    //inherits “length()”
    public void insertFront(Object item) {
        head  = new SListNode(item, head);
        size++;
    }
}
```

A non-abstract class may never contain an abstract method or inherit one without providing an implementation.

```
List myList = new SList():
myList.insertFront(obj);
```

Abstract classes provide a reliable interface:
multiple classes can share
without defining any of them yet

```
public void listSort(List L) {…}
```

List L can be any subclass of List: SList, DList, TailList, TimedList, etc….
TimedList: records time spent on operations
TransactionList: logs all changes on a disk

Java Interfaces

interface: public fields, method prototypes and behaviors.
java interface: interface keyword

Java interface is like an abstract class. 2 differences:

1. A class can inherit from only one class. You can implement (inherit) many java interfaces.
2. A java interface cannot implement any methods or include and fields except "final static" constants.

```
public interface Nukable {                    /// in Nukeable.java
        public void nuke();                   /// java assumes the method is abstract
}
```

java interface only takes abstract methods

```
public interface Comparable {          // in java.lang
        public int compareTo(Object o);
}

public class SList extends List implements Nukeable, Comparable {
        public void nuke() {
                head = null;
                size = 0;
        }
        public int compareTo(Object o) {
                [Returns a number < 0 if this < object o. Returns 0 if this.equals(o). Returns # > 0
                if this > object o.
        }
}

Nukeable n = new SList();
Comparable c = (Comparable) n;
```

Arrays class in java.util sorts arrays of Comparable objects

```
public static void sort(Object[] a)
```

A subinterface can have multiple superinterfaces.

```
public interface NukeAndCompare extends Nukeable, Comparable {}
```

WHEN A SUBINTERFACE uses superinterfaces, use the **extends** keyword. When classes use any kind of interface, they use **implements**

Java Packages (Not on midterm 1, useful for hw and projects)

packages: a collection of classes and Java interfaces and sub packages

1. packages can contain hidden classes not visible outside package
2. classes can have fields and methods visible inside package only
3. Different packages can have classes with same name.

Examples of packages:

1. java.io
2. Homework 4 uses list package containing

Using Packages: use a fully qualified name

```
java.lang.System.out.println(" ");
```

```
import java.io.*                    // imports everything in java.io package
```

every program imports java.lang.*

package is a protection level, provided by default if not declared private or protected.

A class or variable with package protection is visible to any class in the same package, but not outside (outside the directory where the package is stored.

Homework 4: DListNode class is public (not package protected). We want applications to hold DLists, but not change them. "prev" and "next" will still have package protection;

Public class must be declared in file named after class. "package" classes can appear in any .java file.

Compiling and running must be done from outside pacakage.

```
javac -g list/SList.java
java list.Slist
```

Iterators

```
public interface Iterator{
    boolean hasNext();
    Object next();
    void remove();
}
```

An iterator is like a bookmark. Can have many iterators in the same data structure. Calling next() nth time returns nth item in sequence. Subsequent calls to next() throw an exception.

hasNext(): true iff more items to return in the list.

```
public interface Iterable {
        Iterator iterator();
}
```

DS.iterate() constructs a DSIterator for DS

"for each" loop iterates over item in data structure.

```
        for (Object o : l ) {
                System.out.println(o);
        }

        equivalent to:
        for (Iterator i = l.iterator()
```

Midterm 2 Lecture Notes
Lecture 15-16

Exceptions

Run-time error: Java "throws an exception" (Exception object). Prevent the error by "catching" the Exception

1. Coping with errors

```
                try {
                        f = new FileInputStream("~cs61b/pj2.solution");
                        i = f.read();
                }
                catch (FileNotFoundException e1) {
                        System.out.println(e1);                 ///Exceptions have toString()
                }
                catch (IOException e2) {
                        f.close();
                }
```

- Executes the code inside the try clause
- If try code executes normally, skip catch clauses
- If try code throws an exception, do not finish try code. Jump to first catch clause that matches exception; execute the catch body. Any code that throws an exception will throw a specific exception class type that the catch clauses look for

FileNotFoundException & IOException are subclasses of exception

2. Escaping a Sinking Ship—throw your own exception;

```
public class ParserException extends Exception {}
public ParseTree parseExpression() throws ParserException{
        if (somethingWrong) {
                throw new ParserException();
        }
```

Need to declare what type of exception the class throws.
Exception vs return: Exception does not let you return anything. An exception can propagate down the stack through many stack frames. Will propagate until something catches it, or until the end of main in which case there is an error.

```
public ParseTree parse() throws ParserException, DumbCodeException {
        p = parseExpressoin();

public void compile() {
        ParseTree p;
        try {
                p = parse();
                p.toByteCode();
        } catch (ParserException e1){
        } catch (DumbCodeException){}
```

Checked and Unchecked Throwables

throwable has two subclasses: exceptions and errors
Exceptions: IOException, RunTimeException(NullPointerException, ClassCastExceptions), etc.
Error: VirtualMachineError, OutOfMemoryError,AssertionError
        Errors are things you should not try to catch. You need to be able to fix errors immediately, because they will most likely crash the program
Unchecked: errors and exceptions that you don't have to declare, they will always be thrown when they come up. Almost any method can throw a NullPointer, ClassCast, etc.
Checked Method: a throwable that you must declare yourself.

"Finally" Keyword

```
f = new FIleInputStream("filename");
try {
        statement X;
        return 1;
} catch (IOException e) {
        e.printStackTrace();
        return 2;
} finally {
        f.close();
}
```

If "try" statement begins, the "finally" clause will execute at the end, NO MATTER WHAT.
If statement X does not cause an exception. So try returns 1. But the return statement does not stop the finally clause; it executes the finally clause and then the entire method returns the value of 1.
If statement X throws an IOException, then the catch clause is executed. Then the finally clause is executed, and then 2 is returned.
If statement X causes some other exception, first the finally clause executes, then exception continues down the stack. The finally clause delays the exception but does not stop it from being thrown and propagating down the stack

Return in the finally clause? the entire return value of the method is replaced by the finally return
Exception thrown in “catch” clause: terminate the “catch” clause, execute “finally”, exception continues down stack.
Exception thrown in “finally” clause: replaces old exception, “finally” clause & method end.
Inside a catch clause or final clause, you can have another nested pair of try and catch statements.

Exception constructors

Convention: most throwables have 2 constructors.

