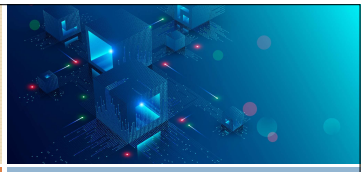


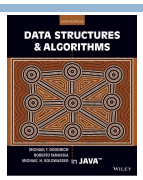
Data STRUCTURES (CSE 201)

Prof. Ümit D. ULUŞAR

1

Textbook

- Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014



- E-mail : umitulusar@gmail.com

2

Grading

- Lab Assignments/Homeworks/Term Projects 35%
- Midterm 25%
- Final Exam 40%

3

Attendance and Assignments

- **Attendance:** If you miss a class it is your responsibility to find out what was discussed in the class. There will be no make up for popup quizzes if you miss one. In addition attending lab sessions are mandatory. If you do not attend three or more lab sessions without any provable excuse, your lab score will be zero.
- **Late Assignments:** All homework assignments are due to the date set by instructor or TA as the deadline. No late submission.

4

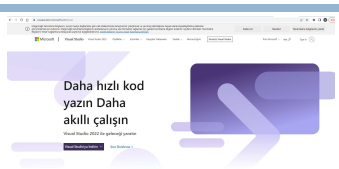
Software Installation Instructions

- **JDK**
- http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
- jdk-8u20-windows-i586.exe (32-bit)
- jdk-8u20-windows-x64.exe (64-bit)

- **Eclipse**
- Eclipse Standard (Free)
  - https://www.eclipse.org/downloads/

5

C# Visual Studio



6

JAVA PRIMER 1: TYPES AND OPERATORS

Prof. Ümit D. ULUŞAR

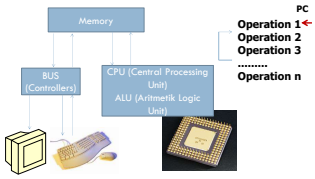
1

Purpose of Programming

- Learn the way computers process "Computational thinking" and be able to write small size programs.
- Ability to use a vocabulary of computational tools in order to be able to understand programs written by others.
- Have the ability to map scientific problems into a computational framed programs written by others.

2

Program



3

Programming Language

- Language is a set of valid sentences.
- What makes a language valid?
  - Syntax (Grammatical (syntactically valid))
  - Semantics (Sensible (Semantically valid))
- Trees are walking.

4

Programming Languages

- Machine languages — interpreted directly in hardware
- Assembly languages — thin wrappers over a corresponding machine language
- **High-level languages — anything machine-independent**
- System languages — designed for writing low-level tasks, like memory and process management
- Scripting languages — generally extremely high-level and powerful
- Domain-specific languages — used in highly special-purpose areas only
- Visual languages — non-text based

5

Programming Language Java

- Java Features
  - Widely used, widely available.
  - Embraces full set of modern abstraction.
  - Variety of automatic checks for mistakes in programs.
- Java Economy
  - Mars rover.
  - Cell phones (Android)
  - Web servers
  - Medical Devices
  - Super Computing

6

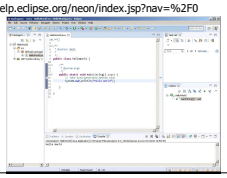
Java Installation (Eclipse)

http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/marsr

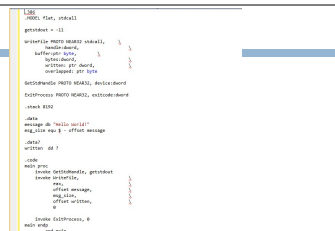
7

Eclipse IDE

Geting Started
http://help.eclipse.org/neon/index.jsp?nav=%2F0



9



11

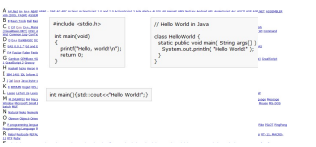
The Java Compiler

- Java is a compiled language.
- Programs are compiled into byte-code executable files, which are executed through the Java virtual machine (JVM).
  - The JVM reads each instruction and executes that instruction.
- A programmer defines a Java program in advance and saves that program in a text file known as source code.
- For Java, source code is conventionally stored in a file named with the **.java** suffix (e.g., **demo.java**) and the byte-code file is stored in a file named with a **.class** suffix, which is produced by the Java compiler.

8

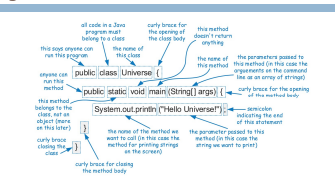
Hello World in Different Languages

- http://www.helloworldexample.net/java-hello-world-example.html



10

Program Structure



12

Components of a Java Program

- In Java, executable statements are placed in functions, known as **methods**, that belong to class definitions.
- The static method named **main** is the first method to be executed when running a Java program.
- Any set of statements between the braces "{" and "}" define a program block.

Built-in Types

- Java has several built-in types, which are basic ways of storing data.
- An identifier variable can be declared to hold any base type and it can later be reassigned to hold another value of the same type.

boolean	a boolean value: true or false
char	16-bit Unicode character
byte	8-bit signed two's complement integer
short	16-bit signed two's complement integer
int	32-bit signed two's complement integer
long	64-bit signed two's complement integer
float	32-bit floating-point number (IEEE 754-1985)
double	64-bit floating-point number (IEEE 754-1985)

```
boolean flag = true;
boolean verbose, debug;
char grade = 'A';
byte b = 12;
short s = 24;
int i, j, k = 257;
long l = 890L;
float pi = 3.1416F;
double e = 2.71828, a = 6.022e23;
```


Text

- String data type is useful for input and output.

Values	Sequence of characters
Typical literals	"Hello," " " " "
Operation	Concatenate
Operator	+

expression	value
"Hi, " + "Bob"	"Hi,Bob"
"1"+"2"+"1"	"121"
"1 "+99	"1 99"

```
public class Ruler {
public static void main(String[] args) {
    String ruler1="1";
    String ruler2=ruler1+" 2 "+ruler1;
    String ruler3=ruler2+" 3 "+ruler2;
    String ruler4=ruler3+" 4 "+ruler3;
    System.out.println(ruler4);
}
}
```

```
1 2 1 3 1 2 1 4 1 2 1 3 1 2 1
```

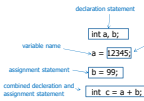

Identifiers

- The name of a class, method, or variable in Java is called an **identifier**, which can be any string of characters as long as it begins with a letter and consists of letters.
- Exceptions:

Reserved Words				
abstract	default	goto	package	synchronized
assert	do	if	private	this
boolean	double	implements	protected	throw
break	else	import	public	throws
byte	enum	instanceof	return	transient
case	extends	int	short	true
catch	false	interface	static	try
char	final	long	strictfp	void
class	finally	native	super	volatile
const	float	new	switch	while
continue	for	null		

Variables

- Variable is a name that refers to a value.
- Assignment statement associates a value with a variable.



Expressions and Operators

- Existing values can be combined into expressions using special symbols and keywords known as operators.
- The semantics of an operator depends upon the type of its operands.
- For example, when a and b are numbers, the syntax a + b indicates addition, while if a and b are strings, the operator + indicates concatenation.

Arithmetic Operators

- Java supports the following arithmetic operators:

+	addition
–	subtraction
*	multiplication
/	division
%	the modulo operator

- If both operands have type int, then the result is an int; if one or both operands have type float, the result is a float.
- Integer division has its result truncated.

Integer Operations

```
public class IntOps {
public static void main(String[] args) {
    int a = Integer.parseInt(args[0]);
    int b = Integer.parseInt(args[1]);
    int sum = a + b;
    int prod = a * b;
    int quot = a / b;
    int rem = a % b;
    System.out.println(a + " + " + b + " = " + sum);
    System.out.println(a + " * " + b + " = " + prod);
    System.out.println(a + " / " + b + " = " + quot);
    System.out.println(a + " % " + b + " = " + rem);
}
}
```

```
1234 + 99 = 1333
1234 * 99 = 122166
1234 / 99 = 12
1234 % 99 = 46
```


Excerpts from Java's Math Library

- public class Math
  - double abs(double a) //absolute value of a
  - double max(double a, double b) //maksimum of a and b
  - double min(double a, double b) //minimum of a and b
  - double cos(double theta) //cosine function
  - double tan(double theta) //tongent function
  - double exp(double a) //exponential (e^a)
  - double log(double a) //natural log(log_e a or ln a)
  - double pow(double a, double b) //a^b
  - long round(double a) //round to the nearest integer
  - double random() //random number in [0,1)
  - double sqrt(double a) //square root of a
  - double E // value of e (constant)
  - double PI // value of PI (constant)

http://docs.oracle.com/javase/6/docs/api/java/lang/Math.html

Integers

- int 32 bit signed 2's complement integer

Values	integers between -2^{31} and $+2^{31}-1$
Typical literals	1234 99 0 89
Operation	Add, Subtract , Multiply, Divide, Reminder
Operator	+ - * / %

expression	value	comment
5 + 3	8	
5 / 3	1	No fractional part
1 / 0		Run-time error
3 + 5 / 2	5	/ has precedence
3 – 5 - 2	-4	Left association
3 – (5 - 2)	0	Unambiguous

Floating-Point Numbers

- double real numbers (specified by IEEE 754 standards)

Values	Real numbers between 0x0.0000000000001p-1022 and 0x1.fffffffffffffp1023
Typical literals	3.14159 6.02e23 -3.0
Operation	Add, Subtract , Multiply, Divide, Reminder
Operator	+ - * / %

expression	value
5.0 / 3.0	1.6666666666666667
6.02e23 / 2.0	3.01e23
1 / 0	Infinity
Math.sqrt(2.0)	1.4142135623730951
Math.sqrt(-1.0)	NaN

Ex: Quadratic Equation

- Ex. Solve quadratic equation $x^2 + bx + c = 0$

$$\text{roots} = \frac{-b \pm \sqrt{b^2 - 4c}}{2}$$

```
public class Quadratic {
public static void main(String[] args) {
    // parse coefficients from command-line
    double b = Double.parseDouble(args[0]);
    double c = Double.parseDouble(args[1]);
    // calculate roots
    double discriminant = b*b - 4.0*c;
    double d = Math.sqrt(discriminant);
    double root1 = (-b + d) / 2.0;
    double root2 = (-b - d) / 2.0;
    // print them out
    System.out.println(root1);
    System.out.println(root2);
} }
```

```
% java Quadratic -3.0 2.0
2.0
1.0
```


Increment and Decrement Ops

- Java provides the plus-one increment (++) and decrement (--) operators.
  - If such an operator is used in front of a variable reference, then 1 is added to (or subtracted from) the variable and its value is read into the expression.
  - If it is used after a variable reference, then the value is first read and then the variable is incremented or decremented by 1.

```
int i = 8;
int j = i++;        // j becomes 8 and then i becomes 9
int k = ++i;        // i becomes 10 and then k becomes 10
int m = i--;        // m becomes 10 and then i becomes 9
int n = 9 + --i;    // i becomes 8 and then n becomes 17
```


Logical Operators (Comparisons)

- Java supports the following operators for numerical values, which result in Boolean values:

op	meaning	true	false
==	equal	2 == 2	2 == 3
!=	not equal	2 != 3	2 != 2
<	less than	2 < 13	2 < 2
<=	less than or equal	2 <= 2	3 <= 2
>	greater than	13 > 2	2 > 13
>=	greater than or equal	3 >= 2	2 >= 3

Typical comparison expressions

Non-negative discriminant ?	(b*b – 4.0*a*c) >= 0.0
Beginning of a century?	(year % 100) == 0
Legal month?	(month >=1) && (month <=12)

Bitwise Operators

- Java provides the following bitwise operators for integers and booleans:

~	bitwise complement (prefix unary operator)
&	bitwise and
|	bitwise or
^	bitwise exclusive-or
<<	shift bits left, filling in with zeros
>>	shift bits right, filling in with sign bit
>>>	shift bits right, filling in with zeros

Booleans

- a boolean takes either true or false value

Values	true or false				
Typical literals	true false				
Operation	and,	or ,	not		
Operator	&&	||	!		

a	!a	b	b	a && b	a || b
true	false	false	false	false	false
false	true	false	true	false	true
		false	false	false	true
		true	true	true	True

The && and || operators **short circuit**, in that they do not evaluate the second operand if the result can be determined based on the value of the first operand.

Ex: Leap Year

- Is a given year a leap year?
- Every year that is exactly divisible by four is a leap year, except for years that are exactly divisible by 100, but these centurial years are leap years if they are exactly divisible by 400. For example, the years 1700, 1800, and 1900 are not leap years, but the years 1600 and 2000 are [Wiki].

```
public class LeapYear {
public static void main(String[] args) {
    int year = Integer.parseInt(args[0]);
    boolean isLeapYear;
    // divisible by 4 but not 100
    isLeapYear = (year % 4 == 0) && (year % 100 != 0);
    // or divisible by 400
    isLeapYear = isLeapYear || (year % 400 == 0);
    System.out.println(isLeapYear);
} }
```

```
% java LeapYear 2004
true
% java LeapYear 1900
false
% java LeapYear 2000
true
```


Operator Precedence

Operator Precedence		
	Type	Symbols
1	array index	[ ]
	method call	( )
	dot operator	.
2	postfix ops	*exp++ exp--*
	prefix ops	*++exp --exp +exp -exp ~exp !exp*
	cast	*(type) exp*
3	mult.div.	* / %
4	add./subt.	+ -
5	shift	<< >> >>>
6	comparison	< <= > >= instanceof
7	equality	== !=
8	bitwise-and	&
9	bitwise-xor	^
10	bitwise-or	|
11	and	&&
12	or	||
13	conditional	*booleanExpression ? valueIfTrue : valueIfFalse*
14	assignment	= += -= *= /= %= <<= >>= >>>= &= ^= |=

JAVA PRIMER: I/O METHODS AND CONTROL FLOW

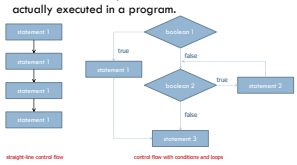
Simple Output

- Java provides a built-in static object, called System.out, that performs output to the "standard output" device, with the following methods:

print(String *s*): Print the string *s*.
print(Object *o*): Print the object *o* using its toString method.
print(*baseType b*): Print the base type value *b*.
println(String *s*): Print the string *s*, followed by the newline character.
println(Object *o*): Similar to print(*o*), followed by the newline character.
println(*baseType b*): Similar to print(*b*), followed by the newline character.

Flow Control

- Control flow is the squence of statements that are actually executed in a program.



Ex. If Statement

- Heads or Tails

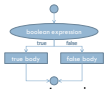
```
public class Flip {
    public static void main(String[] args) {
        if (Math.random() < 0.5)
            System.out.println("Heads");
        else
            System.out.println("Tails");
    }
}
```

```
% java Flip
Heads
% java Flip
Heads
% java Flip
Tails
% java Flip
Heads
```


If Statements

- The syntax of a simple **if** statement is as follows:

if (*booleanExpression*)
    trueBody
else
    falseBody



- booleanExpression is a boolean expression and trueBody and falseBody are each either a single statement or a block of statements enclosed in braces ("{" and "}").

If Statement Examples

Absolute value	if (x < 0) x = -x;

Put x and y into sorted order	if (x>y)
	{
	  int t = x;
	  x = y;
	  y = t;
	}

Maximum of x and y	if (x > y) max = x; else max = y;
Error check for division operation	if (den == 0) System.out.println("Division by zero"); else System.out.println("Quotient = " + num/den);

Error check for quadratic formula	double discriminant = b*b – 4.0*c;
	if (discriminant < 0.0)
	{
	  System.out.println("No real roots");
	}
	else
	{
	  System.out.println(-b + Math.sqrt(discriminant)/2.0);
	  System.out.println(-b - Math.sqrt(discriminant)/2.0);
	}

Compound if Statements

- There is also a way to group a number of boolean tests, as follows:

if (*firstBooleanExpression*)
    firstBody
else if (*secondBooleanExpression*)
    secondBody
else
    thirdBody

Enum Types

- Java supports an elegant approach to representing choices from a finite set by defining what is known as an enumerated type, or enum for short.
- These are types that are only allowed to take on values that come from a specified set of names. They are declared as follows:

modifier **enum** *name* { *valueName0* , *valueName1* , . . . };

- Once defined, Day becomes an official type and we may declare variables or parameters with type Day. A variable of that type can be declared as:

```
public enum Day { MON, TUE, WED, THU, FRI, SAT, SUN };
public static Day today;
public static void main(String[] args) {
    today = Day.TUE;
}
```


Switch Example

```
public enum Day { MON, TUE, WED, THU, FRI, SAT, SUN };
public static Day today;
public static void main(String[] args) {
    today = Day.TUE;
    switch (today) {
        case MON:
        System.out.println("This is tough.");
        break;
        case TUE:
        System.out.println("This is getting better.");
        break;
        case WED:
        System.out.println("Half way there.");
        break;
        case THU:
        System.out.println("I can see the light.");
        break;
        case FRI:
        System.out.println("Now we are talking.");
        break;
        default:
        System.out.println("Day off!");
    }
}
```


Ex. Compound If Statement

- Pay a certain tax rate depending on income level.

Income	Rate
0-47,500	22%
47,500 – 120,000	25%
120,000 -	35%

```
double rate;
if (income < 47500) rate = 0.22;
else if (income < 120000) rate = 0.25;
else rate = 0.35;
```


Switch Statements

- Java provides for multiple-value control flow using the switch statement.
- The switch statement evaluates an integer, string, or enum expression and causes control flow to jump to the code location labeled with the value of this expression.
- If there is no matching label, then control flow jumps to the location labeled "default."
- This is the only explicit jump performed by the switch statement, however, so flow of control "falls through" to the next case if the code for a case is not ended with a **break** statement

Break and Continue

- Java supports a **break** statement that immediately terminate a while or for loop when executed within its body.
- Java also supports a **continue** statement that causes the current iteration of a loop body to stop, but with subsequent passes of the loop proceeding as expected.

Ex. Break and Continue

```
public class BreakAndContinue {

public static void main(String[] args)
{
int N = 15;
for (int i = 1; i <= N; i++)
{
    if (i>3 && i<12) continue;
    System.out.println(i);
}
}

}
```

```
1
2
3
12
13
14
15
```

```
public class BreakAndContinue {

public static void main(String[]
args) {
int N = 15;
for (int i = 1; i <= N; i++)
{
    if (i>3 && i<12) break;
    System.out.println(i);
}
}
}
```

```
1
2
3
```


Ex. While Statement

- Powers of 2

```
public class PowersOfTwo {
public static void main(String[] args) {
    // last power of two to print
    int N = 10;
    int i = 0; // loop control counter
    int v = 1; // current power of two
    while (i <= N) {
        System.out.println(i + " " + v);
        i = i + 1;
        v = 2 * v;
    }
}
}
```

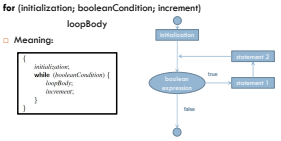
```
0 1
1 2
2 4
3 8
4 16
5 32
6 64
7 128
8 256
9 512
10 1024
```


For Loops

- The traditional **for**-loop syntax consists of four sections—an initialization, a boolean condition, an increment statement, and the body—although any of those can be empty.

for (initialization; booleanCondition; increment)
    loopBody

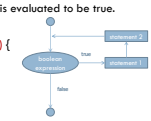
- Meaning:



While Loops

- The while loop is a common repretition structure.
- Such a loop tests that a certain condition is satisfied and will perform the body of the loop each time this condition is evaluated to be true.

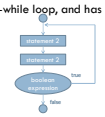
while (booleanExpression) {
    loopBody
}



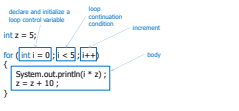
Do-While Loops

- Java has another form of the while loop that allows the boolean condition to be checked at the end of each pass of the loop rather than before each pass.
- This form is known as a do-while loop, and has syntax shown below:

do
    loopBody
while (booleanExpression)



Ex. For Loops



Ex. For Loops

- Subdivisor of a ruler.

```
public class RulerN {
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        String ruler = " ";
        for (int i = 1; i <= N; i++) {
            ruler = ruler + i + ruler;
        }
        System.out.println(ruler);
    }
}
```

Input	Output
1	" 1 "
2	" 1 2 1 "
3	" 1 2 1 3 1 2 1 "

For-Each Loops

- Since looping through elements of a collection is such a common construct, Java provides a shorthand notation for such loops, called the **for-each** loop.
- The syntax for such a loop is as follows:

```
for (elementType name : container)
    loopBody
```


Ex. Loops

print largest power of two less than or equal to N	`int v = 1;`
	`while (v <= N/2)`
	`  v = 2 * v;`
	`System.out.println(v);`

compute a finite sum (1 + 2 + ... + N)	`int sum = 0;`
	`for (int i = 1 ; i <= N; i++)`
	`  sum += i;`
	`System.out.println(sum);`

compute finite product (1 X 2 X ..... X N)	`int product = 1;`
	`for (int i = 1 ; i <= N; i++)`
	`  product *= i;`
	`System.out.println(product);`

print a table of function values	`for (int i = 0 ; i <= N; i++)` `  System.out.println(i + " " + 2*Math.PI*i/N);`

Ex. For Loops

- Compute the sum of an array of doubles:

```
public static double sum(double[ ] data) {
double total = 0;
for (int j=0; j < data.length; j++) // note the use of length
    total += data[j];
return total;
}
```

- Compute the maximum in an array of doubles:

```
public static double max(double[ ] data) {
double currentMax = data[0]; // assume first is biggest (for now)
for (int j=1; j < data.length; j++) // consider all other entries
if (data[j] > currentMax) // if data[j] is biggest thus far...
currentMax = data[j]; // record it as the current max
return currentMax;
}
```


For-Each Loop Example

- Computing a sum of an array of doubles:

```
public static double sum(double[] data) {
    double total = 0;
    for (double val : data) // Java's for-each loop style
        total += val;
    return total;
}
```

- When using a for-each loop, there is no explicit use of array indices.
- The loop variable represents one particular element of the array.

Simple Input

- There is also a special object, **System.in**, for performing input from the Java console window.
- A simple way of reading input with this object is to use it to create a **Scanner** object, using the expression

  new Scanner(System.in)

```
import java.util.Scanner; // loads Scanner definition for our use
public class InputExample {
    public static void main(String[ ] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter your age in years: ");
        double age = input.nextDouble( );
        System.out.print("Enter your maximum heart rate: ");
        double rate = input.nextDouble( );
        double fb = (rate - age) * 0.65;
        System.out.println("Your ideal fat-burning heart rate is " + fb);
        input.close(); // close input stream
    }
}
```


java.util.Scanner Methods

- The Scanner class reads the input stream and divides it into tokens, which are strings of characters separated by delimiters.

`hasNext()`: Return **true** if there is another token in the input stream.

`next()`: Return the next token string in the input stream; generate an error if there are no more tokens left.

`hasNextType()`: Return **true** if there is another token in the input stream and it can be interpreted as the corresponding base type, *Type*, where *Type* can be Boolean, Byte, Double, Float, Int, Long, or Short.

`nextType()`: Return the next token in the input stream, returned as the base type corresponding to *Type*; generate an error if there are no more tokens left or if the next token cannot be interpreted as a base type corresponding to *Type*.

CLASSES AND OBJECTS

Another Example

```
public class Counter {
    private int count; // a simple integer instance variable
    public Counter( ) { } // default constructor (count is 0)
    public Counter(int initial) { count = initial; } // an alternate constructor
    public int getCount( ) { return count; } // an accessor method
    public void increment( ) { count++; } // an update method
    public void increment(int delta) { count += delta; } // an update method
    public void reset( ) { count = 0; } // an update method
}
```

- This class includes one instance variable, named count, which will have a default value of zero, unless we otherwise initialize it.
- The class includes two special methods known as constructors, one accessor method, and three update methods.

H.W. 1.

- Write a short method in any language that counts the number of vowels in a given character string.
- Write a method that takes an array of float values and determines if all the numbers are different from each other (that is, they are distinct).
- Write a method that takes an array containing the set of all integers in the range 1 to 52 and shuffles it into random order. Your method should output each possible order with equal probability.

Classes and Objects

- Every **object** is an instance of a **class**, which serves as the type of the object and as a blueprint, defining the data which the object stores and the methods for accessing and modifying that data. The critical members of a class in Java are the following:
  - **Instance variables**, which are also called **fields**, represent the data associated with an object of a class. Instance variables must have a type, which can either be a base type (such as Int, Float, or double) or any class type.
  - **Methods** in Java are blocks of code that can be called to perform actions. Methods can accept parameters as arguments, and their behavior may depend on the object upon which they are invoked and the values of any parameters that are passed. A method that returns information to the caller without changing any instance variables is known as an **accessor** method, while an **update** method is one that may change one or more instance variables when called.

Creating and Using Objects

- Classes are known as **reference types** in Java, and a variable of that type is known as a **reference variable.**
- A reference variable is capable of storing the location (i.e., memory address) of an object from the declared class.
  - So we might assign it to reference an existing instance or a newly constructed instance.
  - A reference variable can also store a special value, null, that represents the lack of an object.
- In Java, a new object is created by using the **new** operator followed by a call to a constructor for the desired class.
- A **constructor** is a method that always shares the same name as its class. The new operator returns a reference to the newly created instance; the returned reference is typically assigned to a variable for further use.

Continued Example

```
public class CounterDemo {
    public static void main(String[ ] args) {
    Counter c; // declares a variable; no counter yet constructed
    c = new Counter( ); // constructs a counter; assigns its reference to c
    c.increment( ); // increases its value by one
    c.increment(3); // increases its value by three more
    int temp = c.getCount( ); // will be 4
    c.reset( ); // value becomes 0
    Counter d = new Counter(5);// declares and constructs a counter having value 5
    d.increment( ); // value becomes 6
    Counter e = d; // assigns e to reference the same object as d
    temp = e.getCount( ); // will be 6 (as e and d reference the same counter)
    e.increment(2); // value of e (also known as d) becomes 8
    }
}
```

- Here, a new Counter is constructed at line 4, with its reference assigned to the variable c. That relies on a form of the constructor, Counter( ), that takes no arguments between the parentheses.

Wrapper Types

- There are many data structures and algorithms in Java's libraries that are specifically designed so that they only work with object types (not primitives).
- To get around this obstacle, Java defines a **wrapper** class for each base type.
  - Java provides additional support for implicitly converting between base types and their wrapper types through a process known as automatic **boxing** and **unboxing**.

Literals

- *A literal is any "constant" value that can be used in an assignment or other expression.*
- Java allows the following kinds of literals:
  - null
  - true and false.
  - Integer, floating point, char , string

'\n'	**newline**	'\t'	**tab**
'\b'	**backspace**	'\r'	**return**
'\f'	**form feed**	'\\'	**backslash**
'\''	**single quote**	'\"'	**double quote**

The Dot Operator

- One of the primary uses of an object reference variable is to access the members of the class for this object, an instance of its class.
- This access is performed with the dot (".") operator.
- We call a method associated with an object by using the reference variable name, following that by the dot operator and then the method name and its parameters.

Example Wrapper Types

Base Type	*Class Name*	*Creation Example*	*Access Example*
boolean	Boolean	obj = new Boolean(true);	obj.booleanValue()
char	Character	obj = new Character('Z');	obj.charValue()
byte	Byte	obj = new Byte((byte) 34);	obj.byteValue()
short	Short	obj = new Short((short) 100);	obj.shortValue()
int	Integer	obj = new Integer(1045);	obj.intValue()
long	Long	obj = new Long(10849L);	obj.longValue()
float	Float	obj = new Float(3.934F);	obj.floatValue()
double	Double	obj = new Double(3.934);	obj.doubleValue()

```
int j = 8;
Integer a = new Integer(12);
int k = a; // implicit call to a.intValue()
int m = j + a; // a is automatically unboxed before the addition
a = 3 * m; // result is automatically boxed before assignment
Integer b = new Integer("-135"); // constructor accepts a String
int n = Integer.parseInt("2013"); // using static method of Integer class
```


Signatures

- If there are several methods with this same name defined for a class, then the Java runtime system uses the one that matches the actual number of parameters sent as arguments, as well as their respective types.
- A method's name combined with the number and types of its parameters is called a method's **signature**, for it takes all of these parts to determine the actual method to perform for a certain method call.
- A reference variable v can be viewed as a "pointer" to some object o.

Defining Classes

- A **class definition** is a block of code, delimited by braces "{" and "}" , within which is included declarations of instance variables and methods that are the members of the class.
- Immediately before the definition of a class, instance variable, or method in Java, keywords known as modifiers can be placed to convey additional stipulations about that definition.

Access Control Modifiers

- The **public** class modifier designates that all classes may access the defined aspect.
- The **protected** class modifier designates that access to the defined aspect is only granted to classes that are designated as subclasses of the given class through inheritance or in the same package.
- The **private** class modifier designates that access to a defined member of a class be granted only to code within that class.
- When a variable or method of a class is declared as **static,** it is associated with the class as a whole, rather than with each individual instance of that class.

Implicit Casting

- There are cases where Java will perform an implicit cast based upon the context of an expression.
- You can perform a **widening cast** between primitive types (such as from an int to a double), without explicit use of the casting operator.
- However, if attempting to do an implicit **narrowing cast**, a compiler error results.

```
int i1 = 42;
double d1 = i1; // d1 gets value 42.0
i1 = d1; // compile error: possible loss of precision
```


Sample Program

```
public class CreditCard {
// Instance variables:
private String customer; // name of the customer (e.g., "John Bowman")
private String bank; // name of the bank (e.g., "California Savings")
private String account; // account identifier (e.g., "5391 0375 9387 5309")
private int limit; // credit limit (measured in dollars)
protected double balance; // current balance (measured in dollars)
// Constructors:
public CreditCard(String cust, String bk, String acnt, int lim, double
initialBal) {
customer = cust;
    bank = bk;
    account = acnt;
    limit = lim;
    balance = initialBal;
}
public CreditCard(String cust, String bk, String acnt, int lim) {
    this(cust, bk, acnt, lim, 0.0); // use a balance of zero as default
}
```


Parameters

- A method's parameters are defined in a comma-separated list enclosed in parentheses after the name of the method.
  - A parameter consists of two parts, the parameter type and the parameter name.
  - If a method has no parameters, then only an empty pair of parentheses is used.
- All parameters in Java are **passed by value,** that is, any time we pass a parameter to a method, a copy of that parameter is made for use within the method body.
  - So if we pass an int variable to a method, then that variable's integer value is copied.
  - The method can change the copy but not the original.
  - If we pass an object reference as a parameter to a method, then the reference is copied as well.

The Keyword this

- Within the body of a method in Java, the keyword **this** is automatically defined as a reference to the instance upon which the method was invoked. There are three common uses:
  1. To store the reference in a variable, or send it as a parameter to another method that expects an instance of that type as an argument.
  2. To differentiate between an instance variable and a local variable with the same name.
  3. To allow one constructor body to invoke another constructor body.

Sample Program

```
// Accessor methods:
public String getCustomer( ) { return customer; }
public String getBank( ) { return bank; }
public String getAccount( ) { return account; }
public int getLimit( ) { return limit; }
public double getBalance( ) { return balance; }
// Update methods:
public boolean charge(double price) { // make a charge
    if (price + balance > limit) // if charge would surpass limit
    return false; // refuse the charge
    // at this point, the charge is successful
    balance += price; // update the balance
    return true; // announce the good news
}
public void makePayment(double amount) { // make a payment
balance -= amount;
}
// Utility method to print a card's information
public static void printSummary(CreditCard card) {
    System.out.println("Customer = " + card.customer);
    System.out.println("Bank = " + card.bank);
    System.out.println("Account = " + card.account);
    System.out.println("Balance = " + card.balance); // implicit cast
    System.out.println("Limit = " + card.limit); // implicit cast
}
// main method shown on next page...
```


Sample Program

```
public static void main(String[ ] args) {
    CreditCard[ ] wallet = new CreditCard[3];
    wallet[0] = new CreditCard("John Bowman", "California Savings", "5391 0375 9387 5309",
    5000);
    wallet[1] = new CreditCard("John Bowman", "California Federal","3485 0399 3395 1954",
    3500);
    wallet[2] = new CreditCard("John Bowman", "California Finance", "5391 0375 9387 5309",
    2500, 300);

    for (int val = 1; val <= 16; val++) {
        wallet[0].charge(3*val);
        wallet[1].charge(2*val);
        wallet[2].charge(val);
    }
    for (CreditCard card : wallet) {
        CreditCard.printSummary(card); // calling static method
    while (card.getBalance( ) > 200.0) {
        card.makePayment(200);
    System.out.println("New balance = " + card.getBalance( ));
}
}
}
}
```


Casting

- Casting is an operation that allows us to change the type of a value.
- We can take a value of one type and cast it into an equivalent value of another type.
- There are two forms of casting in Java: **explicit casting** and **implicit casting**.

Explicit Casting

- Java supports an explicit casting syntax with the following form:

  (type) exp
- Here "type" is the type that we would like the expression exp to have.
- This syntax may only be used to cast from one primitive type to another primitive type, or from one reference type to another reference type.

```
double d1 = 3.2;
double d2 = 3.9999;
int i1 = (int) d1; // i1 gets value 3
int i2 = (int) d2; // i2 gets value 3
double d3 = (double) i2; // d3 gets value 3.0
```


Packages

- The Java language takes a general and useful approach to the organization of classes into programs. Every stand-alone public class defined in Java must be given in a separate file. The file name is the name of the class with a .java extension. So a class declared as public class Window is defined in a file Window.java. That file may contain definitions for other stand-alone classes, but none of them may be declared with public visibility.
- To aid in the organization of large code repository, Java allows a group of related type definitions (such as classes and enums) to be grouped into what is known as a ***package. For types to belong to a package named packageName, their source*** code must all be located in a directory named *packageName and each file must* begin with the line:

  package *packageName;*

Import Statement

- A type can be refered within a package using its fully qualified name. For example, the Scanner class is defined in the java.util package, and can refer to it as java.util.Scanner.
- We could declare and construct a new instance of that class in a project using the following statement:

  java.util.Scanner input = new java.util.Scanner(System.in);
- However, all the extra typing needed to refer to a class outside of the current package can get tiring. In Java, we can use the import keyword to include external classes or entire packages in the current file. To import an individual class from a specific package, we type the following at the beginning of the file:

  import *packageName.className;*
- Ex:

  import java.util.Scanner;
- and then we were allowed to use the less burdensome syntax:

  Scanner input = new Scanner(System.in);

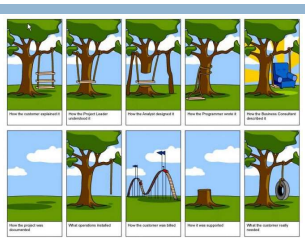
  ❑To import all the classes from a specific package, we type the following at the beginning of the file import *packageName.**;

  import java.util.*;

SOFTWARE DEVELOPMENT

Software Development

- Traditional software development involves several phases. Three major steps are:
- 1. Design
- 2. Coding
- 3. Testing and Debugging



Design

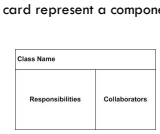
- It is in the design step that we decide how to divide the workings of our program into classes, when we decide how these classes will interact, what data each will store, and what actions each will perform.
- There are some rules of thumb that we can apply when determining how to define our classes:

Design

- ***Responsibilities:***
  - *Divide the work into different actors, each with a different* responsibility. Try to describe responsibilities using action verbs. These actors will form the classes for the program.
- ***Independence:***
  - *Define the work for each class to be as independent from* other classes as possible. Subdivide responsibilities between classes so that each class has autonomy over some aspect of the program. Give data (as instance variables) to the class that has jurisdiction over the actions that require access to this data.
- ***Behaviors:***
  - *Define the behaviors for each class carefully and precisely, so* that the consequences of each action performed by a class will be well understood by other classes that interact with it. These behaviors will define the methods that this class performs, and the set of behaviors for a class form the *protocol by which other pieces of code will interact with objects from the* class.

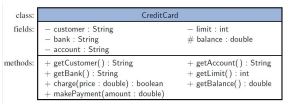
Class-Responsibility-Collaborator (CRC) Cards

- A common tool for developing an initial high-level design for a project is the use of CRC *cards*.
  - Each card represent a component.



UML Diagrams

- As the design takes form, a standard approach to explain and document the design is the use of UML (Unified Modeling Language) diagrams to express the organization of a program.
- One type of UML figure is known as a ***class diagram.***



A UML Class diagram for the CreditCard class

Pseudocode

- As an intermediate step before the implementation of a design, programmers are often asked to describe algorithms in a way that is intended for human eyes only called ***pseudocode.***

Coding

- One of the key steps in implementing an object-oriented program is coding the descriptions of classes and their respective data and methods.
- In order to accelerate the development of this skill, we will discuss various ***design patterns for designing*** object-oriented programs.
- These patterns provide templates for defining classes and the interactions between these classes.

Documentation and Style

- Javadoc : In order to encourage good use of block comments and the automatic production of documentation, the Java programming environment comes with a documentation production program called ***javadoc. This program takes a collection of Java source*** files that have been commented using certain keywords, called ***tags, and it produces*** a series of HTML documents that describe the classes, methods, variables, and constants contained in these files.

Documentation rendered by javadoc for the CreditCard.charge method.

Javadoc

- Each javadoc comment is a block comment that starts with "/**" and ends with "*/", and each line between these two can begin with a single asterisk, "*", which is ignored.
- The block comment is assumed to start with a descriptive sentence, which is followed by special lines that begin with javadoc tags. A block comment that comes just before a class definition, instance variable declaration, or method definition is processed by javadoc into a comment about that class, variable, or method. The primary javadoc tags that we use are the following:
  - @author *text: Identifies each author (one per line) for a class.*
  - @throws *exceptionName description: Identifies an error condition that is* signaled by this method.
  - @param *parameterName description: Identifies a parameter accepted by this* method.
  - @return *description: Describes the return type and its range of values for a* method.

Javadoc

```
/**
 * Constructs a new credit card instance.
 * @param cust the name of the customer (e.g., "John Bowman")
 * @param bk the name of the bank (e.g., "California Savings")
 * @param acnt the account identifier (e.g., "5391 0375 9387 5309")
 * @param lim the credit limit (measured in dollars)
 * @param initialBal the initial balance (measured in dollars)
 */
public CreditCard(String cust, String bk, String acnt, int lim, double initialBal) {
      customer = cust;
      bank = bk;
      account = acnt;
      limit = lim;
      balance = initialBal;
}
```

Constructs a new credit card instance.
Parameters:
cust the name of the customer (e.g., "John Bowman")
bk the name of the bank (e.g., "California Savings")
acnt the account identifier (e.g., "5391 0375 9387 5309")
lim the credit limit (measured in dollars)
initialBal the initial balance (measured in dollars)

Readability and Programming Conventions

- Programs should be made easy to read and understand.
- Good programmers should therefore be mindful of **their coding style, and develop a style that communicates the important aspects of a program's design for both humans and computers.** Much has been written about good coding style, with some of the main principles being the following:
- Use meaningful names for identifiers.
  - The tradition in most Java circles is to **capitalize the first letter** of each word in an identifier, **except for the first word for a variable or method name**. By this convention, "Date," "Vector," "DeviceManager" would identify classes, and "isFull( )," "insertItem( )," "studentName," and "studentHeight" would respectively identify methods and variables
- Use named constants or enum types instead of literals. The tradition in Java is to **fully capitalize such constants.**

```
public static final int MIN_CREDITS = 12; // min credits per term
```

Readability and Programming Conventions

- Indent statement blocks.
  - Typically programmers indent each statement block by 4 spaces;
- Organize each class in the following order:
  - Constants
  - Instance variables
  - Constructors
  - Methods
- Use comments that add meaning to a program and explain ambiguous or confusing constructs. In-line comments are good for quick explanations and do not need to be sentences. Block comments are good for explaining the purpose of a method and complex code sections.

