Computer Science Department

Laboratory Workbook

COMP242

Data Structure and Algorithm

Prepared by:

Mr. Iyad Jaber

Approved By:

Computer Science Department

2015/2016

Table of Contents

Lab 1: Recursion

AIM: To write a program that implements a recursive functions.

Recursion

Recursion is a programming technique that allows the programmer to express operations in terms of themselves. In Java, this takes the form of a function that calls itself. A useful way to think of recursive functions is to imagine them as a process being performed where one of the instructions is to "repeat the process". This makes it sound very similar to a loop because it repeats the same code, and in some ways it is similar to looping. On the other hand, recursion makes it easier to express ideas in which the result of the recursive call is necessary to complete the task. Of course, it must be possible for the "process" to sometimes be completed without the recursive call. One simple example is the idea of building a wall that is ten feet high; if I want to build a ten foot high wall, then I will first build a 9 foot high wall, and then add an extra foot of bricks. Conceptually, this is like saying the "build wall" function takes a height and if that height is greater than one, first calls itself to build a lower wall, and then adds one a foot of bricks.

Example

Fibonacci sequence

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 nth 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 1, with iterative and recursive methods, and tested by main() driver.

Program 1: 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));
    }
}
```

Exercises

Question 1

Write a Recursive Function to print number from Given input down to 0.

Question 2

Use a recursive function to write a program that performs the following tasks.

1. Find The Factorial Number for an input n.
2. Check If a Given Array Values Are Palindrome or not.

Question 3

(**Anagrams**) Here's a different kind of situation in which recursion provides a neat solution to a problem. Suppose you want to list all the anagrams of a specified word; that is, all possible letter combinations (whether they make a real English word or not) that can be made from the letters of the original word. We'll call this anagramming a word.

Anagramming cat, for example, would produce

- cat
- cta
- atc
- act
- tca
- tac

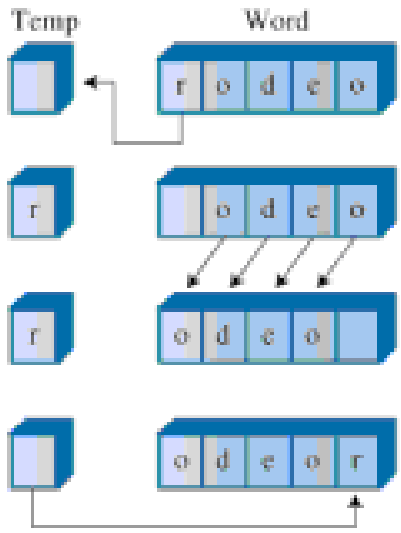
Try anagramming some words yourself. You'll find that the number of possibilities is the factorial of the number of letters. For 3 letters there are 6 possible words, for 4 letters there are 24 words, for 5 letters 120 words, and so on. (This assumes that all letters are distinct; if there are multiple instances of the same letter, there will be fewer possible words.)

How would you write a program to anagram a word? Here's one approach. Assume the word has n letters.

1. Anagram the rightmost n–1 letters.
2. Rotate all n letters.
3. Repeat these steps n times.

To rotate the word means to shift all the letters one position left, except for the leftmost letter, which "rotates" back to the right, as shown in Figure below.

Rotating the word n times gives each letter a chance to begin the word. While the selected letter occupies this first position, all the other letters are then anagrammed (arranged in every possible position). For cat, which has only 3 letters, rotating the remaining 2 letters simply switches them.



Question 4

Write a recursive function to reverse a string. Write a recursive function to reverse the words in a string, i.e., "cat is running" becomes "running is cat".

Question 5

Write a recursive Java method that counts the number of occurrences of the character 'a' in a string. Hint: a method signature that works is public static int countA (String s).

You can test your method in Eclipse. Consider using the charAt or startsWith methods in String.

1. Prove that your method is correct.
2. Prove that your method terminates.

Lab 2: Singly Linked List Implementation of the List ADT

AIM: To write a program that implements the basic operations of the linked list in Java.

Linked List Overview

- Static memory allocation may insert in wastage of memory and also shortage of memory. (Like in array implementations)
- To outcome these problems, dynamic memory allocation is used.
- Linked list is a linear data structure which consists of number of nodes created dynamically.
- A linked list has no fix size but grows or shrinks as elements are added or deleted from the list.
- A singly linked list is a dynamic data structure. It may grow or shrink. Growing or shrinking depends on operation made.
- Every node of list has two fields, information and next address. Next address field contains memory address location of next node.

Operations on Singly Linked List:

- **Algorithm for inserting a node at the beginning**
    1. Allocate memory for the new node, temp = new Node ().
    2. Assign the value of the data field of the new node.
    3. Make the link field of the new node to point to the starting node of the linked list, temp.next=start.
    4. Then, set the external pointer (which was pointing to the starting node) to point to the new node, start=temp.

- **Algorithm for inserting a node at the end**
    1. If the list is empty then create the new node, temp = new Node ().
    2. If the list is not empty, then go to the last node and then insert the new node after the last node, r=start; go till last node then r.next=temp.

- **Algorithm for inserting node at the specified position**
    1. Allocate memory for the new node, temp= new Node ().

2. Assign value to the data field of the new node, temp.info.
3. Go till node R, also mark next node as S.
4. Make the next field of node temp to node S and next field of node R to point to new node.

- **Algorithm for deleting the first node**
    1. Mark the first node as temp.
    2. Then, make start point to the next node.
    3. Make next field of temp, null.
    4. Free temp.

- **Algorithm for deleting the last node**
    1. Mark the last node as temp & last but one node as R.
    2. Make the next field of R as NULL.
    3. Free temp.

- **Algorithm for deleting a middle node**
    1. Mark the node to be deleted as temp.
    2. Mark the previous node as R & next nodes as S.
    3. Make the next field of R to point S.
    4. Make the next field of temp, NULL.
    5. Free Temp.

Definition of a class Link

```
class Link
{
        public int iData; // data
        public double dData; // data
        public Link next; // reference to next link
}
```

A Simple Linked List

Our first example program, linkList.java, demonstrates a simple linked list. The only operations allowed in this version of a list are

- Inserting an item at the beginning of the list
- Deleting the item at the beginning of the list
- Iterating through the list to display its contents

These operations are fairly easy to carry out, so we'll start with them. (As we'll see later, these operations are also all you need to use a linked list as the basis for a stack.) Before we get to the complete linkList.java program, we'll look at some important parts of the Link and LinkList classes.

The LinkList Class

The LinkList class contains only one data item: a reference to the first link on the list. This reference is called first. It's the only permanent information the list maintains about the location of any of the links. It finds the other links by following the chain of references from first, using each link's next field.

```
class LinkList
{
        private Link first; // ref to first link on list
        // -------------------------------------------------------------
        public void LinkList() // constructor
        {
                first = null; // no items on list yet
        }
        // -------------------------------------------------------------
        public boolean isEmpty() // true if list is empty
        {
                return (first==null);
        }
        // -------------------------------------------------------------
        ...                             // other methods go here
}
```

The constructor for LinkList sets first to null. This isn't really necessary because as we noted, references are set to null automatically when they're created. However, the explicit constructor makes it clear that this is how first begins. When first has the value null, we

know there are no items on the list. If there were any items, first would contain a reference to the first one. The isEmpty() method uses this fact to determine whether the list is empty.

The insertFirst() Method

The insertFirst() method of LinkList inserts a new link at the beginning of the list. This is the easiest place to insert a link, because first already points to the first link. To insert the new link, we need only set the next field in the newly created link to point to the old first link, and then change first so it points to the newly created link.

In insertFirst() we begin by creating the new link using the data passed as arguments. Then we change the link references as we just noted.