```
class MyException extends Exception {
        public MyException() { super();}
        public MyException(String s) { super(s); }’
```

Generics

declare general classes (class that can store any type of object) that produce specialized objects.
SList for only strings, SList for only integers, but only one SList class
SList takes a type parameter.

```
class SListNode<T> {
        T item;
        SListNode<T> next;
        SListNode(T i, SListNode<T> n) {
                item = i;
                next  = n;
        }
}
public class SList<T> {
        SListNode<T> head;
        public void insertFront(T item) {
                head = new SListNode<T>(item, head);
        }
}
```

Create an SList of Strings:

```
SList<String> l = new SList<String>();
l.insertFront(“Hello”);
```

Advantages: compiler ensure at compile time that nothing but strings enter your SList<String>. Makes it easier to debug than a class casat exception later if you try to use an item in a list that isn't the same as all the other items(say someone accidentally put an integer into a list only mean for strings without you knowing….hmm….)

Filed Shadowing

fields can be shadowed in subclasses. This is different from overriding. Choice of methods dictated by dynamic type. Choice of fields dictated by static type

```
class Super {
        int x = 2;
```

```
        int f() {
                return 2;
        }
}
class sub extends Super {
        int x = 4;              ///Shadows Super.x
        int f(){                /// overrides Super.f()
                return 4;
        }
}
Sub sub new Sub()
super supe sub;
int i;
i = supe.x;             //2
i = sub.x       ///4
i = ((super) sub).x             //2
i = ((sub) supe).x              // 4
```

Lecture 17

Game Tree Search

Alpha-Beta Pruning

```
public class Grid {
public Best chooseMove(boolean side) {
Best myBest = new Best(); // My best move
Best reply; // Opponent's best reply
if ("this" Grid is full or has a win) {
return a Best with Grid's score, no move;
}
if (side == COMPUTER) {
myBest.score = -1;
} else {
myBest.score = 1;
}
myBest.move = any legal move;
for (each legal move m) {
perform move m; // Modifies "this" Grid
reply = chooseMove(! side);
undo move m; // Restores "this" Grid
if ((side == COMPUTER &&
reply.score > myBest.score) ll
(side == HUMAN &&
reply.score < myBest.score)) {
myBest.move = m;
myBest.score = reply.score;
}
}
return myBest;
```

```
}
}
```

To turn this insight into an algorithm, we pass two additional parameters to the chooseMove() method: α and β. The pa- rameter α is a score that the computer knows with certainty it can achieve; for instance, if α = 0, then the computer knows it can force a draw, and is only interested in searching for moves guaran- teed to do better. Conversely, β is a guarantee that the opponent can achieve a score of β or lower. For any grid, we maintain values of α and β based on our current knowledge of the best moves dis- covered thus far. If β becomes equal to or less than α, then further investigation of the current grid is useless.

```
public Best chooseMove(boolean side,
int alpha, int beta) {
Best myBest = new Best(); // My best move
Best reply; // Opponent's best reply
if ("this" Grid is full or has a win) {
return a Best with the Grid's score, no move;
}
if (side == COMPUTER) {
myBest.score = alpha;
} else {
myBest.score = beta;
}
myBest.move = any legal move;
for (each legal move m) {
perform move m; // Modifies "this" Grid
reply = chooseMove(! side, alpha, beta);
undo move m; // Restores "this" Grid
if (side == COMPUTER &&
reply.score > myBest.score) {
myBest.move = m;
myBest.score = reply.score;
alpha = reply.score;
} else if (side == HUMAN &&
reply.score < myBest.score) {
myBest.move = m;
myBest.score = reply.score;
beta = reply.score;
}
if (alpha >= beta) { return myBest; }
}
return myBest;
}
```

Lectures 18-19: encapsulation and encapsulated lists, see homework and project

Lecture 20

Asymptotic Analysis

Inventory
- 10,000ms to read inventory from disk
- 10ms to process each transaction
- n transactions takes (10,000+10n) ms
  - n plays a bigger part in the computation time when n is large

Big O Notation

puts an upper bound on computation time.
n is the size of the program's input.
    n can be word size, list length, bit size
Let T(n) be a function that models the running time in ms, or the memory allocation.
Let f(n) to be another function similar to T(n), but is preferably much simpler.
T(n) is O(f(n)) if and only if T(n) <= cf(n) whenever n is big and for any constant c
How big is n? c?
    c and n can be any constants (as big as possible) as long as T(n) is still less than cf(n).
    c cannot change as n changes. We can pick c and n arbitrarily.
EX:
    T(n) = 10,000 + 10 n (actual running time)
    consider f(n) = n
    pick c = 20
    T(n) and cf(n) intersect at n=1000. For n>= 1000, T(n)<=cf(n)
    Therefore T(n) is in O(f(n))=O(n)

Formal Definition of Big O Notation: O(f(n)) is the **set** of all function T(n) that satisfy: there exists positive constants **c** and **N** such that for all values n>= N, T(n)<=cf(n)
    In the above example, N=1000.
O(n) is a subset of O(n^2)
T(n)=1000000n is in O(n) with c = 1000000 and N = 0
O(2n) = O(n) unnecessary but not wrong
n^3 + n^2 + n is order O(n^3)
Proof: c = 3, N = 1
Big O notation only shows dominant polynomial term

Important Big O Sets

these sets are most often used in algorithms/data structure analysis. From smallest to largest (slowest to fastest growing).

FUNCTION	NAME
O(1)	constant
O(logn)	logarithmic
O(log^2(n))	log squared
O(n^1/2)	root n
O(n)	linear
O(nlogn)	nlogn
O(n^2)	quadratic

FUNCTION	NAME
O(2^n)	exponential base 2
O(e^n)	exponential

Log to any power is still slower growing than root n, or any n to a constant power.
Algorithms running in O(nlogn) time or faster are considered efficient
n^7 time or more is useless
logn (base 2) is usually smaller than 50 in practice

Lecture 21 and 22

Dictionaries

Two-letter words and defintions.
Word is the key that address the definition.
26*26 = 676 possible words
To insert a definition into a dictionary:
function hasCode() maps each word (key) to integer 0…675