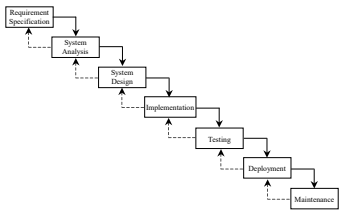
Testing

- A careful testing plan is an essential part of writing a program.
- Make sure that every method of a program is tested at least once (method coverage). Even better, each code statement in the program should be executed at least once (statement coverage).
- Programs often tend to fail on ***special cases of the input***.
  - The array has zero length (no elements).
  - The array has one element.
  - All the elements of the array are the same.
  - The array is already sorted.
  - The array is reverse sorted.

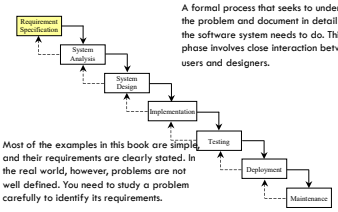
Debuging

- The simplest debugging technique consists of using ***print statements to track the*** values of variables during the execution of the program. A problem with this approach is that eventually the print statements need to be removed or commented out, so they are not executed when the software is finally released.
- A ***debugger, which is a specialized*** environment for controlling and monitoring the execution of a program. The basic functionality provided by a debugger is the insertion of ***breakpoints within*** the code. When the program is executed within the debugger, it stops at each breakpoint. While the program is stopped, the current value of variables can be inspected.
- ***Conditional breakpoints, which are triggered only if a given expression is*** satisfied.
- The standard Java toolkit includes a basic debugger named jdb, which has a command-line interface. Most IDEs for Java programming provide advanced debugging environments with graphical user interfaces.

Software Development Process (Waterfall)



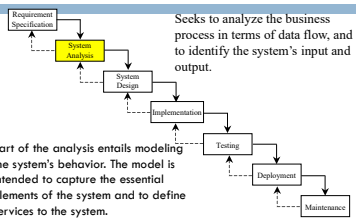
Requirement Specification



A formal process that seeks to understand the problem and document in detail what the software system needs to do. This phase involves close interaction between users and designers.

Most of the examples in this book are simple, and their requirements are clearly stated. In the real world, however, problems are not well defined. You need to study a problem carefully to identify its requirements.

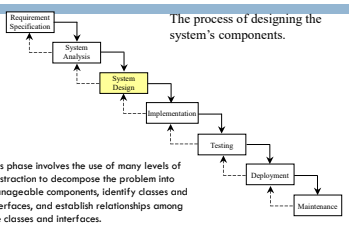
System Analysis



Seeks to analyze the business process in terms of data flow, and to identify the system's input and output.

Part of the analysis entails modeling the system's behavior. The model is intended to capture the essential elements of the system and to define services to the system.

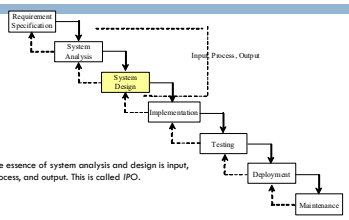
System Design



The process of designing the system's components.

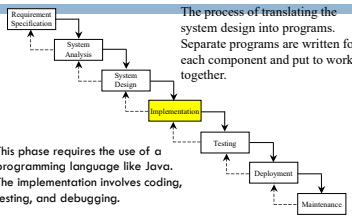
This phase involves the use of many levels of abstraction to decompose the problem into manageable components, identify classes and interfaces, and establish relationships among the classes and interfaces.

IPO



The essence of system analysis and design is input, process, and output. This is called *IPO*.

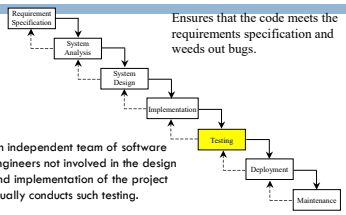
Implementation



The process of translating the system design into programs. Separate programs are written for each component and put to work together.

This phase requires the use of a programming language like Java. The implementation involves coding, testing, and debugging.

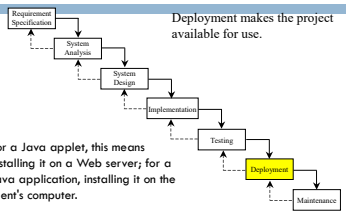
Testing



Ensures that the code meets the requirements specification and weeds out bugs.

An independent team of software engineers not involved in the design and implementation of the project usually conducts such testing.

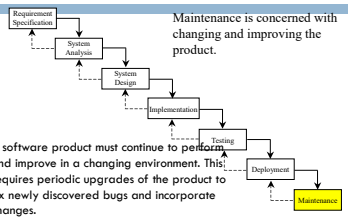
Deployment



Deployment makes the project available for use.

For a Java applet, this means installing it on a Web server; for a Java application, installing it on the client's computer.

Maintenance



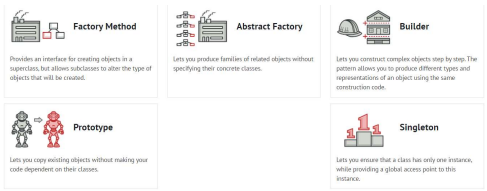
Maintenance is concerned with changing and improving the product.

A software product must continue to perform and improve in a changing environment. This requires periodic upgrades of the product to fix newly discovered bugs and incorporate changes.

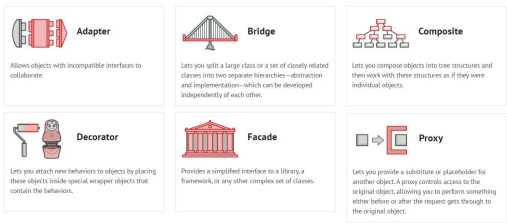
Patterns

- Design patterns are typical solutions to common problems in software design. Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.
- Patterns are a toolkit of solutions to common problems in software design. They define a common language that helps your team communicate more efficiently.
- They are typically categorized into 3 groups:
  - Creational Design Patterns
  - Structural Design Patterns

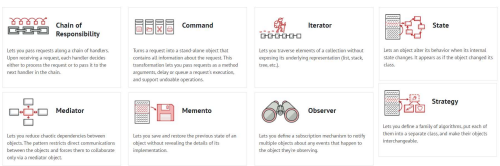
Creational Design Patterns



Structural Design Patterns



Behavioral Design Patterns



Example: Observer Pattern

- What Is the Observer Pattern?
  - Observer is a behavioral design pattern. It specifies communication between objects: **observable and observers. An observable is an object which notifies observers about the changes in its state.**
  - For example, a news agency can notify channels when it receives news. Receiving news is what changes the state of the news agency, and it causes the channels to be notified.

https://www.baeldung.com/java-observer-pattern

- Let's see how we can implement it ourselves.
- First, let's define the NewsAgency class:

```
public class NewsAgency {
      private String news;
      private List<Channel> channels = new ArrayList<>();
      public void addObserver(Channel channel) {
      this.channels.add(channel); }

      public void removeObserver(Channel channel) {
      this.channels.remove(channel); }

      public void setNews(String news) {
      this.news = news;
      for (Channel channel : this.channels)
        {
          channel.update(this.news);
        }
      }
    }
```

NewsAgency is an observable, and when news gets updated, the state of NewsAgency changes. When the change happens, NewsAgency notifies the observers about this fact by calling their update() method.

To be able to do that, the observable object needs to keep references to the observers, and in our case, it's the channels variable.

Let's now see how the observer, the Channel class, can look like. It should have the update() method which is invoked when the state of NewsAgency changes:

```
public class NewsChannel implements Channel {
        private String news;
        @Override
        public void update(Object news) {
        this.setNews((String) news);
        }
    }
```

The Channel interface has only one method:

```
    public interface Channel {
            public void update(Object o);
    }
```

Now, if we add multiple instances of NewsChannel to the list of observers, and change the state of NewsAgency, the instance of NewsChannel will be updated:

```
NewsAgency observable = new NewsAgency();
NewsChannel observer = new NewsChannel();
observable.addObserver(observer1);
observable.addObserver(observer2);
observable.addObserver(observer3);
observable.setNews("news");
```

https://www.baeldung.com/java-observer-pattern

Terminology

- Each **object** created in a program is an **instance** of a **class**.
- Each class presents to the outside world a concise and consistent view of the objects that are instances of this class, without going into too much unnecessary detail or giving others access to the inner workings of the objects.
- The class definition typically specifies **instance variables**, also known as **data member**s, that the object contains, as well as the **methods**, also known as **member functions**, that the object can execute.

Goals

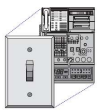
- Robustness
  - We want software to be capable of handling unexpected inputs that are not explicitly defined for its application.
- Adaptability
  - Software needs to be able to evolve over time in response to changing conditions in its environment.
- Reusability
  - The same code should be usable as a component of different systems in various applications.

Object-Oriented Design Principles

- Modularity
- Abstraction
- Encapsulation


Modularity


Abstraction


Encapsulation

Abstract Data Types

- **Abstraction** is to distill a system to its most fundamental parts.
- Applying the abstraction paradigm to the design of data structures gives rise to **abstract data types** (ADTs).
- An ADT is a model of a data structure that specifies the **type** of data stored, the **operations** supported on them, and the types of parameters of the operations.
- An ADT specifies what each operation does, but not how it does it.
- The collective set of behaviors supported by an ADT is its **public interface**.

Encapsulation

- ***Encapsulation:*** *Different* components of a software system should not reveal the internal details of their respective implementations.
- Encapsulation gives one programmer freedom to implement the details of a component, without concern that other programmers will be writing code that intricately depends on those internal decisions.
- The only constraint on the programmer of a component is to maintain the public interface for the component, as other programmers will be writing code that depends on that interface.
- Encapsulation yields robustness and adaptability, for it allows the implementation details of parts of a program to change without adversely affecting other parts, thereby making it easier to fix bugs or add new functionality with relatively local changes to a component.

Modularity

- Modern software systems typically consist of several different components that must interact correctly in order for the entire system to work properly. Keeping these interactions straight requires that these different components be well organized.
- ***Modularity refers to an organizing principle in which different components*** of a software system are divided into separate functional units.
- Robustness is greatly increased because it is easier to test and debug separate components before they are integrated into a larger software system.

Design Patterns

- Object-oriented design facilitates **reusable**, **robust**, and **adaptable** software. Designing good code takes more than simply understanding object-oriented methodologies, however. It requires the effective use of object-oriented design techniques.
- Computing researchers and practitioners have developed a variety of organizational concepts and methodologies for designing quality object-oriented software that is concise, correct, and reusable. They are called **design patterns, which describe solutions to "typical" software design** problems.
- A pattern provides a general template for a solution that can be applied in many different situations. It describes the main elements of a solution in an abstract way that can be specialized for a specific problem at hand. It consists of
  - a name, which identifies the pattern;
  - a context, which describes the scenarios for which this pattern can be applied;
  - a template, which describes how the pattern is applied;
  - and a result, which describes and analyzes what the pattern produces.

Design Patterns

- **Algorithmic patterns:**
- Recursion
- Amortization
- Divide-and-conquer
- Prune-and-search
- Brute force
- Dynamic programming
- The greedy method

- **Software design patterns:**
- Iterator
- Adapter
- Position
- Composition
- Template method
- Locator
- Factory method

Object-Oriented Software Design

- **Responsibilities**: Divide the work into different actors, each with a different responsibility.
- **Independence**: Define the work for each class to be as independent from other classes as possible.
- **Behaviors**: Define the behaviors for each class carefully and precisely, so that the consequences of each action performed by a class will be well understood by other classes that interact with it.

Unified Modeling Language (UML)

A **class diagram** has three portions.

1. The name of the class
2. The recommended instance variables
3. The recommended methods of the class.

class:	CreditCard	
fields:	− customer : String − bank : String − account : String	− limit : int # balance : double
methods:	+ getCustomer( ) : String + getBank( ) : String + charge(price : double) : boolean + makePayment(amount : double)	+ getAccount( ) : String + getLimit( ) : int + getBalance( ) : double

Class Definitions

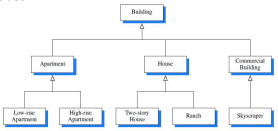
- A class serves as the primary means for abstraction in object-oriented programming.
- In Java, every variable is either a base type or is a reference to an instance of some class.
- A class provides a set of behaviors in the form of member functions (also known as **methods**), with implementations that belong to all its instances.
- A class also serves as a blueprint for its instances, effectively determining the way that state information for each instance is represented in the form of **attributes** (also known as **fields**, **instance variables**, or **data members**).

Constructors

- A user can create an instance of a class by using the **new** operator with a method that has the same name as the class.
- Such a method, known as a **constructor**, has as its responsibility is to establish the state of a newly object with appropriate initial values for its instance variables.

Inheritance

- A mechanism for a modular and hierarchical organization is **inheritance**.
- This allows a new class to be defined based upon an existing class as the starting point.
- The existing class is typically described as the **base class**, parent class, or superclass, while the newly defined class is known as the **subclass** or child class.



An example of an "is a" hierarchy involving architectural buildings.

Inheritance and Constructors

- There are two ways in which a subclass can differentiate itself from its superclass:
  - A subclass may specialize an existing behavior by providing a new implementation that overrides an existing method.
  - A subclass may also extend its superclass by providing brand new methods.
  - Constructors are never inherited in Java; hence, every class must define a constructor for itself
    - All of its fields must be properly initialized, including any inherited fields.
  - The first operation within the body of a constructor must be to invoke a constructor of the superclass, which initializes the fields defined in the superclass.
  - A constructor of the superclass is invoked explicitly by using the keyword **super** with appropriate parameters.
  - If a constructor for a subclass does not make an explicit call to **super** or **this** as its first command, then an implicit call to **super**( ), the zero-parameter version of the superclass constructor, will be made.

Polymorphism and Dynamic Dispatch

- The word **polymorphism literally means "many forms." In the context of object oriented** design, it refers to the ability of a reference variable to take different forms.
- It indicates the ability of a single variable of a given type to be used to reference objects of different types and to automatically call the method tha is specific to the type of object the variable references.

```
class Animal {
  private String type;
  public Animal(String aType) { type = new String(aType);  }
  public String toString() { return "This is a " + type; }
}
```

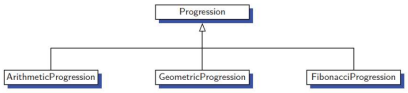
```
class Dog extends Animal {
public Dog(String aType){ super(aType); }
}

class Cat extends Animal {
public Cat(String aName) {
    super("Cat");
    name = aName;
    breed = "Unknown";
  }
  public Cat(String aName, String aBreed) {
    super("Cat");
    name = aName;
    breed = aBreed;
  }
  public String toString() {
    return super.toString() + "\nIt's " + name + " the " + breed;
  }
  private String name;
  private String breed;
}
```

```
public class MainClass {
  public static void main(String[] args) {
    Animal[] theAnimals = { new Dog("Fındık"),
                            new Cat("Minnoş", "Van"),
                            };
    Animal petChoice;

    for (int i = 0; i < 2; i++) {
      petChoice = theAnimals[i];
      System.out.println(petChoice.toString());
    }
  }
}
```

```
This is a Fındık
This is a Cat
It's Minnoş the Van
```

An Extended Example

- A **numeric progression** is a sequence of numbers, where each number depends on one or more of the previous numbers.
  - An **arithmetic progression** determines the next number by adding a fixed constant to the previous value.
  - A **geometric progression** determines the next number by multiplying the previous value by a fixed constant.
  - A **Fibonacci progression** uses the formula $N_{i+1}=N_i+N_{i-1}$



An overview of our hierarchy of progression classes.

The Progression Base Class

```
/** Generates a simple progression. By default: 0, 1, 2, ... */
public class Progression {

  // instance variable
  protected long current;

  /** Constructs a progression starting at zero. */
  public Progression() { this(0); }

  /** Constructs a progression with given start value. */
  public Progression(long start) { current = start; }

  /** Returns the next value of the progression. */
  public long nextValue() {
    long answer = current;
    advance();     // this protected call is responsible for advancing the current value
    return answer;
  }
```

The Progression Base Class, 2

```
  /** Advances the current value to the next value of the progression. */
  protected void advance() {
    current++;
  }

  /** Prints the next n values of the progression, separated by spaces. */
  public void printProgression(int n) {
    System.out.print(nextValue());          // print first value without leading space
    for (int j=1; j < n; j++)
      System.out.print(" " + nextValue()); // print leading space before others
    System.out.println();                   // end the line
  }
}
```

ArithmeticProgression Subclass

```
public class ArithmeticProgression extends Progression {

  protected long increment;

  /** Constructs progression 0, 1, 2, ... */
  public ArithmeticProgression() { this(1, 0); }      // start at 0 with increment of 1

  /** Constructs progression 0, stepsize, 2*stepsize, ... */
  public ArithmeticProgression(long stepsize) { this(stepsize, 0); }      // start at 0

  /** Constructs arithmetic progression with arbitrary start and increment. */
  public ArithmeticProgression(long stepsize, long start) {
    super(start);
    increment = stepsize;
  }

  /** Adds the arithmetic increment to the current value. */
  protected void advance() {
    current += increment;
  }
}
```

GeometricProgression Subclass

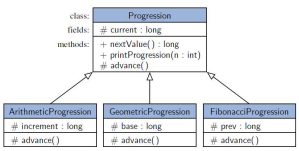
```
public class GeometricProgression extends Progression {

  protected long base;

  /** Constructs progression 1, 2, 4, 8, 16, ... */
  public GeometricProgression() { this(2, 1); }          // start at 1 with base of 2

  /** Constructs progression 1, b, b^2, b^3, b^4, ... for base b. */
  public GeometricProgression(long b) { this(b, 1); }                    // start at 1

  /** Constructs geometric progression with arbitrary base and start. */
  public GeometricProgression(long b, long start) {
    super(start);
    base = b;
  }

  /** Multiplies the current value by the geometric base. */
  protected void advance() {
    current *= base;                    // multiply current by the geometric base
  }
}
```

FibonacciProgression Subclass

```
public class FibonacciProgression extends Progression {

  protected long prev;

  /** Constructs traditional Fibonacci, starting 0, 1, 1, 2, 3, ... */
  public FibonacciProgression() { this(0, 1); }

  /** Constructs generalized Fibonacci, with give first and second values. */
  public FibonacciProgression(long first, long second) {
    super(first);
    prev = second - first;      // fictitious value preceding the first
  }

  /** Replaces (prev,current) with (current, current+prev). */
  protected void advance() {
    long temp = prev;
    prev = current;
    current += temp;
  }
}
```

Progression

- Inheritance diagram for class Progression and its subclasses



```
/** Test program for the progression hierarchy. */
public class TestProgression {
    public static void main(String[ ] args) {
    Progression prog;
    // test ArithmeticProgression
    System.out.print("Arithmetic progression with default increment: ");
    prog = new ArithmeticProgression( );
    prog.printProgression(10);
    System.out.print("Arithmetic progression with increment 5: ");
    prog = new ArithmeticProgression(5);
    prog.printProgression(10);
    System.out.print("Arithmetic progression with start 2: ");
    prog = new ArithmeticProgression(5, 2);
    prog.printProgression(10);
    // test GeometricProgression
    System.out.print("Geometric progression with default base: ");
    prog = new GeometricProgression( );
    prog.printProgression(10);
    System.out.print("Geometric progression with base 3: ");
    prog = new GeometricProgression(3);
    prog.printProgression(10);
    // test FibonacciProgression
    System.out.print("Fibonacci progression with default start values: ");
    prog = new FibonacciProgression( );
    prog.printProgression(10);
    }
}
```

Interfaces and Abstract Classes

- In order for two objects to interact, they must "know" about the various messages that each will accept, that is, the methods each object supports. To enforce this "knowledge," the object-oriented design paradigm asks that classes specify the ***application programming interface (API), or simply interface, that their objects*** present to other objects.
- The main structural element in Java that enforces an application programming interface is an **interface**.
- An interface is a collection of method declarations with no data and no bodies.
- Interfaces do not have constructors and they cannot be directly instantiated.
  - When a class **implements** an interface, it must implement all of the methods declared in the interface.
- An abstract class also cannot be instantiated, but it can define one or more common methods that all implementations of the abstraction will have.

Ex. Interface

```
public interface Sellable {
/** Returns a description of the object. */
public String description( );

/** Returns the list price in cents. */
public int listPrice( );

/** Returns the lowest price in cents we will accept. */
public int lowestPrice( );
}
```

Ex. Interface

```
/** Class for photographs that can be sold. */
public class Photograph implements Sellable {
    private String descript; // description of this photo
    private int price; // the price we are setting
    private boolean color; // true if photo is in color

    public Photograph(String desc, int p, boolean c) {
    // constructor
     descript = desc;
     price = p;
     color = c;
     }

    public String description() { return descript; }

    public int listPrice() {return price; }

    public int lowestPrice() { return price/2;}
}
```

Multiple Inheritance

- In Java, multiple inheritance is allowed for interfaces but not for classes. The reason for this rule is that interfaces do not define fields or method bodies, yet classes typically do.

Ex. Multiple Inheritance

```
public interface Transportable {
/** Returns the weight in grams. */
public int weight( );

/** Returns whether the object is hazardous. */
public boolean isHazardous( );
}
```

Ex. Interface

```
public class BoxedItem implements Sellable, Transportable {
    @Override
    public int weight() {
        // TODO Auto-generated method stub
        return 0;}
    @Override
    public boolean isHazardous() {
        // TODO Auto-generated method stub
        return false;}
    @Override
    public String description() {
        // TODO Auto-generated method stub
        return null;}
    @Override
    public int listPrice() {
        // TODO Auto-generated method stub
        return 0;}
    @Override
    public int lowestPrice() {
        // TODO Auto-generated method stub
        return 0;}
}
```

Abstract Classes

- In Java, an ***abstract class serves a role somewhat between that of a traditional class*** and that of an interface.
- Like an interface, an abstract class may define signatures for one or more methods without providing an implementation of those method bodies; such methods are known as ***abstract methods.***
- ***However, unlike an interface,*** an abstract class may define one or more fields and any number of methods with implementation (so-called ***concrete methods). An abstract class may also extend*** another class and be extended by further subclasses.

Ex. Abstract Class

```
public abstract class Employee {
    private String name; // Employee name
    private String address; // Employee Address
    private int SSN;  // Social Security Number

    // Abstract method does not have a body
    public abstract double computePay();

    // method with body
    public String getName() { return name; };
}
```

Exceptions

- Exceptions are unexpected events that occur during the execution of a program.
- An exception might result due to an unavailable resource, unexpected input from a user, or simply a logical error on the part of the programmer.
- In Java, exceptions are objects that can be **thrown** by code that encounters an unexpected situation.
- An exception may also be **caught** by a surrounding block of code that "handles" the problem.
- If uncaught, an exception causes the virtual machine to stop executing the program and to report an appropriate message to the console.

Catching Exceptions

- The general methodology for handling exceptions is a **try-catch** construct in which a guarded fragment of code that might throw an exception is executed.

```
try {
    guardedBody
} catch (exceptionType1 variable1) {
    remedyBody1
} catch (exceptionType2 variable2) {
    remedyBody2
} ...
```

- If it **throws** an exception, then that exception is caught by having the flow of control jump to a predefined **catch** block that contains the code to apply an appropriate resolution.
- If no exception occurs in the guarded code, all **catch** blocks are ignored.

Ex. Exceptions

```
public class ExceptionExample {
    public static final int DEFAULT=5;
    public static void main(String[] args) {
        int n = DEFAULT;
        try {
        n = Integer.parseInt(args[0]);
        if (n <= 0) {
            System.out.println("n must be positive. Using default.");
            n = DEFAULT;
        }
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("No argument specified for n. Using default.");
        } catch (NumberFormatException e) {
            System.out.println("Invalid integer argument. Using default.");
        }
    }
}
```

Throwing Exceptions

- Exceptions originate when a piece of Java code finds some sort of problem during execution and throws an exception object.
- This is done by using the **throw** keyword followed by an instance of the exception type to be thrown.
- It is often convenient to instantiate an exception object at the time the exception has to be thrown. Thus, a throw statement is typically written as follows:

throw new exceptionType(parameters);

where exceptionType is the type of the exception and the parameters are sent to that type's constructor.

Ex. Throwing Exception

```
public void ensurePositive(int n) {
if (n < 0)
        throw new IllegalArgumentException("That's not positive!");
// ...
}
```

The throws Clause

- When a method is declared, it is possible to explicitly declare, as part of its signature, the possibility that a particular exception type may be thrown during a call to that method.
- The syntax for declaring possible exceptions in a method signature relies on the keyword **throws** (not to be confused with an actual **throw** statement).
- For example, the parseInt method of the Integer class has the following formal signature:

```
public static int parseInt(String s) throws NumberFormatException
{
    // ...
    return 0;
}
```

Casting

- Casting with Objects allows for conversion between classes and subclasses.
- A **widening conversion** occurs when a type T is converted into a "wider" type U:
  - T and U are class types and U is a superclass of T.
  - T and U are interface types and U is a superinterface of T.
  - T is a class that implements interface U.
- Example:

```
CreditCard card = new PredatoryCreditCard(...);
```

Narrowing Conversions

- A **narrowing conversion** occurs when a type T is converted into a "narrower" type S.
  - T and S are class types and S is a subclass of T.
  - T and S are interface types and S is a subinterface of T.
  - T is an interface implemented by class S.
- In general, a narrowing conversion of reference types requires an explicit cast.
- Example:

```
CreditCard card = new PredatoryCreditCard(...); // widening
PredatoryCreditCard pc = (PredatoryCreditCard) card; // narrowing
```

Generics

- Java includes support for writing generic classes and methods that can operate on a variety of data types while often avoiding the need for explicit casts.
- The generics framework allows us to define a class in terms of a set of formal type parameters, which can then be used as the declared type for variables, parameters, and return values within the class definition.
- Those formal type parameters are later specified when using the generic class as a type elsewhere in a program.

Syntax for Generics

- Types can be declared using generic names:

```
public class Pair<A,B> {
  A first;
  B second;
  public Pair(A a, B b) {                    // constructor
    first = a;
    second = b;
  }
  public A getFirst( ) { return first; }
  public B getSecond( ) { return second;}
}
```

- They are then instantiated using actual types:

```
Pair<String,Double> bid;
```


Nested Classes

- Java allows a class definition to be nested inside the definition of another class.
- The main use for nesting classes is when defining a class that is strongly affiliated with another class.
  - This can help increase encapsulation and reduce undesired name conflicts.
- Nested classes are a valuable technique when implementing data structures, as an instance of a nested use can be used to represent a small portion of a larger data structure, or an auxiliary class that helps navigate a primary data structure.

Ex. Nested Class

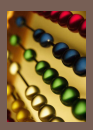
- The Java programming language allows you to define a class within another class. Such a class is called a *nested class*.
- Nested classes are divided into two categories: static and non-static.

```
public class CreditCard {
[private] [static] class Transaction {/* details omitted */ }

// instance variable for a CreditCard

Transaction[] history; // keep log of all transactions for
this card

}
```


HW.

- R-2.14 Give an example of a Java code fragment that performs an array reference that is possibly out of bounds, and if it is out of bounds, the program catches that exception and prints the following error message: "Don't try buffer overflow attacks in Java!"
- Give two examples of Java code framents for Nested Static and Nested Non-Static classes.

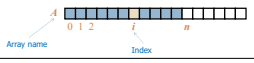
ARRAYS



1

Array Definition

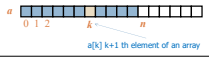
- An ***array*** is a sequenced collection of variables all of the same type. Each variable, or ***cell***, in an array has an ***index***, which uniquely refers to the value stored in that cell. The cells of an array, A, are numbered 0, 1, 2, and so on.
- Each value stored in an array is often called an ***element*** of that array.



2

Array Length and Capacity

- Since the length of an array determines the maximum number of things that can be stored in the array, we will sometimes refer to the length of an array as its ***capacity***.
- In Java, the length of an array named *a* can be accessed using the syntax a.length. Thus, the cells of an array, a, are numbered 0, 1, 2, and so on, up through a.length−1, and the cell with index *k* can be accessed with syntax *a*[*k*].



3

Declaring Arrays (first way)

- The first way to create an array is to use an assignment to a literal form when initially declaring the array, using a syntax as:

$$elementType[]\ arrayName = \{initialValue_0, initialValue_1, \ldots, initialValue_{N-1}\};$$

- The *elementType* can be any Java base type or class name, and *arrayName* can be any valid Java identifier. The initial values must be of the same type as the array.

4

Ex. Arrays

- Dot product.

```
double[] x = { 0.3, 0.6, 0.1 };   // declare an array x
double[] y = { 0.5, 0.1, 0.4 };   // declare second array y
int N = x.length;         // length of array x
double sum = 0.0;
for (int i = 0; i < N; i++) {
    sum = sum + x[i]*y[i];
}
```

5

Declaring Arrays (second way)

- The second way to create an array is to use the **new** operator.
  - However, because an array is not an instance of a class, we do not use a typical constructor. Instead we use the syntax:

new *elementType*[*length*]

- *length* is a positive integer denoting the length of the new array.
- The **new** operator returns a reference to the new array, and typically this would be assigned to an array variable.

6

Ex. Arrays

- An array of size 1000000.

```
// scales to handle large arrays
double[] a = new double[1000000];
…
a[123456] = 3.0;
…
a[987654] = 8.0;
…
double x = a[123456] + a[987654];
```

declares, creates, and initializes

7

Ex. Arrays

```
int N = 10; // size of array
double[] a; // declare the array
a = new double[N]; // create the array
for (int i = 0; i < N; i++) // initialize the array
a[i] = 0.0; // all to 0.0
```

Compact alternative:

```
int N = 10; // size of array
double[] a = new double[N];  // declare create init
```

8

Exercise?

- Find maximum of the array values
- Reverse the elements within an array

9

Ex. Arrays

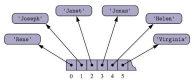
Create an array with random values	`double [] a = new double [N]; for (int i = 0; i < N; i++ ) a[i] = Math.random();`
Print the array values one per line	`for (int i = 0; i < N; i++ ) System.out.println(a[i]);`
Find maximum of the array values	`double max = Double.NEGATIVE_INFINITY; for (int i = 0; i < N; i++ ) if (a[i]>max) max = a[i];`
Reverse the elements within an array	`for (int i = 0; i < N / 2; i++ ) { double temp = b[i]; b[i]=b[N-1-i]; b[N-i-1] = temp; }`

10

Arrays of Characters or Object References

- An array can store primitive elements, such as characters.

- An array can also store references to objects.



Ex. Arrays

- Print a random card.

```
String[] rank = { "2", "3", "4", "5", "6", "7", "8", "9",
"10", "Jack", "Queen", "King", "Ace" };

String[] suit = { "Clubs", "Diamonds", "Hearts", "Spades" };

int i = (int) (Math.random() * 13); // between 0 and 12
int j = (int) (Math.random() * 4); // between 0 and 3

System.out.println(rank[i] + " of " + suit[j]);
```


Ex. Arrays

Matrix Addition	`double[][] c = new double[N][N];`
	`for (int i = 0; i < N; i++)`
	`  for (int j = 0; j < N; j++)`
	`  c[i][j] = a[i][j] + b[i][j];`

Matrix Multiplication	`double[][] c = new double[N][N];`
	`for (int i = 0; i < N; i++)`
	`  for (int j = 0; j < N; j++)`
	`  for (int k = 0; k < N; k++)`
	`  c[i][j] += a[i][k] * b[k][j];`

Java Example: Game Entries

A game entry stores the name of a player and her best score so far in a game

```
public class GameEntry {
private String name; // name of the person earning this score
private int score; // the score value
/** Constructs a game entry with given parameters.. */
public GameEntry(String n, int s) {
    name = n;
    score = s;
}
/** Returns the name field. */
public String getName( ) { return name; }
/** Returns the score field. */
public int getScore( ) { return score; }
/** Returns a string representation of this entry. */
public String toString( ) {
    return "(" + name + ", " + score + ")";
}
}
```

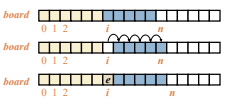

Java Example: Scoreboard

- Keep track of players and their best scores in an array, board
  - The elements of board are objects of class GameEntry
  - Array board is sorted by score

```
/** Class for storing high scores in an array in nondecreasing order.
*/
public class Scoreboard {
private int numEntries = 0; // number of actual entries
private GameEntry[ ] board; // array of game entries (names & scores)
/** Constructs an empty scoreboard with the given capacity for storing
entries. */
public Scoreboard(int capacity) {
  board = new GameEntry[capacity];
}
// more methods will go here
}
```


Adding an Entry

- To add an entry e into array board at index i, we need to make room for it by shifting forward the $n - i$ entries $board[i], \ldots, board[n-1]$

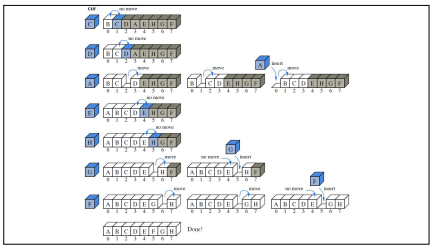


Java Example

```
/** Attempt to add a new score to the collection (if it is high enough) */
public void add(GameEntry e) {
int newScore = e.getScore( );
// is the new entry e really a high score?
if (numEntries < board.length || newScore > board[numEntries-1].getScore( ))
{
  if (numEntries < board.length) // no score drops from the board
    numEntries++; // so overall number increases
  // shift any lower scores rightward to make room for the new entry
  int j = numEntries - 1;
  while (j > 0 && board[j-1].getScore( ) < newScore) {
    board[j] = board[j-1]; // shift entry from j-1 to j
    j--; // and decrement j
  }
  board[j] = e; // when done, add new entry
}
}
```

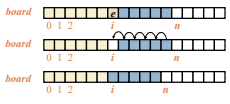

Java Example

```
/** Remove and return the high score at index i. */
public GameEntry remove(int i) throws IndexOutOfBoundsException {
if (i < 0 || i >= numEntries)
    throw new IndexOutOfBoundsException("Invalid index: " + i);
GameEntry temp = board[i]; // save the object to be removed
for (int j = i; j < numEntries - 1; j++) // count up from i (not down)
  board[j] = board[j+1]; // move one cell to the left
  board[numEntries -1 ] = null; // null out the old last score
  numEntries--;
return temp; // return the removed object
}
```

Removing an Entry

- To remove the entry e at index i, we need to fill the hole left by e by shifting backward the $n - i - 1$ elements $board[i+1], \ldots, board[n-1]$



Sorting an Array

- Insertion-Sort Algorithm

Algorithm **InsertionSort(A):**
Input: An array A of n comparable elements
Output: The array A with elements rearranged in nondecreasing order

for k from 1 to n-1 do
  Insert $A[k]$ *at its proper location within* $A[0], A[1], \ldots, A[k]$.

Insertion-Sort Algorithm

```
/** Insertion-sort of an array of characters into nondecreasing order */
public static void insertionSort(char[ ] data) {
int n = data.length;
for (int k = 1; k < n; k++) { // begin with second character

   char cur = data[k]; // time to insert cur=data[k]
   int j = k; // find correct index j for cur
   while (j > 0 && data[j-1] > cur) {
   // thus, data[j-1] must go after cur
      data[j] = data[j-1]; // slide data[j-1] rightward
      j--; // and consider previous j for cur
   }
   data[j] = cur; // this is the proper place for cur

}
}
```


java.util Methods for Arrays

equals(A, B)	Returns true if and only if the array A and the array B are equal. Two arrays are considered equal if they have the same number of elements and every corresponding pair of elements in the two arrays are equal. That is, A and B have the same values in the same order.
fill(A, x)	Stores value x in every cell of array A, provided the type of array A is defined so that it is allowed to store the value x.
copyOf(A, n)	Returns an array of size n such that the first k elements of this array are copied from A, where k=min{n,A.length}. If n > A.length, then the last n−A.length elements in this array will be padded with default values, e.g., 0 for an array of int and null for an array of objects.
copyOfRange(A, s, f)	Returns an array of size t ≈ such that the elements of this array are copied in order from A[s] to A[t−1], where s <= t, padded as with copyOf( ) if t > A.length.

Random Numbers

- PseudoRandom Number Generation
  - Another feature built into Java, which is often useful when testing programs dealing with arrays, is the ability to generate pseudorandom numbers, that is, numbers that appear to be random (but are not necessarily truly random). In particular, Java has a built-in class, java.util.Random, whose instances are ***pseudorandom number generators, that is, objects that compute a sequence of numbers that are statistically*** random.

next = (a * cur + b) % n
where a, b, and n are appropriately chosen integers, and % is the modulus operator. Something along these lines is, in fact, the method used by java.util.Random objects, with $n = 2^{48}$

Multidimensional Array

- 2+ dimensional arrays are similar to the matrix representation.
- Each element can be accessed as a[i] [j]

```
int M = 10;
int N = 3;

double[][] a = new double[M][N];

for (int i = 0; i < M; i++) {
  for (int j = 0; j < N; j++) {
    a[i][j] = 0.0;
  }
}
```

2d array decleration

assign value to i, j th element

java.util Methods for Arrays

toString(A)	Returns a String representation of the array A, beginning with [, ending with], and with elements of A displayed separated by string ", ". The string representation of an element A[i] is obtained using String.valueOf(A[i]), which returns the string "null" for a null reference and otherwise calls A[i].toString( ).
sort(A)	Sorts the array A based on a natural ordering of its elements, which must be comparable.
binarySearch(A, x)	Searches the sorted array A for value x, returning the index where it is found, or else the index of where it could be inserted while maintaining the sorted order.

As static methods, these are invoked directly on the java.util.Arrays class, not on a particular instance of the class. For example, if data were an array, we could sort it with syntax, java.util.Arrays.sort(data), or with the shorter syntax Arrays.sort(data) if we first import the Arrays class.

java.util.Random Methods

nextBoolean( )	Returns the next pseudorandom boolean value.
nextDouble( )	Returns the next pseudorandom double value, between 0.0 and 1.0.
nextInt( )	Returns the next pseudorandom int value.
nextInt(n)	Returns the next pseudorandom int value in the range from 0 up to but not including n.
setSeed(s)	Sets the seed of this pseudorandom number generator to the long s.

