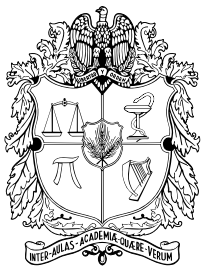


UNIVERSIDAD
NACIONAL
DE COLOMBIA

2016699

ESTRUCTURAS DE DATOS

Yoan Pinzón

Teacher opens the door, but you must enter by yourself

— Chinese Proverb

UNIVERSIDAD NACIONAL DE COLOMBIA

Estructuras de Datos

Sesión 1

Preliminaries and Recursion

Yoan Pinzón, PhD
Profesor Asociado
Universidad Nacional de Colombia
http://disi.unal.edu.co/∼ypinzon/2016699/

Table of Content Session 1

Table of Content Session 1

Preliminares

Horario de Clases

	Lunes	Martes	Miercoles	Jueves	Viernes
07-08	Grupo 1†		Grupo 1†		
08-09	Grupo 1†		Grupo 1†		
09-10	Grupo 2†			Grupo 2†	
10-11	Grupo 2†			Grupo 2†	
11-12					
12-01					
01-02					
02-03					
03-04					

† Salon 453-107

Horario de Consulta

	Lunes	Martes	Miercoles	Jueves	Viernes
07-08					
08-09					
09-10			Consulta‡		
10-11			Consulta‡		
11-12					
12-01					
01-02					
02-03					
03-04					

‡ Oficina 453-116

Durante el *horario de consulta*, el estudiante tiene la oportunidad de consultar sus dudas sobre algunos de los aspectos vistos en clase, y/o discutir sobre los progresos de su trabajo

Créditos : Esta asignatura tiene una valor de 3 (tres) créditos

Intensidad Horaria : Esta es una asignatura teórico-práctica de 4 horas presenciales y 5 horas de trabajo personal por semana

Formato de la Clase :

- La clase comiezá transcurridos diez (10) minutos de la hora establecida para su inicio
  - Si llega tarde: esté suficientemente preparado porque ...

"Procuro ser siempre muy puntual, pues he observado que los defectos de una persona se reflejan muy vivamente en la memoria de quien la espera." — Nicolas Boileau - Despréaux

punctuality is a habit that can be developed. It takes great effort at first, but eventually simply happens. Habits are formed by repetition. If you want to acquire the habit of punctuality, you must repeat this behavior again and again.

Comportamiento en Clase

1. **Por respeto hacia los demás (y hacia mí)**
   - LLegue a tiempo
   - No conteste llamadas en clase
2. **Por respeto hacia usted mismo (y otros)**
   - Haga preguntas
     - Levante la mano; haga la pregunta
     - No sea timido; recuerde que no existen preguntas estúpidas!

> *"He who asks is a fool for five minutes, but he who does not ask remains a fool forever."* — Old Chinese saying

3. **Por respeto hacia el tema (y hacia mí)**
   - Haga preguntas propias
   - Trate de entender "el qué", "el cómo", "el porqué" y "el para qué"
   - Los temas difíciles no son aburridos
   - Los temas aburridos no son difíciles
   - Esté preparado para corregirme (educadamente)

Se cobrarán 2 pts por cada **celular** que *suene* en clase!!!

Bibliografía

- S. Sahni, Data Structures, Algorithms and Applications in Java, Silicon Press; 2nd edition, 2004.
- M. A. Weis, Data Structures and Algorithm Analysis in Java (2nd Edition), Addison-Wesley, 2006.
- M. T. Goodrich and R. Tamassia, Data Structures and Algorithms in Java, Wiley, 2010.
- H. M. Deitel and P. J. Deitel, Java How to Program, Prentice Hall (6th Edition), 2004
- A. Aho, J. Hopcroft and J. Ullman, Data Structures and Algorithms, Addison-Wesley, 1983.

Página Web

La página web del curso contendrá información relevante para los estudiantes, de modo que sugeriría que la revisara con frecuencia. La dirección URL es:

http://disi.unal.edu.co/~ypinzon/2016699/

La forma más fácil de llegar a la página es escribiendo "Yoan Pinzon" en Google, dar click en el primer resultado y después click en "Estructuras de Datos" en el menú superior-izquierdo.

Docente

Yoan Pinzón
Oficina : 453-116
eMail : ypinzon@unal.edu.co
WWW : http://disi.unal.edu.co/~ypinzon/

Monitor del Curso

T.B.A.

Calendario Académico

Feb 2011						Mar 2011						Abr 2011						May 2011						Jun 2011					
L	M	M	J	V	WEEK	L	M	M	J	V	WEEK	L	M	M	J	V	WEEK	L	M	M	J	V	WEEK	L	M	M	J	V	WEEK
31	1	2	3	4			1	2	3	4	4					1	8	2	3	4	5	6	12			1	2	3	16
7	8	9	10	11	1	7	8	9	10	11	5	4	5	6	7	8	9	9	10	11	12	13	13	6	7	8	9	10	
14	15	16	17	18	2	14	15	16	17	18	6	11	12	13	14	15	10	16	17	18	19	20	14	13	14	15	16	17	
21	22	23	24	25	3	21	22	23	24	25	7	18	19	20	21	22		23	24	25	26	27	15	20	21	22	23	24	
28					4	28	29	30	31		8	25	26	27	28	29	11	30	31				16	27	28	29	30		

- inicio/fin de clases
- primera evaluación ordinaria (28Mar2011)
- segunda evaluación ordinaria (16May2011)
- semana santa

Evaluaciones

Se tomarán CINCO (5) evaluaciones con los siguientes porcentajes:

	Valor Puntos	Valor Porcentaje
N1 Quices (07Feb-25Mar)	50	10%
N2 1ra Evaluación (28Mar)	150	30%
N3 Quices (29Mar-13May)	50	10%
N4 2da Evaluación (16May)	150	30%
N5 Talleres (07Feb-03Jun)	100	20%
Total:	**500**	**100%**

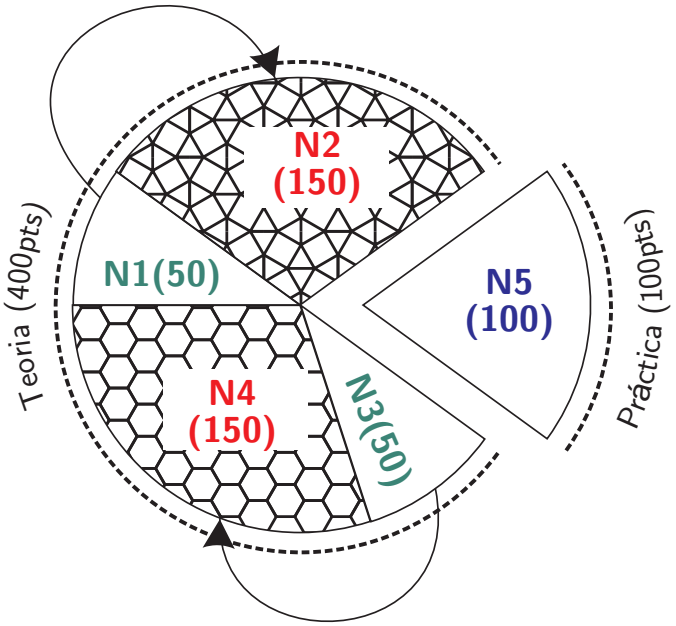
Todas las evaluaciones se darán en puntos. Al terminar el curso se habrán evaluado pruebas por un total de *500 puntos*

Para poder aprobar el curso satisfactoriamente Ud necesita acumular (al menos) 300 pts.

IMPORTANTE : Este curso **NO** es validable, quien lo repruebe deberá, forzosamente, cursar el curso nuevamente!

Evaluaciones

En resumen, esta asignatura tendrá un componente teórico de **80%** y un componente práctico de **20%**



Evaluaciones

- Si por algún caso usted sobrepasa el máximo de 50 puntos en la nota 1 (N1), usted puede trasferir los puntos excedentes para ser utilizados en la nota 2 (N2)
- De igual forma, si sobrepasa el máximo de 50 puntos en la nota 3 (N3), usted puede trasferir los puntos excedentes para ser utilizados en la nota 4 (N4)
- Si alcanza a sacar más de 150 puntos en la nota 2 (N2) o en la nota 4 (N4), los puntos excedentes pasaran a un fondo común que nunca será utilizado
- De igual manera, si alcanza a sacar más de 100 puntos en la nota 5 (N5), los puntos excedentes pasaran al fondo común arriba ya mencionado

Una vez entienda el sistema de puntos verá que es un sistema de evaluación que premia a los estudiantes que más se esfuerzan por aprender y no necesariamente al que sabe más.

Formato de las Evaluaciones Parciales

- Las notas 1 y 4 serán evaluaciones parciales escritas
- Cada evaluación tendra un valor de 150pts o 30% de la nota final
- Cada evaluación contendra 4 preguntas, cada pregunta con un valor de 50 puntos, de las cuales usted debería contestar *tan solo 3* de ellas

IMPORTANTE: Si por algún caso constesta más de tres (3) preguntas, se tomarán las tres respuestas con menor número de puntos!

Evaluaciones Supletorias

- Cuando existen causas justificadas a juicio del profesor
- La solicitud de evaluación supletoria debe hacerse por *escrito* al profesor dentro de los cinco (5) días hábiles siguientes a la fecha de presentación de la evaluación prevista

Publicación de Calificaciones

- Los resultados de las evaluaciones se darán a conocer dentro de los diez (10) días hábiles siguientes a su realización.

Reclamación de Calificaciones

- La revisión de las calificaciones podrá ser reclamada, por una sola vez, dentro de los cinco (5) días hábiles siguientes a la entrega de la nota, ante el Director del Departamento de Ingeniería de Sistemas e Industrial.

Programa—Calendario

a. Identificación de la asignatura: nombre completo, código y número de créditos

Estructuras de Datos, 2016699, 3 créditos

b. Metodología a utilizar

Combinación de clase magistral y clase práctica de 4 horas de duración. Se introducen los temas teóricos, se explica la estructura de datos, se especifica el ADT (tipo de dato abstracto), se explica la codificación en Java haciendo énfasis en las diferentes formas que existen para representar datos en la memoria del computador, se ven algunas aplicaciones y se desarrolla un trabajo práctico sobre el tema visto.

c. Distribución de los temas y subtemas

Semana 1=Recursividad (que es recursividad, pasos de la recursividad, ejemplos:factorial, Fibonacci, el triángulo de Pascal, ejercicios)
Semana 2,3=Introducción al Java (que es Java, siglas, program structure, applications vs applets, platform independent, data types, primitive data types, reference data types, methods, argument passing, method overloading, argument promotion, objects and clases, inheritance, abstract vs interfaces, polymorphism)
Semana 4=LinearList using a Formula-based representation (methods of representing data, linear list ADT, linear list coding in Java)
Semana 5=Chain using a Linked representation (conding in Java, comparing array based representation and linked representation)
Semana 6,7=Stack Data Structure (stack ADT, formula based representation using inheritance and a customised implementation, linked representation using inheritance and a customised implementation, two applications : parentheses matching and switchbox routing)
Semana 8,9=Queue Data Structure (queue ADT, formula based representation using a customised implementation, linked representation using a customised implementation, two applications : image component labeling and lee´s wire router)
Semana 10=Analysis of Algorithms (growth of functions, growth rates, asymptoctic notation, sorting, selection sort, bubble sort, insertion sort, merge sort, quick sort)
Semana 11,12=Tree Data Structure (terminology, binary trees, properties, Binary Tree ADT, fomula based representation, linked representation, binary tree traversals, an application)
Semana 13,14=Priority Queue Data Structure (max priority queues ADT, heaps, formula based representation, an application)
Semana 15,16=Dictionary Data Structure (dictionary ADT, linear list representation, skip list representation, hash table representation)

d. Carácter, tipo y ponderación de pruebas

N1, quices, 10%
N2, evaluación escrita, ordinaria, 30%
N3, quices, 10%
N4, evaluación escrita, ordinaria, 30%
N5, talleres, 20%

e. Fechas de presentación de pruebas

N2(28Mar) N4(16May)

f. Porcentaje mínimo exigido de asistencia

80%

g. Bibliografía

• S. Sahni, Data Structures, Algorithms and Applications in Java, Silicon Press; 2nd edition, 2004.
• A Aho, J Hopcroft and J Ullman, Data Structures and Algorithms, Addison-Wesley, 1983.

h. Si la asignatura es validable o no

NO es validable

Recursion

- A **recursive method** invokes itself
- Recursion normally yields a shorter code
- In general recursive algorithms are very elegant
- Consists of two parts:
  - ▷ **base** step
  - ▷ **recursive** step

The canonical example of a recursive method is *factorial*

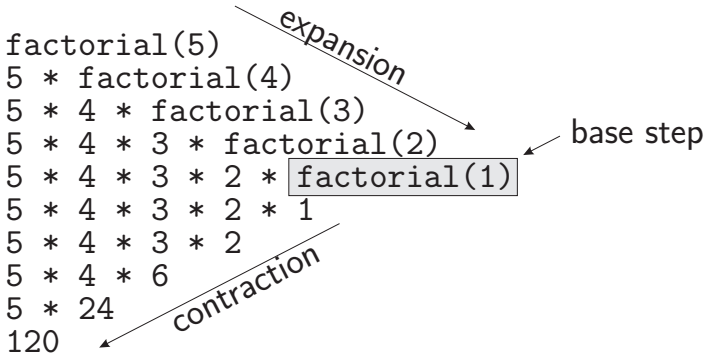
Example 1: Factorial

$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$

$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$

Observation:
$n! = n \cdot (n-1)!$ (recursive step)
$1! = 1$ (base step)

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

   public static void main(String args[])
   {
      System.out.println(factorial(5));
   }
}
```



An **iterative** solution: This is the traditional way (using loops).

```
public static int factorial(int n)
{
   int result = 1 ;

   for ( int i = 1; i <= n; i ++ )
      result = result * i ;

   return result ;
}
```


Example 2: Fibonacci Numbers

The Fibonacci series begins with 1 and 1 and has the property that each subsequent Fibonacci number is the sum of previous two Fibonacci numbers.

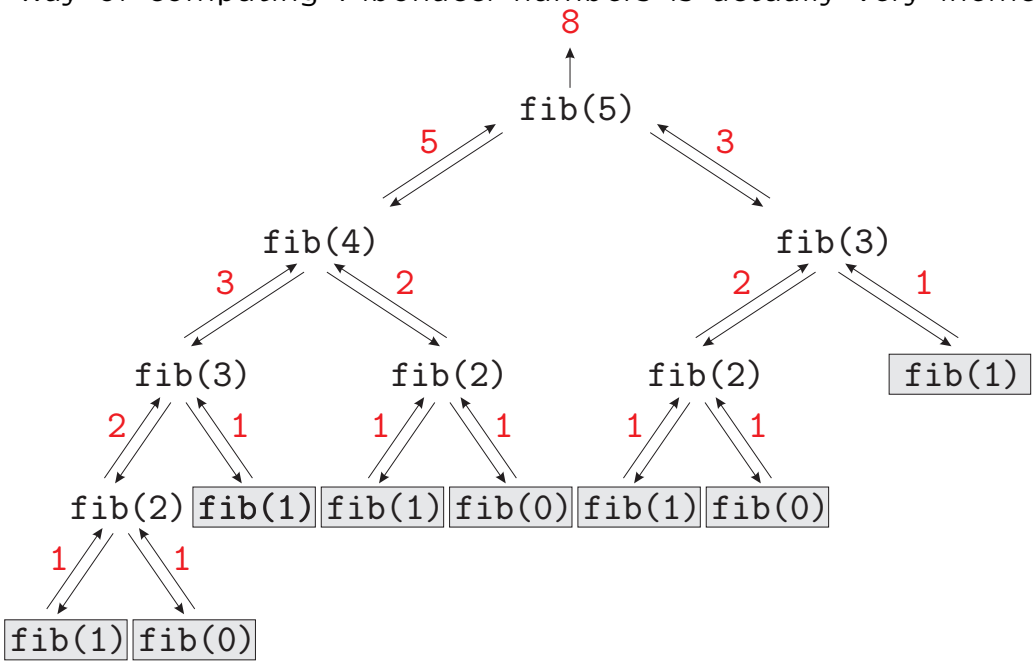
1, 1, 2, 3, 5, 8, 13, 21, 34, . .. Let's say that $fib(0) = 1, fib(1) = 1, fib(2) = 2, fib(3) = 3, fib(4) = 5, fib(5) = 8, \ldots$

Observation:
$fib(n) = fib(n-1) + fib(n-2)$ (recursive step)
$fib(0) = 1, fib(1) = 1$ (base step)

```
public static int fib(int n)
{
   if (n <= 1) return 1;
   else return fib(n-1) + fib(n-2);
}
```

How many times do we invoke the method fib when computing fib(5)?

This way of computing Fibonacci numbers is actually very inefficient.



A non-recursive Fibonacci Sequence

```
public static int fibo(int n)
{
   int n1 = 1;
   int n2 = 1;
   if ((n == 0) || (n == 1))
      return 1;
   else
   {
      int result = 0;
      for (int i = 2; i <= n; i++)
      {
         result = n1 + n2;
         n1 = n2;
         n2 = result;
      }
      return result;
   }
}
```

Example 3: The Pascal Triangle

The **binomial coefficients** are the coefficients that result from the expansion of a binomial expression of the form $(x+1)^n$.

For instance:

$(x+1)^3 = x^3 + 3x^2 + 3x + 1$

$(x+1)^6 = x^6 + 6x^5 + 15x^4 + 20x^3 + 15x^2 + 6x + 1$

Also called *Pascal Triangle*

	k=0	k=1	k=2	k=3	k=4	k=5	k=6	k=7
n=0	1							
n=1	1	1						
n=2	1	2	1					
n=3	1	3	3	1				
n=4	1	4	6	4	1			
n=5	1	5	10	10	5	1		
n=6	1	6	15	20	15	6	1	
n=7	1	7	21	35	35	21	7	1

$$pascal(n,k) = \begin{cases} 1, & k{=}0, k{=}n \\ pascal(n\text{-}1,k\text{-}1) + pascal(n\text{-}1,k), & \text{otherwise} \end{cases}$$


```
class Recursion
{
   public static long pascal (int n, int k)
   {
      if (k == 0 || k == n) return 1;
      else return pascal(n-1,k-1) + pascal(n-1,k);
   }

   public static void main(String args[])
   {
      for(int i=0; i< 20; i++)
      {
         for(int j=0; j <=i; j++)
            System.out.print(pascal(i,j)+"␣");
         System.out.println();
      }
   }
}
```

Example 4: Reversing a string

```
1   // This method makes use of recursion
2   // to print the reverse of a String
3   public static void printInv(String s)
4   {
5      if (s.length() == 1)
6         System.out.print(s);
7      else
8      {
9         printInv(s.substring(1));
10        System.out.print(s.charAt(0));
11     }
12  }
```

What happen if we swap lines 9 and 10?


```
// This method makes use of recursion
// to print a string character by character
public static void print(String s)
{
   if (s.length() == 1)
      System.out.print(s);
   else
   {
      System.out.print(s.charAt(0));
      print(s.substring(1));
   }
}
```

Observe that recursion should be used with care. This method differs from the one in the previous page in the order statements in lines 9 and 10 are executed.

Whereas the former prints the string reversed the later prints the string in its normal order.

Example 5: Reversing the tokens of a string

This recursive method uses a StringTokenizer to print the tokens of a String in the reverse order.

```
class Recursion
{
   public static void printReverse(StringTokenizer st)
   {
      if (st.hasMoreTokens())
      {
         String token = st.nextToken();
         printReverse(st);
         System.out.println(token);
      }
   }

   public static void main(String args[])
   {
      StringTokenizer st=new StringTokenizer(args[0],":");
      printReverse(st);
   }
}
```



```
$ javac Recursion.java
$ java Recursion How:are:you
you
are
How
```

Recursion is Difficult to Master

- Recursion yields elegant and concise code
- Yet, it is difficult to debug
- Minor mistakes can be disastrous for the program
- Infinite recursion is the most common problem
- Infinite recursion is more complicated than infinite loop because it consume all the memory by storing the status of the method before the recursive call
- For the inexperienced programmer, the program may sometimes report an error which has no immediate relation to the recursive method

When to Use Recursion

These rules are not precise, rather they should be seen as guidelines

- If the problem is recursive in nature, it is likely a recursive algorithm will be preferable and will be less complex
- If both recursive and non-recursive algorithms have the same complexity it is likely that a non-recursive version will perform better and therefore should be preferred

Problems You Should Try

- Greatest Common Divisor using the Euclidean's Algorithm

$$gcd(x, y) \leftarrow \begin{cases} x, & y = 0 \\ gcd(y, x \bmod y), & y > 0 \end{cases}$$

- The Ackermann's function

$$A(i, j) \leftarrow \begin{cases} 2^j, & i = 1 \text{ and } j \geq 1 \\ A(i-1, 2), & i \geq 2 \text{ and } j = 1 \\ A(i-1, A(i, j-1)), & i, j \geq 2 \end{cases}$$

- The "Towers of Hanoi" puzzle

Exercises

Find an appropriate name for the following methods:

1)
```
public static int f1(int n)
{
   return n * f1(n-1);
}
```

2)
```
public static int f2(int a, int b)
{
   if (a < 0) return f2(-a, -b);
   else if (a == 0) return 0;
   else return f2(a-1, b) + b;
}
```

3)
```
public static int f3(int n, int k)
{  // assume k >= 0
   if (k == 0) return 1;
   else return n * f3(n, k-1);
}
```

4)
```
public static int f4(int n, int k)
{  // assume k >= 0
   if (k == 0) return 1;
   else if (k % 2 == 0) return f4(n*n, k/2);
   else return n * f4(n, k-1);
}
```

A flawed diamond is better than a common stone that is perfect.

— Chinese Proverb

UNIVERSIDAD NACIONAL DE COLOMBIA

Estructuras de Datos

Sesión 2

Java Review

Yoan Pinzón, PhD
Profesor Asociado
Universidad Nacional de Colombia
http://disi.unal.edu.co/∼ypinzon/2016699/

Table of Content Session 2

What is Java

Java is a system originally developed in the mid 1990s by a team led by James Gosling at *Sun Microsystems*. I say "system" and not "language" because there are different things considered part of Java:

The Language Itself: Java as a language looks like C++, but get away from complexities that make it hard to program in C++. The result is a smaller, simpler language that still allows you to write useful programs.

The Library: Java comes with a very large and useful library of classes and interfaces. The Java class library supports a large number of capabilities, too numerous to mention at this point.

The Virtual Machine: The people who created Java were pretty specific about how a computer should actually run a Java program. They created a virtual machine specification, which is kind of like the design for a computer chip. Java basically runs programs by simulating that chip in software.

Common Acronyms

GUI: Graphical User Interface

AWT: Abstract Windowing Toolkit – This was the GUI used by JDK 1.0 - JDK 1.1

JRE: Java Runtime Environment – The JRE is a JVM that is loaded on an Operating System to run Java Applications. JRE & JVM is used interchangeably.

JVM: Java Virtual Machine – Software that allows Java bytecode to run on a physical machine. A JRE is a JVM. Browsers provide a JVM for Java Applets to execute.

JDK: Java Development Kit – The software package that allows developers to develop Java programs.

SDK: Software Development Kit – The software package that allows developers to develop Java programs. JDK & SDK is used interchangeably.

JAR: Java ARchive – A Jar is just a zip file with a renamed extension.

Program Structure

```
package package_name;

import class_name;
import package_name.*;
. . .

public class ClassName
{

   field-declaration 1;
   field-declaration 2;
   . . .

   method-definition 1;
   method-definition 2;

   . . .
```



```
   public static void main(String [] args)
   {
      declaration 1;
      declaration 2;
      . . .

      execution-statement 1;
      execution-statement 2;
      . . .
   }
}
```

The main() method

When any program is run, the special static method `main` is invoked, possibly with command-line arguments. The parameters types of `main` and the `void` return type shown are required.

Comments

Java has tree types of comments:

(1) Multiline comments (`/* .... */`): Inherited from C. All the text between the delimiters of the comment is ignored by the compiler.

(2) Single-line comments (`//`): Inherited from C++. Anything following the token `//` will be ignored by the compiler but only to the end of the line.

(3) Javadoc comments (`/** .... */`): This form can be used to provide information to the `javadoc` utility, which will extract them and automatically creates some HTML documentation for the program.

A well-commented program is a sign of a good programmer

The Braces

The opening brace ({) and the closing brace (}) mark the beginning and the end of the *program body*. Any code enclosed by the two braces is referred to as a *block*. Any variable, or method, defined in a block only has *scope* (i.e. is only available) within that block.

Identifiers

In Java, identifiers are used to name variables, constants, classes, and methods. Identifiers can not be *reserved words* as:

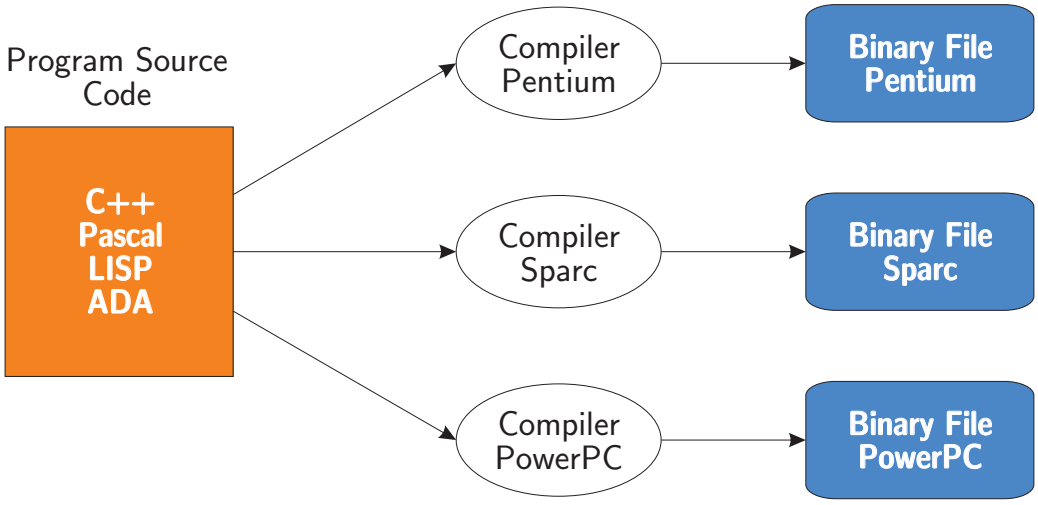
abstract	default	if	private	throw
boolean	do	implements	protected	throws
break	double	import	public	transient
byte	else	instanceof	return	try
case	extends	int	short	void
catch	final	interface	static	volatile
char	finally	long	super	while
class	float	native	switch	
const	for	new	synchronized	
continue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

Good Programming Practice

- Every program should begin with a comment that explains the purpose of the program, author, date and time. We are not showing this information in these slides because it would be redundant.
- Always begin a class name with a capital letter and start each subsequent word in the class name with a capital letter.
- Always begin a variable/method name with a lowercase letter. As with class names, every word in the name after the first word should begin with a capital letter.
- Indent each entire block of each method/class declaration one "level" of indentation between the left brace, {, and the right brace, }.
- Set a convention for the indent size. I recommend using three spaces.
- Place a space after each comma (,) in an argument list.
- Place spaces on either side of an operator to make the operator stand out.

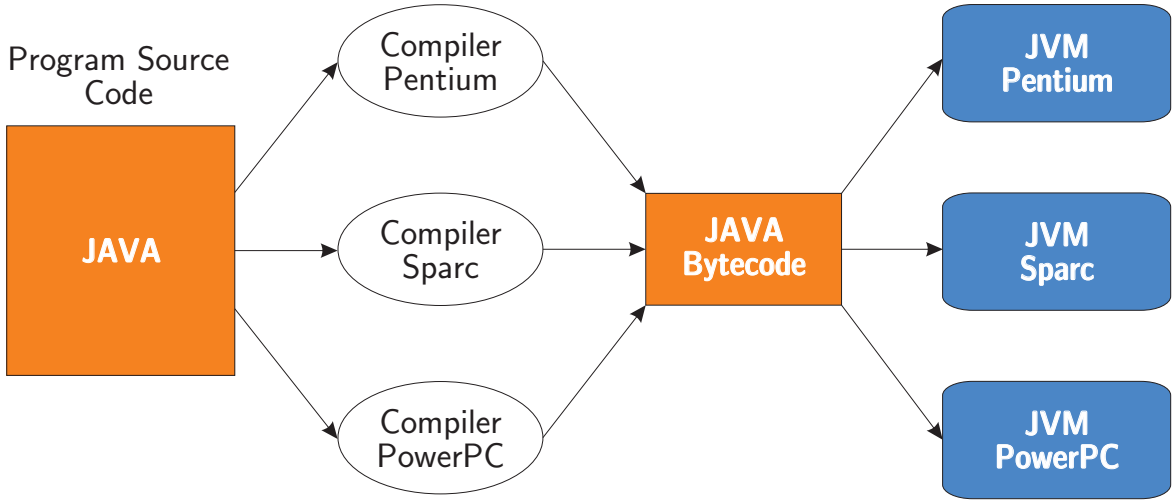
Platform Independent

Platform independence means that a program can run on any computer system. Java programs can run on any system for which a Java virtual machine has been installed.



(a) Traditional Compiled Programs

Platform Independent



(b) Java Programs

Applications Vs Applets

There are two ways to run a Java Executable File:

1) A Java Application: Stand-alone Java program that can be run just by invoking the java interpreter from the command line. All of our programs will be applications.