```
public class Word{
	public static final int LETTERS = 26;
	public static final int WORDS = LETTERS*LETTERS;
	private String word;
	public int hashCode() {
		return LETTERS*(word.charAt(0) - ‘a’) + (word.charAt(1) - ‘a’);
	}
}
public class WordDictionary {
	private Definition[] defTable = new Definition[Word.WORDS];
	public void insert(Word w, Definition) {
		defTable[w.hashCode()] = Definition;
	public Definition findWord(Word w)
		return defTablep[w.hashCode()];
}
```

Hash Tables (most common implementation of dictionaries)
n: number of keys/words that we want to store in the dictionary~400k in English
Table of N buckets, where N is a slightly larger than n
Hash tables maps possible keys into N buckets by applying a compression function to each has code.
h(hashCode) = hasCode mod N
two different words can map to the same bucket— this is called a “collision”
Each bucket references a linked list of definitions, called a **chain**
how do you know which definition corresponds to which word?
each list node stores a word and its definition side by side

Hash Table operations (Goodrich and Tamasia)

```
public Entry insert(key, value) {		//reference to a word and definition
	compute keys hashcode
```

compress to determine bucket
insert entry into bucket's list
}
Hash the key: compute key's hashcode and sort into a bucket
public Entry find(key)
hash the key
search list for entry with given key
If found, return entry
public Entry remove(key)
find the key, then remove its
project 2 idea: key is gameboard, definition is score
will decrease computation time
2 entries with same key?
1. G&T: insert both— you have 2 words with the same key but difference definitions
find should return either definition, arbitrarily
remove should remove either arbitrarily
findAll can return all definitions of a key, not just one
2. inserting a key that already exists replaces the old definition with the new one. Only one entry with the given key exists.
load factor of a hash table is n/N
if the load factor of the hash table is low (less than 1)—and hash code and compression are good enough to avoid extraneous collisions and duplicate keys—then the chains are short, so each operation takes constant time.
If the load factor is large, ie n>>N, then the hash table takes O(n) time.

Hash codes and compression functions

key ————> hash code ————> [0,N-1]
ideally, map each key to a random bucket
Bad compression function:
suppose keys are ints
hashCode(i) = i;
compression function h(hashCode) = hashCode mod N
N = 10000
suppose keys are only divisible by 4 for some reason—we only use a quarter of the buckets
mod N where N is prime is a good compression function
better:
h(hashCode) = ((a*hashCode + b) modp) modN
a,b, p : positive integers
p: large prime, p>>N
now number of buckets N does not need to be prime
Ex: Hash code for strings:

```
private static int hashCode(String key) {
    int hashVal = 0;
    for (int i = 0; i < key.length(); i++) {
        hashVal = (127*hashVal + key.charAt(i))%16908799;
```

```
                    ///kind of like a base 127 number, but with mod for large #s
                    ///16908799 large and prime
            }
            return hashVal;
        }
```

multipier and modulus number must not have any common factors

Resizing Hash Tables

If load factor n/N gets too big, we lose constant time computation

- enlarge has table when load factor > = about .75
- allocate new array at least twice as long
- walk through entries in old array, rehash them into new

we can also shrink a hash table when the load factor is very small in order to free memory, although this will speed up our computation time. Extending a hash table tends to be more important than shrinking one.

Therefore, the hash table should know how many buckets it has, how many entries it has at anyone time, and be able to calculate the load factor in order to know when it needs to expand.

Average running time over long run is still O(1) per operation.

Transposition tables: speed game trees

some grids are reachable through many different sequences of moves. Minimax algorithm means you need to evaluate the same game board many times, which can be time costly. We can create a hash table of previously stored grids.

Stacks

sample application: verifying matched parentheses

- {[(){[]}](()}

algorithm: scan through string, char by char

- left side parenthesis: push it onto stack
- right sight parenthesis: pop its counterpart from stack, check that they match
- If parentheses are not matched, mismatch or null returned or stack not empty at end of string

---

Queues

- “enqueue” item at back of queue
- “dequeue” item at front of queue
- examine “front” item

(a,b) ——> dequeue() ——> (b) ——> enqueue(c) ——> (b,c) ——front()——> b

sample application: printer queue

all methods run in constant time

implemented as a singly-linked list with a TAIL pointer, in order to constant time remove and add queue items to end

DEQUE (pronounced DECK)

- Double-Ended- QUEue
- you can insert and remove items at both ends by using a doubly linked list.

Lecture 23

Asymptotic Analysis

Ω(f(n) is the set of all function f(n) such that there exist positive constants d and N:
    for all n>= N, T(n)>= d*f(n)
omega is the revers of big oh. If T is an element of O(f(n)) then f is an element of Ω(T(n))
    2n is an element of Ω(n) because n is an element of O(2n)
    n^2 is an element of Ω(n) because n is and element of O(n^2)
    n^2 is an element of 3n^2 + nlogn because 3n^2+nlogn is an element of O(n^2)
Omega says your program is at least Ω(n) bad
Big theta precisely specifies functions asymptotic behavior.
ø(f(n)) is the set of al functions tat are in both of O(f(n)) and Ω(f(n)).
    —–> ø(f(n)) = O(f(n)) ∧ Ω(f(n))
if f(n) is an element of ø(g(n)), then g(n) is in ø(f(n))
ex:
f(n) = n(1+sin(n))
f(n) is in O(n)
f(n) is in Ω(0)
f(n) is in ø(n(1+sin(n)))
    —> f(n) has no simple big theta

Algorithm Analysis

Problem 1: given p points, find pair closest to each other
Algorithm 1: calculate distance between each pair, return minimum
    there are p(p-1)/2 pairs
    each pair takes constant time to examine. Running time is ø(p^2)

Problem 2: smoosh array
algorithm 2:
    n is length of array
    2 indices, i and j
    i iterates up to n times
    ø(n)

Functions of several variables

Problem3: matchmaking program for w women and m men
algorithm3: compare compatibility of each possible couple
        compatibility computation is constant
Running time T(w,m) = ø(wm)
    there exists constants c and d, and M and N such that
        dwm <= T(w,m) <= cwm
    for every w>= W and m>= M
T is not theta or omega order w^2 or m^2
These possibilities preluded by w>>>m or m >> w
You can't asymptotically compare wm, w^2, m^2

Lecture 24-25

Trees

Tree: a set of nodes, where any two nodes have exactly **one** path between them

Path: A sequence of one or more nodes, each consecutive pair connected by an edge
Rooted Tree: one distinguished node is a **root**. Every node c, except the root, has one **parent** p, the first on the path from c to the root.
If c is p's child, then p is c's parent for any two nodes
Root has no parent
a node can have any number of children but only one parent
Leaf: a node with no children
Internal node: non-leaf node
Sibling Nodes: have the same parent
Ancestor: the ancestors of d are all the nodes on the path from d to the root, including d itself and the root
Depth: the depth of the node is the length of the path from n to the root. Depth of the root is zero.
Height: length of path from node n to its lowest depth leaf. Height of a leaf is zero
Heigh of a tree is just the height of the root
Subtree rooted at n: tree formed by n and its descendants
Binary Tree: no node has more than 2 children, and every child is either a left child or a right child, even if it is the only child.

Rooted Tree

Each node stores 3 references:
- item
- parent
- children—stored in a list

OR

sibling tree nodes are directly linked