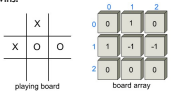
Declaring 2D Arrays

- Initializing 2D arrays by listing values.

```
double[][] p = {
   { .02, .92, .02, .02, .02 },
   { .02, .02, .32, .32, .32 },
   { .02, .02, .02, .92, .02 },
   { .92, .02, .02, .02, .02 },
   { .47, .02, .47, .02, .02 },
};
```

Two-Dimensional Arrays and Positional Games (Tic-Tac-Toe)

- Two players—X and O—alternate in placing their respective marks in the cells of this board, starting with player X. If either player succeeds in getting three of his or her marks in a row, column, or diagonal, then that player wins.



0 indicating an empty cell, a 1 indicating an X, and a −1 indicating an O. if the values of a row, column, or diagonal add up to 3 or −3, respectively, there is a win.

29

```
/** Checks whether the board configuration is a win for the given player. */
public boolean isWin(int mark) {
    return ((board[0][0] + board[0][1] + board[0][2] == mark*3) // row 0
    || (board[1][0] + board[1][1] + board[1][2] == mark*3) // row 1
    || (board[2][0] + board[2][1] + board[2][2] == mark*3) // row 2
    || (board[0][0] + board[1][0] + board[2][0] == mark*3) // column 0
    || (board[0][1] + board[1][1] + board[2][1] == mark*3) // column 1
    || (board[0][2] + board[1][2] + board[2][2] == mark*3) // column 2
    || (board[0][0] + board[1][1] + board[2][2] == mark*3) // diagonal
    || (board[2][0] + board[1][1] + board[0][2] == mark*3)); // rev diag
}
/** Returns the winning player's code, or 0 to indicate a tie (or unfinished game).*/
public int winner( ) {
    if (isWin(X))
        return(X);
    else if (isWin(O))
        return(O);
    else
        return(0);
}
```

31

```
/** Test run of a simple game */
public static void main(String[ ] args) {
    TicTacToe game = new TicTacToe( );
    /* X moves: */ /* O moves: */
    game.putMark(1,1); game.putMark(0,2);
    game.putMark(2,2); game.putMark(0,0);
    game.putMark(0,1); game.putMark(2,1);
    game.putMark(1,2); game.putMark(1,0);
    game.putMark(2,0);
    System.out.println(game);
    int winningPlayer = game.winner( );
    String[ ] outcome = {"O wins", "Tie", "X wins"}; // rely on ordering
    System.out.println(outcome[1 + winningPlayer]);
  }
} // end of class
```

```
O|X|O
-----
O|X|X
-----
X|O|X
Tie
```

33

```
/** Simulation of a Tic-Tac-Toe game (does not do strategy). */
public class TicTacToe {
    public static final int X = 1, O = -1; // players
    public static final int EMPTY = 0; // empty cell
    private int board[ ][ ] = new int[3][3]; // game board
    private int player; // current player
    /** Constructor */
    public TicTacToe( ) { clearBoard( ); }
    /** Clears the board */
    public void clearBoard( ) {
        for (int i = 0; i < 3; i++)
            for (int j = 0; j < 3; j++)
                board[i][j] = EMPTY; // every cell should be empty
                player = X; // the first player is 'X'
    }
    /** Puts an X or O mark at position i,j. */
    public void putMark(int i, int j) throws IllegalArgumentException {
        if ((i < 0) || (i > 2) || (j < 0) || (j > 2))
            throw new IllegalArgumentException("Invalid board position");
        if (board[i][j] != EMPTY)
            throw new IllegalArgumentException("Board position occupied");
        board[i][j] = player; // place the mark for the current player
        player = - player; // switch players (uses fact that O = - X)
    } // Code continues on the next page
```

30

```
/** Returns a simple character string showing the current board. */
public String toString( ) {
    StringBuilder sb = new StringBuilder( );
    for (int i=0; i<3; i++) {
        for (int j=0; j<3; j++) {
            switch (board[i][j]) {
                case X: sb.append("X"); break;
                case O: sb.append("O"); break;
                case EMPTY: sb.append(" "); break;
            }
            if (j < 2) sb.append("|"); // column boundary
        }
        if (i < 2) sb.append("\n-----\n"); // row boundary
    }
    return sb.toString( );
}
/** Test run of a simple game */
```

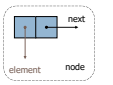
32

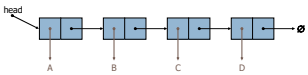
SINGLY LINKED LISTS

34

Singly Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes, starting from a head pointer
- Each node stores
  - element
  - link to the next node





35

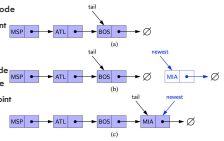
Accessor Methods

```
public class SinglyLinkedList<E> {
  (nested Node class goes here)
  // instance variables of the SinglyLinkedList
  private Node<E> head = null;          // head node of the list (or null if empty)
  private Node<E> tail = null;          // last node of the list (or null if empty)
  private int size = 0;                 // number of nodes in the list
  public SinglyLinkedList( ) { }        // constructs an initially empty list
  // access methods
  public int size( ) { return size; }
  public boolean isEmpty( ) { return size == 0; }
  public E first( ) {                   // returns (but does not remove) the first element
    if (isEmpty( )) return null;
    return head.getElement( );
  }
  public E last( ) {                    // returns (but does not remove) the last element
    if (isEmpty( )) return null;
    return tail.getElement( );
  }
```

37

Inserting at the Tail

- Allocate a new node
- Insert new element
- Have new node point to null
- Have old last node point to new node
- Update tail to point to new node



39

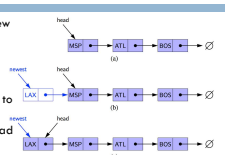
A Nested Node Class

```
public class SinglyLinkedList<E> {
  //---------- nested Node class ----------
  private static class Node<E> {
    private E element;               // reference to the element stored at this node
    private Node<E> next;            // reference to the subsequent node in the list
    public Node(E e, Node<E> n) {
      element = e;
      next = n;
    }
    public E getElement( ) { return element; }
    public Node<E> getNext( ) { return next; }
    public void setNext(Node<E> n) { next = n; }
  } //---------- end of nested Node class ----------
  ... rest of SinglyLinkedList class will follow ...
```

36

Inserting at the Head

- Allocate new node
- Insert new element
- Have new node point to old head
- Update head to point to new node



38

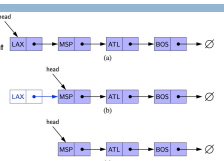
Java Methods

```
public void addFirst(E e) {          // adds element e to the front of the list
  head = new Node<>(e, head);        // create and link a new node
  if (size == 0)
    tail = head;                     // special case: new node becomes tail also
  size++;
}
public void addLast(E e) {           // adds element e to the end of the list
  Node<E> newest = new Node<>(e, null);   // node will eventually be the tail
  if (isEmpty())
    head = newest;                   // special case: previously empty list
  else
    tail.setNext(newest);            // new node after existing tail
  tail = newest;                     // new node becomes the tail
  size++;
}
```

40

Removing at the Head

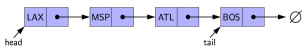
- Update head to point to next node in the list
- Allow garbage collector to reclaim the former first node



41

Removing at the Tail

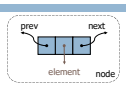
- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node

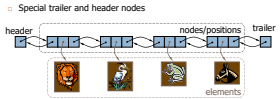


43

Doubly Linked List

- A doubly linked list can be traversed forward and backward
- Nodes store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





45

Java Method

```
public E removeFirst( ) {              // removes and returns the first element
  if (isEmpty( )) return null;         // nothing to remove
  E answer = head.getElement( );
  head = head.getNext( );              // will become null if list had only one node
  size--;
  if (size == 0)                       // special case as list is now empty
    tail = null;
  return answer;
}
```

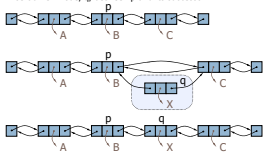
42

DOUBLY LINKED LISTS

44

Insertion

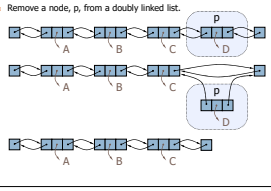
- Insert a new node, q, between p and its successor.



46

Deletion

- Remove a node, p, from a doubly linked list.



47

Doubly-Linked List in Java, 2

```
private Node<E> header;                  // header sentinel
private Node<E> trailer;                 // trailer sentinel
private int size = 0;                    // number of elements in the list
/** Constructs a new empty list. */
public DoublyLinkedList() {
  header = new Node<>(null, null, null);     // create header
  trailer = new Node<>(null, header, null);  // trailer is preceded by header
  header.setNext(trailer);                   // header is followed by trailer
}
/** Returns the number of elements in the linked list. */
public int size() { return size; }
/** Tests whether the linked list is empty. */
public boolean isEmpty() { return size == 0; }
/** Returns (but does not remove) the first element of the list. */
public E first() {
  if (isEmpty()) return null;
  return header.getNext().getElement();      // first element is beyond header
}
/** Returns (but does not remove) the last element of the list. */
public E last() {
  if (isEmpty()) return null;
  return trailer.getPrev().getElement();     // last element is before trailer
}
```

49

Doubly-Linked List in Java, 4

```
  // private update methods
  /** Adds element e to the linked list in between the given nodes. */
  private void addBetween(E e, Node<E> predecessor, Node<E> successor) {
    // create and link a new node
    Node<E> newest = new Node<>(e, predecessor, successor);
    predecessor.setNext(newest);
    successor.setPrev(newest);
    size++;
  }
  /** Removes the given node from the list and returns its element. */
  private E remove(Node<E> node) {
    Node<E> predecessor = node.getPrev();
    Node<E> successor = node.getNext();
    predecessor.setNext(successor);
    successor.setPrev(predecessor);
    size--;
    return node.getElement();
  }
} //---------- end of DoublyLinkedList class ----------
```

51

Doubly-Linked List in Java

```
/** A basic doubly linked list implementation. */
public class DoublyLinkedList<E> {
  //---------- nested Node class ----------
  private static class Node<E> {
    private E element;               // reference to the element stored at this node
    private Node<E> prev;            // reference to the previous node in the list
    private Node<E> next;            // reference to the subsequent node in the list
    public Node(E e, Node<E> p, Node<E> n) {
      element = e;
      prev = p;
      next = n;
    }
    public E getElement() { return element; }
    public Node<E> getPrev() { return prev; }
    public Node<E> getNext() { return next; }
    public void setPrev(Node<E> p) { prev = p; }
    public void setNext(Node<E> n) { next = n; }
  } //---------- end of nested Node class ----------
```

48

Doubly-Linked List in Java, 3

```
  // public update methods
  /** Adds element e to the front of the list. */
  public void addFirst(E e) {
    addBetween(e, header, header.getNext());     // place just after the header
  }
  /** Adds element e to the end of the list. */
  public void addLast(E e) {
    addBetween(e, trailer.getPrev(), trailer);   // place just before the trailer
  }
  /** Removes and returns the first element of the list. */
  public E removeFirst() {
    if (isEmpty()) return null;                  // nothing to remove
    return remove(header.getNext());             // first element is beyond header
  }
  /** Removes and returns the last element of the list. */
  public E removeLast() {
    if (isEmpty()) return null;                  // nothing to remove
    return remove(trailer.getPrev());            // last element is before trailer
  }
```

50

HW.

- P-2.31 Write a Java program to simulate an ecosystem containing two types of creatures, ***bears and fish. The ecosystem consists of a river, which is modeled as a relatively*** large array. Each cell of the array should contain an Animal object, which can be a Bear object, a Fish object, or null. In each time step, based on a random process, each animal either attempts to move into an adjacent array cell or stay where it is. If two animals of the same type are about to collide in the same cell, then they stay where they are, but they create a new instance of that type of animal, which is placed in a random empty (i.e., previously null) cell in the array. If a bear and a fish collide, however, then the fish dies (i.e., it disappears). Use actual object creation, via the new operator, to model the creation of new objects, and provide a visualization of the array after each time step.

52

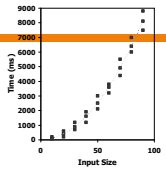
24 25 26

ANALYSIS OF ALGORITHMS

1

Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition, noting the time needed:
- Plot the results




```
long startTime = System.currentTimeMillis( ); // record the starting time
/* (run the algorithm) */
long endTime = System.currentTimeMillis( ); // record the ending time
long elapsed = endTime - startTime; // compute the elapsed time
```

3

Ex. Experimental Studies

n	Repeat1 (ms)	Repeat 2 (ms)
50.000	2.884	1
100.000	7.437	1
200.000	39.158	2
400.000	170.173	3
800.000	690.836	7
1.600.000	2.874.968	13
3.200.000	12.809.631	28
6.400.000	59.594.275	58
12.800.000	265.696.421	135

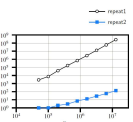


Chart of the results of the timing experiment, displayed on a log-log scale. The divergent slopes demonstrate an order of magnitude difference in the growth of the running times.

5

Analysis of Algorithms

- Typically, the primary analysis tool involves characterizing the running times of algorithms and data structure operations, with space usage also being of interest.
- Running time is a natural measure of "goodness," since time is a precious resource - computer solutions should run as fast as possible.

2

Ex. Experimental Studies

- Two algorithms for constructing long strings in Java.

```
/** Uses repeated concatenation to compose a String with n copies of character c. */
public static String repeat1(char c, int n) {
    String answer = "";
    for (int j=0; j < n; j++)
      answer += c;
    return answer;
}
/** Uses StringBuilder to compose a String with n copies of character c. */
public static String repeat2(char c, int n) {
    StringBuilder sb = new StringBuilder( );
    for (int j=0; j < n; j++)
      sb.append(c);
    return sb.toString( );
}
```

4

HW.

- Implement repeat1 algorithm on Java, Matlab and C# and compare the results for n=1000, 10000 to 10.000.000 various iterations. Write a report about possible reasons of the performance differences.

```
tic
N=100000;
s='';
for i=0:N
    s=s + 'c';
end
a=toc
```

```
using System;
namespace ConsoleApplication2
{
    class Program
    {
        static void Main(string[] args)
        {
            int N = 10000; string s = "";
            DateTime dt = DateTime.Now;
            for (int i = 0; i < N; i++)
                s = s + "c";
            TimeSpan ts = DateTime.Now - dt;
Console.WriteLine(ts.TotalMilliseconds.ToString());
            Console.ReadKey();
        }
    }
}
```

6

1 2 3

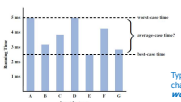
Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used

7

Measuring Operations as a Function of Input Size

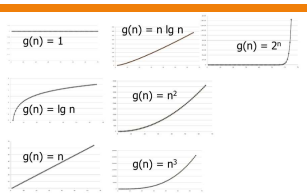
- To capture the order of growth of an algorithm's running time, we will associate, with each algorithm, a function $f(n)$ *that characterizes the number of primitive* operations that are performed as a function of the input size n.



Typically, running times are characterized in terms of the ***worst case !!!!***

9

Functions Graphed Using "Normal" Scale



11

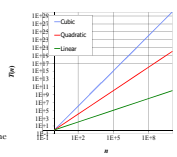
Moving Beyond Experimental Analysis

- Goal is to develop an approach to analyzing the efficiency of algorithms that:
  - Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent of the hardware and software environment.
  - Is performed by studying a high-level description of the algorithm without need for implementation.
  - Takes into account all possible inputs.

8

Seven Important Functions

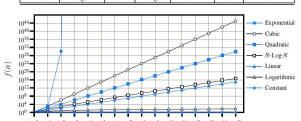
- Seven functions that often appear in algorithm analysis:
  - Constant ≈ 1
  - Logarithmic ≈ $\log n$
  - Linear ≈ n
  - N-Log-N ≈ $n \log n$
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
  - Exponential ≈ 2^n
- In a log-log chart, the slope of the line corresponds to the growth rate



10

Comparing Growth Rates

constant	logarithm	linear	n-log-n	quadratic	cubic	exponential
1	$\log n$	n	$n \log n$	n^2	n^3	a^n



12

4 5 6

Pseudocode

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Algorithm 1 Inter Communication Algorithm

```
1: procedure DEC-MDP(S, A, P, R, O, Ω)
2:     A ← A_1 × A_2
3:     s_1, s_2 ← S
4:     a_1, a_2 ← A
5:     R(s_1, a_1) ← 0, i ← 0, j ← 0
6:     repeat
7:         i ← i + 1, j ← j + 1
8:         for o_1, o_2 do
9:             Determine scenario ∈ [1, 4]
10:            p_1, p_2 ← P(s' | s_1, a_1)
11:            s_1, s_2 ← s'
12:            max_{a_1,a_2} r_{s,2}(s_1, s_2, a_1, a_2)
13:            for s_1, s_2 do check
14:                if d(s_1, s_2) ≤ necessary threshold then
15:                    Update θ_i, θ_j using d(s_1, s_2)
16:                end if
17:            π[s_1, s_2] ← arg max_{a_1,a_2} r_{1,2}
18:        end for
19:    end for
20: until a_1 = a_2 or s_2 = s_2.
21:    return π, R(s_1, a_2)
22: end procedure
```

13

Pseudocode Examples

```
1: neutral.save ← ∅ //Begin Generation
2: covered ← ⋃_{P∈Y} statements visited by P(t)
3: repeat
4:     variant ← single_mutation(P, covered)
5:     if is_neutral(var, T) then
6:         neutral_save ← neutral_save ∪ {variant}
7:     x ← x − 1
8: until x ≤ 0
9: clusters ← ∅ //Begin Composition
10: y' ← y
11: while |clusters| < N do
12:     candidate ← choose_from(neutral_save, k)
13:     if is_neutral(candidate, T) then
14:         clusters ← clusters ∪ {candidate}
15:         y' ← y
16:     else
17:         y' ← y' − 1
18:         if y' ≤ 0 then
19:             k ← |k/2|
20:             if k ≤ 1 then
21:                 return clusters
22:     y' ← y
23: return clusters
```

1. initialize p_0 agents, each with energy $E = \frac{E}{2}$
2. *loop*
3. *foreach* alive agent a:
4. pick link from current document
5. fetch new document D
6. $E_a \leftarrow E_a - c(D) + e(D)$
7. Q-learn with reinforcement signal $e(D)$
8. *if* $(E_a \geq \theta)$
9. $a' \leftarrow$ mutate(recombine(clone(a)))
10. $E_a, E_{a'} \leftarrow E_a/2$
11. *else if* $(E_a \leq 0)$
12. die(a)
13. process optional relevance feedback from user

15

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

```
/** Returns the maximum value of a nonempty array of numbers. */
public static double arrayMax(double[] data) {
3. int n = data.length;
4. double currentMax = data[0]; // assume first entry is biggest (for now)
5. for (int j=1; j < n; j++) // consider all other entries
6. if (data[j] > currentMax) // if data[j] is biggest thus far...
7.       currentMax = data[j]; // record it as the current max

8. return currentMax;
}
```

Step 3: 2 ops, 4: 2 ops, 5: 2n ops + 1, 6: 2n ops, 7: 0 to n ops, 8: 1 op

17

Pseudocode Details

- Indentation replaces braces
- Method declaration
  - **Algorithm** *method* (*arg* [, *arg*…])
  - **Input** …
  - **Output** …
- Method call
  - *method* (*arg* [, *arg*…])
- Return value
  - **return** *expression*
- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …

Type of operation	Symbol	Example
Assignment	← or :=	$c \leftarrow 2\pi r$, $c := 2\pi r$
Comparison	$=, \neq, <, >, \leq, \geq$	
Arithmetic	$+, -, \times, /,$ mod	
Floor/ceiling	$\lfloor \rfloor, \lceil \rceil$	$a \leftarrow \lfloor b \rfloor + \lceil c \rceil$
Logical	**and**, **or**	
Sums, products	$\sum \prod$	$h \leftarrow \sum_{i=0}^{n} 1/a$

14

Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the RAM model

- Examples:
  - Performing an arithmetic operation
  - Following an object reference
  - Assigning a value to a variable
  - Accessing a single element of an array by index
  - Calling a method
  - Returning from a method
  - Comparing two numbers

16

Estimating Running Time

- Algorithm arrayMax executes $5n + 6$ primitive operations in the worst case, $4n + 6$ in the best case. Define:
  - a = Time taken by the fastest primitive operation
  - b = Time taken by the slowest primitive operation
- Let $T(n)$ be worst-case time of arrayMax. Then
  $a(4n + 6) \leq T(n) \leq b(5n + 6)$
- Hence, the running time $T(n)$ is bounded by two linear functions

18

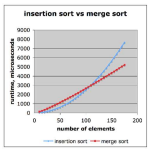
7

Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects $T(n)$ by a constant factor, but
  - Does not alter the growth rate of $T(n)$
- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm arrayMax

19

Comparison of Two Algorithms



insertion sort is
$n^2 / 4$
merge sort is
2 n lg n
sort a million items?
insertion sort takes
roughly 70 hours
while
merge sort takes
roughly 40 seconds

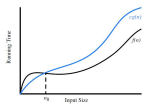
This is a slow machine, but if 100 x as fast then it' s 40 minutes versus less than 0.5 seconds

21

8

Big-Oh Notation

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that
  $f(n) \leq cg(n)$ for $n \geq n_0$
- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$



Illustrating the "big-Oh" notation. The function *f (n) is O(g(n)), since f (n) ≤ c ·g(n) when n ≥ n0.*

23

9

Why Growth Rate Matters

if runtime is…	time for n + 1	time for 2 n	time for 4 n
c lg n	c lg (n + 1)	c (lg n) + 1)	c((lg n) + 2)
c n	c (n + 1)	2c n	4c n
c n lg n	~ c n lg n + c n	2c n lg n + 2cn	4c n lg n + 4cn
c n^2	~ c n^2 + 2c n	4c n^2	16c n^2
c n^3	~ c n^3 + 3c n^2	8c n^3	64c n^3
c 2^n	c 2^{n+1}	c 2^{2n}	c 2^{4n}

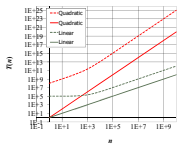
runtime quadruples when problem size doubles

20

10

Constant Factors

- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- Examples
  - $10^2 n + 10^5$ is a linear function
  - $10^5 n^2 + 10^8 n$ is a quadratic function

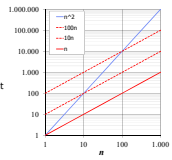


22

11

Big-Oh Example

- Example: the function n^2 is not $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$
  - The above inequality cannot be satisfied since c must be a constant



24

12

More Big-Oh Examples

- 7n - 2
  7n-2 is O(n)
  need c > 0 and $n_0 \geq 1$ such that 7 n - 2 ≤ c n for n ≥ n_0
  this is true for c = 7 and n_0 = 1
- 3 n^3 + 20 n^2 + 5
  3 n^3 + 20 n^2 + 5 is O(n³)
  need c > 0 and $n_0 \geq 1$ such that 3 n^3 + 20 n^2 + 5 ≤ c n^3 for n ≥ n_0
  this is true for c = 4 and n_0 = 21
- 3 log n + 5
  3 log n + 5 is O(log n)
  need c > 0 and $n_0 \geq 1$ such that 3 log n + 5 ≤ c log n for n ≥ n_0
  this is true for c = 8 and n_0 = 2

25

Big-Oh Rules

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

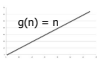
27

Ex:

- What is the complexity/growth rate of the following java function?

```
public static void printAll(double[ ] x) {
    int n = x.length;
    for (int j=0; j < n; j++) {
        System.out.print(x[j]);
    }
}
```

$O(n)$



29

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

26

13

Ex:

- What is the complexity/growth rate of the following java function?

```
public static void printAll(double[ ] x) {
    int n = x.length;
    for (int j=0; j < n; j++) {
        System.out.print(x[j]);
    }
}
```

$O(n)$

28

14

Relatives of Big-Oh

Big-Omega Ω

- f(n) is Ω(g(n)) if there is a constant c > 0 and an integer constant $n_0 \geq 1$ such that f(n) ≥ c g(n) for n ≥ n_0

Big-Theta Θ

- f(n) is Θ(g(n)) if there are constants c' > 0 and c'' > 0 and an integer constant $n_0 \geq 1$ such that c'g(n) ≤ f(n) ≤ c''g(n) for n ≥ n_0

30

15

Intuition for Asymptotic Notation

big-Oh
- f(n) is O(g(n)) if f(n) is asymptotically less than or equal to g(n)

big-Omega
- f(n) is Ω(g(n)) if f(n) is asymptotically greater than or equal to g(n)

big-Theta
- f(n) is Θ(g(n)) if f(n) is asymptotically equal to g(n)

31

Example Uses of the Relatives of Big-Oh

- $5n^2$ is $\Omega(n^2)$
  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$
  let $c = 5$ and $n_0 = 1$
- $5n^2$ is $\Omega(n)$
  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$
  let $c = 1$ and $n_0 = 1$
- $5n^2$ is $\Theta(n^2)$
  $f(n)$ is $\Theta(g(n))$ if it is $\Omega(n^2)$ and $O(n^2)$. We have already seen the former, for the latter recall that $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq c \cdot g(n)$ for $n \geq n_0$
  Let $c = 5$ and $n_0 = 1$

32

Asymptotic Algorithm Analysis

- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We say that algorithm arrayMax "runs in $O(n)$ time"
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

33

Computing Prefix Averages

- We further illustrate asymptotic analysis with two algorithms for prefix averages
- The i-th prefix average of an array X is average of the first $(i + 1)$ elements of X:

$$A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$$

- Computing the array A of prefix averages of another array X has applications to financial analysis

34

Prefix Averages (Quadratic)

The following algorithm computes prefix averages in quadratic time by applying the definition

```
/** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */
public static double[ ] prefixAverage1(double[ ] x) {
    int n = x.length;
    double[ ] a = new double[n]; // filled with zeros by default
    for (int j=0; j < n; j++) {
        double total = 0; // begin computing x[0] + ... + x[j]
        for (int i=0; i <= j; i++)
            total += x[i];
        a[j] = total / (j+1); // record the average
    }
    return a;
}
```

35

Arithmetic Progression

- The running time of prefixAverage1 is $O(1 + 2 + \ldots + n)$
- The sum of the first n integers is $n(n + 1) / 2$
  - There is a simple visual proof of this fact
- Thus, algorithm prefixAverage1 runs in $O(n^2)$ time

36

16

17

18

Prefix Averages 2 (Linear)

The following algorithm uses a running summation to improve the efficiency

```
/** Returns an array a such that, for all j, a[j] equals the average of
x[0], ..., x[j]. */
public static double[ ] prefixAverage2(double[] x) {
int n = x.length;
double[ ] a = new double[n]; // filled with zeros by default
double total = 0; // compute prefix sum as x[0] + x[1] + ...
for (int j=0; j < n; j++) {
    total += x[j]; // update prefix sum to include x[j]
    a[j] = total / (j+1); // compute average based on current sum
}
return a;
}
```

Algorithm prefixAverage2 runs in $O(n)$ time!

37

Justification Techniques (By Example)

- Some claims are of the **generic form**, "There is an element x *in a set S that has* property P." *To justify such a claim, we only need to produce a particular x in S* that has property P. *Likewise, some hard-to-believe claims are of the generic form,* "Every element x *in a set S has property P." To justify that such a claim is false, we* only need to produce a particular x ***from S that does not have property P***. *Such an* instance is called a ***counterexample***.
- Example: Professor Among us claims that every number of the form $2^i - 1$ is a prime, when i *is an integer greater than 1. Professor Amongus is wrong.*
- Justification: To prove Professor Amongus is wrong, we find a counterexample. Fortunately, we need not look too far, for $2^4 - 1 = 15 = 3 \cdot 5$.

39

Contradiction

- **Justification by contradiction technique**, we establish that a statement q *is true by first supposing that q is false and then showing that this assumption leads to a contradiction (such as $2 \neq 2$ or $1 > 3$).* By reaching such a contradiction, we show that no consistent situation exists with q *being false, so q must be true. Of course, in order to reach this* conclusion, we must be sure our situation is consistent before we assume q *is false.*
- **Example:** Let a *and b be integers. If ab is odd, then a is odd and b is odd.*
- **Justification:** Let ab *be odd.We wish to show that a is odd and b is odd. So,* with the hope of leading to a contradiction, let us assume the opposite, namely, suppose a *is even or b is even. In fact, without loss of generality, we can assume* that a *is even (since the case for b is symmetric). Then $a = 2j$ for some integer j. Hence, $ab = (2j)b = 2(jb)$, that is, ab is even. But this is a contradiction:* ***ab cannot simultaneously be odd and even***. Therefore, *a is odd and b is odd.*

41

Math you need to Review

- Summations
- Powers
- Logarithms
- Proof techniques
- Basic probability

- Properties of powers:
  $a^{(b+c)} = a^b a^c$
  $a^{bc} = (a^b)^c$
  $a^b / a^c = a^{(b-c)}$
  $b = a^{\log_a b}$
  $b^c = a^{c * \log_a b}$
- Properties of logarithms:
  $\log_b(xy) = \log_b x + \log_b y$
  $\log_b (x/y) = \log_b x - \log_b y$
  $\log_b xa = a\log_b x$
  $\log_a a = \log_a a / \log_a b$

38

19

20

21

The "Contra" Attack

- Another set of justification techniques involves the use of the negative. The two primary such methods are the use of the ***contrapositive and the contradiction.*** *To* justify the statement "if p *is true, then q is true," we establish that "if q is not true,* then p *is not true" instead. Logically, these two statements are the same, but the* latter, which is called the ***contrapositive of the first, may be easier to think about.***
- Example 4.18: Let *a and b be integers. If ab is even, then a is even or b is even.*
- Justification: To justify this claim, consider the contrapositive, "If a *is odd and b is odd, then ab is odd." So, suppose $a = 2j+1$ and $b = 2k+1$, for some integers j and k. Then $ab = 4jk+2j+2k+1 = 2(2\,jk+\,j+k)+1$; hence, ab is odd.*

40

Induction and Loop Invariants

- Most of the claims we make about a running time or a space bound involve an integer parameter n *(usually denoting an intuitive notion of the "size" of the problem).* Moreover, most of these claims are equivalent to saying some statement $q(n)$ *is true* "for all $n \geq 1$." *Since this is making a claim about an infinite set of numbers, we* cannot justify this exhaustively in a direct fashion.

42

Induction

- We can often justify claims such as those above as true, however, by using the technique of ***induction. This technique amounts to showing that, for any particular $n \geq 1$***, *there is a finite sequence of implications that starts with something known* to be true and ultimately leads to showing that $q(n)$ *is true. Specifically, we begin a* justification by induction by showing that $q(n)$ *is true for $n = 1$ (and possibly some* other values $n = 2, 3, \ldots, k$, *for some constant k). Then we justify that the inductive* "step" is true for $n > k$, *namely, we show "if $q(j)$ is true for all $j < n$, then $q(n)$ is* true." The combination of these two pieces completes the justification by induction.

43

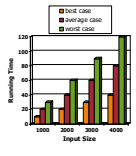
Loop Invariants

- The final justification technique we discuss in this section is the ***loop invariant.*** *To* prove some statement L about a loop is correct, define L in terms of a series of smaller statements $L_0, L_1, \ldots, L_k$, *where:*
- 1. The ***initial claim, L_0, is true before the loop begins.***
- 2. If L_{j-1} *is true before iteration j, then L_j will be true after iteration j.*
- 3. The final statement, L_k, *implies the desired statement L to be true.*
- Let us give a simple example of using a loop-invariant argument to justify the correctness of an algorithm. In particular, we use a loop invariant to justify that the method arrayFind (see Code Fragment 4.11) finds the smallest index at which element val occurs in array A.

45

Running Time

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



47

Induction

- Proposition 4.20: Consider the Fibonacci function $F(n)$, *which is defined such that $F(1) = 1$, $F(2) = 2$, and $F(n) = F(n-2)+F(n-1)$ for $n > 2$. (See Section 2.2.3.)* We claim that $F(n) < 2^n$.
- Justification: We will show our claim is correct by induction.
- ***Base cases: ($n \leq 2$). $F(1) = 1 < 2 = 2^1$ and $F(2) = 2 < 4 = 2^2$.***
- ***Induction step: ($n > 2$). Suppose our claim is true for all $j<n$. Since both $n-2$*** and *$n-1$ are less than n, we can apply the inductive assumption (sometimes called* the "inductive hypothesis") to imply that
- $F(n) = F(n-2)+F(n-1) < 2^{n-2}+2^{n-1}$.
- Since
- $2^{n-2}+2^{n-1} < 2^{n-1}+2^{n-1} = 2 \cdot 2^{n-1} = 2^n$,
- we have that $F(n) < 2^n$, *thus showing the inductive hypothesis for n.*

44

Loop Invariants

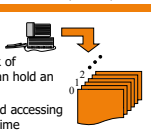
```
1 /** Returns index j such that data[j] == val, or −1 if no such element. */
2 public static int arrayFind(int[ ] data, int val) {
3     int n = data.length;
4     int j = 0;
5     while (j< n) { // val is not equal to any of the first j elements of data
6         if (data[j] == val)
7             return j; // a match was found at index j
8         j++; // continue to next index
9         // val is not equal to any of the first j elements of data
10    }
11    return −1; // if we reach this, no match found
12 }
```

Code Fragment 4.11: Algorithm arrayFind for finding the first index at which a given element occurs in an array. To show that arrayFind is correct, we inductively define a series of statements, L_j, *that lead to the correctness of our algorithm.*
Specifically, we claim the following is true at the beginning of iteration *j of the* ***while loop:***
L_j: *val is not equal to any of the first j elements of data.* This claim is true at the beginning of the first iteration of the loop, because j is 0 and there are no elements among the first 0 in data (this kind of a trivially true claim is said to hold ***vacuously). In iteration j, we compare element val to element data[j]:*** *if these two elements are equivalent, we return the index j, which is clearly* correct since no earlier elements equal *val. If the two elements val and data[j] are* not equal, then we have found one more element not equal to *val and we increment* the index *j. Thus, the claim L_j will be true for this new value of j; hence, it is* true at the beginning of the next iteration. If the while loop terminates without ever returning an index in data, then we have $j = n$. *That is, L_n is true—there are no* elements of data equal to *val. Therefore, the algorithm correctly returns −1* to indicate that *val is not in data.*

46

The Random Access Machine (RAM) Model

A RAM consists of
- A CPU
- An potentially unbounded bank of memory cells, each of which can hold an arbitrary number or character
- Memory cells are numbered and accessing any cell in memory takes unit time



48

22

23

24

RECURSION

The Recursion Pattern

- Recursion: is a technique by which a method makes one or more calls to itself during execution
- Classic example – the factorial function:

  n! = 1· 2· 3· ⋯ · (n-1)· n
- Recursive definition:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & \text{else} \end{cases}$$

Recursion Factorial - Java

- As a Java method:

```
public static int factorial(int n) throws IllegalArgumentException {
    if (n < 0) // argument must be nonnegative.
        throw new IllegalArgumentException( );

    if (n == 0)
        return 1; // base case

    return n * factorial(n-1); // recursive case
}
```


Visualizing Recursion

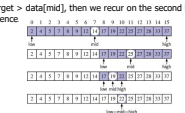
- Recursion trace
  - A box for each recursive call
  - An arrow from each caller to callee
  - An arrow from each callee to caller showing return value
- Example

Binary Search

- Binary search is an efficient algorithm for finding an item from a **sorted list of items**. It works by repeatedly dividing in half the portion of the list that could contain the item, until the list is narrowed down the possible locations to just one.

Visualizing Binary Search

- Middle is identified as mid = ⌊(low+high)/2⌋
- There are three cases:
  - If the target equals data[mid], then we have found the target.
  - If target < data[mid], then we recur on the first half of the sequence.
  - If target > data[mid], then we recur on the second half of the sequence



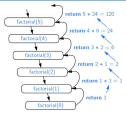
Binary Search

Search for an integer in an ordered list

```
/**
 * Returns true if the target value is found in the indicated portion of the
data array.
 * This search only considers the array portion from data[low] to data[high]
inclusive.
 */
public static boolean binarySearch(int[ ] data, int target, int low, int high) {
    if (low > high)
        return false; // interval empty; no match
    else {
        int mid = (low + high) / 2;
        if (target == data[mid])
            return true; // found a match
        else if (target < data[mid])
            // recur left of the middle
            return binarySearch(data, target, low, mid - 1);
        else
            // recur right of the middle
            return binarySearch(data, target, mid + 1, high);
    }
}
```


Analysis of Computing Factorials

- A sample recursion trace for our factorial method was given on the right.
- To compute factorial(n), we see that there are a total of n+1 activations, as the parameter decreases from n in the first call, to n−1 in the second call, and so on, until reaching the base case with parameter 0.
- Each individual activation of factorial executes a constant number of operations.
- Therefore, we conclude that the overall number of operations for computing factorial(n) is O(n), as there are n+1 activations, each of which accounts for O(1) operations.



Further Examples of Recursion

- If a recursive call starts at most one other, we call this a ***linear recursion.***
- If a recursive call may start two others, we call this a ***binary recursion.***
- If a recursive call may start three or more others, this is ***multiple recursion.***

Analyzing Recursive Algorithms

- Mathematical techniques for analyzing the efficiency of an algorithm, based upon an estimate of the number of primitive operations that are executed by the algorithm.

Analyzing Binary Search

- The remaining portion of the list is of size high – low + 1
- After one comparison, this becomes one of the following:

$$(\text{mid}-1)-\text{low}+1 = \left\lfloor \frac{\text{low}+\text{high}}{2} \right\rfloor - \text{low} \le \frac{\text{high}-\text{low}+1}{2}$$

$$\text{high}-(\text{mid}+1)+1 = \text{high} - \left\lfloor \frac{\text{low}+\text{high}}{2} \right\rfloor \le \frac{\text{high}-\text{low}+1}{2}$$

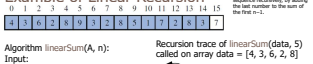
- Thus, each recursive call divides the search region in half; hence, there can be at most log n levels so runs in O(log n) time

Linear Recursion

- Test for base cases
  - Begin by testing for a set of base cases (there should be at least one).
  - Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.
- Recur once
  - Perform **a single recursive call**
  - This step may have a test that decides which of several possible recursive calls to make, but it should ultimately make just one of these calls
  - Define each possible recursive call so that it makes progress towards a base case.

Example of Linear Recursion

Computing the sum of a sequence recursively, by adding the last number to the sum of the first n−1.



Algorithm linearSum(A, n):
Input:
Array, A, of integers
Integer n such that 0 ≤ n ≤ |A|
Output:
Sum of the first n integers in A

```
if n = 0 then
  return 0
else
  return
linearSum(A, n - 1) + A[n - 1]
```

Recursion trace of linearSum(data, 5) called on array data = [4, 3, 6, 2, 8]

Designing Recursive Algorithm

- Base case(s)
  - Values of the input variables for which we perform no recursive calls are called base cases (there should be at least one base case).
  - Every possible chain of recursive calls must eventually reach a base case.
- Recursive calls
  - Calls to the current method.
  - Each recursive call should be defined so that it makes progress towards a base case.