```
public class TrivialApplication {
   public static void main(String[] args) {
      System.out.println("Hello, Folks");
   }
}
```

2) A Java Applet: Light-weight Java program that are embedded in a web page and executed by a Java-capable web browser.

Java Code

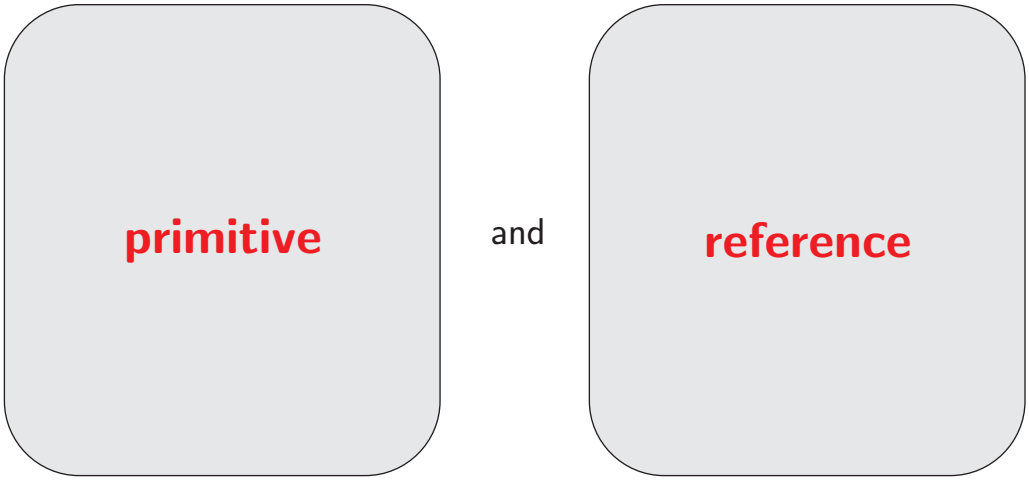
```
import java.awt.*;
import java.applet.Applet;
public class TrivialApplet extends Applet {
   public void paint(Graphics g) {
      g.drawString("Hello, Folks", 20, 20);
   }
}
```

HTML Code

```
<applet code="TrivialApplet.class" width=90 height=90>
</applet>
```

Data Types

Java has two categories of data types:



Primitive Data Types

Java has nine primitive type.

Type	Size (bits)	Range	Wrapper Type	
byte	8	[-128, 127]	Byte	Integers
short	16	[-32668, 32767]	Short	
int	32	[-2147483648, 2147483647]	Integer	
long	64	±9.2e+17	Long	
float	32	±1.4e-45 to ±3.4e+38	Float	Real Numbers
double	64	±4.9e-324 to ±1.8e+308	Double	
char	16	[\u000, \uFFFF]	Character	Other Type
boolean	1	[true, false]	Boolean	
void	--	--	Void	

In Java the format and size of primitive data types are not machine-dependent.

Reference Data Types

A reference is a variables that contains a memory address as its value.

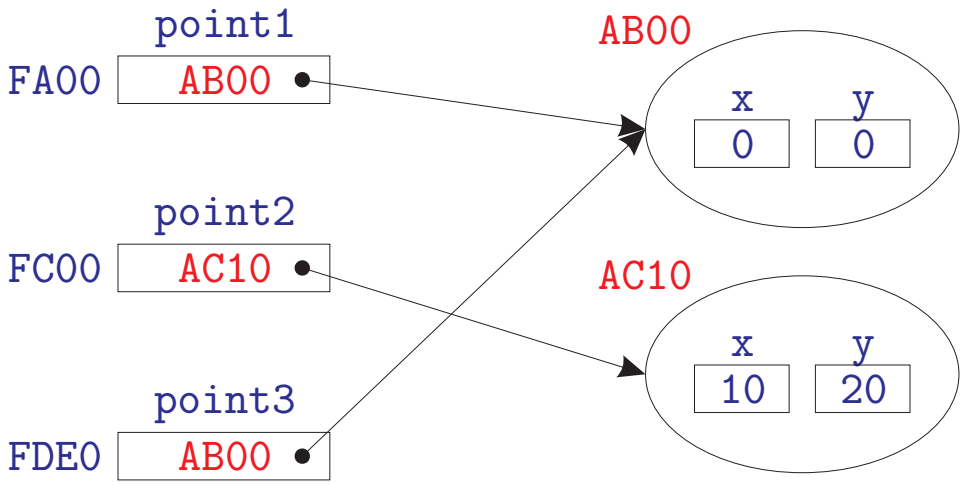
Example:

```
class Point {
   private int x, y;
   public Point() { x = y = 0; }
   public Point( int dx, int dy )
   {
      x = dx; y = dy;
   }
   public void move( int dx, int dy )
   {
      x += dx;
      y += dy;
   }
   public String toString()
   {
      return "(" + x + "," + y + ")";
   }
}
```

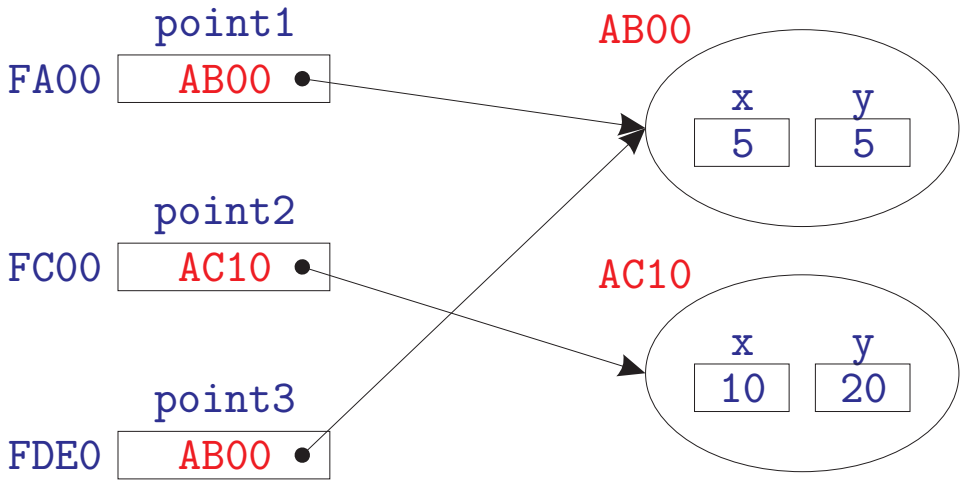


```
1  class PointTest {
2     public static void main( String [] args )
3     {
4        Point point1 = new Point();
5        Point point2 = new Point( 10,20 );
6        Point point3 = point1;
7
8        System.out.println( "point1=" + point1 );
9        System.out.println( "point2=" + point2 );
10       System.out.println( "point3=" + point3 );
11       System.out.println( point1 == point3 );
12       point1.move( 5, 5 );
13       System.out.println( "point1=" + point1 );
14       System.out.println( "point2=" + point2 );
15       System.out.println( "point3=" + point3 );
16       System.out.println( point1 == point3 );
17    }
18 }
```

```
4       Point point1 = new Point();
5       Point point2 = new Point( 10,20 );
6       Point point3 = point1;
```




```
12      point1.move( 5, 5 );
```



Methods

A method basically consists of:

- access modifier (`public`, `private`, `protected`)
- use modifier (`static`, `final`, `abstract`, `native`, `synchronized`)
- return type
- method name
- list of parameters
- body

```
access-modifier use-modifier return-type method-name(par1, par2, ...)
{
   . . .

   body-of-the-method;

   . . .
}
```


Argument Passing

Arguments passed to Java methods are **passed by value**. That is, a copy of the *arguments* is made and passed to the corresponding *parameters*.

Example1:

```
public class TestSwap {

   public static void swap( int x, int y )
   {
      int temp;
      temp = x; x = y; y = temp;
      return;
   }

   public static void main( String [] args )
   {
      int a = 4;
      int b = 5;

      swap( a, b );

      System.out.println( "a=" + a ); // 4
      System.out.println( "b=" + b ); // 5
   }
}
```

Example2:

```
public class TestSwap2 {

   public static void swap( Object x, Object y )
   {
      Object temp = x;
      x = y;
      y = temp;
   }

   public static void main( String [] args )
   {
      Integer a = new Integer( 4 );
      Integer b = new Integer( 5 );

      swap( a, b );

      System.out.println( "a=" + a ); // 4
      System.out.println( "b=" + b ); // 5
   }
}
```

Because Java passes all arguments by value, this code does not work

So how do you write a method in Java to swap the values of two primitive type or two object references?

Example3:

```
class Wraper {
   public Object obj;
}

public class TestSwap3 {

   public static void swap( Wraper x, Wraper y )
   {
      Object temp = x.obj;
      x.obj = y.obj;
      y.obj = temp;
   }

   public static void main( String [] args )
   {
      Wraper a = new Wraper();
      a.obj = new Integer( 4 );
      Wraper b = new Wraper();
      b.obj = new Integer( 5 );
      swap( a, b );
      System.out.println( "a=" + a.obj ); // 5
      System.out.println( "b=" + b.obj ); // 4
   }
}
```

Method Overloading

Simply, the same method name can be given several different definitions. The number and types of arguments supplied to a method tell the compiler which definition to use.

Example:

```
int max( int a, int b, int c )

int max( int a, int b)

int max( float a, float b, float c)

int max( float a, float b)
```

The compiler will ensure that the correct method is called based on the *signature* (that is, their parameter list types) passed to max().

The return type is NOT include in the signature. This means it is illegal to have two methods whose only difference is the return type.

When arguments do not exactly match the formal parameters types, the compiler will perform *argument promotion*.

Argument Promotion

There is a hierarchy of types. Arguments can be promoted to "higher" types.

Type	Valid Promotions
double	None
float	double
long	float or double
int	long, float or double
char	int, long, float or double
short	char, int long, float or double
byte	short, char, int long, float or double
boolean	None

boolean values are not consider to be numbers in Java

Example:

```
System.out.println( Math.sqrt( 4 ) ); // 2
```

Casting

The cast operator (type) is used to enable conversions that would normally be disallowed.

Example:

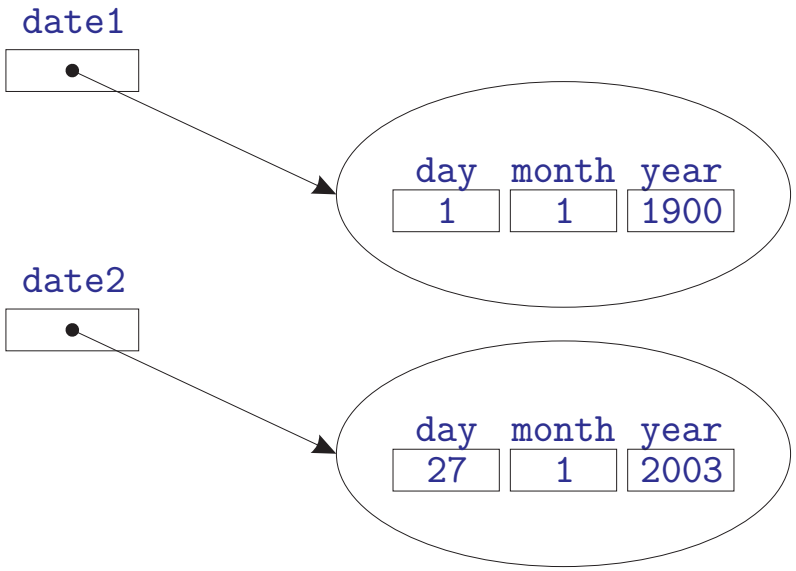
```
byte b; double d = 199.99;

b = d;          // compile error

b = (byte) d; // compiles ok
```

Later on we will learn how to cast objects from one class to another

Objects & Classes

A **class** in Java consist of *fields* that store data and *methods* that are applied to instances of the class.

An **object** in Java is an *instance* of a class.



Example

```
public class Date {

    private int day;
    private int month;
    private int year;

    public Date() {
        day = 1;
        month = 1;
        year = 1900;
    }

    public Date( int theDay, int theMonth, int theYear ) {
        day   = theDay;
        month = theMonth;
        year  = theYear;
    }
```



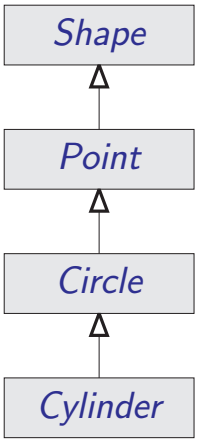
```
    public String toString( ) {
        return day + "/" + month + "/" + year;
    }

    public static void main( String [] args) {
      Date date1 = new Date();
      Date date2 = new Date( 27, 1, 2003 );
      System.out.println( "date1=" + date1 ); // 1/1/1900
      System.out.println( "date2=" + date2 ); // 27/1/2003
   }
}
```

Inheritance

In an object-oriented programming language, the fundamental mechanism for code reuse is *inheritance*.

Inheritance allows us to extend the functionality of an object.

Case Study: The Point-Circle-Cylinder hierarchy beginning with abstract super class Shape.



	getArea	getVolume	getName	toString
Shape	0.0	0.0	abstract	default Object implementation
Point	0.0	0.0	"Point"	[x,y]
Circle	πr^2	0.0	"Circle"	center=[x,y]; radius=r
Cylinder	$2\pi r^2+2\pi rh$	$\pi r^2 h$	"Cylinder"	center=[x,y]; radius=r height=h

Shape.java

```java
public abstract class Shape extends Object {

   public double getArea()
   {
      return 0.0;
   }

   public double getVolume()
   {
      return 0.0;
   }

   public abstract String getName();

}
```


Point.java

```java
public class Point extends Shape {

   private int x;
   private int y;

   public Point() { }

   public Point( int xValue, int yValue ) { x = xValue; y = yValue; }

   public void setX( int xValue ) { x = xValue; }

   public int getX() { return x; }

   public void setY( int yValue ) { y = yValue; }

   public int getY() { return y; }

   public String getName()
   {
      return "Point";
   }

   public String toString()
   {
      return "[" + getX() + ",␣" + getY() + "]";
   }
}
```

Circle.java

```java
public class Circle extends Point {
   private double radius;

   public Circle() { }

   public Circle( int x, int y, double radiusValue ) {
      super( x, y ); setRadius( radiusValue );
   }

   public void setRadius( double radiusValue ) {
      radius = ( radiusValue < 0.0?0.0:radiusValue );
   }

   public double getRadius() { return radius; }

   public double getDiameter() { return 2*getRadius(); }

   public double getCircumference() { return Math.PI * getDiameter(); }

   public double getArea() { return Math.PI * getRadius() * getRadius(); }

   public String getName() { return "Circle"; }

   public String toString() {
      return "Center = " + super.toString() + "; Radius = " + getRadius();
   }
}
```


Cylinder.java

```java
public class Cylinder extends Circle {
   private double height;

   public Cylinder() { }

   public Cylinder( int x, int y, double radius, double heightValue ) {
      super( x, y, radius ); setHeight( heightValue );
   }

   public void setHeight( double heightValue ) {
      height = ( heightValue < 0.0?0.0:heightValue );
   }

   public double getHeight() { return height; }

   public double getArea() {
      return 2 * super.getArea() + getCircumference() * getHeight();
   }

   public double getVolume() {
      return super.getArea() * getHeight();
   }

   public String getName() { return "Cylinder"; }

   public String toString() {
      return super.toString() + "; Height = " + getHeight();
   }
}
```

AbstractInheritanceTest.java

```java
import java.text.DecimalFormat;

public class AbstractInheritanceTest {
   public static void main( String args[] )
   {
      DecimalFormat twoDigits = new DecimalFormat("0.00");
      Shape a[] = {
      new Point( 7, 11 ),
      new Circle( 22, 8, 3.5 ),
      new Cylinder( 20, 30, 3.3, 10.75 ) };
      for ( int i = 0; i < a.length; i++ ) {
         System.out.println(
            a[i].getName() + ":␣" +
            a[i].toString() + "\nArea=" +
            twoDigits.format(a[i].getArea()) +
            "\nVolume␣=␣" +
            twoDigits.format(a[i].getVolume()));
         System.out.println();
      }
   }
}
```



```
$ java AbstractInheritanceTest
Point:  [7, 11]
Area=0,00
Volume = 0,00

Circle:  Center = [22, 8]; Radius = 3.5
Area=38,48
Volume = 0,00

Cylinder:  Center = [20, 30]; Radius = 3.3; Height = 10.75
Area=291,32
Volume = 367,78
```

Abstract Methods & Classes

An **abstract method** has no meaningful definition and is thus always defined in the derived class.

A class with at least one abstract method must be an **abstract class**

Abstract classes are used only as superclasses in inheritance hierarchies. These classes can not be used to instantiate objects because they are incomplete.

Subclasses must declare the "missing pieces".

Classes that can be used to instantiate objects are called **concrete classes**. Such classes provide implementation of every method they declare.

Constructors are not inherited, so they cannot be declared *abstract*

Multiple Inheritance

In **Multiple Inheritance**, a class may be derived from more than one base class. For instance, a `StudentEmployee` could then be derived from `Student` class and `Employee` class.

Java does not allow multiple inheritance!!!. However, Java provides an alternative known as the *interface*

Interfaces

The **interface** is the ultimate abstract class. It consists of public abstract methods and public static final field, only.

A class is said to **implement** the interface if it provides definitions for all of the abstract methods in the interface.

Implementing an interface is like signing a contract with the compiler that states "*I will declare all the methods specified by the interface*"

While a class may *extend* only one other class, it may *implement* more than one interface.

Shape.java

```java
public interface Shape {
   public double getArea();

   public double getVolume()

   public abstract String getName();
}
```

Point.java

```java
public class Point extends Object implements Shape {
   private int x, y;

   public Point() {}

   public Point( int xValue, int yValue ) {
      x = xValue; y = yValue;
   }

```



```java
   public void setX( int xValue ) { x = xValue; }

   public int getX() { return x; }

   public void setY( int yValue ) { y = yValue; }

   public int getY() { return y; }

   public double getArea() { return 0.0; }

   public double getVolumen() { return 0.0; }

   public String getName() { return "Point"; }

   public String toString() {
      return "[" + getX() + ",␣" + getY() + "]";
   }
}
```

InterfaceTest.java

```java
import java.text.DecimalFormat;

public class InterfaceTest {
   public static void main( String args[] ){
      DecimalFormat twoDigits = new DecimalFormat("0.00");
      Shape a[] = {
         new Point( 7, 11 ),
         new Circle( 22, 8, 3.5 ),
         new Cylinder( 20, 30, 3.3, 10.75 ) };

      for ( int i = 0; i < a.length; i++ ) {
         System.out.println(
            a[i].getName() + ":␣" +
            a[i].toString() + "\nArea=" +
            twoDigits.format(a[i].getArea()) +
            "\nVolume␣=␣" +
            twoDigits.format(a[i].getVolume()));
         System.out.println();
      }
   }
}
```



```
$ java InterfaceTest
Point:  [7, 11]
Area=0,00
Volume = 0,00

Circle:  Center = [22, 8]; Radius = 3.5
Area=38,48
Volume = 0,00

Cylinder:  Center = [20, 30]; Radius = 3.3; Height = 10.75
Area=291,32
Volume = 367,78
```

Polymorphism

A polymorphic reference type can reference objects of several different types. When methods are applied to the polymorphic type, the operation that is appropriate to the actual referenced object is automatically selected.

PolymorphismTest.java

```
1  public class PolymorphismTest {
2     public static double totalArea( Shape [] arr ) {
3        double total = 0;
4        for( int i = 0; i < arr.length; i++ )
5           if( arr[ i ] != null )
6              total += arr[ i ].getArea();
7        return total;
8     }
9
10    public static void printAll( Shape [] arr ) {
11       for( int i = 0; i < arr.length; i++ )
12          if( arr[ i ] != null )
13             System.out.println(arr[i].getName() + ":␣"
14             +arr[i].toString() + "\nArea=" +
15             arr[i].getArea() + "\nVolume␣=␣" +
```



```
16             arr[i].getVolume() + "\n");
17    }
18
19    public static void main( String args[] ) {
20       Shape [] a = { new Point( 7, 11 ),
21                      new Circle( 22, 8, 3.5 ), null,
22                      new Cylinder(20, 30, 3.3, 10.75) };
23       printAll( a );
24       System.out.println("Total␣area="+totalArea( a ));
25    }
26 }
```

```
$ java PolymorphismTest
Point:  [7, 11]
Area=0,00
Volume = 0,00

Circle:  Center = [22, 8]; Radius = 3.5
Area=38.48451000647496
Volume = 0,00

Cylinder:  Center = [20, 30]; Radius = 3.3; Height = 10.75
Area=291.3198867673815
Volume = 367.7783979741231

Total area = 329.80439677385647
```

Learning is a treasure that will follow its owner everywhere.

— Chinese Proverb

UNIVERSIDAD NACIONAL DE COLOMBIA

Estructuras de Datos

Sesión 3

Methods of Representing Data, List Data Structure (Part 1)

Yoan Pinzón, PhD
Profesor Asociado
Universidad Nacional de Colombia
http://disi.unal.edu.co/∼ypinzon/2016699/

Table of Content Session 3

Preliminaries

Data Structure: Data object along with the relationships that exist among the instances and elements, and which are provided by specifying the operations of interest.

Our *main* concern will be:

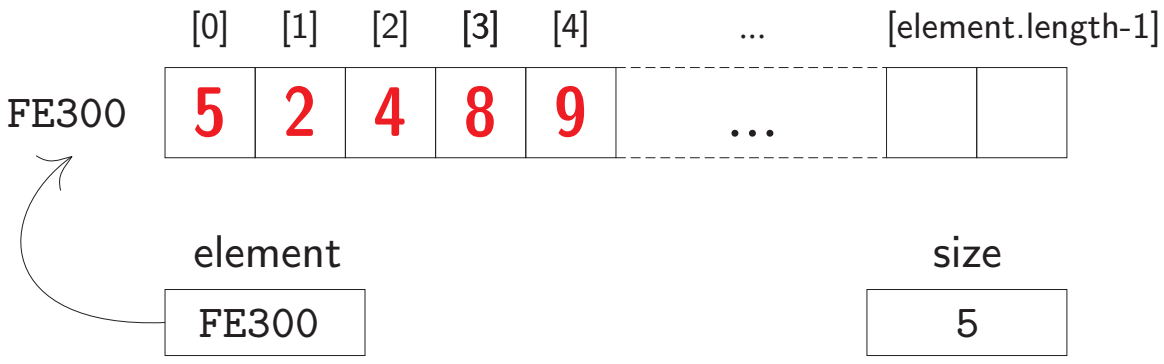
- The representation of data objects (actually of their instances)
- The representation should facilitate an *efficient* implementation of the operations

ADT - Abstract Data Type: A general way that provides a specification of the instances as well as of the operations that are to be performed.

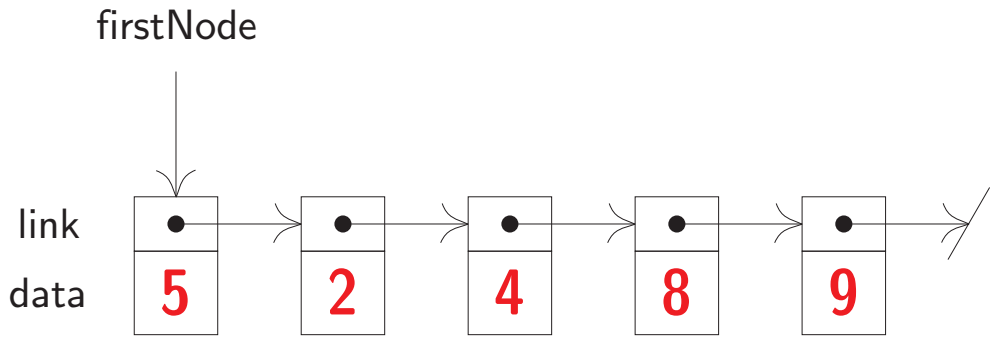
ADT representation is completely *independent* of the implementation

Methods of Representing Data

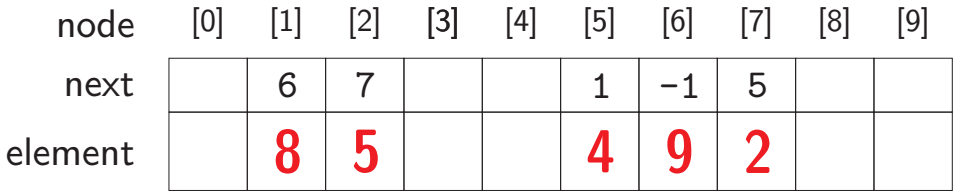
Array-based Representation: Uses an array to store either the list of elements or references to these elements.

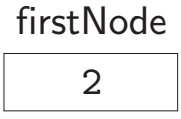


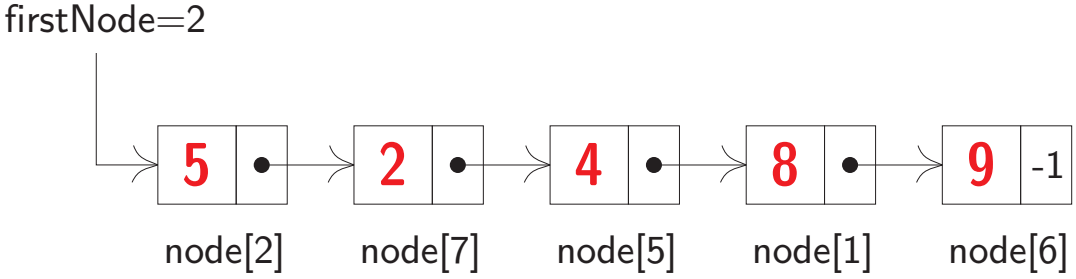
Linked Representation: The elements may be stored in any arbitrary set of memory locations. Each element keeps explicit information about the location of the next element called pointer (or link).



Simulated-pointers Representation: Similar to the linked list representation. However, pointers are replaced by integers.







Linear List Data Structure

Preliminaries

A **linear list** is a data object whose instances are of the form $(e_0, e_1, \ldots, e_{n-1})$ where n is a finite natural number. The e_i items are the elements of the list and n is the length (size).

A linear list may be specified as an abstract data type (ADT) as follows:

AbstractDataType `LinearList`

{

instances: ordered finite collection of zero or more elements

operations:

`isEmpty()`:	return `true` if the list is empty. `false` otherwise
`size()`:	return the number of elements in the list
`get(`$index$`)`:	return the $index$th element of the list
`indexOf(`x`)`:	return the position of the first occurrence of x in the list, return -1 if x is not in the list
`remove(`$index$`)`:	remove and return the $index$th element, elements with higher index have their index reduced by 1
`add(`$index$`, `x`)`:	insert x as the $index$th element, elements with index $\geq index$ have their index increased by 1
`output()`:	output the list elements from left to right

}

Interface Definition of LinearList

```
package dataStructures;

public interface LinearList
{
   public boolean isEmpty();
   public int size();
   public Object get(int index);
   public int indexOf(Object theElement);
   public Object remove(int index);
   public void add(int index, Object theElement);
   public String toString();
}
```

This implementation is programming-language dependent. We have changed the name of the output operation to `toString` because the standard output methods of Java invoke a method by this name for output.

All methods of an interface are abstract (provides an implementation for no methods), so you do not have to use the keyword `abstract` in each method's header.

Abstract Class Definition of LinearList

```
package dataStructures;

public abstract class LinearListAsAbstractClass
{
   public abstract boolean isEmpty();
   public abstract int size();
   public abstract Object get(int index);
   public abstract int indexOf(Object theElement);
   public abstract Object remove(int index);
   public abstract void add(int index, Object theObject);
   public abstract String toString();
}
```

The differences between using an *abstract class* and an *interface* are:

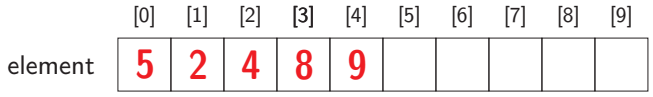
- An abstract class can define nonconstant data members and nonabstract methods.
- An interface gives us more flexibility than using an abstract class because an abstract class can implement many interfaces but can extend at most one class.

We use *interfaces* to specify ADTs throughout this course

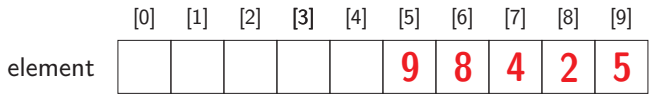
Linear List Data Structure

Array-based Representation

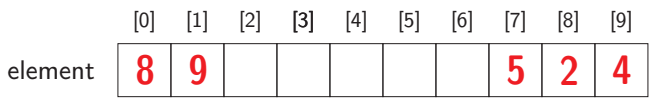
This representation uses an *array* to represent the instances of a linear list. A formula is used to map list elements into array positions. There are different ways of mapping:



(a) $location(i)$=i



(b) $location(i)$=9-i



(c) $location(i)$= (7+i)%10

We will use formula **(a)** $location(i) = i$, thus the ith element of the list (if it exists) is at position i of the array.

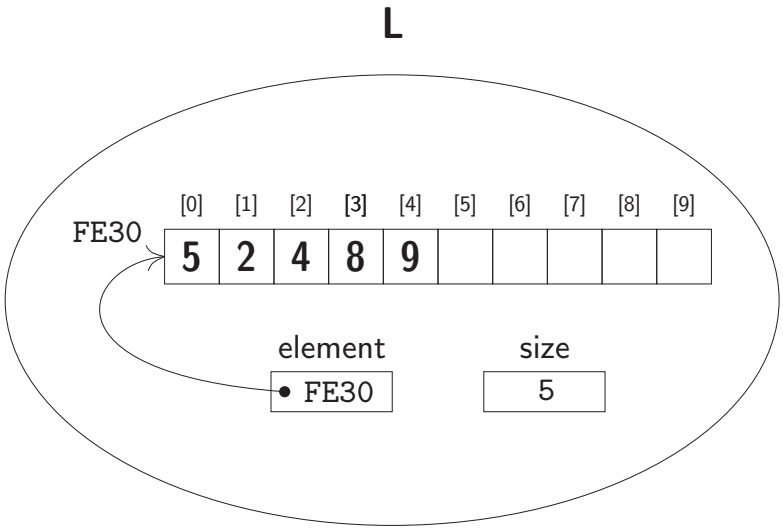
Class definition of ArrayLinearList

```
package dataStructures;

import java.util.*;
import utilities.*;

public class ArrayLinearList implements LinearList
{
   // data members
   protected Object [] element; // array of elements
   protected int size; // number of elements in array

   // constructors
   public ArrayLinearList(int initialCapacity) { ... }
   public ArrayLinearList() { ... }

   // methods
   public boolean isEmpty() { ... }
```



```
   public int size() { ... }

   void checkIndex(int index) { ... }

   public Object get(int index) { ... }

   public int indexOf(Object theElement) { ... }

   public Object remove(int index) { ... }

   public void add(int index, Object theElement) { ... }

   public String toString() { ... }

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

An instance of this class (ArrayLinearList L of integers) with `size=5` will look like this:



By declaring `element` to be of type `Object`, we can use the array `element` to hold references to elements of any user-defined data type.