```
class SibTreeNode {
	Object item;
	SibTreeNode parent;
	SibTreeNode firstChild;
	SibTreeNodenextSibling;
}

class SibTree {
	SibTreeNode root;
	int size;
}
```

Tree Traversals

Traversal: a manner of visiting each node in the tree once
Preorder traversal: visit each node before recursively visiting its children left to right. Starting at the root

```
class SibTreeNode {
	public void preorder() {
		this.visit();
		if (firstChild != null) {
			firstchild.preorder();
		}
		if (nextSibling != null) {
			nextSibling.preorder(){
		}
```

```
        }
}
```

each node visited/preordered only once, so the pre order takes O(n) time

Lecture 26

Binary search trees

every node has a parent, item, left child, and right child

Ordered dictionary: keys have a total order, like in a heap

quickly find entry with min or max key

Binary Search Tree invariant:

for any node x, every entry in x's left child tree has to be less than x. Every entry in x's right child tree has to be greater than x

and in order traversal visits every node of the BST in order of value, from least to greatest.

```
Entry find(Object k) {
        BinaryTreeNode node = root;
        while (node != null) {
                int comp = ((COmparable) k).
                        compareTo(node.entry.key());
                if (comp < 0) {
                        node = node.left;
                } else if (comp > 0) {
                        node = node.right;
                } else {
                        return node.entry;
                }
        }
}
```