Recursive Computing Powers

- The power function, p(x,n)=x^n, can be defined recursively:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x,n-1) & \text{else} \end{cases}$$

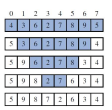
- This leads to an power function that runs in **O(n)** time (for we make n recursive calls)
- We can do better than this, however

```
/** Computes the value of x raised to the nth power, for
nonnegative integer n. */
public static double power(double x, int n) {
    if (n == 0)
        return 1;
    else
        return x * power(x, n-1);
}
```


Reversing an Array

Algorithm reverseArray(A, i, j):
Input: An array A and nonnegative integer indices i and j
Output: The reversal of the elements in A starting at index i and ending at

```
if i < j then
        Swap A[i] and A[ j]
        reverseArray(A, i + 1, j – 1)
return
```



A trace of the recursion for reversing a sequence. The highlighted portion has yet to be reversed.

Defining Arguments for Recursion

- In creating recursive methods, it is important to define the methods in ways that facilitate recursion.
- This sometimes requires we define additional parameters that are passed to the method.
- For example, we defined the array reversal method as reverseArray(A, i, j), not reverseArray(A)

```
/** Reverses the contents of subarray data[low] through data[high]
inclusive. */
public static void reverseArray(int[ ] data, int low, int high) {
    if (low < high) { // if at least two elements in subarray
        int temp = data[low]; // swap data[low] and data[high]
        data[low] = data[high];
        data[high] = temp;
        reverseArray(data, low + 1, high - 1); // recur on the rest
    }
}
```


Recursive Squaring

- We can derive a more efficient linearly recursive algorithm by using repeated squaring:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x,(n-1)/2)^2 & \text{if } n > 0 \text{ is odd} \\ p(x,n/2)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

- For example,

$2^4 = 2^{(4/2)2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16$

$2^5 = 2^{1+(4/2)2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32$

$2^6 = 2^{(6/2)2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64$

$2^7 = 2^{1+(6/2)2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128$

Recursive Squaring Method

Algorithm Power(x, n):
Input: A number x and integer n = 0
Output: The value x^n
if n = 0 **then**
return 1
if n is odd **then**
y = Power(x, (n - 1)/ 2)
return x · y ·y
else
y = Power(x, n/ 2)
return y · y

19

Analysis

Algorithm Power(x, n):
Input: A number x and integer n = 0
Output: The value x^n
if n = 0 **then**
return 1
if n is odd **then**
y = Power(x, (n - 1)/ 2)
return x · y · y
else
y = Power(x, n/ 2)
return y · y

Each time we make a recursive call we halve the value of n; hence, we make log n recursive calls. That is, this method runs in O(log n) time.

It is important that we use a variable twice here rather than calling the method twice.

20

Eliminating Recursion

- The main benefit of a recursive approach to algorithm design is that it allows us to succinctly take advantage of a repetitive structure present in many problems.
- By making our algorithm description exploit the repetitive structure in a recursive way, we can often avoid complex case analyses and nested loops. This approach can lead to more readable algorithm descriptions, while still being quite efficient.
- In general, we can use the stack data structure to convert a recursive algorithm into a non-recursive algorithm.

21

Eliminating Tail Recursion

- Tail recursion occurs when a linearly recursive method makes its recursive call as its last step.
- The array reversal method is an example.
- Such methods can be easily converted to non-recursive methods (which saves on some resources).
- Example:

Algorithm IterativeReverseArray(A, i, j):
Input: An array A and nonnegative integer indices i and j
Output: The reversal of the elements in A starting at index i and ending at j
while i < j **do**
Swap A[i] and A[j]
i = i + 1
j = j - 1
return

22

Ex: A Nonrecursive Implementation of Binary Search

```
/** Returns true if the target value is found in the data array. */
public static boolean binarySearchIterative(int[ ] data, int target) {
    int low = 0;
    int high = data.length - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (target == data[mid]) // found a match
            return true;
        else if (target < data[mid])
            high = mid - 1; // only consider values left of mid
        else
            low = mid + 1; // only consider values right of mid
    }
    return false; // loop ended without success
}
```

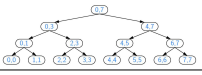
23

Binary Recursive Method

- Binary recursion occurs whenever there are **two** recursive calls for each non-base case
- Problem: add all the numbers in an integer array A:

```
public static int binarySum(int[ ] data, int low, int high) {
if (low > high) // zero elements in subarray
return 0;
else if (low == high) // one element in subarray
return data[low];
else {
int mid = (low + high) / 2;
return binarySum(data, low, mid) + binarySum(data, mid+1, high);
}
}
```

Example trace:



24

Multiple Recursion

- Motivating example:
  - summation puzzles
    - *pot* + *pan* = *bib*
- Multiple recursion:
  - makes potentially many recursive calls
  - not just one or two

25

Algorithm for Multiple Recursion

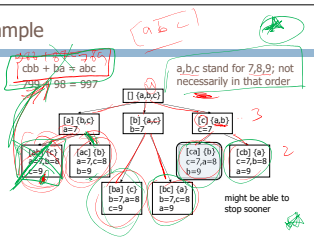
Algorithm PuzzleSolve(k,S,U):
Input: Integer k for the length of sequence, sequence S, and set U (universe of elements to test)
Output: Enumeration of all k-length extensions to S using elements in U without repetitions
for all e in U **do**
Remove e from U {e is now being used}
Add e to the end of S
if k = 1 **then**
Test whether S is a configuration that solves the puzzle
if S solves the puzzle **then**
return "Solution found: " S
else
PuzzleSolve(k - 1, S, U)
Add e back to U {e is now unused}
Remove e from the end of S

26

Example

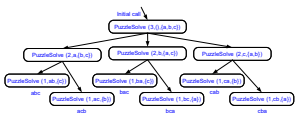
cbb + ba = abc
+ 7aa + 98 = 997

a,b,c stand for 7,8,9; not necessarily in that order



27

Visualizing PuzzleSolve



28

Exercise

- Describe (Pseudocode) a way to use recursion to compute the sum of all the elements in an n×n (two-dimensional) array of integers. What is your running time and space usage?

29

Additional Example

30

An Inefficient Recursion for Computing Fibonacci Numbers

- Fibonacci numbers are defined recursively:

$F_0 = 0$
$F_1 = 1$
$F_i = F_{i-1} + F_{i-2}$ for $i > 1$.

- Recursive algorithm (first attempt):

Algorithm BinaryFib(k):
Input: Nonnegative integer k
Output: The kth Fibonacci number F_k
if $k = 1$ **then**
return k
else
return BinaryFib($k - 1$) + BinaryFib($k - 2$)

31

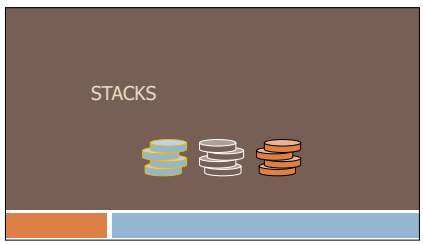
A Better Fibonacci Algorithm

- Use linear recursion instead

Algorithm LinearFibonacci(k):
Input: A nonnegative integer k
Output: Pair of Fibonacci numbers (F_k, F_{k-1})
if k = 1 **then**
return (k, 0)
else
(i, j) = LinearFibonacci(k – 1)
return (i +j, i)

- LinearFibonacci makes k–1 recursive calls

33



1

Analysis

- Let n_k be the number of recursive calls by BinaryFib(k)
  - $n_0 = 1$
  - $n_1 = 1$
  - $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$
  - $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$
  - $n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$
  - $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$
  - $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$
  - $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$
  - $n_8 = n_7 + n_6 + 1 = 41 + 25 + 1 = 67$.
- Note that n_k at least doubles every other time
- That is, $n_k > 2^{k/2}$. It is exponential!

32

Exercise

- Write a program for solving summation puzzles by enumerating and testing all possible configurations. Using your program, solve the **three** different puzzles given
  - *pot* + *pan* = *bib*
  - *dog* + *cat* = *pig*
  - *boy* + *girl* = *baby*

where each char is a digit.

34

Stack

- A stack is a collection of objects that are inserted and removed according **to the last-in, first-out (LIFO)** principle.

2

Abstract Data Types (ADTs)

- An abstract data type (ADT) is an abstraction of a data structure
- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations
- Example: ADT modeling a simple stock trading system
  - The data stored are buy/sell orders
  - The operations supported are
    - order buy(stock, shares, price)
    - order sell(stock, shares, price)
    - void cancel(order)
  - Error conditions:
    - Buy/sell a nonexistent stock
    - Cancel a nonexistent order

Stack Interface in Java

- Java interface corresponding to the Stack ADT
- Assumes null is returned from top() and pop() when stack is empty
- Different from the built-in Java class java.util.Stack

```
public interface Stack<E> {
  int size();
  boolean isEmpty();
  E top();
  void push(E element);
  E pop();
}
```


Exceptions vs. Returning Null

- Attempting the execution of an operation of an ADT may sometimes cause an error condition
- Java supports a general abstraction for errors, called exception
- An exception is said to be "thrown" by an operation that cannot be properly executed
- In our Stack ADT, we do not use exceptions
- Instead, we allow operations pop and top to be performed even if the stack is empty
- For an empty stack, pop and top simply return null

The Stack ADT

- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the last-in first-out scheme
- Main stack operations:
  - push(object): inserts an element
  - object pop(): removes and returns the last inserted element
- Auxiliary stack operations:
  - object top(): returns the last inserted element without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored


```
/**
 * A collection of objects that are inserted and removed according to the last-in
 * first-out principle. Although similar in purpose, this interface differs from
 * java.util.Stack.
 *
 * @author Michael T. Goodrich
 * @author Roberto Tamassia
 * @author Michael H. Goldwasser
 */
public interface Stack<E> {
  /**
   * Returns the number of elements in the stack.
   * @return number of elements in the stack
   */
  int size();

  /**
   * Tests whether the stack is empty.
   * @return true if the stack is empty, false otherwise
   */
  boolean isEmpty();

  /**
   * Inserts an element at the top of the stack.
   * @param e  the element to be inserted
   */
  void push(E e);

  /**
   * Returns, but does not remove, the element at the top of the stack.
   * @return top element in the stack (or null if empty)
   */
  E top();

  /**
   * Removes and returns the top element from the stack.
   * @return element removed (or null if empty)
   */
  E pop();
}
```


A Simple Array-Based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

Algorithm $size()$
  return $t + 1$

Algorithm $pop()$
  if $isEmpty()$ **then**
    return null
  else
    $t \leftarrow t - 1$
    return $S[t + 1]$

Representing a stack with an array; the top element is in cell data[t]

Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then throw a FullStackException
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT

Algorithm $push(o)$
  if $t = S.length - 1$ **then**
    throw *IllegalStateException*
  else
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$

Performance and Limitations

- Performance
  - Let n be the number of elements in the stack
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$
- Limitations
  - The maximum size of the stack must be defined during initialization and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception

Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
top	$O(1)$
push	$O(1)$
pop	$O(1)$

Example Use in Java

```
public class Tester {
  // ... other methods
  public intReverse(Integer a[]) {
    Stack<Integer> s;
    s = new ArrayStack<Integer>();
    ... (code to reverse array a) ...
  }
```

```
public floatReverse(Float f[]) {
  Stack<Float> s;
  s = new ArrayStack<Float>();
  ... (code to reverse array f) ...
}
```


Java implementation based on this strategy is given in Code Fragment 6.2 (with Javadoc comments omitted due to space considerations).

```
1  public class ArrayStack<E> implements Stack<E> {
2    public static final int CAPACITY=1000;  // default array capacity
3    private E[ ] data;                      // generic array used for storage
4    private int t = -1;                     // index of the top element in stack
5    public ArrayStack( ) { this(CAPACITY); } // constructs stack with default capacity
6    public ArrayStack(int capacity) {       // constructs stack with given capacity
7      data = (E[ ]) new Object[capacity];   // safe cast; compiler may give warning
8    }
9    public int size( ) { return (t + 1); }
10   public boolean isEmpty( ) { return (t == -1); }
11   public void push(E e) throws IllegalStateException {
12     if (size( ) == data.length) throw new IllegalStateException("Stack is full");
13     data[++t] = e;                        // increment t before storing new item
14   }
15   public E top( ) {
16     if (isEmpty( )) return null;
17     return data[t];
18   }
19   public E pop( ) {
20     if (isEmpty( )) return null;
21     E answer = data[t];
22     data[t] = null;                       // dereference to help garbage collection
23     t--;
24     return answer;
25   }
26 }
```

Code Fragment 6.2: Array-based implementation of the Stack interface.

Garbage Collection in Java

- The reason for returning the cell to a null reference in Code Fragment 6.2 is to assist Java's garbage collection mechanism, which searches memory for objects that are no longer actively referenced and reclaims their space for future use.

Sample Usage

Method	Return Value	Stack Contents
push(5)	–	(5)
push(3)	–	(5, 3)
size( )	2	(5, 3)
pop( )	3	(5)
isEmpty( )	false	(5)
pop( )	5	( )
isEmpty( )	true	( )
pop( )	null	( )
push(7)	–	(7)
push(9)	–	(7, 9)
top( )	9	(7, 9)
push(4)	–	(7, 9, 4)
size( )	3	(7, 9, 4)
pop( )	4	(7, 9)
push(6)	–	(7, 9, 6)
push(8)	–	(7, 9, 6, 8)
pop( )	8	(7, 9, 6)

```
Stack<Integer> S = new ArrayStack<>();   // contents: ()
S.push(5);                               // contents: (5)
S.push(3);                               // contents: (5, 3)
System.out.println(S.size());            // contents: (5, 3)   outputs 2
System.out.println(S.pop());             // contents: (5)      outputs 3
System.out.println(S.isEmpty());         // contents: (5)      outputs false
System.out.println(S.pop());             // contents: ()       outputs 5
System.out.println(S.isEmpty());         // contents: ()       outputs true
System.out.println(S.pop());             // contents: ()       outputs null
S.push(7);                               // contents: (7)
S.push(9);                               // contents: (7, 9)
System.out.println(S.top());             // contents: (7, 9)   outputs 9
S.push(4);                               // contents: (7, 9, 4)
System.out.println(S.size());            // contents: (7, 9, 4)   outputs 3
System.out.println(S.pop());             // contents: (7, 9)      outputs 4
S.push(6);                               // contents: (7, 9, 6)
S.push(8);                               // contents: (7, 9, 6, 8)
System.out.println(S.pop());             // contents: (7, 9, 6)   outputs 8
```


Applications of Stacks

- Direct applications
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Ex: Parentheses Matching

- Each "(", "{", or "[" must be paired with a matching ")", "}", or "]"
  - correct: ( )(( )){([( )])}
  - correct: (( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

Ex: HTML Tag Matching

- For fully-correct HTML, each <name> should pair with a matching </name>

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

The Little Boat

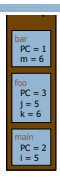
The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

Method Stack in the JVM

- The Java Virtual Machine (JVM) keeps track of the chain of active methods with a stack
- When a method is called, the JVM pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a method ends, its frame is popped from the stack and control is passed to the method on top of the stack
- Allows for recursion

```
main() {
  int i = 5;
  foo(i);
}

foo(int j) {
  int k;
  k = j+1;
  bar(k);
}

bar(int m) {
  …
}
```



Parenthesis Matching (Java)

```
public static boolean isMatched(String expression) {
  final String opening = "({[";  // opening delimiters
  final String closing = ")}]";  // respective closing delimiters
  Stack<Character> buffer = new LinkedStack<>( );
  for (char c : expression.toCharArray( )) {
    if (opening.indexOf(c) != -1)  // this is a left delimiter
      buffer.push(c);
    else if (closing.indexOf(c) != -1) {  // this is a right delimiter
      if (buffer.isEmpty( ))  // nothing to match with
        return false;
      if (closing.indexOf(c) != opening.indexOf(buffer.pop( )))
        return false;  // mismatched delimiter
    }
  }
  return buffer.isEmpty( );  // were all opening delimiters matched?
}
```


HTML Tag Matching (Java)

```
public static boolean isHTMLMatched(String html) {
  Stack<String> buffer = new LinkedStack<>( );
  int j = html.indexOf("<");  // find first '<' character (if any)
  while (j != -1) {
    int k = html.indexOf('>', j+1);  // find next '>' character
    if (k == -1)
      return false;  // invalid tag
    String tag = html.substring(j+1, k);  // strip away < >
    if (!tag.startsWith("/"))  // this is an opening tag
      buffer.push(tag);
    else {  // this is a closing tag
      if (buffer.isEmpty( ))
        return false;  // no tag to match
      if (!tag.substring(1).equals(buffer.pop( )))
        return false;  // mismatched tag
    }
    j = html.indexOf('<', k+1);  // find next '<' character (if any)
  }
  return buffer.isEmpty( );  // were all opening tags matched?
}
```

Ex: Evaluating Arithmetic Expressions

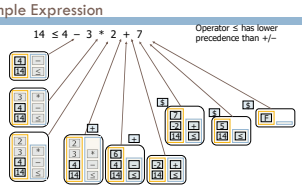
14 – 3 * 2 + 7 = (14 – (3 * 2)) + 7

Operator precedence
- * has precedence over +/–

Associativity
- operators of the same precedence group evaluated from left to right
- Example: (x – y) + z rather than x – (y + z)

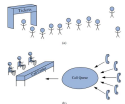
Idea: push each operator on the stack, but first pop and perform higher and *equal* precedence operations.

Algorithm on an Example Expression



The Queue

- The Queue stores arbitrary objects
- Insertions and deletions follow the first-in first-out scheme
- Insertions are at the rear of the queue and removals are at the front of the queue



Algorithm for Evaluating Expressions

Two stacks:
- opStk holds operators
- valStk holds values
- Use $ as special "end of input" token with lowest precedence

Algorithm doOp()
```
x ← valStk.pop();
y ← valStk.pop();
op ← opStk.pop();
valStk.push( y op x )
```

Algorithm repeatOps(refOp):
```
while ( valStk.size() > 1 ∧
        prec(refOp) ≤
        prec(opStk.top())
  doOp()
```

Algorithm EvalExp()
- Input: a stream of tokens representing an arithmetic expression (with numbers)
- Output: the value of the expression

```
while there's another token z
  if isNumber(z) then
    valStk.push(z)
  else
    repeatOps(z);
    opStk.push(z)
repeatOps($);
return valStk.top()
```


QUEUES

The Queue ADT

- Main queue operations:
  - enqueue(object): inserts an element at the end of the queue
  - object dequeue(): removes and returns the element at the front of the queue
- Auxiliary queue operations:
  - object first(): returns the element at the front without removing it
  - integer size(): returns the number of elements stored
  - boolean isEmpty(): indicates whether no elements are stored
- Boundary cases:
  - Attempting the execution of dequeue or first on an empty queue returns null

Java Interface for Queue

```
1  public interface Queue<E> {
2    /** Returns the number of elements in the queue. */
3    int size();
4    /** Tests whether the queue is empty. */
5    boolean isEmpty();
6    /** Inserts an element at the rear of the queue. */
7    void enqueue(E e);
8    /** Returns, but does not remove, the first element of the queue (null if empty). */
9    E first();
10   /** Removes and returns the first element of the queue (null if empty). */
11   E dequeue();
12 }
```

Code Fragment 6.9: A Queue interface defining the queue ADT, with a standard FIFO protocol for insertions and removals.

Applications of Queues

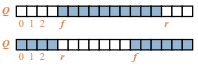
- Direct applications
  - Waiting lists, bureaucracy
  - Access to shared resources (e.g., printer)
  - Multiprogramming
- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures

Queue Operations

- We use the modulo operator (remainder of division)

Algorithm *size*()
  return sz

Algorithm *isEmpty*()
  return $(sz == 0)$

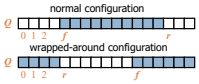


Example

Operation		*Output*	*Q*
enqueue(5)		–	(5)
enqueue(3)		–	(5, 3)
dequeue()		*5*	(3)
enqueue(7)		–	(3, 7)
dequeue()		*3*	(7)
first()		*7*	(7)
dequeue()		*7*	()
dequeue()		*null*	()
isEmpty()		*true*	()
enqueue(9)		–	(9)
enqueue(7)		–	(9, 7)
size()		*2*	(9, 7)
enqueue(3)		–	(9, 7, 3)
enqueue(5)		–	(9, 7, 3, 5)
dequeue()		*9*	(7, 3, 5)

Array-based Queue

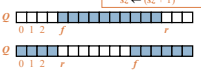
- Use an array of size N in a circular fashion
- Two variables keep track of the front and size
  - f index of the front element
  - sz number of stored elements
- When the queue has fewer than N elements, array location $r = (f + sz) \bmod N$ is the first empty slot past the rear of the queue



Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

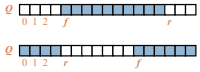
Algorithm *enqueue*(o)
  if $size() = N - 1$ **then**
    throw *IllegalStateException*
  else
    $r \leftarrow (f + sz) \bmod N$
    $Q[r] \leftarrow o$
    $sz \leftarrow (sz + 1)$



Queue Operations (cont.)

- Note that operation dequeue returns null if the queue is empty

Algorithm *dequeue*()
  if *isEmpty*() **then**
    return *null*
  else
    $o \leftarrow Q[f]$
    $f \leftarrow (f + 1) \bmod N$
    $sz \leftarrow (sz - 1)$
    return o



Array-based Implementation

```
1  /** Implementation of the queue ADT using a fixed-length array. */
2  public class ArrayQueue<E> implements Queue<E> {
3    // instance variables
4    private E[ ] data;                    // generic array used for storage
5    private int f = 0;                    // index of the front element
6    private int sz = 0;                   // current number of elements
7
8    // constructors
9    public ArrayQueue( ) {this(CAPACITY);} // constructs queue with default capacity
10   public ArrayQueue(int capacity) {     // constructs queue with given capacity
11     data = (E[ ]) new Object[capacity];  // safe cast; compiler may give warning
12   }
13
14   // methods
15   /** Returns the number of elements in the queue. */
16   public int size( ) { return sz; }
17
18   /** Tests whether the queue is empty. */
19   public boolean isEmpty( ) { return (sz == 0); }
20
```


Comparison to java.util.Queue

- Our Queue methods and corresponding methods of java.util.Queue:

Our Queue ADT	Interface java.util.Queue	
	throws exceptions	returns special value
enqueue(e)	add(e)	offer(e)
dequeue( )	remove( )	poll( )
first( )	element( )	peek( )
size( )	size( )	
isEmpty( )	isEmpty( )	

Queue Interface in Java

- Java interface corresponding to our Queue ADT
- Assumes that first() and dequeue() return null if queue is empty

```
public interface Queue<E> {
  int size();
  boolean isEmpty();
  E first();
  void enqueue(E e);
  E dequeue();
}
```


Array-based Implementation (2)

```
21   /** Inserts an element at the rear of the queue. */
22   public void enqueue(E e) throws IllegalStateException {
23     if (sz == data.length) throw new IllegalStateException("Queue is full");
24     int avail = (f + sz) % data.length;   // use modular arithmetic
25     data[avail] = e;
26     sz++;
27   }
28
29   /** Returns, but does not remove, the first element of the queue (null if empty). */
30   public E first( ) {
31     if (isEmpty( )) return null;
32     return data[f];
33   }
34
35   /** Removes and returns the first element of the queue (null if empty). */
36   public E dequeue( ) {
37     if (isEmpty( )) return null;
38     E answer = data[f];
39     data[f] = null;                       // dereference to help garbage collection
40     f = (f + 1) % data.length;
41     sz--;
42     return answer;
43   }
```


Analyzing the Efficiency of an Array-Based Queue

Method	Running Time
size	$O(1)$
isEmpty	$O(1)$
first	$O(1)$
enqueue	$O(1)$
dequeue	$O(1)$

Implementing a Queue with a Singly Linked List

- Singly linked list to implement the queue ADT while supporting worst-case O(1)-time for all operations

```
1  /** Realization of a FIFO queue as an adaptation of a SinglyLinkedList. */
2  public class LinkedQueue<E> implements Queue<E> {
3    private SinglyLinkedList<E> list = new SinglyLinkedList<>();  // an empty list
4    public LinkedQueue( ) { }                    // new queue relies on the initially empty list
5    public int size( ) { return list.size( ); }
6    public boolean isEmpty( ) { return list.isEmpty( ); }
7    public void enqueue(E element) { list.addLast(element); }
8    public E first( ) { return list.first( ); }
9    public E dequeue( ) { return list.removeFirst( ); }
10 }
```

Code Fragment 6.11: Implementation of a Queue using a SinglyLinkedList.

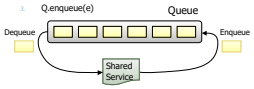
LISTS AND ITERATORS

The java.util.List ADT

- The java.util.List interface includes the following methods:
  - size( ): Returns the number of elements in the list.
  - isEmpty( ): Returns a boolean indicating whether the list is empty.
  - get(*i*): Returns the element of the list having index *i*; an error condition occurs if *i* is not in range $[0, \text{size}( ) - 1]$.
  - set(*i*, *e*): Replaces the element at index *i* with *e*, and returns the old element that was replaced; an error condition occurs if *i* is not in range $[0, \text{size}( ) - 1]$.
  - add(*i*, *e*): Inserts a new element *e* into the list so that it has index *i*, moving all subsequent elements one index later in the list; an error condition occurs if *i* is not in range $[0, \text{size}( )]$.
  - remove(*i*): Removes and returns the element at index *i*, moving all subsequent elements one index earlier in the list; an error condition occurs if *i* is not in range $[0, \text{size}( ) - 1]$.

Application: Round Robin Schedulers

- We can implement a round robin scheduler using a queue Q by repeatedly performing the following steps:
  1. e = Q.dequeue()
  2. Service element e
  3. Q.enqueue(e)



Lists

- List is a data structure which provides the facility to maintain the ordered collection.
- It contains index-based methods to insert, update, delete and search the elements.
- It can have the duplicate elements and null elements in the list.
- The List interface is found in the java.util package and inherits the Collection interface. It is a factory of ListIterator interface.
- Through the ListIterator, user can iterate the list in forward and backward directions.
- The implementation classes of List interface are **ArrayList, LinkedList, Stack** and Vector.
- The ArrayList and LinkedList are widely used in Java programming. The Vector class is deprecated since Java 5.

Example

- A sequence of List operations:

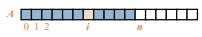
Method	Return Value	List Contents
add(0, A)	–	(A)
add(0, B)	–	(B, A)
get(1)	A	(B, A)
set(2, C)	"error"	(B, A)
add(2, C)	–	(B, A, C)
add(4, D)	"error"	(B, A, C)
remove(1)	A	(B, C)
add(1, D)	–	(B, D, C)
add(1, E)	–	(B, E, D, C)
get(4)	"error"	(B, E, D, C)
add(4, F)	–	(B, E, D, C, F)
set(2, G)	D	(B, E, G, C, F)
get(2)	G	(B, E, G, C, F)

List Interface

```
1  /** A simplified version of the java.util.List interface. */
2  public interface List<E> {
3    /** Returns the number of elements in this list. */
4    int size( );
5
6    /** Returns whether the list is empty. */
7    boolean isEmpty( );
8
9    /** Returns (but does not remove) the element at index i. */
10   E get(int i) throws IndexOutOfBoundsException;
11
12   /** Replaces the element at index i with e, and returns the replaced element. */
13   E set(int i, E e) throws IndexOutOfBoundsException;
14
15   /** Inserts element e to be at index i, shifting all subsequent elements later. */
16   void add(int i, E e) throws IndexOutOfBoundsException;
17
18   /** Removes/returns the element at index i, shifting subsequent elements earlier. */
19   E remove(int i) throws IndexOutOfBoundsException;
20 }
```

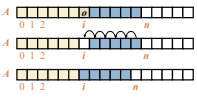

Array Lists

- An obvious choice for implementing the list ADT is to use an array, **A**, where **A[i]** stores (a reference to) the element with index **i**.
- With a representation based on an array **A**, the get(i) and set(i, e) methods are easy to implement by accessing **A[i]** (assuming i is a legitimate index).



Element Removal

- In an operation *remove*(i), we need to fill the hole left by the removed element by shifting backward the $n - i - 1$ elements $A[i+1], \ldots, A[n-1]$
- In the worst case ($i = 0$), this takes $O(n)$ time

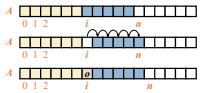


Java List Example

```
1.  import java.util.*;
2.  public class ListExample1{
3.  public static void main(String args[]){
4.      //Creating a List
5.      List<String> list=new ArrayList<String>();
6.      //Adding elements in the List
7.      list.add("Mango");
8.      list.add("Apple");
9.      list.add("Banana");
10.     list.add("Grapes");
11.     //Iterating the List element using for-each loop
12.     for(String fruit:list)
13.         System.out.println(fruit);
14.
15. }
16. }
```


Insertion

- In an operation *add*(**i**, **o**), we need to make room for the new element by shifting forward the $n - i$ elements $A[i], \ldots, A[n-1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



Performance

Method	Running Time
size( )	$O(1)$
isEmpty( )	$O(1)$
get(i)	$O(1)$
set(i, e)	$O(1)$
add(i, e)	$O(n)$
remove(i)	$O(n)$

- In an array-based implementation of a dynamic list:
  - The space used by the data structure is **O(*n*)**
  - Indexing the element at i takes **O(1)** time
  - ***add*** and ***remove*** run in **O(*n*)** time
- In an ***add*** operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one …

Java Implementation

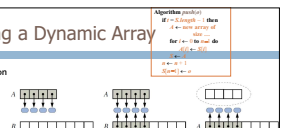
```
11   // public methods
12   /** Returns the number of elements in the array list. */
13   public int size( ) { return size; }
14   /** Returns whether the array list is empty. */
15   public boolean isEmpty( ) { return size == 0; }
16   /** Returns (but does not remove) the element at index i. */
17   public E get(int i) throws IndexOutOfBoundsException {
18     checkIndex(i, size);
19     return data[i];
20   }
21   /** Replaces the element at index i with e, and returns the replaced element. */
22   public E set(int i, E e) throws IndexOutOfBoundsException {
23     checkIndex(i, size);
24     E temp = data[i];
25     data[i] = e;
26     return temp;
27   }
```


Dynamic Arrays

- The ArrayList implementations so far as well as those for a stack, queue, and deque from previous chapters has a serious limitation; it requires that a fixed maximum capacity be declared, throwing an exception if attempting to add an element once full.
- This is a major weakness, and there is risk that either too large of an array will be requested, causing an inefficient waste of memory, or that too small of an array will be requested, causing a fatal error when exhausting that capacity.
- Java's ArrayList class provides a more robust abstraction, allowing a user to add elements to the list, with no apparent limit on the overall capacity.
- To provide this abstraction, Java relies on an algorithmic sleight of hand that is known as a dynamic array.
- In reality, elements of an ArrayList are stored in a traditional array, and the precise size of that traditional array must be internally declared in order for the system to properly allocate a consecutive piece of memory for its storage.

Implementing a Dynamic Array

- Let push(o) be the operation that adds element o at the end of the list
- When the array is full, we replace the array with a larger one
- How large should the new array be?
  - Incremental strategy: increase the size by a constant *c*
  - Doubling strategy: double the size




```
/** Resizes internal array to have given capacity >= size. */
protected void resize(int capacity) {
  E[ ] temp = (E[ ]) new Object[capacity];  // safe cast; compiler may give warning
  for (int k=0; k < size; k++)
    temp[k] = data[k];
  data = temp;                              // start using the new array
}
```

Code Fragment 7.4: An implementation of the ArrayList.resize method.

Java Implementation, 2

```
28   /** Inserts element e to be at index i, shifting all subsequent elements later. */
29   public void add(int i, E e) throws IndexOutOfBoundsException,
30                                      IllegalStateException {
31     checkIndex(i, size + 1);
32     if (size == data.length)                  // not enough capacity
33       throw new IllegalStateException("Array is full");
34     for (int k=size-1; k >= i; k--)           // start by shifting rightmost
35       data[k+1] = data[k];
36     data[i] = e;                              // ready to place the new element
37     size++;
38   }
39   /** Removes/returns the element at index i, shifting subsequent elements earlier. */
40   public E remove(int i) throws IndexOutOfBoundsException {
41     checkIndex(i, size);
42     E temp = data[i];
43     for (int k=i; k < size-1; k++)            // shift elements to fill hole
44       data[k] = data[k+1];
45     data[size-1] = null;                      // help garbage collection
46     size--;
47     return temp;
48   }
49   // utility method
50   /** Checks whether the given index is in the range [0, n-1]. */
51   protected void checkIndex(int i, int n) throws IndexOutOfBoundsException {
52     if (i < 0 || i >= n)
53       throw new IndexOutOfBoundsException("Illegal index: " + i);
54   }
55 }
```


Dynamic Arrays

- Because the system may allocate neighboring memory locations to store other data, the capacity of an array cannot be increased by expanding into subsequent cells. The first key to providing the semantics of an unbounded array is that an array list instance maintains an internal array that often has greater capacity than the current length of the list. For example, while a user may have created a list with five elements, the system may have reserved an underlying array capable of storing eight object references (rather than only five). This extra capacity makes it easy to add a new element to the end of the list by using the next available cell of the array.
- If a user continues to add elements to a list, all reserved capacity in the underlying array will eventually be exhausted. In that case, the class requests a new, larger array from the system, and copies all references from the smaller array into the beginning of the new array.

Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n push operations
- We assume that we start with an empty list represented by a growable array of size 1
- We call amortized time of a push operation the average time taken by a push operation over the series of operations, i.e., $T(n)/n$

Incremental Strategy Analysis

- Over n push operations, we replace the array $k = n/c$ times, where c is a constant
- The total time $T(n)$ of a series of n push operations is proportional to

$$n + c + 2c + 3c + 4c + \ldots + kc = n + c(1 + 2 + 3 + \ldots + k) = n + ck(k+1)/2$$

- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- Thus, the amortized time of a push operation is $O(n)$

Doubling Strategy Analysis

- We replace the array $k = \log_2 n$ times
- The total time $T(n)$ of a series of n push operations is proportional to

$$n + 1 + 2 + 4 + 8 + \ldots + 2^k = n + 2^{k+1} - 1 = 3n - 1$$

- $T(n)$ is $O(n)$
- The amortized time of a push operation is $O(1)$

Positional Lists

- To provide for a general abstraction of a sequence of elements with the ability to identify the location of an element, we define a **positional list** ADT.
- A position acts as a marker or token within the broader positional list.
- A position p is unaffected by changes elsewhere in a list; the only way in which a position becomes invalid is if an explicit command is issued to delete it.
- A position instance is a simple object, supporting only the following method:
  - p.getElement( ): Return the element stored at position p.

Positional List ADT

- Accessor methods:

first( ): Returns the position of the first element of L (or null if empty).
last( ): Returns the position of the last element of L (or null if empty).
before(p): Returns the position of L immediately before position p (or null if p is the first position).
after(p): Returns the position of L immediately after position p (or null if p is the last position).
isEmpty( ): Returns true if list L does not contain any elements.
size( ): Returns the number of elements in list L.

Positional List ADT, 2

- Update methods:

addFirst(e): Inserts a new element e at the front of the list, returning the position of the new element.
addLast(e): Inserts a new element e at the back of the list, returning the position of the new element.
addBefore(p, e): Inserts a new element e in the list, just before position p, returning the position of the new element.
addAfter(p, e): Inserts a new element e in the list, just after position p, returning the position of the new element.
set(p, e): Replaces the element at position p with element e, returning the element formerly at position p.
remove(p): Removes and returns the element at position p in the list, invalidating the position.

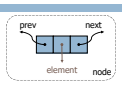
Example

- A sequence of Positional List operations:

Method	Return Value	List Contents
addLast(8)	p	(8_p)
first( )	p	(8_p)
addAfter(p, 5)	q	$(8_p, 5_q)$
before(q)	p	$(8_p, 5_q)$
addBefore(q, 3)	r	$(8_p, 3_r, 5_q)$
r.getElement( )	3	$(8_p, 3_r, 5_q)$
after(p)	r	$(8_p, 3_r, 5_q)$
before(p)	null	$(8_p, 3_r, 5_q)$
addFirst(9)	s	$(9_s, 8_p, 3_r, 5_q)$
remove(last( ))	5	$(9_s, 8_p, 3_r)$
set(p, 7)	8	$(9_s, 7_p, 3_r)$
remove(q)	"error"	$(9_s, 7_p, 3_r)$

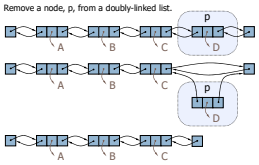
Positional List Implementation

- The most natural way to implement a positional list is with a doubly-linked list.



Deletion

- Remove a node, p, from a doubly-linked list.

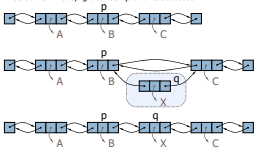


The Iterable Interface

- Java defines a parameterized interface, named Iterable, that includes the following single method:
  - iterator( ): Returns an iterator of the elements in the collection.
- An instance of a typical collection class in Java, such as an ArrayList, is iterable (but not itself an iterator); it produces an iterator for its collection as the return value of the iterator( ) method.
- Each call to iterator( ) returns a new iterator instance, thereby allowing multiple (even simultaneous) traversals of a collection.

Insertion

- Insert a new node, q, between p and its successor.



Iterators

- An iterator is a software design pattern that abstracts the process of scanning through a sequence of elements, one element at a time.

hasNext( ): Returns true if there is at least one additional element in the sequence, and false otherwise.
next( ): Returns the next element in the sequence.

