|  |  |  |
| --- | --- | --- |
| **Contents** | | |
| Prg | **Program Statement** | Page |
| 1 | [Write a Java program that prints all real solutions to the quadratic equation](#_bookmark1)  [a*x*2 + b*x* + c = 0. Read in a, b, c and use the quadratic formula. If the discriminant (b2– 4ac) is negative; display a message stating that there are no real solutions.](#_bookmark1) | **1** |
| 2 | [The Fibonacci sequence is defined by the following rule: The first two values in the sequence are 0 and 1. Every subsequent value is the sum of the two values preceding it. Write a Java program that uses both recursive and non- recursive functions to print the *n*th value in the Fibonacci sequence.](#_bookmark2) | **3** |
| 3 | [Write a Java program that prompts the user for an integer and then prints out all prime numbers up to that integer.](#_bookmark3) | **4** |
| 4 | [Write a Java program that checks whether a given string is a palindrome or not. For example, “MADAM” is a palindrome.](#_bookmark4) | **6** |
| 5 | [Write a Java program for sorting a given list of names in ascending order.](#_bookmark5) | **7** |
| 6 | [Write a Java program to multiply two given matrices.](#_bookmark6) | **8** |
| 7 | [Write a Java Program that reads a line of integers, and then displays each integer, and the sum of all the integers (use **StringTokenizer** class).](#_bookmark7) | **9** |
| 8 | [Write a Java program that reads a file name from the user then displays information about whether the file exists, whether the file is readable, whether the file is writable, the type of file and the length of the file in bytes.](#_bookmark8) | **11** |
| 9 | [Write a Java program that reads a file and displays the contents of the file on the screen, with a line number before each line.](#_bookmark9) | **12** |
| 10 | [Write a Java program that displays the number of characters, lines and words in a text file.](#_bookmark10) | **13** |
| 11 | [Write a Java program for creating multiple threads.](#_bookmark11)   1. [Using Thread class](#_bookmark11) 2. [Using Runnable interface](#_bookmark11) | **14** |
| 12 | [Write a Java program that illustrates how run time polymorphism is achieved.](#_bookmark12) | **16** |

|  |  |  |
| --- | --- | --- |
| **13** | [Write a java program that illustrates the following:](#_bookmark0)   1. [Creation of simple package.](#_bookmark0) 2. [Accessing a package.](#_bookmark0) 3. [Implementing interfaces.](#_bookmark0) | **17** |
| **14** | [Write a java program that illustrates the following](#_bookmark13)   1. [Handling predefined exceptions](#_bookmark13) 2. [Handling user defined exceptions](#_bookmark13) | **19** |
| **15** | [Write Java programs that use both recursive and non-recursive functions for implementing the following searching methods:](#_bookmark14)   1. [Linear search](#_bookmark14) 2. [Binary search](#_bookmark14) | **22** |
| **16** | [Write Java programs to implement the List ADT using arrays and linked lists.](#_bookmark15) | **25** |
| **17** | [Write Java programs to implement the following using an array.](#_bookmark16)   1. [Stack ADT](#_bookmark16) 2. [Queue ADT](#_bookmark16) | **32** |
| **18** | [Write a java program that reads an infix expression, converts the expression to postfix form and then evaluates the postfix expression (use stack ADT).](#_bookmark17) | **37** |
| **19** | [Write a java program that determines whether parenthetic symbols ( ), { } and [ ] are nested correctly in a string of characters(use stack ADT).](#_bookmark18) | **40** |
| **20** | [Write a java program that uses both stack and queue to test whether the given string is a palindrome.](#_bookmark19) | **41** |
| **21** | [Write Java programs to implement the following using a singly linked list.](#_bookmark20)   1. [Stack ADT](#_bookmark20) 2. [Queue ADT](#_bookmark20) | **43** |
| **22** | [Write Java programs to implement the deque (double ended queue) ADT using](#_bookmark21)   1. [Array](#_bookmark21) 2. [Doubly linked list.](#_bookmark21) | **47** |
| **23** | [Write a Java program to implement priority queue ADT.](#_bookmark22) | **53** |
| **24** | [Write Java programs that use recursive and non-recursive functions to traverse the given binary tree in](#_bookmark23)   1. [Preorder](#_bookmark23) 2. [Inorder and](#_bookmark23) 3. [Postorder.](#_bookmark23) | **56** |
| **25** | [Write a Java program that displays node values in a level order traversal (Traverse the tree one level at a time, starting at the root node) for a binary tree.](#_bookmark24) | **60** |

|  |  |  |
| --- | --- | --- |
| **26** | [Write a Java program that uses recursive functions.](#_bookmark25)   1. [To create a binary search tree.](#_bookmark25) 2. [To count the number of leaf nodes.](#_bookmark25) 3. [To copy the above binary search tree.](#_bookmark25) | **63** |
| **27** | [Write a Java program to perform the following operations:](#_bookmark25)   1. [Insert an element into a binary search tree.](#_bookmark25) 2. [Delete an element from a binary search tree.](#_bookmark25) 3. [Search for a key element in a binary search tree.](#_bookmark25) | **63** |
| **28** | [Write a Java program to perform the following operations](#_bookmark26)   1. [Insertion into an AVL-tree](#_bookmark26) 2. [Deletion from an AVL-tree](#_bookmark26) | **66** |
| **29** | [Write a Java program to perform the following operations:](#_bookmark27)   1. [Insertion into a B-tree](#_bookmark27) 2. [Deletion from a B-tree](#_bookmark27) | **71** |
| **30** | [Write a Java program to implement all the functions of a dictionary (ADT) using Hashing.](#_bookmark28) | **76** |
| **31** | [Write Java programs for the implementation of bfs and dfs for a given graph.](#_bookmark29) | **81** |
| **32** | Write Java programs for implementing the following sorting methods:   1. [Bubble sort](#_bookmark30) 2. [Selection sort](#_bookmark31) 3. [Insertion sort](#_bookmark32) 4. [Quick sort](#_bookmark33) 5. [Merge sort](#_bookmark34) 6. [Heap sort](#_bookmark35) 7. [Radix sort](#_bookmark36) 8. [Binary tree sort](#_bookmark37) | **85** |
| **33** | [Write a Java program for implementing KMP pattern matching algorithm.](#_bookmark38) | **98** |

# Overview of Java

This section introduces the students to elementary Java. Java is a vast computer language and programming environment, it is not possible to touch upon all Java-related issues. This section introduces only those aspects of Java that are necessary for understanding the Java code offered in this book. Java program examples presented in this section give you to implement data structures.

## Quadratic equation

1. Write a Java program that prints all real solutions to the quadratic equation a*x*2 + b*x* + c = 0. Read in a, b, c and use the quadratic formula. If the discriminant (b2– 4ac) is negative; display a message stating that there are no real solutions.

The roots of the quadratic equation, *ax*2 + *bx* + *c* = 0 are given by:

*x* = [–*b* ± √(*b*2 – 4*ac*) ]/2*a* The discriminant, *d* = (*b*2 – 4*ac*)

*Case* (1): When *d* is greater than zero, the two roots are real.

*x*1 = (**–***b* + √*d* )/2*a* and *x*2 = (–*b* – √*d* )/2*a* Case (2): When *d* is equal to zero, the two roots are equal.

*x*1 = *x*2 = – *b* /2*a*

Case (3): When *d* is less than zero, the two roots are imaginary. Each of the two roots has two parts: real-part-1, imaginary-part-1 and real-part-2, imaginary-part-2.

real-part-1, *xr*1 = –*b*/2*a* imaginary-part-1, *xi*1 = +√*d* /2*a* real-part-2, *xr*2 = –*b*/2*a* imaginary-part-2, *xi*2 = –√*d* /2*a*

Program 1: Roots of a quadratic equation

import java.lang.Math; import java.util.Scanner;

class **QuadraticEquation**

{

double a, b, c; // coefficients

QuadraticEquation( double a, double b, double c)

{

this.a = a; this.b = b; this.c = c;

}

public void roots()

{

if( a == 0.0 )

{ System.out.println("One root = " + (-c/b)); return;

}

double d = b\*b - 4.0\*a\*c;

if(d < 0) // Roots are imaginary

{

System.out.println("There are no real solutions."); return;

}

if(d == 0) // Roots are equal

{

System.out.println("Two roots are equal: " + (-b /(2.0\*a))); return;

}

// Roots are real

double x1 = (-b + Math.sqrt(d))/(2.0\*a); double x2 = (-b - Math.sqrt(d))/(2.0\*a);

System.out.println("Roots are: " + x1 + ", " + x2);

}

}

//////////////////// **QuadraticEquationDemo.java** ///////////////// class QuadraticEquationDemo

{

public static void main(String[] args)

{

Scanner scr = new Scanner(System.in);

System.out.print(" a = "); double a = scr.nextDouble(); System.out.print(" b = "); double b = scr.nextDouble(); System.out.print(" c = "); double c = scr.nextDouble();

QuadraticEquation qe = new QuadraticEquation(a, b, c);

qe.roots();

}

}

Output of this program is as follows (for different values of a, b, and c):

a = 0

b = 4

c = 1

One root = -0.25

a = 1

b = 4

c = 4

Two roots are equal: -2.0

a = 1

b = 4

c = 8

There are no real solutions

a = 1

b = 4

c = 3

Roots are: -1.0, -3.0

## Fibonacci sequence

1. The Fibonacci sequence is defined by the following rule: The first two values in the sequence are 0 and 1. Every subsequent value is the sum of the two values preceding it. Write a Java program that uses both recursive and non-recursive functions to print the *n*th value in the Fibonacci sequence.

The *Fibonacci sequence* (denoted by *f*0, *f*1, *f*2 …) is as follows: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 …

That is, *f*0 = 0 and *f*1 = 1 and each succeeding term is the sum of the two preceding terms. For example, the next two terms of the sequence are

34 + 55 = 89 and 55 + 89 = 144

Many algorithms have both iterative and recursive formulations. Typically, recursion is more elegant and requires fewer variables to make the same calculations. Recursion takes care of its book-keeping by stacking arguments and variables for each method call. This stacking of arguments, while invisible to the user, is still costly in time and space.

Fibonacci sequence is *recursively* defined by

*f*0 = 0, *f*1 = 1, *f*i+1 = *f*i + *f*i-1 for i = 1, 2 …

Fibonacci class is implemented in Program 2, with iterative and recursive methods, and tested by

main() driver.

Program 2: Fibonacci sequence

import java.io.\*;

class **Fibonacci**

{

public int fibonacciSeq(int n) // Iterative method

{

int term1 = 0, term2 = 1, nextTerm = 0; if( n == 0 || n == 1) return n;

int count = 2; while( count <= n )

{

nextTerm = term1 + term2; term1 = term2;

term2 = nextTerm; count++;

}

return nextTerm;

}

public int recFibonacciSeq(int n) // Recursive method

{

if( n == 0 || n == 1) return n; else

return( recFibonacciSeq(n-1) + recFibonacciSeq(n-2) );

}

}

/////////////////// **FibonacciDemo.java** //////////////////////// class FibonacciDemo

{

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

{

Fibonacci fib = new Fibonacci();

BufferedReader kb = new

BufferedReader(new InputStreamReader(System.in));

// nth value in the Fibonacci sequence

System.out.print("Enter n: ");

int n = Integer.parseInt(kb.readLine());

System.out.println("Iterative method: Fibonacci number "

+ n + " is " + fib.fibonacciSeq(n) );

System.out.println("Recursive method: Fibonacci number "

+ n + " is " + fib.recFibonacciSeq(n) );

}

}

Output of this program is:

Enter n: 12

Iterative method: Fibonacci number 12 is 144 Recursive method: Fibonacci number 12 is 144

## Prime numbers

1. Write a Java program that prompts the user for an integer and then prints out all prime numbers up to that integer.

*A prime number is a positive integer that is exactly divisible only by 1 and itself*. The first few prime numbers are:

2 3 5 7 11 13 17 23 29 31 37 …

All primes apart from 2 are odd.

As a starting point in developing the prime number generator let us explore how we can establish whether or not a particular number is a prime. To do this, let us consider a number 13. The definition of prime number suggests that to determine whether or not 13 is prime, we need to divide it in turn by the set of numbers 2, 3, 4, 5 …12. If any of these numbers divide into 13 without remainder we will know it cannot be prime number. Therefore, to test an integer *n* for prime, *n*-2 calls to the *mod* operation are required. As our *n* is 13, we need 11 calls. As *n* grows, the cost of making these divisions and tests is going to get very expensive. We must therefore look for ways of improving the efficiency of the algorithm. Firstly, we can try to keep to a minimum the number of numbers that we have to test for primes, and secondly we can try to improve the efficiency of testing a number for prime.

Following up these suggestions, we know that apart from 2, we do not need to examine *any* of the *even* numbers, and only first half of the numbers is enough for testing prime. That is, we can test for *mod* operation with numbers, from 3 to *n*/2. Hence, the following set of numbers is sufficient to test 13 for prime.

3 4 5 6

Program 3: Prime numbers

import java.io.\*;

class **PrimeNumber**

{

public void primes(int n)

{

int k, m;

System.out.print("Prime numbers up to " + n + ": 2 3"); for(m=3; m <= n; m = m+2)

{

boolean isPrime = false;

for(k=2; k <= m/2; k++)

{

if(m % k == 0) { isPrime = false; break; } else isPrime = true;

}

if(isPrime) System.out.print(" " + m);

}

}

}

////////////////////////// **PrimeNumberDemo.java** ///////////////// class PrimeNumberDemo

{

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

{

PrimeNumber pn = new PrimeNumber(); BufferedReader kb = new

BufferedReader(new InputStreamReader(System.in));

System.out.print("Enter n: ");

int n = Integer.parseInt(kb.readLine());

pn.primes(n);

}

}

Output:

Enter n: 50

Prime numbers up to 50: 2 3 5 7 11 13 17 19 23 29 31 37

41 43 47

## Palindrome

1. Write a Java program that checks whether a given string is a palindrome or not. For example, “MADAM” is a palindrome.

A string is an array of characters. For example, the string “MADAM” is stored as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| [0] | [1] | [2] | [3] | [4] |
| M | A | D | A | M |

Java supports the manipulation of character data through the primitive data type char and the associated operations for the input, output, assignment, and comparison of characters. Most applications of character data require character sequences - or *strings* - rather than individual characters. In Java a string is represented by the built-in class String. However, manipulating a String data type in Java is quite different from manipulating a set (or array) of characters. In the Java, strings are objects. The Java platform provides the String class to create and manipulate strings. The most direct way to create a string is to write:

String str = "MADAM";

In this case, "MADAM" is a *string literal* - a series of characters that is enclosed in double quotes. The String class provides the method length(), which returns as an int value the number of characters in the String object. The method of the String class

char charAt(int n)

returns the *n*th character in a string (where *n* must be less than string length);

If the first character is equal to the last character of the string, second character is equal to the second one from the last character of the string, and so on, then the string is a *palindrome*.

Program 4: Palindrome

class **Palindrome**

{

public boolean isPalindrome(String str)

{

System.out.print("Given string: " + str);

int n = str.length(); boolean flag = true;

for( int i=0; i < n/2; i++ )

if( str.charAt(i) != str.charAt(n-i-1) ) flag = false; return flag;

}

}

////////////////////////// **PalindromeDemo.java** //////////////////// class PalindromeDemo

{

public static void main(String[] args)

{

Palindrome pal = new Palindrome(); String str = "MADAM";

if( pal.isPalindrome(str)) System.out.print(" is a palindrome.");

else

System.out.print(" is not a palindrome.");

}

}

Output of this program is: (*Note*: You may test the program for different character strings).

Given string: MADAM is a palindrome.

## Sorting names

1. Write a Java program for sorting a given list of names in ascending order.

Bubble sort algorithm is used to sort the names.

Program 5: Sorting strings

class **SortNames**

{

public void sort(String[] a)

{

int i, pass, n = a.length; String tmp;

for( pass = 0; pass < n; pass++ )

{

for( i = 0; i < n-pass-1; i++ )

if( ((Comparable)a[i]).compareTo(a[i+1]) > 0)

{

tmp = a[i]; // Swap a[i], a[i+1] a[i] = a[i+1];

a[i+1] = tmp;

}

}

}

}

////////////////////////// **SortNamesDemo.java** //////////////////// class SortNamesDemo

{

public static void main(String[] args)

{

SortNames sn = new SortNames();

String[] names =

{"Ramu", "John", "Anu", "Priya", "Basha", "Prem"};

int i;

System.out.println("Unsorted names:");

for(i=0; i < names.length; i++) System.out.println(names[i]);

sn.sort(names);

System.out.println("Sorted names:"); for(i=0; i < names.length; i++)

System.out.println(names[i]);

}

}

Output of this program is:

Unsorted names: Ramu

John Anu Priya Basha Prem

Sorted names: Anu

Basha John Prem Priya Ramu

## Matrix multiplication

1. Write a Java program to multiply two given matrices.

If the order of matrix *A* is m x n and of matrix *B* is n x p (number of columns of *A* = number of rows of *B* = n), then the order of matrix *C* is m x p, where *C* = *A* x *B*.

k=n-1

cij = ∑aikbkj for i = 0 to m, and j = 0 to p

k=0

*Example*: The following two matrices *A* and *B* are used in the program to multiply *A* by *B*.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 3 | 2 | 0 |  | 2 | 0 | 1 | 1 |
| *A* = | 1 | 5 | 1 | *B* = | 1 | 3 | 0 | 3 |
|  | 2 | 3 | 4 |  | 2 | 2 | 2 | 1 |

Program 6: Matrix multiplication

class **MatrixMultiplication**

{

public void matrixMult( int[][] a, int[][] b, int[][] c)

{

int i, j, k;

for ( i = 0; i < 3; i++ ) for ( j = 0; j < 4; j++ )

{ c[i][j] = 0;

for ( k = 0; k < 3; k++ )

c[i][j] = c[i][j] + a[i][k] \* b[k][j];

}

}

}

////////////////////////// **MatrixMultiplicationDemo.java** ////////// class MatrixMultiplicationDemo

{

public static void main(String[] args)

{

int[][] a = { {3, 2, 0}, {1, 5, 1}, {2, 3, 4} };

int[][] b = { {2, 0, 1, 1}, {1, 3, 0, 3}, {2, 2, 2, 1} };

int[][] c;

c = new int[3][4]; int i, j;

MatrixMultiplication mm = new MatrixMultiplication(); mm.matrixMult(a,b,c);

System.out.println("matrix C (3 x 4)= A x B :"); for( i = 0; i < 3; i++ )

{ System.out.println(); for( j = 0; j < 4; j++ )

System.out.print( c[i][j] + "\t" );

}

}

}

This program generated the following output:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| matrix | C (3 | x | 4)= | A | x | B : |
| 8 | 6 |  | 3 |  |  | 9 |
| 9 | 17 |  | 3 |  |  | 17 |
| 15 | 17 |  | 10 |  |  | 15 |

## StringTokenizer

1. Write a Java Program that reads a line of integers, and then displays each integer, and the sum of all the integers (use **StringTokenizer** class).