```
Entry remove(Object k);
```

find a node n with key k

return null is k is not in tree

if n has no children, detach it from its parents

if n has one child, move n's child to to take n's place

if n has 2 children, let x be node in n's right subtree with smallest key

remove x. x has no left child so it is easily removed

replace n's entry with x's entry

Running Times

a perfectly balanced binary tree with height h has # of nodes n = 2^(h+1)-1

No node has depth > log_2(n)

Running times of all ops are proportional to the depth of the last node encountered.

O(logn) time on balanced tree

all operations on binary search trees have theta(n) worst case time.

if you make a tree like 1-2-3-4-5-6-7 this will be O(n)

Lecture 27-28

2-3-4 Trees:

Every node has 2 3 or 4 children, except leaves, which are all at the bottom level.
Every node stores 1,2 , or 3 entries. the number of children is entries +1 or zero

1. Entry find(Object k)
   like the normal binary tree search find

insert and remove are more complicated, because the restructure the tree

2. Entry insert(Object k, Object v) { //key and value
   walks down tree in search of k
   If it find k, it proceeds to k's "left child" and continues
   Whenever insert() encounters a 3 key node, middle key is placed in parent node. Parent has at most 2 keys and always has room for a third.
3. Entry remove(Object k);
   Find key k
   If it's in a leaf, remove the key from the leaf
   If its in internal node, replace it with entry with next higher key
   remove() changes nodes as it walks down
   Eliminates 1 key nodes except so keys can be removed from a leaf without leaving it empty

A 2-3-4 tree with height h has between s^h and 4^h leaves
The total number of entries n>= 2^(h+1) -1
   h is in O(logn)
Time spent visiting node is constant time
All find insert and remove has O(logn) time worst case. This is versus a binary search tree that has O(n) time for all methods

---

GRAPHS

A graph G is a set V of vertices and a set E of edges that connect vertices.
Multiple copies of an edge are forbidden. Edges are unique.
A directed graph can have two edge connecting the same two vertices, because the edges can have opposite directions and therefore are not the same edge. (v,w) vs (w,v)
Self edges are also possible: (v,v)
Path: Sequence of vertices with each adjacent pair connected by an edge. If graph is directed, edges must be aligned with direction of path.
Length: of a path is the number of edges in the path
So a path between vertices in <4,5,6,3> has a length of 3
<2>: path of length 0
<2,2>: length 1
<2,2,2>: length 2
Strong connected: there is a path from every vertex to every other vertex. Usually applied to directed graphs. Otherwise a unidirectional graph is just called connected.
Degree of vertex: number of edges incident on vertex. Self edges count just once
For directed graphs, we talk about in degree and out degree

GRAPH REPRESENTATIONS

Adjacency matrix: |V| by |V| array of booleans (number of vertices by number of vertices). Maximum number of edges is |V|^2
Planer graphs have O(v) edges
graph is sparse if it has far fewer edges than maximum
Adjacency list:

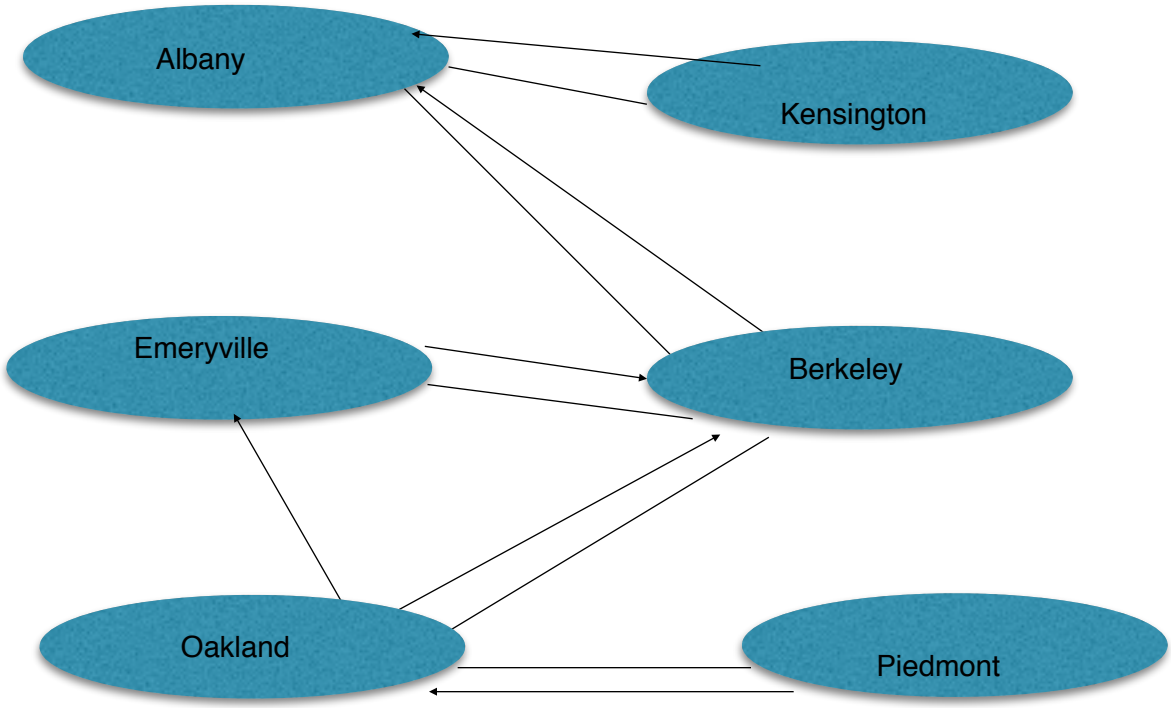
Each vertex v has a list of edges out
1: 4
2:1
3: 2,6
4: 5
5: 2, 6
6: 3
Memory used is THETA(|V|+|E|)
If vertices are labeled by strings, you can use a hash table

GRAPH TRAVERSAL
A way of visiting every vertex once. Depth first search or Breath first search
DFS: searches a graph as deeply as possible as soon as possible. If graph is a tree, the DFS performs pre order traversals
BFS: visits all vertices whose distance from starting vertex is one, then two, and so on. If graph is tree, BFS performs level order traversal

Lecture 29

```
public void BFS(Vertex u) {
        u.visit(null);
        u.visited = true;
        q = new Queue(u);
        q.enqueue(u);
        while (q is not empty) {
                v= q.dequeue();
                for( each vertex w such that (v,w) is an edge in E) {
                        if (!w.visited) {
                                w.visit(v);
                                w.visited = true;
                                q.enqueue(w);
                        }
                }
        }
}
public class Vertex {
        protected Vertex parent;
        protected in depth;
        protected boolean visited;
        public void visit(Vertex origin) {
                this.parent = origin;
                if (origin == null) {
                        this.depth = 0;
                } else {
                        this.depth - origin.depth + 1;
                }
        }
}
```

when edge (v,w) is traversed and we discover w is unvisited, we visit w, the depth ow w id the depth of v + 1, and v becomes parent of w



A: 0
K: 1
B: 1
E: 2
O: 2
P: 3
Shortest distant from Piedmont to Albany is P to O, O to B, B to A. Length of shortest path is just the depth of P, which is just 3. The shortest path can be found by following parent pointers
BFS runs in O(|V| + |E|) with adjacency list
For adjacency matrix, takes O(v^2)

Weighted Graphs

Each edge labeled with numerical number which is usually called a weight (but might be called a cost, slightly different meaning). Could be the capacity of the route(ie traffic, pipes, etc).
Adjacency matrix: array of ints or doubles.

Source are rows
columns are destination

For digraph, you need 2 adjacency matrices

Adjacency list: list of lists, where the list nodes has a weight field.

Problems

Shortest path: just like a google maps problem, for example, berkeley to LA

each edge is labeled with a rough approximation for the time, which is the weight

the shortest path is the path with the sum of the edge times is the shortest
BFS solves this is all edge weights are 1
BFS would not work on a map, because different edges take different time
CS 170: non uniform edge weights

Minimum spanning tree:

for example, making the least amount of wiring to outlets from electrical source, by chaining/ series. Each vertex is outlet or source of electricity. We use a graph that shows all the ways we could possibly connect all the outlets to the source. Edges are labeled with the length of the wire, which is proportional to its cost ($). Connect all nodes with the shortest length of wire is the goal.

Kruskal's Algorithm

G = (V, E) undirect graph
A spanning tree T = (v,F) of G is a graph with same vertices as G, and lvl-1 edges of that form a tree
If G is weighted, we want the minimum spanning tree, which is the tree with the least sum of edge weights. There could be more than one minimum, but they have the same total weight, so we use first one.
Create a new graph T with same vertices as G, but has no edges to being with.
We add edges one by one until we have a tree
We need to decided which ones to add and which ones not to add
Make a list of all edges in G, and sort by weight. Just reorder the adjacency list into one linked list with ordered weights. First make the list the sort from least to greatest.
Yes if adding it makes the graph a tree/forest still, no if it makes a loop. Iterate through the edges in sorted order from smalled to biggest. For each edge (u,w): check: if u and w are not connected by a path in T, then add the edge because it is the least edge that can connect them. If they are connected, don't add the edge to T.
T is always a tree (if G is connected). If G is not connected, then T is going to be a forest. So T will never have a loop.

Proof of G&T on page 649

Lecture 30-31- Midterm on Monday

Sorting

Insertion Sort:

O(n^2) time
Insertion enforces sorted order invariant, only inserts in correct place
Start with empty list s and unsorted list I of n items.