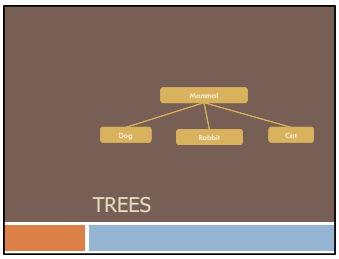
The for-each Loop

- Java's Iterable class also plays a fundamental role in support of the "for-each" loop syntax:

```
for (ElementType variable : collection) {
  loopBody                                  // may refer to "variable"
}
```

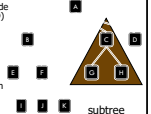
is equivalent to:

```
Iterator<ElementType> iter = collection.iterator();
while (iter.hasNext()) {
  ElementType variable = iter.next();
  loopBody                                  // may refer to "variable"
}
```

Tree Terminology

- **Root:** node without parent (A)
- **Internal node:** node with at least one child (A, B, C, F)
- **External node (a.k.a. leaf):** node without children (E, I, J, K, G, H, D)
- **Ancestors of a node:** parent, grandparent, grand-grandparent, etc.
- **Depth of a node:** number of ancestors
- **Height of a tree:** maximum depth of any node (3)
- **Descendant of a node:** child, grandchild, grand-grandchild, etc.
- Subtree: tree consisting of a node and its descendants

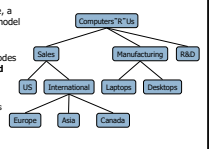


Tree ADT

- We use positions to abstract nodes
- Generic methods:
  - integer size()
  - boolean isEmpty()
  - Iterator iterator()
  - Iterable positions()
- Accessor methods:
  - position root()
  - position parent(p)
  - Iterable children(p)
  - Integer numChildren(p)
- Query methods:
  - boolean isInternal(p)
  - boolean isExternal(p)
  - boolean isRoot(p)
- Additional update methods may be defined by data structures implementing the Tree ADT

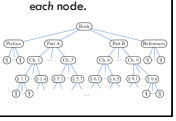
What is a Tree

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a **parent-child** relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments



Tree Terminology

- **Sibling:** Two nodes that are children of the same parent.
- ***Edge of tree T:*** *is a pair of nodes (u,v) such that u is the parent of v, or vice versa.*
- ***A path of T:*** *is a sequence of nodes such that any two consecutive nodes in the sequence form an edge.* (Ex: C,A,B,F,K)
- **Ordered Trees:** A tree is *ordered if there is a meaningful linear order among the children of each node.*



Java Interface

Methods for a Tree interface:

```
1  /** An interface for a tree where nodes can have an arbitrary number of children. */
2  public interface Tree<E> extends Iterable<E> {
3    Position<E> root();
4    Position<E> parent(Position<E> p) throws IllegalArgumentException;
5    Iterable<Position<E>> children(Position<E> p)
6                                  throws IllegalArgumentException;
7    int numChildren(Position<E> p) throws IllegalArgumentException;
8    boolean isInternal(Position<E> p) throws IllegalArgumentException;
9    boolean isExternal(Position<E> p) throws IllegalArgumentException;
10   boolean isRoot(Position<E> p) throws IllegalArgumentException;
11   int size();
12   boolean isEmpty();
13   Iterator<E> iterator();
14   Iterable<Position<E>> positions();
15 }
```

Computing Depth

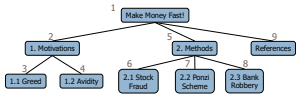
- Let p be a *position within tree T. The* ***depth of p is the number of ancestors*** of p, other than p itself.
- The depth of p can be recursively defined as follows:
  - If p is the root, then the depth of p is 0.
  - Otherwise, the depth of p is one plus the depth of the parent of p.
- The running time of depth(p) for position p is $O(d_p + 1)$, where d_p denotes the depth of p in the tree.

```
/** Returns the number of levels separating Position p
from the root. */
public int depth(Position<E> p) {
if (isRoot(p))
    return 0;
else
    return 1 + depth(parent(p));
}
```


Preorder Traversal

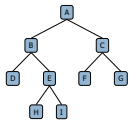
- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

Algorithm *preOrder*(v)
  visit(v)
  for each child w of v
    preorder (w)



Binary Trees

- A binary tree is a tree with the following properties:
  - Each internal node has at most two children (exactly two for proper binary trees)
  - The children of a node are an ordered pair
- We call the children of an internal node left child and right child
- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, or
  - a tree whose root has an ordered pair of children, each of which is a binary tree
- Applications:
  - arithmetic expressions
  - decision processes
  - searching



Computing Height

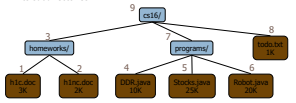
- The *height of a tree to be equal to the maximum of the depths of* its positions (or zero, if the tree is empty).

```
/** Returns the height of the tree. */
private int heightBad( ) { // works, but quadratic worst-case time
    int h = 0;
    for (Position<E> p : positions( ))
        if (isExternal(p)) // only consider leaf positions
            h = Math.max(h, depth(p));
    return h;
}
```


Postorder Traversal

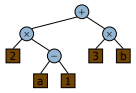
- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

Algorithm *postOrder*(v)
  for each child w of v
    postOrder (w)
  visit(v)



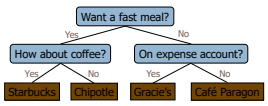
Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression $(2 \times (a - 1) + (3 \times b))$



Decision Tree

- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: dining decision



BinaryTree ADT

- The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT
- Additional methods:
  - position left(p)
  - position right(p)
  - position sibling(p)
- The above methods return null when there is no left, right, or sibling of p, respectively
- Update methods may be defined by data structures implementing the BinaryTree ADT

Preorder Traversal

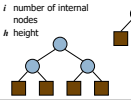
- In a ***preorder traversal of a tree T, the root of T is visited first and then the subtrees*** rooted at its children are traversed recursively. If the tree is ordered, then the subtrees are traversed according to the order of the children.

Algorithm preorder(*p*):
  perform the "visit" action for position p *{ this happens before any recursion }*
  for each child *c in children(p)* **do**
    preorder(*c*) *{ recursively traverse the subtree rooted at c }*

Properties of Proper Binary Trees

A full **binary tree** (sometimes **proper binary tree** or 2-**tree**) is a **tree** in which every node other than the leaves has two children. A complete **binary tree** is a **binary tree** in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

- Notation
  - n number of nodes
  - e number of external nodes
  - i number of internal nodes
  - h height



- Properties of Proper Binary Trees:
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$
  - $h \geq \log_2 e$
  - $h \geq \log_2 (n + 1) - 1$

Three Traversal Algorithms

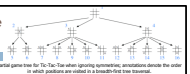
- A ***traversal of a tree T is a systematic way of accessing, or "visiting," all the positions*** of T.
- *The specific action associated with the "visit" of a position p depends* on the application of this traversal, and could involve anything from incrementing a counter to performing some complex computation for p.

Postorder

- Another important tree traversal algorithm is the ***postorder traversal. In some*** sense, this algorithm can be viewed as the opposite of the preorder traversal, because it recursively traverses the subtrees rooted at the children of the root first, and then visits the root (hence, the name "postorder").

Algorithm postorder(*p*):
  for each child *c in children(p)* **do**
    postorder(*c*) *{ recursively traverse the subtree rooted at c }*
  perform the "visit" action for position p *{ this happens after any recursion }*

Breadth-First Tree Traversal


Partial game tree for Tic-Tac-Toe when ignoring symmetries; annotations denote the order in which positions are visited in a breadth-first tree traversal.

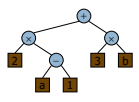
- Although the preorder and postorder traversals are common ways of visiting the positions of a tree, another approach is to traverse a tree so that we visit all the positions at depth d *before we visit the positions at depth* $d+1$. *Such an algorithm* is known as a *breadth-first traversal*.

Algorithm breadthfirst( ):
  Initialize queue Q *to contain root( )*
  while Q *not empty* **do**
    p = Q.dequeue( ) *{ p is the oldest entry in the queue }*
    perform the "visit" action for position p
    for each child *c in children(p)* **do**
      Q.enqueue(*c*) *{ add p's children to the end of the queue for later visits }*

Print Arithmetic Expressions

- Specialization of an inorder traversal
  - print operand or operator when visiting node
  - print "(" before traversing left subtree
  - print ")" after traversing right subtree

Algorithm *printExpression*(v)
  if *left* (v) $\neq$ **null**
    print("(")
    inOrder (*left*(v))
  print(*v*.*element* ())
  if *right*(v) $\neq$ **null**
    inOrder (*right*(v))
    print (")")

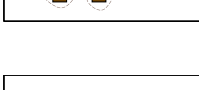


$((2 \times (a - 1)) + (3 \times b))$

Euler Tour Traversal

- Generic traversal of a binary tree
- Includes a special cases the preorder, postorder and inorder traversals
- Walk around the tree and visit each node three times:
  - on the left (preorder)
  - from below (inorder)
  - on the right (postorder)

Algorithm eulerTour(*T, p*):
  perform the "pre visit" action for position p
  for each child *c in T.children(p)* **do**
    eulerTour(*T, c*) *{ recursively tour the subtree rooted at c }*
  perform the "post visit" action for position p



Inorder Traversal

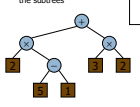
- In an inorder traversal a node is visited after its left subtree and before its right subtree
- Application: draw a binary tree
  - x(v) = inorder rank of v
  - y(v) = depth of v

Algorithm *inOrder*(v)
  if *left* (v) $\neq$ **null**
    inOrder (*left* (v))
  visit(v)
  if *right*(v) $\neq$ **null**
    inOrder (*right* (v))

Evaluate Arithmetic Expressions

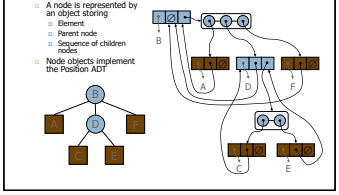
- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

Algorithm *evalExpr*(v)
  if *isExternal* (v)
    return v.*element* ()
  else
    $x \leftarrow$ *evalExpr*(*left*(v))
    $y \leftarrow$ *evalExpr*(*right*(v))
    $\diamond \leftarrow$ operator stored at v
    return $x \diamond y$



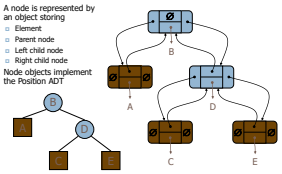
Linked Structure for Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes
- Node objects implement the Position ADT



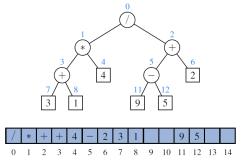
Linked Structure for Binary Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node
- Node objects implement the Position ADT



Ex. Binary Tree Array Representation

Representation of a binary tree by means of an array.

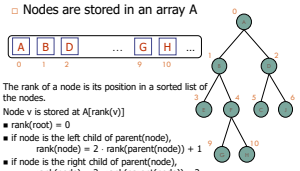


Hw.

- Two ordered trees *T′ and T″ are said to be* ***isomorphic if one of the following*** holds:
- Both *T′ and T″ are empty.*
- Both *T′ and T″ consist of a single node*
- The roots of *T′ and T″ have the same number k ≥ 1 of subtrees, and the i th such subtree of T′ is isomorphic to the i th such subtree of T″ for i = 1, . . . ,k.*
- Design and implement an algorithm that tests whether two given ordered trees are isomorphic. What is the running time of your algorithm?

Array-Based Representation of Binary Trees

- Nodes are stored in an array A



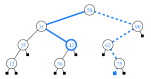
The rank of a node is its position in a sorted list of the nodes.

Node v is stored at A[rank(v)]

- rank(root) = 0
- if node is the left child of parent(node), rank(node) = 2 · rank(parent(node)) + 1
- if node is the right child of parent(node), rank(node) = 2 · rank(parent(node)) + 2

Binary Search Trees

- An important application of the inorder traversal algorithm arises when we store an ordered sequence of elements in a binary tree, defining a structure we call a ***binary search tree.***
- *Let S be a set whose unique elements have an order relation. For example, S could be a set of integers. A binary search tree for S is a proper binary tree T such that, for each internal position p of T:*
  - *Position p stores an element of S, denoted as e(p).*
  - *Elements stored in the left subtree of p (if any) are less than e(p).*
  - *Elements stored in the right subtree of p (if any) are greater than e(p).*



A binary search tree storing integers. The solid path is traversed when searching (successfully) for 42. The dashed path is traversed when searching (unsuccessfully) for 70.

PRIORITY QUEUES

Priorities

- Sometimes the ***First-In, First-Out (FIFO)*** *principle is not enough and the removal is based on some priorities.*
  - ***Ex:*** Air-traffic control center that has to decide which flight to clear for landing from among many approaching the airport. This choice may be influenced by factors such as each plane's distance from the runway, time spent waiting in a holding pattern, or amount of remaining fuel. (By the way some of these values constantly change!!!)
- A new abstract data type known as a *priority queue can be used to handle priority situations.*

Entry ADT

- An entry in a priority queue is simply a key-value pair
- Priority queues store entries to allow for efficient insertion and removal based on keys
- Methods:
  - getKey: returns the key for this entry
  - getValue: returns the value associated with this entry
- As a Java interface:

```
// Interface for a key-value pair entry
public interface Entry<K,V> {
    K getKey();
    V getValue();
}
```

Priority Queue ADT

- A priority queue stores a collection of entries
- Main methods of the Priority Queue ADT
  - **insert(k, v)**
    inserts an entry with key k and value v
  - **removeMin()**
    removes and returns the entry with smallest key, or null if the priority queue is empty
- Additional methods
  - **min()**
    returns, but does not remove, an entry with smallest key, or null if the priority queue is empty
  - **size(), isEmpty()**
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

Example

- A sequence of priority queue methods:

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min( )	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin( )	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin( )	(5,A)	{ (7,D), (9,C) }
removeMin( )	(7,D)	{ (9,C) }
removeMin( )	(9,C)	{ }
removeMin( )	null	{ }
isEmpty( )	true	{ }

Comparing Keys with Total Order Relations

- Keys in a priority queue can be arbitrary objects on which an order is defined (*must be able to compare keys to each other in a meaningful way)*
- Two distinct entries in a priority queue can have the same key
- Mathematical concept of total order relation ≤
  - Comparability property: either $x \leq y$ or $y \leq x$
  - Antisymmetric property: $x \leq y$ and $y \leq x \Rightarrow x = y$
  - Transitive property: $x \leq y$ and $y \leq z \Rightarrow x \leq z$

Comparator ADT

- A comparator encapsulates the action of comparing two objects according to a given total order relation
- A generic priority queue uses an auxiliary comparator
- The comparator is external to the keys being compared
- When the priority queue needs to compare two keys, it uses its comparator

- Primary method of the Comparator ADT
- **compare(x, y):** returns an integer i such that
  - **i < 0 if a < b,**
  - **i = 0 if a = b**
  - **i > 0 if a > b**
  - An error occurs if a and b cannot be compared.

Ex: Lexicographic Comparison of 2-D Points

```
/** Comparator for 2D points under the standard
    lexicographic order. */
public class  Lexicographic  implements
    Comparator  {
      int  xa, ya, xb, yb;
public int  compare(Object a, Object b)  throws
  ClassCastException  {
        xa = ((Point2D) a).getX();
        ya = ((Point2D) a).getY();
        xb = ((Point2D) b).getX();
        yb = ((Point2D) b).getY();
        if  (xa != xb)
 return  (xb - xa);
        else
 return  (yb - ya);
    }  }
```

- Point objects:

```
/** Class representing a point in the plane
  with integer coordinates */
public class  Point2D{
    protected int xc, yc; // coordinates
    public  Point2D(int  x,  int  y)  {
       xc = x;
       yc = y;
    }
    public int  getX()  {
return  xc;
    }
    public int  getY()  {
return  yc;
    }
}
```

Sequence-based Priority Queue

- Implementation with an unsorted list

- Implementation with a sorted list

Unsorted List Implementation

```
/** An implementation of a priority queue with an unsorted list. */
public class UnsortedPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
/** primary collection of priority queue entries */
private PositionalList<Entry<K,V>> list = new LinkedPositionalList<>( );
/** Creates an empty priority queue based on the natural ordering of its keys.
*/
public UnsortedPriorityQueue( ) { super( ); }
/** Creates an empty priority queue using the given comparator to order keys. */
public UnsortedPriorityQueue(Comparator<K> comp) { super(comp); }
/** Returns the Position of an entry having minimal key. */
private Position<Entry<K,V>> findMin( ) { // only called when nonempty
Position<Entry<K,V>> small = list.first( );
for (Position<Entry<K,V>> walk : list.positions( ))
if (compare(walk.getElement( ), small.getElement( )) < 0)
small = walk; // found an even smaller key
return small;
 }
```

Unsorted List Implementation, 2

```
/** Inserts a key-value pair and returns the entry created. */
public Entry<K,V> insert(K key,V value) throws IllegalArgumentException
{checkKey(key);// auxiliary key-checking method (could throw exception)
     Entry<K,V> newest = new PQEntry<>(key, value);
     list.addLast(newest);
     return newest;
 }
/** Returns (but does not remove) an entry with minimal key. */
public Entry<K,V> min( ) {
     if (list.isEmpty( )) return null;
      return findMin( ).getElement( );
 }
/** Removes and returns an entry with minimal key.*/
public Entry<K,V> removeMin( ) {
     if (list.isEmpty( )) return null;
     return list.remove(findMin( ));
}
/** Returns the number of items in the priority queue. */
public int size( ) { return list.size( ); } }
```

Sorted List Implementation

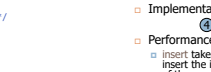
```
/** An implementation of a priority queue with a sorted list. */
public class SortedPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
/** primary collection of priority queue entries */
private PositionalList<Entry<K,V>> list = new LinkedPositionalList<>( );
/** Creates an empty priority queue based on the natural ordering of its keys.
*/
public SortedPriorityQueue( ) { super( ); }
/** Creates an empty priority queue using the given comparator to order keys. */
public SortedPriorityQueue(Comparator<K> comp) { super(comp); }
/** Inserts a key-value pair and returns the entry created. */
public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
     checkKey(key); // auxiliary key-checking method (could throw exception)
     Entry<K,V> newest = new PQEntry<>(key, value);
     Position<Entry<K,V>> walk = list.last( );
     // walk backward, looking for smaller key
     while (walk != null && compare(newest, walk.getElement( )) < 0)
         walk = list.before(walk);
     if (walk == null)
         list.addFirst(newest); // new key is smallest
     else
         list.addAfter(walk, newest); // newest goes after walk
     return newest; }
```

Sorted List Implementation, 2

```
/** Returns (but does not remove) an entry with minimal key. */
public Entry<K,V> min( ) {
    if (list.isEmpty( )) return null;
        return list.first( ).getElement( );
}
/** Removes and returns an entry with minimal key. */
public Entry<K,V> removeMin( ) {
    if (list.isEmpty( )) return null;
        return list.remove(list.first( ));
}
/** Returns the number of items in the priority queue. */
public int size( ) { return list.size( ); } }
```

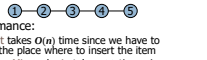
Sequence-based Priority Queue

- Implementation with an unsorted list



- Performance:
  - insert takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - removeMin and min take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

- Implementation with a sorted list



- Performance:
  - insert takes $O(n)$ time since we have to find the place where to insert the item
  - removeMin and min take $O(1)$ time, since the smallest key is at the beginning

Method	Unsorted List	Sorted List
size	$O(1)$	$O(1)$
isEmpty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min	$O(n)$	$O(1)$
removeMin	$O(n)$	$O(1)$

Worst-case running times of the methods of a priority queue of size n, realized by means of an unsorted and sorted, doubly linked list. The space requirement is O(n).

Priority Queue Sorting

- We can use a priority queue to sort a list of comparable elements
  1. Insert the elements one by one with a series of insert operations
  2. Remove the elements in sorted order with a series of removeMin operations
- The running time of this sorting method depends on the priority queue implementation

Algorithm *PQ-Sort*(S, C)
Input list S, comparator C for the elements of S
Output list S sorted in increasing order according to C
$P \leftarrow$ priority queue with comparator C
while $\neg S.isEmpty()$
$e \leftarrow S.remove(S.first())$
$P.insert(e, \emptyset)$
while $\neg P.isEmpty()$
$e \leftarrow P.removeMin().getKey()$
$S.addLast(e)$

Selection-Sort

- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence
- Running time of Selection-sort:
  1. Inserting the elements into the priority queue with n insert operations takes $O(n)$ time
  2. Removing the elements in sorted order from the priority queue with n removeMin operations takes time proportional to
     $$1 + 2 + \ldots + n$$
- Selection-sort runs in $O(n^2)$ time

Selection-Sort Example

	Sequence S	Priority Queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1 $O(1)$		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(7,4)
..	..	..
(g)	()	(7,4,8,2,5,3,9)
Phase 2 ($O(n+(n-1)+\cdots+2+1) = n(n+1)/2) = O(n^2)$		
(a)	(2)	(7,4,8,5,3,9)
(b)	(2,3)	(7,4,8,5,9)
(c)	(2,3,4)	(7,8,5,9)
(d)	(2,3,4,5)	(7,8,9)
(e)	(2,3,4,5,7)	(8,9)
(f)	(2,3,4,5,7,8)	(9)
(g)	(2,3,4,5,7,8,9)	()

Insertion-Sort

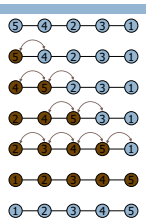
- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence
- Running time of Insertion-sort:
  1. Inserting the elements into the priority queue with n insert operations takes time proportional to
     $$1 + 2 + \ldots + n$$
  2. Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes $O(n)$ time
- Insertion-sort runs in $O(n^2)$ time

Insertion-Sort Example

	Sequence S	Priority queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1	($O(n+(n-1)+\cdots+2+1) = n(n+1)/2) = O(n^2)$	
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(4,7)
(c)	(2,5,3,9)	(4,7,8)
(d)	(5,3,9)	(2,4,7,8)
(e)	(3,9)	(2,4,5,7,8)
(f)	(9)	(2,3,4,5,7,8)
(g)	()	(2,3,4,5,7,8,9)
Phase 2	O(1)	
(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
..	..	..
(g)	(2,3,4,5,7,8,9)	()

In-place Insertion-Sort

- Instead of using an external data structure, we can implement selection-sort and insertion-sort in-place
- A portion of the input sequence itself serves as the priority queue
- For in-place insertion-sort
  - We keep sorted the initial portion of the sequence
  - We can use swaps instead of modifying the sequence



Recall PQ Sorting

- We use a priority queue
  - Insert the elements with a series of insert operations
  - Remove the elements in sorted order with a series of removeMin operations
- The running time depends on the priority queue implementation:
  - Unsorted sequence gives selection-sort: $O(n^2)$ time
  - Sorted sequence gives insertion-sort: $O(n^2)$ time
- **Can we do better?**

Algorithm *PQ-Sort*(S, C)
Input sequence S, comparator C for the elements of S
Output sequence S sorted in increasing order according to C
$P \leftarrow$ priority queue with comparator C
while $\neg S.isEmpty()$
$e \leftarrow S.remove(S.first())$
$P.insert(e, e)$
while $\neg P.isEmpty()$
$e \leftarrow P.removeMin().getKey()$
$S.addLast(e)$

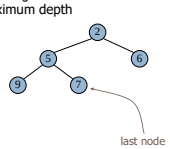
Heaps

- A more efficient realization of a priority queue is possible with using a data structure called a ***binary heap.***
- ***This data structure allows us to perform both*** insertions and removals in logarithmic time.
- The fundamental way the heap achieves this improvement is to use **the structure of a binary tree to find a compromise between elements being entirely unsorted and perfectly sorted.**

Heaps

A full **binary tree** (sometimes **proper binary tree** or 2-**tree**) is a **tree** in which every node other than the leaves has two children. A **complete binary tree** is a **binary tree** in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.

- A heap is a binary tree storing keys at its nodes and satisfying the following properties:
- **Heap-Order: for every internal node v other than the root,** $key(v) \geq key(parent(v))$
- Complete Binary Tree: let h be the height of the heap
  - for $i = 0, \ldots, h-1$, there are 2^i nodes of depth i
  - at depth $h-1$, the internal nodes are to the left of the external nodes
- The last node of a heap is the rightmost node of maximum depth



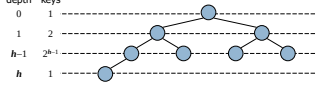
For the sake of efficiency, as will become clear later, we want the heap T to have as small a height as possible. We enforce this requirement by insisting that the heap T satisfy an additional structural property; it must be what we term complete.

Height of a Heap

- Theorem: A heap storing n keys has height $\lfloor \log n \rfloor$ *Proposition 9.2*

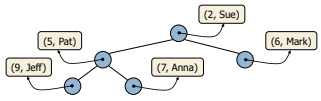
Proof: (we apply the complete binary tree property)
- Let h be the height of a heap storing n keys
- Since there are 2^i keys at depth $i = 0, \ldots, h-1$ and at least one key at depth h, we have $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$
- Thus, $n \geq 2^h$, i.e., $h \leq \log n$

$$h = \lfloor \log n \rfloor$$



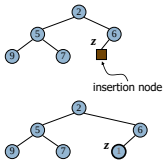
Heaps and Priority Queues

- We can use a heap to implement a priority queue
- We store a (key, element) item at each internal node
- We keep track of the position of the last node



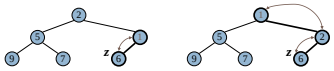
Insertion into a Heap

- Method insertItem of the priority queue ADT corresponds to the insertion of a key k to the heap
- The insertion algorithm consists of three steps
  - Find the insertion node z (the new last node)
  - Store k at z
  - Restore the heap-order property (discussed next)



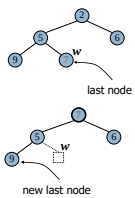
Upheap

- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time



Removal from a Heap

- Method removeMin of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node w
  - Remove w
  - Restore the heap-order property (discussed next)

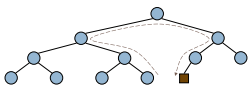


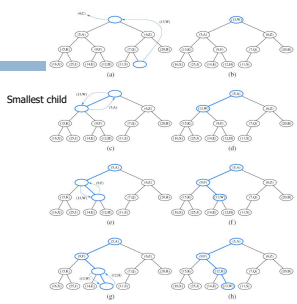
Downheap

- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key k along a downward path from the root
- Algorithm terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time

Updating the Last Node

- The insertion node can be found by traversing a path of $O(\log n)$ nodes
  - Go up until a left child or the root is reached
  - If a left child is reached, go to the right child
  - Go down left until a leaf is reached
- Similar algorithm for updating the last node after a removal





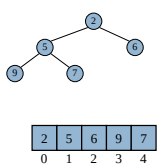
Heap-Sort

- Consider a priority queue with n items implemented by means of a heap
  - the space used is $O(n)$
  - methods insert and removeMin take $O(\log n)$ time
  - methods size, isEmpty, and min take time $O(1)$ time

- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

Array-Based Heap Implementation

- We can represent a heap with n keys by means of an array of length n
- For the node at rank i
  - the left child is at rank $2i + 1$
  - the right child is at rank $2i + 2$
- Links between nodes are not explicitly stored
- Operation add corresponds to inserting at rank $n + 1$
- Operation remove_min corresponds to removing at rank n
- Yields in-place heap-sort



Java Implementation

```
/** An implementation of a priority queue using an array-based heap. */
public class HeapPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
  /** primary collection of priority queue entries */
  protected ArrayList<Entry<K,V>> heap = new ArrayList<>();
  /** Creates an empty priority queue based on the natural ordering of its keys. */
  public HeapPriorityQueue( ) { super(); }
  /** Creates an empty priority queue using the given comparator to order keys. */
  public HeapPriorityQueue(Comparator<K> comp) { super(comp); }
  // protected utilities
  protected int parent(int j) { return (j-1) / 2; }          // truncating division
  protected int left(int j) { return 2*j + 1; }
  protected int right(int j) { return 2*j + 2; }
  protected boolean hasLeft(int j) { return left(j) < heap.size(); }
  protected boolean hasRight(int j) { return right(j) < heap.size(); }
  /** Exchanges the entries at indices i and j of the array list. */
  protected void swap(int i, int j) {
    Entry<K,V> temp = heap.get(i);
    heap.set(i, heap.get(j));
    heap.set(j, temp);
  }
  /** Moves the entry at index j higher, if necessary, to restore the heap property. */
  protected void upheap(int j) {
    while (j > 0) {                          // continue until reaching root (or break statement)
      int p = parent(j);
      if (compare(heap.get(j), heap.get(p)) >= 0) break;   // heap property verified
      swap(j, p);
      j = p;                                 // continue from the parent's location
    }
  }
```

Java Implementation

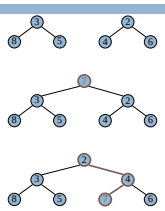
```
  /** Moves the entry at index j lower, if necessary, to restore the heap property. */
  protected void downheap(int j) {
    while (hasLeft(j)) {                     // continue to bottom (or break statement)
      int leftIndex = left(j);
      int smallChildIndex = leftIndex;       // although right may be smaller
      if (hasRight(j)) {
        int rightIndex = right(j);
        if (compare(heap.get(leftIndex), heap.get(rightIndex)) > 0)
          smallChildIndex = rightIndex;      // right child is smaller
      }
      if (compare(heap.get(smallChildIndex), heap.get(j)) >= 0)
        break;                               // heap property has been restored
      swap(j, smallChildIndex);
      j = smallChildIndex;                   // continue at position of the child
    }
  }

  // public methods
  /** Returns the number of items in the priority queue. */
  public int size( ) { return heap.size( ); }
  /** Returns (but does not remove) an entry with minimal key (if any). */
  public Entry<K,V> min( ) {
    if (heap.isEmpty( )) return null;
    return heap.get(0);
  }
```

Java Implementation, 3

```
  /** Inserts a key-value pair and returns the entry created. */
  public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
    checkKey(key);          // auxiliary key-checking method (could throw exception)
    Entry<K,V> newest = new PQEntry<>(key, value);
    heap.add(newest);                        // add to the end of the list
    upheap(heap.size( ) - 1);                // upheap newly added entry
    return newest;
  }
  /** Removes and returns an entry with minimal key (if any). */
  public Entry<K,V> removeMin( ) {
    if (heap.isEmpty( )) return null;
    Entry<K,V> answer = heap.get(0);
    swap(0, heap.size( ) - 1);               // put minimum item at the end
    heap.remove(heap.size( ) - 1);           // and remove it from the list;
    downheap(0);                             // then fix new root
    return answer;
  }
}
```

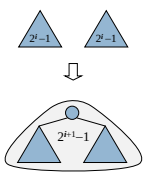
Merging Two Heaps

- We are given two two heaps and a key k
- We create a new heap with the root node storing k and with the two heaps as subtrees
- We perform downheap to restore the heap-order property

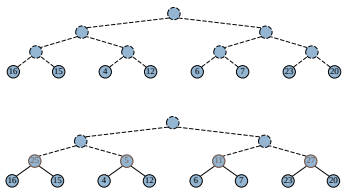


Bottom-up Heap Construction

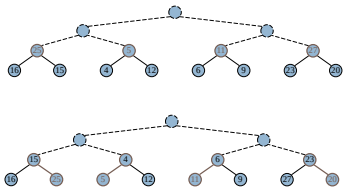
- We can construct a heap storing n given keys in using a bottom-up construction with $\log n$ phases
- In phase i, pairs of heaps with 2^i-1 keys are merged into heaps with $2^{i+1}-1$ keys



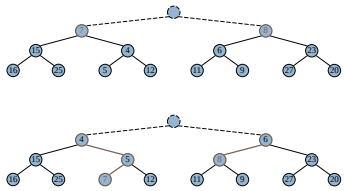
Example



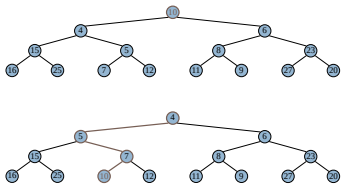
Example (contd.)



Example (contd.)

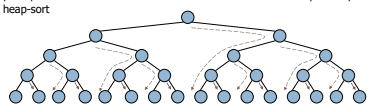


Example (end)

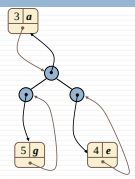


Analysis

- We visualize the worst-case time of a downheap with a proxy path that goes first right and then repeatedly goes left until the bottom of the heap (this path may differ from the actual downheap path)
- Since each node is traversed by at most two proxy paths, the total number of nodes of the proxy paths is $O(n)$
- Thus, bottom-up heap construction runs in $O(n)$ time
- Bottom-up heap construction is faster than n successive insertions and speeds up the first phase of heap-sort



Adaptable Priority Queues



Entry and Priority Queue ADTs

- An entry stores a (key, value) pair
- Entry ADT methods:
  - getKey(): returns the key associated with this entry
  - getValue(): returns the value paired with the key associated with this entry
- Priority Queue ADT:
  - insert(k, x)
    inserts an entry with key k and value x
  - removeMin()
    removes and returns the entry with smallest key
  - min()
    returns, but does not remove, an entry with smallest key
  - size(), isEmpty()

Example

- Online trading system where orders to purchase and sell a stock are stored in two priority queues (one for sell orders and one for buy orders) as (p,s) entries:
  - The key, p, of an order is the price
  - The value, s, for an entry is the number of shares
  - A buy order (p,s) is executed when a sell order (p',s') with price p' $\leq$p is added (the execution is complete if s' $\geq$s)
  - A sell order (p,s) is executed when a buy order (p',s') with price p' $\geq$p is added (the execution is complete if s' $\geq$s)
- What if someone wishes to cancel their order before it executes?
- What if someone wishes to update the price or number of shares for their order?

Methods of the Adaptable Priority Queue ADT

- remove(e): Remove from P and return entry e.
- replaceKey(e,k): Replace with k and return the key of entry e of P; an error condition occurs if k is invalid (that is, k cannot be compared with other keys).
- replaceValue(e,v): Replace with v and return the value of entry e of P.

Example

Operation	*Output*	*P*
insert(5,*A*)	e_1	(5,*A*)
insert(3,*B*)	e_2	(3,*B*),(5,*A*)
insert(7,*C*)	e_3	(3,*B*),(5,*A*),(7,*C*)
min()	e_2	(3,*B*),(5,*A*),(7,*C*)
key(e_2)	3	(3,*B*),(5,*A*),(7,*C*)
remove(e_1)	e_1	(3,*B*),(7,*C*)
replaceKey(e_2,9)	3	(7,*C*),(9,*B*)
replaceValue(e_2,*D*)	*C*	(7,*D*),(9,*B*)
remove(e_2)	e_2	(7,*D*)

Locating Entries

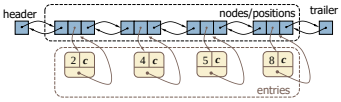
- In order to implement the operations remove(e), replaceKey(e,k), and replaceValue(e,v), we need fast ways of locating an entry e in a priority queue.
- We can always just search the entire data structure to find an entry e, but there are better ways for locating entries.

Location-Aware Entries

- A location-aware entry identifies and tracks the location of its (key, value) object within a data structure
- Intuitive notion:
  - Coat claim check
  - Valet claim ticket
  - Reservation number
- Main idea:
  - Since entries are created and returned from the data structure itself, it can return location-aware entries, thereby making future updates easier

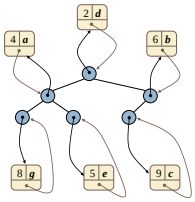
List Implementation

- A location-aware list entry is an object storing
  - key
  - value
  - position (or rank) of the item in the list
- In turn, the position (or array cell) stores the entry
- Back pointers (or ranks) are updated during swaps



Heap Implementation

- A location-aware heap entry is an object storing
  - key
  - value
  - position of the entry in the underlying heap
- In turn, each heap position stores an entry
- Back pointers are updated during entry swaps



Performance

- Improved times thanks to location-aware entries are highlighted in red

Method	Unsorted List	Sorted List	Heap
size, isEmpty	$O(1)$	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$	$O(\log n)$
min	$O(n)$	$O(1)$	$O(1)$
removeMin	$O(n)$	$O(1)$	$O(\log n)$
remove	$O(1)$	$O(1)$	$O(\log n)$
replaceKey	$O(1)$	$O(n)$	$O(\log n)$
replaceValue	$O(1)$	$O(1)$	$O(1)$

Java Implementation

```
/** An implementation of an adaptable priority queue using an array-based heap. */
public class HeapAdaptablePriorityQueue<K,V> extends HeapPriorityQueue<K,V>
                                  implements AdaptablePriorityQueue<K,V> {

  //---------- nested AdaptablePQEntry class ----------
  /** Extension of the PQEntry to include location information. */
  protected static class AdaptablePQEntry<K,V> extends PQEntry<K,V> {
    private int index;                  // entry's current index within the heap
    public AdaptablePQEntry(K key, V value, int j) {
      super(key, value);                // this sets the key and value
      index = j;                        // this sets the new field
    }
    public int getIndex() { return index; }
    public void setIndex(int j) { index = j; }
  } //---------- end of nested AdaptablePQEntry class ----------

  /** Creates an empty adaptable priority queue using natural ordering of keys. */
  public HeapAdaptablePriorityQueue() { super(); }
  /** Creates an empty adaptable priority queue using the given comparator. */
  public HeapAdaptablePriorityQueue(Comparator<K> comp) { super(comp);}
```

Java Implementation, 2

```
  // protected utilites
  /** Validates an entry to ensure it is location-aware. */
  protected AdaptablePQEntry<K,V> validate(Entry<K,V> entry)
                                  throws IllegalArgumentException {
    if (!(entry instanceof AdaptablePQEntry))
      throw new IllegalArgumentException("Invalid entry");
    AdaptablePQEntry<K,V> locator = (AdaptablePQEntry<K,V>) entry;   // safe
    int j = locator.getIndex();
    if (j >= heap.size() || heap.get(j) != locator)
      throw new IllegalArgumentException("Invalid entry");
    return locator;
  }

  /** Exchanges the entries at indices i and j of the array list. */
  protected void swap(int i, int j) {
    super.swap(i,j);                        // perform the swap
    ((AdaptablePQEntry<K,V>) heap.get(i)).setIndex(i);   // reset entry's index
    ((AdaptablePQEntry<K,V>) heap.get(j)).setIndex(j);   // reset entry's index
  }
```

Java Implementation, 3

```
  /** Restores the heap property by moving the entry at index j upward/downward.*/
  protected void bubble(int j) {
    if (j > 0 && compare(heap.get(j), heap.get(parent(j))) < 0)
      upheap(j);
    else
      downheap(j);                          // although it might not need to move
  }

  /** Inserts a key-value pair and returns the entry created. */
  public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
    checkKey(key);                          // might throw an exception
    Entry<K,V> newest = new AdaptablePQEntry<>(key, value, heap.size());
    heap.add(newest);                       // add to the end of the list
    upheap(heap.size( ) – 1);               // upheap newly added entry
    return newest;
  }
```

Java Implementation, 4

```
  /** Removes the given entry from the priority queue. */
  public void remove(Entry<K,V> entry) throws IllegalArgumentException {
    AdaptablePQEntry<K,V> locator = validate(entry);
    int j = locator.getIndex( );
    if (j == heap.size( ) – 1)              // entry is at last position
      heap.remove(heap.size( ) – 1);        // so just remove it
    else {
      swap(j, heap.size( ) – 1);            // swap entry to last position
      heap.remove(heap.size( ) – 1);        // then remove it
      bubble(j);                            // and fix entry displaced by the swap
    }
  }

  /** Replaces the key of an entry. */
  public void replaceKey(Entry<K,V> entry, K key)
                                  throws IllegalArgumentException {
    AdaptablePQEntry<K,V> locator = validate(entry);
    checkKey(key);                          // might throw an exception
    locator.setKey(key);                    // method inherited from PQEntry
    bubble(locator.getIndex( ));            // with new key, may need to move entry
  }

  /** Replaces the value of an entry. */
  public void replaceValue(Entry<K,V> entry, V value)
                                  throws IllegalArgumentException {
    AdaptablePQEntry<K,V> locator = validate(entry);
    locator.setValue(value);                // method inherited from PQEntry
  }
```


Maps

Maps

- A map models a searchable collection of key-value entries
- The main operations of a map are for searching, inserting, and deleting items
- Multiple entries with the same key are not allowed
- Applications:
  - address book
  - A university's information system relies on some form of a student ID as a key that is mapped to that student's associated record (such as the student's name, address, and course grades) serving as the value.
  - The domain-name system (DNS)maps a host name, such as www.wiley.com, to an Internet-Protocol (IP) address, such as 208.215.179.146

The Map ADT

- get(k): if the map M has an entry with key k, return its associated value; else, return null
- put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- size(), isEmpty()
- **entrySet():** return an iterable collection of the entries in M
- **keySet():** return an iterable collection of the keys in M
- **values():** return an iterator of the values in M

java.util.map Example

```
// Java program to demonstrate
// the working of Map interface

import java.util.*;
class HashMapDemo {
    public static void main(String args[])
    {
        Map<String, Integer> hm
            = new HashMap<String, Integer>();

        hm.put("a", new Integer(100));
        hm.put("b", new Integer(200));
        hm.put("c", new Integer(300));
        hm.put("d", new Integer(400));

        // Traversing through the map
        for (Map.Entry<String, Integer> me : hm.entrySet()) {
            System.out.print(me.getKey() + ":");
            System.out.println(me.getValue());
        }
    }
}
```

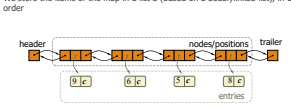

Example

Operation	Output	Map
isEmpty()	true	Ø
put(5,A)	null	(5,A)
put(7,B)	null	(5,A),(7,B)
put(2,C)	null	(5,A),(7,B),(2,C)
put(8,D)	null	(5,A),(7,B),(2,C),(8,D)
put(2,E)	C	(5,A),(7,B),(2,E),(8,D)
get(7)	B	(5,A),(7,B),(2,E),(8,D)
get(4)	null	(5,A),(7,B),(2,E),(8,D)
get(2)	E	(5,A),(7,B),(2,E),(8,D)
size()	4	(5,A),(7,B),(2,E),(8,D)
remove(5)	A	(7,B),(2,E),(8,D)
remove(2)	E	(7,B),(8,D)
get(2)	null	(7,B),(8,D)
isEmpty()	false	(7,B),(8,D)

A Simple List-Based Unsorted Map

We can implement a map using an unsorted list

- We store the items of the map in a list S (based on a doublylinked list), in arbitrary order



The get(k) Algorithm

```
Algorithm get(k):
 B = S.positions() //B is an iterator of the positions in S
 while B.hasNext() do
      p = B.next()     // the next position in B
      if p.element().getKey() = k   then
            return p.element().getValue()
 return null //there is no entry with key equal to k
```


The put(k,v) Algorithm

```
Algorithm put(k,v):
 B       = S.positions()
 while B.hasNext() do
  p = B.next()
  if p.element().getKey() = k  then
        t = p.element().getValue()
        S.set(p,(k,v))
        return t          //return the old value
 S.addLast((k,v))
 n = n + 1          //increment variable storing number of entries
 return null        // there was no entry with key equal to k
```


The remove(k) Algorithm

```
Algorithm remove(k):
 B =S.positions()
 while B.hasNext() do
  p = B.next()
  if p.element().getKey() = k  then
        t = p.element().getValue()
        S.remove(p)
        n = n − 1          //decrement number of entries
        return t           //return the removed value
 return null               //there is no entry with key equal to k
```


Performance of a List-Based Map

Performance:

- put **may** take $O(1)$ time since we can insert the new item at the beginning or at the end of the sequence. Previous implementation takes $O(n)$ time.
- get and remove take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key

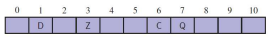
The unsorted list implementation is effective only for maps of small size or for maps in which puts are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

Hash Tables

Intuitive Notion of a Map

Intuitively, a map M supports the abstraction of using keys as indices with a syntax such as M[k].