The length of the array `element` is often difficult to estimate. To overcome this, we ask the user to provide an estimate and then dynamically increase the length in case the user underestimated.

Constructors

```
/** create a list with initial capacity initialCapacity
  * @throws IllegalArgumentException when
  * initialCapacity < 1 */
public ArrayLinearList(int initialCapacity)
{
   if (initialCapacity < 1)
      throw new IllegalArgumentException
            ("initialCapacity must be >= 1");

   element = new Object [initialCapacity];
}

/** create a list with initial capacity 10 */
public ArrayLinearList()
{// use default capacity of 10
   this(10);
}
```

isEmpty, size, checkIndex

```
/** @return true iff list is empty */
public boolean isEmpty()
{
   return size == 0;
}

/** @return current number of elements in list */
public int size()
{
   return size;
}

/** @throws IndexOutOfBoundsException when
  * index is not between 0 and size - 1 */
void checkIndex(int index)
{
   if (index < 0 || index >= size)
      throw new IndexOutOfBoundsException
            ("index = " + index + "  size = " + size);
}
```


get, indexOf

```
/** @return element with specified index
  * @throws IndexOutOfBoundsException when
  * index is not between 0 and size - 1 */
public Object get(int index)
{
   checkIndex(index);
   return element[index];
}

/** @return index of first occurrence of theElement,
  * return -1 if theElement not in list */
public int indexOf(Object theElement)
{
   // search element[] for theElement
   for (int i = 0; i < size; i++)
      if (element[i].equals(theElement))
         return i;

   // theElement not found
   return -1;
}
```

Deleting the $index$th element

If the list does not have an $index$th element, then throw an exception (`IndexOutOfBoundsException`). Otherwise, move elements $e_{index+1}, e_{index+2}, \ldots, e_{n-1}$ one position to the left and reduce the value of n (`size`).

```
/** Remove the element with specified index. All elements with
  * higher index have their index reduced by 1.
  * @throws IndexOutOfBoundsException when
  * index is not between 0 and size - 1
  * @return removed element */
public Object remove(int index)
{
   checkIndex(index);

   // valid index, shift elements with higher index
   Object removedElement = element[index];
   for (int i = index + 1; i < size; i++)
      element[i-1] = element[i];

   element[--size] = null;
   return removedElement;
}
```


Inserting an element

Move elements $e_{index}, e_{index+1}, \ldots, e_{n-1}$ one position to the right, insert the new element at position $index$, and increase the value of n (`size`).

```
/** All elements with equal or higher index
  * have their index increased by 1.
  * @throws IndexOutOfBoundsException when
  * index is not between 0 and size */
public void add(int index, Object theElement)
{
   if (index < 0 || index > size)
      throw new IndexOutOfBoundsException
            ("index = " + index + "  size = " + size);

   if (size == element.length) // no space, double capacity
      element = ChangeArrayLength.changeLength1D(element, 2 * size);

   // shift elements right one position
   for (int i = size - 1; i >= index; i--)
      element[i + 1] = element[i];

   element[index] = theElement;
   size++;
}
```

Why do we double the array length and not simply increase the length by 1 or 2?

Changing the Length of an Array

To increase or decrease the length of an array `a` that contains elements at positions `a[0:n-1]`, we first define an array of the new length, then copy the `n` elements from `a` to the new array, and finally change the value of `a` so that it references the new array.

The changeLength1D method

```
public static Object [] changeLength1D(Object [] a,
                            int n, int newLength)
{
   // make sure new length is adequate
   if (n > newLength)
      throw new IllegalArgumentException
                ("new␣length␣is␣too␣small");

   // allocate a new array of desired length and same type
   Object [] newArray = (Object []) Array.newInstance
           (a.getClass().getComponentType(), newLength);

   // copy from old space to new space
   System.arraycopy(a, 0, newArray, 0, n);

   return newArray;
}
```

```
// full array a is to be copied into new array
public static Object [] changeLength1D(Object [] a,
                        int newLength)
{
   return changeLength1D(a, a.length, newLength);
}
```

changeLength1D → utilities.ChangeArrayLength
newInstance, getComponentType → java.lang.Class
arraycopy → java.lang.System

Outputting the list

```
   /** convert to a string */
   public String toString()
   {
      StringBuffer s = new StringBuffer("[");

      // put elements into the buffer
      for (int i = 0; i < size; i++)
         if (element[i] == null)
            s.append("null, ");
         else
            s.append(element[i].toString() + ", ");

      if (size > 0)
         s.delete(s.length() - 2, s.length());

      s.append("]");

      // create equivalent String
      return new String(s);
   }
```

Testing ArrayLinearList Class

```
public static void main(String [] args)
{
   LinearList x = new ArrayLinearList();
   System.out.println("Initial␣size␣is␣" + x.size());
   if (x.isEmpty()) System.out.println("List␣is␣empty");
   else System.out.println("The␣list␣is␣not␣empty");
   x.add(0, new Integer(2));
   x.add(1, new Integer(6));
   x.add(0, new Integer(1));
   x.add(2, new Integer(4));
   System.out.println("List␣size␣is␣" + x.size());
   System.out.println("The␣list␣is␣" + x);
   int index = x.indexOf(new Integer(4));
   if (index < 0) System.out.println("4␣not␣found");
   else System.out.println("The␣index␣of␣4␣is␣" + index);
   index = x.indexOf(new Integer(3));
   if (index < 0) System.out.println("3␣not␣found");
```



```
   else System.out.println("The␣index␣of␣3␣is␣" + index);
   System.out.println("Element␣at␣0␣is␣" + x.get(0));
   System.out.println("Element␣at␣3␣is␣" + x.get(3));
   System.out.println(x.remove(1) + "␣removed");
   System.out.println("The␣list␣is␣" + x);
   System.out.println(x.remove(2) + "␣removed");
   System.out.println("The␣list␣is␣" + x);
   if (x.isEmpty()) System.out.println("List␣is␣empty");
   else System.out.println("List␣is␣not␣empty");
   System.out.println("List␣size␣is␣" + x.size());
}
```

```
C:\2016699\all>javac -d . dataStructures\ArrayLinearList.java
C:\2016699\all>java dataStructures.ArrayLinearList
Initial size is 0
List is empty
List size is 4
The list is [1, 2, 4, 6]
The index of 4 is 2
3 not found
Element at 0 is 1
Element at 3 is 6
2 removed
The list is [1, 4, 6]
6 removed
The list is [1, 4]
List is not empty
List size is 2
```

If your strength is small, don't carry heavy burdens. If your words are worthless, don't give advice.

— Chinese Proverb

UNIVERSIDAD NACIONAL DE COLOMBIA

Estructuras de Datos

Sesión 4

List Data Structure (Part 2)

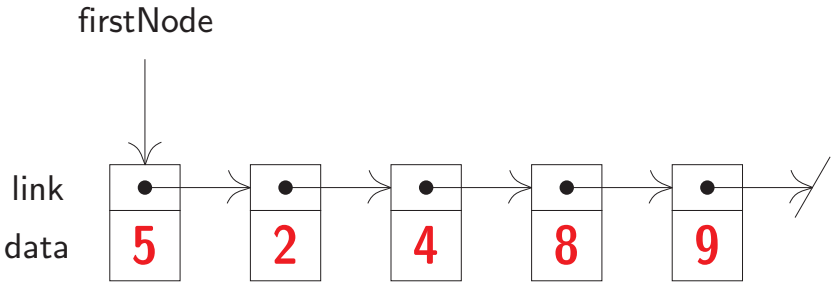
Yoan Pinzón, PhD
Profesor Asociado
Universidad Nacional de Colombia
http://disi.unal.edu.co/∼ypinzon/2016699/

Table of Content Session 4

LinearList Data Structure

Linked Representation

- Each element of an instance of a data object is represented as a group of memory locations called **cell** or **node**.
- The elements may be stored in any arbitrary set of memory locations.
- Each element keeps explicit information about the location of the next element through a **link** (or **pointer**).
- This data structure is also called **chain**



Singly linked list or chain

Class definition for a ChainNode

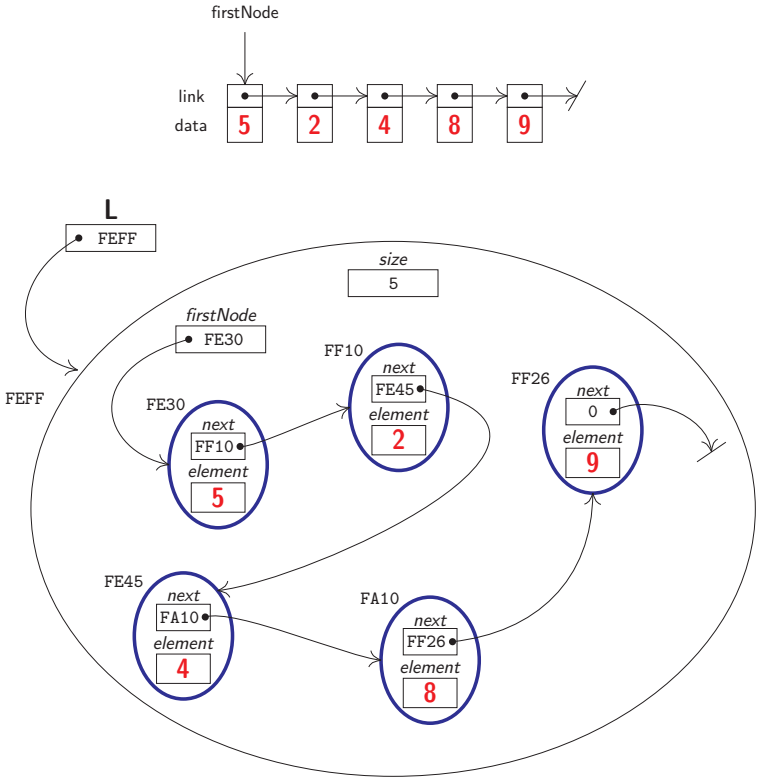
```
class ChainNode
{
   Object element;
   ChainNode next;

   // package visible constructors
   ChainNode() {}

   ChainNode(Object element)
      {this.element = element;}

   ChainNode(Object element, ChainNode next)
      {this.element = element;
       this.next = next;}
}
```

Class definition for a Chain

```
public class Chain implements LinearList
{
   // data members
   protected ChainNode firstNode;
   protected int size;

   // constructors
   public Chain(int initialCapacity) { ... }
   public Chain() { ... }

   // methods
   public boolean isEmpty() { ... }
   public int size() { ... }
   void checkIndex(int index) { ... }
   public Object get(int index) { ... }
   public int indexOf(Object theElement) { ... }
   public Object remove(int index) { ... }
   public void add(int index, Object theElement) { ... }
   public String toString() { ... }
   public static void main(String [] args) { ... }
}
```

Note: This representation does not specify the maximum size!

An instance of this class (Chain L of integers) with `size=5` will look like this:



Constructors, isEmpty, size, checkIndex

```
public Chain(int initialCapacity) { }

public Chain() { this(0); }

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

public int size()
{
   return size;
}

void checkIndex(int index)
{
   if (index < 0 || index >= size)
      throw new IndexOutOfBoundsException
            ("index␣=␣" + index + "␣␣size␣=␣" + size);
}
```


get

```
public Object get(int index)
{
   checkIndex(index);
   ChainNode currentNode = firstNode;
   for (int i = 0; i < index; i++)
      currentNode = currentNode.next;

   return currentNode.element;
}
```

indexOf

```
public int indexOf(Object theElement)
{
   ChainNode currentNode = firstNode;
   int index = 0;
   while (currentNode != null &&
         !currentNode.element.equals(theElement))
   {
      currentNode = currentNode.next;
      index++;
   }
   if (currentNode == null) return -1;
   else return index;
}
```

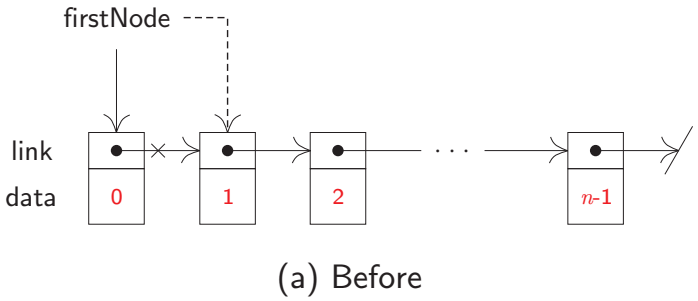

Outputting a Chain

```
/** convert to a string */
public String toString()
{
   StringBuffer s = new StringBuffer("[");

   // put elements into the buffer
   ChainNode currentNode = firstNode;
   while(currentNode != null)
   {
      if (currentNode.element == null)
         s.append("null, ");
      else
         s.append(currentNode.element.toString() + ", ");
      currentNode = currentNode.next;
   }
   // remove last ", "
   if (size > 0)
      s.delete(s.length() - 2, s.length());
   s.append("]");

   // create equivalent String
   return new String(s);
}
```

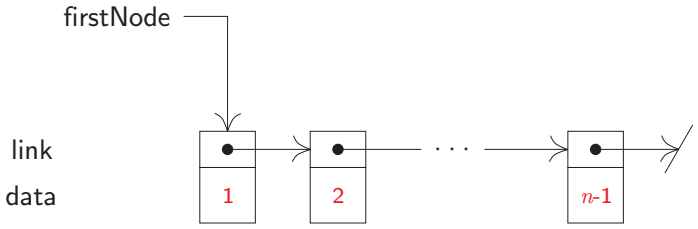
Deleting the $index$th element

There are two cases to consider:

1) If $index$=0

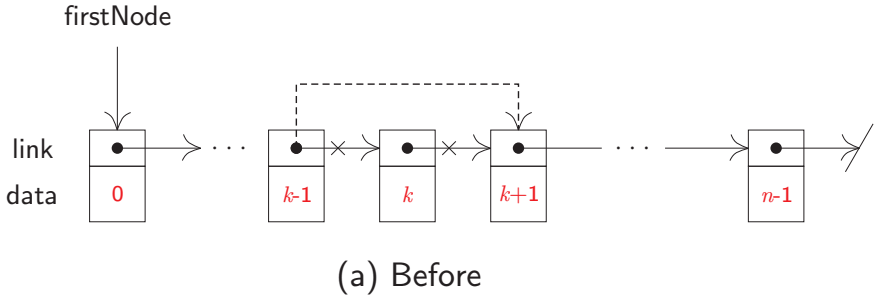


(a) Before

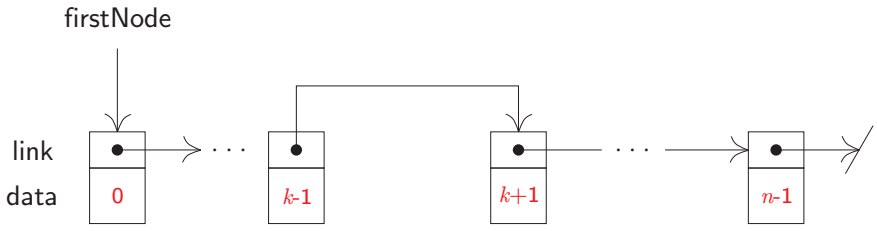


(b) After

2) If $index > 0$



(a) Before



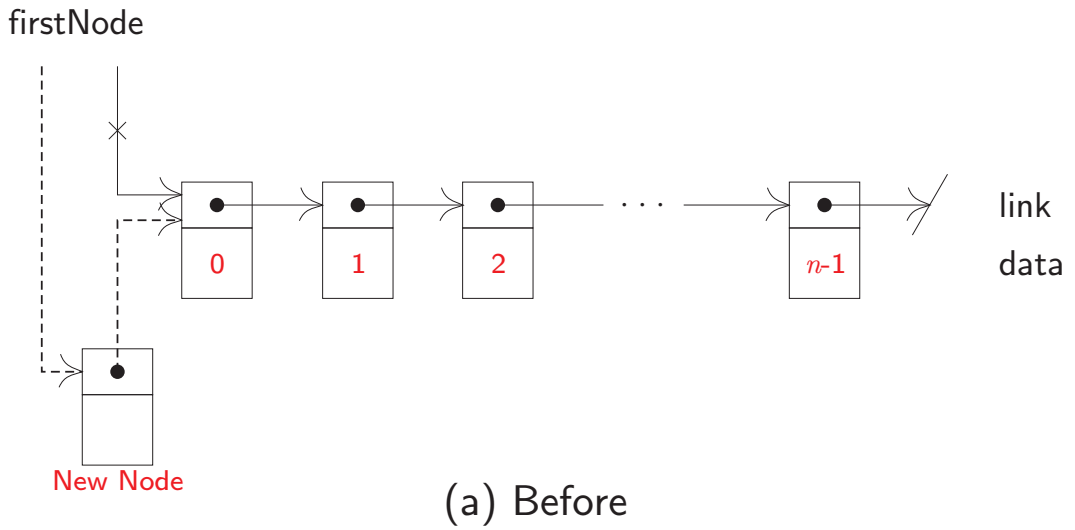
(b) After

```java
/** Remove the element with specified index.
  * All elements with higher index have their index reduced by 1.
  * @throws IndexOutOfBoundsException when
  * index is not between 0 and size - 1
  * @return removed element */
public Object remove(int index)
{
   checkIndex(index);

   Object removedElement;
   if (index == 0) // remove first node
   {
      removedElement = firstNode.element;
      firstNode = firstNode.next;
   }
   else
   {  // use q to get to predecessor of desired node
      ChainNode q = firstNode;
      for (int i = 0; i < index - 1; i++)
         q = q.next;
      removedElement = q.next.element;
      q.next = q.next.next; // remove desired node
   }
   size--;
   return removedElement;
}
```

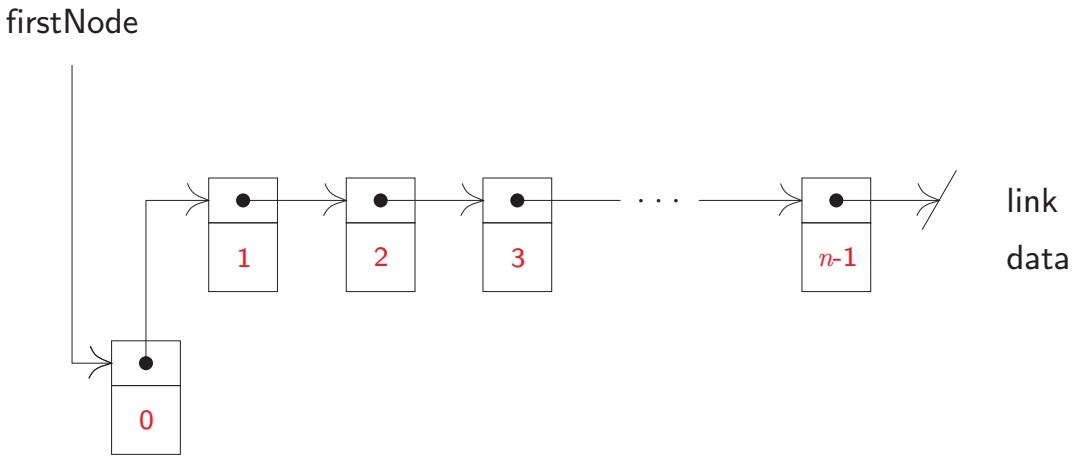

Inserting an element

Insertion and removal work in a similar way. There are two cases to consider:

1) If $index$=0

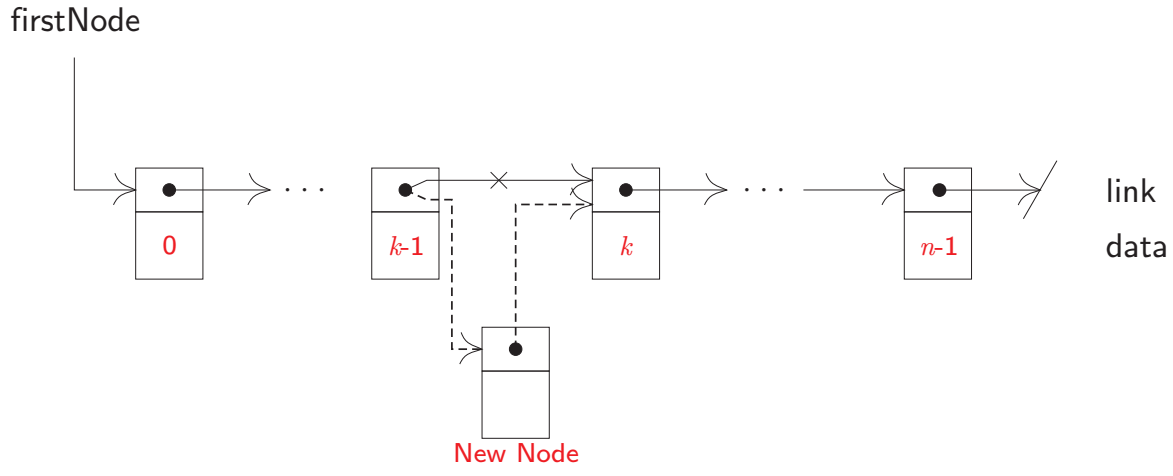


(a) Before

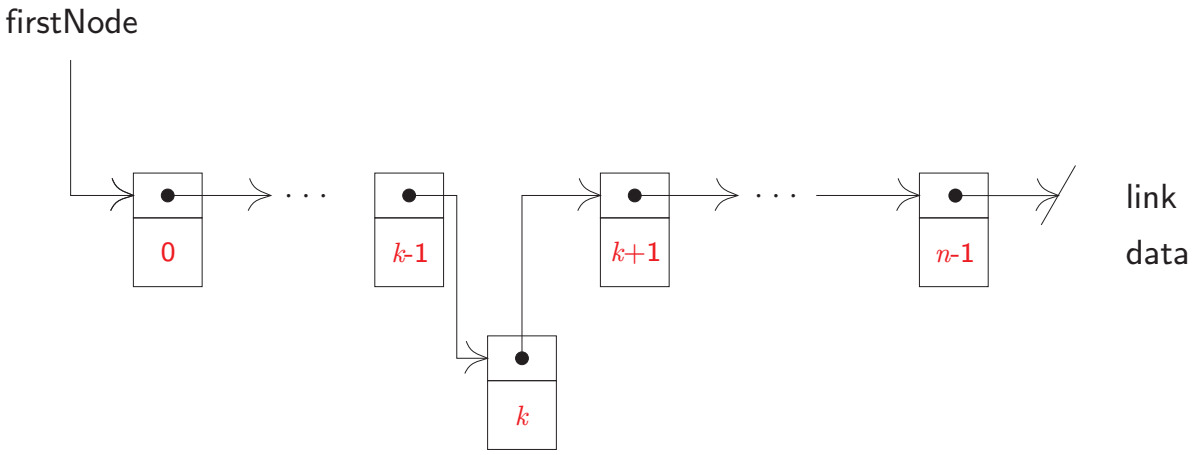


(b) After

2) If $index > 0$



(a) Before



(b) After


```
/** Insert an element with specified index.
  * All elements with equal or higher index
  * have their index increased by 1.
  * @throws IndexOutOfBoundsException when
  * index is not between 0 and size */
public void add(int index, Object theElement)
{
   if (index < 0 || index > size)
      // invalid list position
      throw new IndexOutOfBoundsException
            ("index = " + index + "  size = " + size);

   if (index == 0)
      // insert at front
      firstNode = new ChainNode(theElement, firstNode);
   else
   {  // find predecessor of new element
      ChainNode p = firstNode;
      for (int i = 0; i < index - 1; i++)
         p = p.next;
       // insert after p
      p.next = new ChainNode(theElement, p.next);
   }
   size++;
}
```

Testing Chain Class

```
public static void main(String [] args)
{
   Chain x = new Chain();
   System.out.println("Initial size is " + x.size());
   if (x.isEmpty())
      System.out.println("The list is empty");
   else System.out.println("The list is not empty");
   x.add(0, new Integer(2)); x.add(1, new Integer(6));
   x.add(0, new Integer(1)); x.add(2, new Integer(4));
   System.out.println("List size is " + x.size());
   System.out.println("The list is " + x);
   int index = x.indexOf(new Integer(4));
   if (index < 0) System.out.println("4 not found");
   else System.out.println("The index of 4 is " + index);
   index = x.indexOf(new Integer(3));
   if (index < 0) System.out.println("3 not found");
   else System.out.println("The index of 3 is " + index);
```



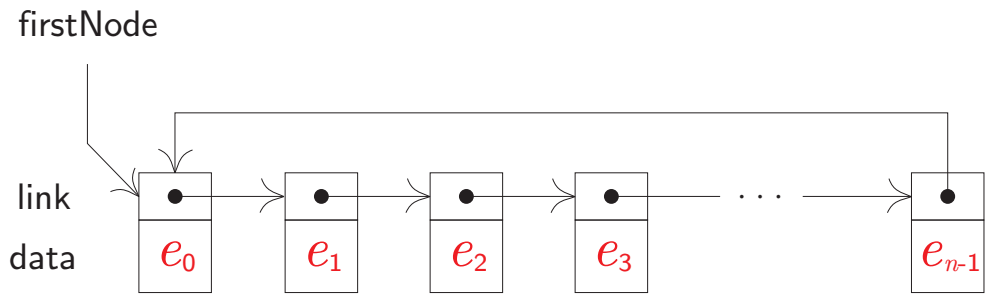
```
   System.out.println("Element at 0 is " + x.get(0));
   System.out.println("Element at 3 is " + x.get(3));
   System.out.println(x.remove(1) + " removed");
   System.out.println("The list is " + x);
   System.out.println(x.remove(2) + " removed");
   System.out.println("The list is " + x);
   if (x.isEmpty())
      System.out.println("The list is empty");
   else System.out.println("The list is not empty");
   System.out.println("List size is " + x.size());
}
```

```
C:\2016699\all>javac -d . dataStructures\Chain.java
C:\2016699\all>java dataStructures.Chain
Initial size is 0
The list is empty
List size is 4
The list is [1, 2, 4, 6]
The index of 4 is 2
3 not found
Element at 0 is 1
Element at 3 is 6
2 removed
The list is [1, 4, 6]
6 removed
The list is [1, 4]
The list is not empty
List size is 2
```

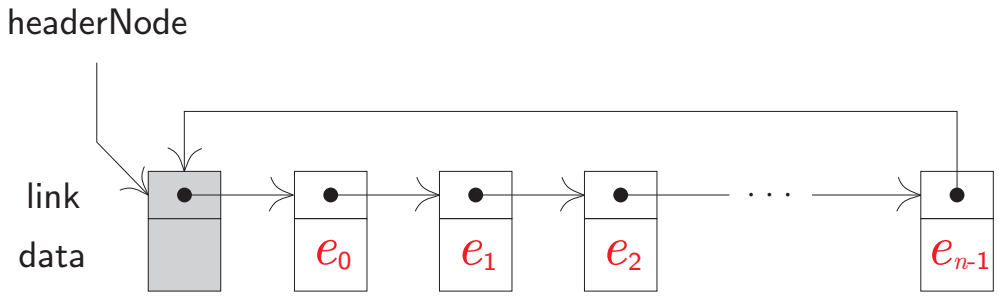

Circular Chain and Header Nodes

Special cases are always problematic in algorithm design and frequently lead to bugs. The code of a chain can be simplified (to run faster) by doing the following:

1) Represent the chain as a **circular chain**.



2) Adding an additional node, called the **header node**, at the front.