```
for (each item  x in I) {
        insert x int S in sorted order
}
```

Linked list insertion: Theta(n) worst case time to find right position
Array insertion: theta(n) worst case time to shift items over
Run time proportional to the number of inversions, plus n
Inversion in an array is any pair of numbers out of order. 19 7 5 has 3.

If S is array, insertion sort is an in place sort
If S is balanced search tree, the running time is O(nlogn)
Selection Sort
runs in quadratic time (worst case and best case are the same). Invariant: sorted list S
Start with empty list S and unsorted list of I of n items.
for (i = 0; i<n; i++){
x = item in I with smallest key
remove x from I
Append x to end of S
}
Whether S is array or linked list, Theta(n^2) time, even in best case
Arrays still happen in place
Heapsort
Selection sort in which I is a heap
Start with S and I
toss all items in I onto heap h. Ignore the heap order property
h.bottomUpHeap();
for (i = 0; i<n;i++) {
x = h.removeMin()
Append x to end of S
}
Takes O(n) time to do bottomUpHeap()
Takes n for loops
removeMin() takes O(logn) time
Heap sort runs in O(nlogn)
Heapsort works in place:
maintain heap backward
Good for arrays, weird for linked lists
MergeSort
Merging 2 sorted lists into one sorted list takes linear time
Let Q1 and Q2 be 2 sorted queues. Let Q start as an empty queue
While (neither Q1 nor Q2 is empty) {
item1= q1.front();
item2 = q2.front();
move smaller of item1 and item 2 from p resent queue to end of Q
}
Concatenate the rest of nonempty queue to Q
Merge sort is a divide and conquer recursive algorithm
Start with unsorted list I of n items.
Break I into 2 halves I1 and I2. I1 has n/2 ceiling, I2 has n/2 floor items.
Sort I1 recursively, yielding S1.
Sort I2 recursively, yielding S2.
Merge S1 and S2 into S
Base case is a list of length 1. That base case is already sorted.
1+ceiling(log2(n)) levels. Every level takes linear work
O(n) time per level
O(nlogn) time total
Not possible to do merge sort in place
Quicksort

Recursive divide and conquer algorithm
Fatest comparison based sort for arrays
Theta(n^2) worst case time
almost always runs in O(nlogn)
Start with listI of n items
Choose pivot item v from I
Partition I into 2 unsorted lists I1 and I2.
    I1: all keys smaller than v
    I2: all keys great than v
    Items with same key as v go in either list
    Pivot v does not go into either list
Quicksort I1, yielding S1
Quicksort I2, yielding S2
concatenate S1,v and S2 to make sorted list S
Best way to choose the picot is to randomly select an item from I.
On average, you can expect a 1/4-3/4 split
Linked Lists:
    Suppose we put all items with the same key as v into one of the lists, say I1. This will work, but it does not run fast.
    Better: partition I into 3 lists. I1,I2, Iv. Iv contains pivot v and all items with the same key as pivot v.
Quick sort on Arrays:
    In place
    Input: array a; sort items a[low]…a[high]
    choose pivot v; swap it with last item, a[high]
    i = low -1
    j = high
    i and j sandwich item to be sorted
    Invariants:
        all items at or left of index i have a key<=pivot
        all items at or right of index j have a key>= to pivot
    advance i to key>= pivot
    decrement j to key<= pivot
    swap items and i and j
    Repeat until i>= j
    Swap pivot back to the middle with i

61b Final Review
Lecture 33-34

Disjoint sets

Data structure that represents a bunch of sets. The sets must be disjoint: no item is in more than one set. Every item is in exactly one set.
Collection of disjoint sets is a partition.
Universe of items: All of the items that can be a member of a set
Operations:
    union: merges 2 sets into one
    find: takes an item, tells us which set it is in

these operations should run very quickly, other operations can have at least O(n). These are sometimes called union-find data structures, or partitions.

Project 3 is out: use kruskal's algorithm

List Based Disjoint Sets and the Quick Find Algorithm

- union: slower
- find: O(1)—given an item just look at the set field it contains
- Each set references a list of the items in that set
- Each item references the set it contains
- Union is slow because you have to walk through all sets being merged

Tree Based Disjoint Sets and the Quick Union Algorithm

- union: O(1)
- find: slower
- Quick union is faster overall than quick find
- Each set is stored as a tree
- Data structure is a forest
- Each item is initially root of its own tree
- No child or sibling references, only parent references, so you can only walk up the tree
- root records the identity of each set
- Union: make the root of one set become the child of the root of the new set whose true identity is the root
- Find: follow parent references from item to root of tree. Cost is proportional to the depth of the item that find

Union by size: keeps items from getting too deep. At each root, record the size of the tree. Make the smaller tree be a subtree of the larger one

Quick Union with array
array: store the parent of each item, or if the item has no parent(root) store the negative of the size of the tree that the item is in.