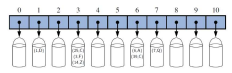
As a mental warm-up, consider a restricted setting in which a map with n items uses keys that are known to be integers in a range from 0 to N – 1, for some **N ≥ n**.



Limitations

There are two challenges in extending this framework to the more general setting of a map.

- First, not wish to devote an array of length N if it is the case that N ≫ n.
- Second, in general it is not required that a map's keys be integers.
- Would like to be able to store more than one entry in one map. (Bucket array)



A bucket array of capacity 11 with entries (1,D), (25,C), (3,F), (14,Z), (6,A), (39,C), and (7,Q), using a simple hash function.

Hash Functions and Hash Tables

A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$

Example:

$h(x) = x \bmod N$

is a hash function for integer keys

The integer $h(x)$ is called the hash value of key x

A hash table for a given key type consists of

- Hash function h
- Array (called table) of size N

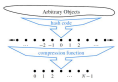
When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$

Hash Functions

A hash function is usually specified as the composition of two functions:

Hash code:
h_1: keys → integers

Compression function:
h_2: integers → $[0, N-1]$



The hash code is applied first, and the compression function is applied next on the result, i.e.,

$h(x) = h_2(h_1(x))$

The goal of the hash function is to "disperse" the keys in an apparently random way

More General Kinds of Keys

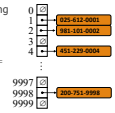
But what should we do if our keys are not integers in the range from 0 to N – 1?

- Use a **hash function** to map general keys to corresponding indices in a table.
- For instance, the last four digits of a Social Security number.

Example

We design a hash table for a map storing entries as (SSN, Name), where SSN (social security number) is a nine-digit positive integer

Our hash table uses an array of size $N = 10{,}000$ and the hash function $h(x) =$ last four digits of x



Hash Codes

Memory address:

- We reinterpret the memory address of the key object as an integer (default hash code of all Java objects)
- Good in general, except for numeric and string keys

Integer cast:

- We reinterpret the bits of the key as an integer
- Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in Java)

Component sum:

- We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
- Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in Java)

Hash Codes (cont.)

Polynomial accumulation:

- We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)
  $a_0 a_1 \ldots a_{n-1}$
- We evaluate the polynomial
  $p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$
  at a fixed value z, ignoring overflows
- Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)

Polynomial $p(z)$ can be evaluated in $O(n)$ time using Horner's rule:

- The following polynomials are successively computed, each from the previous one in $O(1)$ time
  $p_0(z) = a_{n-1}$
  $p_i(z) = a_{n-i-1} + z p_{i-1}(z)$ $(i = 1, 2, \ldots, n-1)$

We have $p(z) = p_{n-1}(z)$

Compression Functions

Division:

- $h_2(y) = y \bmod N$
- The size N of the hash table is usually chosen to be a prime
- The reason has to do with number theory and is beyond the scope of this course

Multiply, Add and Divide (MAD):

- $h_2(y) = (ay + b) \bmod N$
- a and b are nonnegative integers such that
  $a \bmod N \neq 0$
- Otherwise, every integer would map to the same value b

If a hash function is chosen well, it should ensure that the probability of two different keys getting hashed to the same bucket is 1/N.

Open Addressing

The separate chaining rule requires the use of an auxiliary data structure to hold entries with colliding keys.

If space is at a premium (for example, if we are writing a program for a small handheld device), then we can use the alternative approach of storing each entry directly in a table slot.

This approach saves space because no auxiliary structures are employed, but it requires a bit more complexity to properly handle collisions.

There are several variants of this approach, collectively referred to as open addressing schemes.

Open addressing requires that the load factor is always at most 1 and that entries are stored directly in the cells of the bucket array itself.

Hash Codes (cont.)

Cyclic-Shift hash codes:

- A variant of the polynomial hash code replaces multiplication by a with a cyclic shift of a partial sum by a certain number of bits.
- For example, a 5-bit cyclic shift of the 32-bit value 00111101100101101010100010101000 is achieved by taking the leftmost five bits and placing those on the rightmost side of the representation, resulting in 10110010110101010001010100000111

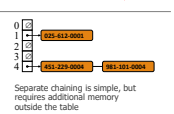
```
static int hashCode(String s)
{
int h=0;
for (int i=0; i<s.length( ); i++) {
   h = (h << 5) | (h >>> 27);
   // 5-bit cyclic shift of the running sum
   h += (int) s.charAt(i);
// add in next character
}
return h;
}
```

Collisions		
Shift	Total	Max
1	163076	63
2	76417	15
3	7174	3
4	1378	3
5	190	3
6	302	2
7	360	2
8	5946	4
9	393	3
10	5781	5
11	11539	5
12	822	2
13	900	4
14	2881	4
15	19251	8
16	211781	57

Collision Handling

Collisions occur when different elements are mapped to the same cell

Separate Chaining: let each cell in the table point to a linked list of entries that map there



Separate chaining is simple, but requires additional memory outside the table

Assuming we use a good hash function to index the n entries of our map in a bucket array of capacity N, the expected size of a bucket is n/N. The ratio lamda = n/N, called the load factor of the hash table, should be bounded by a small constant, preferably below 1. As long as lamda is O(1), the core operations on the hash table run in O(1) expected time.

Linear Probing

Open addressing: the colliding item is placed in a different cell of the table

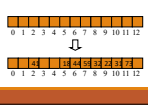
Linear probing: handles collisions by placing the colliding item in the next (circularly) available table cell

Each table cell inspected is referred to as a "probe"

Colliding items lump together, causing future collisions to cause a longer sequence of probes

Example:

- $h(x) = x \bmod 13$
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order



Search with Linear Probing

Consider a hash table A that uses linear probing

get(k)
- We start at cell $h(k)$
- We probe consecutive locations until one of the following occurs
  - An item with key k is found, or
  - An empty cell is found, or
  - N cells have been unsuccessfully probed

Algorithm *get*(k)
  $i \leftarrow h(k)$
  $p \leftarrow 0$
  repeat
    $c \leftarrow A[i]$
    if $c = \emptyset$
      return *null*
    else if *c.getKey* $() = k$
      return *c.getValue*$()$
    else
      $i \leftarrow (i + 1) \bmod N$
      $p \leftarrow p + 1$
  until $p = N$
  return *null*

Probe Hash Map in Java

Probe Hash Map in Java, 3

Updates with Linear Probing

To handle insertions and deletions, we introduce a special object, called *DEFUNCT*, which replaces deleted elements

remove(k)
- We search for an entry with key k
- If such an entry (k, o) is found, we replace it with the special item *DEFUNCT* and we return element o
- Else, we return *null*

put(k, o)
- We throw an exception if the table is full
- We start at cell $h(k)$
- We probe consecutive cells until one of the following occurs
  - A cell i is found that is either empty or stores *DEFUNCT*, or
  - N cells have been unsuccessfully probed
- We store (k, o) in cell i

Probe Hash Map in Java, 2

Double Hashing

Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series

$$(i + jd(k)) \bmod N$$

for $j = 0, 1, \ldots, N-1$

The secondary hash function $d(k)$ cannot have zero values

The table size N must be a prime to allow probing of all the cells

Common choice of compression function for the secondary hash function:

$$d_2(k) = q - k \bmod q$$

where
- $q < N$
- q is a prime

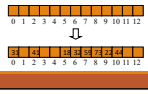
The possible values for $d_2(k)$ are $1, 2, \ldots, q$

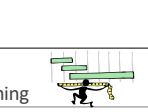
Example of Double Hashing

Consider a hash table storing integer keys that handles collision with double hashing
- $N = 13$
- $h(k) = k \bmod 13$
- $d(k) = 7 - k \bmod 7$
- $A[(h(k)+ i*d (k)) \bmod N]$ **next, for** $i = 1,2,3, \ldots$

Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes
18	5	3	5
41	2	1	2
22	9	6	9
44	5	5	5 10
59	7	4	7
32	6	3	6
31	5	4	5 9 0
73	8	4	8





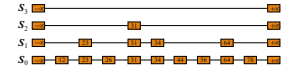
Ex: Counting Word Frequencies

```
/** A program that counts words in a document, printing the most frequent. */
public class WordCount {
    public static void main(String[] args) {
        Map<String,Integer> freq = new ChainHashMap<>( ); // or any concrete map
        // scan input for words, using all nonletters as delimiters
        Scanner doc = new Scanner(System.in).useDelimiter("[^a-zA-Z]+");
        while (doc.hasNext( )) {
            String word = doc.next( ).toLowerCase( );//convert next word to lowercase
            Integer count = freq.get(word); //get the previous count for this word
            if (count == null)
                count = 0; // if not in map, previous count is zero
            freq.put(word, 1 + count); // (re)assign new count for this word
        }
        int maxCount = 0;
        String maxWord = "no word";
        for (Entry<String,Integer> ent : freq.entrySet( )) // find max-count word
            if (ent.getValue( ) > maxCount) {
                maxWord = ent.getKey( );
                maxCount = ent.getValue( );
            }
        System.out.print("The most frequent word is '" + maxWord);
        System.out.println("' with " + maxCount + " occurrences.");
    }
}
```


What is a Skip List

A skip list for a set S of distinct (key, element) items is a series of lists $S_0, S_1, \ldots, S_h$ such that
- Each list S_i contains the special keys $+\infty$ and $-\infty$
- List S_0 contains the keys of S in nondecreasing order
- Each list is a subsequence of the previous one, i.e., $S_0 \supseteq S_1 \supseteq \ldots \supseteq S_h$
- List S_h contains only the two special keys

We show how to use a skip list to implement the map ADT



Performance of Hashing

In the worst case, searches, insertions and removals on a hash table take $O(n)$ time

The worst case occurs when all the keys inserted into the map collide

The load factor $\alpha = n/N$ affects the performance of a hash table

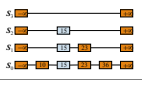
Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is $1 / (1 - \alpha)$

The expected running time of all the dictionary ADT operations in a hash table is $O(1)$

In practice, hashing is very fast provided the load factor is not close to 100%

Applications of hash tables:
- small databases
- compilers
- browser caches

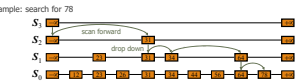
Skip Lists



Search

We search for a key x in a skip list as follows:
- We start at the first position of the top list
- At the current position p, we compare x with $y \leftarrow key(next(p))$
  - $x < y$: we return *element*(*next*(p))
  - $x > y$: we "scan forward"
  - $x = y$: we "drop down"
- If we try to drop down past the bottom list, we return *null*

Example: search for 78



Randomized Algorithms

A randomized algorithm performs coin tosses (i.e., uses random bits) to control its execution

It contains statements of the type

b ← *random*()
if *b* = 0
  do A …
else { *b* = 1}
  do B …

Its running time depends on the outcomes of the coin tosses

We analyze the expected running time of a randomized algorithm under the following assumptions
- the coins are unbiased, and
- the coin tosses are independent

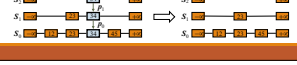
The worst-case running time of a randomized algorithm is often large but has very low probability (e.g., it occurs when all the coin tosses give "heads")

We use a randomized algorithm to insert items into a skip list

Deletion

To remove an entry with key x from a skip list, we proceed as follows:
- We search for x in the skip list and find the positions $p_0, p_1, \ldots, p_i$ of the items with key x, where position p_i is in list S_i
- We remove positions $p_0, p_1, \ldots, p_i$ from the lists $S_0, S_1, \ldots, S_i$
- We remove all but one list containing only the two special keys

Example: remove key 34



Space Usage

The space used by a skip list depends on the random bits used by each invocation of the insertion algorithm

We use the following two basic probabilistic facts:

Fact 1: The probability of getting i consecutive heads when flipping a coin is $1/2^i$

Fact 2: If each of n entries is present in a set with probability p, the expected size of the set is np

Consider a skip list with n entries
- By Fact 1, we insert an entry in list S_i with probability $1/2^i$
- By Fact 2, the expected size of list S_i is $n/2^i$

The expected number of nodes used by the skip list is

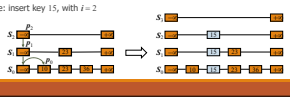
$$\sum_{i=0}^{h} \frac{n}{2^i} = n \sum_{i=0}^{h} \frac{1}{2^i} < 2n$$

- Thus, the expected space usage of a skip list with n items is $O(n)$

Insertion

To insert an entry (x, o) into a skip list, we use a randomized algorithm:
- We repeatedly toss a coin until we get tails, and we denote with i the number of times the coin came up heads
- If $i \geq h$, we add to the skip list new lists $S_{h+1}, \ldots, S_{i+1}$, each containing only the two special keys
- We search for x in the skip list and find the positions $p_0, p_1, \ldots, p_i$ of the items with largest key less than x in each list $S_0, S_1, \ldots, S_i$
- For $j \leftarrow 0, \ldots, i$, we insert item (x, o) into list S_j after position p_j

Example: insert key 15, with $i = 2$

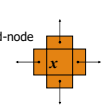


Implementation

We can implement a skip list with quad-nodes

A quad-node stores:
- entry
- link to the node prev
- link to the node next
- link to the node below
- link to the node above

Also, we define special keys PLUS_INF and MINUS_INF, and we modify the key comparator to handle them



Search and Update Times

The search time in a skip list is proportional to
- the number of drop-down steps, plus
- the number of scan-forward steps

The drop-down steps are bounded by the height of the skip list and thus are $O(\log n)$ with high probability

To analyze the scan-forward steps, we use yet another probabilistic fact:

Fact 4: The expected number of coin tosses required in order to get tails is 2

When we scan forward in a list, the destination key does not belong to a higher list
- A scan-forward step is associated with a former coin toss that gave tails

By Fact 4, in each list the expected number of scan-forward steps is 2

Thus, the expected number of scan-forward steps is $O(\log n)$

We conclude that a search in a skip list takes $O(\log n)$ expected time

The analysis of insertion and deletion gives similar results

Summary

A skip list is a data structure for maps that uses a randomized insertion algorithm

In a skip list with n entries
- The expected space used is $O(n)$
- The expected search, insertion and deletion time is $O(\log n)$

Using a more complex probabilistic analysis, one can show that these performance bounds also hold with high probability

Skip lists are fast and simple to implement in practice

Multisets and Multimaps

Definitions

A **set** is an unordered collection of elements, without duplicates that typically supports efficient membership tests.
- Elements of a set are like keys of a map, but without any auxiliary values.

A **multiset** (also known as a **bag**) is a set-like container that allows duplicates.

A **multimap** is similar to a traditional map, in that it associates values with keys; however, in a multimap the same key can be mapped to multiple values.
- For example, the index of a book maps a given term to one or more locations at which the term occurs.

add(e): Adds the element e to S (if not already present).
remove(e): Removes the element e from S (if it is present).
contains(e): Returns whether e is an element of S.
iterator(): Returns an iterator of the elements of S.

There is also support for the traditional mathematical set operations of ***union***, ***intersection***, and ***subtraction*** of two sets S and T:

$$S \cup T = \{e\text{: } e \text{ is in } S \text{ or } e \text{ is in } T\},$$
$$S \cap T = \{e\text{: } e \text{ is in } S \text{ and } e \text{ is in } T\},$$
$$S - T = \{e\text{: } e \text{ is in } S \text{ and } e \text{ is not in } T\}.$$

addAll(T): Updates S to also include all elements of set T, effectively replacing S by $S \cup T$.
retainAll(T): Updates S so that it only keeps those elements that are also elements of set T, effectively replacing S by $S \cap T$.
removeAll(T): Updates S by removing any of its elements that also occur in set T, effectively replacing S by $S - T$.

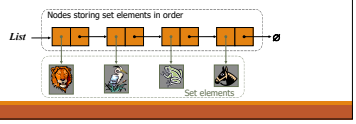
Set ADT

Storing a Set in a List

We can implement a set with a list

Elements are stored sorted according to some canonical ordering

The space used is $O(n)$



Generic Merging

Generalized merge of two sorted lists A and B

Template method genericMerge

Auxiliary methods
- aIsLess
- bIsLess
- bothAreEqual

Runs in $O(n_A + n_B)$ time provided the auxiliary methods run in $O(1)$ time

Algorithm *genericMerge*(A, B)
  $S \leftarrow$ *empty sequence*
  while $\neg A.isEmpty() \wedge \neg B.isEmpty()$
    $a \leftarrow A.first().element()$; $b \leftarrow B.first().element()$
    if $a < b$
      $aIsLess(a, S)$; $A.remove(A.first())$
    else if $b < a$
      $bIsLess(b, S)$; $B.remove(B.first())$
    else { $b = a$ }
      $bothAreEqual(a, b, S)$
      $A.remove(A.first())$; $B.remove(B.first())$
  while $\neg A.isEmpty()$
    $aIsLess(a, S)$; $A.remove(A.first())$
  while $\neg B.isEmpty()$
    $bIsLess(b, S)$; $B.remove(B.first())$
  return S

Using Generic Merge for Set Operations

Any of the set operations can be implemented using a generic merge

For example:
- For intersection: only copy elements that are duplicated in both list
- For union: copy every element from both lists except for the duplicates

All methods run in linear time

get(k): Returns a collection of all values associated with key k in the multimap.
put(k, v): Adds a new entry to the multimap associating key k with value v, without overwriting any existing mappings for key k.
remove(k, v): Removes an entry mapping key k to value v from the multimap (if one exists).
removeAll(k): Removes all entries having key equal to k from the multimap.
size(): Returns the number of entries of the multiset (including multiple associations).
entries(): Returns a collection of all entries in the multimap.
keys(): Returns a collection of keys for all entries in the multimap (including duplicates for keys with multiple bindings).
keySet(): Returns a nonduplicative collection of keys in the multimap.
values(): Returns a collection of values for all entries in the multimap.

Multimaps

Multimap

A **multimap** is similar to a map, except that it can store multiple entries with the same key

We can implement a multimap M by means of a map M'
- For every key k in M, let E(k) be the list of entries of M with key k
- The entries of M' are the pairs (k, E(k))


```
1  public class HashMultimap<K,V> {
2    Map<K,List<V>> map = new HashMap<>();  // the primary map
3    int total = 0;                         // total number of entries in the multimap
4    /** Constructs an empty multimap. */
5    public HashMultimap() { }
6    /** Returns the total number of entries in the multimap. */
7    public int size() { return total; }
8    /** Returns whether the multimap is empty. */
9    public boolean isEmpty() { return (total == 0); }
10   /** Returns a (possibly empty) iterator of all values associated with the key. */
11   Iterable<V> get(K key) {
12     List<V> secondary = map.get(key);
13     if (secondary != null)
14       return secondary;
15     return new Array(List<>();             // return an empty list of values
16   }
```

Java Implementation


```
17   /** Adds a new entry associating key with value. */
18   void put(K key, V value) {
19     List<V> secondary = map.get(key);
20     if (secondary == null) {
21       secondary = new ArrayList<>();
22       map.put(key, secondary);          // begin using new list as secondary structure
23     }
24     secondary.add(value);
25     total++;
26   }
27   /** Removes the (key,value) entry, if it exists. */
28   boolean remove(K key, V value) {
29     boolean wasRemoved = false;
30     List<V> secondary = map.get(key);
31     if (secondary != null) {
32       wasRemoved = secondary.remove(value);
33       if (wasRemoved) {
34         total--;
35         if (secondary.isEmpty())
36           map.remove(key);              // remove secondary structure from primary map
37       }
38     }
39     return wasRemoved;
40   }
```

Java Implementation, 2


```
41   /** Removes all entries with the given key. */
42   Iterable<V> removeAll(K key) {
43     List<V> secondary = map.get(key);
44     if (secondary != null) {
45       total -= secondary.size();
46       map.remove(key);
47     } else
48       secondary = new ArrayList<>();      // return empty list of removed values
49     return secondary;
50   }
51   /** Returns an iteration of all entries in the multimap. */
52   Iterable<Map.Entry<K,V>> entries() {
53     List<Map.Entry<K,V>> result = new ArrayList<>();
54     for (Map.Entry<K,List<V>> secondary : map.entrySet()) {
55       K key = secondary.getKey();
56       for (V value : secondary.getValue())
57         result.add(new AbstractMap.SimpleEntry<K,V>(key,value));
58     }
59     return result;
60   }
61 }
```

Java Implementation, 3

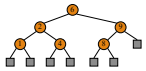
Binary Search Trees

Binary Search Trees

A binary search tree is a proper binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:
- Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) \leq key(v) < key(w)$

External nodes do not store items

An inorder (left, node,right) traversal of a binary search trees visits the keys in increasing order



Search

To search for a key k, we trace a downward path starting at the root

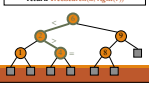
The next node visited depends on the comparison of k with the key of the current node

If we reach a leaf, the key is not found

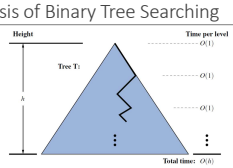
Example: get(4):
- Call TreeSearch(4,root)

The algorithms for nearest neighbor queries are similar

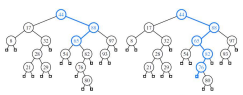
Algorithm *TreeSearch*(k, v)
  if $T.isExternal(v)$
    return v
  if $k < key(v)$
    return *TreeSearch*(k, $left(v)$)
  else if $k = key(v)$
    return v
  else { $k > key(v)$ }
    return *TreeSearch*(k, $right(v)$)



Analysis of Binary Tree Searching



Ex. Search In Binary Search Tree



(a) A successful search for key 65 in a binary search tree. (b) an unsuccessful search for key 68 that terminates at the leaf to the left of the key 76.

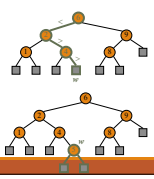
Insertion

To perform operation put(k, o), we search for key k (using TreeSearch)

Assume k is not already in the tree, and let w be the leaf reached by the search

We insert k at node w and expand w into an internal node

Example: insert 5



Algorithm TreeInsert(k, v):
Input: A search key k to be associated with value v
p ← TreeSearch(root(), k)
if k == key(p) **then**
  Change p's value to (v)
else
  expandExternal(p, (k,v)

Deletion

Example: Remove 4

To perform operation remove(**k**), we search for key **k**

Assume key **k** is in the tree, and let **v** be the node storing **k**

If node **v** has a leaf child **w**, we remove **v** and **w** from the tree with operation removeExternal(**w**), which removes **w** and its parent

Deletion (cont.)

Example: Remove 3

We consider the case where the key **k** to be removed is stored at a node **v** whose children are both internal

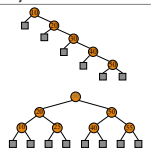
- we find the internal node **w** that follows **v** in an inorder traversal
- we copy **key**(**w**) into node **v**
- we remove node **w** and its left child **z** (which must be a leaf) by means of operation removeExternal(**z**)

Performance of a Binary Search Tree

Consider an ordered map with n items implemented by means of a binary search tree of height h

- the space used is $O(n)$
- methods get, put and remove take $O(h)$ time

The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case



Performance

Method	Running Time
size, isEmpty	O(1)
get, put, remove	O(h)
firstEntry, lastEntry	O(h)
ceilingEntry, floorEntry, lowerEntry, higherEntry	O(h)
subMap	O(s+h)
entrySet, keySet, values	O(n)

Table : Worst-case running times of the operations for a TreeMap. We denote the current height of the tree with h, and the number of entries reported by submap as s. The space usage is O(n), where n is the number of entries stored in the map.

Balanced Search Trees

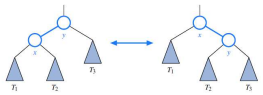
Assume a random series of insertions and removals, the standard binary search tree supports O(logn) expected running times for the basic map operations. However, we may only claim O(n) worst-case time, because some sequences of operations may lead to an unbalanced tree with height proportional to n.

In the remainder of this chapter, we will explore four search-tree algorithms that provide stronger performance guarantees.

Three of the four data structures (**AVL trees**, **splay trees**, and **red-black trees**) are based on augmenting a standard binary search tree with occasional operations to reshape the tree and reduce its height. **The primary operation to rebalance a binary search tree is known as a rotation.**

Rotation Operation in a Binary Search Tree

During a rotation, we "rotate" a child to be above its parent, as diagrammed below



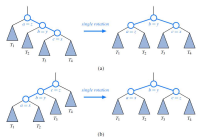
A rotation operation in a binary search tree. A rotation can be performed to transform the left formation into the right, or the right formation into the left. Note that all keys in subtree T1 have keys less than that of position x, all keys in subtree T2 have keys that are between those of positions x and y, and all keys in subtree T3 have keys that are greater than that of position y.

Rotation

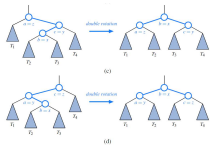
A single rotation modifies a constant number of parent-child relationships, it can be implemented in O(1) time with a linked binary tree representation.

A rotation allows the shape of a tree to be modified while maintaining the search-tree property.

Single Rotation



Double Rotation



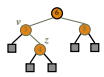
The Trinode Restructuring Operation in a Binary Search Tree

Algorithm restructure(x):
Input: A position x of a binary search tree T that has both a parent y and a grandparent z
Output: Tree T after a trinode restructuring (which corresponds to a single or double rotation) involving positions x, y, and z
1: Let (a, b, c) be a left-to-right (inorder) listing of the positions x, y, and z, and let (T1, T2, T3, T4) be a left-to-right (inorder) listing of the four subtrees of x, y, and z not rooted at x, y, or z.
2: Replace the subtree rooted at z with a new subtree rooted at b.
3: Let a be the left child of b and let T1 and T2 be the left and right subtrees of a, respectively.
4: Let c be the right child of b and let T3 and T4 be the left and right subtrees of c, respectively.

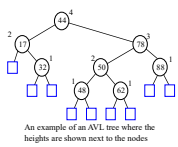
Left Rotation Pseudocode

```
LEFT-ROTATE(T, x)
 1  y = x.right                  // set y
 2  x.right = y.left             // turn y's left subtree into x's right subtree
 3  if y.left ≠ T.nil
 4      y.left.p = x
 5  y.p = x.p                    // link x's parent to y
 6  if x.p == T.nil
 7      T.root = y
 8  elseif x == x.p.left
 9      x.p.left = y
10  else x.p.right = y
11  y.left = x                   // put x on y's left
12  x.p = y
```


AVL Trees



AVL Tree Definition



An example of an AVL tree where the heights are shown next to the nodes

AVL trees are balanced

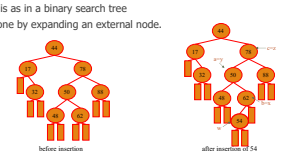
An AVL Tree is a binary search tree such that for every internal node v of T, the **heights of the children of v can differ by at most 1**

Insertion

Insertion is as in a binary search tree

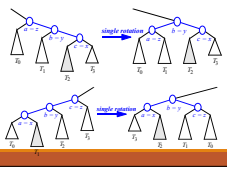
Always done by expanding an external node.

Example:



before insertion after insertion of 54

Restructuring (as Single Rotations)



Height of an AVL Tree

Fact: The height of an AVL tree storing n keys is O(log n).

Proof (by induction): Let us bound n(h): the minimum number of internal nodes of an AVL tree of height h.

We easily see that n(1) = 1 and n(2) = 2

For n > 2, an AVL tree of height h contains the root node, one AVL subtree of height n-1 and another of height n-2.

That is, n(h) = 1 + n(h-1) + n(h-2)

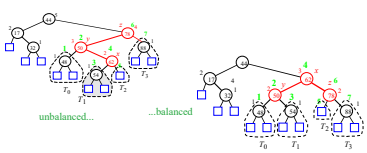
Knowing n(h-1) > n(h-2), we get n(h) > 2n(h-2). So
n(h) > 2n(h-2), n(h) > 4n(h-4), n(h) > 8n(h-6), ... (by induction),
n(h) > 2^i n(h-2i)

Solving the base case we get: n(h) > $2^{h/2-1}$

Taking logarithms: h < 2log n(h) +2

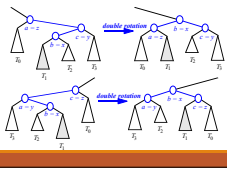
Thus the height of an AVL tree is O(log n)

Insertion (54) Example, continued



unbalanced... ...balanced

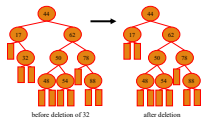
Restructuring (as Double Rotations)



Removal

Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, w, may cause an imbalance.

Example:



AVL Tree Performance

AVL tree storing n items
- The data structure uses O(n) space
- A single restructuring takes O(1) time
  - using a linked-structure binary tree
- Searching takes O(log n) time
  - height of tree is O(log n), no restructures needed
- Insertion takes O(log n) time
  - initial find is O(log n)
  - restructuring up the tree, maintaining heights is O(log n)
- Removal takes O(log n) time
  - initial find is O(log n)
  - restructuring up the tree, maintaining heights is O(log n)

Java Implementation, 2

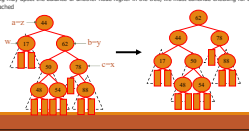
```
/** Returns a child of p with height no smaller than that of the other child. */
protected Position<Entry<K,V>> tallerChild(Position<Entry<K,V>> p) {
  if (height(left(p)) > height(right(p))) return left(p);          // clear winner
  if (height(left(p)) < height(right(p))) return right(p);         // clear winner
  // equal height children; break tie while matching parent's orientation
  if (isRoot(p)) return left(p);                                   // choice is irrelevant
  if (p == left(parent(p))) return left(p);                        // return aligned child
  else return right(p);
}
```


Rebalancing after a Removal

Let z be the first unbalanced node encountered while travelling up the tree from w. Also, let y be the child of z with the larger height, and let x be the child of y with the larger height.

We perform a trinode restructuring to restore balance at z

As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of T is reached



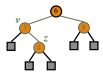

```
/** An implementation of a sorted map using an AVL tree. */
public class AVLTreeMap<K,V> extends TreeMap<K,V> {
  /** Constructs an empty map using the natural ordering of keys. */
  public AVLTreeMap() { super(); }
  /** Constructs an empty map using the given comparator to order keys. */
  public AVLTreeMap(Comparator<K> comp) { super(comp); }
  /** Returns the height of the given tree position. */
  protected int height(Position<Entry<K,V>> p) {
    return tree.getAux(p);
  }
  /** Recomputes the height of the given position based on its children's heights. */
  protected void recomputeHeight(Position<Entry<K,V>> p) {
    tree.setAux(p, 1 + Math.max(height(left(p)), height(right(p))));
  }
  /** Returns whether a position has balance factor between -1 and 1 inclusive. */
  protected boolean isBalanced(Position<Entry<K,V>> p) {
    return Math.abs(height(left(p)) - height(right(p))) <= 1;
  }
```

Java Implementation


```
protected void rebalance(Position<Entry<K,V>> p) {
  int oldHeight, newHeight;
  do {
    oldHeight = height(p);                       // not yet recalculated if internal
    if (!isBalanced(p)) {                        // rebalance detected!
      // perform trinode restructuring, setting p to resulting root,
      // and recompute new local heights after the restructuring
      p = restructure(tallerChild(tallerChild(p)));
      recomputeHeight(left(p));
      recomputeHeight(right(p));
    }
    recomputeHeight(p);
    newHeight = height(p);
    p = parent(p);
  } while (oldHeight != newHeight && p != null);
}
/** Overrides the TreeMap rebalancing hook that is called after an insertion. */
protected void rebalanceInsert(Position<Entry<K,V>> p) {
  rebalance(p);
}
/** Overrides the TreeMap rebalancing hook that is called after a deletion. */
protected void rebalanceDelete(Position<Entry<K,V>> p) {
  if (!isRoot(p))
    rebalance(parent(p));
}
}
```


Java Implementation, 3

Splay Trees



Splaying

Given a node x of a binary search tree T, we splay x by moving x to the root of T through a sequence of restructurings.

The particular restructurings we perform are important, for it is not sufficient to move x to the root of T by just any sequence of restructurings. The specific operation we perform to move x up depends upon the relative positions of x, its parent y, and x's grandparent z (if it exists). There are three cases that we will consider.

Splaying Cases – Zig Zag

One of x and y is a left child and the other is a right child. In this case, we promote x by making x have y and z as its children, while maintaining the inorder relationships of the nodes in T.

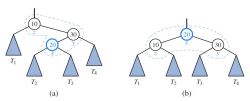


Figure 11.16: Zig-zag: (a) before; (b) after. There is another symmetric configuration where x is a right child and y is a left child.

Splay Trees

Splay tree is conceptually quite different from the other balanced search trees we will discuss in this chapter, for a splay tree does not strictly enforce a logarithmic upper bound on the height of the tree. In fact, no additional height, balance, or other auxiliary data need be stored with the nodes of this tree.

The efficiency of splay trees is due to a certain move-to-root operation, called splaying, that is performed at the bottommost position p reached during every insertion, deletion, or even a search.

Intuitively, a splay operation causes more frequently accessed elements to remain nearer to the root, thereby reducing the typical search times.

The surprising thing about splaying is that it allows us to guarantee a logarithmic amortized running time, for insertions, deletions, and searches.

Splaying Cases – Zig Zig

The node x and its parent y are both left children or both right children. We promote x, making y a child of x and z a child of y, while maintaining the inorder relationships of the nodes in T.

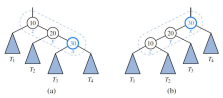


Figure 11.15: Zig-zig: (a) before; (b) after. There is another symmetric configuration where x and y are left children.

Splaying Cases – Zig

x does not have a grandparent. In this case, we perform a single rotation to promote x over y, making y a child of x, while maintaining the relative inorder relationships of the nodes in T.

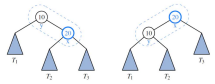
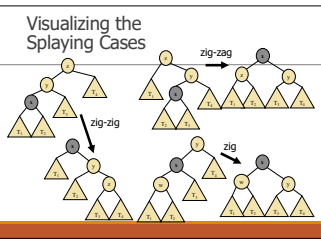


Figure 11.17: Zig: (a) before; (b) after. There is another symmetric configuration where x is originally a left child of y.

Visualizing the Splaying Cases



When to Splay

The rules that dictate when splaying is performed are as follows:

- When searching for key k, if k is found at position p, we splay p, else we splay the parent of the leaf position at which the search terminates unsuccessfully. For example, the splaying in the previous slide would be performed after searching successfully for key 14 or unsuccessfully for key 15.
- When inserting key k, we splay the newly created internal node where k gets inserted. For example, the splaying in the previous figure would be performed if 14 were the newly inserted key.

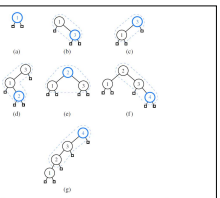


Figure 11.20: A sequence of insertions in a splay tree: (a) initial tree; (b) after inserting 3, but before a zig step; (c) after splaying; (d) after inserting 2, but before a zig-zag step; (e) after splaying; (f) after inserting 4, but before a zig-zig step; (g) after splaying.

Amortized Analysis of Splay Trees

Running time of each operation is proportional to time for splaying.

Define rank(v) as the logarithm (base 2) of the number of nodes in subtree rooted at v.

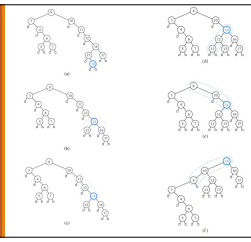
Costs: zig = $1, zig-zig = $2, zig-zag = $2.

Thus, cost for playing a node at depth d = $d.

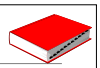
Imagine that we store rank(v) cyber-dollars at each node v of the splay tree (just for the sake of analysis).

We perform a zig-zig or a zig-zag when x has a grandparent, and we perform a zig when x has a parent but not a grandparent. A splaying step consists of repeating these restructurings at x until x becomes the root of T.

Example of splaying a node: (a) splaying the node storing 14 starts with a zig-zag; (b) after the zig-zag; (c) the next step will be a zig-zig; (d) after the zig-zig; (e) the next step is again a zig-zig; (f) after the zig-zig.