The processing of data often consists of parsing a formatted input data (or string). The string tokenizer class allows an application to break a string into tokens. The discrete parts of a string are called *tokens*. The StringTokenizer methods do not distinguish among identifiers, numbers, and quoted strings, nor do they recognize and skip comments. The set of *delimiters* (the characters that separate tokens) may be specified either at creation time or on a per-token basis. StringTokenizer implements the Enumeration interface. Therefore, given an input string, we can enumerate the individual tokens contained in it using StringTokenizer. The default delimiters are whitespace

characters: space, tab, newline, and carriage return. Other than default delimiters, each character in the delimiters string is considered a valid delimiter.

A StringTokenizer object internally maintains a current position within the string to be tokenized. Some operations advance this current position past the characters processed. A token is returned by taking a substring of the string that was used to create the StringTokenizer object. The following is one example of the use of the tokenizer. The code:

StringTokenizer st = new StringTokenizer("this is a test"); while (st.hasMoreTokens())

{

System.out.println(st.nextToken());

}

prints the following output: (Here, *space* is the default delimiter)

this is

a test

StringTokenizer is a legacy class that is retained for compatibility reasons although its use is discouraged in new code.

StringTokenizer constructors are:

StringTokenizer(String *str*)

StringTokenizer(String *str*, String *delimiters*)

Program 7: StringTokenizer

import java.util.StringTokenizer; import java.io.\*;

class **StringTokenizerDemo**

{

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

{

BufferedReader kb = new

BufferedReader(new InputStreamReader(System.in));

System.out.print("Enter a string of digits: "); String str = kb.readLine();

StringTokenizer st = new StringTokenizer(str); int a, sum = 0;

String s;

while (st.hasMoreTokens())

{

s = st.nextToken();

a = Integer.parseInt(s); sum = sum + a;

System.out.println(a);

}

System.out.println("Sum of these integers = " + sum);

}

}

This program prints the following output:

Enter a string of digits: 45 100 750 5

45

100

750

5

Sum of these integers = 900

## File class

1. Write a Java program that reads a file name from the user then displays information about whether the file exists, whether the file is readable, whether the file is writable, the type of file and the length of the file in bytes.

Most of the classes defined by **java.io** operate on streams. The **File** class deals directly with files and the file system. That is, the **File** class does not specify how data is retrieved from or stored in files; it describes the properties of a file itself. A **File** object is used to get information or manipulate the information associated with a disk file; such as the permissions, time, date, directory path, is it readable or writable, file size, and so on.

Files are a primary source and destination for data within many programs. Files are central resource for storing persistent and shared information.

The following two constructors can be used to create **File** objects (we have two more constructors):

File(String *directoryPath*)

File(String *directoryPath*, String *fileName*)

Here, *directoryPath* is the path name of the file, *fileName* is the name of the file.

**File** class defines many methods that obtain the standard properties of a **File** object. For example, **getName()** returns the file name, and **exists()** returns true if the file exists, false if it does not. The **File** class has many methods that allow us to examine the properties of a sample file object. Program 8 illustrates some of the **File** methods. In this example program, an existing “**HashSetDemo.java**” file is used.

Many times a program will be developed that requires the user to enter a small amount of information at the terminal keyboard. The information may consist of number of data elements to be used in an array. Rather than having the program request this type of data from the user, we can supply the information to the program at the time of program execution. This capability is provided by what is known as *command line arguments*.

The **main(String[] args)** is a method that has arguments. The parameter **args** stands for *argument vector*. It is an array of strings. The successive elements of the array refer to successive words in the command line. We can pass arguments to **main()** method. Java provides a connection to the arguments on the command line. We illustrate “passing command line arguments to **main()**” through the following example.

**D:\Java>java FileDemo HashSetDemo.java**

The string “**HashSetDemo.java**” is stored in **args[0]**. If we supply second string on command line, it will be stored in **args[1]**, and so on.

Program 8: Demonstration of File class

import java.io.File; class **FileDemo**

{

public static void main(String[] args)

{

File f = new File(args[0]);

System.out.println("File name: " + f.getName()); System.out.println(f.exists() ? "exists" : "does not exist"); System.out.println(f.canWrite() ? "is writable" : "is not writable"); System.out.println(f.canRead() ? "is readable" : "is not readable"); System.out.println(f.isFile()? "is normal file" : "might be a pipe");

System.out.println("File size (in bytes): " + f.length());

}

}

This program, **FileDemo** is executed as follows:

**D:\Java>java FileDemo HashSetDemo.java**

Output of this program is:

File name: HashSetDemo.java exists

is writable is readable

is normal file

File size (in bytes): 258

*Note*: Most of the **File** methods are self-explanatory.

## FileInputStream class

1. Write a Java program that reads a file and displays the contents of the file on the screen, with a line number before each line.

The **FileInputStream** class creates an **InputStream** that we can use to read bytes from a file. The most common constructors are:

FileInputStream(String *filePath*)

FileInputStream(String *fileObject*)

Here, *filePath* is the full path name of a file, and *fileObject* is a **File** object that describes the file. When a **FileInputStream** is created, it is opened for reading. For example,

InputStream **f** = new FileInputStream("HashSetDemo.java");

creates a **File** object, **f** for reading. This object refers to the file: "HashSetDemo.java". The following program demonstrates the FileInputStream.

Program 9: FileInputStream

import java.io.\*;

class FileInputStreamDemo

{

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

{

InputStream f = new FileInputStream("HashSetDemo.java");

int size = f.available(); // file size in bytes

int lineNo = 0; char ch;

System.out.print(++lineNo + ": "); // print line 1:

ch = (char)f.read(); // read first character of line 1

System.out.print(ch); // print first character of line 1

for(int i=1; i < size; i++) // read next character & print

{

if( ch == '\n' ) // if it is a newline,

System.out.print(++lineNo + ": "); // print line number

ch = (char)f.read(); // read next character

System.out.print(ch); // print next character

}

}

}

Output of this program is:

1: import java.util.\*;

2: public class HashSetDemo

3: { public static void main(String[] args) 4: {

5: HashSet<Integer> hs = new HashSet<Integer>(); 6:

7: for( int k = 1; k <= 5; k++ ) 8: hs.add(11\*k);

9:

10: System.out.println("HashSet: " + hs); 11: }

12: }

## Counting characters, words, and lines in a text file

1. Write a Java program that displays the number of characters, lines and words in a text file.

Program 10: Number of characters, lines and words in a text file

import java.io.\*; class TextFileDemo

{

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

{

InputStream f = new FileInputStream("MYDATA.TXT");

int size = f.available(); int nLines = 0;

int nWords = 0; char ch;

System.out.println("Data file: MYDATA.TXT

contains the following text:\n");

for(int i=0; i < size; i++)

{

ch = (char)f.read(); System.out.print(ch);

if( ch == **' '** ) nWords++; if( ch == **'**\n**'** ) nLines++;

}

nWords++; // to count the *last word* in the text file System.out.println("\nNo: of characters = " + size); System.out.println("No: of words = " + nWords); System.out.println("No: of lines = " + nLines);

}

}

The following output is generated by this program:

Data file: MYDATA.TXT contains the following text:

Real-life applications require that the data be written to or read from secondary storage devices. The secondary storage devices are

also called auxiliary storage devices such as hard disk, compact disk, floppy, and magnetic tape. The data is permanently stored on these devices in the form of data files. Once data is available on these devices, we can access and change it whenever required.

No: of characters = 405 No: of words = 64

No: of lines = 6

## Multiple Threads

1. Write a Java program for creating multiple threads.
   1. Using Thread class
   2. Using Runnable interface

***Refer*** Chapter 6: Multi-Threading to understand thread class, Runnable interface and multiple threads.

Program 11: Multiple threads using Runnable interface

// create multiple threads

class AThread implements Runnable

{ String name; Thread th;

AThread( String threadName)

{ name = threadName;

th = new Thread(this, name); System.out.println("A Thread: " + th); th.start();

}

public void run() // entry point for the thread

{ try

{ for(int k = 3; k > 0; k--)

{ System.out.println(name + ":" + k); Thread.sleep(1000);

}

} catch(InterruptedException e)

{ System.out.println(name + "Interrupted"); } System.out.println(name + " exiting");

}

}

/////////////////// **MultiThreadDemo.java** //////////////////// class MultiThreadDemo

{ public static void main(String[] args)

{ new AThread("One"); // start threads

new AThread("Two"); new AThread("Three"); try

{ Thread.sleep(10000); // wait for others to end

} catch(InterruptedException e)

{ System.out.println("Main thread Interrupted"); } System.out.println("Main thread exiting");

}

}

Output:

A Thread: Thread[One,5,main] A Thread: Thread[Two,5,main] One:3

Two:3

1. Thread: Thread[Three,5,main] Three:3

One:2 Two:2 Three:2 One:1 Two:1 Three:1

One exiting Two exiting Three exiting

Main thread exiting

## Run-time Polymorphism

1. Write a Java program that illustrates how run time polymorphism is achieved.

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be *hidden*.

Method overriding forms the basis for one of Java’s most powerful concepts: ***dynamic method dispatch***. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run-time, rather than compile-time. Dynamic method dispatch is important because this is how Java implements ***run-time polymorphism***.

A superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run-time. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of object being referred to at run-time the call occurs. When different types of objects are referred to, different versions of an overridden method will be called. Here is an example that illustrates dynamic method dispatch:

Program 12: Run-time polymorphism through dynamic dispatch

class Animal

{ void callme()

{ System.out.println("It is an Animal."); }

}

class Cat extends Animal

{ void callme()

{ System.out.println("It is a Cat."); }

}

class Dog extends Animal

{ void callme()

{ System.out.println("It is a Dog."); }

}

class DynamicDispatch

{

public static void main(String[] args)

{

Animal a = new Animal(); // create Animal object, a Cat c = new Cat(); // create Cat object, c

Dog d = new Dog(); // create Dog object, d

|  |  |  |
| --- | --- | --- |
|  | Animal ref; | // obtain a reference of type Animal |
| ref = a; | // “ref” refers to Animal object |
| ref.callme(); | // calls Animal’s version of callme() |
| ref = c; | // “ref” refers to Cat object |
| ref.callme(); | // calls Cat’s version of callme() |
| ref = d; | // “ref” refers to Dog object |
| } | ref.callme(); | // calls Dog’s version of callme() |
| } |  |  |

Output of this program is:

It is an Animal. It is a Cat.

It is a Dog.

## Packages and Interfaces

1. Write a java program that illustrates the following:
   1. Creation of simple package.
   2. Accessing a package.
   3. Implementing interfaces.

Packages are a feature of Java that helps you to organize and structure your classes and their relationships to one another. A ***package*** is a grouping of related types providing access protection and name space management. Note that *types* refer to classes, interfaces and others. The types that are part of the Java platform are members of various packages that bundle classes by function: fundamental classes are in java.lang, classes for reading and writing (input and output) are in java.io, and so on. You can put your types in packages too.

For example, consider the following package specification:

package MyPack;

In order for a program to find MyPack, we use one of the two alternatives:

1. Program is executed from a directory immediately above MyPack, or
2. CLASSPATH must be set to include the path to MyPack. SET CLASSPATH =C:\MyPack\

We create a directory, MyPack below the current development directory, put the .class files into the MyPack directory, and then execute the program from the development directory.

Program 13(a): A simple package example

package MyPack; // A simple package

class Balance

{

String name; double bal;

Balance(String s, double b) // constructor

{ name = s; bal = b;

}

void show() // method

{ if(bal < 0) System.out.print("-->> ");

System.out.println(name + ": Rs" + bal);

}

}

class AccountBalance

{

public static void main(String args[])

{ Balance current[] = new Balance[3];

current[0] = new Balance("R. Lepakshi", 15230.50); current[1] = new Balance("K. Siva Kumar", 350.75); current[2] = new Balance("Michael Jackson", -120.30);

for(int i = 0; i < 3; i++) current[i].show();

}

}

Let the current development directory be C:\java

Create (make directory: md) MyPack directory as follows:

C:\java>md MyPack

Edit the AccountBalance.java program and compile it from current development directory as follows:

C:\java>edit AccountBalance.java C:\java>javac AccountBalance.java

Then, copy Balance.class and AccountBalance.class files from the directory C:\java to the directory C:\java\MyPack

Execute the AccountBalance class, using the following command line:

C:\java>java MyPack.AccountBalance

AccountBalance is now part of the package MyPack. This means that it cannot be executed by itself. That is, we cannot use this command line:

C:\java>java AccountBalance AccountBalance must be qualified with its package name.

In Java, an ***interface*** is a reference type, similar to a class that can contain *only* constants, method signatures, and nested types. There are no method bodies. Interfaces cannot be instantiated - they can only be *implemented* by classes or *extended* by other interfaces.

Here is an example of an interface definition (defining an interface is similar to creating a new class). Program 13(b) declares an interface called **Items** which contains four methods. Note that the method signatures have no braces and are terminated with a semicolon.

Program 13(b): Defining an interface Items

interface Items

{ boolean equal(); // test whether two numbers are equal

int add(); // sum of two numbers

int max(); // maximum of two numbers

void display(); // print two numbers

}

To declare a class that implements an interface, you include an **implements** clause in the class declaration. Your class can implement more than one interface, so the implements keyword is followed by a comma-separated list of the interfaces implemented by the class. When an instantiable class implements an interface, it provides a method body for each of the methods declared in the interface. Here is an example class **Client** that implements the **Items** interface.

Program 13(c): Implementing an interface Items

class **Client** implements **Items**

{

int a, b;

Client( int i, int j) // constructor

{ a = i; b = j; }

public boolean equal()

{ return (a == b); }

public int add()

{ return(a + b); }

public int max()

{ if( a > b ) return a; else return b;

}

public void display()

{ System.out.println(a + " " + b); }

public int multiply() // non-interface method

{ return(a \* b); }

}

Program 13(d): Testing a class which implements an interface

class ClientDemo

{ public static void **main**(String[] srgs)

{

Client ob = new Client(5, 7); ob.display();

if( ob.equal())

System.out.println("Two numbers are equal"); else

System.out.println("Two numbers are not equal");

System.out.println("Sum of two numbers is " + ob.add()); System.out.println("Max of two numbers is " + ob.max()); System.out.println("Product of two numbers is " + ob.multiply());

}

}

Output of this program is:

5 7

Two numbers are not equal Sum of two numbers is 12 Max of two numbers is 7

Product of two numbers is 35

## Exception Handling

1. Write a java program that illustrates the following
   1. Handling predefined exceptions
   2. Handling user defined exceptions
2. Java uses ***exceptions*** to handle errors and other exceptional events. An *exception* (error) is an event, which occurs during the execution of a program, which disrupts the normal flow of the program instructions. When an error occurs within a method, the method creates an object and hands it off to

the runtime system. The object, called an *exception object*, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called *throwing an exception*.

The first step in constructing an exception handler is to enclose the code that might throw an exception within a **try** block. To illustrate how this can be done, the following program includes a try block and a catch clause which processes the ArithmeticException generated by “division- by-zero” error. This is a RuntimeException, predefined in java.lang.

Program 14(a): Handling predefined exceptions

class ExceptionDemo

{

public static void main(String args[])

{

int a = 25, c = 3, d = 0;

try

{

a = 25 / d; System.out.println(a);

} catch(ArithmeticException ae)

{

System.out.println(ae);

}

System.out.println(a\*c);

}

}

This program generates the following output:

java.lang.ArithmeticException: / by zero 75

Notice that the call to println(a) inside the try block is never executed. Once an exception is thrown, program control transfers out of the try block into the catch block. After displaying error message, it executes the println(a\*c).

1. Although there are a number of exceptions in the Java class library that you can use in your own methods; but you might need to create your own exceptions to handle the different kinds of errors in your program. Creating new exceptions is easy.

Your new exception should inherit from some other exception in the Java hierarchy. All user- created exceptions should be part of the Exception hierarchy. Look for an exception that is close to the one you are creating. For example, an exception for a bad file format would logically be an IOException. If you cannot find a closely related exception for your new exception, consider inheriting from Exception, which forms the top of exception hierarchy.

The Exception class does not define any methods of its own. It inherits those methods provided by Throwable. One such method is the toString() which returns a String object containing a description of the exception (notice the MyException class in the following program).

The following program illustrates the user-defined exception.

Program 14(b): Handling user defined exceptions

class **MyException** extends **Exception**

{

public String toString()

{

String str = "MyException (a/d): division by zero"; return str;

}

}

class **MyExceptionDemo**

{

public static void main(String args[]) throws MyException

{

int a = 20, b = 3, d = 2;

System.out.println("a = " + a + ", b = " + b + ", d = " + d);

try

{

if( d != 0)

System.out.println( "a/d = " + a/d ); else

throw new MyException(); System.out.println("Normal exit");

} catch (MyException e)

{ System.out.println(e); }

System.out.println( "a\*b = " + a\*b );

}

}

Output of this is as follows, when d = 0:

a = 20, b = 3, d = 0

MyException (a/d): division by zero a\*b = 60

when d = 2:

a = 20, b = 3, d = 2

a/d = 10 Normal exit a\*b = 60

# Searching

1. Write Java programs that use both recursive and non-recursive functions for implementing the following searching methods:
   1. Linear search
   2. Binary search

## Linear search

1. The simplest form of a search is the ***linear search***. This technique is meant for searching a particular item in an unsorted data set in a sequential manner until the desired data item is found. Linear search is easy to write and efficient for short lists, but inefficient for long ones. To find any element in a long array (list), there are far more efficient methods, provided that the array is sorted. A program for linear search is as follows.