```
public void union(int root1, int root2) {
        if (array[root2]<array[root1]) {                    ///root2 has larger tree
                array[root2]+=array[root1]
                array[root1] = root2;
        } else {
                array[root1]+=array[roo2];
                array[root2] = root1;
        }
}
```

Path Compression:

```
public int find(int x) {
                if (array[x]<0) {
                        return x;          ///x is the root of the tree
        } else {
                /// find out who the root is; compress path
                array[x] = find(array[x]);
                return array[x];
```

}

Additional array maps root and item to set names
Often we only want to know if items x and y are in the same set. Run find(x), find(y), check if root is the same
Running time of quick union:
union: theta(1) time
find: theta(logu) worst case time, where u is the number of unions. Average running time close to constant
A sequence of f find and u unions takes
Theta(u + f*a(f+u,u)) worst case time
where a is a function that is extremely slow growing, called the inverse Ackermann function
a is never greater than 4 for any values of f and u
Selection
Find kth smallest key in list
(item at index j if list is sorted, j = k -1
Quickselect
modifies quicksort
start with unsorted list I of n items
choose pivot v from I
Partition I into lists I1 Iv and I2
items with same key as v go into any list, v has to go into IV
if (j< lI1l) {
recursively find item with index j in I1; return it
} else if (j<lI1l+lIVl){
return v
} else {
recursively find item with index j - lI1l-lIVl in I2; return it;
}
Average time is thetha(n) if we select pivots randomly

Lecture 35

Counting Sort

count keys in x
```
for (i=0; i<x.length;i++){
    counts[x[i].key]++;
}
```
Do a scan so counts[i] contains number of keys<i
```
total=0;
for(j=0;j<counts.length;j++){
    c=counts[j];
    counts[j] = total;
```