Splay Trees

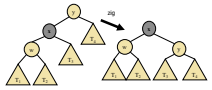


which nodes are splayed after each operation?

method	splay node
Search for k	if key found, use that node if key not found, use parent of ending external node
Insert (k,v)	use the new node containing the entry inserted
Remove item with key k	use the parent of the internal node that was actually removed from the tree (the parent of the node that the removed item was swapped with)

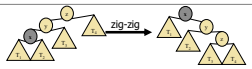
Cost per zig

Rank of key k the number of keys that are less than k. In other words, it returns the index of k in a sorted array, or an ordered tree.

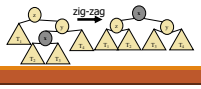


Doing a zig at x costs at most rank' (x) - rank(x):
- cost = rank' (x) + rank' (y) - rank(y) - rank(x) ≤ rank' (x) - rank(x).

Cost per zig-zig and zig-zag



Doing a zig-zig or zig-zag at x costs at most 3(rank' (x) - rank(x)) - 2



Performance of Splay Trees

Recall: rank of a node is logarithm of its size.

Thus, amortized cost of any splay operation is O(log n)

In fact, the analysis goes through for any reasonable definition of rank(x)

This implies that splay trees can actually adapt to perform searches on frequently-requested items much faster than O(log n) in some cases


```
23    // override the various TreeMap rebalancing hooks to perform the appropriate splay
24    protected void rebalanceAccess(Position<Entry<K,V>> p) {
25      if (isExternal(p)) p = parent(p);
26      if (p != null) splay(p);
27    }
28    protected void rebalanceInsert(Position<Entry<K,V>> p) {
29      splay(p);
30    }
31    protected void rebalanceDelete(Position<Entry<K,V>> p) {
32      if (!isRoot(p)) splay(parent(p));
33    }
34  }
```

Java Implementation

Cost of Splaying

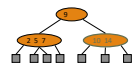
Cost of splaying a node x at depth d of a tree rooted at r:

- at most 3(rank(r) - rank(x)) - d + 2:
- Proof: Splaying x takes d/2 splaying substeps:

$$\begin{aligned}\text{cost} &\le \sum_{i=1}^{d/2} \text{cost}_i \\ &\le \sum_{i=1}^{d/2} (3(\text{rank}_i(x) - \text{rank}_{i-1}(x)) - 2) + 2 \\ &= 3(\text{rank}(r) - \text{rank}_0(x)) - 2(d/d) + 2 \\ &\le 3(\text{rank}(r) - \text{rank}(x)) - d + 2.\end{aligned}$$


```
1   /** An implementation of a sorted map using a splay tree. */
2   public class SplayTreeMap<K,V> extends TreeMap<K,V> {
3     /** Constructs an empty map using the natural ordering of keys. */
4     public SplayTreeMap() { super(); }
5     /** Constructs an empty map using the given comparator to order keys. */
6     public SplayTreeMap(Comparator<K> comp) { super(comp); }
7     /** Utility used to rebalance after a map operation. */
8     private void splay(Position<Entry<K,V>> p) {
9       while (!isRoot(p)) {
10        Position<Entry<K,V>> parent = parent(p);
11        Position<Entry<K,V>> grand = parent(parent);
12        if (grand == null)                                    // zig case
13          rotate(p);
14        else if ((parent == left(grand)) == (p == left(parent))) {   // zig-zig case
15          rotate(parent);          // move PARENT upward
16          rotate(p);               // then move p upward
17        } else {                                              // zig-zag case
18          rotate(p);               // move p upward
19          rotate(p);               // move p upward again
20        }
21      }
22    }
```

Java Implementation

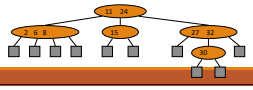


(2,4) Trees

Multi-Way Search Tree

A multi-way search tree is an ordered tree such that

- Each internal node has at least two children and stores $d-1$ key-element items (k_i, o_i), where d is the number of children
- For a node with children $v_1\ v_2 \ldots v_d$ storing keys $k_1 k_2 \ldots k_{d-1}$
  - keys in the subtree of v_1 are less than k_1
  - keys in the subtree of v_i are between k_{i-1} and k_i $(i = 2, \ldots, d-1)$
  - keys in the subtree of v_d are greater than k_{d-1}
- The leaves store no items and serve as placeholders



Multi-Way Searching

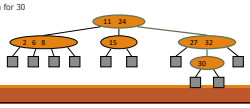
Similar to search in a binary search tree

A each internal node with children $v_1, v_2, \ldots v_d$ and keys $k_1 k_2 \ldots k_{d-1}$

- $k = k_i$ $(i = 1, \ldots, d-1)$: the search terminates successfully
- $k < k_1$: we continue the search in child v_1
- $k_{i-1} < k < k_i$ $(i = 2, \ldots, d-1)$: we continue the search in child v_i
- $k > k_{d-1}$: we continue the search in child v_d

Reaching an external node terminates the search unsuccessfully

Example: search for 30



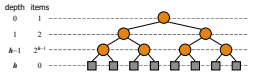
Height of a (2,4) Tree

Theorem: A (2,4) tree storing n items has height $O(\log n)$

Proof:

- Let h be the height of a (2,4) tree with n items
  - Since there are at least 2^i items at depth $i = 0, \ldots, h-1$ and no items at depth h, we have $n \ge 1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$
- Thus, $h \le \log(n+1)$

Searching in a (2,4) tree with n items takes $O(\log n)$ time

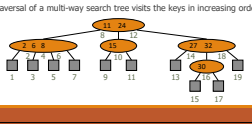


Multi-Way Inorder Traversal

We can extend the notion of inorder traversal from binary trees to multi-way search trees

Namely, we visit item (k_i, o_i) of node v between the recursive traversals of the subtrees of v rooted at children v_i and v_{i+1}

An inorder traversal of a multi-way search tree visits the keys in increasing order

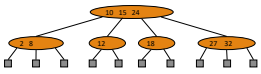


(2,4) Trees

A (2,4) tree (also called 2-4 tree or 2-3-4 tree) is a multi-way search with the following properties

- Node-Size Property: every internal node has at most four children
- Depth Property: all the external nodes have the same depth

Depending on the number of children, an internal node of a (2,4) tree is called a 2-node, 3-node or 4-node

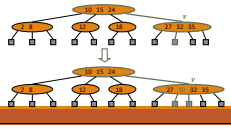


Insertion

We insert a new item (k, o) at the parent v of the leaf reached by searching for k

- We preserve the depth property but
- We may cause an overflow (i.e., node v may become a 5-node)

Example: inserting key 30 causes an overflow

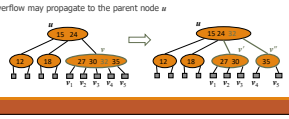


Overflow and Split

We handle an overflow at a 5-node v with a split operation:

- let $v_1 \ldots v_5$ be the children of v and $k_1 \ldots k_4$ be the keys of v
- node v is replaced nodes v' and v''
  - v' is a 3-node with keys k_1 k_2 and children v_1 v_2 v_3
  - v'' is a 2-node with key k_4 and children v_4 v_5
- key k_3 is inserted into the parent u of v (a new root may be created)

The overflow may propagate to the parent node u

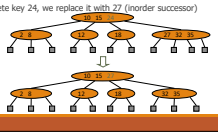


Deletion

We reduce deletion of an entry to the case where the item is at the node with leaf children

Otherwise, we replace the entry with its inorder successor (or, equivalently, with its inorder predecessor) and delete the latter entry

Example: to delete key 24, we replace it with 27 (inorder successor)

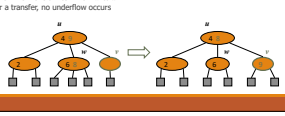


Underflow and Transfer

To handle an underflow at node v with parent u, we consider two cases

Case 2: an adjacent sibling w of v is a 3-node or a 4-node

- Transfer operation:
  1. we move a child of w to v
  2. we move an item from u to v
  3. we move an item from w to u
- After a transfer, no underflow occurs



Analysis of Insertion

Algorithm $put(k, o)$

1. We search for key k to locate the insertion node v
2. We add the new entry (k, o) at node v
3. **while** $overflow(v)$
   if $isRoot(v)$
   create a new empty root above v
   $v \leftarrow split(v)$

Let T be a (2,4) tree with n items

- Tree T has $O(\log n)$ height
- Step 1 takes $O(\log n)$ time because we visit $O(\log n)$ nodes
- Step 2 takes $O(1)$ time
- Step 3 takes $O(\log n)$ time because each split takes $O(1)$ time and we perform $O(\log n)$ splits

Thus, an insertion in a (2,4) tree takes $O(\log n)$ time

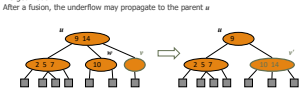
Underflow and Fusion

Deleting an entry from a node v may cause an underflow, where node v becomes a 1-node with one child and no keys

To handle an underflow at node v with parent u, we consider two cases

Case 1: the adjacent siblings of v are 2-nodes

- Fusion operation: we merge v with an adjacent sibling w and move an entry from u to the merged node v'
- After a fusion, the underflow may propagate to the parent u



Analysis of Deletion

Let T be a (2,4) tree with n items

- Tree T has $O(\log n)$ height

In a deletion operation

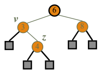
- We visit $O(\log n)$ nodes to locate the node from which to delete the entry
- We handle an underflow with a series of $O(\log n)$ fusions, followed by at most one transfer
- Each fusion and transfer takes $O(1)$ time

Thus, deleting an item from a (2,4) tree takes $O(\log n)$ time

Comparison of Map Implementations

	Search	Insert	Delete	Notes
Hash Table	1 expected	1 expected	1 expected	◦ no ordered map methods ◦ simple to implement
Skip List	$\log n$ high prob.	$\log n$ high prob.	$\log n$ high prob.	◦ randomized insertion ◦ simple to implement
AVL and (2,4) Tree	$\log n$ worst-case	$\log n$ worst-case	$\log n$ worst-case	◦ complex to implement

63



Red-Black Trees

64

Red-Black Trees

Although AVL trees and (2,4) trees have a number of nice properties, they also have some disadvantages. For instance, AVL trees may require many restructure operations (rotations) to be performed after a deletion, and (2,4) trees may require many **split** or **fusing** operations to be performed after an insertion or removal.

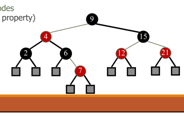
The red-black tree, does not have these drawbacks; it uses $O(1)$ structural changes after an update in order to stay balanced.

65

Red-Black Trees

A red-black tree can also be defined as a binary search tree that satisfies the following properties:

- Every node is red or black
- **Root** is black (if it is not we make it black)
- New insertions are always red
- Every path from root to leaf has the same # of **BLACK** nodes (all the leaves (nil) have the same black **depth**)
- No path can have two consecutive RED nodes
- Every leaf is black (nil is black) (**external** property)



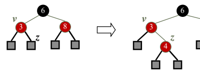
66

Insertion

To insert (k, o), we execute the insertion algorithm for binary search trees and color red the newly inserted node z unless it is the root

- We preserve the **root**, **external**, and **depth** properties
- If the **parent v of z is black**, we also preserve the internal property and we are done
- Else (v **is red**) we have a double red (i.e., a violation of the internal property), which requires a reorganization of the tree

Example where the insertion of 4 causes a double red:



67

Remedying a Double Red

Consider a double red with child z and parent v, and let w be the sibling of v

Case 1: w is black (Aunt is Black)

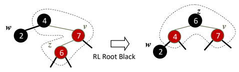
Case 2: w is red (Aunt is Red)

68

Restructuring

A restructuring remedies a child-parent double red when the parent red node has a black sibling. (Aunt is Black)

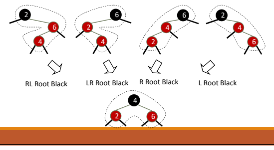
The internal property is restored and the other properties are preserved



69

Restructuring (cont.)

There are four restructuring configurations depending on whether the double red nodes are left or right children

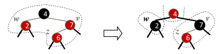


70

Recoloring (Aunt is Red)

A recoloring remedies a child-parent double red when the parent red node has a red sibling

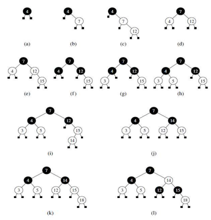
The parent v and its sibling w become black and the grandparent u becomes red, unless it is the root



71

Example

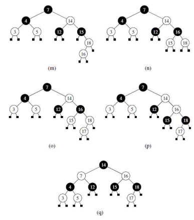
A sequence of insertions in a red-black tree: (a) initial tree; (b) insertion of 7; (c) insertion of 12, which causes a double red; (d) after restructuring; (e) insertion of 15, which causes a double red; (f) after recoloring (the root remains black); (g) insertion of 3; (h) insertion of 5; (i) insertion of 14, which causes a double red; (j) after restructuring; (k) insertion of 18, which causes a double red; (l) after recoloring.



72

Example

A sequence of insertions in a red-black tree (continued from previous figure): (m) insertion of 16, which causes a double red; (n) after restructuring; (o) insertion of 17, which causes a double red; (p) after recoloring there is again a double red, to be handled by a restructuring; (q) after restructuring.



73

Height of a Red-Black Tree

Theorem: A red-black tree storing n items has height $O(\log n)$

Proof:

- The height of a red-black tree is at most twice the height of its associated (2,4) tree, which is $O(\log n)$

The search algorithm for a binary search tree is the same as that for a binary search tree

By the above theorem, searching in a red-black tree takes $O(\log n)$ time

74

Analysis of Insertion

Algorithm *insert*(k, o)

1. We search for key k to locate the insertion node z
2. We add the new entry (k, o) at node z and color z red
3. **while** *doubleRed*(z)
   if *isBlack*(*sibling*(*parent*(z)))
   $z \leftarrow$ ***restructure***(z)
   return
   else { *sibling*(*parent*(z)) is red }
   $z \leftarrow$ ***recolor***(z)

Recall that a red-black tree has $O(\log n)$ height

Step 1 takes $O(\log n)$ time because we visit $O(\log n)$ nodes

Step 2 takes $O(1)$ time

Step 3 takes $O(\log n)$ time because we perform

- $O(\log n)$ recolorings, each taking $O(1)$ time, and
- at most one restructuring taking $O(1)$ time

Thus, an insertion in a red-black tree takes $O(\log n)$ time

75

Deletion

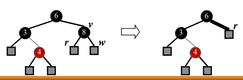
Reminder: Every path from root to leaf has the same # of **BLACK** nodes (all the leaves (nil) have the same black **depth**)

To perform operation remove(k), **we first execute the deletion algorithm for binary search trees**

Let v be the internal node removed, w the external node removed, and r the sibling of w

- If either v **or** r was RED, we color r **BLACK** and we are done
- Else (v **and** r were both black) we color r ***double black***, which is a violation of the internal property requiring a reorganization of the tree

Example where the deletion of 8 causes a double black:



76

Remedying a Double Black

The algorithm for remedying a double black node w with sibling y considers three cases

Case 1: y **is black and has a red child**

- We perform a restructuring, equivalent to a transfer , and we are done

Case 2: y **is black and its children are both black**

- We perform a recoloring, equivalent to a fusion, which may propagate up the double black violation

Case 3: y **is red**

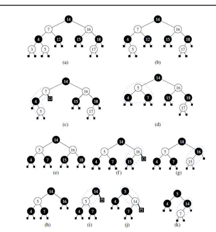
- We perform an adjustment, equivalent to choosing a different representation of a 3-node, after which either Case 1 or Case 2 applies

Deletion in a red-black tree takes $O(\log n)$ time

77

Example

A sequence of deletions from a red-black tree: (a) initial tree; (b) removal of 3; (c) removal of 12, causing a black deficit to the right of 7 (handled by restructuring); (d) after restructuring; (e) removal of 17; (f) removal of 18, causing a black deficit to the right of 16 (handled by recoloring); (g) after recoloring; (h) removal of 15; (i) removal of 16, causing a black deficit to the right of 14 (handled initially by a rotation); (j) after the rotation the black deficit needs to be handled by a recoloring; (k) after the recoloring.



78

Red-Black Tree Reorganization

Insertion	remedy double red	
Red-black tree action	**(2,4) tree action**	**result**
restructuring	change of 4-node representation	double red removed
recoloring	split	double red removed or propagated up

Deletion	remedy double black	
Red-black tree action	**(2,4) tree action**	**result**
restructuring	transfer	double black removed
recoloring	fusion	double black removed or propagated up
adjustment	change of 3-node representation	restructuring or recoloring follows

79

Java Implementation

```
/** An implementation of a sorted map using a red-black tree. */
public class RBTreeMap<K,V> extends TreeMap<K,V> {
  /** Constructs an empty map using the natural ordering of keys. */
  public RBTreeMap() { super(); }
  /** Constructs an empty map using the given comparator to order keys. */
  public RBTreeMap(Comparator<K> comp) { super(comp); }
  // we use the inherited aux field with convention that 0=black and 1=red
  // (note that new leaves will be black by default, as aux=0)
  private boolean isBlack(Position<Entry<K,V>> p) { return tree.getAux(p)==0;}
  private boolean isRed(Position<Entry<K,V>> p) { return tree.getAux(p)==1; }
  private void makeBlack(Position<Entry<K,V>> p) { tree.setAux(p, 0); }
  private void makeRed(Position<Entry<K,V>> p) { tree.setAux(p, 1); }
  private void setColor(Position<Entry<K,V>> p, boolean toRed) {
    tree.setAux(p, toRed ? 1 : 0);
  }
  /** Overrides the TreeMap rebalancing hook that is called after an insertion. */
  protected void rebalanceInsert(Position<Entry<K,V>> p) {
    if (!isRoot(p)) {
      makeRed(p);                  // the new internal node is initially colored red
      resolveRed(p);               // but this may cause a double-red problem
    }
  }
```

80

```
/** Remedies potential double-red violation above red position p. */
private void resolveRed(Position<Entry<K,V>> p) {
  Position<Entry<K,V>> parent,uncle,middle,grand; // used in case analysis
  parent = parent(p);
  if (isRed(parent)) {                              // double-red problem exists
    uncle = sibling(parent);
    if (isBlack(uncle)) {                           // Case 1: misshapen 4-node
      middle = restructure(p);                      // do trinode restructuring
      makeBlack(middle);
      makeRed(left(middle));
      makeRed(right(middle));
    } else {                                        // Case 2: overfull 5-node
      makeBlack(parent);                            // perform recoloring
      makeBlack(uncle);
      grand = parent(parent);
      if (!isRoot(grand)) {
        makeRed(grand);                             // grandparent becomes red
        resolveRed(grand);                          // recur at red grandparent
      }
    }
  }
}
```

Java Implementation 2

81

```
/** Overrides the TreeMap rebalancing hook that is called after a deletion. */
protected void rebalanceDelete(Position<Entry<K,V>> p) {
  if (isRed(p))                                     // deleted parent was black
    makeBlack(p);                                   // so this restores black depth
  else if (!isRoot(p)) {
    Position<Entry<K,V>> sib = sibling(p);
    if (isInternal(sib) && (isBlack(sib) || isInternal(left(sib))))
      remedyDoubleBlack(p);                  // sib's subtree has nonzero black height
  }
}

/** Remedies a presumed double-black violation at the given (nonroot) position. */
private void remedyDoubleBlack(Position<Entry<K,V>> p) {
  Position<Entry<K,V>> z = parent(p);
  Position<Entry<K,V>> y = sibling(p);
  if (isBlack(y)) {
    if (isRed(left(y)) || isRed(right(y))) {        // Case 1: trinode restructuring
      Position<Entry<K,V>> x = (isRed(left(y)) ? left(y) : right(y));
      Position<Entry<K,V>> middle = restructure(x);
      setColor(middle, isRed(z)); // root of restructured subtree gets z's old color
      makeBlack(left(middle));
      makeBlack(right(middle));
    } else {                                        // Case 2: recoloring
      makeRed(y);
      if (isRed(z))
        makeBlack(z);                               // problem is resolved
      else if (!isRoot(z))
        remedyDoubleBlack(z);                       // propagate the problem
    }
```

Java Implementation 3

82


```
      makeBlack(p);                                 // so this restores black depth
  else if (!isRoot(p)) {
    Position<Entry<K,V>> sib = sibling(p);
    if (isInternal(sib) && (isBlack(sib) || isInternal(left(sib))))
      remedyDoubleBlack(p);                  // sib's subtree has nonzero black height
  }
}

/** Remedies a presumed double-black violation at the given (nonroot) position. */
private void remedyDoubleBlack(Position<Entry<K,V>> p) {
  Position<Entry<K,V>> z = parent(p);
  Position<Entry<K,V>> y = sibling(p);
  if (isBlack(y)) {
    if (isRed(left(y)) || isRed(right(y))) {        // Case 1: trinode restructuring
      Position<Entry<K,V>> x = (isRed(left(y)) ? left(y) : right(y));
      Position<Entry<K,V>> middle = restructure(x);
      setColor(middle, isRed(z)); // root of restructured subtree gets z's old color
      makeBlack(left(middle));
      makeBlack(right(middle));
    } else {                                        // Case 2: recoloring
      makeRed(y);
      if (isRed(z))
        makeBlack(z);                               // problem is resolved
      else if (!isRoot(z))
        remedyDoubleBlack(z);                       // propagate the problem
    }
  } else {                                          // Case 3: reorient 3-node
    rotate(y);
    makeBlack(y);
    makeRed(z);
```

Java Implementation 4

83

Practice Questions

R-11.15 Perform the following sequence of operations in an initially empty splay tree and draw the tree after each set of operations.

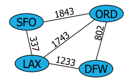
a. Insert keys 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, in this order.
b. Search for keys 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, in this order.
c. Delete keys 0, 2, 4, 6, 8, 10, 12, 14, 16, 18, in this order.

R-11.24 Consider a tree T storing 100,000 entries. What is the worst-case height of T in the following cases?

a. T is a binary search tree.
b. T is an AVL tree.
c. T is a splay tree.
d. T is a (2,4) tree.
e. T is a red-black tree.

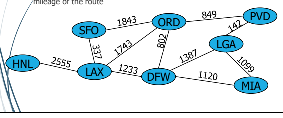
84

Graphs



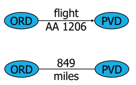
Graphs

- A graph is a pair (V, E), where
  - V is a set of nodes, called vertices
  - E is a collection of pairs of vertices, called edges
  - Vertices and edges are positions and store elements
- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stores the mileage of the route



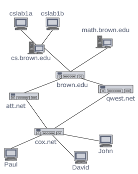
Edge Types

- Directed edge
  - ordered pair of vertices (u,v)
  - first vertex u is the origin
  - second vertex v is the destination
  - e.g., a flight
- Undirected edge
  - unordered pair of vertices (u,v)
  - e.g., a flight route
- Directed graph
  - all the edges are directed
  - e.g., route network
- Undirected graph
  - all the edges are undirected
  - e.g., flight network



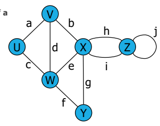
Applications

- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
- Databases
  - Entity-relationship diagram



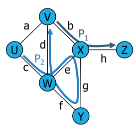
Terminology

- End vertices (or endpoints) of an edge
  - U and V are the **endpoints** of a
- Edges incident on a vertex
  - a, d, and b are **incident** on V
- Adjacent vertices
  - U and V are **adjacent**
- Degree of a vertex
  - X has degree 5
- Parallel edges
  - h and i are parallel edges
- Self-loop
  - j is a self-loop



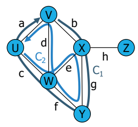
Terminology (cont.)

- Path
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- Simple path
  - path such that all its vertices and edges are distinct
- Examples
  - P_1=(V,b,X,h,Z) is a simple path
  - P_2=(U,c,W,e,X,g,Y,f,W,d,V) is a path that is not simple



Terminology (cont.)

- Cycle
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- Simple cycle
  - cycle such that all its vertices and edges are distinct
- Examples
  - C_1=(V,b,X,g,Y,f,W,c,U,a,↵) is a simple cycle
  - C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,↵) is a cycle that is not simple

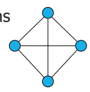


Properties of Graphs

Notation
- n number of vertices
- m number of edges
- $\deg(v)$ degree of vertex v

Property 1
- $\Sigma_v \deg(v) = 2m$
- Proof: each edge is counted twice

Property 2
- In an undirected graph with no self-loops and no multiple edges
- $m \leq n(n-1)/2$
- Proof: each vertex has degree at most $(n-1)$



Example
- $n = 4$
- $m = 6$
- $\deg(v) = 3$

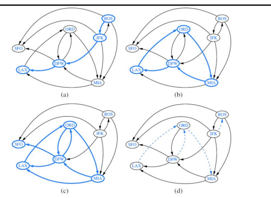


Figure 14.3: Examples of reachability in a directed graph: (a) a directed path from BOS to LAX is highlighted; (b) a directed cycle (ORD, MIA, DFW, LAX, ORD) is highlighted; its vertices induce a strongly connected subgraph; (c) the subgraph of the vertices and edges reachable from ORD is highlighted; (d) the removal of the dashed edges results in a directed acyclic graph.

Vertices and Edges

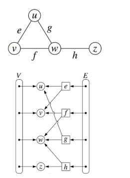
- A **graph** is a collection of **vertices** and **edges**.
- We model the abstraction as a combination of three data types: Vertex, Edge, and Graph.
- A **Vertex** is a lightweight object that stores an arbitrary element provided by the user (e.g., an airport code)
  - We assume it supports a method, element(), to retrieve the stored element.
- An **Edge** stores an associated object (e.g., a flight number, travel distance, cost), retrieved with the element( ) method.

Graph ADT

- numVertices( ): Returns the number of vertices of the graph.
- vertices( ): Returns an iteration of all the vertices of the graph.
- numEdges( ): Returns the number of edges of the graph.
- edges( ): Returns an iteration of all the edges of the graph.
- getEdge(u, v): Returns the edge from vertex u to vertex v, if one exists; otherwise return null. For an undirected graph, there is no difference between getEdge(u, v) and getEdge(v, u).
- endVertices(e): Returns an array containing the two endpoint vertices of edge e. If the graph is directed, the first vertex is the origin and the second is the destination.
- opposite(v, e): For edge e incident to vertex v, returns the other vertex of the edge; an error occurs if e is not incident to v.
- outDegree(v): Returns the number of outgoing edges from vertex v.
- inDegree(v): Returns the number of incoming edges to vertex v. For an undirected graph, this returns the same value as does outDegree(v).
- outgoingEdges(v): Returns an iteration of all outgoing edges from vertex v.
- incomingEdges(v): Returns an iteration of all incoming edges to vertex v. For an undirected graph, this returns the same collection as does outgoingEdges(v).
- insertVertex(x): Creates and returns a new Vertex storing element x.
- insertEdge(u, v, x): Creates and returns a new Edge from vertex u to vertex v, storing element x; an error occurs if there already exists an edge from u to v.
- removeVertex(v): Removes vertex v and all its incident edges from the graph.
- removeEdge(e): Removes edge e from the graph.

Edge List Structure

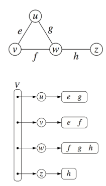
- Vertex object (stored in an unordered list V)
  - element
  - reference to position in vertex sequence
- Edge object (stored in an unordered list E)
  - element
  - origin vertex object
  - destination vertex object
  - reference to position in edge sequence
- Vertex sequence
  - sequence of vertex objects
- Edge sequence
  - sequence of edge objects



Schematic representation of the edge list structure for G. Notice that an edge object refers to the two vertex objects that correspond to its endpoints, but that vertices do not refer to incident edges.

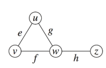
Adjacency List Structure

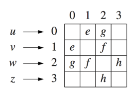
- Incidence sequence for each vertex
  - sequence of references to edge objects of incident edges
- Augmented edge objects
  - references to associated positions in incidence sequences of end vertices



Adjacency Matrix Structure

- Edge list structure
- Augmented vertex objects
  - Integer key (index) associated with vertex
- 2D-array adjacency array
  - Reference to edge object for adjacent vertices
  - Null for non nonadjacent vertices
- The "old fashioned" version just has 0 for no edge and 1 for edge





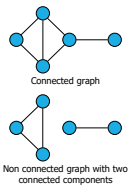
Performance

- n vertices, m edges
- no parallel edges
- no self-loops

	Edge List	Adjacency List	Adjacency Matrix
Space	$n+m$	$n+m$	n^2
incidentEdges(v)	m	deg(v)	n
areAdjacent (v, w)	m	min(deg(v), deg(w))	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	deg(v)	n^2
removeEdge(e)	1	1	1

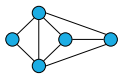
Connectivity

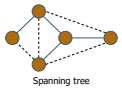
- A graph is connected if there is a path between every pair of vertices
- A connected component of a graph G is a maximal connected subgraph of G



Spanning Trees and Forests

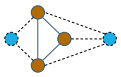
- A spanning tree of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks
- A spanning forest of a graph is a spanning subgraph that is a forest

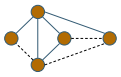




Subgraphs

- A subgraph S of a graph G is a graph such that
  - The vertices of S are a subset of the vertices of G
  - The edges of S are a subset of the edges of G
- A spanning subgraph of G is a subgraph that contains **all the vertices** of G (not necessarily the edges)

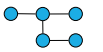


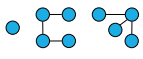


Trees and Forests

- A (free) tree is an undirected graph T such that
  - T is connected
  - T has no cycles

  This definition of tree is different from the one of a rooted tree
- A forest is an undirected graph without cycles
- The connected components of a forest are trees





Graph Traversals

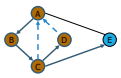
Graph Traversal

- A **traversal** is a systematic procedure for exploring a graph by examining all of its vertices and edges. A traversal is efficient if it visits all the vertices and edges in time proportional to their number, that is, in linear time.
- Graph traversal algorithms are key to answering many fundamental questions about graphs involving the notion of reachability, that is, in determining how to travel from one vertex to another while following paths of a graph.

Interesting Problems of Reachability in an Directed Graph G

- Computing a directed path from vertex u to vertex v, or reporting that no such
- path exists.
- Finding all the vertices of G that are reachable from a given vertex s.
- Determine whether G is acyclic.
- Determine whether G is strongly connected.

Depth-First Search



Interesting Problems of Reachability in an Undirected Graph G

- Computing a path from vertex u to vertex v, or reporting that no such path exists.
- Given a start vertex s of G, computing, for every vertex v of G, a path with the minimum number of edges between s and v, or reporting that no such path exists.
- Testing whether G is connected.
- Computing a spanning tree of G, if G is connected.
- Computing the connected components of G.
- Identifying a cycle in G, or reporting that G has no cycles.

Graph Traversal Algorithms

- There are two efficient graph traversal algorithms, called
  - Depth-first Search
  - Breadth-first Search

Depth-First Search

- Depth-first search (DFS) is a general technique for traversing a graph
- A DFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- DFS on a graph with n vertices and m edges takes $O(n+m)$ time
- DFS can be further extended to solve other graph problems
  - Find and report a path between two given vertices
  - Find a cycle in the graph
- Depth-first search is to graphs what Euler tour is to binary trees

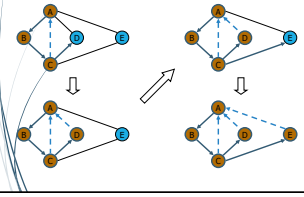
Depth-First Search

- Depth-first search in a graph G is analogous to wandering in a labyrinth with a string and a can of paint without getting lost. We begin at a specific starting vertex s in G, which we initialize by fixing one end of our string to s and painting s as "visited." The vertex s is now our "current" vertex. In general, if we call our current vertex u, we traverse G by considering **an arbitrary edge (u,v) incident to the current vertex u**. If the edge (u,v) leads us to a vertex v that is already visited (that is, painted), we ignore that edge. If, on the other hand, (u,v) leads to an unvisited vertex v, then we unroll our string, and go to v. We then paint v as "visited," and make it the current vertex, repeating the computation above.

Java Implementation

```
1  /** Performs depth-first search of Graph g starting at Vertex u. */
2  public static <V,E> void DFS(Graph<V,E> g, Vertex<V> u,
3                               Set<Vertex<V>> known, Map<Vertex<V>,Edge<E>> forest) {
4    known.add(u);                          // u has been discovered
5    for (Edge<E> e : g.outgoingEdges(u)) {  // for every outgoing edge from u
6      Vertex<V> v = g.opposite(u, e);
7      if (!known.contains(v)) {
8        forest.put(v, e);                  // e is the tree edge that discovered v
9        DFS(g, v, known, forest);          // recursively explore from v
10     }
11   }
12 }
```


Example (cont.)

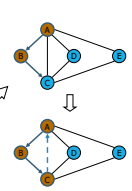


DFS Algorithm from a Vertex

Algorithm DFS(G, u):
Input: A graph G and a vertex u of G
Output: A collection of vertices reachable from u, with their discovery edges
Mark vertex u as visited.
for each of u's outgoing edges, $e = (u, v)$ **do**
  if vertex v has not been visited **then**
    Record edge e as the discovery edge for vertex v.
    Recursively call DFS(G, v).

Example

- unexplored vertex
- visited vertex
- unexplored edge
- discovery edge
- back edge



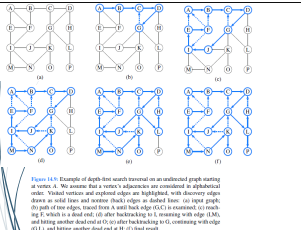
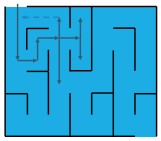


Figure 14.9: Example of depth-first search traversal on an undirected graph starting at vertex A. We assume that a vertex's adjacencies are considered in alphabetical order. Visited vertices and explored edges are highlighted, with discovery edges drawn as solid lines and nontree (back) edges as dashed lines: (a) input graph; (b) path of tree edges, traced from A until back edge (G,C) is examined; (c) reaching F, which is a dead end; (d) after backtracking to I, resuming with edge (I,M), and hitting another dead end at O; (e) after backtracking to G, continuing with edge (G,L), and hitting another dead end at H; (f) final result.

DFS and Maze Traversal

- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



Analysis of DFS

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or BACK
- Method incidentEdges is called once for each vertex
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\sum_v \deg(v) = 2m$

Path Finding in Java

```
1  /** Returns an ordered list of edges comprising the directed path from u to v. */
2  public static <V,E> PositionalList<Edge<E>>
3  constructPath(Graph<V,E> g, Vertex<V> u, Vertex<V> v,
4                Map<Vertex<V>,Edge<E>> forest) {
5    PositionalList<Edge<E>> path = new LinkedPositionalList<>();
6    if (forest.get(v) != null) {                    // v was discovered during the search
7      Vertex<V> walk = v;                           // we construct the path from back to front
8      while (walk != u) {
9        Edge<E> edge = forest.get(walk);
10       path.addFirst(edge);                        // add edge to *front* of path
11       walk = g.opposite(walk, edge);              // repeat with opposite endpoint
12     }
13   }
14   return path;
15 }
```

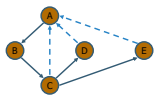

Properties of DFS

Property 1

$DFS(G, v)$ visits all the vertices and edges in the connected component of v

Property 2

The discovery edges labeled by $DFS(G, v)$ form a spanning tree of the connected component of v



Depth-First Search

Path Finding

- We can specialize the DFS algorithm to find a path between two given vertices u and z using the template method pattern
- We call $DFS(G, u)$ with u as the start vertex
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as destination vertex z is encountered, we return the path as the contents of the stack

Algorithm *pathDFS(G, v, z)*
- *setLabel(v, VISITED)*
- *S.push(v)*
- **if** *v = z*
  - **return** *S.elements()*
- **for all** *e ∈ G.incidentEdges(v)*
  - **if** *getLabel(e) = UNEXPLORED*
    - *w ← opposite(v,e)*
    - **if** *getLabel(w) = UNEXPLORED*
      - *setLabel(e, DISCOVERY)*
      - *S.push(e)*
      - *pathDFS(G, w, z)*
      - *S.pop(e)*
    - **else**
      - *setLabel(e, BACK)*
- *S.pop(v)*

Cycle Finding

- We can specialize the DFS algorithm to find a simple cycle using the template method pattern
- We use a stack S to keep track of the path between the start vertex and the current vertex
- As soon as a back edge (v, w) is encountered, we return the cycle as the portion of the stack from the top to vertex w

Algorithm *cycleDFS(G, v, z)*
- *setLabel(v, VISITED)*
- *S.push(v)*
- **for all** *e ∈ G.incidentEdges(v)*
  - **if** *getLabel(e) = UNEXPLORED*
    - *w ← opposite(v,e)*
    - *S.push(e)*
    - **if** *getLabel(w) = UNEXPLORED*
      - *setLabel(e, DISCOVERY)*
      - *pathDFS(G, w, z)*
      - *S.pop(e)*
    - **else**
      - *T ← new empty stack*
      - **repeat**
        - *o ← S.pop()*
        - *T.push(o)*
      - **until** *o = w*
      - **return** *T.elements()*
- *S.pop(v)*

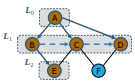
DFS for an Entire Graph

- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

Algorithm *DFS(G)*
- **Input** graph G
- **Output** labeling of the edges of G as discovery edges and back edges
- **for all** *u ∈ G.vertices()*
  - *setLabel(u, UNEXPLORED)*
- **for all** *e ∈ G.edges()*
  - *setLabel(e, UNEXPLORED)*
- **for all** *v ∈ G.vertices()*
  - **if** *getLabel(v) = UNEXPLORED*
    - *DFS(G, v)*

Algorithm *DFS(G, v)*
- **Input** graph G and a start vertex v of G
- **Output** labeling of the edges of G in the connected component of v as discovery edges and back edges
- *setLabel(v, VISITED)*
- **for all** *e ∈ G.incidentEdges(v)*
  - **if** *getLabel(e) = UNEXPLORED*
    - *w ← opposite(v,e)*
    - **if** *getLabel(w) = UNEXPLORED*
      - *setLabel(e, DISCOVERY)*
      - *DFS(G, w)*
    - **else**
      - *setLabel(e, BACK)*

Breadth-First Search



Breadth-First Search

BFS Algorithm

- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

Algorithm *BFS(G)*
- **Input** graph G
- **Output** labeling of the edges and partition of the vertices of G
- **for all** *u ∈ G.vertices()*
  - *setLabel(u, UNEXPLORED)*
- **for all** *e ∈ G.edges()*
  - *setLabel(e, UNEXPLORED)*
- **for all** *v ∈ G.vertices()*
  - **if** *getLabel(v) = UNEXPLORED*
    - *BFS(G, v)*

Algorithm *BFS(G, s)*
- L_0 ← new empty sequence
- *L_0.addLast(s)*
- *setLabel(s, VISITED)*
- $i \leftarrow 0$
- **while** *¬L_i.isEmpty()*
  - L_{i+1} ← new empty sequence
  - **for all** *v ∈ L_i.elements()*
    - **for all** *e ∈ G.incidentEdges(v)*
      - **if** *getLabel(e) = UNEXPLORED*
        - *w ← opposite(v,e)*
        - **if** *getLabel(w) = UNEXPLORED*
          - *setLabel(e, DISCOVERY)*
          - *setLabel(w, VISITED)*
          - *L_{i+1}.addLast(w)*
        - **else**
          - *setLabel(e, CROSS)*
  - $i \leftarrow i + 1$

Breadth-First Search

All Connected Components

- Loop over all vertices, doing a DFS from each unvisted one.

```
1  /** Performs DFS for the entire graph and returns the DFS forest as a map. */
2  public static <V,E> Map<Vertex<V>,Edge<E>> DFSComplete(Graph<V,E> g) {
3    Set<Vertex<V>> known = new HashSet<>();
4    Map<Vertex<V>,Edge<E>> forest = new ProbeHashMap<>();
5    for (Vertex<V> u : g.vertices())
6      if (!known.contains(u))
7        DFS(g, u, known, forest);            // (re)start the DFS process at u
8    return forest;
9  }
```


Breadth-First Search

- Breadth-first search (BFS) is a general technique for traversing a graph
- A BFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- BFS on a graph with n vertices and m edges takes $O(n + m)$ time
- BFS can be further extended to solve other graph problems
  - Find and report a path with the minimum number of edges between two given vertices
  - Find a simple cycle, if there is one

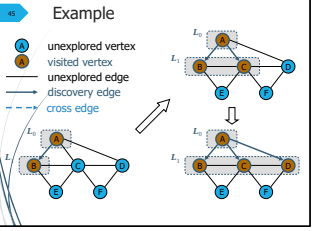
Breadth-First Search

Java Implementation

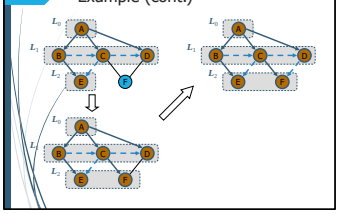

```
1  /** Performs breadth-first search of Graph g starting at Vertex u. */
2  public static <V,E> void BFS(Graph<V,E> g, Vertex<V> s,
3                               Set<Vertex<V>> known, Map<Vertex<V>,Edge<E>> forest) {
4    PositionalList<Vertex<V>> level = new LinkedPositionalList<>();
5    known.add(s);
6    level.addLast(s);                        // first level includes only s
7    while (!level.isEmpty()) {
8      PositionalList<Vertex<V>> nextLevel = new LinkedPositionalList<>();
9      for (Vertex<V> u : level)
10       for (Edge<E> e : g.outgoingEdges(u)) {
11         Vertex<V> v = g.opposite(u, e);
12         if (!known.contains(v)) {
13           known.add(v);
14           forest.put(v, e);                // e is the tree edge that discovered v
15           nextLevel.addLast(v);            // v will be further considered in next pass
16         }
17       }
18     level = nextLevel;                     // relabel 'next' level to become the current
19   }
20 }
```


Example

- A unexplored vertex
- A visited vertex
- unexplored edge
- discovery edge
- cross edge



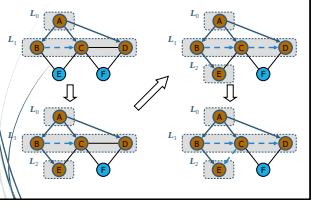
Example (cont.)