Program 15(a): Iterative Linear search

class LinearSearchDemo

{

static Object[] a = { 89, 45, 175, 7, 50, 43, 126, 90 };

static Object key = 126;

public static void main(String args[])

{

if( linearSearch() )

System.out.println(key + " found in the list"); else

System.out.println(key + " not found in the list");

}

static boolean linearSearch()

{

for( int i=0; i<a.length; i++) if(key == a[i]) return true;

return false;

}

}

Program 15(b): Recursive Linear search

class RecursiveLinearSearchDemo

{

static Object[] a = { 89, 45, 175, 7, 50, 43, 126, 90 };

static Object key = 43;

public static void main(String args[])

{

if( linearSearch(a.length-1) ) System.out.println(key + " found in the list");

else

System.out.println(key + " not found in the list");

}

static boolean linearSearch(int n)

{

if( n < 0 ) return false;

2. Searching 23

if(key == a[n]) return true;

else

return linearSearch(n-1);

}

}

## Binary search

1. ***Binary search*** is a simple method of accessing a particular item in a sorted (ordered) data set. A search for a particular item with a certain key value resembles the search for a name in telephone directory or a word in a dictionary. The approximate middle item of the data set is located, and its key value is examined. If its value is too high, then the key of the middle element of the first half of the set is examined and procedure is repeated on the first half until the required item is found. If the value is too low, then the key of the middle entry of the second half of the data set is tried and the procedure is repeated on the second half. This process continues until the desired key is found or search interval becomes empty. The binary search algorithm is based on binary search tree.

Program 15(c): Iterative Binary search

class BinarySearchDemo

{

static Object[] a = { "AP", "KA", "MH", "MP", "OR", "TN", "UP", "WB"};

static Object key = "UP";

public static void main(String args[])

{

if( binarySearch() )

System.out.println(key + " found in the list"); else

System.out.println(key + " not found in the list");

}

static boolean binarySearch()

{

int c, mid, low = 0, high = a.length-1;

while( low <= high)

{

mid = (low + high)/2;

c = ((Comparable)key).compareTo(a[mid]); if( c < 0) high = mid-1;

else if( c > 0) low = mid+1; else return true;

}

return false;

}

}

**Program 15(d): Recursive Binary search**

class RecursiveBinarySearchDemo

{

static Object[] a = { "AP", "KA", "MH", "MP", "OR", "TN", "UP", "WB"};

static Object key = "XP";

public static void main(String args[])

{

if( binarySearch(0, a.length-1) ) System.out.println(key + " found in the list");

else

System.out.println(key + " not found in the list");

}

static boolean binarySearch(int low, int high)

{

if( low > high ) return false; int mid = (low + high)/2;

int c = ((Comparable)key).compareTo(a[mid]);

if( c < 0) return binarySearch(low, mid-1);

else if( c > 0) return binarySearch(mid+1, high); else return true;

}

}

# Linked Lists

1. Write Java programs to implement the List ADT using arrays and linked lists.

## List ADT

The elements in a list are of generic type Object. The elements form a linear structure in which list elements follow one after the other, from the beginning of the list to its end. The list ADT supports the following operations:

**createList**(*int n*): Creates (initially) a list with *n* nodes.

*Input*: *integer*; *Output*: None

**insertFirst**(*obj*): Inserts object *obj* at the beginning of a list.

*Input*: Object; *Output*: None

**insertAfter**(*obj*, *obj p*): Inserts object *obj* after the *obj p* in a list.

*Input*: Object and position; *Output*: None

*obj* **deleteFirst**(): Deletes the object at the beginning of a list.

*Input*: None; *Output*: Deleted object *obj*.

*obj* **deleteAfter**(*obj p*): Deletes the object after the *obj p* in a list.

*Input*: Position; *Output*: Deleted object *obj*.

*boolean* **isEmpty**(): Returns a *boolean* indicating if the list is empty.

*Input*: None; *Output*: *boolean* (*true* or *false*). *int* **size**(): Returns the number of items in the list.

*Input*: None; *Output*: *integer*.

Type Object may be any type that can be stored in the list. The actual type of the object will be provided by the user. The ADT is translated into a Java interface in the following program.

Program 16(a): A **List** Interface

public interface **List**

{

public void **createList**(int n); public void **insertFirst**(Object ob);

public void **insertAfter**(Object ob, Object pos); public Object **deleteFirst**();

public Object **deleteAfter**(Object pos); public boolean **isEmpty**();

public int **size**();

}

#### Array Implementation of List

Program 16(b): An ArrayList Class

class **ArrayList** implements **List**

{

class **Node**

{ Object data; int next;

Node(Object ob, int i) // constructor

{ data = ob; next = i;

}

}

int MAXSIZE; // max number of nodes in the list

Node list[]; // create list array

int head, count; // count: current number of nodes in the list

ArrayList( int s) // constructor

{ MAXSIZE = s;

list = new Node[MAXSIZE];

}

public void **initializeList**()

{ for( int p = 0; p < MAXSIZE-1; p++ ) list[p] = new Node(null, p+1);

list[MAXSIZE-1] = new Node(null, -1);

}

public void **createList**(int n) // create ‘n’ nodes

{ int p;

for( p = 0; p < n; p++ )

{

list[p] = new Node(11+11\*p, p+1); count++;

}

list[p-1].next = -1; // end of the list

}

public void **insertFirst**(Object item)

{

if( count == MAXSIZE )

{ System.out.println("\*\*\*List is FULL"); return;

}

int p = getNode(); if( p != -1 )

{

list[p].data = item;

if( isEmpty() ) list[p].next = -1; else list[p].next = head;

head = p; count++;

}

}

public void **insertAfter**(Object item, Object x)

{

if( count == MAXSIZE )

{ System.out.println("\*\*\*List is FULL"); return;

}

int q = getNode(); // get the available position to insert new node int p = find(x); // get the index (position) of the Object x if( q != -1 )

{ list[q].data = item; list[q].next = list[p].next; list[p].next = q;

count++;

}

}

public int **getNode**() // returns available node index

{ for( int p = 0; p < MAXSIZE; p++ ) if(list[p].data == null) return p;

return -1;

}

public int **find**(Object ob) // find the index (position) of the Object ob

{ int p = head; while( p != -1)

{ if( list[p].data == ob ) return p;

p = list[p].next; // advance to next node

}

return -1;

}

public Object **deleteFirst**()

{ if( isEmpty() )

{ System.out.println("List is empty: no deletion"); return null;

}

Object tmp = list[head].data;

if( list[head].next == -1 ) // if the list contains one node,

head = -1; // make list empty.

else

head = list[head].next;

count--; // update count

return tmp;

}

public Object **deleteAfter**(Object x)

{ int p = find(x);

if( p == -1 || list[p].next == -1 )

{ System.out.println("No deletion"); return null;

}

int q = list[p].next; Object tmp = list[q].data;

list[p].next = list[q].next; count--;

return tmp;

}

public void **display**()

{ int p = head; System.out.print("\nList: [ " ); while( p != -1)

{ System.out.print(list[p].data + " "); // print data

p = list[p].next; // advance to next node

}

System.out.println("]\n");//

}

public boolean **isEmpty**()

{ if(count == 0) return true; else return false;

}

public int **size**()

{ return count; }

}

Program 16(c): Testing ArrayList Class

class ArrayListDemo

{

public static void main(String[] args)

{

ArrayList linkedList = new ArrayList(10);

linkedList.initializeList(); linkedList.createList(4); // create 4 nodes

linkedList.display(); // print the list

System.out.print("InsertFirst 55:"); linkedList.insertFirst(55); linkedList.display();

System.out.print("Insert 66 after 33:");

linkedList.insertAfter(66, 33); // insert 66 after 33

linkedList.display();

Object item = linkedList.deleteFirst(); System.out.println("Deleted node: " + item);

linkedList.display();

System.out.print("InsertFirst 77:"); linkedList.insertFirst(77);

linkedList.display();

item = linkedList.deleteAfter(22); // delete node after node 22

System.out.println("Deleted node: " + item); linkedList.display();

System.out.println("size(): " + linkedList.size());

}

}

The following output is generated by this program:

InsertFirst 77:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| List: [ 11 22 33 44  InsertFirst 55: | | | | ] |  |
| List: [ 55 11 22 33  Insert 66 after 33:  List: [ 55 11 22 33 | | | | 44  66 | ]  44 ] |
| Deleted node: 55  List: [ 11 22 33 66 | | | | 44 | ] |
| List: [ | 77 11 | 22 | 33 | 66 | 44 ] |
| Deleted | node: | 33 |  |  |  |
| List: [ | 77 11 | 22 | 66 | 44 | ] |
| size(): | 5 |  |  |  |  |

#### Linked Implementation of List

**LinkedList** class implemented by **List** interface is given Program 16(d) and it is tested in Program 16(e).

Program 16(d): A LinkedList Class

class **LinkedList** implements **List**

{

class **Node**

{ Object data; // data item

Node next; // refers to next node in the list

Node( Object d ) // constructor

{ data = d; } // ‘next’ is automatically set to null

}

Node head; // head refers to first node Node p; // p refers to current node int count; // current number of nodes

public void **createList**(int n) // create 'n' nodes

{

p = new Node(11); // create first node

head = p; // assign mem. address of 'p' to 'head'

for( int i = 1; i < n; i++ ) // create 'n-1' nodes

p = p.next = new Node(11 + 11\*i); count = n;

}

public void **insertFirst**(Object item) // insert at the beginning of list

{

p = new Node(item); // create new node

p.next = head; // new node refers to old head

head = p; // new head refers to new node

count++;

}

public void **insertAfter**(Object item,Object key)

{

p = find(key); // get “location of key item”

if( p == null )

System.out.println(key + " key is not found"); else

{ Node q = new Node(item); // create new node

q.next = p.next; // new node next refers to p.next

p.next = q; // p.next refers to new node

count++;

}

}

public Node **find**(Object key)

{

p = head;

while( p != null ) // start at beginning of list until end of list

{

if( p.data == key ) return p; // if found, return key address

p = p.next; // move to next node

}

return null; // if key search is unsuccessful, return null

}

public Object **deleteFirst**() // delete first node

{

if( isEmpty() )

{ System.out.println("List is empty: no deletion"); return null;

}

Node tmp = head; // tmp saves reference to head head = tmp.next;

count--;

return tmp.data;

}

public Object **deleteAfter**(Object key) // delete node after key item

{ p = find(key); // p = “location of key node”

if( p == null )

{ System.out.println(key + " key is not found"); return null;

}

if( p.next == null ) // if(there is no node after key node)

{ System.out.println("No deletion"); return null;

}

else

{ Node tmp = p.next; // save node after key node p.next = tmp.next; // point to next of node deleted count--;

return tmp.data; // return deleted node

}

}

public void **displayList**()

{ p = head; // assign mem. address of 'head' to 'p'

System.out.print("\nLinked List: ");

while( p != null ) // start at beginning of list until end of list

{ System.out.print(p.data + " -> "); // print data

p = p.next; // move to next node

}

System.out.println(p); // prints 'null'

}

public boolean **isEmpty**() // true if list is empty

{ return (head == null); }

public int size()

{ return count; }

} // end of LinkeList class

Program 16(e): Testing LinkedList Class

class LinkedListDemo

{ public static void **main**(String[] args)

{ LinkedList list = new LinkedList(); // create list object

list.createList(4); // create 4 nodes

list.displayList();

list.insertFirst(55); // insert 55 as first node

list.displayList();

list.insertAfter(66, 33); // insert 66 after 33

list.displayList();

Object item = list.deleteFirst(); // delete first node

if( item != null )

{ System.out.println("deleteFirst(): " + item); list.displayList();

}

item = list.deleteAfter(22); // delete a node after node(22)

if( item != null )

{ System.out.println("deleteAfter(22): " + item); list.displayList();

}

System.out.println("size(): " + list.size());

}

}

Here is the output from LinkedListDemo.java: Linked List: 11 -> 22 -> 33 -> 44 -> null

Linked List: 55 -> 11 -> 22 -> 33 -> 44 -> null

Linked List: 55 -> 11 -> 22 -> 33 -> 66 -> 44 -> null

deleteFirst(): 55

Linked List: 11 -> 22 -> 33 -> 66 -> 44 -> null

deleteAfter(22): 33

Linked List: 11 -> 22 -> 66 -> 44 -> null

size(): 4

# Stacks and Queues

1. Write Java programs to implement the following using an array.
   1. Stack ADT
   2. **Queue ADT**

**Stack ADT**

A Stack is an Abstract Data Type (ADT) that supports the following methods:

**push**(*obj*): Add object *obj* at the top of the stack.

*Input*: Object; *Output*: None.

*obj* **pop**(): Delete an item from the top of the stack and returns object *obj*; an error occurs if the stack is empty.

*Input*: None; *Output*: Object.

*obj* **peek**(): Returns the top object *obj* on the stack , without removing it; an error occurs if the stack is empty.

*Input*: None; *Output*: Object.

*boolean* **isEmpty**(): Returns a *boolean* indicating if the stack is empty.

*Input*: None; *Output*: *boolean* (*true* or *false*). *int* **size**(): Returns the number of items on the stack.

*Input*: None; *Output*: *integer*.

Type Object may be any type that can be stored in the stack. The actual type of the object will be provided by the user. The ADT is translated into a Java interface in Program 17(a).

Program 17(a): A Stack Interface

public interface **Stack**

{ public void **push**(Object ob); public Object **pop**();

public Object **peek**(); public boolean **isEmpty**();

public int **size**();

}

The *push*, *pop*, *peek*, *empty*, and *size* operations are translated directly into specifications for methods named push(), pop(), peek(), isEmpty(), and size() respectively. These are conventional names for stack operations. Each method is defined by specifying its return value and any changes that it makes to the object.

**Stack Implementation**

There are several ways to implement the Stack interface. The simplest is to use an ordinary array. This is done in Program 17(b). The ArrayStack implementation uses an array a[] to store elements of the stack. Its other data field is the integer top, which refers top element of the stack. The top is also used to count the current number of items in the stack.

**Program 17(b): An ArrayStack Class**

public class **ArrayStack** implements **Stack**

{

private Object a[];

private int top; // stack top

public ArrayStack(int n) // constructor

{ a = new Object[n]; // create stack array

top = -1; // no items in the stack

}

public void **push**(Object item) // add an item on top of stack

{

if(top == a.length-1)

{ System.out.println("Stack is full"); return;

}

top++; // increment top

a[top] = item; // insert an item

}

public Object **pop**() // remove an item from top of stack

{

if( isEmpty() )

{ System.out.println("Stack is empty"); return null;

}

Object item = a[top]; // access top item

top--; // decrement top

return item;

}

public Object **peek**() // get top item of stack

{ if( isEmpty() ) return null; return a[top];

}

public boolean **isEmpty**() // true if stack is empty

{ return (top == -1); }

public int **size**() // returns number of items in the stack

{ return top+1; }

}

The constructor creates a new stack of a size, n specified in its argument. The variable top stores

the index of the item on the top of the stack.

The push() method increments top so it points to the space just above the previous top, and stores a data item there. Notice that top is incremented before the item is inserted. The pop() method returns the value at top and then decrements top. This effectively removes the item from the stack; it is inaccessible, although the value remains in the array (until another item is pushed into the cell). The peek() method simply returns the value at top, without changing the stack. The specifications for the pop() and peek() methods in the Stack interface require that the stack be not empty. The isEmpty() method returns true if the stack is empty. The top variable is at –1 if the stack is empty.

The ArrayStack class is tested in the Program 17(c).

**Program 17(c): Testing ArrayStack Class**

class ArrayStackDemo

{

public static void **main**(String[] args)

{

ArrayStack stk = new ArrayStack(4); // create stack of size 4

Object item;

stk.push('A'); // push 3 items onto stack

stk.push('B');

stk.push('C');

System.out.println("size(): "+ stk.size());

item = stk.pop(); // delete item

System.out.println(item + " is deleted");

stk.push('D'); // add three more items to the stack

stk.push('E');

stk.push('F');

System.out.println(stk.pop() + " is deleted"); stk.push('G'); // push one item

item = stk.peek(); // get top item from the stack

System.out.println(item + " is on top of stack");

}

}

Output of this program is:

size(): 3

C is deleted Stack is full E is deleted

G is on top of stack

## Queue ADT

The elements in a queue are of generic type Object. The queue elements are linearly ordered from the front to the rear. Elements are inserted at the rear of the queue (*enqueued*) and are removed from the front of the queue (*dequeued*). A Queue is an Abstract Data Type (ADT) that supports the following methods:

**insert**(*obj*): Adds object *obj* at the rear of a queue.

*Input*: Object; *Output*: None.

*obj* **remove**(): Deletes an item from the front of a queue and returns object *obj*; an error occurs if the queue is empty.

*Input*: None; *Output*: Object.

*obj* **peek**(): Returns the object *obj* at the front of a queue , without removing it; an error occurs if the queue is empty.

*Input*: None; *Output*: Object.

*boolean* **isEmpty**(): Returns a *boolean* indicating if the queue is empty.

*Input*: None; *Output*: *boolean* (*true* or *false*). *int* **size**(): Returns the number of items in the queue.

*Input*: None; *Output*: *integer*.

Type Object may be any type that can be stored in the queue. The actual type of the object will be provided by the user. The ADT is translated into a Java interface in Program 17(d).

Program 17(d): A Queue Interface

public interface **Queue**

{

public void **insert**(Object ob); public Object **remove**(); public Object **peek**();

public boolean **isEmpty**(); public int **size**();

}

Note the similarities between these specifications and that of the stack interface. The only real difference, between the names of the operations, is that the queue adds new elements at the opposite end from which they are accessed, while the stack adds them at the same end.

## Queue Implementation

The ArrayQueue implementation of queue interface is done by taking an array, *que*[*n*] and treating it as if it were *circular*. The elements are inserted by increasing *rear* to the next free position. When *rear*

= *n*-1, the next element is entered at *que*[0] in case that spot is free. That is, the element *que*[*n*-1] follows *que*[0]. Program 17(e) implements the ArrayQueue class, and Program 17(f) tests this class.