```
// insert at start of list
public void insertFirst(int id, double dd)
{
        // make new link
        Link newLink = new Link(id, dd);
        newLink.next = first;
        first = newLink;
}
```

The deleteFirst() Method

The deleteFirst() method is the reverse of insertFirst(). It disconnects the first link by rerouting first to point to the second link. This second link is found by looking at the next field in the first link.

```
public Link deleteFirst() // delete first item
{
        // (assumes list not empty)
        Link temp = first; // save reference to link
        first = first.next;
        return temp; // return deleted link
}
```

The displayList() Method

To display the list, you start at first and follow the chain of references from link to link. A variable current points to (or technically refers to) each link in turn. It starts off pointing to first, which holds a reference to the first link. The statement

```
current = current.next;
```

changes current to point to the next link, because that's what's in the next field in each link.

Here's the entire displayList() method:

```
public void displayList()
{
	System.out.print("List (first-->last): ");
	Link current = first; // start at beginning of list
	while(current != null) // until end of list,
	{
		current.displayLink(); // print data
		current = current.next; // move to next link
	}
	System.out.println("");
}
```

The Driver Class

```
class LinkListApp
{
	public static void main(String[] args)
	{
		LinkList theList = new LinkList(); // make new list
		theList.insertFirst(22, 2.99); // insert four items
		theList.insertFirst(44, 4.99);
		theList.insertFirst(66, 6.99);
		theList.insertFirst(88, 8.99);
		theList.displayList(); // display list
		while( !theList.isEmpty() ) // until it's empty,
		{
			Link aLink = theList.deleteFirst(); // delete link
			System.out.print("Deleted "); // display it
			aLink.displayLink();
			System.out.println("");
		}
		theList.displayList(); // display list
	} // end main()
} // end class LinkListApp
-
```

In main () we create a new list, insert four new links into it with insertFirst(), and display it. Then, in the while loop, we remove the items one by one with deleteFirst() until the list is empty. The empty list is then displayed. Here's the output from linkList.java:

List (first-->last): {88, 8.99} {66, 6.99} {44, 4.99} {22,2.99}

Deleted {88, 8.99}

Deleted {66, 6.99}

Deleted {44, 4.99}

Deleted {22, 2.99}

List (first-->last):

Exercises

Question 1

Write a program to create a **Linked List** and different functionality related to the following operations:

1. Traversing Linked List.
2. Searching in Linked List.
3. Insertion in Linked List.
4. Deletion from Linked List.

Question 2

Write a program which creates two ordered list, based on an integer key, and then merges them together.

Question 3

Imagine an effective dynamic structure for storing polynomials. Write operations for addition, subtraction, and multiplication of polynomials.

Question 4

Write Java programs to implement the List ADT using arrays

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.

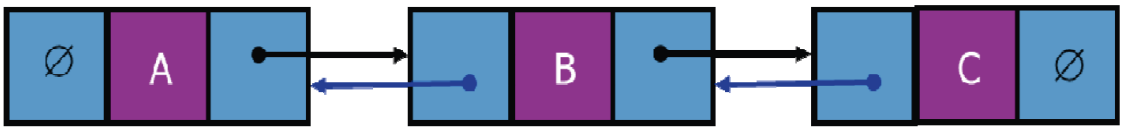
Type Object may be any type that can be stored in the list. The actual type of the object will be provided by the user.

Lab 3: Doubly Linked List Implementation of the List ADT

AIM: To write a program for double linked list implementation.

A doubly linked list overview:

A doubly linked list is a linked list in which every node has a next pointer and a back pointer. Every node contains the address of the next node (except the last node), and every node contains the address of the previous node (except the first node). A doubly linked list can be traversed in either direction.



Algorithm

Doubly-Linked List add

- adding after a given node (node) means updating the previous and *next* node's next and *prev* references:
    - DLinkedNode<E> followingNode = node.next;
    - node.next = new DLinkedNode<E> (value);
    - followingNode.prev = node.next;
- The above code assumes that there is both a previous and next node.
- If not, the code needs special cases.
- A circular list, if coded correctly, does not need as many special cases.

Doubly-Linked List remove

- removing a given node (node) means updating the node's predecessor's *next* field, and the node's successor's *prev* field:

```
node.prev.next = node.next;
node.next.prev = node.prev;
```

- here are the special cases for a doubly linked-list that is not circular:

```
if ((node == head) && (head.next == null))
{
        head = null;
```

```
        }
        else
        {
                if (node.prev != null)
                        node.prev.next = node.next;
                if (node.next != null)
                        node.next.prev = node.prev;
        }
```

Looping over the elements of a collection

- sometimes we want do something with all of the elements of a collection
- for example, we might want to print the values
- or we might want to add all the values in a collection of numbers
- we can do a loop with get:

```
for (int i = 0; i < List.size(); i++) {
  E element = List.get(i);
  ... // do something with element
}
```

- If get takes more than constant time, this is very inefficient: the outer loop is repeated List.size times, so if the inner loop is also linear in the list size, the entire loop takes time $(List.size)^2$.
- with a linked list, it is sufficient to keep a reference to the last node that was visited
- but letting the user program directly have access to this reference would not be very safe
- instead, the reference is encapsulated in an object called an iterator, which only provides a small set of operations
- a Java iterator only provides two or three operations:
    - E next(), which returns the next element, and also advances the references
    - booleans hasNext(), which returns whether there is at least one more element
    - void remove(), which removes the last element returned by next() (this method is optional)
- using remove may invalidate any other existing (concurrent) iterators

Exercises

Question 1

Write a program to create a **doubly linked list** and different functionality related to the following operations:

- Insert after a specific object.
- Delete a specific object.
- Insert element in the front of doubly linked list.
- Insert element at last in a doubly linked list.
- Delete first element in a doubly linked list
- Display elements in a double linked list

Question 2

Read a paragraph containing words from an input file. Then create a doubly-linked list containing the distinct words read, where the words of the same length are placed in the same list, in ascending order.

Print the lists in descending order of their word length, case insensitive.

Example data:

Input:

Everything LaTeX numbers for you has a counter associated with it. The name of the counter is the same as the name of the environment or command that produces the number, except with no \. Below is a list of some of the counters used in LaTeX's standard document styles to control numbering.

Output:

1: a

2: \. as in is no of or to

3: for has it. the you

4: list name same some that used with

5: Below LaTeX

6: except styles

7: command control counter LaTeX's number, numbers

8: counters document produces standard

10: associated Everything numbering

11: environment

Lab 4: Cursor Implementation of List ADT

AIM: To implement the List ADT using cursors.

Cursor Implementation:

Some languages like BASIC and FORTRAN do not support pointers. In such cases we go for cursor implementation of lists. Here, we define a global array of structures. Each array has an index number. The malloc and free functions are implemented by using a Cursor Free Space list, which gives the indexes of the structures that are free to use at the moment. The header index value for the Cursor Free Space is 0. For example, if there are 10 global structures that are defined then the Cursor Free Space would be as follows.

Index	Element	Next Free Space
0	-	1
1	-	2
2	-	3
3	-	4
4	-	5
5	-	6
6	-	7
7	-	8
8	-	9
9	-	10
10	-	NULL

To insert an element, first take the first free structure that the header is pointing to, and change the header pointer to the next free space of the structure that has been taken. For e.g. In the above case to insert an element, first take the structure 1 and change the next free space of the header pointer to the value of 2.

To delete an element simply return the structure index to the Cursor Free Space and make the header pointer point to that index. Make the next free space of the returned structure to point to the next free space that the header was pointing to before the return of the structure to the Cursor Free Space.

Algorithm:

1. Declare a structure with the two fields, Element – to store the value of the elements and Next – that denotes the index where the next element resides. Also initialize the necessary variables.
2. Create an array of such structures (say n in number).
3. Initialize the CursorSpace[ ] array by making the header pointer point to the first array index, (i.e.CursorSpace[0].Next=1) and that one to point to the next array index (i.e.CursorSpace[1].Next=2) and subsequently until the last index is reached. The next free space of the last array index is made to point to 0 (i.e.CursorSpace[n].Next=Null)
4. To insert an element in the list, use a function called CursorAlloc ( ) that is equivalent to malloc function. In CursorAlloc(), get the array index that is pointed to by the header index 0 and store it in a value P. Now make the header's next free space to be the next free space of the index P by the command CursorSpace[0].Next = CursorSpace[P].Next. Now assign the new element to CursorSpace[P].Element.
5. To delete an element from the List ADT, take the position P of that element. Now return P to the CursorSpace array and make CursorSpace[P].Next to point to CursorSpace[0].Next. Also make the header pointer point to the position P by the command CursorSpace[0].Next = P.
6. To find the position of an element in a List ADT, get the element; initialize an index P as the index of the array pointed to by the header. Now keep on incrementing the value of P by the command P= CursorSpace[P].Next, till the element is found in Cursor[P].Element.

Exercises

Question 1

Write a program to create a cursor implementation of linked lists and different functionality related to the following operations:

1. Initialized CURSOR_SPACE.
2. Cursor-alloc and cursor-free.
3. Test whether a linked list is empty--cursor implementation.
4. Test whether p is last in a linked list--cursor implementation.
5. Find routine--cursor implementation.
6. Deletion routine for linked lists--cursor implementation.
7. Insertion routine for linked lists--cursor implementation.

Question 2

Given a linked list and two keys in it, swap nodes for two given keys. Nodes should be swapped by changing links. Swapping data of nodes may be expensive in many situations when data contains many fields. It may be assumed that all keys in linked list are distinct.
Example:

Input: 10 -> 15 -> 12 -> 13 -> 20 -> 14, x = 12, y = 20

Output: 10 -> 15 -> 20 -> 13 -> 12 -> 14

This may look a simple problem, but is interesting question as it has following cases to be handled.

1. X and y may or may not be adjacent.
2. Either x or y may be a head node.
3. Either x or y may be last node.
4. X and/or y may not be present in linked list.

How to write a clean working code that handled all of the above possibilities.

Lab 5: Stack ADT

AIM: To write a program for linked implementation of stack in Java.

Overview:

Many applications that use a linear data structure do not require the full range of operations supported by the List ADT. Although you can develop these applications using the List ADT, the resulting programs are likely to be somewhat cumbersome and inefficient. An alternative approach is to define new linear data structures that support more constrained sets of operations.

By carefully defining these ADTs, you can produce ADTs that meet the needs of a diverse set of applications, but yield data structures that are easier to apply- and are often more efficient-than the List ADT.

The stack is one example of a constrained linear data structure. In a stack, the elements are ordered from most recently added (the top) to the least recently added (the bottom). All insertions and deletions are performed at the top of the stack. You use the push operation to insert an element onto the stack and the pop operation to remove the topmost stack element. A sequence of pushes and pops is shown below.

Push a	Push b	Push c	Pop	Pop
		c		
	b	b	b	
A	a	a	a	a
__	__	__	__	__

These constraints on insertion and deletion produce the “last in, First out” (LIFO) behavior that characterizes a stack. Although the stack data structure is narrowly defined, it is used so extensively by systems software that support for a primitive stack is one of the basic elements of most computer architectures.

Stack ADT

Elements

The elements in a stack are of generic type Object.

Structure

The stack elements are linearly ordered from most recently added (the top) to least recently added (the bottom). Elements are inserted onto (pushed) and removed from (popped) the top of the stack.

Constructors and their Helper Method

Stack ( )

Precondition:

None.

Post condition:

Default Constructor. Calls setup, which creates an empty stack and (if necessary) allocates enough memory for a stack containing DEF_MAX_STACK_SIZE (a constant value) elements.

Stack (int size)

Precondition:

size > 0.

Post condition:

Constructor. Calls setup, which creates an empty stack and (if necessary) allocates enough memory for a stack containing size elements.

void setup(int size)

Precondition:

size > 0. A helper method for the constructors. Is declared private since only stack constructors should call this method.

Post condition:

Creates an empty stack of a specific size (where applicable) based on the value of size received from the constructor.

Methods in the Interface

void push (Object newElement)

Precondition:

Stack is not full and newElement is not null.

Post condition:

Inserts newElement onto the top of a stack.

Object pop ( )

Precondition:

Stack is not empty.

Post condition:

Removes the most recently added (top) element from a stack and returns it.

void clear ( )

Precondition:

None.

Post condition:

Removes all the elements in a stack.

boolean isFull ( )

Precondition:

None.

Post condition:

Returns true if a stack is full. Otherwise, returns false.

Exercises

Question 1

We commonly write arithmetic expressions in infix form, that is, with each operator placed between its operands, as in the following expression:

(3+4)*(5/2)

Although we are comfortable writing expressions in this form, infix form has the disadvantage that parentheses must be used to indicate the order in which operators are to be evaluated.

These parentheses, in turn, greatly complicate the evaluation process.

Evaluation is much easier if we can simply evaluate operators from left to right. Unfortunately, this left-to-right evaluation strategy will not work with the infix form of arithmetic expressions. However, it will work if the expression is in postfix form. In the postfix form of an arithmetic expression, each operator is placed immediately after its operands. The expression above is written below in postfix form as

34+52/*

Note that both forms place the numbers in the same order (reading from left to right). The order of the operators is different, however, because the operators in the postfix form are positioned in the order that they are evaluated. The resulting postfix expression is hard to read at first, but it is easy to evaluate. All you need is a stack on which to place intermediate results.

Suppose you have an arithmetic expression in postfix form that consists of a sequence of single digit, nonnegative integers and the four basic arithmetic operators (addition, subtraction, multiplication, and division). This expression can be evaluated using the following algorithm in conjunction with a stack of floating-point numbers.

Read in the expression character by character. As each character is read in,

- If the character corresponds to a single digit number (characters ‘0’ to ‘9’), then push the corresponding floating-point number onto the stack.
- If the character corresponds to one of the arithmetic operators (characters ‘+’, ‘-‘, ‘*’, and ‘/’), then
    - Pop a number off of the stack. Call it operand1.
    - Pop a number off of the stack. Call it operand2.

- Combine these operands using the arithmetic operator, as follows

  result = operand2 operator operand1

- Push result onto the stack.

- When the end of the expression is reached, pop the remaining number off the stack. This number is the value of the expression.

Applying this algorithm to the arithmetic expression

34+52/*

Yields the following computation:

'3': Push 3.0

'4': Push 4.0

'+': Pop, operand1 = 4.0

Pop, operand2 = 3.0

Combine, result = 3.0 + 4.0 = 7.0

Push 7.0

'5': Push 5.0

'2': Push 2.0

'/': Pop, operand1 = 2.0

Pop, operand2 = 5.0

Combine, result = 5.0 / 2.0 = 2.5

Push 2.5

'*': Pop, operand1 = 2.5

Pop, operand2 = 7.0

Combine, result = 7.0 * 2.5 = 17.5

Push 17.5

'\n': Pop, Value of expression = 17.5

Step 1: Create a program (call it PostFix.java) that reads the postfix form of an arithmetic expression, evaluates it, and outputs the result. Assume that the expression consists of single digit, nonnegative integers (‘0’ to ‘9’) and the four basic arithmetic operators (‘+’, ‘-‘, ‘*’, and ‘/’).

Further assume that the arithmetic expression is input from the keyboard with all the characters on one line. In PostFix.java, values of type float will be large enough for our purposes.

Hints:

1. Review the code for TestAStack.java to recall how to read in characters, deal with whitespace and push an Object onto the stack.
2. To convert from Object to Float and then to the primitive float requires a cast and the Float.floatValue( ) method. For example,