Class definition for a CircularWithHeader

```
public class CircularWithHeader implements LinearList
{
   // data members
   protected ChainNode headerNode;
   protected int size;

   // constructor
   public CircularWithHeader() { ... }

   // methods
   public boolean isEmpty() { ... }
   public int size() { ... }
   void checkIndex(int index) { ... }
   public Object get(int index) { ... }
   public int indexOf(Object theElement) { ... }
   public Object remove(int index) { ... }
   public void add(int index, Object theElement) { ... }
   public String toString() { ... }
   public static void main(String [] args) { ... }
}
```

Constructor, isEmpty, size, checkIndex

```
public CircularWithHeader()
{
   headerNode = new ChainNode();
   headerNode.next = headerNode;
}

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

public int size()
{
   return size;
}

void checkIndex(int index)
{
   if (index < 0 || index >= size)
      throw new IndexOutOfBoundsException
            ("index = " + index + "  size = " + size);
}
```


get

```
public Object get(int index)
{
   checkIndex(index);

   ChainNode currentNode = headerNode.next;
   for (int i = 0; i < index; i++)
      currentNode = currentNode.next;

   return currentNode.element;
}
```

indexOf

```
public int indexOf(Object theElement)
{
   headerNode.element = theElement;

   ChainNode currentNode = headerNode.next;
   int index = 0;
   while (!currentNode.element.equals(theElement))
   {
      currentNode = currentNode.next;
      index++;
   }

   if (currentNode == headerNode)
      return -1;
   else
      return index;
}
```


remove

```
public Object remove(int index)
{
   checkIndex(index);

   Object removedElement;
   ChainNode q = headerNode;
   for (int i = 0; i < index; i++) q = q.next;
   removedElement = q.next.element;
   q.next = q.next.next;
   size--;

   return removedElement;
}
```

add

```
public void add(int index, Object theElement)
{
   if (index < 0 || index > size)
      throw new IndexOutOfBoundsException
            ("index = " + index + "  size = " + size);

   ChainNode p = headerNode; // Fixed 9/2/4 YP
   for (int i = 0; i < index; i++) p = p.next;
   p.next = new ChainNode(theElement, p.next);
   size++;
}
```


Testing Chain Class

```
public static void main(String [] args)
{
   CircularWithHeader x = new CircularWithHeader();
   for (int i = 0; i < 10; i++)
      x.add(i, new Integer(i));
   System.out.println("List=" + x );

   for (int i = 0; i < 5; i++)
      x.remove(2);
   System.out.println("List=" + x );

   for (int i = 0; i < 10; i++)
       System.out.println(i + " is element " +
             x.indexOf(new Integer(i)));
}
```

```
List=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
List=[0, 1, 7, 8, 9]
0 is element 0
1 is element 1
2 is element -1
3 is element -1
4 is element -1
5 is element -1
6 is element -1
7 is element 2
8 is element 3
9 is element 4
```


Time Complexity Comparison of Representations

Operation	ArrayLinearList†	Chain‡
`isEmpty`	$\Theta(1)$	$\Theta(1)$
`size`	$\Theta(1)$	$\Theta(1)$
`checkIndex`	$\Theta(1)$	$\Theta(1)$
`get`	$\Theta(1)$	$O(\texttt{index})$
`indexOf`	$O(\texttt{size})$	$O(\texttt{size})$
`remove`	$O(\texttt{size-index})$	$O(\texttt{index})$
`add`	$O(\texttt{size})$	$O(\texttt{index})$
`toString`	$\Theta(\texttt{size})$	$\Theta(\texttt{size})$

†Array-based Representation
‡Linked Representation

The one who turns in his own, shall dig two graves.

— **Chinese Proverb**

UNIVERSIDAD NACIONAL DE COLOMBIA

Estructuras de Datos

Sesión 5

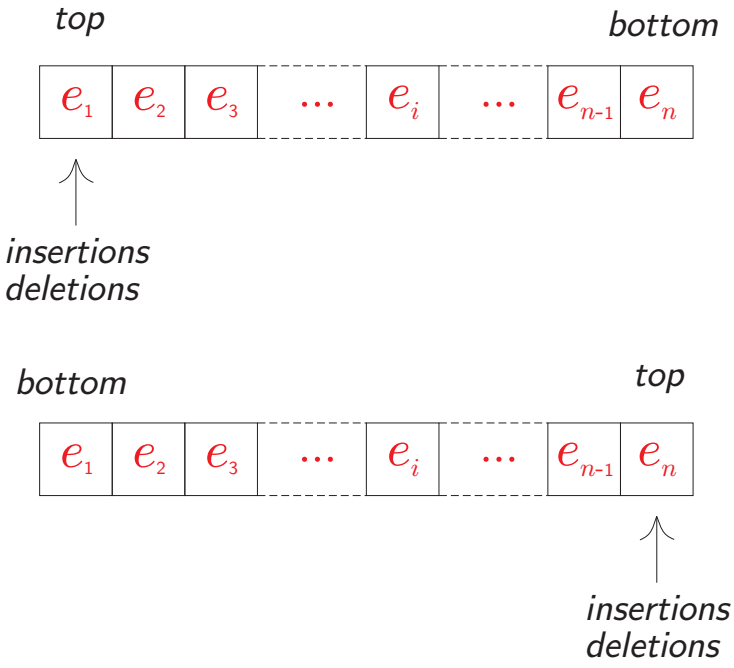
Stack Data Structure

Yoan Pinzón, PhD
Profesor Asociado
Universidad Nacional de Colombia
http://disi.unal.edu.co/∼ypinzon/2016699/

Table of Content Session 5

Stacks Data Structure

A **stack** is a linear list in which insertions (also called additions) and deletions take place at the *same* end. This end is called the **top**. The other end is called the **bottom**.



In other words, a stack is a LIFO (last-in-first out) list. Lists of this type appear frequently in computing.

The Abstract Data Type

AbstractDataType Stack

{

instances: linear list of elements; one end called the *bottom*; the other is the *top*;

operations:

`empty()`: return `true` if the stack is empty. `false` otherwise.

`peek()`: return top element of the stack

`push(`x`)`: add element x at the top of the stack

`pop()`: remove the top element from the stack and return it

}

Interface Definition of Stack

```
package dataStructures;

public interface Stack
{
   public boolean empty();
   public Object peek();
   public void push(Object theObject);
   public Object pop();
}
```


Observations

- Stack is a specialized or restricted version of a more general data object linear list.
- Every instance of the data object stack is also an instance of the data object linear list. Moreover, all the stack operations can be performed as linear list operations.
- As a result of these observations, we will defined the stack class as a class which inherit all the data member and function from the linear list class.
- We will also use two methods of representation. namely, array-based and linked representation.

Stack Data Structure

Array-based Representation

- Implementation using inheritance
- Customised Implementation

Implementing Stack using inheritance

Basic design decision:

designate the left end of the list as the bottom and the right end as the top

Class definition of DerivedArrayStack

```
public class DerivedArrayStack extends ArrayLinearList implements Stack
{
   // constructors
   public DerivedArrayStack(int initialCapacity)
      {super(initialCapacity);}

   public DerivedArrayStack()
      {this(10);}

   // methods
   public boolean empty() {return isEmpty();}

   public Object peek()
   {
      if (empty()) throw new EmptyStackException();
      return get(size() - 1);
   }

   public void push(Object theElement)
      {add(size(), theElement);}

   public Object pop()
   {
      if (empty()) throw new EmptyStackException();
      return remove(size() - 1);
   }
}
```


Test program

```
   public static void main(String [] args)
   {
      DerivedArrayStack s = new DerivedArrayStack(3);
      // add a few elements
      s.push(new Integer(1));
      s.push(new Integer(2));
      s.push(new Integer(3));
      s.push(new Integer(4));


      // delete all elements
      while (!s.empty())
      {
         System.out.println("Top␣element␣is␣" + s.peek());
         System.out.println("Removed␣the␣element␣" + s.pop());
      }
   }
```

```
Top element is 4
Removed the element 4
Top element is 3
Removed the element 3
Top element is 2
Removed the element 2
Top element is 1
Removed the element 1
```


Time Complexity of Operations

First Constructor	:	$O(\texttt{initialCapacity})$
Second Constructor	:	$\Theta(1)$
Other operations*	:	$\Theta(1)$

* The complexity of `push` is $\Theta(1)$ except when the addition of an element requires us to increase the capacity of the stack. In this latter case the complexity is $O(\texttt{capacity})$.

Implementing Stack as a base class

```
public class ArrayStack implements Stack
{
   // data members
   int top;        // current top of stack
   Object [] stack; // element array

   // constructors
   public ArrayStack(int initialCapacity) { ... }
   public ArrayStack() { ... }

   // methods
   public boolean empty() { ... }
   public Object peek() { ... }
   public void push(Object theElement) { ... }
   public Object pop() { ... }
   public static void main(String [] args) { ... }
}
```


Constructors

```
   public ArrayStack(int initialCapacity)
   {
      if (initialCapacity < 1)
         throw new IllegalArgumentException
               ("initialCapacity must be >= 1");
      stack = new Object [initialCapacity];
      top = -1;
   }

   public ArrayStack()
   {
      this(10);
   }
```

empty, peek

```
public boolean empty()
{
   return top == -1;
}

public Object peek()
{
   if (empty())
      throw new EmptyStackException();
   return stack[top];
}
```


push, pop

```
public void push(Object theElement)
{
   // increase array size if necessary
   if (top == stack.length - 1)
      stack = ChangeArrayLength.changeLength1D(stack,
              2 * stack.length);

   // put theElement at the top of the stack
   stack[++top] = theElement;
}

public Object pop()
{
   if (empty())
      throw new EmptyStackException();
   Object topElement = stack[top];
   stack[top--] = null; // enable garbage collection
   return topElement;
}
```

Stack Data Structure

Linked Representation

- Implementation using inheritance
- Customised Implementation

In both cases, we have to decide which end of the chain will be the top of the stack and which the bottom.

Using Inheritance

```
public class DerivedLinkedStack extends Chain implements Stack
{
   public DerivedLinkedStack(int initialCapacity)
      {super();}

   public DerivedLinkedStack()
      {this(0);}

   public boolean empty() {return isEmpty();}

   public Object peek()
   {
      if (empty()) throw new EmptyStackException();
      return get(0);
   }

   public void push(Object theElement)
   {
      add(0, theElement);
   }

   public Object pop()
   {
      if (empty()) throw new EmptyStackException();
      return remove(0);
   }
}
```

Implementing Stack as a base class

```
public class LinkedStack implements Stack
{
   // data members
   protected ChainNode topNode;

   // constructors
   public LinkedStack(int initialCapacity) { ... }
   public LinkedStack() { ... }

   // methods
   public boolean empty() { ... }
   public Object peek() { ... }
   public void push(Object theElement) { ... }
   public Object pop() { ... }
}
```


Constructors, empty, peek

```
   public LinkedStack(int initialCapacity)
   {
       // the default initial value of topNode is null
   }

   public LinkedStack()
   {
      this(0);
   }

   public boolean empty()
   {
      return topNode == null;
   }

   public Object peek()
   {
      if (empty())
         throw new EmptyStackException();
      return topNode.element;
   }
```

push, pop

```
public void push(Object theElement)
{
   topNode = new ChainNode(theElement, topNode);
}

public Object pop()
{
   if (empty())
      throw new EmptyStackException();
   Object topElement = topNode.element;
   topNode = topNode.next;
   return topElement;
}
```


Time Complexity of Operations

Constructors	:	$\Theta(1)$
Other operations	:	$\Theta(1)$

Stack Application

Parentheses Matching

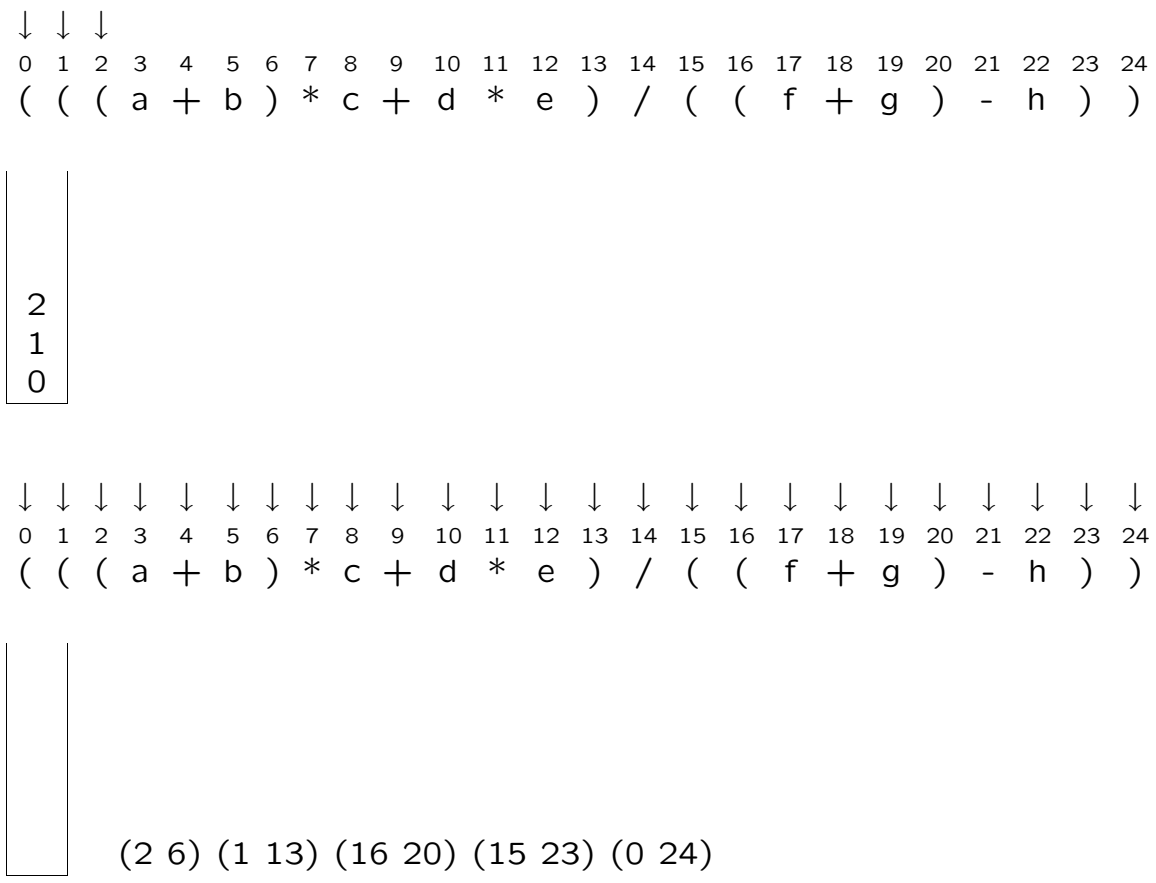
How do we match parentheses in an expression?

```
(((a+b)*c+d*e)/((f+g)-h+i))
```

```
(a*(a+b))/(b+d)
```


Parentheses Matching

- scan expression from left to right
- when a left parenthesis is encountered, add its position to the stack
- when a right parenthesis is encountered, remove matching position from stack




```
public static void printMatchedPairs(String expr)
{
   ArrayStack s = new ArrayStack();
   int length = expr.length();

   // scan expression expr for ( and )
   for (int i = 0; i < length; i++)
      if (expr.charAt(i) == '(')
         s.push(new Integer(i));
      else
         if (expr.charAt(i) == ')')
            try
            {
               System.out.println(s.pop() + "␣␣" + i);
            }
            catch (Exception e)
            {  // stack was empty, no match exists
               System.out.println( "No␣match␣for␣right␣parenthesis"
               + "␣at␣" + i);
            }

   // remaining '(' in stack are unmatched
   while (!s.empty())
      System.out.println("No␣match␣for␣left␣parenthesis␣at␣" + s.pop());
}
```

```
public class ParenthesisMatching
{
   public static void printMatchedPairs(String expr)
      { ... }

   /** test program */
   public static void main(String [] args)
   {
      MyInputStream keyboard = new MyInputStream();

      System.out.println("Type an expression");
      String expression = keyboard.readString();

      System.out.println("The matching parentheses in");
      System.out.println(expression);
      System.out.println("are (indexing begins at 0)");
      printMatchedPairs(expression);
   }
}
```

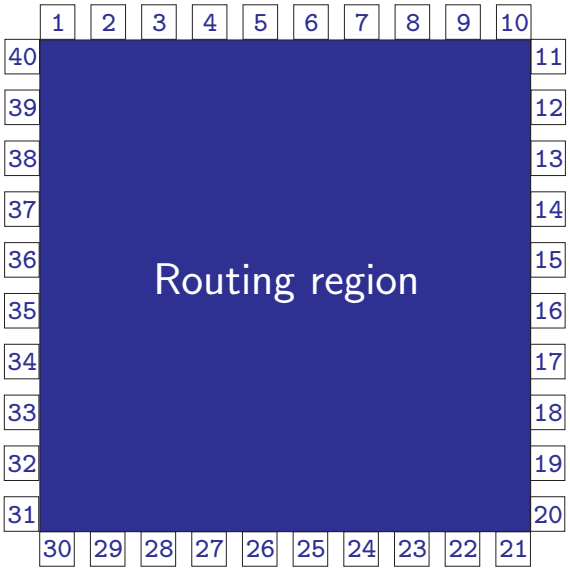


```
C:\2016699\all>java applications.ParenthesisMatching
Type an expression with no spaces
(((a+b)*c+d*e)/((f+g)-h))
The pairs of matching parentheses in
(((a+b)*c+d*e)/((f+g)-h))
are (indexing begins at 0)
2 6
1 13
16 20
15 23
0 24
```

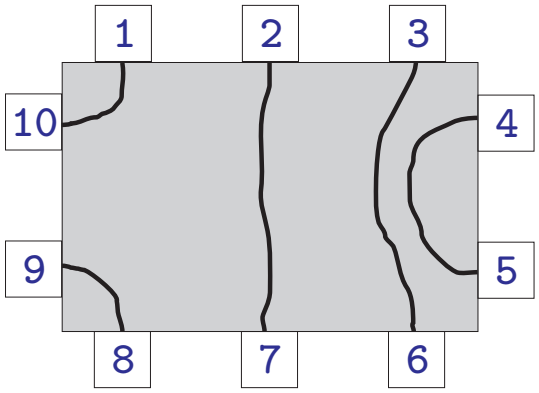
Stack Application

Switchbox Routing

The switchbox routing problem arises in the fabrication of computer chips, where certain components need to be connected to other components.

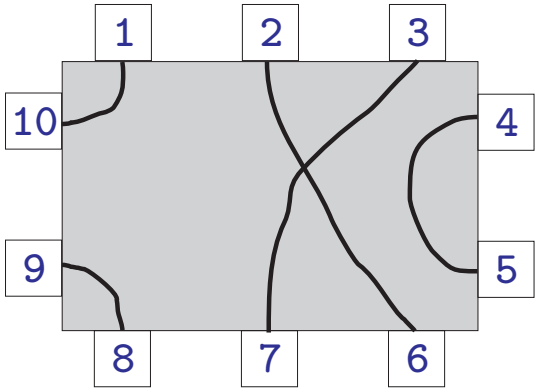


	1	2	3	4	5	6	7	8	9	10	
Net={	1,	2,	3,	4,	4,	3,	2,	5,	5,	1	}



Routable!

	1	2	3	4	5	6	7	8	9	10	
Net={	1,	2,	3,	4,	4,	2,	3,	5,	5,	1	}



Not Routable!


```
public static boolean checkBox(int [] net)
{
   int n = net.length; // number of pins
   ArrayStack s = new ArrayStack();

   // scan nets clockwise
   for(int i = 0; i < n; i++)
      // process pin i
      if(!s.empty())
         // check with top net
         if(net[i]==net[((Integer) s.peek()).intValue()])
            s.pop(); // net[i] is routable, delete from stack
         else s.push(new Integer(i));
      else s.push(new Integer(i));

   // any unrouted nets left?
   if(s.empty())
   {// no nets remain
      System.out.println("Switch box is routable");
      return true;
   }

   System.out.println("Switch box is not routable");
   return false;
}
```

```
public class SwitchBox
{
   public static boolean checkBox(int [] net) { ... }

   /** test program */
   public static void main(String [] args)
   {
      MyInputStream keyboard = new MyInputStream();

      System.out.println("Type␣number␣of␣pins");
      int n = keyboard.readInteger();

      int [] net = new int[n];

     System.out.print("Type␣net␣numbers␣")
     System.out.println("for␣pins␣1␣through␣" + n);
     for (int i = 0; i < n; i++)
        net[i] = keyboard.readInteger();

     checkBox(net);
   }
}
```



```
C:\2016699\all>java applications.SwitchBox
Type number of pins
20
Type net numbers for pins 1 through 20
1 2 3 4 4 5 6 6 7 8 9 9 10 10 8 7 5 3 2 1
Switch box is routable
```

He who cannot agree with his enemies is controlled by them.

— **Chinese Proverb**

UNIVERSIDAD NACIONAL DE COLOMBIA

Estructuras de Datos

Sesión 6

Queue Data Structure

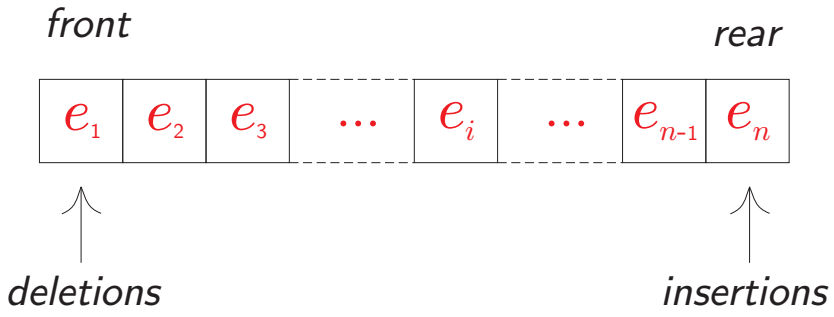
Yoan Pinzón, PhD
Profesor Asociado
Universidad Nacional de Colombia
http://disi.unal.edu.co/∼ypinzon/2016699/

Table of Content Session 6

Queue Data Structure

A **queue** is a special case of linear list where insertions and deletions take place at *different* ends

rear: end at which a new element is added.
front: end at which an element is deleted.



In other words, a queue is a FIFO (first-in-first-out) list.

The Abstract Data Type

AbstractDataType Queue
{
instances: linear list of elements; one end is called the the *front*; the other is the *rear*;

operations:

`isEmpty()`: return `true` if the queue is empty. `false` otherwise.

`getFrontElement()`: return the front element of the queue

`getRearElement()`: return the rear element of the queue

`put(`x`)`: add element x at the rear of the queue

`remove()`: remove an element from the front of the queue and return it

}

Interface Definition of Queue

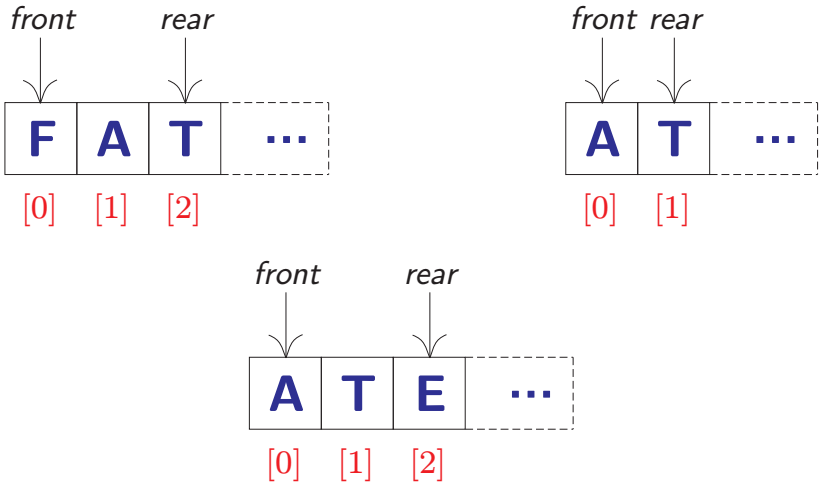
```
package dataStructures;

public interface Queue
{
   public boolean isEmpty();
   public Object getFrontElement();
   public Object getRearElement();
   public void put(Object theObject);
   public Object remove();
}
```


Queue Data Structure

Array-based Representation

We can use three different approaches:

1) Using the formula $\boxed{location(i) = i}$



- **Empty queue:** rear = -1
- **Addition:**

  rear = rear + 1

  queue[rear] = new_element

  $\Theta(1)$ time

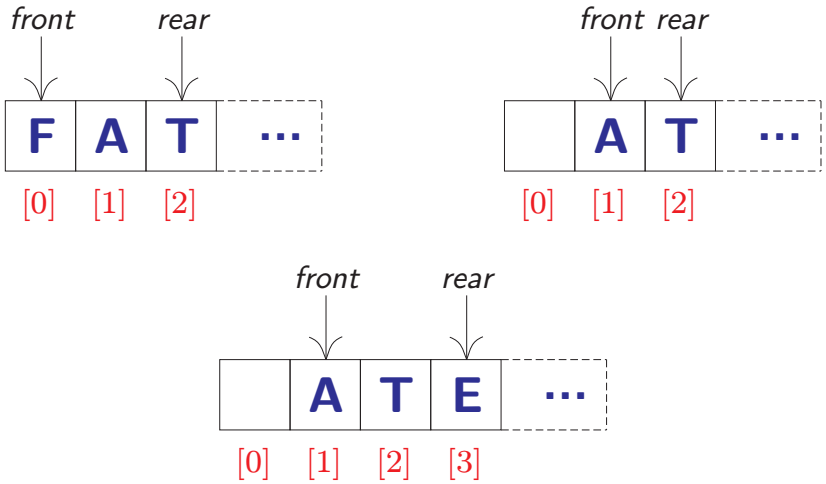
- **Deletion:**

  Shift all elements one position to the left.

  $\Theta(n)$ time

2) Using the formula

$$location(i) = location(\text{front element}) + i$$



- **Empty queue:** rear $<$ front

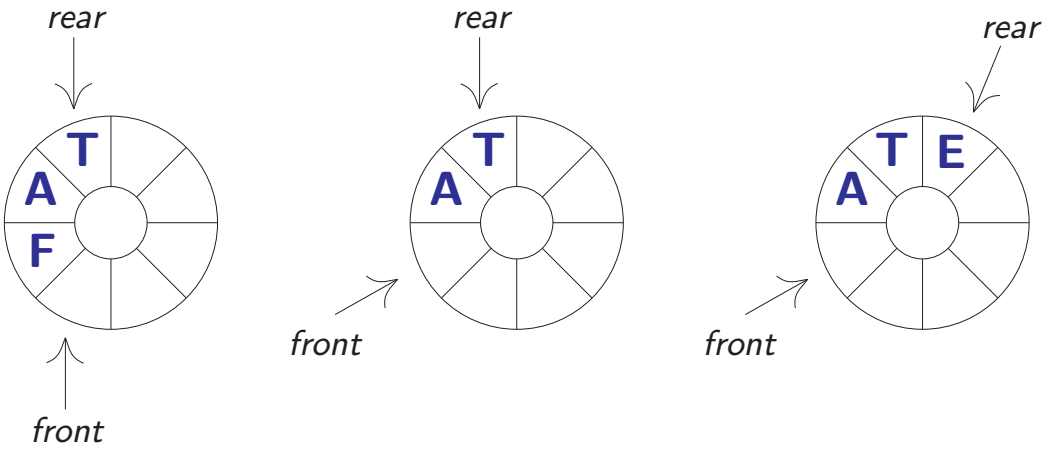
- front $= location$(front element)
- rear $= location$(last element)

- **Deletions & Insertions:** $\Theta(1)$ time

What happens if rear = queue.length and front > 0 ?

3) Using the formula:

$$location(i) = (location(\text{front element}) + i)\%queue.length$$



front points one position before the position of the first element in the queue.