**Program 17(e): An ArrayQueue Class**

class **ArrayQueue** implements **Queue**

{ private int maxSize; // maximum queue size

private Object[] que; // que is an array

private int front; private int rear;

private int count; // count of items in queue (queue size)

public ArrayQueue(int s) // constructor

{ maxSize = s;

que = new Object[maxSize]; front = rear = -1;

count = 0;

}

public void **insert**(Object item) // add item at rear of queue

{

if( count == maxSize )

{ System.out.println("Queue is Full"); return; }

if(rear == maxSize-1 || rear == -1)

{ que[0] = item; rear = 0;

if( front == -1) front = 0;

}

else que[++rear] = item;

count++; // update queue size

}

public Object **remove**() // delete item from front of queue

{

if( isEmpty() )

{System.out.println("Queue is Empty"); return 0; }

Object tmp = que[front]; // save item to be deleted que[front] = null; // make deleted item’s cell empty if( front == rear )

rear = front = -1;

else if( front == maxSize-1 ) front = 0; else front++;

count--; // less one item from the queue size

return tmp;

}

public Object **peek**() // peek at front of the queue

{ return que[front]; }

public boolean **isEmpty**() // true if the queue is empty

{ return (count == 0); }

public int **size**() // current number of items in the queue

{ return count; }

public void **displayAll**()

{

System.out.print("Queue: ");

for( int i = 0; i < maxSize; i++ ) System.out.print( que[i] + " ");

System.out.println();

}

}

**Program 17(f): Testing ArrayQueue class**

class **QueueDemo**

{

public static void **main**(String[] args)

{

/\* queue holds a max of 5 items \*/ ArrayQueue q = new ArrayQueue(5); Object item;

q.insert('A'); q.insert('B'); q.insert('C'); q.displayAll();

item = q.remove(); // delete item System.out.println(item + " is deleted"); item = q.remove(); System.out.println(item + " is deleted"); q.displayAll();

q.insert('D'); // insert 3 more items

q.insert('E');

q.insert('F');

q.displayAll(); item = q.remove();

System.out.println(item + " is deleted"); q.displayAll();

System.out.println("peek(): " + q.peek()); q.insert('G');

q.displayAll();

System.out.println("Queue size: " + q.size());

}

}

Output of this program is as follows:

Queue: A B C null null A is deleted

1. is deleted

Queue: null null C null null Queue: F null C D E

1. is deleted

Queue: F null null D E peek(): D

Queue: F G null D E Queue size: 4

## Infix to Postfix Conversion

1. Write a java program that reads an infix expression, converts the expression to postfix form and then evaluates the postfix expression (use stack ADT).

Program 18(a) translates an infix expression to a postfix expression. The main method of the InfixToPostfix class is toPostfix(). This method takes “infix” string as an input parameter and returns “postfix” string. In the process, we use a stack as a temporary storage. So, instead of writing a separate program for stack operations, the InfixToPostfix class uses java.util.Stack class.

Program 18(a): Infix to Postfix Conversion

class **InfixToPostfix**

{

java.util.Stack<Character> stk =

new java.util.Stack<Character>();

public String **toPostfix**(String infix)

{

infix = "(" + infix + ")"; // enclose infix expr within parentheses

String postfix = "";

/\* scan the infix char-by-char until end of string is reached \*/

for( int i=0; i<infix.length(); i++)

{

char ch, item;

ch = infix.charAt(i);

if( isOperand(ch) ) // if(*ch* is an operand), then

postfix = postfix + ch; // append *ch* to postfix string

if( ch == '(' ) // if(*ch* is a left-bracket), then

stk.push(ch); // push onto the stack

if( isOperator(ch) ) // if(*ch* is an operator), then

{

item = stk.pop(); // pop an *item* from the stack

/\* if(*item* is an operator), then check the precedence of *ch* and *item\*/*

if( isOperator(item) )

{

if( precedence(item) >= precedence(ch) )

{

stk.push(item); stk.push(ch);

}

else

{ postfix = postfix + item; stk.push(ch);

}

}

else

{ stk.push(item);

stk.push(ch);

}

} // end of if(isOperator(ch))

if( ch == ')' )

{

item = stk.pop(); while( item != '(' )

{

postfix = postfix + item; item = stk.pop();

}

}

} // end of for-loop

return postfix;

} // end of toPostfix() method

public boolean **isOperand**(char c)

{ return(c >= 'A' && c <= 'Z'); }

public boolean **isOperator**(char c)

{

return( c=='+' || c=='-' || c=='\*' || c=='/' );

}

public int **precedence**(char c)

{

int rank = 1; // rank = 1 for '\*’ or '/'

if( c == '+' || c == '-' ) rank = 2; return rank;

}

}

///////////////////////// **InfixToPostfixDemo.java** /////////////// class **InfixToPostfixDemo**

{

public static void **main**(String args[])

{

InfixToPostfix obj = new InfixToPostfix();

String infix = "A\*(B+C/D)-E"; System.out.println("infix: " + infix );

System.out.println("postfix:"+obj.toPostfix(infix) );

}

}

Output of this program is:

infix: A\*(B+C/D)-E postfix: ABCD/+\*E-

## Evaluation of postfix expression

One of the most useful characteristics of postfix expression is that the value of postfix expression can be computed easily with the aid of a stack. The components of a postfix expression are processed from left to right as follows:

|  |  |
| --- | --- |
| *Item scanned from Postfix Expression* | *Action* |
| Operand | Push operand onto the stack |
| Operator | Pop the top two operands from the stack, apply the operator to them, and evaluate it. Push this result onto the stack. |

This algorithm finds the value of postfix expression, *P*. Each character of the expression, *P* is denoted by *ch*. We use a variable *stack***,** which is an array of integers. Initially *stack* is empty. The *result*, *tmp1* and *tmp2* are integer variables.

Program 18(b) illustrates the evaluation of postfix expression. The program works for expressions that contain only *single-digit integers* and the *four arithmetic operators*. The program uses **java.util.Stack** class for creating a stack of integer values.

Program 18(b): Evaluation of postfix expression

class **EvaluatePostfixExpression**

{

public static void **main**(String args[])

{

String postfix = "5 6 2 + \* 8 4 / -";

java.util.Stack<Integer>stk =

new java.util.Stack<Integer>();

char ch;

for( int i = 0; i < postfix.length(); i++ )

{

ch = postfix.charAt(i);

if( isDigit(ch) )

stk.push( new Integer(Character.digit(ch,10)));

if( isOperator(ch) )

{

int tmp1 = stk.pop(); int tmp2 = stk.pop();

int result = evaluate(tmp2, tmp1, ch); stk.push(result);

}

}

System.out.println("Value of postfix expr = " + stk.pop());

}

static boolean **isDigit**(char c)

{ return( c >= '0' && c <= '9' ); }

static boolean **isOperator**(char c)

{ return( c=='+' || c=='-' || c=='\*' || c=='/' ); }

static int **evaluate**(int a, int b, char op)

{ int res = 0; switch(op)

{

case '+' : res = (a+b); break; case '-' : res = (a-b); break; case '\*' : res = (a\*b); break; case '/' : res = (a/b); break;

}

return res;

}

}

Output of this program is: Value of postfix expr = 38

## Matching Brackets (Delimiters)

1. Write a Java program that determines whether parenthetic symbols **( )**, **{ }** and **[ ]** are nested correctly in a string of characters (use stack ADT).

A stack is used to check for matching the left and right brackets in an expression. We want to ensure that the parentheses are nested correctly; that is, we need to check that (1) there are an equal number of left and right parentheses, and (2) every right parenthesis is preceded by a matching left parenthesis.

For example, the expressions such as [A+(B\*C))] or {X\*Y+(Z–5} violate condition (1), and expressions {)A+B(-C} or [(A+B))-(C+D] violate condition (2).

The Program 19 uses Arraystack to check for matching left and right *brackets*: [ ], { }, and ( ) in an expression. The elements of the stack are *characters*. Here, we are concerned about the brackets. The items of the stack contain only *left* brackets. The valid is a boolean variable, which is initially made false.

The isExpressionValid() method makes use of the Arraystack class. Notice how easy it is to reuse this class. All the code you need is in one place. This is one of the payoffs for object-oriented programming.

Program 19: Matching left and right brackets in an expression using Stack

class **Expression**

{

private String expression;

Expression( String str ) // constructor

{ expression = str; }

public boolean **isExpressionValid()**

{ int n = expression.length(); // get max size (chars) of expression

ArrayStack stk = new ArrayStack(n); // create stack

char ch, chx; // ch: char scanned and chx: char popped

boolean valid = false;

for(int i = 0; i < n; i++) // get a char until ‘n’ chars are scanned

{

ch = expression.charAt(i); // get char

if( ch == '[' || ch == '{' || ch == '(' ) stk.push(ch);

if( ch == ']' || ch == '}' || ch == ')' )

{ if( stk.isEmpty() ) valid = false;

else

{ chx = (Character)stk.pop(); // pop a char from stack

if( chx == '[' && ch == ']' ) valid = true; if( chx == '{' && ch == '}' ) valid = true; if( chx == '(' && ch == ')' ) valid = true;

}

}

}

if( !stk.isEmpty() ) // stack not empty

valid = false;

return valid;

}

}

///////////////////// **ExpressionDemo.java** ///////////////////// class **ExpressionDemo**

{

public static void **main**(String[] args)

{

String expr = “[A+25\*{Y\*(B+C-X)-K}/D\*(E+13)+M]”;

Expression ob = new Expression(expr);

System.out.println("expression: " + expr); if( ob.isExpressionValid() )

System.out.println("expression is valid"); else

System.out.println("expression is not valid");

}

}

Output of this program is:

expression: [A+25\*{Y\*(B+C-X)-K}/D\*(E+13)+M] expression is valid

## Palindrome

1. Write a Java program that uses both stack and queue to test whether the given string is a palindrome.

First the characters are extracted one by one from the input string and pushed onto the stack. Then they are popped from the stack and compared with each character of the given string. It is enough to compare with first half of the string. Because of its last-in-first-out characteristic, the stack reverses the order of the characters. Program 20(a) uses the java.util.Stack (Refer chapter 10: Stacks).

Program 20(a): Testing whether the given string is a palindrome using stack

import java.util.Stack; class **Palindrome**

{

public static void main(String args[])

{

String str = "MALAYALAM";

if( isPalindrome(str) )

System.out.println( str + " is a Palindrome"); else

System.out.println( str + " is not a Palindrome");

}

static boolean isPalindrome(String str)

{

Stack<Character> stk = new Stack<Character>();

for( int i=0; i < str.length(); i++ ) stk.push(str.charAt(i));

for( int i=0; i < str.length()/2; i++ )

if( str.charAt(i) != stk.pop() ) return false;

return true;

}

}

First the characters are extracted one by one from the input string and inserted into the queue. Then they are removed from the queue and compared with each character of the given string (in the reverse order of the string). It is enough to compare with second half of the string (of course in the reverse order). Because of its first-in-first-out characteristic, the characters are deleted from the queue in the order of the characters of the string. Program 20(b) uses the java.util.LinkedList to implement the queue (Refer chapter 11: Queues).

Program 20(b): Testing whether the given string is a palindrome using queue

import java.util.LinkedList; class **Palindrome**

{

public static void main(String args[])

{

String str = "RADAR";

if( isPalindrome(str) )

System.out.println( str + " is a Palindrome"); else

System.out.println( str + " is not a Palindrome");

}

static boolean isPalindrome(String str)

{

LinkedList<Character> que = new LinkedList<Character>(); int n = str.length();

for( int i=0; i < n; i++ ) que.addLast(str.charAt(i));

for( int i=n-1; i > n/2; i-- )

if( str.charAt(i) != que.removeFirst() ) return false;

return true;

}

}

## Linked Stack

1. Write Java programs to implement the following using a singly linked list.
   1. Stack ADT
   2. Queue ADT

Another way to represent a stack is by using a linked list. A stack can be represented by using nodes of the linked list. Each node contains two fields: **data** (*info*) and **next** (*link*). The **data** field of each node contains an item in the stack and the corresponding **next** field points to the node containing the next item in the stack. The **next** field of the last node is **null** – that is, the bottom of the stack. The **top** refers to the topmost node (the last item inserted) in the stack. The ***empty*** stack is represented by setting **top** to **null**. Because the way the nodes are pointing, **push** and **pop** operations are easy to accomplish. Program 21(a) is a complete listing, demonstrating the push and pop operations of a stack using singly linked list.

Program 21(a): Linked Implementation of a Stack

class **Node**

{ int data; // data item

Node next; // next node in linked-stack

Node( int d ) // constructor

{ data = d; } // next is automatically set to null

}

class **LinkedStack**

{

Node top; // top refers to top-node

Node p; // p refers to current node

public void **push**(int item) // add item onto stack

{

p = new Node(item); // create new node

p.next = top; // new node refers to old top

top = p; // top refers to new node

}

public Node **pop**() // remove a node from the stack

{

if( isEmpty() )

{ System.out.println("Stack is empty"); return null;

}

Node tmp = top; // tmp saves reference to top node

top = tmp.next; // now, top refers to next node of old top

return tmp; // return the popped item

}

public Node **peek**() // get top node from the stack, without deleting

{

if( isEmpty() )

{ System.out.println("Stack is empty"); return null;

}

return top;

}

public void **displayStack**()

{

p = top; // p refers to top

System.out.print("\nContents of Stack: [ ");

while( p != null ) // start printing from top of stack to bottom of stack

{

System.out.print(p.data + " "); // print data

p = p.next; // move to next node

}

System.out.println("]");

}

public boolean **isEmpty**() // true if stack is empty

{ return (top == null); }

}

///////////////////////// **LinkedStackDemo.java** ///////////////// class **LinkedStackDemo**

{

public static void **main**(String[] args)

{

LinkedStack stk = new LinkedStack(); // create stack object

Node item; // item stores popped node

stk.push(20); // add 20, 35, 40 to stack

stk.push(35);

stk.push(40);

stk.displayStack(); // print contents of stack

item = stk.pop(); // remove a node from the top and print it

if( item != null )

{

System.out.println("Popped item: " + item.data); stk.displayStack();

}

stk.push(65); // insert 65, 70, 75

stk.push(70);

stk.push(75);

stk.displayStack(); // display contents of stack

item = stk.pop(); // remove a node from the top and display it

if( item != null )

{

System.out.println(“Popped item: ” + item.data); stk.displayStack();

}

System.out.println(“peek(): ” + stk.peek());// get top item

stk.push(90); // insert 90

stk.displayStack();

}

}

Output from LinkedStack operations program:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Contents of Stack: | [ | 40 | 35 | 20 | ] |  |
| Popped item: 40 |  |  |  |  |  |
| Contents of Stack: | [ | 35 | 20 | ] |  |
| Contents of Stack: | [ | 75 | 70 | 65 | 35 | 20 ] |
| Popped item: 75 |  |  |  |  |  |  |
| peek(): 70 |  |  |  |  |  |  |
| Contents of Stack: | [ | 70 | 65 | 35 | 20 | ] |
| Contents of Stack: | [ | 90 | 70 | 65 | 35 | 20 ] |

## Linked Queue

In contiguous storage (using arrays), queues were harder to manipulate than were stacks. It causes difficulties to handle full queues and empty queues. It is for queues that linked storage really comes into its own. The linked implementation has two advantages over the array implementation: (1) it is faster – locations for insertion and deletion are same – at the back and at the front, and (2) it wastes no space – removed nodes are deleted by the automatic garbage collector process.

Linked queues are just as easy to handle as are linked stacks. This section presents a queue implementation which makes use of the singly linked list. We keep two pointers, *front* and *rear*. The operations of LinkedQueue class is given Program 21(b) and LinkedQueue class is tested in Program 21(c).

Program 21(b): A LinkedQueue Class

public class **LinkedQueue**

{

class **Node**

{ Object data; Node next;

Node(Object item) // constructor

{ data = item; }

}

Node front, rear; int count;

public void **insert**(Object item)

{

Node p = new Node(item);

if(front == null) // queue is empty; insert first item

{ front = rear = p; rear.next = null;

}

if(front == rear) // queue contains one item; insert second item

{ rear = p; front.next = rear; rear.next = null;

}

else // queue contains 2 or more items

{ rear.next = p; // old rear.next refers to p rear = p; // new rear refers to p rear.next = null;

}

count++; // increment queue size

}

public Object **remove**()

{ if(isEmpty())

{ System.out.println("Q is empty"); return null; }

Object item = front.data; front = front.next;

count--; // decrement queue size

return item;

}

public boolean **isEmpty**()

{ return (front == null); }

public Object **peek**()

{ return front.data; }

public int **size**()

{ return count; }

public void **display**()

{ Node p = front; System.out.print("Linked Q: ");

if(p == null) System.out.println("empty"); while( p != null )

{

System.out.print(p.data + " "); p = p.next;

}

System.out.println();

}

}

Program 21(c): Testing LinkedQueue Class

class **LinkedQueueDemo**

{ public static void **main**(String[] args)

{

LinkedQueue q = new LinkedQueue();

q.display(); q.insert('A');

q.insert('B');

q.insert('C');

q.insert('D'); q.display();

System.out.println("delete(): " + q.remove()); q.display();

System.out.println("peek(): " + q.peek()); q.insert('E');

q.insert('F');

System.out.println("delete(): " + q.remove()); q.display();

System.out.println("size(): " + q.size());

}

}

Here is the output of this program:

Linked Q: empty Linked Q: A B C D remove(): A Linked Q: B C D peek(): B remove(): B Linked Q: C D E F size(): 4

## Deque ADT

1. Write Java programs to implement the deque (double ended queue) ADT using
   1. Array
   2. Doubly linked list.

A *deque* is a double-ended queue. You can insert items at either end and delete them from either end. Methods are addFirst(), addLast(), removeFirst() and removeLast().

If you restrict yourself to addFirst() and removeFirst() (or their equivalents on the right), then the deque acts like a *stack*. If you restrict yourself to addFirst() and removeLast() (or the opposite pair), then it acts like a *queue*.

A deque provides a more versatile data structure than either a stack or a queue, and is sometimes used in container class libraries to serve both purposes. However, it is not used as often as stacks and queues.

The deque is maintained by either an array or linked list with pointers first and last which point to the two ends of the deque. Such a structure is represented by the following figure.

**first last**

 

**Deletion ** **Insertion **

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |

 **Insertion ** **Deletion**

The methods of deque ADT are as follows:

|  |  |
| --- | --- |
| **addFirst(Object)** | Inserts an item at the left side of deque. |
| **addLast(Object)** | Inserts an item at the right side of deque. |
| **removeFirst()** | Deletes an item from the left of deque. |
| **removeLast()** | Deletes an item from the right of deque. |
| **getFirst()** | Returns an item from the left, without deleting the item. |
| **getLast()** | Returns an item from right, without deleting the item. |
| **size()** | Returns the current number of items in the deque. |
| **isEmpty()** | Returns true, if deque is empty else returns false. |

### ArrayDeque

Program 22(a) is an array implementation of ArrayDeque class and it is tested in Program 22(b).