```
float operand1;
operand1 = ((Float)resultStack.pop( )).floatValue( );
```

3. To set precision to two decimal places, import java.text.DecimalFormat and use code similar to the following:

```
float outResult;
DecimalFormat fmt = new DecimalFormat("0.##");
System.out.println(fmt.format(outResult));
```

Step 2: Complete the following test plan by filling in the expected result for each arithmetic expression. You may wish to include additional arithmetic expressions in this test plan.

Step 3: Execute the test plan. If you discover mistakes in your program, correct them and execute the test plan again.

Test Plan for the Postfix Arithmetic Expression Evaluation Program

Test case	Arithmetic expression	Expected result	Checked
One operator	34+		
Nested operators	34+52/*		
Uneven nesting	93*2+1-		
All operators at end	4675-+*		
Zero dividend	02/		
Single-digit number	7		

Question 2

Checking 'Balanced Parentheses' using Array implementation of stack

Aim: To check whether an expression has balanced parentheses using array implementation of a stack.

Theory:

Compilers check your programs for syntax errors, but frequently a lack of one symbol will cause the compiler to spill out a hundred lines of diagnostics without identifying the real error. Thus, every right brace, bracket and parentheses must correspond to its left counterpart.

This can be verified using a stack.

Algorithm:

1. Make an empty stack implemented as an array.
2. Scan the expression from left to right, character by character.
3. During your scanning:
    a. If you find a left parentheses push it into the stack.
    b. If you find a right parentheses examine the status of the stack:
        I. If the stack is empty then the right parentheses does not have a matching left parentheses. So stop scanning and print expression is invalid.
        II. If the stack is not empty, pop the stack and continue scanning.
4. When the end of the expression is reached, the stack must be empty. Otherwise one or more left parentheses has been opened and not closed.

Lab 6: Queue ADT

AIM: To write a program for linked implementation of queue in Java.

Overview:

This laboratory focuses on another constrained linear data structure, the queue. The elements in a queue are ordered from least recently added (the front) to most recently added (the rear). Insertions are performed at the rear of the queue, and deletions are performed at the front. You use the enqueue operation to insert elements and the dequeue operation to remove elements. A sequence of enqueues and dequeues is shown below.

Enqueue a	Enqueue b	Enqueue c	Dequeue	Dequeue
a	a b	a b c	b c	c

The movement of elements through a queue reflects the “first in, first out” (FIFO) behavior that is characteristic of the flow of customers in a line or the transmission of information across a data channel. Queues are routinely used to regulate the flow of physical objects, information, and requests for resources (or services) through a system. Operating systems, for example, use queues to control access to system resources such as printers, files, and communications lines. Queues also are widely used in simulations to model the flow of objects or information through a system.

Queue ADT

Elements

The elements in a queue are of generic type Object.

Structure

The queue elements are linearly ordered from least recently added (the front) to most recently added (the rear). Elements are inserted at the rear of the queue (enqueued) and are removed from the front of the queue (dequeued).

Constructors and their Helper Method

Queue ( )

Precondition:

None.

Post condition:

Default Constructor. Calls setup, which creates an empty queue and (if necessary) allocates enough memory for a queue containing DEF_MAX_QUEUE_SIZE (a constant value) elements.

Queue (int size)

Precondition:

size > 0.

Post condition:

Constructor. Calls setup, which creates an empty queue and (if necessary) allocates enough memory for a queue containing size elements.

void setup (int size)

Precondition:

size > 0. A helper method for the constructors. Is declared private since only queue constructors should call this method.

Post condition:

Creates an empty queue of a specific size (where applicable) based on the value of size received from the constructor.

Methods in the Interface

void enqueue (Object newElement)

Precondition:

Queue is not full and newElement is not null.

Post condition:

Inserts newElement at the rear of a queue.

Object dequeue ( )

Precondition:

Queue is not empty.

Post condition:

Removes the least recently added (front) element from a queue and returns it.

void clear ( )

Precondition:

None.

Post condition:

Removes all the elements in a queue.

boolean isEmpty ( )

Precondition:

None.

Post condition:

Returns true if a queue is empty. Otherwise, returns false

boolean isFull ( )

Precondition:

None.

Post condition:

Returns true if a queue is full. Otherwise, returns false.

Exercises

Question 1

In this exercise, you use a queue to simulate the flow of customers through a checkout line in a store. In order to create this simulation, you must model both the passage of time and the flow of customers through the line. You can model time using a loop in which each pass corresponds to a set time interval-one minute, for example. You can model the flow of customers using a queue in which each element corresponds to a customer in the line.

In order to complete the simulation, you need to know the rate at which customers join the line, as well as the rate at which they are served and leave the line. Suppose the checkout line has the following properties.

- One customer is served and leaves the line every minute (assuming there is at least one customer waiting to be served during that minute).
- Between zero and two customers join the line every minute, where there is a 50% chance that no customers arrive, a 25% chance that one customer arrives, and a 25% chance that two customers arrive.

You can simulate the flow of customers through the line during a time period n minutes long

Using the following algorithm.

Initialize the queue to empty.
for (minute = 0 ; minute < n ; minute++)
{
    If the queue is not empty, then remove the customer at the front of the queue.
    Compute a random number k between 0 and 3.
    If k is 1, then add one customer to the line. If k is 2, then add two customers to the line. Otherwise (if k is 0 or 3), do not add any customers to the line.

}

Step 1: Create a program that uses the Queue ADT to implement the model described above. Your program should update the following information during each simulated minute, that is, during each pass through the loop:

- The total number of customers served
- The combined length of time these customers spent waiting in line
- The maximum length of time any of these customers spent waiting in line

In order to compute how long a customer waited to be served, you need to store the "minute" that the customer was added to the queue as part of the queue element corresponding to that customer.

Step 2: Use your program to simulate the flow of customers through the line and complete the following table. Note that the average wait is the combined waiting time divided by the total number of customers served.

Take special note that, in this program shell, the length of the simulation, simLength, is read in as an argument to the program itself. In other words, it is a value assigned to the args parameter of the main method-now you know the use of the args array in the main method of your program. This argument is entered as an additional string when you run your program from the command line prompt. If your Java development system does not allow you to invoke your program from the command line, you will need to determine how this argument is passed in the particular Java development system you are using.

Since each args in main is a String array, args[0] must be converted to an int, which is the data type assigned to the simLength. As illustrated in the statement, this conversion is commonly done in Java by using the static method parseInt in the class Integer.