- **Empty queue:** front = rear
(initially front=rear=0)
- **Full queue:** (rear+1)%queue.length = front

This is also called a **Circular Queue**

Class Definition

```
public class ArrayQueue implements Queue
{
   // data members
   int front; // one counterclockwise from first element
   int rear;  // position of rear element of queue
   Object [] queue; // element array

   // constructors
   public ArrayQueue(int initialCapacity) { ... }
   public ArrayQueue() { ... }

   // methods
   public boolean isEmpty() { ... }
   public Object getFrontElement() { ... }
   public Object getRearElement() { ... }
   public void put(Object theElement) { ... }
   public Object remove() { ... }
   public static void main(String [] args){ ... }
}
```

Constructors, isEmpty

```
public ArrayQueue(int initialCapacity)
{
   if (initialCapacity < 1)
      throw new IllegalArgumentException
            ("initialCapacity must be >= 1");
   queue = new Object [initialCapacity + 1];
   // default front = rear = 0
}

public ArrayQueue()
{  // use default capacity of 10
   this(10);
}

public boolean isEmpty()
{
   return front == rear;
}
```


getFrontElement, getRearElement

```
public Object getFrontElement()
{
   if (isEmpty())
      return null;
   else
      return queue[(front + 1) % queue.length];
}

public Object getRearElement()
{
   if (isEmpty())
      return null;
   else
      return queue[rear];
}
```

remove

```
public Object remove()
{
   if (isEmpty())
      return null;
   front = (front + 1) % queue.length;
   Object frontElement = queue[front];
   queue[front] = null; // enable garbage collection
   return frontElement;
}
```


put

```
public void put(Object theElement)
{
   // increase array length if necessary
   if ((rear + 1) % queue.length == front)
   {// double array size
      // allocate a new array
      Object [] newQueue = new Object [2 * queue.length];
      // copy elements into new array
      int start = (front + 1) % queue.length;
      if (start < 2)
         // no wrap around
         System.arraycopy(queue, start, newQueue, 0, queue.length - 1);
      else
      {  // queue wraps around
         System.arraycopy(queue, start, newQueue, 0, queue.length - start);
         System.arraycopy(queue, 0, newQueue, queue.length - start, rear + 1);
      }
      // switch to newQueue and set front and rear
      front = newQueue.length - 1;
      rear = queue.length - 2;
      queue = newQueue;
   }
   // put theElement at the rear of the queue
   rear = (rear + 1) % queue.length;
   queue[rear] = theElement;
}
```

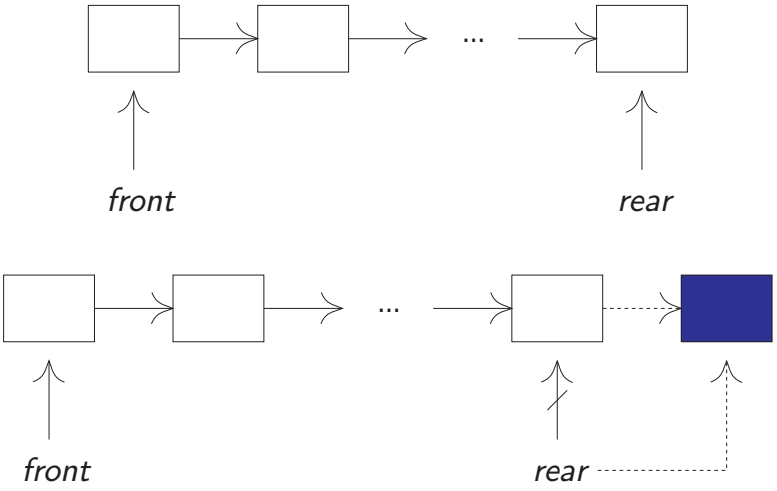
```
public static void main(String [] args)
{
   ArrayQueue q = new ArrayQueue(3);
   q.put(new Integer(1));
   q.put(new Integer(2));
   q.put(new Integer(3));
   q.put(new Integer(4));

   while (!q.isEmpty())
   {
      System.out.println("Rear element is "
                         + q.getRearElement());
      System.out.println("Front element is "
                         + q.getFrontElement());
      System.out.println("Removed the element "
                         + q.remove());
   }
}
```



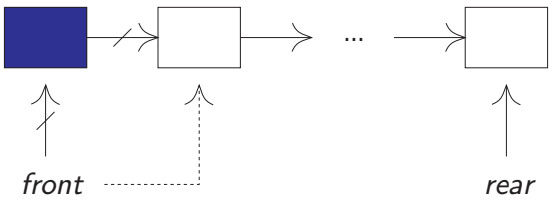
```
Rear element is 4

Front element is 1

Removed the element 1

Rear element is 4

Front element is 2

Removed the element 2

Rear element is 4

Front element is 3

Removed the element 3

Rear element is 4

Front element is 4

Removed the element 4
```

Queue Data Structure

Linked Representation



(a) Addition



(b) Deletion

- **Empty queue:** front = null
- **Deletions & Insertions:** $\Theta(1)$ time

Class Definition

```
public class LinkedQueue implements Queue
{
   // data members
   protected ChainNode front;
   protected ChainNode rear;

   // constructors
   public LinkedQueue(int initialCapacity) {}
   public LinkedQueue() {this(0);}

   // methods
   public boolean isEmpty() { return front == null; }

   public Object getFrontElement()
   {
      if (isEmpty()) return null;
      else return front.element;
   }
```



```
   public Object getRearElement()
   {
      if (isEmpty()) return null;
      else return rear.element;
   }

   public void put(Object theElement) { ... }
   public Object remove() { ... }
   public static void main(String [] args) { ... }
}
```

put

```
public void put(Object theElement)
{
   // create a node for theElement
   ChainNode p = new ChainNode(theElement, null);

   // append p to the chain
   if (front == null)
      front = p;               // empty queue
   else
      rear.next = p;           // nonempty queue
   rear = p;
}
```


remove

```
public Object remove()
{
   if (isEmpty())
      return null;
   Object frontElement = front.element;
   front = front.next;
   if (isEmpty())
      rear = null; // enable garbage collection
   return frontElement;
}
```

Queue Application

Image-Component Labeling

		1				
		1	1			
				1		
			1	1		
	1			1		1
1	1	1				1
1	1	1			1	1

(a) Input

		2				
		2	2			
				3		
			3	3		
	4			3		5
4	4	4				5
4	4	4			5	5

(b) Output

- Digitised image: $m \times m$ matrix of pixels (0,1).
  0-pixel represents image background;
  1-pixel represents a point on an image component.
- Two pixels are adjacent if one is to the left, above, right, or below the other.

- Two 1-pixels (component pixels) that are adjacent belong to the same image component.

Objective: Label the component pixels such that two pixels get the same label if and only if they are pixels of the same image component.

```
import dataStructures.*;
import utilities.*;

public class ImageComponents
{
   // top-level nested class
   private static class Position { ... }

   // data members
   private static int [][] pixel;
   private static int size;

   // methods
   private static void inputImage() { ... }
   private static void labelComponents() { ... }
   private static void outputImage() { ... }
   public static void main(String [] args)
   {
      inputImage();
      labelComponents();
      outputImage();
   }
}
```



```
private static class Position
{
   // data members
   private int row; // row number of the position
   private int col; // column number of the position

   // constructor
   private Position(int theRow, int theCol)
   {
      row = theRow;
      col = theCol;
   }

   // convert to string suitable for output
   public String toString()
   {
      return new String(row + "␣" + col);
   }
}
```

```
private static void inputImage()
{
   // define the input stream to be the standard
   // input stream
   MyInputStream keyboard = new MyInputStream();

   System.out.println("Enter␣image␣size");
   size = keyboard.readInteger();

   // create and input the pixel array
   pixel = new int [size + 2][size + 2];
   System.out.print("Enter␣the␣pixel␣array␣");
   System.out.println("in␣row-major␣order");
   for (int i = 1; i <= size; i++)
      for (int j = 1; j <= size; j++)
           pixel[i][j] = keyboard.readInteger();
}
```



```
private static void outputImage()
{
   System.out.println("The␣labeled␣image␣is");
   for (int i = 1; i <= size; i++)
   {
      for (int j = 1; j <= size; j++)
         System.out.print(pixel[i][j] + "␣␣");
      System.out.println();
   }
}
```

```
private static void labelComponents()
{
   // initialize offsets
   Position [] offset = new Position [4];
   offset[0] = new Position(0, 1); // right
   offset[1] = new Position(1, 0); // down
   offset[2] = new Position(0, -1); // left
   offset[3] = new Position(-1, 0); // up

   // initialize wall of 0 pixels
   for (int i = 0; i <= size + 1; i++)
   {
      pixel[0][i] = pixel[size + 1][i] = 0; // bottom and top
      pixel[i][0] = pixel[i][size + 1] = 0; // left and right
   }

   int numOfNbrs = 4; // neighbors of a pixel position
   ArrayQueue q = new ArrayQueue();
   Position nbr = new Position(0, 0);
   int id = 1; // component id
```



```
   // scan all pixels labeling components
   for (int r = 1; r <= size; r++)
      for (int c = 1; c <= size; c++)
         if (pixel[r][c] == 1)
         {  // new component
            pixel[r][c] = ++id; // get next id
            Position here = new Position(r, c);

            do
            {  // find rest of component
               for (int i = 0; i < numOfNbrs; i++)
               {  // check all neighbors of here
                  nbr.row = here.row + offset[i].row;
                  nbr.col = here.col + offset[i].col;
                  if (pixel[nbr.row][nbr.col] == 1)
                  {  // pixel is part of current component
                     pixel[nbr.row][nbr.col] = id;
                     q.put(new Position(nbr.row, nbr.col));
                  }
               }
               // any unexplored pixels in component?
               here = (Position) q.remove();
            } while(here != null);
         } // end of if, for c, and for r
}
```

File ImageComponents.input:

```
7
0 0 1 0 0 0 0
0 0 1 1 0 0 0
0 0 0 0 1 0 0
0 0 0 1 1 0 0
1 0 0 0 1 0 0
1 1 1 0 0 0 0
1 1 1 0 0 0 0
```

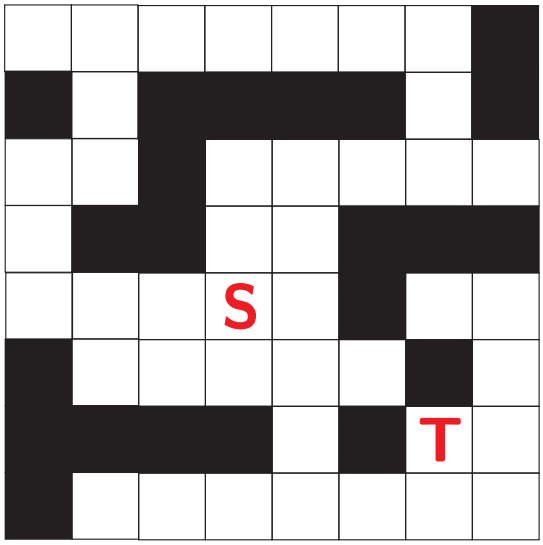
File ImageComponents.output:

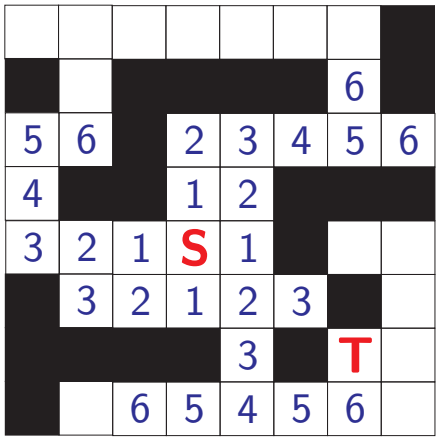
```
Enter image size
Enter the pixel array in row-major order
The labeled image is
0  0  2  0  0  0  0
0  0  2  2  0  0  0
0  0  0  0  3  0  0
0  0  0  3  3  0  0
4  0  0  0  3  0  0
4  4  4  0  0  0  0
4  4  4  0  0  0  0
```


Queue Application

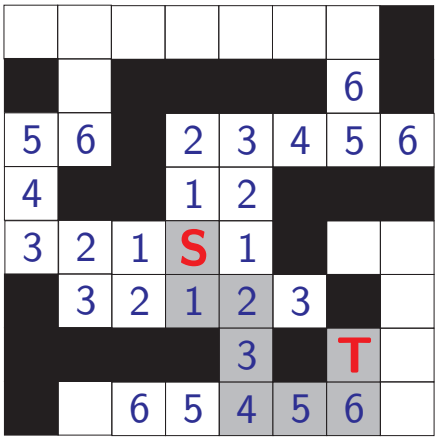
Lee's Wire Router

Find a path from **S** to **T** by *wave propagation*.





(a) Filing



(b) Retrace


```
import dataStructures.*;
import utilities.*;

public class WireRouter
{
   // top-level nested class
   private static class Position { ... }

   // data members
   private static int [][] grid;
   private static int size;
   private static int pathLength;
   private static ArrayQueue q;
   private static Position start, finish;
   private static Position [] path; // the shortest path

   // methods
   private static void inputData() { ... }
   private static boolean findPath() { ... }
   private static void outputPath() { ... }
```

```
   public static void main(String [] args)
   {
      inputData();
      if ( findPath() ) outputPath();
      else System.out.println("There is no wire path");
   }
}
```



```
private static class Position
{
   // data members
   private int row; // row number of the position
   private int col; // column number of the position

   // constructor
   private Position(int theRow, int theCol)
   {
      row = theRow;
      col = theCol;
   }

   // convert to string suitable for output
   public String toString()
   {
      return new String(row + " " + col);
   }
}
```

```
private static void inputData()
{
   MyInputStream keyboard = new MyInputStream();

   System.out.println("Enter grid size");
   size = keyboard.readInteger();

   System.out.println("Enter the start position");
   start = new Position(keyboard.readInteger(),
      keyboard.readInteger());

   System.out.println("Enter the finish position");
   finish = new Position(keyboard.readInteger(),
      keyboard.readInteger());

   // create and input the wiring grid array
   grid = new int [size + 2][size + 2];
   System.out.print("Enter the wiring grid ");
   System.out.println("in row-major order");
   for (int i = 1; i <= size; i++)
      for (int j = 1; j <= size; j++)
           grid[i][j] = keyboard.readInteger();
}
```



```
private static void outputPath()
{
   System.out.println("The wire path is");
   for (int i = 0; i < pathLength; i++)
      System.out.println(path[i]);
}
```

```
private static boolean findPath()
{
   if ((start.row == finish.row) && (start.col == finish.col))
   {  // start == finish
      pathLength = 0;
      return true;
   }
   // initialize offsets
   Position [] offset = new Position [4];
   offset[0] = new Position(0, 1); // right
   offset[1] = new Position(1, 0); // down
   offset[2] = new Position(0, -1); // left
   offset[3] = new Position(-1, 0); // up
   // initialize wall of blocks around the grid
   for (int i = 0; i <= size + 1; i++)
   {
      grid[0][i] = grid[size + 1][i] = 1;
      grid[i][0] = grid[i][size + 1] = 1;
   }
   Position here = new Position(start.row, start.col);
   grid[start.row][start.col] = 2; // block
   int numOfNbrs = 4; // neighbors of a grid position
   ArrayQueue q = new ArrayQueue();
   Position nbr = new Position(0, 0);
```



```
   do
   {// label neighbors of here
         for (int i = 0; i < numOfNbrs; i++)
         {// check out neighbors of here
            nbr.row = here.row + offset[i].row;
            nbr.col = here.col + offset[i].col;
            if (grid[nbr.row][nbr.col] == 0)
            {  // unlabeled nbr, label it
               grid[nbr.row][nbr.col]
                  = grid[here.row][here.col] + 1;
               if ((nbr.row == finish.row) &&
                  (nbr.col == finish.col)) break; // done
               // put on queue for later expansion
               q.put(new Position(nbr.row, nbr.col));
            }
         }
      // have we reached finish?
      if ((nbr.row == finish.row) &&
            (nbr.col == finish.col)) break;  // done
      if (q.isEmpty()) return false;      // no path
      here = (Position) q.remove(); // get next position
   } while(true);
```

```
   // construct path
   pathLength = grid[finish.row][finish.col] - 2;
   path = new Position [pathLength];

   // trace backwards from finish
   here = finish;
   for (int j = pathLength - 1; j >= 0; j--)
   {
      path[j] = here;
      // find predecessor position
      for (int i = 0; i < numOfNbrs; i++)
      {
         nbr.row = here.row + offset[i].row;
         nbr.col = here.col + offset[i].col;
         if (grid[nbr.row][nbr.col] == j + 2) break;
      }
      here = new Position(nbr.row, nbr.col);
   }

   return true;
}
```


File WireRouter.input:

```
7
3 2
4 6
0 0 1 0 0 0 0
0 0 1 1 0 0 0
0 0 0 0 1 0 0
0 0 0 1 1 0 0
1 0 0 0 1 0 0
1 1 1 0 0 0 0
1 1 1 0 0 0 0
```

File WireRouter.output:

```
Enter grid size
Enter the start position
Enter the finish position
Enter the wiring grid in row-major order
The wire path is
4 2
5 2
5 3
5 4
6 4
6 5
6 6
5 6
4 6
```

A wise man makes his own decisions, an ignorant man follows the public opinion.

— Chinese Proverb

UNIVERSIDAD NACIONAL DE COLOMBIA

Estructuras de Datos

Sesión 7

Analysis of Algorithms & Sorting

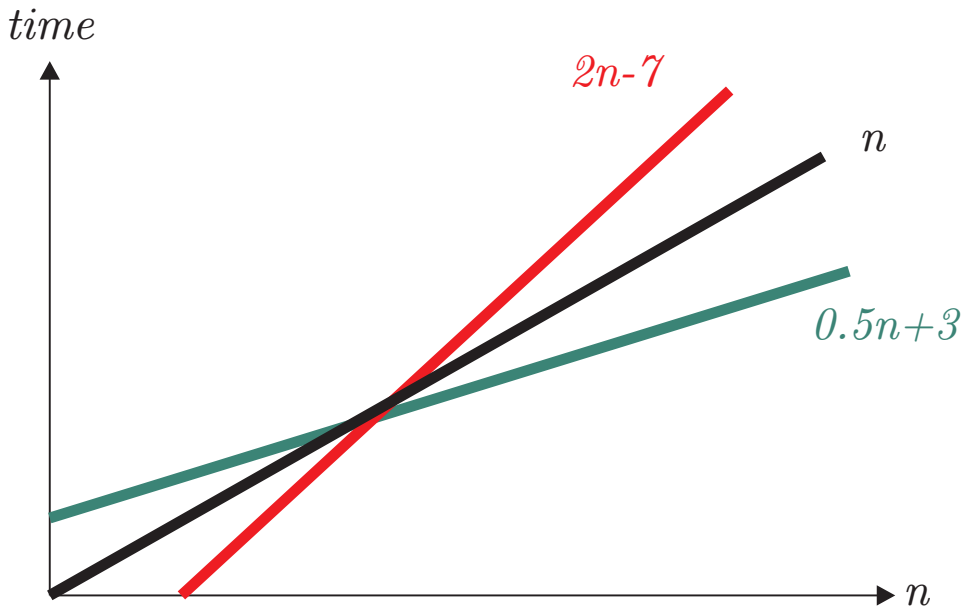
Yoan Pinzón, PhD
Profesor Asociado
Universidad Nacional de Colombia
http://disi.unal.edu.co/∼ypinzon/2016699/

Table of Content Session 7

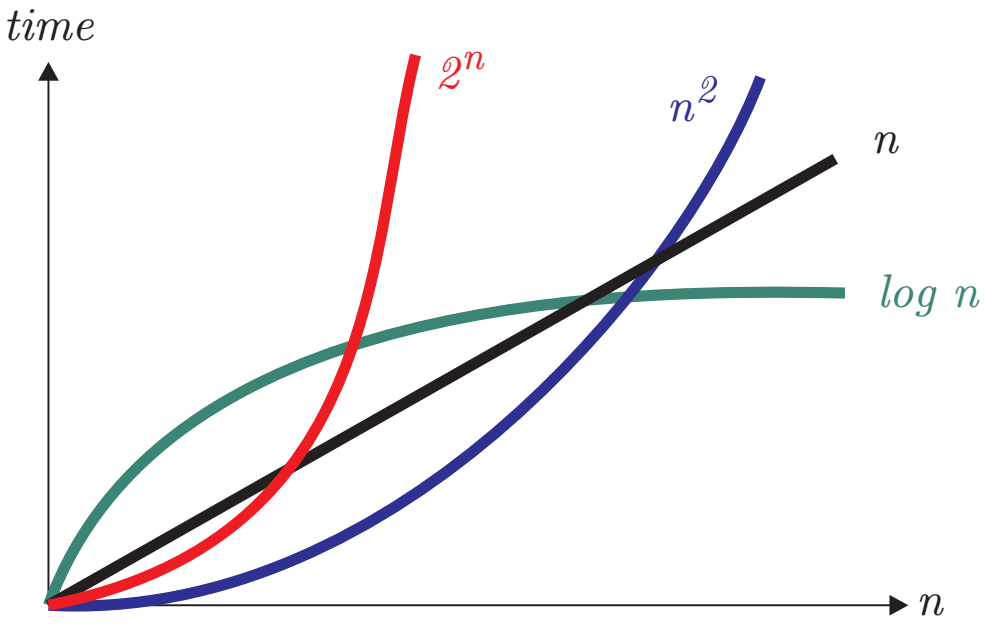
Growth of Functions

- **Asymptotic efficiency of algorithms**
  - How does the running time of an algorithm increase with the size of the input in the limit as the input increases in size without bound?

- **Asymptotic notation (the order of)**
  - Define sets of functions that satisfy certain criteria and use these to characterize time and space complexity of algorithms



(a) Growth of Functions (Linear)



(b) Growth Rates

Growth Rates

Let $f(n)$ and $g(n)$ be functions

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \leftarrow \begin{cases} 0, f(n) \text{ grows slower than } g(n) \\ \infty, f(n) \text{ grows faster than } g(n) \\ \text{otherwise } f(n) \text{ and } g(n) \\ \text{have the same growth rate} \end{cases}$$

$$f(n) \prec g(n)$$

$$\lim_{n\to\infty} f(n)/g(n) = 0$$

$f(n) \prec g(n)$: $f(n)$ grows slower than $g(n)$

$0.5^n \prec 1 \prec \log n \prec \log^6 n \prec n^{0.5} \prec n^3 \prec 2^n \prec n!$

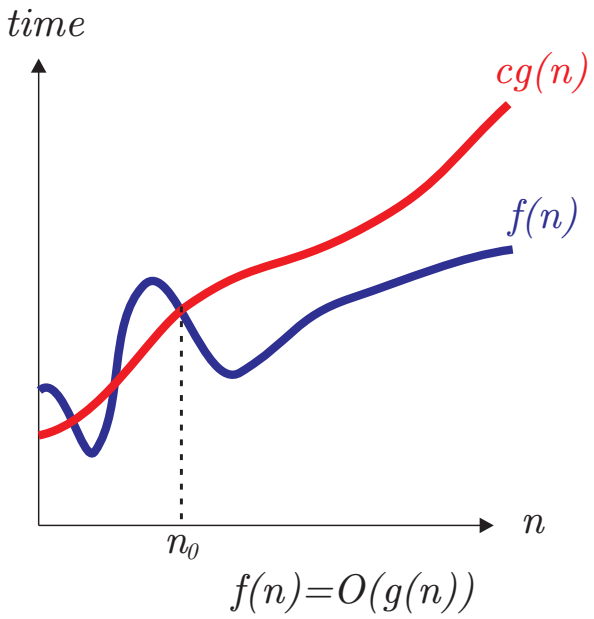
Asymptotic Notations

- Helps in simplifying the analysis of running time
- Deal with the behavior of functions in the limit (for sufficiently large value of its parameters)
- Permit substantial simplification (napkin mathematic, rough order of magnitude). Simple Rule: Drop lower order terms and constant factors
  - $7n - 3$ is $O(n)$
  - $8n^2 \log n + 5n^2 + n$ is $O(n^2 \log n)$
- Classify functions by their growth rates

The Big-Oh Notation

$f(n) = O(g(n))$ if there exist positive constants c and n_0 such that for every $n \geq n_0$,

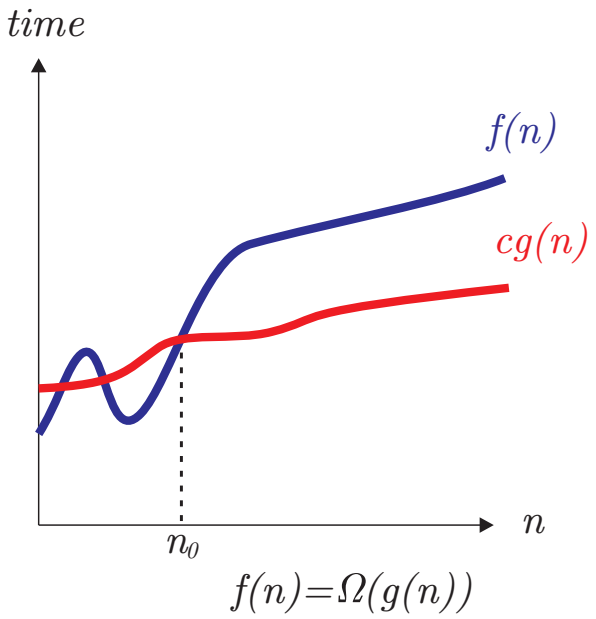
$$f(n) \leq cg(n)$$



The Ω Notation

$f(n) = \Omega(g(n))$ if there exist positive constants c and n_0 such that for every $n \geq n_0$,

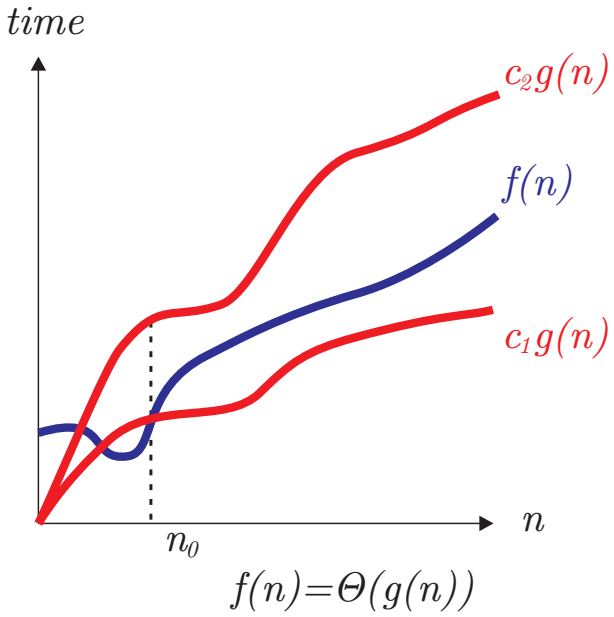
$$f(n) \geq cg(n)$$



The Θ Notation

$f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(n)$, or equivalently, if there are positive constants c_1, c_2 and n_0 such that for every $n \geq n_0$,

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$



Danger of Asymptotic Notation

- We typically try to get algorithms with the best running time.
- Then we might prefer an algorithm requiring $1,000,000n$ operations to an algorithm requiring $1.000n \log n$ operations for inputs of length n.
- Even though the former algorithm is better for all but finitely many instances, the latter is better for all the instances that can exist in the known universe.

Special Orders of Growth

•	constant	$O(1)$
•	logarithmic	$O(\log n)$
•	polylogarithmic	$O(\log^c n), c \geq 1$
•	n log n	$O(n \log n)$
•	sublinear	$O(n^a), 0 < a < 1$
•	linear	$O(n)$
•	quadratic	$O(n^2)$
•	polynomial	$O(n^c), c \geq 1$
•	exponential	$O(c^n), c > 1$
•	factorial	$O(n!)$

Sorting Algorithms

- Sorting is a fundamental application for computers
- Sorting is perhaps the most intensively studied and important operation in computer science.
- An initial sort of the data can significantly enhance the performance of an algorithm.

Examples

- Words in a dictionary
- Files in a directory
- The index of a book
- The card catalog in a library
- List of students taking the course

Preliminaries

- Java provides a comparison method that is imparted to a class by implementing the **java.lang.Comparable** interface.
- The **Comparable** interface is quite simple, it only has a single method called **compareTo( )**.
- The **compareTo()** method takes another Object as an argument, and produces a negative value, zero, or a positive value if the current object is less than, equal to, or greater than the argument, respectively.
- Each algorithm describe here is passed an array containing the elements, and only objects that implement the **Comparable** interface can be sorted.

Implementing a Comparable Interface

```
public abstract class Shape implements Comparable
{
   public abstract double area();
   public abstract double perimeter();

   public int compareTo( Object rhs )
   {
      Shape other = (Shape) rhs;
      double diff = area() - other.area();
     if( diff == 0 )
        return 0;
      else if( diff < 0 )
        return -1;
      else
        return 1;
   }

   public double semiperimeter ()
   {
      return perimeter() / 2;
   }
}
```


Selection Sort

Let `a=[0:`$n-1$`]` and n`=a.length`. In *selection sort* we find the largest item and put it in `a[`$n-1$`]`, then determine the largest of the remaining $n-1$ elements and move it to `a[`$n-2$`]`, and so on.