Program 22(a): An ArrayDeque Class

public class **ArrayDeque**

{

private int maxSize; private Object[] que; private int first; private int last;

private int count; // current number of items in deque

public ArrayDeque(int s) // constructor

{ maxSize = s;

que = new Object[maxSize]; first = last = -1;

count = 0;

}

public void **addLast**(Object item)

{ if(count == maxSize)

{ System.out.println("Deque is full"); return; }

last = (last+1) % maxSize; que[last] = item;

if(first == -1 && last == 0) first = 0; count++;

}

public Object **removeLast**()

{ if(count == 0)

{ System.out.println("Deque is empty"); return(' ');}

Object item = que[last]; que[last] = ‘ ’;

if(last > 0) last = (last-1) % maxSize; count--;

if(count == 0) first = last = -1;

return(item);

}

public void **addFirst**(Object item)

{ if(count == maxSize)

{ System.out.println("Deque is full"); return; }

if(first > 0) first = (first-1) % maxSize; else if(first == 0) first = maxSize-1;

que[first] = item; count++;

}

public Object **removeFirst**()

{ if(count == 0)

{ System.out.println("Deque is empty"); return(' ');

}

Object item = que[first]; que[first] = ‘ ’;

if(first == maxSize-1) first = 0;

else first = (first+1) % maxSize; count--;

if(count == 0) first = last = -1;

return(item);

}

void **display**()

{ System.out.println("----------------------------");

System.out.print("first:"+first + ", last:"+ last); System.out.println(", count: " + count); System.out.println(" 0 1 2 3 4 5"); System.out.print("Deque: ");

for( int i=0; i<maxSize; i++ ) System.out.print(que[i]+ " ");

System.out.println("\n----------------------------");

}

public boolean **isEmpty**() // true if queue is empty

{ return (count == 0); }

public boolean **isFull**() // true if queue is full

{ return (count == maxSize); }

}

Program 22(b): Testing ArrayDeque Class

class **ArrayDequeDemo**

{ public static void **main**(String[] args)

{ ArrayDeque q = new ArrayDeque(6); // queue holds a max of 6 items

q.insertLast('A'); /\* (a) \*/

q.insertLast('B');

q.insertLast('C');

q.insertLast('D');

System.out.println("deleteFirst():"+q.deleteFirst()); q.display();

q.insertLast('E'); /\* (b) \*/

q.display();

/\* (c) \*/ System.out.println("deleteLast():"+q.deleteLast()); System.out.println("deleteLast():"+q.deleteLast()); q.display();

q.insertFirst('P'); q.insertFirst('Q'); /\* (d) \*/

q.insertFirst('R'); q.display();

q.deleteFirst(); q.display(); /\* (e) \*/

q.insertFirst('X'); q.display(); /\* (f) \*/

q.insertLast('Y'); q.display(); /\* (g) \*/

q.insertLast('Z'); q.display(); /\* (h) \*/

}

}

Output of this program is as follows:

deleteFirst(): A

----------------------------

first:1, last:3, count: 3 0 1 2 3 4 5

Deque: B C D

----------------------------

first:1, last:4, count: 4 0 1 2 3 4 5

Deque: B C D E

----------------------------

deleteLast(): E deleteLast(): D

----------------------------

first:1, last:2, count: 2 0 1 2 3 4 5

Deque: B C

----------------------------

first:4, last:2, count: 5 0 1 2 3 4 5

Deque: P B C R Q

----------------------------

first:5, last:2, count: 4 0 1 2 3 4 5

Deque: P B C Q

----------------------------

first:4, last:2, count: 5 0 1 2 3 4 5

Deque: P B C X Q

----------------------------

first:4, last:3, count: 6 0 1 2 3 4 5

Deque: P B C Y X Q

----------------------------

Deque is full

----------------------------

first:4, last:3, count: 6 0 1 2 3 4 5

Deque: P B C Y X Q

----------------------------

### Linked Deque

A deque is implemented by using a *doubly linked list* with references to *first* and *last*. Following figure illustrates the linked deque operations. Each node of the doubly linked list contains three fields: data, prev and next. The fields prev and next refer to the node itself. Program 22(c) implements LinkedDeque class which is tested in Program 22(d).

Program 22(c): A LinkedDeque class

public class **LinkedDeque**

{

public class **DequeNode**

{

DequeNode prev; Object data; DequeNode next;

DequeNode( Object item ) // constructor

{

data = item;

} // prev & next automatically refer to null

}

private DequeNode first, last; private int count;

public void **addFirst**(Object item)

{ if( isEmpty() )

first = last = new DequeNode(item); else

{ DequeNode tmp = new DequeNode(item); tmp.next = first;

first.prev = tmp; first = tmp;

}

count++;

}

public void **addLast**(Object item)

{

if( isEmpty() )

first = last = new DequeNode(item); else

{ DequeNode tmp = new DequeNode(item); tmp.prev = last;

last.next = tmp; last = tmp;

}

count++;

}

public Object **removeFirst**()

{

if( isEmpty() )

{ System.out.println("Deque is empty"); return null;

}

else

{ Object item = first.data; first = first.next; first.prev = null;

count--; return item;

}

}

public Object **removeLast**()

{

if( isEmpty() )

{ System.out.println("Deque is empty"); return null;

}

else

{ Object item = last.data; last = last.prev; last.next = null;

count--; return item;

}

}

public Object **getFirst**()

{

if( !isEmpty() ) return( first.data ); else return null;

}

public Object **getLas**t()

{

if( !isEmpty() ) return( last.data ); else return null;

}

public boolean **isEmpty**()

{ return (count == 0); }

public int **size**()

{ return(count); }

public void **display**()

{ DequeNode p = first; System.out.print("Deque: [ "); while( p != null )

{ System.out.print( p.data + " " ); p = p.next;

}

System.out.println("]");

}

}

Program 22(d): Testing LinkedDeque Class

public class LinkedDequeDemo

{

public static void **main**( String args[])

{

LinkedDeque dq = new LinkedDeque();

System.out.println("removeFirst():" + dq.removeFirst()); dq.addFirst('A');

dq.addFirst('B');

dq.addFirst('C'); dq.display();

dq.addLast('D');

dq.addLast('E');

System.out.println("getFirst():" + dq.getFirst()); System.out.println("getLast():" + dq.getLast()); dq.display();

System.out.println("removeFirst():"+dq.removeFirst()); System.out.println("removeLast():"+ dq.removeLast()); dq.display();

System.out.println("size():" + dq.size());

}

}

Output of this program is:

Deque is empty removeFirst(): null Deque: [ C B A ] getFirst(): C getLast(): E

Deque: [ C B A D E ] removeFirst(): C

removeLast(): E Deque: [ B A D ] size(): 3

## Priority Queue

1. Write a Java program to implement a priority queue ADT.

In many situations, ordinary queues are inadequate, as when FIFO arrangement has to be overruled using some priority criteria.

The problem with a priority queue is in finding an efficient implementation that allows relatively fast insertion and deletion. Because items may arrive randomly to the queue, there is no guarantee that the items at front will be the most likely to be removed and that the items inserted at the rear will be the last items for deletion. Priority queues are implemented by using (1) arrays, (2) linked lists, (3) binary heaps

Linked Implementation of a Priority Queue

We can represent a priority queue as an ordered linked list. The items of the queue are in ascending order – the items of higher priority are at starting of the list. Each list node consists of three fields: *data*, *priority number* and *next*. A node of higher priority is processed (removed) before any item of lower priority. The items with the same priority are deleted according to the order in which they were added to the queue – that is, FIFO discipline. The priority numbers work like this: the lower the priority number, the higher the priority. The Node class is declared as follows (data field is a String type, priority number, prn is an int, and next refers to the next node in the list):

class Node

{ **data prn next**

String data; int prn;

Node next;

}

**Node**

The *head* indicates (or refers to) first node of the list. The delete routine removes the *head* node and makes the *new head* to refer to *head next* node.

Adding an item to the priority queue is complicated than deleting a node from the queue, because we need to find the correct location to insert the node.

The insert method traverses the list until finding a node (call it *N*) whose priority number is greater than priority number of new node to be added. Insert new node in front of *N*. If no such node is found, insert new node after the end of the list (that is, as a last node of the list). While traversing the list, the object reference of preceding node of the new node is to be saved.

**LinkedPriorityQueue** class implemented as a linked list is given in Program 23(b), and it is tested in Program 23(c). Node class is defined in Program 23(a).

Program 23(a): Linked Priority Queue Node Class

public class **Node**

{ String data; // data item

int prn; // priority number (minimum has highest priority)

Node next; // "next" refers to the next node

Node( String str, int p ) // constructor

{ data = str;

prn = p;

} // "next" is automatically set to null

}

Program 23(b): LinkedPriorityQueue Class

class **LinkedPriorityQueue**

{

Node head; // “head” refers to first node

public void insert(String item, int pkey) // insert item after pkey

{

Node newNode = new Node(item, pkey); // create new node

int k;

if( head == null ) k = 1;

else if( newNode.prn < head.prn ) k = 2; else k = 3;

switch( k )

{ case 1: head = newNode; // Q is empty, add head node

head.next = null; break;

case 2: Node oldHead = head; // add one item before head

head = newNode; newNode.next = oldHead; break;

case 3: Node p = head; // add item before a node

Node prev = p;

Node nodeBefore = null;

while( p != null )

{

if( newNode.prn < p.prn )

{ nodeBefore = p; break;

}

else

{ prev = p; // save previous node of current node

p = p.next; // move to next node

}

} // end of while

newNode.next = nodeBefore; prev.next = newNode;

} // end of switch

} // end of insert() method

public Node **delete**()

{

if( isEmpty() )

{ System.out.println("Queue is empty"); return null;

}

else

{ Node tmp = head; head = head.next; return tmp;

}

}

public void **displayList**()

{

Node p = head; // assign address of head to p System.out.print("\nQueue: ");

while( p != null ) // start at beginning of list until end of list

{

System.out.print(p.data+"(" +p.prn+ ")" + " "); p = p.next; // move to next node

}

System.out.println();

}

public boolean **isEmpty**() // true if list is empty

{ return (head == null); }

public Node **peek**() // get first item

{ return head; }

}

Program 23(c): Testing of LinkedPriorityQueue Class

class **LinkedPriorityQueueDemo**

{

public static void **main**(String[] args)

{

LinkedPriorityQueue pq = new LinkedPriorityQueue(); // create new queue list

Node item; pq.insert("Babu", 3);

pq.insert("Nitin", 2);

pq.insert("Laxmi", 2);

pq.insert("Kim", 1);

pq.insert("Jimmy", 3); pq.displayList();

item = pq.delete(); if( item != null )

System.out.println("delete():" + item.data

+ "(" +item.prn+")");

pq.displayList();

pq.insert("Scot", 2);

pq.insert("Anu", 1);

pq.insert("Lehar", 4); pq.displayList();

}

}

Output of this program is:

Queue: Kim(1) Nitin(2) Laxmi(2) Babu(3) Jimmy(3) delete(): Kim(1)

Queue: Nitin(2) Laxmi(2) Babu(3) Jimmy(3)

Queue: Anu(1) Nitin(2) Laxmi(2) Scot(2) Babu(3) Jimmy(3) Lehar(4)

# Binary Trees

A *binary tree* is a special form of a tree. A binary tree is more important and frequently used in various applications of computer science. When binary trees are in sorted form, they facilitate quick search, insertion and deletion.

A *binary tree* is either empty, or it consists of a node called the *root* together with two binary trees called the *left subtree* or the *right subtree* of the root. This definition is that of a mathematical structure. To specify binary trees as an abstract data type, we must state what operations can be performed on binary trees.

## Binary Tree Traversals

1. Write Java programs that use recursive and non-recursive functions to traverse the given binary tree in
   1. Preorder
   2. **Inorder,** and
   3. Postorder.

**Java Program - Binary Tree**

We now implement a Java program for the following operations on binary trees:

* Representation of binary tree: a single array represents the node data linearly – this array is the input for building the binary tree.
* Linked implementation of binary tree in memory: binary tree is being built by reading the node data. In the process, the object reference of the tree root is saved - this is very important to do any further tree operations.
* Traversals: inorder, preorder and postorder traversals are performed recursively and iteratively; and their linear orders are printed.

The buildTree() method

This is a recursive method that builds a linked binary tree. It uses input node data which is represented as a linear array (**null** links are added to complete the tree).



**0**

**E**

**1**

**C**

**2**

**G**

**3**

**A**

**4**

**D**

**5**

**F**

**6**

**H**

**7**

**x**

**8**

**B**

**x x**

**x x**

**x x**

**9 10 11**

**12**

**13**

**14**

**x x**

**x x**

**x** indicates **null**

**15 16 17 18**

**0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **E** | **C** | **G** | **A** | **D** | **F** | **H** | **x** | **B** | **x** | **x** | **x** | **x** | **x** | **x** | **x** | **x** | **x** | **x** |

**Array representation of a binary tree**

The buildTree() method takes the index of the array as an input parameter. Initially, the index is

0 (root node). The method calls recursively with updated index – for left child, the index is (2\*index+1) and for right child, it is (2\*index+2). The routine in Java is as follows:

public Node **buildTree**( int index )

{ Node p = null; // ‘p’ refers to current node

if( tree[index] != null )

{ p = new Node(tree[index]); // create a node from array data

// call buildTree() to create a left node

p.left = buildTree(2\*index+1);

// call buildTree() to create a right node

p.right = buildTree(2\*index+2);

}

return p;

}

The recursive algorithms for tree traversals are simple and easy. The non-recursive algorithms use

*stack* to temporarily hold the addresses of nodes for further processing.

Given a binary tree whose root node address is given by a reference variable *root* and whose structure is {*left, data, right*}. The *left* and *right* are reference variables referring to left and right subtrees respectively. We define another data structure *stack* that is used to save node reference as the tree is traversed. The *stack* is an array of nodes that stores the node addresses of the tree. The reference variable *p* denotes the current node in the tree.

The iterative inorder and postorder traversals are straight forward, whereas the postorder traversal is complicated. In non-recursive postorder traversal, we may have to save a node twice in two different situations.

Program 24 implements the binary tree traversals recursively and iteratively.

Program 24: Binary Tree Traversals

class Node

{ Object data; Node left; Node right;

Node( Object d ) // constructor

{ data = d; }

}

class BinaryTree

{

Object tree[]; int maxSize;

java.util.Stack<Node> stk = new java.util.Stack<Node>();

BinaryTree( Object a[], int n ) // constructor

{ maxSize = n;

tree = new Object[maxSize]; for( int i=0; i<maxSize; i++ )

tree[i] = a[i];

}

public Node buildTree( int index )

{ Node p = null;

if( tree[index] != null )

{ p = new Node(tree[index]);

p.left = buildTree(2\*index+1); p.right = buildTree(2\*index+2);

}

return p;

}

/\* Recursive methods - Binary tree traversals \*/ public void inorder(Node p)

{

if( p != null )

{

inorder(p.left); System.out.print(p.data + " "); inorder(p.right);

}

}

public void preorder(Node p)

{

if( p != null )

{

System.out.print(p.data + " "); preorder(p.left); preorder(p.right);

}

}

public void postorder(Node p)

{

if( p != null )

{

postorder(p.left); postorder(p.right); System.out.print(p.data + " ");

}

}

/\* Non-recursive methods - Binary tree traversals \*/ public void preorderIterative(Node p)

{

if(p == null )

{ System.out.println("Tree is empty"); return;

}

stk.push(p);

while( !stk.isEmpty() )

{

p = stk.pop(); if( p != null )

{

System.out.print(p.data + " "); stk.push(p.right); stk.push(p.left);

}

}

}

public void inorderIterative(Node p)

{

if(p == null )

{ System.out.println("Tree is empty"); return;

}

while( !stk.isEmpty() || p != null )

{

if( p != null )

{ stk.push(p); // push left-most path onto stack

p = p.left;

}

else

{

p = stk.pop(); // assign popped node to p System.out.print(p.data + " "); // print node data p = p.right; // move p to right subtree

}

}

}

public void postorderIterative(Node p)

{

if(p == null )

{ System.out.println("Tree is empty"); return;

}

Node tmp = p; while( p != null )

{

while( p.left != null )

{ stk.push(p); p = p.left;

}

while( p != null && (p.right == null || p.right == tmp ))

{ System.out.print(p.data + " "); // print node data tmp = p;

if( stk.isEmpty() ) return;

p = stk.pop();

}

stk.push(p); p = p.right;

}

}

} // end of BinaryTree class

//////////////////////// BinaryTreeDemo.java ////////////////////// class BinaryTreeDemo

{

public static void main(String args[])

{

Object arr[] = {'E', 'C', 'G', 'A', 'D', 'F', 'H',

null,'B', null, null, null, null, null, null, null, null, null, null};

BinaryTree t = new BinaryTree( arr, arr.length );

Node root = t.buildTree(0); // buildTree() returns reference to root

System.out.print("\n Recursive Binary Tree Traversals:"); System.out.print("\n inorder: ");

t.inorder(root);

System.out.print("\n preorder: "); t.preorder(root);

System.out.print("\n postorder: "); t.postorder(root);

System.out.print("\n Non-recursive Binary Tree Traversals:"); System.out.print("\n inorder: ");

t.inorderIterative(root);

System.out.print("\n preorder: "); t.preorderIterative(root);

System.out.print("\n postorder: "); t.postorderIterative(root);

}

}

**Output of this binary tree program is:** Recursive Binary Tree Traversals: inorder: A B C D E F G H preorder: E C A B D G F H postorder: B A D C F H G E

Non-recursive Binary Tree Traversals: inorder: A B C D E F G H

preorder: E C A B D G F H postorder: B A D C F H G E

## Level Order Traversal

1. Write a Java program that displays node values in a level order traversal (Traverse the tree one level at a time, starting at the root node) for a binary tree.

Traverse a binary tree level by level. That is, the root is visited first, then the immediate children of the root (from left to right), then grandchildren of the root (from left to right), and so on. The algorithm uses a *queue* to keep track of the children of a node until it is time to visit them. This algorithm is also known as *breadth-first traversal*.