```
simLength = Integer.parseInt( args[0] );
```

Java programmers use this parseInt method on a daily basis as a quick and easy way to convert a String to an int in Java.

Time in minutes	Total number of customers served	Average wait	Longest wait
30			
60			
120			
480			

Question 2

Write a Java program to perform the implementation of queue using array

Algorithm

Step 1: start the program

Step 2: initialize the array variable a () for storing elements and declare the required variable

Step 3: Define a function to insert, delete display the data items for data

Step 4: For insert () function

Get the new elements

Create the rear is greater than array size display queue is “overflow”.

Else

Insert the new element & increment the top pointer

Step 5: For delete () function

If the queue is empty than display “queue is over flow”

Else

Decrement the top pointer

Step 6: For display () function

Apply the loop for to display queue elements.

Lab 7: Binary Search Tree

AIM: To write a program that implement the basic operations of a Binary search tree.

Overview:

The binary search algorithm allows you to efficiently locate an element in an array provided that each array element has a unique identifier-called its key-and provided that the array elements are stored in order based on their keys. Given the following array of keys:

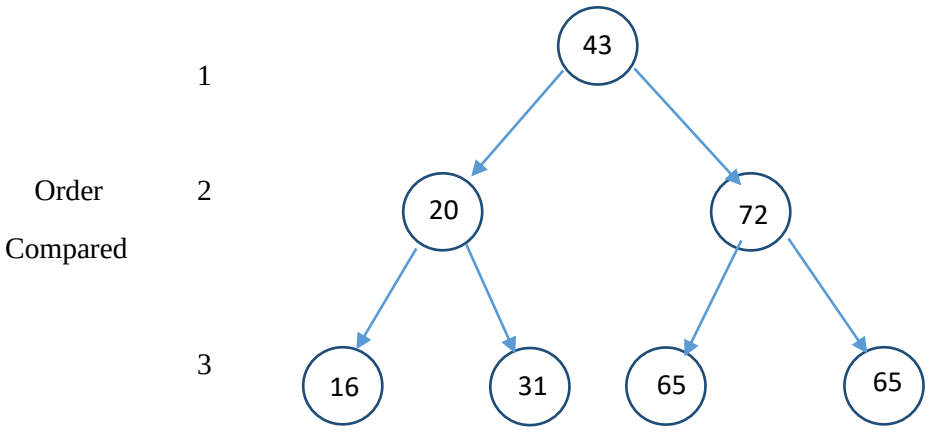
Index	0	1	2	3	4	5	6
Key	16	20	31	43	65	72	86

A binary search for the element with key 31 begins by comparing 31 with the key in the middle of the array, 43. Because 31 is less than 43, the element with key 31 must lie in the lower half of the array (entries 0-2). The key in the middle of this subarray is 20. Because 31 is greater than 20, the element with key 31 must lie in the upper half of this subarray (entry 2). This array entry contains the key 31. Thus, the search terminates with success.

Although the comparisons made during a search for a given key depend on the key, the relative order in which comparisons are made is invariant for a given array of elements. For instance, when searching through the previous array, you always compare the key that you are searching for with 43 before you compare it with either 20 or 72. Similarly, you always compare the key with 72 before you compare it with either 65 or 86. The order of comparisons associated with this array is shown below.

Index	0	1	2	3	4	5	6
Key	16	20	31	43	65	72	86
Order Compared	3	2	3	1	3	2	3

The hierarchical nature of the comparisons that are performed by the binary search algorithm is reflected in the following tree.



Observe that for each key K in this tree, all of the keys in K's left subtree are less than K and all of the keys in K's right subtree are greater than K (or equal to it-if all the keys are not unique). Trees with this property are referred to as binary search trees.

When searching for a key in a binary search tree, you begin at the root node and move downward along a branch until either you find the node containing the key or you reach a leaf node without finding the key. Each move along a branch corresponds to an array subdivision in the binary search algorithm. At each node, you move down to the left if the key you are searching for is less than the key stored in the node, or you move down to the right if the key you are searching for is greater than the key stored in the node.

Binary Search Tree ADT

Elements

The elements in a binary search tree are of generic type *TreeElem*. Each element has a key that uniquely identifies the element. Elements usually include additional data. Objects of type *TreeElem* must provide a method called key ( ) that returns an element's key.

Structure

The elements form a binary tree. For each element E in the tree, all the elements in E's left subtree have keys that are less than E's key and all the elements in E's right subtree have keys that are greater than E's key.

Constructor and Methods

BSTree ( )

Precondition:

None.

Post condition:

Constructor. Creates an empty binary search tree.

void insert (TreeElem newElement)

Precondition:

Binary search tree is not full.

Post condition:

Inserts *newElement* into a binary search tree. If an element with the same key as *newElement* already exists in the tree, then updates that element's nonkey fields with *newElement's* nonkey fields.

TreeElem retrieve (int searchKey)

Precondition:

None.

Post condition:

Searches a binary search tree for the element with key *searchKey*. If this element is found, then returns the element. Otherwise, returns a null element.

void remove (int deleteKey)

Precondition:

None.

Post condition:

Deletes the element with key *deleteKey* from a binary search tree.

void writeKeys ( )

Precondition:

None.

Post condition:

Outputs the keys of the elements in a binary search tree. The keys are output in ascending order, one per line.

void clear ( )

Precondition:

None.

Post condition:

Removes all the elements in a binary search tree.

boolean isEmpty ( )

Precondition:

None.

Post condition:

Returns true if a binary search tree is empty. Otherwise, returns false.

boolean isFull ( )

Precondition:

None.

Post condition:

Returns true if a binary search tree is full. Otherwise, returns false.

EXERCISES

Question 1

Write a JAVA program to perform the following operations:

a) To construct a binary search tree of integers.
b) To traverse the tree using all the methods i.e., **inorder**, **preorder** and **postorder**.
c) To display the elements in the tree.

ALGORITHM FOR TREES

MAIN FUNCTION ()

Step 1: initialize a list called tree using tree data structures, consider root as the start node.

Step 2: Read the operation from the keyboard, if the operation says **preorder** display then go to the **preorder** display function else if the operation specifies post order then go to the post order display function else if the operation specifies in order then go to the in order display function else if the operation specified is exit then go to step step3.

Step 3: Return to the main program.

INORDER FUNCTION ()

Step 1: Traverse the left subtree by calling **Inorder** function recursively.

Step 2: Visit root and print root information.

Step 3: Traverse the right subtree by calling **Inorder** function recursively.

Step 4: Return to the main program.

PREORDER DISPLAY FUNCTON ()

Step 1: Visit root and print root information.

Step 2: Traverse the left subtree by calling **Preorder** function recursively.

Step 3: Traverse the right subtree by calling **Preorder** function recursively.

Step 4: Return to the main program

POST ORDER DISPLAY FUNCTION ()

Step 1: Traverse the left subtree by calling **Postorder** function recursively.

Step 2: Traverse the right subtree by calling **Postorder** function recursively.

Step 3: Visit root and print root information.

Step 4: Return to the main program.

INSERTION FUNCTION ()

Step 1: Create a node called *newnode* with a data field and a left and the right link.

Step 2: Insert the value entered from the user to the data field of the *newnode* and assign the right and left links to the null value.

Step3: Check if the tree is empty, if yes then *newnode* is the root node and go to step9 else go to step4.

Step 4: Consider a temporary node p and assign it with root.

Step 5: Repeat S6 and S7 till p becomes Null.

Step 6: Consider a temporary node q and assign it with p.

Step 7: Check if *newnode* data is greater than p node data, if yes proceed towards right subtree else proceed towards left subtree.

Step 8: If new node data is less than q node data insert *newnode* as left child of q node else insert *newnode* as right child of q node.

Step 9: Return to the main program.

Question 2

You have to maintain information for a shop owner. For each of the products sold in his/hers shop the following information is kept: a unique code, a name, a price, amount in stock, date received, and expiration date. For keeping track of its stock, the clerk would use a computer program based on a search tree data structure. Write a program to help this person, by implementing following the following operations:

- Insert an item with all its associated data.
- Find an item by its code, and support updating of the item found.
- List valid items in alphabetical order of their names.
- List expired items in alphabetical order of their names.
- List all items.
- Delete an item given by its code.
- Delete all expired items.
- Create a separate search tree for expired items.
- Save stock in file stock.data.

- Exit

If file *stock.data* exists, your program must automatically load its contents.

I/O description. This program should be interactive.

Question 3

You have to maintain information for school classes. For each of the students in a class the following information is kept: a unique code, student name, birth date, home address, id of class he is currently in, date of enrollment, status (undergrad, graduate). For keeping track of the students, the school secretary would use a computer program based on a search tree data structure. Write a program to help the secretary, by implementing following the following operations:

- Insert an item with all its associated data.
- Find a student by his/hers unique code, and support updating of the student info if found.
- List students by class in lexicographic order of their names.
- List all students in lexicographic order of their names.
- List all graduated students.
- List all undergrads by their class in lexicographic order of their names.
- Delete a student given by its code.
- Delete all graduates.
- Save all students in file *student.data*.
- Exit.

If file *student.data* exists, your program must automatically load its contents. .

I/O description. This program should be interactive.

Question 4

A database is a collection of related pieces of information that is organized for easy retrieval. The set of account records shown below, for instance, forms an accounts database.

Record No.	Account ID	First Name	Last Name	Balance
0	6274	James	Johnson	415.56
1	2843	Marcus	Wilson	9217.23
2	4892	Maureen	Albrigh	51462.56
3	8337	Debra	Douglas	27.26
4	9523	Bruce	Gold	719.32
5	3165	John	Carlson	1496.24

Each record in the accounts database is assigned a record number based on that record's relative position within the database file. You can use a record number to retrieve an account record directly, much as you can use an array index to reference an array element directly. The following program from the file AccountRec.java, for example, retrieves a record from the accounts database in the file Accounts.dat. Notice that both keyboard and file input are used in this program.