Analysis

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence L_i
- Method incidentEdges is called once for each vertex
- BFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\sum_v \deg(v) = 2m$

Example (cont.)



Properties

Notation

G_s: connected component of s

Property 1

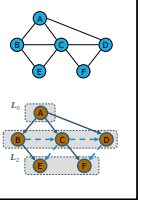
$BFS(G, s)$ visits all the vertices and edges of G_s

Property 2

The discovery edges labeled by $BFS(G, s)$ form a spanning tree T_s of G_s

Property 3

For each vertex v in L_i
- The path of T_s from s to v has i edges
- Every path from s to v in G_s has at least i edges

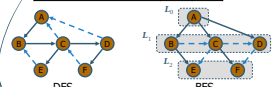


Applications

- Using the template method pattern, we can specialize the BFS traversal of a graph G to solve the following problems in $O(n + m)$ time
  - Compute the connected components of G
  - Compute a spanning forest of G
  - Find a simple cycle in G, or report that G is a forest
  - Given two vertices of G, find a path in G between them with the minimum number of edges, or report that no such path exists

DFS vs. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√
Biconnected components	√	



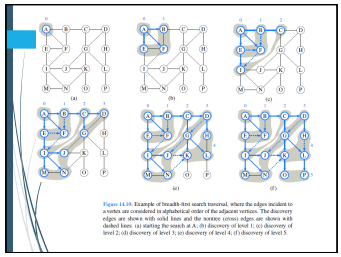
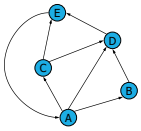


Figure 14.10: Example of breadth-first search traversal, where the edges incident to a vertex are considered in alphabetical order of the adjacent vertices. The discovery edges are shown with solid lines and the nontree (cross) edges are shown with dashed lines: (a) starting the search at A; (b) discovery of level 1; (c) discovery of level 2; (d) discovery of level 3; (e) discovery of level 4; (f) discovery of level 5.

Digraphs

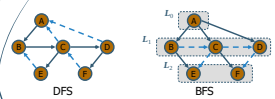
- A digraph is a graph whose edges are all directed
  - Short for "directed graph"
- Applications
  - one-way streets
  - flights
  - task scheduling



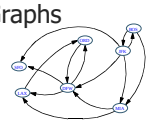
DFS vs. BFS (cont.)

Back edge (v,w)
- w is an ancestor of v in the tree of discovery edges

Cross edge (v,w)
- w is in the same level as v or in the next level

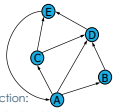


Directed Graphs



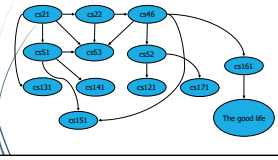
Digraph Properties

- A graph G=(V,E) such that
  - Each edge goes in one direction:
  - Edge (a,b) goes from a to b, but not b to a
- If G is simple, $m \leq n \cdot (n-1)$
- If we keep in-edges and out-edges in separate adjacency lists, we can perform listing of incoming edges and outgoing edges in time proportional to their size



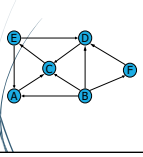
Digraph Application

- Scheduling: edge (a,b) means task a must be completed before b can be started



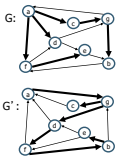
Reachability

- DFS tree rooted at v: vertices reachable from v via directed paths



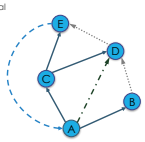
Strong Connectivity Algorithm

- Pick a vertex v in G
- Perform a DFS from v in G
  - If there's a w not visited, print "no"
- Let G' be G with edges reversed
- Perform a DFS from v in G'
  - If there's a w not visited, print "no"
  - Else, print "yes"
- Running time: O(n+m)



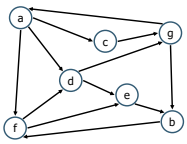
Directed DFS

- We can specialize the traversal algorithms (DFS and BFS) to digraphs by traversing edges only along their direction
- In the directed DFS algorithm, we have four types of edges
  - discovery edges
  - back edges
  - forward edges
  - cross edges
- A directed DFS starting at a vertex s determines the vertices reachable from s



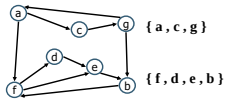
Strong Connectivity

- Each vertex can reach all other vertices



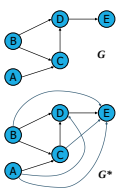
Strongly Connected Components

- Maximal subgraphs such that each vertex can reach all other vertices in the subgraph
- Can also be done in O(n+m) time using DFS, but is more complicated (similar to biconnectivity).

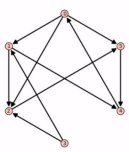


Transitive Closure

- Given a digraph G, the transitive closure of G is the digraph G^* such that
  - G^* has the same vertices as G
  - if G has a directed path from u to v ($u \neq v$), G^* has a directed edge from u to v
- The transitive closure provides reachability information about a digraph



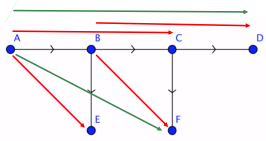
Adjacency Matrix (Rem.)



$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

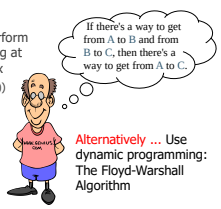
M[i][j] or M[i,j] = Entry in row i, column j

Warshall Algorithm - Concept



Computing the Transitive Closure

- We can perform DFS starting at each vertex
  - O(n(n+m))

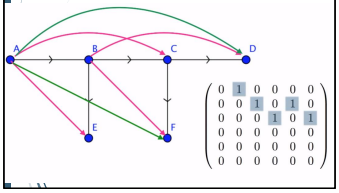


Reminder: And / Or

b1	b2	b1 AND b2
1	1	1
1	0	0
0	1	0
0	0	0

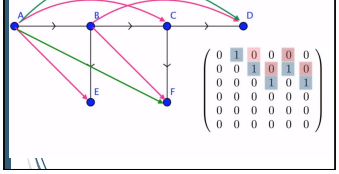
b1	b2	b1 OR b2
1	1	1
1	0	1
0	1	1
0	0	0

First Level Connections



$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Second Level Connections



$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Warshall's Algorithm (Adjacency Matrix)

```
1  def warshall(M):
2      n = M.nrows()
3      W = M
4      for k in range(n):
5          for i in range(n):
6              for j in range(n):
7                  W[i,j] = W[i,j] or (W[i,k] and W[k,j])
8      return W
```

- Time Efficiency $O(n^3)$
- Space Efficiency: **Matrices can be written over their predecessors.**

Floyd-Warshall's Algorithm (Generic – Adjacency List)

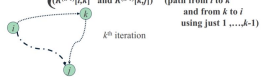
- Number vertices $v_1, \ldots, v_n$
- Compute digraphs $G_0, \ldots, G_n$
  - $G_0 = G$
  - G_k has directed edge (v_i, v_j) if G has a directed path from v_i to v_j with intermediate vertices in $\{v_1, \ldots, v_k\}$
- We have that $G_n = G^*$
- In phase k, digraph G_k is computed from G_{k-1}
- Running time: $O(n^3)$, assuming areAdjacent is $O(1)$ (e.g., adjacency matrix)

Algorithm *FloydWarshall(G)*
Input digraph G
Output transitive closure G^* of G
$i \leftarrow 1$
for all $v \in G.vertices()$
  denote v as v_i
  $i \leftarrow i + 1$
$G_0 \leftarrow G$
for $k \leftarrow 1$ **to** n **do**
  $G_k \leftarrow G_{k-1}$
  for $i \leftarrow 1$ **to** n ($i \neq k$) **do**
    for $j \leftarrow 1$ **to** n ($j \neq i, k$) **do**
      if $G_{k-1}.areAdjacent(v_i, v_k) \wedge$ $G_{k-1}.areAdjacent(v_k, v_j) \wedge$ $\neg G_{k-1}.areAdjacent(v_i, v_j)$
        $G_k.insertDirectedEdge(v_i, v_j, k)$
return G_n

Warshall's Algorithm (Math)

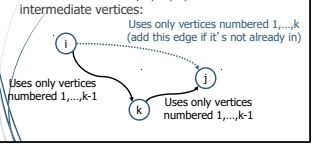
- On the kth iteration the algorithm determines if a path exists between two vertices i,j using just vertices among 1,.....,k allowed as intermediate

$$R^{(k)}[i,j] = \begin{cases} R^{(k-1)}[i,j] & \text{(path using just } 1,\ldots,k\text{-1)} \\ \text{or} & \\ R^{(k-1)}[i,k] \text{ and } R^{(k-1)}[k,j] & \text{(path from } i \text{ to } k \text{ and from } k \text{ to } j \text{ using just } 1,\ldots,k\text{-1)} \end{cases}$$



Floyd-Warshall Transitive Closure

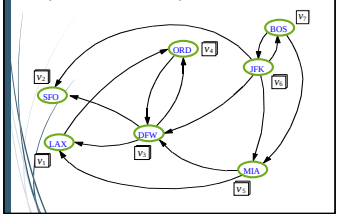
- Idea #1: Number the vertices 1, 2, …, n.
- Idea #2: Consider paths that use only vertices numbered 1, 2, …, k, as intermediate vertices:



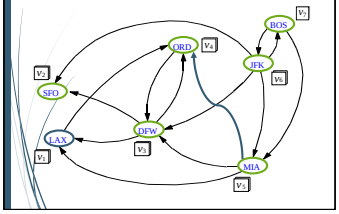
Java Implementation

```
1  /** Converts graph g into its transitive closure. */
2  public static <V,E> void transitiveClosure(Graph<V,E> g) {
3    for (Vertex<V> k : g.vertices())
4      for (Vertex<V> i : g.vertices())
5        // verify that edge (i,k) exists in the partial closure
6        if (i != k && g.getEdge(i,k) != null)
7          for (Vertex<V> j : g.vertices())
8            // verify that edge (k,j) exists in the partial closure
9            if (i != j && j != k && g.getEdge(k,j) != null)
10             // if (i,j) not yet included, add it to the closure
11             if (g.getEdge(i,j) == null)
12               g.insertEdge(i, j, null);
13 }
```

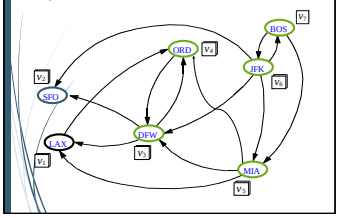

Floyd-Warshall Example



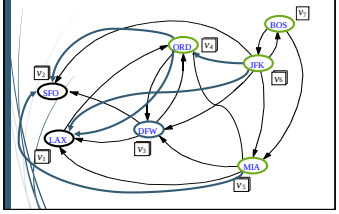
Floyd-Warshall, Iteration 1



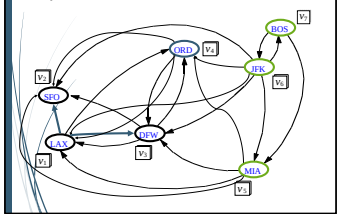
Floyd-Warshall, Iteration 2



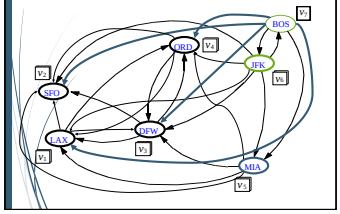
Floyd-Warshall, Iteration 3



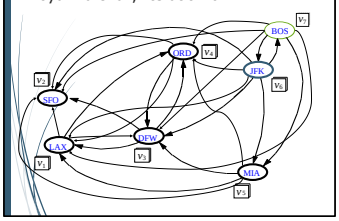
Floyd-Warshall, Iteration 4



Floyd-Warshall, Iteration 5



Floyd-Warshall, Iteration 6

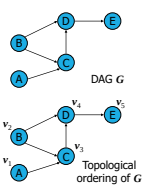


DAGs and Topological Ordering

- A directed acyclic graph (DAG) is a digraph that has no directed cycles
- A topological ordering of a digraph is a numbering $v_1, \ldots, v_n$ of the vertices such that for every edge (v_i, v_j), we have $i < j$
- Example: in a task scheduling digraph, a topological ordering a task sequence that satisfies the precedence constraints

Theorem

A digraph admits a topological ordering if and only if it is a DAG



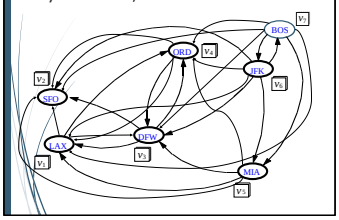
Algorithm for Topological Sorting

- Note: This algorithm is different than the one in the book

Algorithm TopologicalSort(*G*)
  $H \leftarrow G$ // Temporary copy of G
  $n \leftarrow G.numVertices()$
  while H is not empty **do**
    Let v be a vertex with no outgoing edges
    Label $v \leftarrow n$
    $n \leftarrow n - 1$
    Remove v from H

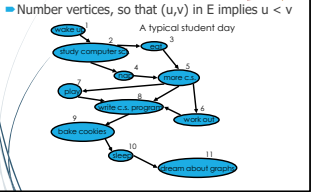
- Running time: O(n + m)

Floyd-Warshall, Conclusion



Topological Sorting

- Number vertices, so that (u,v) in E implies u < v



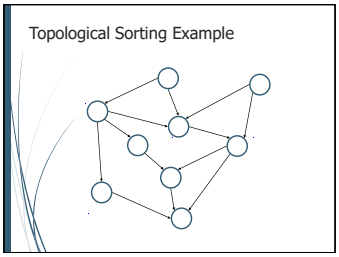
Implementation with DFS

- Simulate the algorithm by using depth-first search
- O(n+m) time.

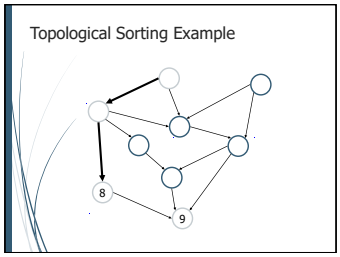
Algorithm *topologicalDFS(G)*
Input dag G
Output topological ordering of G
$n \leftarrow G.numVertices()$
for all $u \in G.vertices()$
  setLabel(u, UNEXPLORED)
for all $v \in G.vertices()$
  if *getLabel(v) = UNEXPLORED*
    topologicalDFS(G, v)

Algorithm *topologicalDFS(G, v)*
Input graph G and a start vertex v of G
Output labeling of the vertices of G in the connected component of v
setLabel(v, VISITED)
for all $e \in G.outEdges(v)$
  { outgoing edges }
  $w \leftarrow opposite(v,e)$
  if *getLabel(w) = UNEXPLORED*
    { e is a discovery edge }
    topologicalDFS(G, w)
  else
    { e is a forward or cross edge }
  Label v with topological number n
  $n \leftarrow n - 1$

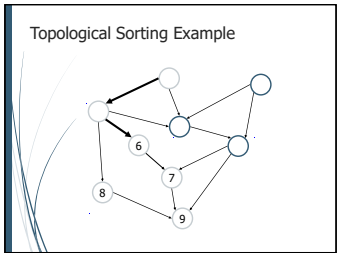
Topological Sorting Example



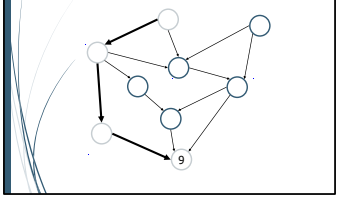
Topological Sorting Example



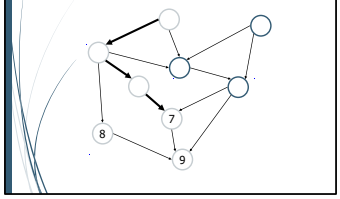
Topological Sorting Example



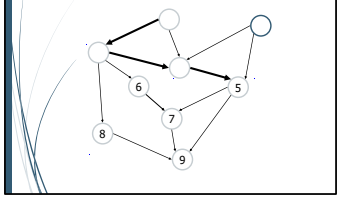
Topological Sorting Example



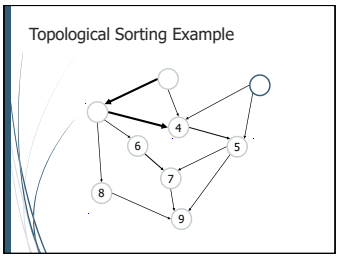
Topological Sorting Example



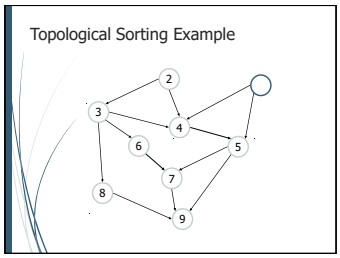
Topological Sorting Example



Topological Sorting Example



Topological Sorting Example

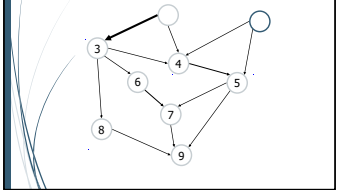


Java Implementation

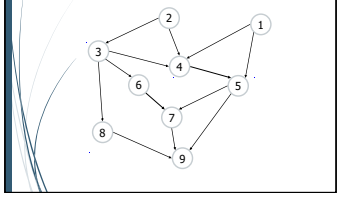
```
1  /** Returns a list of vertices of directed acyclic graph g in topological order. */
2  public static <V,E> PositionalList<Vertex<V>> topologicalSort(Graph<V,E> g) {
3    // list of vertices placed in topological order
4    PositionalList<Vertex<V>> topo = new LinkedPositionalList<>();
5    // container of vertices that have no remaining constraints
6    Stack<Vertex<V>> ready = new LinkedStack<>();
7    // map keeping track of remaining in-degree for each vertex
8    Map<Vertex<V>, Integer> inCount = new ProbeHashMap<>();
9    for (Vertex<V> u : g.vertices()) {
10     inCount.put(u, g.inDegree(u));          // initialize with actual in-degree
11     if (inCount.get(u) == 0)                // if u has no incoming edges,
12       ready.push(u);                        // it is free of constraints
13   }
14   while (!ready.isEmpty()) {
15     Vertex<V> u = ready.pop();
16     topo.addLast(u);
17     for (Edge<E> e : g.outgoingEdges(u)) {  // consider all outgoing neighbors of u
18       Vertex<V> v = g.opposite(u, e);
19       inCount.put(v, inCount.get(v) - 1);   // v has one less constraint without u
20       if (inCount.get(v) == 0)
21         ready.push(v);
22     }
23   }
24   return topo;
25 }
```

Directed Graphs

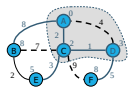
Topological Sorting Example



Topological Sorting Example



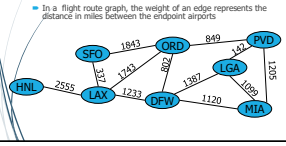
Shortest Paths



99

Weighted Graphs

- In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- Edge weights may represent, distances, costs, etc.
- Example:
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports



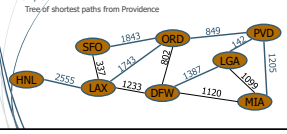
101

Shortest Path Properties

Property 1:
A subpath of a shortest path is itself a shortest path

Property 2:
There is a tree of shortest paths from a start vertex to all the other vertices

Example:
Tree of shortest paths from Providence

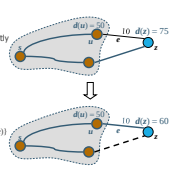


103

Edge Relaxation

- Consider an edge $e = (u,z)$ such that
  - u is the vertex most recently added to the cloud
  - z is not in the cloud
- The relaxation of edge e updates distance $d(z)$ as follows:

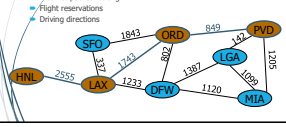
$d(z) \leftarrow \min(d(z), d(u) + weight(e))$



100

Shortest Paths

- Given a weighted graph and two vertices u and v, we want to find a path of minimum total weight between u and v.
  - Length of a path is the sum of the weights of its edges.
- Example:
  - Shortest path between Providence and Honolulu
- Applications
  - Internet packet routing
  - Flight reservations
  - Driving directions



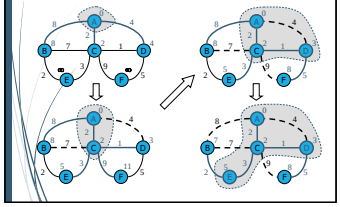
102

Dijkstra's Algorithm

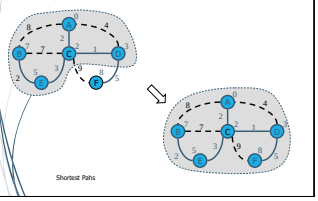
- The distance of a vertex v from a vertex s is the length of a shortest path between s and v
- Dijkstra's algorithm computes the distances of all the vertices from a given start vertex s
- Assumptions:
  - the graph is connected
  - the edges are undirected
  - the edge weights are nonnegative
- We grow a "cloud" of vertices, beginning with s and eventually covering all the vertices
- We store with each vertex v a label $d(v)$ representing the distance of v from s in the subgraph consisting of the cloud and its adjacent vertices
- At each step
  - We add to the cloud the vertex u outside the cloud with the smallest distance label, $d(u)$
  - We update the labels of the vertices adjacent to u

104

Example



Example (cont.)



Shortest Paths

Analysis of Dijkstra's Algorithm

- Graph operations
  - We find all the incident edges once for each vertex
- Label operations
  - We set/get the distance and locator labels of vertex z $O(\deg(z))$ times
  - Setting/getting a label takes $O(1)$ time
- Priority queue operations
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $O(\log n)$ time
  - The key of a vertex in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time
- Dijkstra's algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list/map structure
  - Recall that $\sum_v \deg(v) = 2m$
- The running time can also be expressed as $O(m \log n)$ since the graph is connected

Java Implementation, 2

```
24  // now begin adding reachable vertices to the cloud
25  while (!pq.isEmpty()) {
26    Entry<Integer, Vertex<V>> entry = pq.removeMin();
27    int key = entry.getKey();
28    Vertex<V> u = entry.getValue();
29    cloud.put(u, key);                    // this is actual distance to u
30    pqTokens.remove(u);                   // u is no longer in pq
31    for (Edge<Integer> e : g.outgoingEdges(u)) {
32      Vertex<V> v = g.opposite(u,e);
33      if (cloud.get(v) == null) {
34        // perform relaxation step on edge (u,v)
35        int wgt = e.getElement();
36        if (d.get(u) + wgt < d.get(v)) {  // better path to v?
37          d.put(v, d.get(u) + wgt);       // update the distance
38          pq.replaceKey(pqTokens.get(v), d.get(v));  // update the pq entry
39        }
40      }
41    }
42  }
43  return cloud;                           // this only includes reachable vertices
44  }
```

Dijkstra's Algorithm

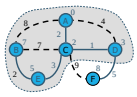
Algorithm ShortestPath(G,s):
Input: A weighted graph G with nonnegative edge weights, and a distinguished vertex s of G.
Output: The length of a shortest path from s to v for each vertex v of G.
Initialize $D[s] = 0$ and $D[v] = \infty$ for each vertex $v \neq s$.
Let a priority queue Q contain all the vertices of G using the D labels as keys.
while Q is not empty **do**
  {pull a new vertex u into the cloud}
  u = value returned by Q.remove_min()
  for each vertex v adjacent to u such that v is in Q **do**
    {perform the *relaxation* procedure on edge (u, v)}
    if $D[u] + w(u, v) < D[v]$ **then**
      $D[v] = D[u] + w(u, v)$
      Change to $D[v]$ the key of vertex v in Q.
return the label $D[v]$ of each vertex v

Java Implementation

```
1  /** Computes shortest-path distances from src vertex to all reachable vertices of g. */
2  public static <V> Map<Vertex<V>, Integer>
3  shortestPathLengths(Graph<V,Integer> g, Vertex<V> src) {
4    // d.get(v) is upper bound on distance from src to v
5    Map<Vertex<V>, Integer> d = new ProbeHashMap<>();
6    // map reachable v to its d value
7    Map<Vertex<V>, Integer> cloud = new ProbeHashMap<>();
8    // pq will have vertices as elements, with d.get(v) as key
9    AdaptablePriorityQueue<Integer, Vertex<V>> pq;
10   pq = new HeapAdaptablePriorityQueue<>();
11   // maps from vertex to its pq locator
12   Map<Vertex<V>, Entry<Integer,Vertex<V>>> pqTokens;
13   pqTokens = new ProbeHashMap<>();
14
15   // for each vertex v of the graph, add an entry to the priority queue, with
16   // the source having distance 0 and all others having infinite distance
17   for (Vertex<V> v : g.vertices()) {
18     if (v == src)
19       d.put(v,0);
20     else
21       d.put(v, Integer.MAX_VALUE);
22     pqTokens.put(v, pq.insert(d.get(v), v));      // save entry for future updates
23   }
```

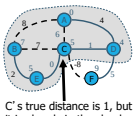
Why Dijkstra's Algorithm Works

- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.
  - Suppose it didn't find all shortest distances. Let F be the first wrong vertex the algorithm processed.
  - When the previous node, D, on the true shortest path was considered, its distance was correct
  - But the edge (D,F) was relaxed at that time!
  - Thus, so long as d(F)≥d(D), F's distance cannot be wrong. That is, there is no wrong vertex



Why It Doesn't Work for Negative-Weight Edges

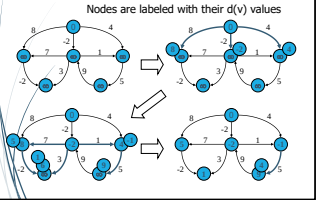
- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.
- If a node with a negative incident edge were to be added late to the cloud, it could mess up distances for vertices already in the cloud.



C's true distance is 1, but it is already in the cloud with d(C)=5!

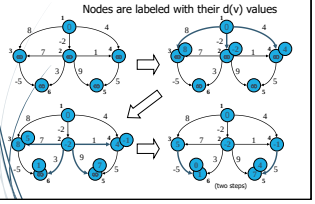
Bellman-Ford Example

Nodes are labeled with their d(v) values



DAG Example

Nodes are labeled with their d(v) values



(two steps)

Bellman-Ford Algorithm (not in book)

- Works even with negative-weight edges
- Must assume directed edges (for otherwise we would have negative-weight cycles)
- Iteration i finds all shortest paths that use i edges.
- Running time: O(nm).
- Can be extended to detect a negative-weight cycle if it exists
  - How?

Algorithm *BellmanFord*(G, s)
  for all $v \in G.vertices()$
    if $v = s$
      setDistance(v, 0)
    else
      setDistance(v, ∞)
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    for each $e \in G.edges()$
      { relax edge e }
      $u \leftarrow G.origin(e)$
      $z \leftarrow G.opposite(u,e)$
      $r \leftarrow getDistance(u) + weight(e)$
      if $r < getDistance(z)$
        setDistance(z,r)

DAG-based Algorithm (not in book)

- Works even with negative-weight edges
- Uses topological order
- Doesn't use any fancy data structures
- Is much faster than Dijkstra's algorithm
- Running time: O(n+m).

Algorithm *DagDistances*(G, s)
  for all $v \in G.vertices()$
    if $v = s$
      setDistance(v, 0)
    else
      setDistance(v, ∞)
  { Perform a topological sort of the vertices }
  for $u \leftarrow 1$ **to** n **do** {in topological order}
    for each $e \in G.outEdges(u)$
      { relax edge e }
      $z \leftarrow G.opposite(u,e)$
      $r \leftarrow getDistance(u) + weight(e)$
      if $r < getDistance(z)$
        setDistance(z,r)

Minimum Spanning Trees

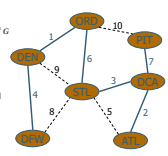
Minimum Spanning Trees

Spanning subgraph
- Subgraph of a graph G containing all the vertices of G

Spanning tree
- Spanning subgraph that is itself a (free) tree

Minimum spanning tree (MST)
- Spanning tree of a weighted graph with minimum total edge weight

Applications
- Communications networks
- Transportation networks

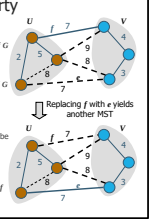


Minimum Spanning Trees

Partition Property

Partition Property:
- Consider a partition of the vertices of G into subsets U and V
- Let e be an edge of minimum weight across the partition
- There is a minimum spanning tree of G containing edge e

Proof:
- Let T be an MST of G
- If T does not contain e, consider the cycle C formed by e with T and let f be an edge of C across the partition
- By the cycle property, $weight(f) \leq weight(e)$
- Thus, $weight(f) = weight(e)$
- We obtain another MST by replacing f with e



Minimum Spanning Trees

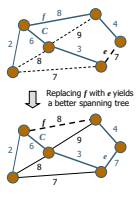
Prim-Jarnik Pseudo-code

Algorithm PrimJarnik(G):
  Input: An undirected, weighted, connected graph G with n vertices and m edges
  Output: A minimum spanning tree T for G
  Pick any vertex s of G
  $D[s] = 0$
  for each vertex $v \neq s$ **do**
    $D[v] = \infty$
  Initialize $T = \emptyset$.
  Initialize a priority queue Q with an entry $(D[v], (v, \text{None}))$ for each vertex v, where $D[v]$ is the key in the priority queue, and (v, None) is the associated value.
  while Q is not empty **do**
    (u, e) = value returned by Q.remove_min()
    Connect vertex u to T using edge e.
    for each edge $e' = (u, v)$ such that v is in Q **do**
      {check if edge (u, v) better connects v to T}
      if $w(u, v) < D[v]$ **then**
        $D[v] = w(u, v)$
        Change the key of vertex v in Q to $D[v]$.
        Change the value of vertex v in Q to (v, e').
  return the tree T

Cycle Property

Cycle Property:
- Let T be a minimum spanning tree of a weighted graph G
- Let e be an edge of G that is not in T and C let be the cycle formed by e with T
- For every edge f of C, $weight(f) \leq weight(e)$

Proof:
- By contradiction
- If $weight(f) > weight(e)$ we can get a spanning tree of smaller weight by replacing e with f.



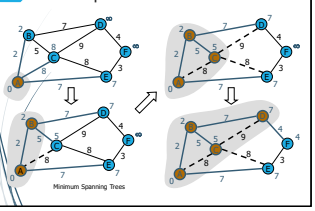
Minimum Spanning Trees

Prim-Jarnik's Algorithm

- Similar to Dijkstra's algorithm
- We pick an arbitrary vertex s and we grow the MST as a cloud of vertices, starting from s
- We store with each vertex v label $d(v)$ representing the smallest weight of an edge connecting v to a vertex in the cloud
- At each step:
  - We add to the cloud the vertex u outside the cloud with the smallest distance label
  - We update the labels of the vertices adjacent to u

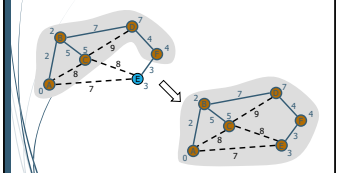
Minimum Spanning Trees

Example



Minimum Spanning Trees

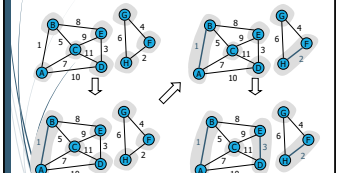
Example (contd.)



Kruskal's Approach

- Maintain a partition of the vertices into clusters
  - Initially, single-vertex clusters
  - Keep an MST for each cluster
  - Merge "closest" clusters and their MSTs
- A priority queue stores the edges outside clusters
  - Key: weight
  - Element: edge
- At the end of the algorithm
  - One cluster and one MST

Example



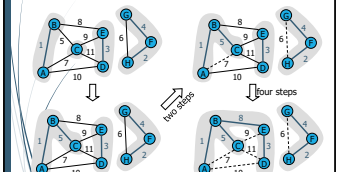
Analysis

- Graph operations
  - We cycle through the incident edges once for each vertex
- Label operations
  - We set/get the distance, parent and locator labels of vertex z $O(\deg(z))$ times
  - Setting/getting a label takes $O(1)$ time
- Priority queue operations
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $O(\log n)$ time
  - The key of a vertex w in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time
- Prim-Jarnik's algorithm runs in $O((n + m) \log n)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\sum_v \deg(v) = 2m$
- The running time is $O(m \log n)$ since the graph is connected

Kruskal's Algorithm

Algorithm Kruskal(G):
Input: A simple connected weighted graph G with n vertices and m edges
Output: A minimum spanning tree T for G
for each vertex v in G **do**
  Define an elementary cluster $C(v) = \{v\}$.
Initialize a priority queue Q to contain all edges in G, using the weights as keys.
$T = \emptyset$ {T will ultimately contain the edges of the MST}
while T has fewer than $n - 1$ edges **do**
  (u, v) = value returned by Q.remove_min()
  Let $C(u)$ be the cluster containing u, and let $C(v)$ be the cluster containing v.
  if $C(u) \neq C(v)$ **then**
    Add edge (u, v) to T.
    Merge $C(u)$ and $C(v)$ into one cluster.
return tree T

Example (contd.)



Data Structure for Kruskal's Algorithm

- The algorithm maintains a forest of trees
- A priority queue extracts the edges by increasing weight
- An edge is accepted if it connects distinct trees
- We need a data structure that maintains a partition, i.e., a collection of disjoint sets, with operations:
  - makeSet(u): create a set consisting of u
  - find(u): return the set storing u
  - union(A, B): replace sets A and B with their union

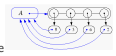
Partition-Based Implementation

- Partition-based version of Kruskal's Algorithm
  - Cluster merges as unions
  - Cluster locations as finds
- Running time $O((n + m) \log n)$
  - Priority Queue operations: $O(m \log n)$
  - Union-Find operations: $O(n \log n)$

Java Implementation, 2

```
18  int size = g.numVertices();
19  // while tree not spanning and unprocessed edges remain...
20  while (tree.size() != size - 1 && !pq.isEmpty()) {
21    Entry<Integer, Edge<Integer>> entry = pq.removeMin();
22    Edge<Integer> edge = entry.getValue();
23    Vertex<V>[] endpoints = g.endVertices(edge);
24    Position<Vertex<V>> a = forest.find(positions.get(endpoints[0]));
25    Position<Vertex<V>> b = forest.find(positions.get(endpoints[1]));
26    if (a != b) {
27      tree.addLast(edge);
28      forest.union(a,b);
29    }
30  }
31
32  return tree;
33  }
}
```


List-based Partition



- Each set is stored in a sequence
- Each element has a reference back to the set
  - operation find(u) takes O(1) time, and returns the set of which u is a member.
  - in operation union(A,B), we move the elements of the smaller set to the sequence of the larger set and update their references
  - the time for operation union(A,B) is min(|A|, |B|)
- Whenever an element is processed, it goes into a set of size at least double, hence each element is processed at most log n times

Java Implementation

```
1  /** Computes a minimum spanning tree of graph g using Kruskal's algorithm. */
2  public static <V> PositionalList<Edge<Integer>> MST(Graph<V,Integer> g) {
3    // tree is where we will store result as it is computed
4    PositionalList<Edge<Integer>> tree = new LinkedPositionalList<>();
5    // pq entries are edges of graph, with weights as keys
6    PriorityQueue<Integer, Edge<Integer>> pq = new HeapPriorityQueue<>();
7    // union-find forest of components of the graph
8    Partition<Vertex<V>> forest = new Partition<>();
9    // map each vertex to the forest position
10   Map<Vertex<V>,Position<Vertex<V>>> positions = new ProbeHashMap<>();
11
12   for (Vertex<V> v : g.vertices())
13     positions.put(v, forest.makeGroup(v));
14
15   for (Edge<Integer> e : g.edges())
16     pq.insert(e.getElement(), e);
17
```


Baruvka's Algorithm (Exercise)

- Like Kruskal's Algorithm, Baruvka's algorithm grows many clusters at once and maintains a forest T
- Each iteration of the while loop halves the number of connected components in forest T
- The running time is $O(m \log n)$

Algorithm *BaruvkaMST*(G)
  $T \leftarrow V$ {just the vertices of G}
while T has fewer than $n - 1$ edges **do**
  for each connected component C in T **do**
    Let edge e be the smallest-weight edge from C to another component in T
    if e is not already in T **then**
      Add edge e to T
return T

Example of Baruvka's Algorithm (animated)