```
        total = total + c;
Let y be output array. counts[i[ tells us the first index of y to put items with key i
for (i = 0; i<x.length;i++){
        y[counts[x[i].key]]=x[i];
        counts[x[i].key]++;
}
```

Bucket sort and counting sort take theta(q+n) time
If q is in O(n), then they take O(n) time
Counting sort is faster on arrays
Bucket sort is quicker for sorting linked lists
What if the q>>n?
    counting sort and bucket sort normally won't work for this situation

Radix Sort

radix= base of number (ie radix 12 numbers)
Sor 1000 items in range 0,…,99999999
q=10: sort on last digit first
numbers are 0 padded to 8 digits
then sort second digit, then third, but not recursively—in place. This is because radix is stable
ones: 771, 721, 822, 955,405,5,929,825,777,28,829
tens: 405, 5,721, 822,925,825,28,829,955,771,777
hundres:5,28,405,721,771,777,822,825,829,925,955
faster if we sort 2 digits at a time (radix q= 100) or sort on 3 digits(radix q=1000)
computers: powers of 2 radix (ie q= 256)
    faster to take a bit from a base 2 number than digit from base 10
q = number of buckets for bucket sort, or number of counters you have if you do counting sort, or the radix of digits you use to sort keys
Time/How many passes?
    each pass inspects log2q bits of each key
    If all keys represented in b bits
    # of passes = ceiling(b/ log2q)
    one pass take n + q time
    time = O((n+q)ceiling(b/log2q))1
    choose q = O(n)—>each pass takes O(n)
    q large enough to make number of passes small
    choose q = c*n—>time = O(n+bn/logn)

Lecture 36-37

Splay Tree

Binary search tree
each operation dynamically balances the tree
All operations: O(logn) time on average
    n is number of entries in the tree

A single operation: theta(n) worst case
any sequence of k ops, with initially empty and never> items, takes O(klogn) worst case time
Fast access to entries accessed recently
Tree Rotations
splay tree are kept balance with rotations
rotate left and rotate right are reverse/mirror image operations
Operations:

1. Entry find(Object k);
   begins like ordinary BST: walk down tree to entry
   with key k or dead end.
   Let X be node where search ended. Splay X up the tree through a sequence of rotations, so X becomes root of tree.
   3 Cases:
   a. X is right child of a left child, or X is the left child of a right child
   "Zig-Zag Case"
   b. X is a left child of a left child, or the right child of a right child
   let X be the found node (or dead end).
   P is X's parent
   G is P's parent, and X's grandparent
   rotate P through G
   Then rotate X through P
   "Zig-Zig Case"
   Repeat A and B until X reaches root or child of root
   c. X is child of the root
   "Zig Case"
   rotate X with root

   A node initially at depth d on access path from root to X moves to depth <= 3+d/2
   If d is very big, then d is cut in half approximately
2. Entry min();
   Entry max();
   Find entry with min/max key
   Splay it to the root
3. Entry insert(Object k, Object v);
   Insert new entry (k,v)
   Splay new node to root
4. Entry remove(Object k);
   An entry having key k is removed from tree, as with ordinary BST.
   Let X be the node removed from the tree (NOT THE KEY DELETED). If the key has 0 or 1 children, just remove it. If it has 2 children, find a child to replace it. Splay X's parent to the root.

IF key k is not in the tree, splay the node where the algorithm ended up to the root

---

Amortized Analysis

formal way of figuring out average running times
Average time analysis for hash tables, disjoint sets, splay trees
The Averaging Method

Assume we have a good hashcode; chains in hash table are constant sized. Any hash table operation that doesn't resize the hash table takes constant time. For example, each non-resizing operations takes at most one second(in real life, about a microsecond).

n: items in hash table

N: number of buckets

One second to insert a new item, assuming no resizing, along with n++, and check the load factor < 1; If the load factor reaches one, double the size of the hash table from N to 2N., taking 2N seconds. Ensures that the load factor is always less than 1.

Suppose new table has n = 0, N = 1. After i insert operations, n = i. N must be a power of two, and >n. N is smallest power of two>n, which is <=2n.

Total seconds for all resizing ops is 2N-2

Cost of i inserts is <= i + 2N-2<=5i - 2 seconds

but N <=2n<2i

Average time of insert <=5i-2/i=5-2/i seconds = O(1)

Amortized running time is O(1), worst-case time theta(n) (when you need to resize)

The Accounting Method

Dollar: unit of time to execute slowest constant time computation

each operation:

an amortized cost: number of dollars we charge to do operation

an actual cost: actual number of O(1)-time computations

usually the actually cost is less that then amortized cost

When amortized cost>actual cost, extra $$ saved in bank to be spent on later ops

When actual cost>amortized cost, withdraw saved up time to pay for the expensive operation

The bank balance must never fall below zero. If you can prove this, you have a proper amortized analysis

Accounting of Hash Tables

Every op: 1$ actual time, not counting resizing

Insert() doubles table size if n= N after new item inserts and n++. This takes $2N to resize to 2N buckets

remove halves table is n=N/4 after item deleted and n—

taking $N to resize to N/2 buckets

Amortized Costs:

insert: $5

remove: $5

find: $1

at any time, we know the $$ in the bank from looking at n and N

Every insert/remove cost one actual dollar, puts $4 in bank. There must be >=4*ll(n-N/2)ll. This is a function of the state of the data structure itself, not the the history of its operations. Sometimes called the potential function.

By induction, if after 1 resize we have nonnegative balance, every resize after won't bankrupt us either

insert() resizes table if n reaches N. There are at least $2N in the bank

Resizing from N to 2N buckets costs $2N, so we can afford it.
remove() resizes table if n drops to N/4
There are at least $N dollars in the bank. Resizing from N to N/2 takes $N, so we can afford it.
Bank balance never drops below 0, so amortized costs= O(1)

Lectures 38,39,40

Randomized Analysis

Randomized algorithms use random probability to make decisions.
Ex: quicksort, quickselect, hash tables with random hash codes.
Expected value: of random variable X is the average value X takes after an infinite number of trials. Calculate by summing over the product of outcomes and their probabilities. Expectation values add linearly.

Randomized Hash tables: a random hash code maps each key to a random number. However, equal keys always map to the same random number.
find(k) —-> key k hashes to bucket b.
n keys labeled k_i. V_i is r.v. —> 1 if k_i hashes to bucket b, 0 otherwise
cost to find k = T = 1 + V1+V2……+V_n.
E[T] = 1+n/N since E[V_i] = 1/N —->random hashcode
So find(k) takes O(1) time since we can keep n/N constant
Insert and remove take O(1) expected amortized time.

Quicksort
n keys —> 1/n chance of choosing pivot.
first and fourth quartile are bad pivots —> 1/2 chance of choosing good pivot
If we choose a good pivot, the split is .25-.75 or better, and key k is in a subset with at most .75 of the keys.
Expected Depth for key k in n keys is 2log_4/3_(n) which is O(logn)
Quickselect
Expected number of keys partitioned in all partitioning steps is <= 8n.
So the running time of quick select on n keys is O(n)

Garbage Collection

Roots and Reachability
An object your program might use again is called live
The opposite is called garbage—objects your program cannot reference again
Root: any reference your program can access directly
local variables on stack
class variables
Object is live if:
referenced by a root or referenced by a field in another live object—ie reachable from the roots
Garbage collection runs DFS from the roots.

JVM keeps a hidden “visited” field that only the garbage collectors have access to

Memory Addresses

Memory is array of bytes with addresses
Declare local variable = naming a memory location
Java picks the address
Memory addresses stored in another memory address = pointers

Mark and Sweep Garbage Collection

2 phases

Mark phase performs a DFS from eery root; marks all live objects
Sweep phase passes over all object in memory, Garbage is reclaimed

JVM data structures keep track of free and allocated memory

Compaction

Fragmentation: the tendency of free memory to get broken up into small pieces. This means we would be unable to allocate a large object despite lots of free memory

References

In JVM a refernece is a handle. A handle is a pointer to a pointer. The second pointers are kept in a special table

Copying Garbage Collection

Faster than mark and sweep because only one phase
Memory is divided into 2 spaces, old space and new space
Finds live objects by DFS

when it encounters an object in the old space, it immediately moves it to the new space, compaction included.

Next time, new space is relabeled old space, old space is relabeled new space.
Advantage: very fast
Disadvantage: you can only use half of your memory, but its fast

Generational Garbage Collection

separated into young and old generation

the old generation doesn't change very much or very quickly. Use a mark and sweep garbage collector on the old generation, but use it very infrequently.

The young generation is a lot larger. All objects are born in the young generation, most die there. When Eden fills, live objects move to the upper half young generation into one of the two survivor spaces. The survivor space is expandable and is collected with copying garage collector. When objects get old, they are put into the old generation.

Minor collections: frequent; only affect young generation
Major collections: cover all objects
Special table of reference form old objects to young is added to the roots for minor collections

Augmenting Data Structures

2-3-4 trees with fast neighbors, no duplicate keys
Given key k, determine whether k is in tree and what next smaller and next larger keys are. These keys are called neighbors of k, and we want to do this in O(1) time.
Combine 2-3-4 tree with a hash table
Maps key to record with next smaller and next larger key

insert: determine by tree search in O(logn) time what next smaller and larger keys are.
Update all 3 keys records in hashtable; takes O(1) time.
Splay Trees with Node information
Record size and height of each subtree at each node
Height of empty tree is -1
Rotation changes heights of ancestors.