```
import java.io.*;
import java.util.StringTokenizer;
class AccountRec
{
        // Constants
        private static final long bytesPerRecord = 38; // Number of bytes used to store
                                                       // each record in the accounts
                                                       // database file
                                                       // Data members
        private int acctID;                            // Account identifier
```

```
private String firstName;                    // Name of account holder
private String lastName;
private double balance;                      // Account balance
public static void main (String args[ ]) throws IOException
{
        AccountRec acctRec = new AccountRec( ); // Account record
        long recNum; // User input record number
        String str, // For reading a String
        name;
        // Need random access on the accounts database file; r = read only
        RandomAccessFile inFile =
                new RandomAccessFile("Accounts.dat", "r");
        // Also need tokenized input stream from keyboard
        BufferedReader reader =
                new BufferedReader(new InputStreamReader(System.in));
        StreamTokenizer keybdTokens = new StreamTokenizer(reader);
        // Get the record number to retrieve.
        System.out.println( );
        System.out.print("Enter record number: ");
        keybdTokens.nextToken( );
        recNum = (long)keybdTokens.nval;
        // Move to the corresponding record in the database file using the
        // seek( ) method in RandomAccessFile.
        inFile.seek(recNum * bytesPerRecord);

        str = inFile.readLine( ); // Read the record
        if (str != null) // Is there something in the string?
        {
                // Need to tokenize the String read by readline( )
                StringTokenizer strTokens = new StringTokenizer(str);
                name = strTokens.nextToken( ); // first String token
                acctRec.acctID = Integer.parseInt(name); // Convert String to an int
```

```
                    acctRec.firstName = strTokens.nextToken( ); // 2nd String token -
                                                                // firstName
                    acctRec.lastName = strTokens.nextToken( ); // 3rd String token -
                                                                // lastName
                    name = strTokens.nextToken( ); // 4th String token
                    // Convert the String to a double
                    acctRec.balance = Double.parseDouble(name);
                    // Display the record.
                    System.out.println(recNum + " : " + acctRec.acctID + " "
                            + acctRec.firstName + " " + acctRec.lastName + " "
                            + acctRec.balance);
            }
            else
                    System.out.println("Reached EOF");
            // Close the file streams
            inFile.close( );
    } // main
} // class AccountRec
```

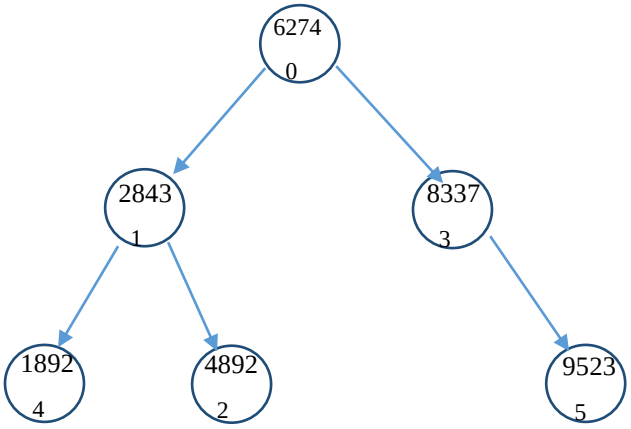
Record numbers are assigned by the database file mechanism and are not part of the account information. As a result, they are not meaningful to database users. These users require a different record retrieval mechanism, one that is based on an account ID (the key for the database) rather than a record number.

Retrievals based on account ID require an index that associates each account ID with the corresponding record number. You can implement this index using a binary search tree in which each element contains the two fields: an account ID (the key) and a record number. Since the binary search tree stores elements of type *TreeElem*, the class IndexEntry given below can be used to implement this index.