1. Initialize a queue.
2. Insert the root into the queue.
3. Repeat steps 4 to 7 until the queue is empty.
4. Delete the *node* from the queue.
5. Process the *node* being deleted.
6. If the left child of *node* exists (is not null), insert it into the queue.
7. If the right child of *node* exists (is not null), insert it into the queue

Program 25: Level order traversal

class **Node**

{

Object data; Node left; Node right;

Node( Object d ) // constructor

{ data = d; }

}

class **BinaryTree**

{ Object tree[]; int maxSize;

java.util.LinkedList<Node> que =

new java.util.LinkedList<Node>();

BinaryTree( Object a[], int n ) // constructor

{ maxSize = n;

tree = new Object[maxSize]; for( int i=0; i<maxSize; i++ )

tree[i] = a[i];

}

public Node **buildTree**( int index )

{ Node p = null;

if( tree[index] != null )

{ p = new Node(tree[index]); p.left = buildTree(2\*index+1); p.right = buildTree(2\*index+2);

}

return p;

}

public void **levelorder**(Node p)

{

que.addLast(p);

while( !que.isEmpty() )

{

p = que.removeFirst(); System.out.print(p.data + " "); if(p.left != null)

que.addLast(p.left); if(p.right != null)

que.addLast(p.right);

}

}

} // end of BinaryTree class

//////////////////////// **LevelOrderTraversal.java** ////////////////////// class LevelOrderTraversal

{

public static void main(String args[])

{

Object arr[] = {'E', 'C', 'G', 'A', 'D', 'F', 'H',

null,'B', null, null, null, null, null, null, null, null, null, null};

BinaryTree t = new BinaryTree( arr, arr.length );

Node root = t.buildTree(0); // buildTree() returns reference to root

System.out.print("\n Level Order Tree Traversal: ");

t.levelorder(root);

}

}

Output of this program is:

Level Order Tree Traversal: E C G A D F H B

# Search Trees

1. Write a Java program that uses recursive functions.
   1. To create a binary search tree.
   2. To count the number of leaf nodes.
   3. To copy the above binary search tree.
2. Write a Java program to perform the following operations:
   1. Insert an element into a binary search tree.
   2. Delete an element from a binary search tree.
   3. Search for a key element in a binary search tree.

## Binary Search Tree

A ***binary search tree*** is a binary tree that is either empty or in which every node contains a key and satisfies the conditions:

* + 1. The key in the left child of a node (if it exists) is less than the key in its parent node.
    2. The key in the right child of a node (if it exists) is greater than the key in its parent node.
    3. The left and right subtrees of the root are again binary search trees.

The first two properties describe the ordering relative to the key in the root node, and that the third property extends them to all nodes in the tree; hence we can continue to use the *recursive* structure of the binary tree. After we examine the root of the tree, we shall move to either its left or right subtree, and this subtree is again a binary search tree. Thus we can use the same method again on this smaller tree. This definition assumes that no duplicate keys are permitted.



**45**

**25**

**65**

**15**

**30**

**55**

**75**

**10**

**20**

**50**

**60**

**80**

Program 26 & 27: Binary search tree operations

class BSTNode

{

int data; BSTNode left; BSTNode right;

BSTNode( int d ) // constructor

{ data = d; }

}

class BinarySearchTree

{

public BSTNode **insertTree**(BSTNode p, int key) // create BST

{

if( p == null )

p = new BSTNode(key); else if( key < p.data)

p.left = insertTree( p.left, key); else p.right = insertTree( p.right, key); return p; // return root

}

public BSTNode **search**(BSTNode root, int key)

{

BSTNode p = root; // initialize p with root

while( p != null )

{ if( key == p.data ) return p;

else if( key < p.data ) p = p.left; else p = p.right;

}

return null;

}

public int **leafNodes**(BSTNode p)

{

int count = 0; if( p != null)

{ if((p.left == null) && (p.right == null)) count = 1;

else

count = count + leafNodes(p.left)

+ leafNodes(p.right);

}

return count;

}

public BSTNode **deleteTree**(BSTNode root, int key)

{

BSTNode p; // refer to node to be deleted

BSTNode parent = root; // refer to parent of node to be deleted

BSTNode inorderSucc; //refer to inorder succ. of node to be deleted

if(root == null)

{ System.out.println("Tree is empty"); return null;

}

p = root; // initialize p with root

/\* find node being deleted & its parent \*/

while( p != null && p.data != key)

{ parent = p;

if( key < p.data) p = p.left; else p = p.right;

}

if( p == null )

{ System.out.println("\n Node " + key + " not found for deletion"); return null;

}

/\* find inorder successor of the node being deleted and its parent \*/

if(p.left != null && p.right != null) // case-3

{ parent = p; inorderSucc = p.right;

while(inorderSucc.left != null)

{

parent = inorderSucc; inorderSucc = inorderSucc.left;

}

p.data = inorderSucc.data; p = inorderSucc;

}

if(p.left == null && p.right == null) // case-1

{

if( parent.left == p ) parent.left = null; else parent.right = null;

}

if(p.left == null && p.right != null) *// case-2(a)*

{

if(parent.left == p) parent.left = p.right; else parent.right = p.right;

}

if(p.left != null && p.right == null) // case-2(b)

{

if(parent.left == p) parent.left = p.left; else parent.right = p.left;

}

return root;

}

public void **inorder**(BSTNode p) // 'p' starts with root

{ if( p != null )

{ inorder(p.left); System.out.print(p.data + " "); inorder(p.right);

}

}

public void **preorder**(BSTNode p)

{ if( p != null )

{ System.out.print(p.data + " "); preorder(p.left); preorder(p.right);

}

}

public void **postorder**(BSTNode p)

{ if( p != null )

{ postorder(p.left); postorder(p.right); System.out.print(p.data + " ");

}

}

} // end of BinarySearchTree class

////////////////////// **BinarySearchTreeDemo.java** //////////////////// class BinarySearchTreeDemo

{ public static void **main**(String args[])

{

int arr[] = { 45, 25, 15, 10, 20, 30, 65, 55, 50, 60, 75, 80 };

BinarySearchTree bst = new BinarySearchTree();

BSTNode root = null;

// build tree by repeated insertions

for( int i = 0; i < arr.length; i++ ) root = bst.insertTree( root, arr[i]);

BSTNode root2 = root; // copy BST

int key = 66;

BSTNode item = bst.search(root2, key); if( item != null )

System.out.print("\n item found: " + item.data); else System.out.print("\n Node " + key + " not found");

System.out.print("\n Number of leaf nodes: " + bst.leafNodes(root)); System.out.print("\n Inorder: ");

bst.inorder(root);

System.out.print("\n Preorder: "); bst.preorder(root); System.out.print("\n Postorder: "); bst.postorder(root);

key = 55; // delete 55

bst.deleteTree(root, key);

System.out.print("\n Inorder, after deletion of " + key + ": "); bst.inorder(root);

key = 44; // insert 44

bst.insertTree(root, key);

System.out.print("\n Inorder, after insertion of " + key + ": "); bst.inorder(root);

}

}

Output of this program is as follows:

Node 66 not found Number of leaf nodes: 6

Inorder: 10 15 20 25 30 45 50 55 60 65 75 80

Preorder: 45 25 15 10 20 30 65 55 50 60 75 80

Postorder: 10 20 15 30 25 50 60 55 80 75 65 45

Inorder, after deletion of 55: 10 15 20 25 30 45 50 60 65 75 80

Inorder, after insertion of 44: 10 15 20 25 30 44 45 50 60 65 75 80

## AVL Tree

1. Write a Java program to perform the following operations
   1. Insertion into an AVL-tree
   2. Deletion from an AVL-tree

An *AVL* tree is a binary search tree in which the heights of the left and right subtrees of the root differ at most 1 and in which the left and right subtrees are again *AVL* trees. Each node of an *AVL* tree is associated with a balance factor that is the left subtree has height *greater than*, *equal to*, or *less than* that of the right subtree.

**Program 28: AVL tree operations** (deletion is not implemented; left as an exercise)

class **AVLTree**

{

private class **AVLNode**

{

|  |  |  |
| --- | --- | --- |
| int | data; | // Data in the node |
| AVLNode | left; | // Left child |
| AVLNode | right; | // Right child |
| int | height; | // Height |

AVLNode( int d ) // Constructors

{

this( d, null, null );

}

AVLNode( int d, AVLNode lt, AVLNode rt )

{

data = d;

left = lt;

right = rt;

height = 0;

}

}

private AVLNode root; // The tree root

public AVLTree( ) // Construct the tree

{

root = null;

}

/\*

Insert into the tree; duplicates are ignored. x the item to insert.

\*/

public void **insert**( int x )

{

root = insert( x, root );

}

/\*

Find an item in the tree. x the item to search for. return true if x is found.

\*/

public boolean **search**( int x )

{

return search( x, root );

}

public void **makeEmpty**( ) // Make the tree logically empty.

{

root = null;

}

/\*

Test if the tree is logically empty. return true if empty, false otherwise.

\*/

public boolean **isEmpty**( )

{

return root == null;

}

/\*

Print the tree contents in sorted order.

\*/

public void **printTree**( )

{

if( isEmpty( ) )

System.out.println( "Empty tree" );

else

printTree( root );

}

/\*

method to insert into a subtree. x the item to insert.

t the node that roots the subtree. return the new root of the subtree.

\*/

private AVLNode **insert**( int x, AVLNode t )

{

if( t == null )

return new AVLNode( x, null, null );

if( x < t.data )

{

t.left = insert( x, t.left );

if( height( t.left ) - height( t.right ) == 2 ) if( x < t.left.data )

t = rotateLeft( t );

else

t = doubleLeft( t );

}

else if( x > t.data )

{

t.right = insert( x, t.right );

if( height( t.right ) - height( t.left ) == 2 ) if( x > t.right.data )

t = rotateRight( t );

else

t = doubleRight( t );

}

else

; // Duplicate; do nothing

t.height = Math.max( height( t.left ), height( t.right )) + 1; return t;

}

/\*

method to find an item in a subtree. x is item to search for.

t the node that roots the tree. return true if x is found in subtree.

\*/

private boolean **search**( int x, AVLNode t )

{

while( t != null )

{

if( x < t.data ) t = t.left;

else if( x > t.data ) t = t.right;

else

return true; // Match

}

return false; // No match

}

/\*

method to print the tree in sorted order. t the node that roots the tree.

\*/

private void **printTree**( AVLNode t ) // inorder traversal

{

if( t != null )

{

printTree( t.left ); System.out.print( t.data + " "); printTree( t.right );

}

}

private int **height**( AVLNode t ) // return height of node t, or -1, if null.

{

if( t == null ) return -1; else return t.height;

}

/\*

Rotate binary tree node with left child. For AVL trees, this is a single rotation. Update heights, then return new root.

\*/

private AVLNode **rotateLeft**( AVLNode node2 )

{

AVLNode node1 = node2.left; node2.left = node1.right; node1.right = node2;

node2.height = Math.max(height(node2.left), height(node2.right))+1; node1.height = Math.max(height(node1.left), node2.height)+1;

return node1;

}

/\*

Rotate binary tree node with right child. For AVL trees, this is a single rotation. Update heights, then return new root.

\*/

private AVLNode **rotateRight**( AVLNode node1 )

{

AVLNode node2 = node1.right; node1.right = node2.left; node2.left = node1;

node1.height = Math.max(height(node1.left), height(node1.right))+1; node2.height = Math.max(height(node2.right), node1.height)+1; return node2;

}

/\*

Double rotate binary tree node: first left child with its right child; then node node3 with new left child.

For AVL trees, this is a double rotation. Update heights, then return new root.

\*/

private AVLNode **doubleLeft**( AVLNode node3 )

{

node3.left = rotateRight( node3.left ); return rotateLeft( node3 );

}

/\*

Double rotate binary tree node: first right child with its left child; then node node1 with new right child.

For AVL trees, this is a double rotation. Update heights, then return new root.

\*/

private AVLNode **doubleRight**( AVLNode node1 )

{

node1.right = rotateLeft( node1.right ); return rotateRight( node1 );

}

}

//////////////////// **AVLTreeDemo.java** ///////////////////////////// class AVLTreeDemo

{

public static void **main**( String [] args )

{

AVLTree avl = new AVLTree();

int[] a = {30,80,50,40,20,60,70,10,90,95};

// build tree by successive insertions of 30, 80, 50, 40 ...

for( int i = 0; i < a.length; i++ ) avl.insert( a[i] );

System.out.println( "\nAVL Tree nodes in sorted order:"); avl.printTree();

avl.insert( 15 );

avl.insert( 85 );

System.out.println( "\n\nAfter insertion of 15 & 85"); avl.printTree();

int item = 82; // search for an item

if( avl.search( item ) )

System.out.println( "\n\n" + item + " found");

else

}

}

System.out.println( "\n\n" + item + " not found");

Output of this program is as follows:

AVL Tree nodes in sorted order: 10 20 30 40 50 60 70 80 90 95

After insertion of 15 & 85

10 15 20 30 40 50 60 70 80 85 90 95

82 not found

## B-Tree

1. Write a Java program to perform the following operations:
   1. Insertion into a B-tree
   2. Deletion from a B-tree

A B-tree of order *m* is an *m*-way tree in which

1. All leaf nodes are on the same level.
2. All nodes, except the root and the leaves, have between [*m*/2] and *m* children.
3. The nonleaf nodes store up to *m*-1 keys to guide the searching; and these keys partition the keys in the children in the fashion of a search tree.
4. The root is either a leaf or has between two and *m* children.
5. If a node has ‘*d*’ number of children, then it must have *d*-1 number of keys.

|  |  |  |  |
| --- | --- | --- | --- |
| **11** | **12** |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **18** | **20** |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **23** | **25** | **27** |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **31** | **33** |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **36** | **39** |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **45** | **47** | **50** | **55** |

**B-Tree of order 5**



**29**

**15**

**21**

**35**

**42**

**Program 29: B-Tree operations** (deletion is not implemented; left as an exercise)

class **BTree**

{

final int MAX = 4; final int MIN = 2;

class **BTNode** // B-Tree node

{

int count;

int key[] = new int[MAX+1];

BTNode child[] = new BTNode[MAX+1];

}

BTNode root = new BTNode();

class Ref // This class creates an object reference

{ int m; } // and is used to retain/save index values

// of current node between method calls.

/\*

* New key is inserted into an appropriate node.
* No node has key equal to new key (duplicate keys are not allowed.

\*/

void **insertTree**( int val )

{

Ref i = new Ref(); BTNode c = new BTNode();

BTNode node = new BTNode(); boolean pushup;

pushup = pushDown( val, root, i, c ); if ( pushup )

{

node.count = 1; node.key[1] = i.m; node.child[0] = root; node.child[1] = c; root = node;

}

}

/\*

* New key is inserted into subtree to which current node points.
* If pushup becomes true, then height of the tree grows.

\*/

boolean **pushDown**( int val, BTNode node, Ref p, BTNode c )

{

Ref k = new Ref();

if ( node == null )

{

p.m = val; c = null;

return true;

}

else

{

if ( searchNode( val, node, k ) ) System.out.println("Key already exists.");

if ( pushDown( val, node.child[k.m], p, c ) )

{

if ( node.count < MAX )

{

pushIn( p.m, c, node, k.m ); return false;

}

else

{

split( p.m, c, node, k.m, p, c ); return true;

}

}

return false;

}

}

/\*

* Search through a B-Tree for a target key in the node: val
* Outputs target node and its position (pos) in the node

\*/

BTNode **searchTree**( int val, BTNode root, Ref pos )

{

if ( root == null ) return null ;

else

{

if ( searchNode( val, root, pos ) ) return root;

else

return searchTree( val, root.child[pos.m], pos );

}

}

/\*

* This method determines if the target key is present in
* the current node, or not. Seraches keys in the current node;
* returns position of the target, or child on which to continue search.

\*/

boolean **searchNode**( int val, BTNode node, Ref pos )

{

if ( val < node.key[1] )

{

pos.m = 0 ; return false ;

}

else

{

pos.m = node.count ;

while ( ( val < node.key[pos.m] ) && pos.m > 1 ) (pos.m)--;

if ( val == node.key[pos.m] ) return true;

else

return false;

}

}

/\*

* Inserts the key into a node, if there is room
* for the insertion

\*/

void **pushIn**( int val, BTNode c, BTNode node, int k )

{

int i ;

for ( i = node.count; i > k ; i-- )

{

node.key[i + 1] = node.key[i]; node.child[i + 1] = node.child[i];

}

node.key[k + 1] = val ;

node.child[k + 1] = c ; node.count++ ;

}

/\*

* Splits a full node into current node and new right child
* with median.

\*/

void **split**( int val, BTNode c, BTNode node, int k, Ref y, BTNode newnode )

{

int i, mid; // mid is median

if ( k <= MIN ) mid = MIN;

else

mid = MIN + 1; newnode = new BTNode();

for ( i = mid+1; i <= MAX; i++ )

{

newnode.key[i-mid] = node.key[i]; newnode.child[i-mid] = node.child[i];

}

newnode.count = MAX - mid; node.count = mid;

if ( k <= MIN ) else

pushIn ( val, c, node, k );

pushIn ( val, c, newnode, k-mid ) ;

y.m = node.key[node.count]; newnode.child[0] = node.child[node.count] ; node.count-- ;

}

// calls display( ) void **displayTree**()

{

display( root );

}

// displays the B-Tree

void display( BTNode root )

{

int i;

if ( root != null )

{

for ( i = 0; i < root.count; i++ )

{

display( root.child[i] ); System.out.print( root.key[i+1] + " " );

}

display( root.child[i] );

}

}

} // end of BTree class

////////////////////////// **BTreeDemo.java** ///////////////////////////// class BTreeDemo

{

public static void main( String[] args )

{

BTree bt = new BTree();

/\*

* Refer Textbook, the section 13.3 B-Trees,
* inserting into a B-Tree
* Figure 13.30: Building a B-tree of order 5

\*/

int[] arr = { 11, 23, 21, 12, 31, 18, 25, 35, 29, 20, 45,

27, 42, 55, 15, 33, 36, 47, 50, 39 };

for ( int i = 0; i < arr.length; i++ ) bt.insertTree( arr[i] );

System.out.println("B-Tree of order 5:"); bt.displayTree();

}

}

Output of this program is as follows:

B-Tree of order 5:

11 12 15 18 20 21 23 25 27 29 31 33 35 36 39 42 45 47 50 55

# Dictionary

1. Write a Java program to implement all the functions of a **dictionary** (ADT) using **Hashing**.

An ADT that supports the operations insert, delete, and search is called a ***dictionary***. Dictionaries are found applications in the design of numerous algorithms. A dictionary is a collection of pairs of key and element. No two pairs in a dictionary have the same key.