▶ **Example:** Sort the array `a[0:5]=[6,5,8,4,3,1]`


```
public class SelectionSort
{
   public static void selectionSort(Comparable [] a)
   {
      for (int size = a.length; size > 1; size--)
      {
         int j = MyMath.max(a, size-1);
         MyMath.swap(a, j, size - 1);
      }
   }

   public static void main(String [] args)
   {
      Integer [] a = {new Integer(6), new Integer(5),
                      new Integer(8), new Integer(4),
                      new Integer(3), new Integer(1)};
      System.out.println("The␣elements␣are");
      for (int i = 0; i < a.length; i++)
         System.out.print(a[i] + "␣");
      selectionSort(a);
      System.out.println("\nThe␣sorted␣order␣is");
      for (int i = 0; i < a.length; i++)
         System.out.print(a[i] + "␣");
   }
}
```

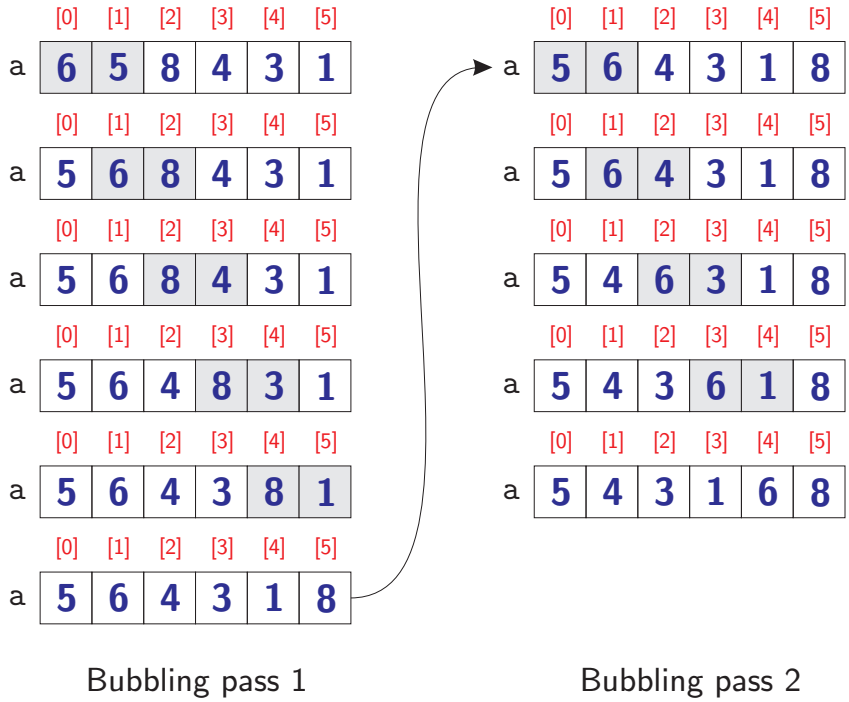
```
The elements are

6 5 8 4 3 1

The sorted order is

1 3 4 5 6 8
```

Time complexity: $O(n^2)$

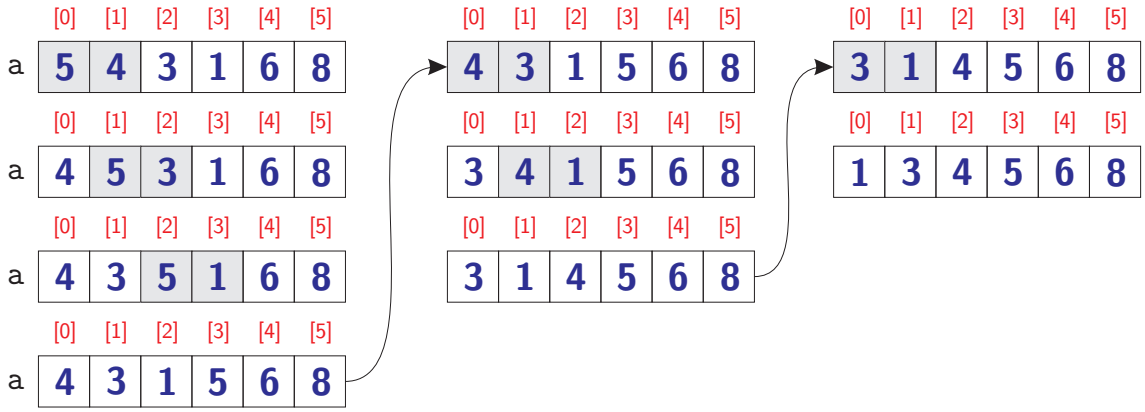
Bubble Sort

In *bubble sort* (or the sinking sort) we make several *bubbling passes* through the array. Each pass compares adjacent elements. The elements are swaped in the case the one on the left is greater than the one on the right. At the end of the bubbling pass, we are assured that the largest element is in the right-most position.

► **Example:** Sort the array `a[0:5]=[6,5,8,4,3,1]`



Bubbling pass 1 Bubbling pass 2



Bubbling pass 3 Bubbling pass 4 Bubbling pass 5

It is like air bubbles rising in water, while the larger values sink to the bottom of the array

```
public class BubbleSort
{
   /** bubble largest element in a[0:n-1] to right */
   private static void bubble(Comparable [] a, int n)
   {
      for (int i = 0; i < n - 1; i++)
         if (a[i].compareTo(a[i+1]) > 0)
            MyMath.swap(a, i, i + 1);
   }

   /** sort the array a using the bubble sort method */
   public static void bubbleSort(Comparable [] a)
   {
      for (int i = a.length; i > 1; i--)
         bubble(a, i);
   }

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

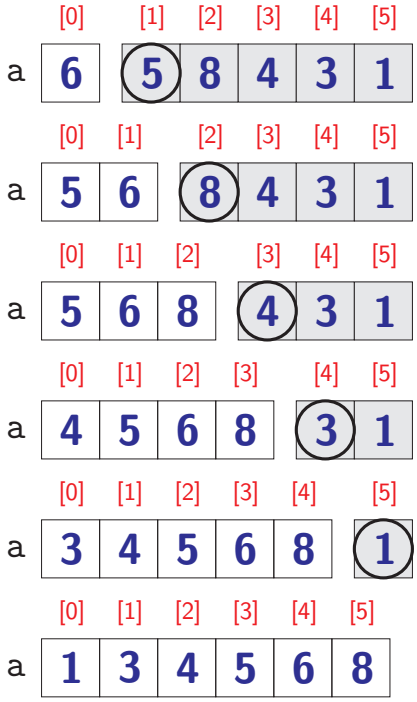
Time complexity: $O(n^2)$

Insertion Sort

In *insertion sort* we traverse the list of items. For each item, we insert it into its correct position amongst the items that we have already sorted.

We start with an array that contains just the first element, then we insert the second element into the one-element array, we get a sorted array of size 2. The insertion of the third element yields a sorted array of size 3, and so on.

► **Example:** Sort the array `a[0:5]=[6,5,8,4,3,1]`




```
public class InsertionSort
{
   public static void insertionSort(Comparable [] a)
   {
      for (int i = 1; i < a.length; i++)
      {  // insert a[i] into a[0:i-1]
         Comparable t = a[i];

         // find proper place for t
         int j;
         for (j = i - 1;
              j >= 0 && t.compareTo(a[j]) < 0; j--)
            a[j+1] = a[j];

         a[j+1] = t; // insert t = original a[i]
      }
   }

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

Time complexity: $O(n^2)$

Merge Sort

The basic idea for merge sorting a list, such as (5,3,8,6,7,2,4,1), is a three-step process:

1) *divide* the list in two halves, L = (5,3,8,6) and R = (7,2,4,1);

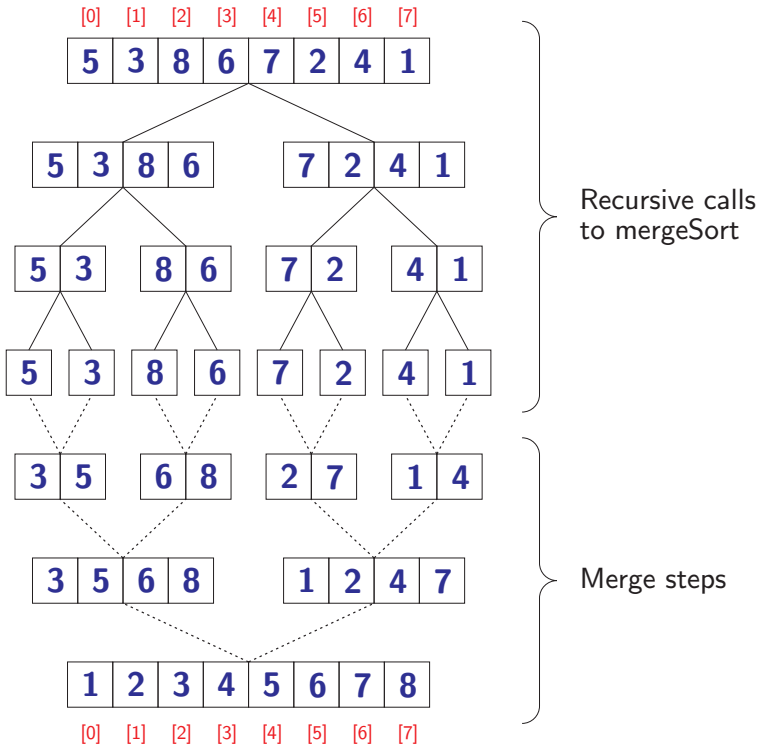
2) *sort* the two half-lists, getting L = (3,5,6,8) and R = (1,2,4,7); and

3) *merge* the two half-lists into the final sorted list Merge(L,R)=(1,2,3,4,5,6,7,8).

Although the merge step of mergesort produces a sorted array, how do you sort the array halves prior to the merge step? Mergesort sorts the array halves by using mergesort – that is, by calling itself recursively.


```
public static void mergeSort(Comparable [] a, int left, int right)
{
   if (left < right)
   {
      int middle = (left + right) / 2;
      mergeSort(a, left, middle);
      mergeSort(a, middle + 1, right);
      merge(a, b, left, middle, right); // merge a to b
      copy(b, a, left, right); // copy result back to a
   }
}
```

▶ **Example:** Sort the array `a[0:7]=[5,3,8,6,7,2,4,1]`

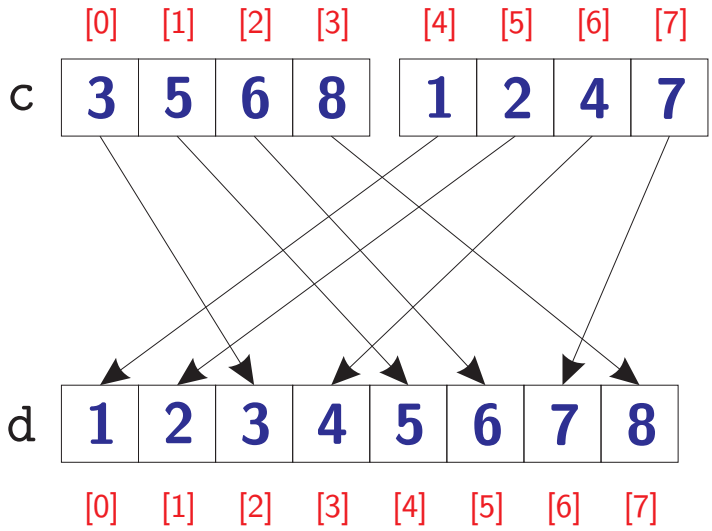


How to merge two sorted half-lists, L and R, into a single sorted result list, SL?

We can start with an empty list, SL, and we repeatedly remove the smaller first item from the beginning of L or R and put it at the end of SL. When both L and R are empty, SL will be a merged list sorted in ascending order.

This merging process involves moving all n items into SL, and may involve comparing as few as $n/2$ of them or as many as $(n-1)$ of them. Thus merging is a linear time process known to require $O(n)$ running time.

`startOfFirst=0, endOfFirst=3, endOfSecond=7`

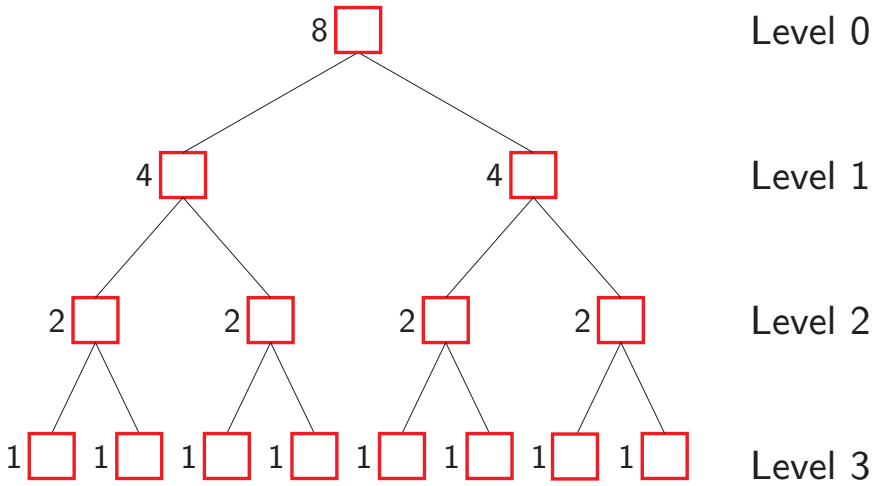


Complexity Analysis

Each mergeSort recursively calls itself twice. If the original call to mergeSort is at level 0, two calls to mergeSort occur at level 1 of the recursion. Each of these calls then calls mergeSort twice, so four calls to mergeSort occur at level 2 of the recursion, and so on.

The recursive calls continue until the array pieces each contain one item. Then the recursion goes $k = \log n$ levels deep.

For example: For an array of eight items the levels of recursion are four.



Each level of recursion requires $O(n)$ time, then mergeSort is $O(nk) = O(n \log n)$ time.

We can improve the performance by eliminating the recursion. An iterative version of merge sort begins by merging pairs of adjacent segments of size 1, then it merges pairs of adjacent segments of size 2, and so on.


```
public class MergeSort
{
   /** sort the elements a[0 : a.length - 1] using
     * the merge sort method */
   public static void mergeSort(Comparable [] a)
      { ... }

   /** merge adjacent segments from x to y */
   public static void mergePass(Comparable [] x,
              Comparable [] y, int segmentSize)
      { ... }

   /** merge two adjacent segments from c to d */
   public static void merge(Comparable [] c,
              Comparable [] d, int startOfFirst,
              int endOfFirst, int endOfSecond)
      { ... }

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

We can eliminate virtually all the copying from `b` to `a` by merging alternatively from `a` to `b` and from `b` to `a`.

```
public static void mergeSort(Comparable [] a)
{
   Comparable [] b = new Comparable [a.length];
   int segmentSize = 1;
   while (segmentSize < a.length)
   {
      mergePass(a, b, segmentSize); // merge from a to b
      segmentSize += segmentSize;
      mergePass(b, a, segmentSize); // merge from b to a
      segmentSize += segmentSize;
   }
}
```



```
public static void mergePass(Comparable [] x,
          Comparable [] y, int segmentSize)
{
   int i = 0;  // start of the next segment
   while (i <= x.length - 2 * segmentSize)
   {  // merge two adjacent segments from x to y
      merge(x, y, i, i + segmentSize - 1,
            i + 2 * segmentSize - 1);
      i = i + 2 * segmentSize;
   }
   // fewer than 2 full segments remain
   if (i + segmentSize < x.length)
      // 2 segments remain
      merge(x, y, i, i + segmentSize - 1, x.length - 1);
   else
      // 1 segment remains, copy to y
      for (int j = i; j < x.length; j++) y[j] = x[j];
}
```

```
public static void merge(Comparable [] c, Comparable [] d,
          int startOfFirst, int endOfFirst, int endOfSecond)
{
   int first = startOfFirst, // cursor for first segment
       second = endOfFirst + 1, // cursor for second
       result = startOfFirst; // cursor for result
   // merge until one segment is done
   while ((first <= endOfFirst) && (second <= endOfSecond))
      if (c[first].compareTo(c[second]) <= 0)
         d[result++] = c[first++];
      else
         d[result++] = c[second++];
   // take care of leftovers
   if (first > endOfFirst)
      for (int q = second; q <= endOfSecond; q++)
           d[result++] = c[q];
   else
      for (int q = first; q <= endOfFirst; q++)
           d[result++] = c[q];
}
```

Time complexity: $O(n \log n)$

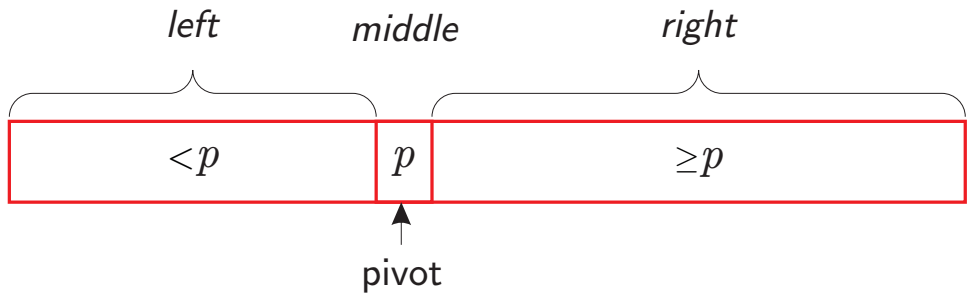
Quick Sort

In *quick sort* you first choose an element in the array as a **pivot** element. The pivot is used to separate the array into three partitions:

1) A *left* partition containing elements $<$ pivot.

2) A *middle* partition containing the pivot.

3) A *right* partition containing elements $\geq$ pivot.

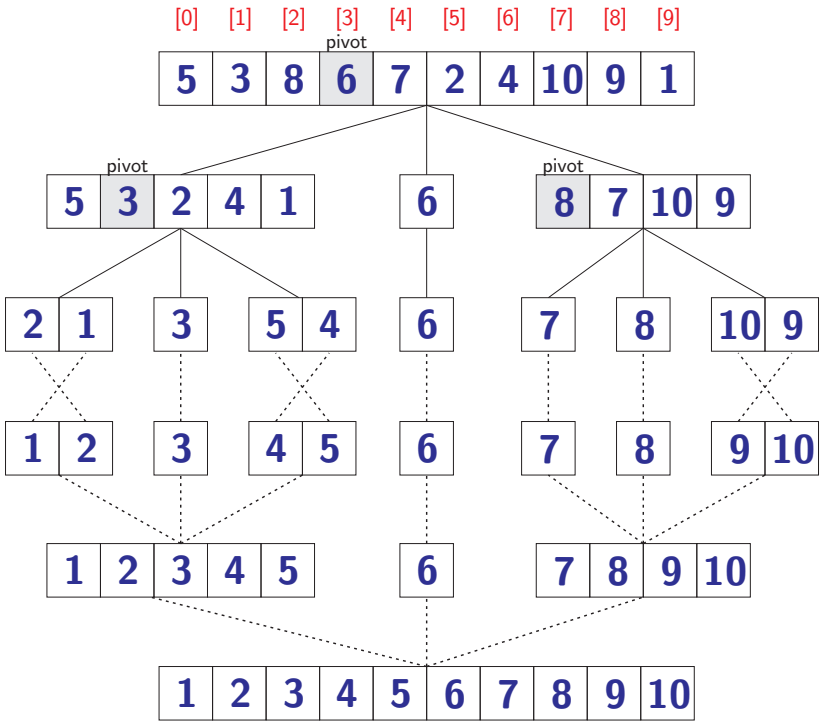


As a result, the elements in *left*, *right* and *middle* can be sorted independently, and no merge is required following the sorting of *left* and *right*.

Steps of the algorithm

- Select an element from `a[0:length-1]` for *middle*. This element is the pivot.
- Partition the remaining elements into the segments *left* and *right* so that
  - no element in *left* has a key $\geq$ pivot and
  - no element in *right* has a key $<$ pivot.
- Sort *left* using quick sort recursively.
- Sort *right* using quick sort recursively.
- The answer is *left* followed by *middle* followed by *right*.

▶ **Example:** Sort the array `a[0:9]= [5,3,8,6,7,2,4,10,9,1]`



Complexity Analysis

The worst-case computing time for quick sort is $\Theta(n^2)$, and it is achieved, for instance, when *left* is always empty.

If we are lucky and *left* and *right* are always of about the same size, then the complexity is $\Theta(n \log n)$. Therefore, the best-case complexity of quick sort is $\Theta(n \log n)$.

Surprisingly, the average complexity of quick sort is also $\Theta(n \log n)$


```
public class QuickSort
{
   // data member
   static Comparable [] a; // array of elements to sort

   /** sort a[0 : a.length - 1] using the quick sort method */
   public static void quickSort(Comparable [] a) { ... }

   /** sort a[leftEnd:rightEnd], a[rightEnd+1] >= a[leftEnd:rightEnd] */
   private static void quickSort(int leftEnd, int rightEnd) { ... }

   public static void main(String [] args) { ... }
}

public static void quickSort(Comparable [] a)
{
   QuickSort.a = a;
   if (a.length <= 1) return;
   // move largest element to right end
   MyMath.swap(a, a.length - 1, MyMath.max(a, a.length - 1));
   quickSort(0, a.length - 2);
}
```

```
private static void quickSort(int leftEnd, int rightEnd)
{
   if (leftEnd >= rightEnd) return;
   int leftCursor = leftEnd,    // left-to-right cursor
       rightCursor = rightEnd + 1; // right-to-left cursor
   Comparable pivot = a[leftEnd];
   // swap elements >= pivot on left side
   // with elements <= pivot on right side
   while (true)
   {
      do
      {  // find >= element on left side
         leftCursor++;
      } while (a[leftCursor].compareTo(pivot) < 0);
      do
      {  // find <= element on right side
         rightCursor--;
      } while (a[rightCursor].compareTo(pivot) > 0);
      if (leftCursor >= rightCursor) break;
      MyMath.swap(a, leftCursor, rightCursor);
   }
```



```
   a[leftEnd] = a[rightCursor];
   a[rightCursor] = pivot;
   quickSort(leftEnd, rightCursor - 1);
   quickSort(rightCursor + 1, rightEnd);
}
```

Time complexity: $O(n^2)$ $\Leftarrow$ worst-case

$O(n \log n)$ $\Leftarrow$ best-case

Comparison of Sort Methods

Method	Worst	Average
bubble sort	n^2	n^2
count sort	n^2	n^2
insertion sort	n^2	n^2
selection sort	n^2	n^2
heap sort	$n \log n$	$n \log n$
merge sort	$n \log n$	$n \log n$
quick sort	n^2	$n \log n$

Behind every able man, there are always other able men.

— Chinese Proverb

UNIVERSIDAD NACIONAL DE COLOMBIA

Estructuras de Datos

Sesión 8

Tree Data Structure

Yoan Pinzón, PhD
Profesor Asociado
Universidad Nacional de Colombia
http://disi.unal.edu.co/∼ypinzon/2016699/

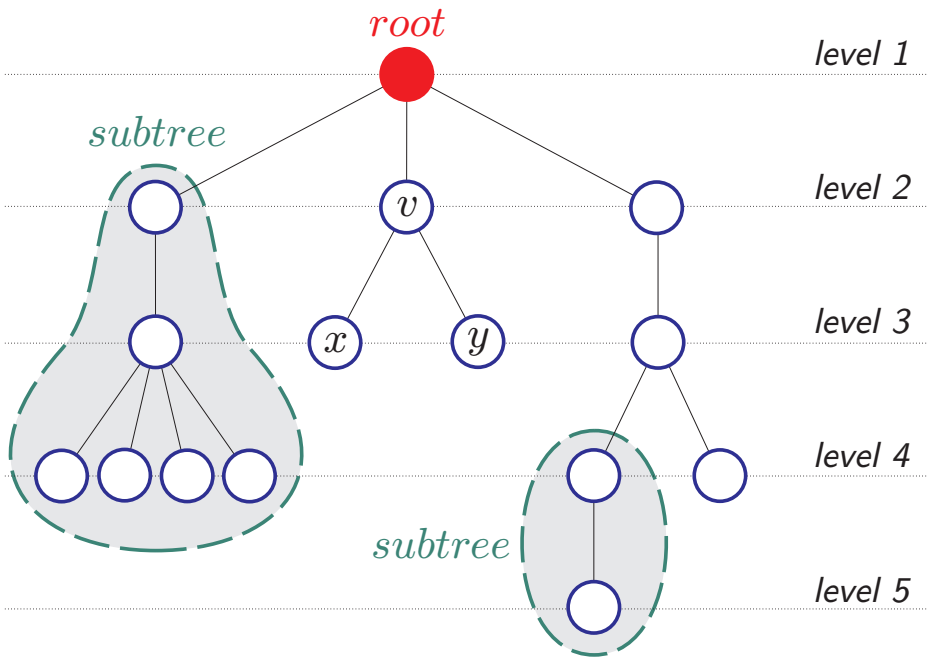
Table of Content Session 8

Tree Data Structure

- Until now: linear and tabular data
- How can we represent hierarchical data?
  - *E.g. somebody's descendants,*
  - *governmental/company subdivisions,*
  - *modular decomposition of programs, etc.*
- Answer: Tree Data Structure

Tree Terms

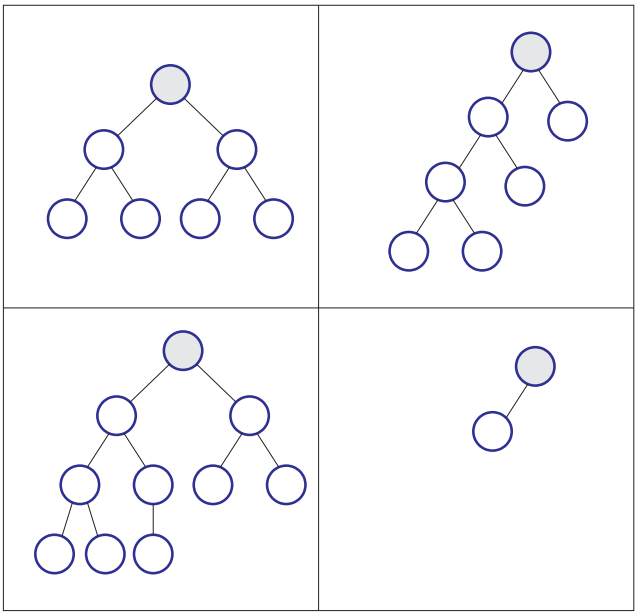
A **tree** is a finite nonempty set of elements



- x, y are **children** of v; v is a **parent** of x, y
- x, y are **siblings**
- Elements with no children are called **leaves**
- **Level:** root=level 1; children=level 2,3, ...
- **Degree of an element:** number of children
- **Height or Depth:** number of levels

Binary Trees

A **binary tree** is a tree (possible empty) in which every element has degree ≤ 2, except for the leaves.



Properties of Binary Trees

P1: *Every binary tree with n elements, $n > 0$, has exactly $n - 1$ edges.*

Proof: Each element (except the root) has one parent. $\exists$ exactly one edge between each child and its parent. Hence, $\exists n - 1$ edges. $\square$

P2: *The number of elements at level i is $\leq 2^{i-1}, i > 0$.*

Proof: By induction on i.
Basis: $i = 1$; number of elements $= 1 = 2^0$
Ind. Hypothesis: $i = k$; number of elements at level $k \leq 2^{k-1}$.
Look at level $i = k + 1$
(number of elements at level $k + 1$) $\leq 2 \cdot$(number of elements at level k)
$\leq 2 \times 2^{k-1} = 2^k$. $\square$

P3: *A binary tree of height h, $h > 0$, has at least h and at most $2^h - 1$ elements.*

Poof: Let n be the number of elements. $\exists$ must be ≥ 1 elements at each level, hence, $n \geq h$.
Now, if $h = 0$, then $n = 0 = 2^0 - 1$.
For $h > 0$, we have by P2 that

$$n \leq \sum_{i=1}^{h} 2^{i-1} = 2^h - 1$$

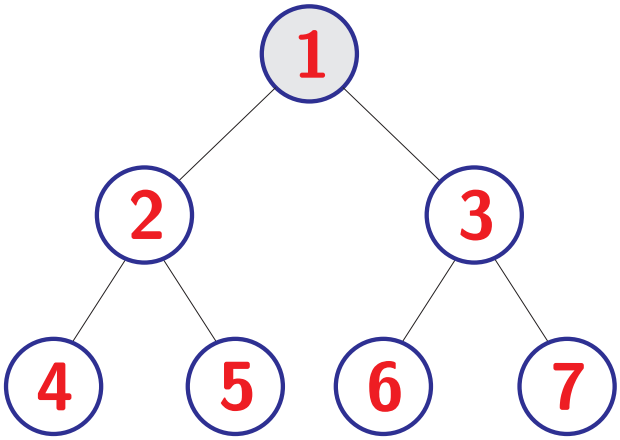
$\square$

P4: *Let h be the height of an n-elements binary tree, $n \geq 0$. Then, $\lceil \log_2(n + 1) \rceil \leq h \leq n$*

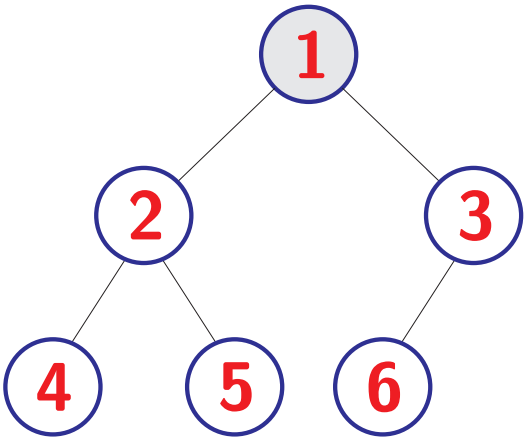
Proof: $\exists$ must be ≥ 1 element at each level, hence, $h \leq n$.
P3 $\Rightarrow n \leq 2^h - 1 \Rightarrow 2^h \geq n + 1 \Rightarrow h \geq \log_2(n + 1)$.
Since h is an integer, we have that $h \geq \lceil \log_2(n + 1) \rceil$. $\square$

Full Vs Complete

Full binary tree: A binary tree of height h is *full* if contains exactly $2^h - 1$ elements.



Complete binary tree: Is a binary tree of height h in which all levels (except perhaps for the last) have a maximum number of elements.



Number the elements from 1 through $2^h - k$, starting from level 1 and proceed in a left-to-right fashion, for some $k \geq 1$

P5: *Let i, $1 \leq i \leq n$, be the number assigned to an element v of a complete binary tree. Then:*

(i) If $i = 1$, then v is the root. If $i > 1$, then the parent of v has been assigned the number $\lfloor i/2 \rfloor$.

(ii) If $2i > n$, then v has no left child. Otherwise, its left child has been assigned the number $2i$.

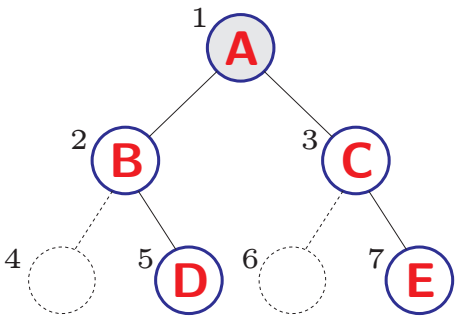
(iii) If $2i + 1 > n$, then v has no right child. Otherwise, its right child has been assigned the number $2i + 1$.

Proof: By induction on i. □

Binary Tree Data Structure

Array-based Representation

Uses **P5**



1	2	3	4	5	6	7
A	B	C		D		E

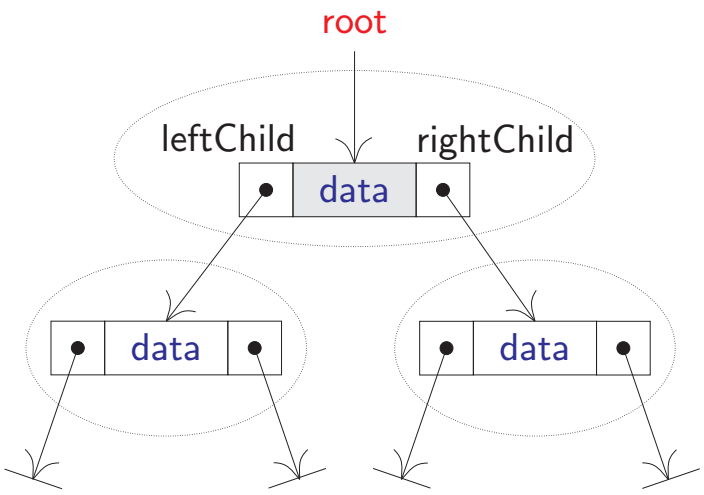
Note: An $n-$element binary tree may require an array of size $2^n - 1$ for its representation. $\Rightarrow$ Can be a waste of space

Binary Tree Data Structure

Linked Representation

The most popular way to represent a binary tree is by using links or pointers. Each node is represented by three fields:

- data
- leftChild
- rightChild



Binary Tree Traversal

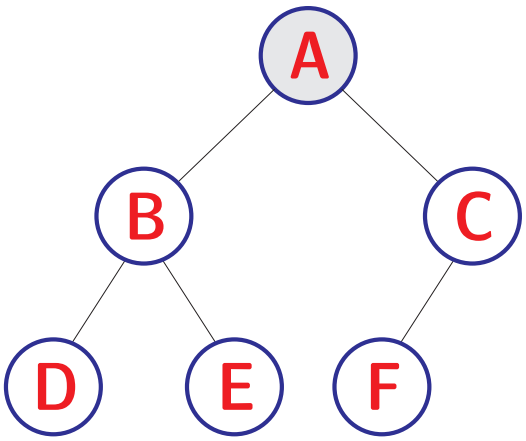
There are four common ways to traverse a binary tree:

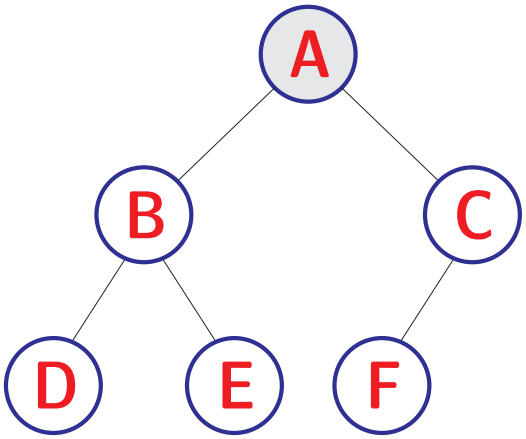
1) **Pre-order:** Visit-Left-Right

2) **In-order:** Left-Visit-Right

3) **Post-order:** Left-Right-Visit

4) **Level order**





1) **Pre-order:** ABDECF

2) **In-order:** DBEAFC

3) **Post-order:** DEBFCA

4) **Level order:** ABCDEF

The ADT BinaryTree

AbstractDataType BinaryTree
{

instances: collection of elements; if not empty, the collection is partitioned into a root, left subtree, and right subtree; each subtree is also a binary tree.
operations

`isEmpty()`: return `true` if empty, `false` otherwise

`root()`: return the root element, returns `null` if the tree is empty

`makeTree(root, left, right)`: creates a binary tree root as the root element, and left (right) as the left (right) subtree.

`removeLeftSubtree()`: remove the left subtree and return it

`removeRightSubtree()`: remove the right subtree and return it

`preOrder`: preorder traversal of the binary tree

`inOrder`: inorder traversal of the binary tree

`postOrder`: postorder traversal of the binary tree

`levelOrder`: level-order traversal of the binary tree

}

Interface Definition of BinaryTree