```
class IndexEntry implements TreeElem
{
    int acctID; // (Key) Account identifier
```

```
        long recNum; // Record number
        public int key ( )
        {
                return acctID;
        } // Return key field
}
```

You build the index by reading through the database account by account, inserting successive (account ID, record number) pairs into the tree as you progress through the file. The following index tree, for instance, was produced by inserting the account IndexEntry elements from the database records shown above into an (initially) empty tree.



Given an account ID, retrieval of the corresponding account record is a two-step process. First, you retrieve the element from the index tree that has the specified account ID. Then, using the record number stored in the index element, you read the corresponding account record from the database file. The result is an efficient retrieval process that is based on account ID.

Step 1: Create a program that builds an index tree for the accounts database in the file Accounts.dat. Once the index is built, your program should

- Output the account IDs in ascending order.
- Read an account ID from the keyboard and output the corresponding account record.

Step 2: Test your program using the accounts database in the text file Accounts.dat. A copy of this database in given below. Try to retrieve several account IDs, including account IDs that do not occur in the database. A test plan form follows.

Record No.	Account ID	First Name	Last Name	Balance
0	6274	James	Johnson	415.56
1	2843	Marcus	Wilson	9217.23
2	4892	Maureen	Albrigh	51462.56
3	8337	Debra	Douglas	27.26
4	9523	Bruce	Gold	719.32
5	3165	John	Carlson	1496.24
6	1892	Mary	Smith	918.26
7	3924	Simon	Chang	386.85
8	6023	John	Edgar	9.65
9	5290	George	Truman	16110.68