Consider a database of books maintained in a library system. When a user wants to check whether a particular book is available, a *search* operation is called for. If the book is available and is issued to the user, a *delete* operation can be performed to remove this book from the set of available books. When the user returns the book, it can be *insert*ed back into the set.

It is essential that we are able to support the above-mentioned operations as efficiently as possible since these operations are performed quite frequently. A number of data structures have been developed to realize a dictionary. These can be broadly divided into comparison methods and direct access methods. Hashing is an example of the latter. Comparison methods fit into binary search trees.

Program 30(a): Dictionary operations using Hash tables

class **Entry**

{ public String key; // word

public String element; // word meaning

public Entry(String k, String e) // constructor

{

key = k; element = e;

}

}

class **HashTable**

{

Entry[] hashArray; // array holds hash table

int size; // table size

int count; // current number of items in the table

public HashTable(int s) // constructor

{

size = s; count = 0;

hashArray = new Entry[size];

}

int **hashFunc**( String theKey ) // convert the string into a numeric key

{

int hashVal=0;

// convert the string into a numeric key

for(int i = 0; i < theKey.length(); i++)

hashVal = 37\*hashVal + (int)theKey.charAt(i); hashVal = hashVal % size;

if(hashVal < 0 )

hashVal = hashVal + size;

return hashVal;

}

public void **insert**(String theKey, String str) // insert a record

{

if( !isFull() )

{

int hashVal = hashFunc(theKey); // hash the key

// until empty cell or null, while(hashArray[hashVal] != null )

{

++hashVal; // go to next cell

hashVal %= size; // wraparound if necessary

}

hashArray[hashVal] = new Entry(theKey, str);

count++; // update count

}

else

System.out.println("Table is full");

}

public Entry **delete**(String theKey) // delete a record with the key

{

if( !isEmpty() )

{

int hashVal = hashFunc(theKey); // hash the key

while(hashArray[hashVal] != null) // until empty cell,

{

if(hashArray[hashVal].key == theKey) // found the key?

{ Entry tmp = hashArray[hashVal]; // save item hashArray[hashVal] = null; // delete item count--;

return tmp; // return item

}

++hashVal; // go to next cell

hashVal %= size; // wraparound if necessary

}

return null; // cannot find item

}

else

System.out.println("Table is empty");

return null;

}

public Entry **search**(String theKey) // find item with key

{

int hashVal = hashFunc(theKey); // hash the key

while(hashArray[hashVal] != null) // until empty cell,

{

if(hashArray[hashVal].key == theKey) // found the key?

return hashArray[hashVal]; // yes, return item

++hashVal; // go to next cell

hashVal %= size; // wraparound if necessary

}

return null; // cannot find item

}

public void **displayTable**()

{

System.out.println("<< Dictionary Table >>\n");

for(int i=0; i<size; i++)

{

if(hashArray[i] != null )

System.out.println( hashArray[i].key + "\t" + hashArray[i].element );

}

}

public boolean isEmpty() // returns true, if table is empty

{

return count == 0;

}

public boolean isFull() // returns true, if table is full

{

return count == size;

}

public int currentSize()

{

return count;

}

}

///////////////////////// **Dictionary.java** //////////////////////////// class **Dictionary**

{

public static void **main**(String[] args)

{

HashTable ht = new HashTable(19); // create hash table of size, 19

// Insert the following items into hash table

ht.insert("man", "gentleman");

ht.insert("watch", "observe");

ht.insert("hope", "expect"); ht.insert("arrange", "put together"); ht.insert("run", "sprint");

ht.insert("wish", "desire"); ht.insert("help", "lend a hand"); ht.insert("insert", "put in");

ht.insert("count", "add up"); ht.insert("format", "arrangement");

ht.displayTable(); // Display the table items

// Search an item

String word = "wish";

Entry item = ht.search(word); if( item != null )

System.out.println("found: " + item.key + "\t" + item.element); else

System.out.println(word + " not found");

// Delete an item

word = "hope";

item = ht.delete(word); if( item != null )

System.out.println("deleted: " + item.key + "\t" + item.element); else

System.out.println(word + " not found - no deletion");

// Current number of items in the table System.out.println("size: " + ht.currentSize());

}

}

Output:

<< Dictionary Table >> insert put in

help lend a hand

man gentleman watch observe format arrangement run sprint

wish desire arrange put together hope expect

count add up

found: wish desire deleted: hope expect size: 9

**Program 30(b): Dictionary operations using java.util.Hashtable**

import java.util.\*; class HashtableDemo

{ public static void main(String[] args)

{

Hashtable<String, String> htab = new Hashtable<String, String>();

// Insert the following items

htab.put("man", "gentleman");

htab.put("watch", "observe");

htab.put("hope", "expect");

htab.put("arrange", "put together"); htab.put("run", "sprint");

htab.put("wish", "desire");

htab.put("help", "lend a hand");

htab.put("insert", "put in");

htab.put("count", "add up"); htab.put("format", "arrangement");

System.out.println(htab); // Display the table items

System.out.println("get(hope): " + htab.get("hope"));

System.out.println("remove(arrange): " + htab.remove("arrange")); System.out.println("remove(help): " + htab.remove("help"));

// returns a set containing all the pairs (key, value).

System.out.println(htab.entrySet());

}

}

Output:

{ arrange=put together, man=gentleman, wish=desire, run=sprint, help=lend a hand, count=add up, watch=observe, hope=expect, format=arrangement, insert=put in }

get(hope): expect remove(arrange): put together remove(help): lend a hand

[ man=gentleman, wish=desire, run=sprint, count=add up, watch=observe, hope=expect, format=arrangement, insert=put in ]

# Graphs

1. Write Java programs for the implementation of *bfs* and *dfs* for a given graph.

*Note*: Refer **chapter 16: Graphs** for more detailed discussion.

## Graph Traversals

A primary problem concerning the graphs is the reachability. A number of graph problems involve traversal of a graph. Traversal of a graph means visiting each of its nodes exactly once. This is accomplished by visiting the nodes in a systematic manner. Two commonly used techniques of graph traversal are depth-first search (DFS) and breadth-first search (BFS). These algorithms work for both directed and undirected graphs.

### Depth-First Search

Depth-first search is a generalization of the preorder traversal of a tree. DFS can serve as a structure around which many other efficient graph algorithms can be built.

As an example of dfs, suppose in the graph of the following figure, we start at node A. A complete program listing is given in Program 31(a).



**A**

**B**

**D**

**B**

**A**

**C**

**C**

**B**

**D**

**D**

**A**

**B**

**E**

**A**

**C X**



**A**

**B**

**D**

**E**

**C**

|  |  |
| --- | --- |
| **E** | **X** |
| **D** | **X** |
| **E** | **X** |
| **C** | **X** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **A** | **B** | **C** | **D** | **E** |
| **A** | **0** | **1** | **0** | **1** | **1** |
| **B** | **1** | **0** | **1** | **1** | **0** |
| **C** | **0** | **1** | **0** | **1** | **1** |
| **D** | **1** | **1** | **1** | **0** | **0** |
| **E** | **1** | **0** | **1** | **0** | **0** |

Undirected Graph Adjacency list Adjacency matrix

Program 31(a): Depth-First Search

class **Node**

{ int label; // vertex label

Node next; // next node in list

Node( int b ) // constructor

{ label = b; }

}

class **Graph**

{ int size;

Node adjList[]; int mark[];

Graph(int n) // constructor

{ size = n;

adjList = new Node[size];

mark = new int[size]; // elements of mark are initialized to 0

}

public void **createAdjList**(int a[][]) // create adjacent lists

{

Node p; int i, k;

for( i = 0; i < size; i++ )

{ p = adjList[i] = new Node(i); //create first node of ith adj. list

for( k = 0; k < size; k++ )

{ if( a[i][k] == 1 )

{ p.next = new Node(k); // create next node of ith adj. list

p = p.next;

}

}

}

}

public void **dfs**(int head) // recursive depth-first search

{ Node w; int v; mark[head] = 1;

System.out.print( head + " "); w = adjList[head];

while( w != null)

{ v = w.label;

if( mark[v] == 0 ) dfs(v); w = w.next;

}

}

}

///////////////////////// **DfsDemo.java** //////////////////////// class DfsDemo

{ public static void **main**(String[] args)

{ Graph g = new Graph(5); // graph is created with 5 nodes

int a[][] = { {0,1,0,1,1}, {1,0,1,1,0}, {0,1,0,1,1},

{1,1,1,0,0}, {1,0,1,0,0}};

g.createAdjList(a);

g.dfs(0); // starting node to dfs is 0 (i.e., A)

}

}

**Output of this program is**: 0 1 2 3 4

Here, 0 is for A, 1 is for B, 2 is for C, 3 is for D, and 4 is for E

### Breadth-First Search

Another systematic way of visiting the nodes is called *breadth-first search* (bfs). The approach is called “breadth-first” because from each node *v* that we visit we search as broadly as possible by next visiting all nodes adjacent to *v*.

The breadthFirstSearch algorithm inserts a node into a queue, which we assume is initially empty. Every entry in the array *mark* is assumed to be initialized to the value *unvisited*. If the graph is not connected, bfs must be called on a node of each connected component. Note that in bfs we must mark a node visited before inserting it into the queue, to avoid placing it on the queue more than once. The algorithm terminates when the queue becomes empty.

8. Graphs 83

We assume the graph is represented by an adjacency list and is globally available. The algorithm depends on the availability of an implementation of a queue of integers (here, a simple queue is assumed; not circular queue). Three methods relating to queue: *qinsert()*, *qdelete()*, and *isEmpty()* are to be properly defined before the method for bfs is defined.

**Program 31(b): Breadth-First Search**

class **Node**

{ int label; // vertex label

Node next; // next node in list

Node( int b ) // constructor

{ label = b; }

}

class **Graph**

{ int size;

Node adjList[]; int mark[];

Graph(int n) // constructor

{ size = n;

adjList = new Node[size]; mark = new int[size];

}

public void **createAdjList**(int a[][]) // create adjacent lists

{ Node p; int i, k;

for( i = 0; i < size; i++ )

{ p = adjList[i] = new Node(i);

for( k = 0; k < size; k++ )

{ if( a[i][k] == 1 )

{ p.next = new Node(k); p = p.next;

}

} // end of inner for-loop

} // end of outer for-loop

} // end of *createAdjList*()

public void **bfs**(int head)

{ int v; Node adj;

Queue q = new Queue(size); v = head;

mark[v] = 1; System.out.print(v + " "); q.qinsert(v);

while( !q.IsEmpty() ) // while(queue not empty)

{

v = q.qdelete(); adj = adjList[v];

while( adj != null )

{ v = adj.label;

if( mark[v] == 0 )

{ q.qinsert(v); mark[v] = 1;

System.out.print(v + " ");

}

adj = adj.next;

}

}

}

} // end of Graph class

class **Queue**

{ private int maxSize; // max queue size

private int[] que; // que is an array of integers

private int front; private int rear;

private int count; // count of items in queue

public Queue(int s) // constructor

{ maxSize = s;

que = new int[maxSize]; front = rear = -1;

}

public void **qinsert**(int item)

{ if( rear == maxSize-1 ) System.out.println("Queue is Full");

else { rear = rear + 1;

que[rear] = item;

if( front == -1 ) front = 0;

}

}

public int **qdelete**()

{ int item;

if( IsEmpty() )

{ System.out.println("\n Queue is Empty"); return(-1);

}

item = que[front];

if( front == rear ) front = rear = -1; else front = front+1;

return(item);

}

public boolean **IsEmpty**()

{ return( front == -1 ); }

} // end of Queue class

////////////////////////////////////////////////////////// class BfsDemo

{ public static void **main**(String[] args)

{ Graph g = new Graph(5);

int a[][] = { {0,1,0,1,1}, {1,0,1,1,0}, {0,1,0,1,1},

{1,1,1,0,0}, {1,0,1,0,0}};

g.createAdjList(a); g.bfs(0);

}

}

**Output of this program is:** 0 1 3 4 2

# Sorting

1. Write Java programs for implementing the following sorting methods:
   1. Bubble sort
   2. Selection sort
   3. Insertion sort
   4. Quick sort
   5. Merge sort
   6. Heap sort
   7. Radix sort
   8. Binary tree sort

As soon as you create an important database, you will probably think of reasons to sort it in various ways. You need to arrange names in alphabetical order, students by marks, customers by pin code, cities in order of increasing population, countries by GNP, and so on.

Sorting data may also be a preliminary step to searching it. As we saw in the previous chapter, a binary search, which can be applied only to sorted data, is much faster than a linear search.

Because sorting is so important and potentially so time-consuming, it has been the subject of extensive research in computer science, and some very sophisticated methods have been developed. First we will look at three of the simpler algorithms: the bubble sort, the selection sort, and the insertion sort. Besides being easier to understand, they are actually better in some circumstances than the more sophisticated algorithms. The insertion sort, for example, is preferable to quick sort for small files and for almost-sorted files. In fact, an insertion sort is commonly used as a part of a quick sort implementation.

## Bubble Sort

The familiar sorting procedure is the *bubble sort* (*exchange sort*). This is the widely known among beginning students of programming and easy to understand and program. Of course, it is probably the least efficient.

We consider an array, *a*i of size, *n*. When this approach is used, there are at most *n*-1 passes are required. During the first pass, *a*0 and *a*1 are compared, and if they are out of order, then *a*0 and *a*1 are interchanged; this process is repeated for *a*1 and *a*2, *a*2 and *a*3, and so on. This method will cause small elements to move or “bubble up”. After the first pass, the array with the largest element will be in the position *n*-1 (*last* location). On each successive pass, the array with the next largest element will be placed in position *n*-2, *n*-3,.., 1, respectively, thereby resulting in a sorted array.

After each pass through the array, a check can be made to determine whether any interchanges were made during that pass. If no interchanges occurred, then the array must be sorted and no further passes are required. An example of bubble sort for the array {27, 49, 35, 37, 15, 75, 63, 65} is illustrated in the Table 9.1.

Program 9(a): Bubble Sort

void bubbleSort(Object[] a)

{

int i, pass, exch, n = a.length; Object tmp;

for( pass = 0; pass < n; pass++ )

{ exch = 0;

for( i = 0; i < n-pass-1; i++ )

if( ((Comparable)a[i]).compareTo(a[i+1]) > 0)

{ tmp = a[i];

a[i] = a[i+1]; a[i+1] = tmp; exch++;

}

if( exch == 0 ) return; // sorting finished – return early

}

}

**Table 9.1**: Trace of Bubble sort (elements to be interchanged are shown in **red** colour)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***pass*** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | ***Process*** |
| - | 27 | **49** | **35** | 37 | 15 | 75 | 63 | 65 | Original array |
| 1 | 27 | 35 | **49** | **37** | 15 | 75 | 63 | 65 | 49, 35 interchanged |
| 1 | 27 | 35 | 37 | **49** | **15** | 75 | 63 | 65 | 49, 37 interchanged |
| 1 | 27 | 35 | 37 | 15 | 49 | **75** | **63** | 65 | 49, 15 interchanged |
| 1 | 27 | 35 | 37 | 15 | 49 | 63 | **75** | **65** | 75, 63 interchanged |
| 1 | 27 | 35 | **37** | **15** | 49 | 63 | 65 | 75 | 75, 65 interchanged |
| 2 | 27 | **35** | **15** | 37 | 49 | 63 | 65 | 75 | 37, 15 interchanged |
| 3 | **27** | **15** | 35 | 37 | 49 | 63 | 65 | 75 | 35, 15 interchanged |
| 4 | 15 | 27 | 35 | 37 | 49 | 63 | 65 | 75 | 27, 15 interchanged |

## Selection Sort

One of the easiest ways to sort a list is by *selection*. Beginning with the first element in the array, a search is performed to locate the smallest element. When this item is found, it is exchanged with the first element. This interchange places the smallest element in the first position of the array. A search for the second smallest element is then carried out. Examining the items from second position onward does this. The smallest element is exchanged with the item in second position. This process continues until all elements of the array have been sorted in ascending order. The following is the selection sort algorithm.

An example of the selection sort is given in the Table 9.2. The second row (*pass* 0) of the table shows the original unordered array.

**Table 9.2**: Trace of Selection sort (elements to be interchanged are shown in **red** colour)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **0 1 2 3 4 5 6 7** | ***pass*** | ***min*** | ***a*[*pass*]** | ***a*[*min*]** | **swap *a*[*pass*], *a*[*min*]** |
| **49** 27 65 37 **15** 75 63 60 | 0 | 4 | 49 | 15 | 49, 15 |
| 15 27 65 37 49 75 63 60 | 1 | 1 | 27 | 27 | *min* = *pass*  no exchange |
| 15 27 **65 37** 49 75 63 60 | 2 | 3 | 65 | 37 | 65, 37 |
| 15 27 37 **65 49** 75 63 60 | 3 | 4 | 65 | 49 | 65, 49 |
| 15 27 37 49 **65** 75 63 **60** | 4 | 7 | 65 | 60 | 65, 60 |
| 15 27 37 49 60 **75 63** 65 | 5 | 6 | 75 | 63 | 75, 63 |
| 15 27 37 49 60 63 **75 65** | 6 | 7 | 75 | 65 | 75, 65 |
| 15 27 37 49 60 63 65 75 |  |  |  |  | Sorted list |

Program 9(b): Selection Sort

class SelectionSortDemo

{

public static void main(String[] args)

{

int[] arr = { 49, 27, 65, 37, 15, 75, 63, 60 };

System.out.print("\n Unsorted array: "); display( arr );

selectionSort( arr );

System.out.print("\n Sorted array: "); display( arr );

}

static void **selectionSort**( int a[] )

{

int n = a.length;

for( int pass = 0; pass < n-1; pass++ )

{

int min = pass;

for( int i = pass+1; i < n; i++ ) if( a[i] < a[min] ) min = i;

if( min != pass )

{

int tmp = a[min]; a[min] = a[pass]; a[pass] = tmp;

}

}

}

static void display( int a[] )

{

for( int i = 0; i < a.length; i++ ) System.out.print( a[i] + " " );

}

}

Following output is generated from this program:

Unsorted array: 49 27 65 37 15 75 63 60

Sorted array: 15 27 37 49 60 63 65 75

## Insertion Sort

Insertion sort works very fast on small size arrays. The insertion sort procedure scans array, *a* from *a*[0] to *a*[*n*-1], inserting each element *a*[*j*] into its proper position in the previously sorted sub-array *a*[0], *a*[1], …, *a*[*j*-1]. We consider an array of six elements shown in Table 9.3.