```
public interface BinaryTree
{
   public boolean isEmpty();
   public Object root();
   public void makeTree(Object root, Object left, Object right);
   public BinaryTree removeLeftSubtree();
   public BinaryTree removeRightSubtree();
   public void preOrder(Method visit);
   public void inOrder(Method visit);
   public void postOrder(Method visit);
   public void levelOrder(Method visit);
}
```


The Class BinaryTreeNode

```
public class BinaryTreeNode
{  // package visible data members
   Object element;
   BinaryTreeNode leftChild; // left subtree
   BinaryTreeNode rightChild; // right subtree

   // constructors
   public BinaryTreeNode() {}
   public BinaryTreeNode(Object theElement) {element = theElement;}
   public BinaryTreeNode(Object theElement,
                         BinaryTreeNode theleftChild,
                         BinaryTreeNode therightChild)
   {
       element = theElement;
       leftChild = theleftChild;
       rightChild = therightChild;
   }

   // output method
   public String toString() { return element.toString(); }
}
```

The Class LinkedBinaryTree

```
public class LinkedBinaryTree implements BinaryTree
{
   BinaryTreeNode root; // root node

   // class data members
   static Method visit;    // visit method for traversal
   static Object [] visitArgs = new Object [1];
                           // parameters of visit method
   static int count;       // counter
   static Class [] paramType = {BinaryTreeNode.class};
                           // type of parameter for visit
   static Method theAdd1;  // method to increment count
                           // by 1
   static Method theOutput; // method to output a node

   // method to initialize class data members
   static
```



```
   {
      try
      {
         Class lbt = LinkedBinaryTree.class;
         theAdd1 = lbt.getMethod("add1", paramType);
         theOutput = lbt.getMethod("output", paramType);
      }
      catch (Exception e) {}
         // exception not possible
   }

   // class methods
   public static void output(BinaryTreeNode t)
      {System.out.print(t.element + "␣");}
   public static void add1(BinaryTreeNode t) { ... }
      {count++;}
   public boolean isEmpty() { ... }
   public Object root() { ... }
   public void makeTree(Object root, Object left, Object right) { ... }
```

```
   public BinaryTree removeLeftSubtree() { ... }
   public BinaryTree removeRightSubtree() { ... }
   public void preOrder(Method visit) { ... }
   static void thePreOrder(BinaryTreeNode t) { ... }
   public void inOrder(Method visit) { ... }
   static void theInOrder(BinaryTreeNode t) { ... }
   public void postOrder(Method visit) { ... }
   static void thePostOrder(BinaryTreeNode t) { ... }
   public void levelOrder(Method visit) { ... }
   public void preOrderOutput() { ... }
   public void inOrderOutput() { ... }
   public void postOrderOutput() { ... }
   public void levelOrderOutput() { ... }
   public int size() { ... }
   public int height() { ... }
   static int theHeight(BinaryTreeNode t) { ... }
   public static void main(String [] args) { ... }
}
```


isEmpty, root, makeTree

```
/** @return true iff tree is empty */
public boolean isEmpty()
   {return root == null;}

/** @return root element if tree is not empty
   * @return null if tree is empty */
public Object root()
   {return (root == null) ? null : root.element;}

/** set this to the tree with the given root and subtrees
   * CAUTION: does not clone left and right */
public void makeTree(Object root, Object left,
                     Object right)
{
   this.root = new BinaryTreeNode(root,
                     ((LinkedBinaryTree) left).root,
                     ((LinkedBinaryTree) right).root);
}
```

removeLeftSubtree

```
/** remove the left subtree
   * @throws IllegalArgumentException when tree is empty
   * @return removed subtree */
public BinaryTree removeLeftSubtree()
{
   if (root == null)
      throw new IllegalArgumentException("tree is empty");

   // detach left subtree and save in leftSubtree
   LinkedBinaryTree leftSubtree = new LinkedBinaryTree();
   leftSubtree.root = root.leftChild;
   root.leftChild = null;
   return (BinaryTree) leftSubtree;
}
```


removeRightSubtree

```
/** remove the right subtree
   * @throws IllegalArgumentException when tree is empty
   * @return removed subtree */
public BinaryTree removeRightSubtree()
{
   if (root == null)
      throw new IllegalArgumentException("tree is empty");

   // detach right subtree and save in rightSubtree
   LinkedBinaryTree rightSubtree = new LinkedBinaryTree();
   rightSubtree.root = root.rightChild;
   root.rightChild = null;
   return (BinaryTree) rightSubtree;
}
```

preOrder

```
/** preorder traversal */
public void preOrder(Method visit)
{
   this.visit = visit;
   thePreOrder(root);
}
```



```
/** actual preorder traversal method */
static void thePreOrder(BinaryTreeNode t)
{
   if (t != null)
   {
      visitArgs[0] = t;
      // visit tree root
      try {visit.invoke(null, visitArgs);}
      catch (Exception e) { System.out.println(e); }
      thePreOrder(t.leftChild); // do left subtree
      thePreOrder(t.rightChild); // do right subtree
   }
}

/** output elements in preorder */
public void preOrderOutput()
   {preOrder(theOutput);}
```

inOrder

```
/** inorder traversal */
public void inOrder(Method visit)
{
   this.visit = visit;
   theInOrder(root);
}
```



```
/** actual inorder traversal method */
static void theInOrder(BinaryTreeNode t)
{
   if (t != null)
   {
      theInOrder(t.leftChild); // do left subtree
      visitArgs[0] = t;
      // visit tree root
      try {visit.invoke(null, visitArgs);}
      catch (Exception e)
         {System.out.println(e);}
      theInOrder(t.rightChild); // do right subtree
   }
}

/** output elements in inorder */
public void inOrderOutput()
   {inOrder(theOutput);}
```

postOrder

```
/** postorder traversal */
public void postOrder(Method visit)
{
   this.visit = visit;
   thePostOrder(root);
}
```



```
/** actual postorder traversal method */
static void thePostOrder(BinaryTreeNode t)
{
   if (t != null)
   {
      thePostOrder(t.leftChild); // do left subtree
      thePostOrder(t.rightChild); // do right subtree
      visitArgs[0] = t;
      // visit tree root
      try {visit.invoke(null, visitArgs);}
      catch (Exception e)
         {System.out.println(e);}
   }
}

/** output elements in postorder */
public void postOrderOutput()
   {postOrder(theOutput);}
```

levelOrder

```
/** level order traversal */
public void levelOrder(Method visit)
{
   ArrayQueue q = new ArrayQueue();
   BinaryTreeNode t = root;
   while (t != null)
   {
      visitArgs[0] = t;
      // visit tree root
      try {visit.invoke(null, visitArgs);}
      catch (Exception e) { System.out.println(e); }
      // put t's children on queue
      if (t.leftChild != null) q.put(t.leftChild);
      if (t.rightChild != null) q.put(t.rightChild);
      // get next node to visit
      t = (BinaryTreeNode) q.remove();
   }
}
/** output elements in level order */
public void levelOrderOutput() { levelOrder(theOutput); }
```


size

```
/** count number of nodes in tree */
public int size()
{
   count = 0;
   preOrder(theAdd1);
   return count;
}
```

height

```
/** @return tree height */
public int height()
   {return theHeight(root);}

/** @return height of subtree rooted at t */
static int theHeight(BinaryTreeNode t)
{
   if (t == null) return 0;
   // height of left subtree
   int hl = theHeight(t.leftChild);
   // height of right subtree
   int hr = theHeight(t.rightChild);
   if (hl > hr) return ++hl;
   else return ++hr;
}
```


Test program

```
public static void main(String [] args)
   LinkedBinaryTree a = new LinkedBinaryTree(),
                    x = new LinkedBinaryTree(),
                    y = new LinkedBinaryTree(),
                    z = new LinkedBinaryTree();

   y.makeTree(new Integer(1), a, a);
   z.makeTree(new Integer(2), a, a);
   x.makeTree(new Integer(3), y, z);
   y.makeTree(new Integer(4), x, a);

   System.out.println("Preorder␣sequence␣is␣");
   y.preOrderOutput();

   System.out.println("\nInorder␣sequence␣is␣");
   y.inOrderOutput();

   System.out.println("\nPostorder␣sequence␣is␣");
```

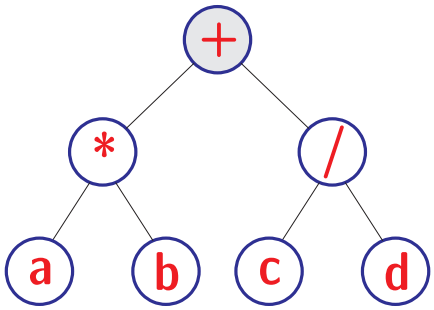
```
    y.postOrderOutput();

    System.out.println("\nLevel␣order␣sequence␣is␣");
    y.levelOrderOutput();

    System.out.println("\nNumber␣of␣nodes␣=␣" + y.size());

    System.out.println("Height␣=␣" + y.height());
}
```



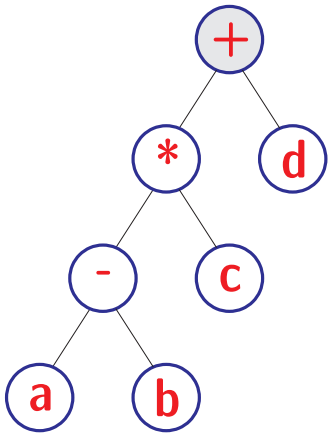
```
Preorder sequence is
4 3 1 2
Inorder sequence is
1 3 2 4
Postorder sequence is
1 2 3 4
Level order sequence is
4 3 1 2
Number of nodes = 4
Height = 3
```

Binary Tree Application

Expression Trees



(a*b)+(c/d)



(((a-b)*c)+d)

infix form:	a*b+c/d	a-b*c+d
prefix form:	+*ab/cd	+*-abcd
postfix form:	ab*cd/+	ab-c*d+

- infix: ambiguous; pre/postfix: unambiguous
- postfix evaluation:
  - Scan left-to-right
  - If an operand is encountered, it is stacked in to a stack of operands
  - If an operator is encountered, apply operator to the correct number of operands in the top of the stack and replace them for the result produced by the operator

If you are planning for a year, sow rice; if you are planning for a decade, plant trees; if you are planning for a lifetime, teach people.

— Chinese Proverb

UNIVERSIDAD NACIONAL DE COLOMBIA

Estructuras de Datos

Sesión 9

Priority Queue Data Structure

Yoan Pinzón, PhD
Profesor Asociado
Universidad Nacional de Colombia
http://disi.unal.edu.co/∼ypinzon/2016699/

Table of Content Session 9

Priority Queue Data Structure

- **Priority Queues:** FIFO structure where elements are deleted in increasing (decreasing) order of priority rather than in the order in which they arrived in the queue.

- **Max Priority Queues**: The Find/Delete operations apply to the element of maximum priority.

- **Min Priority Queues**: The Find/Delete operations apply to the element of minimum priority.

The ADT MaxPriorityQueue

AbstractDataType MaxPriorityQueue
{
instances: finite collection of elements, each has a priority
operations:

`isEmpty()`: return `true` iff the queue is empty

`size()`: return number of elements in the queue

`getMax()`: return element with maximum priority

`put(x)`: inserts the element `x` into the queue

`removeMax()`: remove the element with maximum priority and return this element;

}

The Interface MaxPriorityQueue

```
public interface MaxPriorityQueue
{
   public boolean isEmpty();
   public int size();
   public Comparable getMax();
   public void put(Comparable theObject);
   public Comparable removeMax();
}
```

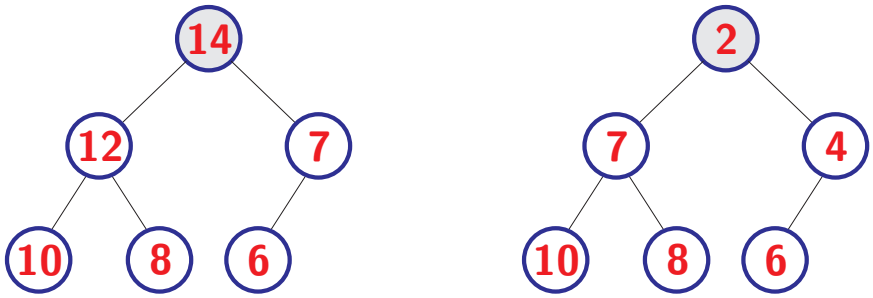

Representation of a MaxPriorityQueue

- As a Linear list
- As a Heap

Heaps

- **Max tree (min tree):** is a tree in which the value in each node is greater (less) than or equal to those in its children.

- **Max heap (min heap):** is a max (min) tree that is also a complete binary tree.



(a) MaxHeap *(b) MinHeap*

Representation of a Heap

- Since a heap is a complete binary tree, a heap can be efficiently represented as an *array*

- We can make use of property P5 to move from one node in the heap to its parent or to one of its children

- A heap with n elements has height $\lceil \log_2(n+1) \rceil$

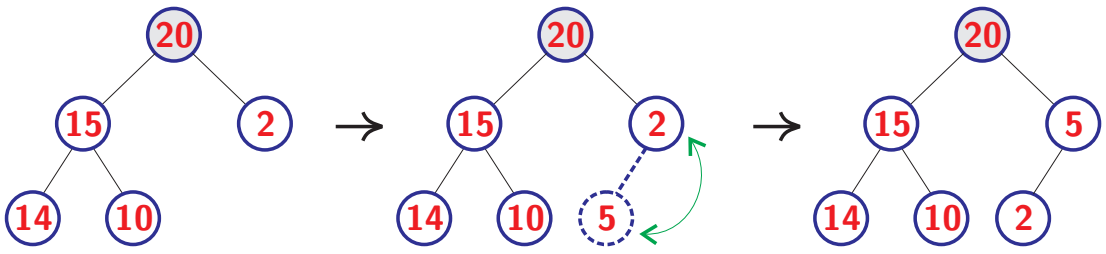
Insertion into a MaxHeap

1) Insert a new element as a leaf of the heap

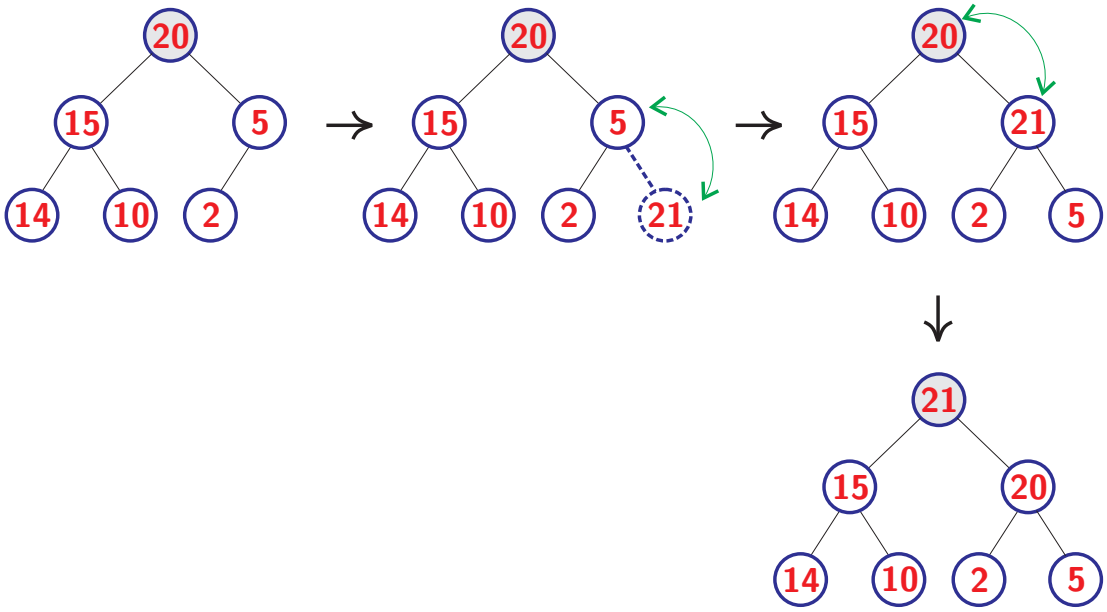
2) Walk up to the root to restore the heap properties

Time Complexity: $O(\log n)$

Example: Insert 5 into heap



Example: Insert 21 into heap



Deletion from a MaxHeap

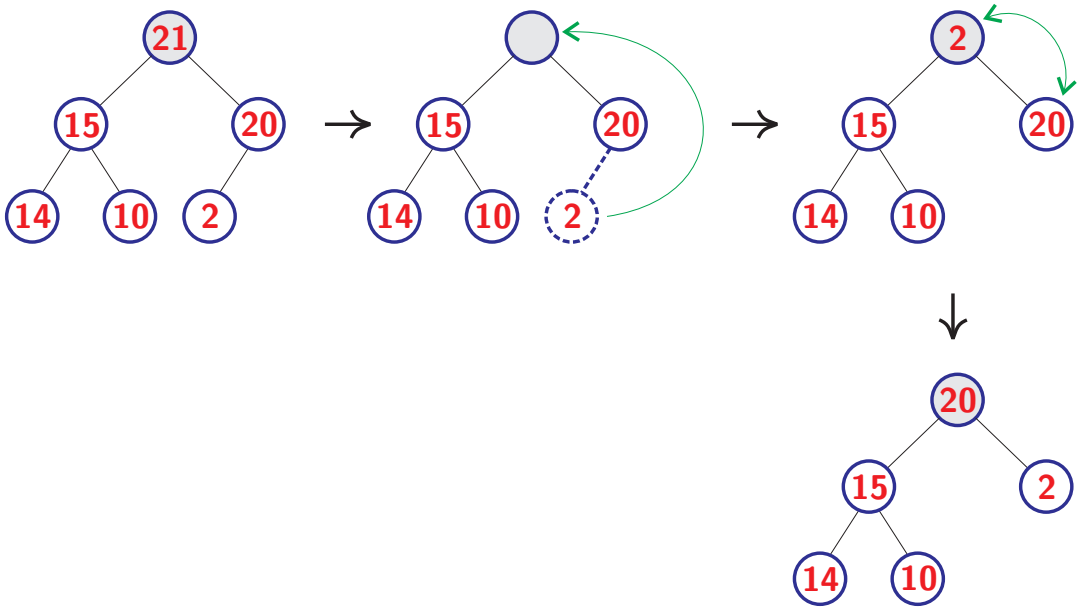
1) Delete the root element

2) Delete the rightmost leaf at the highest level and put it in the root

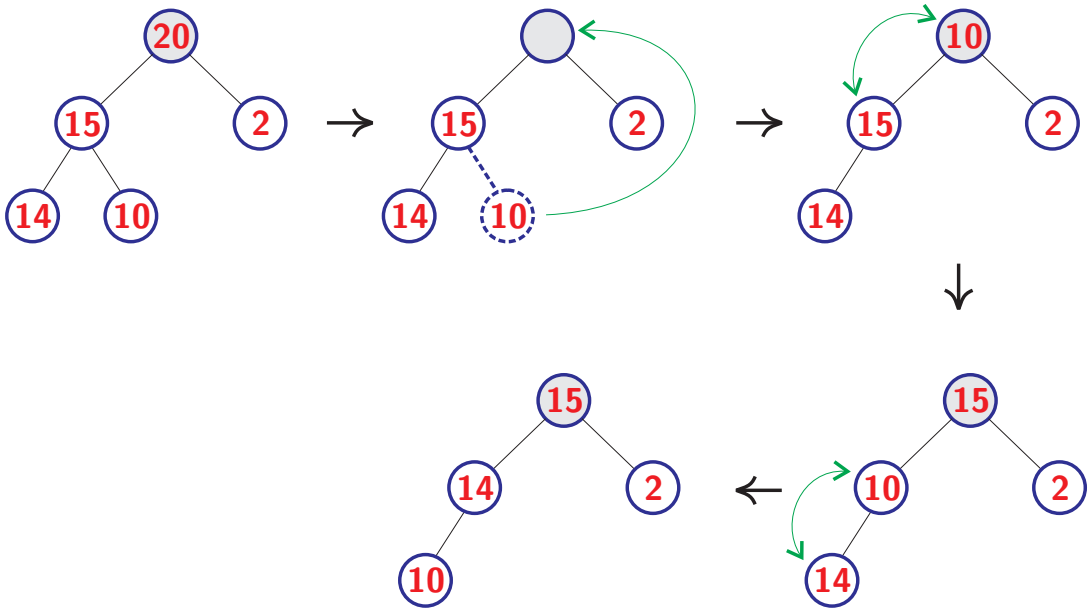
3) Restore the heap properties by walking down from root to a leaf by following the path determined by the child having the largest value

Time Complexity: $O(\log n)$

Example: Remove Max element from heap



Example: Delete 20.

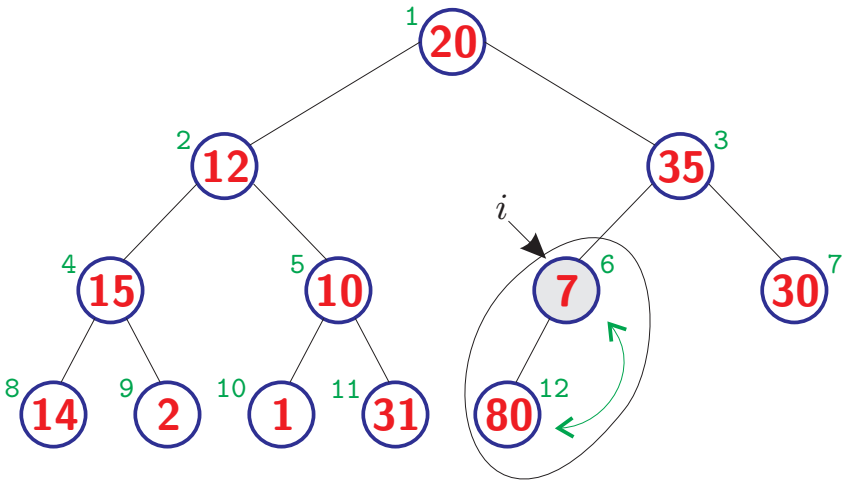


Initializing a MaxHead

- n insertions. Time $O(n \log n)$.
- Playing a tournament. Time $O(n)$.

Example: Initialize a heap with `a[1:12] = [20, 12, 35, 15, 10, 7, 30, 14, 2, 1, 31, 80]`

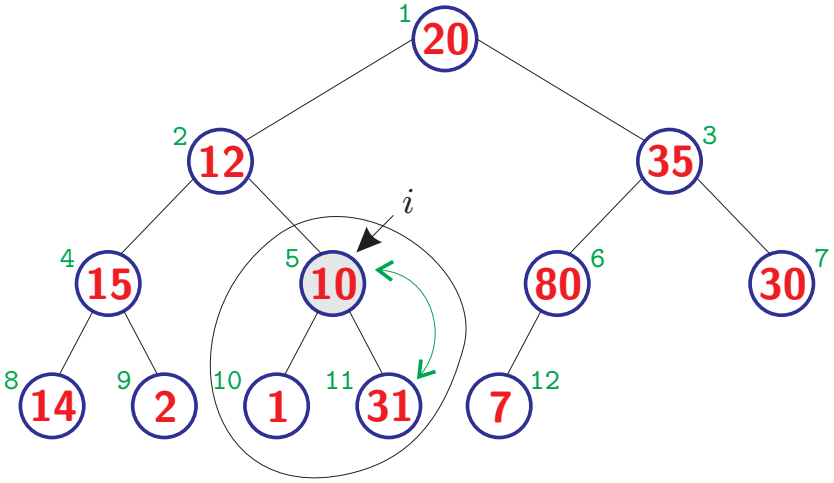
Array `a` may be interpreted as a complete binary tree as follows



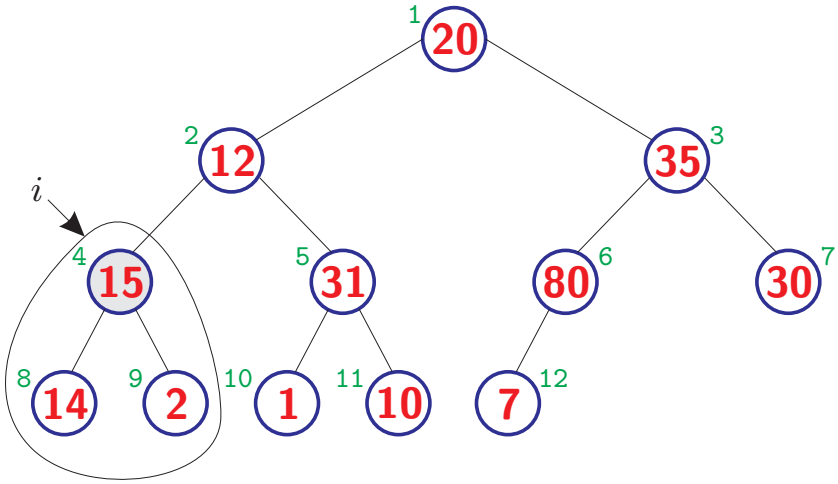
To *heapify* (i.e. make into a max heap) we begin with the last element that has a child. This element must be at position $i = \lfloor n/2 \rfloor$=6

Restore the heap properties for the subtree rooted at node 6

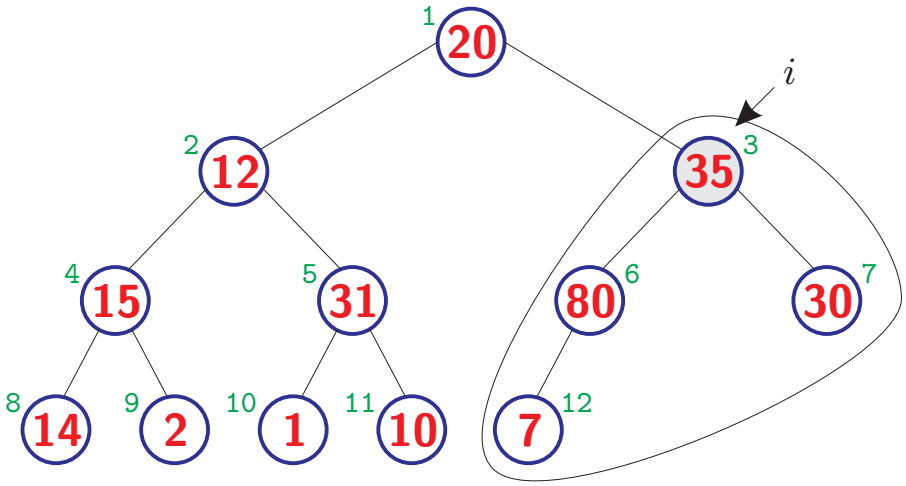
now examine the heap properties for the subtree rooted at node 5



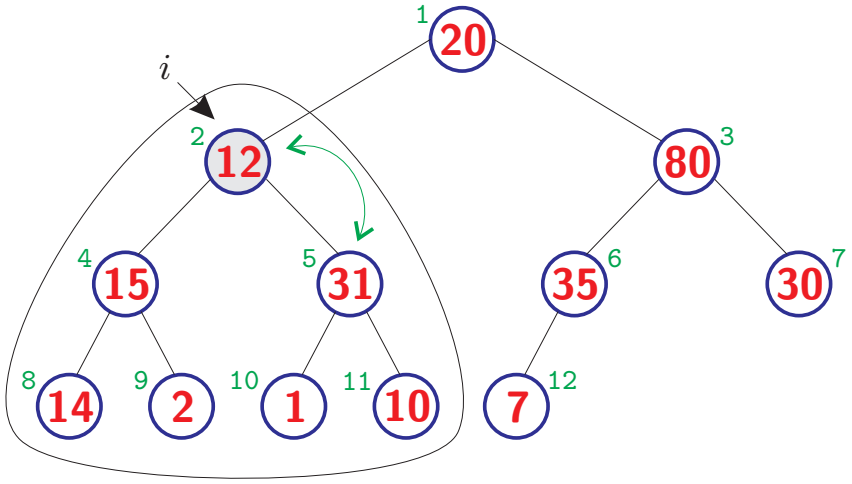
now for the subtree rooted at node 4



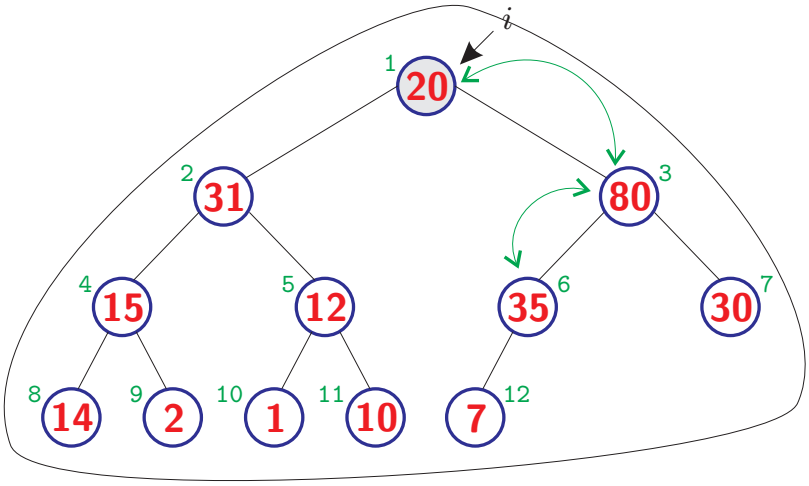
now for the subtree rooted at node 3



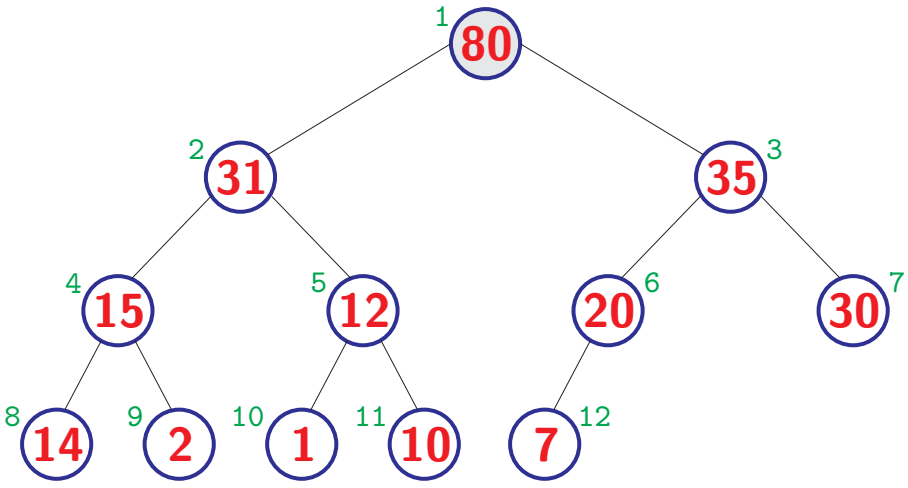
now for the subtree rooted at node 2



now for the subtree rooted at node 1



The resulting max heap is as follows



The Class MaxHeap

```
public class MaxHeap implements MaxPriorityQueue
{
   // data members
   Comparable [] heap; // array for complete binary tree
   int size;           // number of elements in heap

   // constructors
   public MaxHeap(int initialCapacity) { ... }
   public MaxHeap() { ... }

   // methods
   public boolean isEmpty() {return size == 0;}
   public int size() {return size;}
   public Comparable getMax() {return (size == 0) ? null : heap[1];}
   public void put(Comparable theElement) { ... }
   public Comparable removeMax() { ... }
   public void initialize(Comparable [] theHeap, int theSize) { ... }
   public String toString() { ... }
   public static void main(String [] args) { ... }
}
```


Constructors

```
/** create a heap with the given initial capacity
   * @throws IllegalArgumentException when
   * initialCapacity < 1 */
public MaxHeap(int initialCapacity)
{
   if (initialCapacity < 1)
      throw new IllegalArgumentException
                ("initialCapacity must be >= 1");
   heap = new Comparable [initialCapacity + 1];
   size = 0;
}

/** create a heap with initial capacity 10 */
public MaxHeap()
   {this(10);}
```

Insert

```
/** put theElement into the heap */
public void put(Comparable theElement)
{
   // increase array size if necessary
   if (size == heap.length - 1)
      heap = (Comparable []) ChangeArrayLength.changeLength1D
             (heap, 2 * heap.length);
   // find place for theElement
   // currentNode starts at new leaf and moves up tree
   int currentNode = ++size;
   while (currentNode != 1 &&
      heap[currentNode / 2].compareTo(theElement)<0)
   {
      // cannot put theElement in heap[currentNode]
      heap[currentNode] = heap[currentNode / 2];
      currentNode /= 2; // move to parent
   }

   heap[currentNode] = theElement;
}
```


Delete

```
/** remove max element and return it */
public Comparable removeMax()
{
   if (size == 0) return null;
   Comparable maxElement = heap[1];
   Comparable lastElement = heap[size--];

   // find place for lastElement starting at root
   int currentNode = 1,
       child = 2;     // child of currentNode
   while (child <= size)
   {
      // heap[child] should be larger child of currentNode
      if (child < size && heap[child].compareTo(heap[child+1])<0)
         child++;

      // can we put lastElement in heap[currentNode]?
      if (lastElement.compareTo(heap[child]) >= 0)
```

```
            break;  // yes
         // no
         heap[currentNode] = heap[child]; // move child up
         currentNode = child;          // move down a level
         child *= 2;
      }
      heap[currentNode] = lastElement;

      return maxElement;
   }
```


Initialize

```
/** initialize max heap to element array theHeap */
public void initialize(Comparable [] theHeap,
            int theSize)
{
   heap = theHeap;
   size = theSize;

   for (int root = size / 2; root >= 1; root--)
   {
      Comparable rootElement = heap[root];
      // find place to put rootElement
      int child = 2 * root; // parent of child is target
                            // location for rootElement
      while (child <= size)
      {
         // heap[child] should be larger sibling
         if (child < size &&
               heap[child].compareTo(heap[child+1])<0)
```

```
            child++;
         // can we put rootElement in heap[child/2]?
         if (rootElement.compareTo(heap[child]) >= 0)
            break; // yes
         // no
         heap[child / 2] = heap[child]; // move child up
         child *= 2; // move down a level
      }
      heap[child / 2] = rootElement;
   }
}
```


Test Program

```
/** test program */
public static void main(String [] args)
{
   // test constructor and put
   MaxHeap h = new MaxHeap(4);
   h.put(new Integer(10));
   h.put(new Integer(20));
   h.put(new Integer(5));
   h.put(new Integer(15));
   h.put(new Integer(30));
   System.out.println("Elements␣in␣array␣order␣are");
   System.out.println(h);
   System.out.println();

   // test remove max
   System.out.print("Deleted␣max␣element␣")
   System.out.println(h.removeMax());
   System.out.print("Deleted␣max␣element␣")
   System.out.println(h.removeMax());
   System.out.println("Elements␣in␣array␣order␣are");
   System.out.println(h);
   System.out.println();
}
```

```
Elements in array order are

The 5 elements are [30, 20, 5, 10, 15]

Deleted max element 30

Deleted max element 20

Elements in array order are

The 3 elements are [15, 10, 5]
```


Heap Application

Heap Sort

A heap can be used to sort n elements in $O(n \log n)$ time

1) Initialize a max heap with n elements (time $O(n)$)

2) Extract (i.e. delete) elements from the heap one at a time. Each deletion takes $O(\log n)$ time, so the total time is $O(n \log n)$

```
public class HeapSort
{
   public static void heapSort(Comparable [] a)
   {
      MaxHeap h = new MaxHeap();
      h.initialize(a, a.length - 1);

      for (int i = a.length - 2; i >= 1; i--)
         a[i + 1] = h.removeMax();
   }

   public static void main(String [] args)
   {
      Integer [] a = {null, new Integer(3), new Integer(2),
                      new Integer(4), new Integer(1)};
      heapSort(a);

      System.out.println("The␣sorted␣order␣is");
      for (int i = 1; i < a.length; i++) System.out.print(a[i] + "␣");
      System.out.println();
   }
}
```



```
The sorted order is

1 2 3 4
```

You cannot prevent the birds of sorrow from flying over your head, but you can prevent them from building nests in your hair.

— Chinese Proverb

UNIVERSIDAD NACIONAL DE COLOMBIA

Estructuras de Datos

Sesión 10

Dictionary Data Structure

Yoan Pinzón, PhD
Profesor Asociado
Universidad Nacional de Colombia
http://disi.unal.edu.co/∼ypinzon/2016699/

Table of Content Session 10

Dictionary Data Structure

Dictionary: collection of pairs of the form (k, e), where k is the *key* and e is the element associated with the key k, supporting the following operations:

- **Insert** an element with a specific key value
- **Search** the dictionary for an element with a specific key value
- **Delete** an element with a specific key value

No two pairs in a dictionary have the same key

The ADT Dictionary

AbstractDataType Dictionary
{
instances: collection of elements with distinct keys
operations:

`get(k)`: return the element with key `k`

`put(k, x)`: put the element `x` whose key is `k` into the dictionary and return the old element (if any) associated with `k`

`remove(k)`: remove the element with key `k` and return it

}

The Interface Dictionary

```
public interface Dictionary
{
   public Object get(Object key);
   public Object put(Object key, Object theElement);
   public Object remove(Object key);
}
```


Dictionary Data Structure

Linear List Representation

A dictionary is maintained as an ordered linear list $(e_0, e_1, \ldots)$, where the e_is are dictionary pairs in ascending order of key.

Using the linked representation, the class definition for a dictionary looks as follow:

The Class SortedChain

```
public class SortedChain implements Dictionary
{
   // top-level nested class
   protected static class SortedChainNode
   {
      // data members
      protected Comparable key;
      protected Object element;
      protected SortedChainNode next;
   }

   // data members of SortedChain
   protected SortedChainNode firstNode;
   protected int size;

   // methods
   public boolean isEmpty() {return size == 0;}
   public Object get(Object theKey) { ... }
```



```
   public Object put(Object theKey, Object theElement)
      { ... }
   public Object remove(Object theKey) { ... }
   public String toString() { ... }

   /** test program */
   public static void main(String [] args) { ... }
}
```

get

```
/** @return element with specified key
   * @return null if there is no matching element */
public Object get(Object theKey)
{
   SortedChainNode currentNode = firstNode;

   // search for match with theKey
   while (currentNode != null &&
            currentNode.key.compareTo(theKey) < 0)
      currentNode = currentNode.next;

   // verify match
   if (currentNode != null &&
         currentNode.key.equals(theKey))
      // yes, found match
      return currentNode.element;

   // no match
   return null;
}
```


remove

```
/** @return matching element and remove it
   * @return null if no matching element */
public Object remove(Object theKey)
{
   SortedChainNode p = firstNode, tp = null; // tp trails p
   // search for match with theKey
   while (p != null && p.key.compareTo(theKey) < 0)
   {
      tp = p; p = p.next;
   }
   // verify match
   if (p != null && p.key.equals(theKey))
   {// found a match
      Object e = p.element; // the matching element
      // remove p from the chain
      if (tp == null) firstNode = p.next;
      else tp.next = p.next;
      size--;
      return e;
   }
   // no matching element to remove
   return null;
}
```

put

```
/** insert an element with the specified key
   * overwrite old element if there is already an
   * element with the given key
   * @return old element (if any) with key theKey */
public Object put(Object theKey, Object theElement)
{
   SortedChainNode p = firstNode,
                   tp = null; // tp trails p
   // move tp so that theElement can be inserted after tp
   while (p != null && p.key.compareTo(theKey) < 0)
   {
      tp = p;
      p = p.next;
   }
   // check if there is a matching element
   if (p != null && p.key.equals(theKey))
   {// replace old element
      Object elementToReturn = p.element;
```



```
      p.element = theElement;
      return elementToReturn;
   }
   // no match, set up node for theElement
   SortedChainNode q = new
              SortedChainNode(theKey, theElement, p);
   // insert node just after tp
   if (tp == null) firstNode = q;
   else tp.next = q;
   size++;
   return null;
}
```

Dictionary Data Structure

Skip List Representation

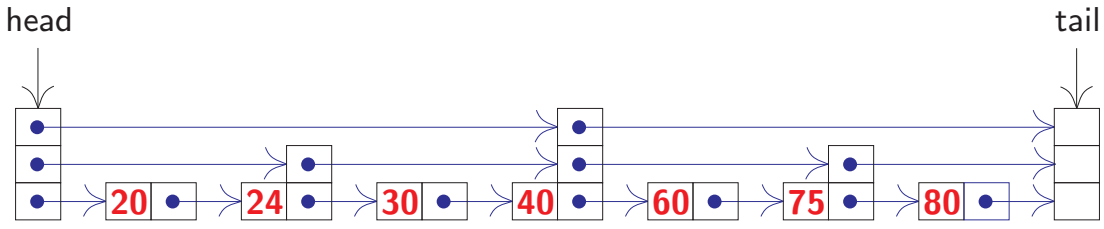
- Searching for a given element in a dictionary of size n represented as a sorted chain may required up to n comparisons
- Can we do better?
- The number of comparisons can be reduced to $n/2 + 1$ if we keep a pointer to the middle element:
  - Compare first the given element to the middle one
  - If it is smaller, look at the left half of the chain; otherwise, look at the right half
- Apply this idea recursively to each half of the chain

In general we generate chains at various levels:

- Level 0 chain includes all n elements
- Level 1 chain includes every second element
- Level 2 chain includes every fourth element
- ...
- Level i chain includes every 2^ith element

An element is a *level i element* iff it is in the chain for levels 0 through i and it is not on the level $i + 1$ chain.

The above data structure is called **skip list**.



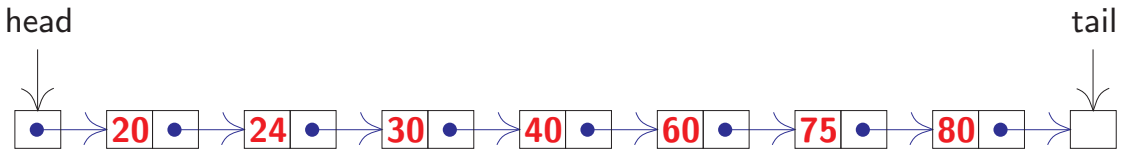
Insertions and Deletions

- When insertion and deletion occur, we cannot maintain the regular structure of a skip list without doing $O(n)$ work
- However, we can approximate this structure
- Observation: there are $n/2^i$ level i elements. Hence, when insertion occurs, the element level is i with probability $1/2^i$
- In general, we can allow any probability to be used: the newly inserted element is assigned to level i with probability p^i, for some $0 < p < 1$
- The number of chain levels is $\lfloor \log_{1/p} n \rfloor + 1$
- Level i includes every $1/p$th element of level $i - 1$

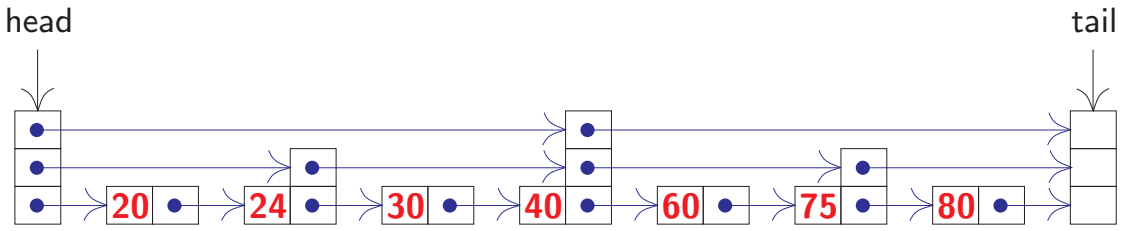
Insertion of an element with key k

1. First search to check that no element with key k is present
2. During the search determine those pointers of elements at various levels that could break after the insertion
3. Make the insertion by assigning a level to the new element. This assignment can be made using a random number generator

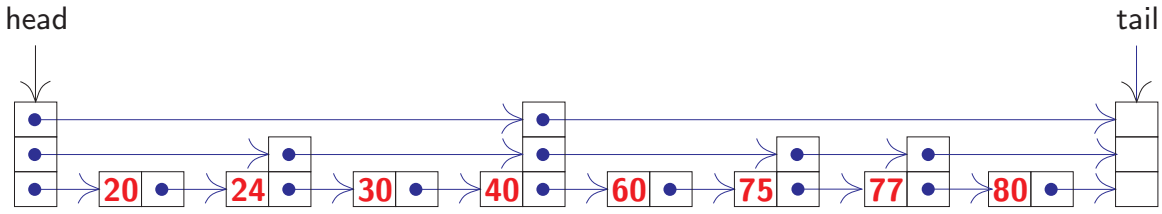
Example:



(a) A sorted chain with head and tail nodes



(b) Pointers to middle and every second node added



(c) 77 Inserted

Deletion of an element with key k

1. First search to find the element with key k
2. During the search determine those pointers of elements at various levels that need to be updated after the deletion
3. Let the element to be delete be at level i chain. Then, the pointers determined during step 2 and belong to levels i to 0 should made to point to element after the one to be deleted

Assigning Levels

- Observation: in a regular skip list a fraction p of elements on the level $i-1$ chain are also on the level i chain. Hence,
- Pr[$e \in$ level $i-1$ and $e \in$ level i]=p
- Assume that we have a uniform random number generator that generates numbers in the range 0 trough 1. Then,
- The above suggest the following method:
  1. Generate a random number. If it is $\leq$ p, then the new element should be at level 1 chain
  2. Generate another random number. If it is again $\leq$ p, then the new element is also at level 2 chain
  3. Repeat the above steps until a random number $>$ p is generated

```
int level()
{
   int lev = 0;
   while (r.nextFloat() <= prob) lev++;
   return (lev <= maxLevel) ? lev : maxLevel;
}
```

The Class SkipNode

```
protected static class SkipNode
{
   // data members
   protected Comparable key;
   protected Object element;
   protected SkipNode [] next;

   // constructor
   protected SkipNode(Object theKey, Object theElement,
                      int size)
   {
      key = (Comparable) theKey;
      element = theElement;
      next = new SkipNode [size];
   }
}
```


The Class SkipList

```
public class SkipList implements Dictionary
{
   // data members of SkipList
   protected float prob; // probability to decide level
   protected int maxLevel; // max permissible chain level
   protected int levels; // max current nonempty chain
   protected int size; // current number of elements
   protected Comparable tailKey; // a large key
   protected SkipNode headNode; // head node
   protected SkipNode tailNode; // tail node
   protected SkipNode [] last; // last node on each level
   protected Random r; // needed for random numbers

   // constructor
   public SkipList(Comparable largeKey, int maxElements,
              float theProb)
   { ... }
```

```
   // methods
   public boolean isEmpty() {return size == 0;}
   public Object get(Object theKey) { ... }
   SkipNode search(Object theKey) { ... }
   int level() { ... }
   public Object put(Object theKey, Object theElement)
      { ... }
   public Object remove(Object theKey) { ... }
   public static void main(String [] args) { ... }
}
```


Constructor

```
/** create an empty skip list
   * @param largekey used as key in tail node
   * all elements must have a smaller key than this
   * @param maxElements largest number of elements
   * to be stored in the dictionary
   * @param theProb probability that element on one
   * level is also on the next level */
public SkipList(Comparable largeKey, int maxElements,
          float theProb)
{
   prob = theProb;
   maxLevel = (int) Math.round(Math.log(maxElements) /
                              Math.log(1/prob)) - 1;
   // size and levels have default initial value 0
   tailKey = largeKey;

   // create head & tail nodes and last array
   headNode = new SkipNode (null, null, maxLevel + 1);
```

```
   tailNode = new SkipNode (tailKey, null, 0);
   last = new SkipNode [maxLevel + 1];

   // headNode points to tailNode at all levels initially
   for (int i = 0; i <= maxLevel; i++)
         headNode.next[i] = tailNode;

   // initialize random number generator
   r = new Random();
}
```


get

```
/** @return element with specified key
   * @return null if there is no matching element */
public Object get(Object theKey)
{
   if (tailKey.compareTo(theKey) <= 0) return null;
   SkipNode p = headNode;
   for (int i = levels; i >= 0; i--)
      while (p.next[i].key.compareTo(theKey) < 0)
         p = p.next[i];

   if (p.next[0].key.equals(theKey))
      return p.next[0].element;
   return null; // no matching element
}
```

search

```
/** search for theKey saving last nodes seen at each
   * level in the array last
   * @return node that might contain theKey */
SkipNode search(Object theKey)
{
   SkipNode p = headNode;
   for (int i = levels; i >= 0; i--)
   {
      while (p.next[i].key.compareTo(theKey) < 0)
         p = p.next[i];
      last[i] = p; // last level i node seen
   }
   return (p.next[0]);
}
```


put

```
/** insert an element with the specified key
   * overwrite old element if there is already an
   * element with the given key
   * @return old element (if any) with key theKey
   * @throws IllegalArgumentException when
   * theKey >= largeKey = tailKey */
public Object put(Object theKey, Object theElement)
{
   SkipNode p = search(theKey);
   if (p.key.equals(theKey))
   {  // update p.element
      Object elementToReturn = p.element;
      p.element = theElement;
      return elementToReturn;
   }
   int lev = level(); // level of new node
   if (lev > levels)
   {
```

```
      lev = ++levels; last[lev] = headNode;
   }
   SkipNode y = new SkipNode (theKey,theElement,lev+1);
   for (int i = 0; i <= lev; i++)
   {  // insert into level i chain
      y.next[i] = last[i].next[i];
      last[i].next[i] = y;
   }
   size++;
   return null;
}
```


remove

```
/** @return matching element and remove it
   * @return null if no matching element */
public Object remove(Object theKey)
{
   if (tailKey.compareTo(theKey) <= 0) // too large
      return null;

   // see if matching element present
   SkipNode p = search(theKey);
   if (!p.key.equals(theKey)) // not present
      return null;

   // delete node from skip list
   for (int i = 0; i <= levels && last[i].next[i] == p; i++)
      last[i].next[i] = p.next[i];

   // update Levels
   while (levels>0 && headNode.next[levels] == tailNode) levels--;

   size--;
   return p.element;
}
```

Dictionary Data Structure

Hash Table Representation

- It uses a **hash function** f to map keys into positions in a table called the **hash table**

- Let $\mathcal{K}$ be the domain of all keys and let $\mathcal{N}$ be the set of natural numbers. Then,
  $f : \mathcal{K} \rightarrow \mathcal{N}$. The element with key k is stored in position $f(k)$ of the hash table

Search for an element with key k:

- Compute $f(k)$ and see if there is an element at position $f(k)$ of the table

Delete an element with key k:

- Search for the element. If found, then make the position $f(k)$ of the table empty

Insert an element with key k:

- Search for the element. If not found, i.e., position $f(k)$ of the table is empty, then place the element at that position

- If the key range is small, then we can easily implement hashing

  E.g., let keys for a student record dictionary be 6 digit ID numbers, and that we have 1000 students with IDs ranging from 771000 to 772000. Then, $f(k) = k - 771000$ maps IDs to positions 0 through 1000 of the hash table

- The above situation is called *ideal hashing*

- However, what we do when the key range is large?

  E.g., keys are 12-character strings; each key has to be converted into a numeric value by say mapping a blank to 0, an 'A' to 1, ..., and a 'Z' to 26. This conversion maps the keys into integers in the range [1, $27^{12} - 1$]

Hashing with Linear Open Addressing

- The size of the hash is smaller than the key range

- Find a hash function which maps several keys into the same position of the hash table

- A commonly used such function is

  $$f(k) = k\%D$$

  where D is the size of the hash table

- Each position of the hash table is called a *bucket*

- What happens if $f(k_1) = f(k_2)$, for $k_1 \neq k_2$, i.e., a so-called *collision* has occurred

- If the relevant bucket has space to store an additional element, then we are done. Otherwise, we have an *overflow* problem

- How do we overcome overflows?

- Search the table (sequentially) to find the next available bucket to store the new element

Search for element with key k

- Search starting from the *home bucket* $f(k)$ and by examining successive buckets (and considering the table as circular) until one of the following happens:

  1. A bucket containing the element with the key k is found

  2. An empty bucket is reached

  3. We returned to the home bucket

- In case 1, we have found the element we are looking for. In the other two cases, the table doesn't contain the requested element

Deleting the element with key k

- Deletion needs special care: if we simply make table position empty, then we may invalidate the correctness of the Search method
- This implies that deletion may require to move several elements in order to leave the table in a state appropriate for the Search method
- *Alternative solution:* introduce a field `neverUsed` in each bucket. Initially this is set to true

  When an element is placed into a bucket, its `neverUsed` field becomes false

  Case 2 of Search is replaced by: "a bucket with `neverUsed` field equal to true is reached"
- Deletion is accomplished by simply vacating the relevant bucket
- The alternative solution requires the re-organization of the hash table when the number of buckets with false `neverUsed` is large

The Class `HashEntry`

```
protected static class HashEntry
{
   // data members
   protected Object key;
   protected Object element;

   // constructors
   private HashEntry() {}

   private HashEntry(Object theKey, Object theElement)
   {
      key = theKey;
      element = theElement;
   }
}
```

The Class HashTable

```
public class HashTable
{
   // data members of HashTable
   protected int divisor; // hash function divisor
   protected HashEntry [] table; // hash table array
   protected int size; // number of elements in table

   // constructors here

   // methods
   public boolean isEmpty() {return size == 0;}

   private int search(Object theKey) { ... }
   public Object get(Object theKey) { ... }
   public Object put(Object theKey, Object theElement)
      { ... }

   /** test method */
   public static void main (String [] args) { ... }
}
```


search

```
/** search an open addressed hash table for an element
   * with key theKey
   * @return location of matching element if found,
   * otherwise return location where an element with
   * key theKey may be inserted provided the hash
   * table is not full */
private int search(Object theKey)
{
   int i = Math.abs(theKey.hashCode()) % divisor;
   int j = i;   // start at home bucket
   do
   {
      if (table[j] == null || table[j].key.equals(theKey))
         return j;
      j = (j + 1) % divisor; // next bucket
   } while (j != i);        // returned to home bucket?

   return j; // table full
}
```

get

```
/** @return element with specified key
   * @return null if no matching element */
public Object get(Object theKey)
{
   // search the table
   int b = search(theKey);

   // see if a match was found at table[b]
   if (table[b] == null || !table[b].key.equals(theKey))
      return null;         // no match

   return table[b].element; // matching element
}
```


put

```
/** insert an element with the specified key
   * overwrite old element if there is already an
   * element with the given key
   * @throws IllegalArgumentException when full
   * @return old element (if any) with key theKey */
public Object put(Object theKey, Object theElement)
{
   // search the table for a matching element
   int b = search(theKey);
   // check if matching element found
   if (table[b] == null)
   {
      table[b] = new HashEntry(theKey, theElement);
      size++;
      return null;
   }
   else
   {  // check if duplicate or table full
```

```
      if (table[b].key.equals(theKey))
      {  // duplicate, change table[b].element
         Object elementToReturn = table[b].element;
         table[b].element = theElement;
         return elementToReturn;
      }
      else // table is full
         throw new IllegalArgumentException("full table");
   }
}
```