**Table 9.3**: Trace of Insertion Sort (inserted elements are shown in red)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ***pass*** | ***a*[0]** | ***a*[1]** | ***a*[2]** | ***a*[3]** | ***a*[4]** | ***a*[5]** | ***Process*** |
|  | 65 | 50 | 30 | 35 | 25 | 45 | Original array |
| 1 | **50** | 65 | 30 | 35 | 25 | 45 | 50 is inserted |
| 2 | **30** | 50 | 65 | 35 | 25 | 45 | 30 is inserted |
| 3 | 30 | **35** | 50 | 65 | 25 | 45 | 35 is inserted |
| 4 | **25** | 30 | 35 | 50 | 65 | 45 | 25 is inserted |
| 5 | 25 | 30 | 35 | **45** | 50 | 65 | 45 is inserted |

Program 9(c): Insertion sort method

void insertionSort(Object a[])

{

int i, j, n = a.length; Object item;

for( j = 1; j < n; j++ ) // repeat loop starting from a[1] to a[n-1]

{ item = a[j]; // element to be inserted

i = j-1;

while( i >= 0 && ((Comparable)item).compareTo(a[i]) < 0)

{

a[i+1] = a[i]; // shift element to the right

i = i-1;

}

a[i+1] = item; // insert element in proper position

}

}

## Quick Sort

The algorithm solely depends on the data it receives. If the data has certain properties, quick sort is one of the fastest, if not; quick sort can be very slow. Quick sort can perform quite fast, on average about *O*(*n* log *n*), but its worst case is a degrading *O*(*n*2). For quick sort, the worst case is usually when the data is already sorted.

Quick sort is naturally recursive. We partition the array into two sub-arrays, and then re-start the algorithm on each of these sub-arrays. The partition procedure involves choosing some object (usually, already in the array); If some other object is greater than the chosen object, it is added to one of the sub-arrays, if it is less than the chosen object, it is added to another sub-array. Thus, the entire array is partitioned into two sub-arrays, with one sub-array having everything that is larger than the chosen object, and the other sub-array having everything that is smaller.

Variable *a* is an integer array of size, *n*. The *left* and *right* invoke procedure and they are initialized with 0 and *n* -1 respectively; and are the current lower and upper bounds of the sub-arrays. The indices *newleft* and *newright* are used to select certain elements during the processing of each sub-array. Variable *amid* is the element which is placed in its final location in the array.

Index *left* scans the list from left to right, and index *right* scans the list from right to left. A swap is performed when left is at an element larger than the *pivot* and right is at an element smaller than the

*pivot*. A final swap with the pivot completes the divide step. The *pivot* element is placed in its final proper position in the array. We take the *pivot* as the middle element of the array (or sub-array). **Table**

* 1. illustrates the trace of Quick sort.

Program 9(d): Quick Sort

class QuickSortDemo

{

public static void main(String[] args)

{

int[] arr = { 65, 35, 15, 90, 75, 45,40, 60, 95, 25, 85, 55 };

System.out.print("\n Unsorted array: "); display( arr );

quickSort( arr, 0, arr.length-1 );

System.out.print("\n Sorted array: "); display( arr );

}

static void **quickSort**(int a[], int left, int right)

{

int newleft = left, newright = right; int amid, tmp;

amid = a[(left + right)/2]; // pivot is amid

do // do-while-loop

{

while( (a[newleft] < amid) && (newleft < right)) newleft++;

while( (amid < a[newright]) && (newright > left)) newright--;

if(newleft <= newright)

{ tmp = a[newleft]; a[newleft] = a[newright]; a[newright] = tmp; newleft++; newright--;

}

} while(newleft <= newright); // end of do-while-loop

if(left < newright) quickSort(a, left, newright); if(newleft < right) quickSort(a, newleft, right);

}

static void display( int a[] )

{

for( int i = 0; i < a.length; i++ ) System.out.print( a[i] + " " );

}

}

Output:

|  |  |  |
| --- | --- | --- |
| Unsorted array: 65 35 15 90 | 75 45 40 60 95 25 85 | 55 |
| Sorted array: 15 25 35 40 | 45 55 60 65 75 85 90 | 95 |

**Table 9.4**: Trace of Quick sort

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ***step*** | ***a*0** | ***a*1** | ***a*2** | ***a*3** | ***a*4** | ***a*5** | ***a*6** | ***a*7** | ***a*8** | ***a*9** | ***a*10** | ***a*11** | ***left*** | ***right*** | ***pivot*** |
| 0 | 65 | 35 | 15 | 90 | 75 | **45** | 40 | 60 | 95 | 25 | 85 | 55 | 0 | 11 | **45** |
| 1 | 25 | 35 | **15** | 40 | *45* | 75 | 90 | 60 | 95 | 65 | 85 | 55 | 0 | 4 | **15** |
| 2 | *15* | 35 | **25** | 40 | *45* | 75 | 90 | 60 | 95 | 65 | 85 | 55 | 1 | 4 | **25** |
| 3 | *15* | *25* | 35 | **40** | *45* | 75 | 90 | 60 | 95 | 65 | 85 | 55 | 2 | 4 | **40** |
| 4 | *15* | *25* | *35* | *40* | *45* | 75 | 90 | 60 | **95** | 65 | 85 | 55 | 5 | 11 | **95** |
| 5 | *15* | *25* | *35* | *40* | *45* | 75 | 90 | **60** | 55 | 65 | 85 | *95* | 5 | 10 | **60** |
| 6 | *15* | *25* | *35* | *40* | *45* | **55** | *60* | 90 | 75 | 65 | 85 | *95* | 5 | 6 | **55** |
| 7 | *15* | *25* | *35* | *40* | *45* | *55* | *60* | 90 | **75** | 65 | 85 | *95* | 7 | 10 | **75** |
| 8 | *15* | *25* | *35* | *40* | *45* | *55* | *60* | *65* | *75* | **90** | 85 | *95* | 9 | 10 | **90** |
| 9 | *15* | *25* | *35* | *40* | *45* | *55* | *60* | *65* | *75* | *85* | *90* | *95* | sorted array | | |

## Merge Sort

Merging is the combination of two or more sorted arrays (or sequences) into a single sorted array. Following figure illustrates the basic, two-way merge operation. In a two-way merge, two sorted sequences are merged into one. Clearly, two sorted sequences each of length *n* can be merged into a sorted sequence of length 2*n* in *O*(2*n*)=*O*(*n*) steps. However in order to do this, we need space in which to store the result. That is, it is not possible to merge the two sequences *in place* in *O*(*n*) steps.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **10** | **40** | **50** | **75** | **90** |

|  |  |  |  |
| --- | --- | --- | --- |
| **15** | **55** | **65** | **95** |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **10** | **15** | **40** | **50** | **55** | **65** | **75** | **90** | **95** |

Sorting by merging is a recursive, divide-and-conquer strategy. In the base case, we have a sequence with exactly one element in it. Since such a sequence is already sorted, there is nothing to be done. To sort a sequence of *n >*1 elements:

* Divide the sequence into two sequences of length ⎡*n* / 2⎤ and ⎣*n* / 2⎦ ;
* Recursively sort each of the two subsequences; and then,
* Merge the sorted subsequences to obtain the final result.

Following figure illustrates the operation of the two-way merge sort algorithm. We assume to sort the given array *a*[*n*] into ascending order. We split it into two subarrays: *a*[0]…*a*[*n*/2] and *a*[(*n*/2)+1]…*a*[*n*-1]. Each subarray is individually sorted, and the resulting sorted subarrays are merged to produce a single sorted array of *n* elements. Consider an array of nine elements: {75, 40, 10, 90, 50, 95, 55, 15, 65}. The **sort()** method divides the array into subarrays and merges them into sorted subarrays by **merge()** method as illustrated in the figure (Dashed-line arrows indicate the process of splitting and the regular arrows the merging process).

Program 9(e): Merge sort

class **MergeSort**

{ int[] a;

int[] tmp;

MergeSort(int[] arr)

{ a = arr;

tmp = new int[a.length];

}

void **msort**()

{ sort(0, a.length-1); }

void **sort**(int left, int right)

{

if(left < right)

{

int mid = (left+right)/2; sort(left, mid); sort(mid+1, right); merge(left, mid, right);

}

}

void **merge**(int left, int mid, int right)

{ int i = left; int j = left; int k = mid+1;

while( j <= mid && k <= right )

{

if(a[j] < a[k])

tmp[i++] = a[j++]; else

tmp[i++] = a[k++];

}

while( j <= mid ) tmp[i++] = a[j++];

for(i=left; i < k; i++) a[i] = tmp[i];

}

}

//////////////////// **MergeSortDemo.java** /////////////////////// class MergeSortDemo

{ public static void main(String[] args)

{ int[] arr = {75, 40, 10, 90, 50, 95, 55, 15, 65};

MergeSort ms = new MergeSort(arr); ms.msort(); System.out.print("Sorted array:"); for( int i=0; i < arr.length; i++)

System.out.print(arr[i] + " ");

}

}

Output of this program is:

Sorted array: 10 15 40 50 55 65 75 90 95

## Heap Sort

The efficiency of the heap data structure provides itself to a surprisingly simple and very efficient sorting algorithm called *heapsort*. The *heap sort* algorithm uses a *heap tree* to sort an array either in ascending or descending order. It consists of two phases:

* + 1. Using unsorted data array, build the heap by repeatedly inserting each element into the heap.
    2. Remove the top element (item at the root) from the heap, and place it in the last location of the array. Rebuild the heap from the remaining elements of the array. This process is repeated until all items from the heap are deleted. Finally, the array is in sorted order.

A *heap* is a complete binary tree such that the root is the largest item (max-heap). The ordering in a heap is top-down, but not left-to-right. Each root is greater than or equal to each of its children, but some left siblings may be greater than their right siblings and some may be less. Since heaps are typically stored as arrays, we can apply the heap operations to an array of integers. We illustrate the operations as though they are being performed on binary trees, while they are really defined for the arrays that represent them by natural mapping.

Program 9(f): Heap sort

class **Heap**

{ int[] a; // heap array

int maxSize; // size of array

int currentSize; // number of nodes in array

public Heap(int m) // constructor

{ maxSize = m; currentSize = 0;

a = new int[maxSize]; // create heap array

}

public boolean **insert**(int key)

{

if(currentSize == maxSize) // if array is full,

return false; // return without insertion a[currentSize] = key; // store the key in the heap array moveUp(currentSize++); // move it up

return true; // insertion is successful

}

public void **moveUp**(int index)

{ int parent = (index-1)/2; int bottom = a[index];

while(index > 0 && a[parent] < bottom)

{ a[index] = a[parent]; // move node down index = parent; // move index up parent = (parent-1)/2; // parent <<< its parent

}

a[index] = bottom;

}

public int **remove**()

{

if( isEmpty() )

{ System.out.println("Heap is empty"); return -1;

}

int root = a[0]; // save the root

a[0] = a[--currentSize]; // root <<< last node

moveDown(0); // move down the root

return root; // return deleted item

}

public void **moveDown**(int index)

{ int largerChild;

int top = a[index]; // save root in top

while(index < currentSize/2) // while node has at least one child

{ int leftChild = 2\*index+1; int rightChild = 2\*index+2;

// find larger child

if(rightChild<currentSize && a[leftChild]<a[rightChild]) largerChild = rightChild;

else

largerChild = leftChild; if(top >= a[largerChild]) break;

a[index] = a[largerChild]; // shift child up

index = largerChild; // go down

}

a[index] = top; // root to index

}

public boolean **isEmpty**()

{ return currentSize==0; }

public void **displayHeap**(int[] a)

{ for(int i=0; i < maxSize; i++) System.out.print(" [" + i + "] ");

System.out.println();

for(int i=0; i < maxSize; i++) System.out.print(" " + a[i] + " ");

}

}

//////////////////////// **HeapsortDemo.java** /////////////////////// class HeapsortDemo

{

public static void **main**(String[] args)

{

int[] arr = { 50, 20, 30, 10, 40, 70, 60 };

Heap h = new Heap(arr.length); // create a Heap

// Build heap by repeated insertions

for(int i = 0; i < arr.length; i++) h.insert(arr[i]);

// delete and copy heap’s top item

for( int i = arr.length-1; i >= 0; i-- ) arr[i] = h.remove();

System.out.println("\nSorted array: "); h.displayHeap(arr);

}

}

Output:

Sorted array:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| [0] [1] | [2] | [3] | [4] | [5] | [6] |
| 10 20 | 30 | 40 | 50 | 60 | 70 |

## Radix Sort

This sorting algorithm is known as *least-significant-digit-first radix sorting*. Radix sorting is practical for much larger universal sets. Radix sorting can be used when each element of the universal set can be viewed as a sequence of digits (or letters or any other symbols).

An implementation of radix sort follows. The integer **radix** is assigned a value **10** and integer **maxDigits** is also assigned the maximum number of digits for the maximum element of the array. This algorithm does not depend on comparison of array elements. For each array element, two operations are performed:

1. To get the significant digit (**m**) of the number, the array element is divided by the **divisor**, and division modulo **radix(10)**.
2. This array element is inserted in the **m**th queue.

In each pass, all array elements are added to the queues, and then the elements are deleted from all the queues and stored back to array. The method requires additional space for queues. The ten queues are implemented by linked lists using **java.util.LinkedList** class.

Program 9(g): Radix sort

class RadixSortDemo

{

public static void **main**(String[] args)

{

int[] a = { 3305, 99, 52367, 125, 10, 12345, 7, 35, 7509, 3, 345 };

radixSort(a, 10, 5); System.out.println("Sorted list: ");

for(int i = 0; i < a.length; i++ ) System.out.print( a[i] + " ");

}

static void **radixSort**(int[] arr, int radix, int maxDigits)

{

int d, j, k, m, divisor; java.util.LinkedList[] queue

= new java.util.LinkedList[radix];

for( d = 0; d < radix; d++ )

queue[d] = new java.util.LinkedList(); divisor = 1;

for(d = 1; d <= maxDigits; d++) // Pass: 1, 2, 3, . . .

{

for(j = 0; j < arr.length; j++)

{

m = (arr[j]/divisor) % radix; queue[m].addLast(new Integer(arr[j]));

}

divisor = divisor\*radix; // 1, 10, 100, ...

for(j = k = 0; j < radix; j++)

{

while( !queue[j].isEmpty())

arr[k++] = (Integer)queue[j].removeFirst();

}

}

}

}

Output:

Sorted list:

3 7 10 35 99 125 345 3305 7509 12345 52367

## Binary Tree Sort

When we traverse a *binary search tree* in *inorder*, the keys will come out in sorted order. In *tree sort*, We take the array to be sorted, use the method buildTree() to construct the array elements into a binary search tree, and use inorder traversal to put them out in sorted order.

Program 9(h): Binary Tree Sort

class **BSTNode**

{ int data; BSTNode left; BSTNode right;

BSTNode( int d ) // constructor

{ data = d; }

}

class **BinarySearchTree**

{

int i; int[] a;

BSTNode root;

BinarySearchTree(int[] arr) // constructor

{

a = new int[arr.length]; a = arr;

}

private void **buildTree**()

{

for( i = 0; i < a.length; i++ )

root = insertTree( root, a[i] );

}

private BSTNode **insertTree**(BSTNode p, int key)

{

if( p == null )

p = new BSTNode(key); else if( key < p.data)

p.left = insertTree( p.left, key); else p.right = insertTree( p.right, key); return p;

}

public void **treeSort**()

{

buildTree(); i = 0;

inorder(root);

}

private void **inorder**(BSTNode p) // 'p' starts with root

{

if( p != null )

{ inorder(p.left); a[i++] = p.data; inorder(p.right);

}

}

public void **display**()

{

for( i = 0; i < a.length; i++ ) System.out.print(a[i] + " " );

}

}

//////////////////////// **TreeSortDemo.java** ////////////////////////// class TreeSortDemo

{

public static void **main**(String args[])

{

int arr[] = { 55, 22, 99, 77, 11, 88, 44, 66, 33 };

BinarySearchTree bst = new BinarySearchTree(arr);

bst.treeSort();

System.out.print("Sorted array: "); bst.display();

}

}

Output:

Sorted array: 11 22 33 44 55 66 77 88 99

# KMP Algorithm

1. Write a Java program for implementing KMP pattern matching algorithm.

The KMP algorithm compares the pattern to the text in *left-to-right*, but shifts the pattern, *P* more intelligently than the brute-force algorithm. When a mismatch occurs, what is the *most* we can shift the pattern so as to avoid redundant comparisons. The answer is that the largest prefix of *P*[0..*j*] that is a suffix of *P*[1..*j*].

Program 33: KMP pattern matching

class KMPDemo

{

public static void main(String[] args)

{

String T = "THE RIVER MISSISSIPPI FLOWS IN NORTH AMERICA.";

String P = "SSIPP";

boolean isMatch = kmp(T, P); if(isMatch)

System.out.println("Pattern " + P

+ " is present in text: " + T);

else

System.out.println("Pattern " + P

+ " is not present in text: " + T);

}

static boolean kmp(String T, String P)

{

int n = T.length(); int m = P.length();

int[] fail = computeFailFunction(P); int i = 0; // text index

int j = 0; // pattern index

while( i < n )

{

if( P.charAt(j) == T.charAt(i) )

{

if( j == m-1 ) return true; i++;

j++;

}

else

if( j > 0 ) j = fail[j-1]; else i++;

}

return false;

}

static int[] computeFailFunction( String P )

{

int m = P.length(); int[] fail = new int[m]; fail[0] = 0;

int i = 1;

10. KMP Algorithm 99

int j = 0; while( i < m )

{

if( P.charAt(j) == P.charAt(i) ) // j+1 characters match

{

fail[i] = j+1; i++;

j++;

}

else if( j > 0 ) // j follows a matching prefix

j = fail[j-1];

else // no match

{ fail[i] = 0; i++;

}

}

return fail;

}

}

Output:

**Pattern SSIPP is present in text:**

**THE RIVER MISSISSIPPI FLOWS IN NORTH AMERICA**