10	8529	Elena	Gomez	86.77
11	1144	Donald	Williams	4114.26

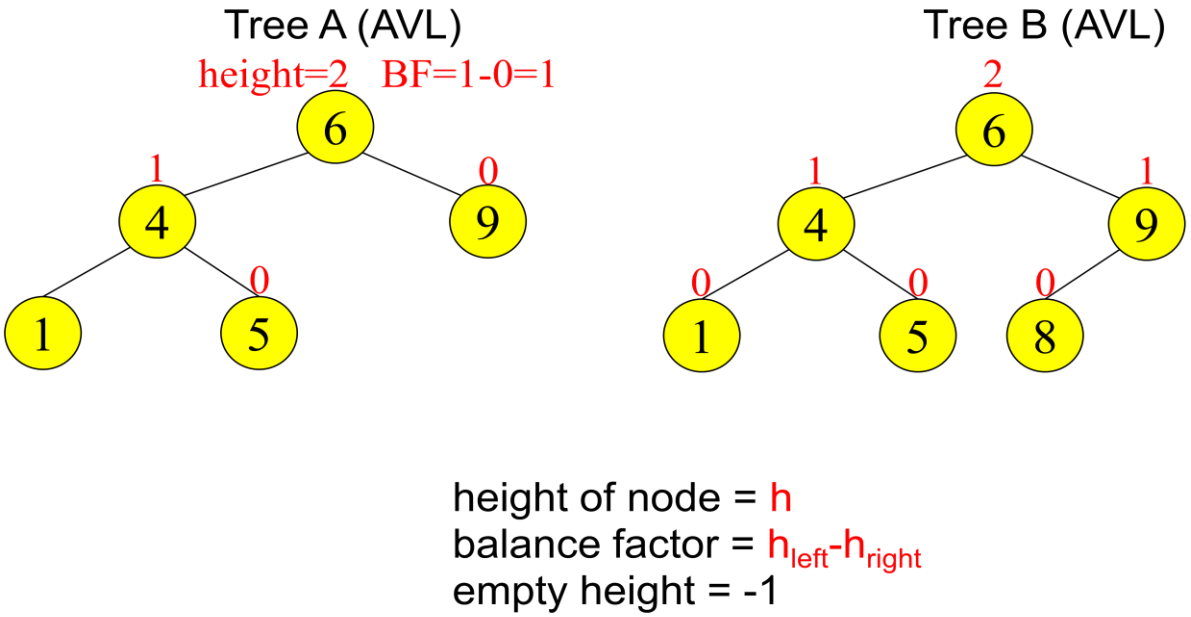
Lab 8: AVL Tree

AIM: To write a program that implement the basic operations of an AVL tree.

Overview:

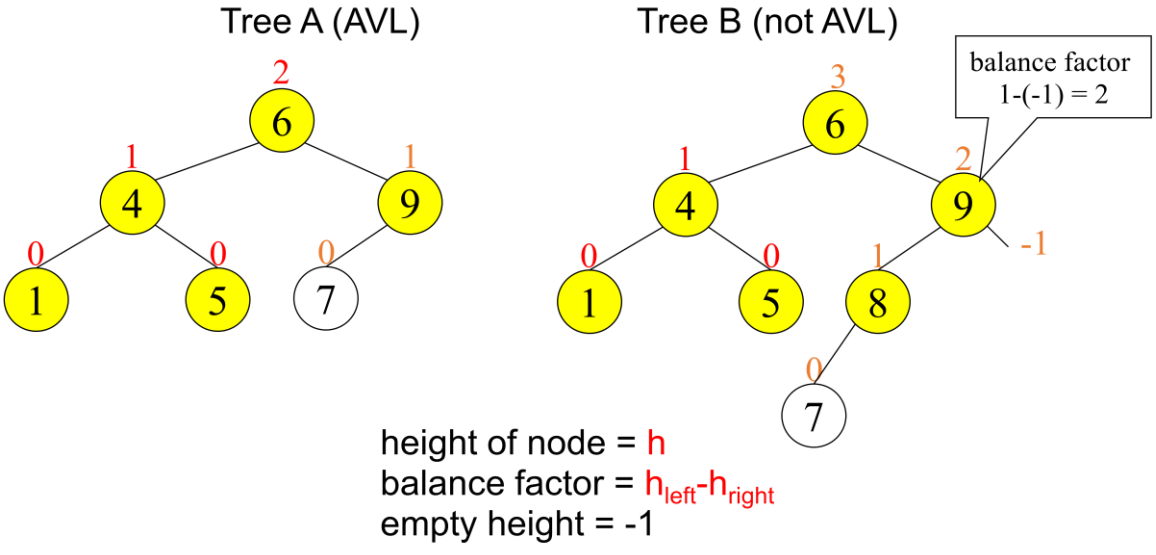
An AVL tree is a binary search tree in which the heights of the left and right subtrees of the root differ at most 1 and in which the left and right subtrees are again AVL trees. Each node of an AVL tree is associated with a balance factor that is the left subtree has height greater than, equal to, or less than that of the right subtree.

Node Heights



height of node = h
balance factor = h_{left}-h_{right}
empty height = -1

Node Heights after Insert 7

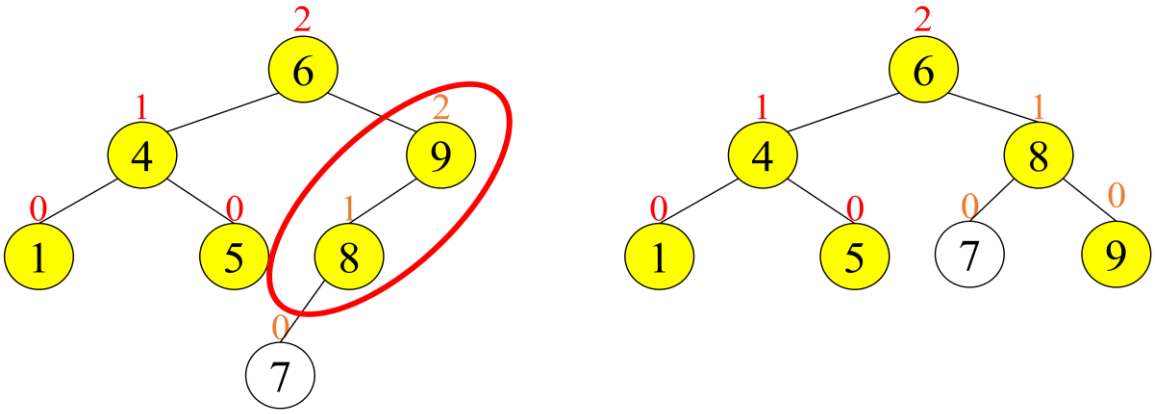


height of node = h
balance factor = h_{left}-h_{right}
empty height = -1

Insert and Rotation in AVL Trees

- Insert operation may cause balance factor to become 2 or –2 for some node
  - › only nodes on the path from insertion point to root node have possibly changed in height
  - › So after the Insert, go back up to the root node by node, updating heights
  - › If a new balance factor (the difference hleft-hright) is 2 or –2, adjust tree by rotation around the node

Single Rotation in an AVL Tree



Insertions in AVL Trees

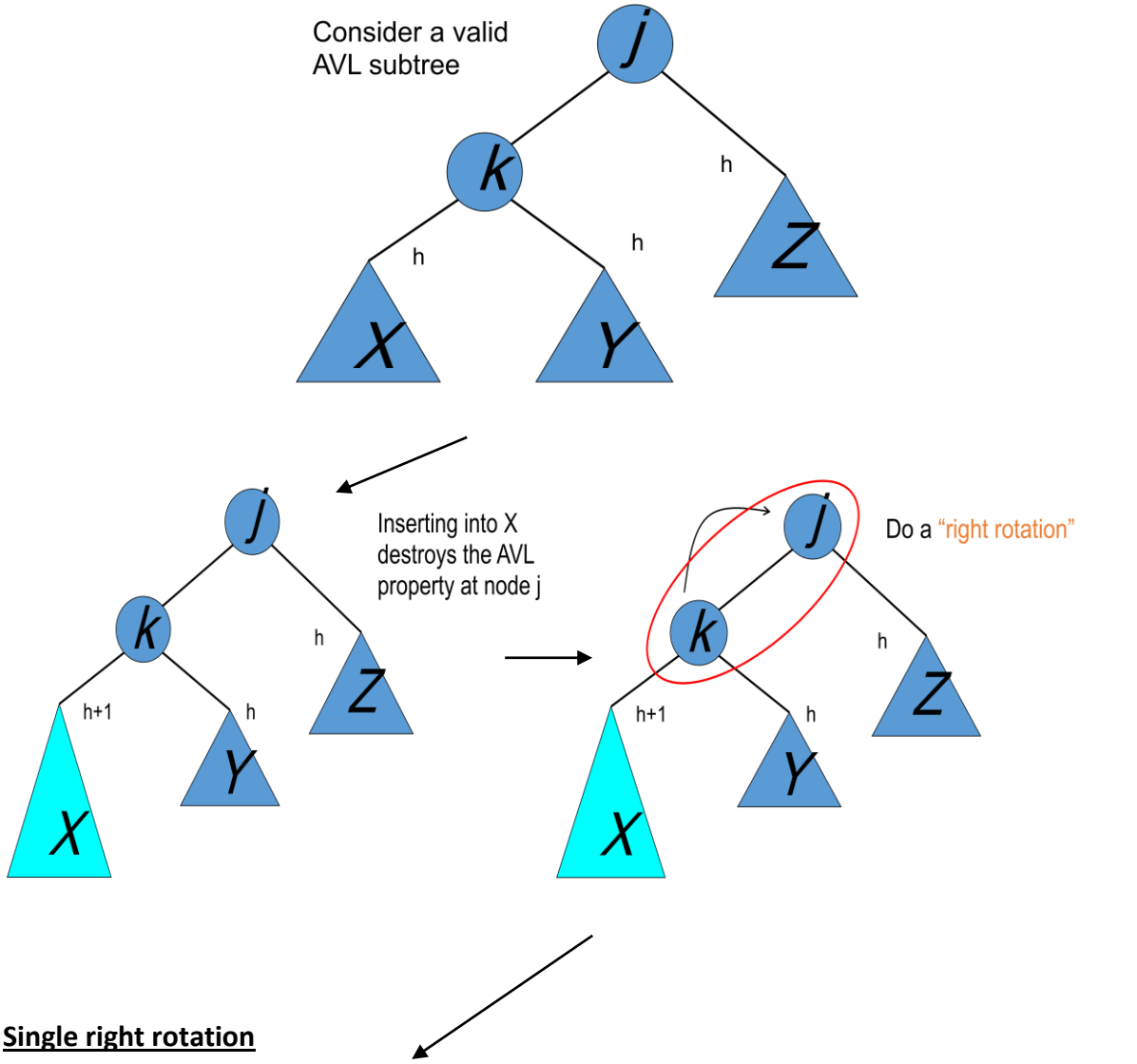
Let the node that needs rebalancing be α.

There are 4 cases:

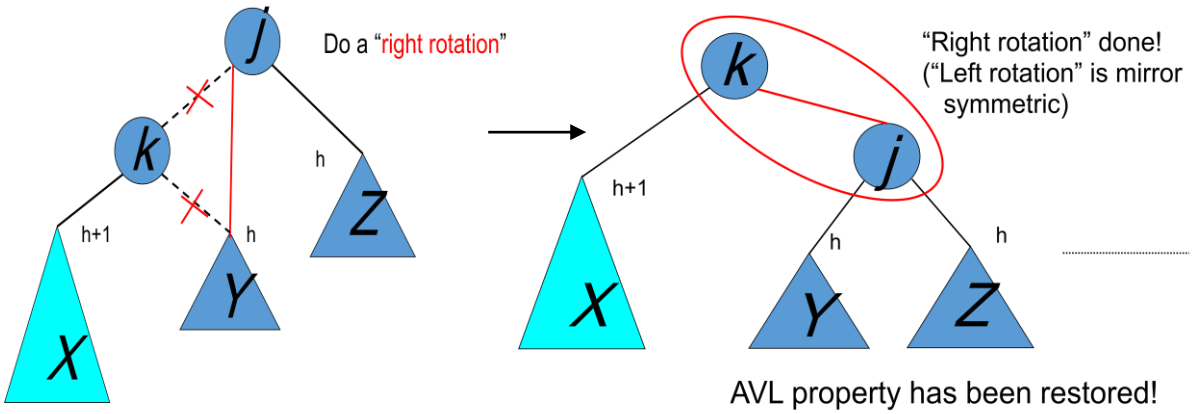
- Outside Cases (require single rotation) :
  a. Insertion into left subtree of left child of α.
  b. Insertion into right subtree of right child of α.
- Inside Cases (require double rotation) :
  a. Insertion into right subtree of left child of α.
  b. Insertion into left subtree of right child of α.

The rebalancing is performed through four separate rotation algorithms.

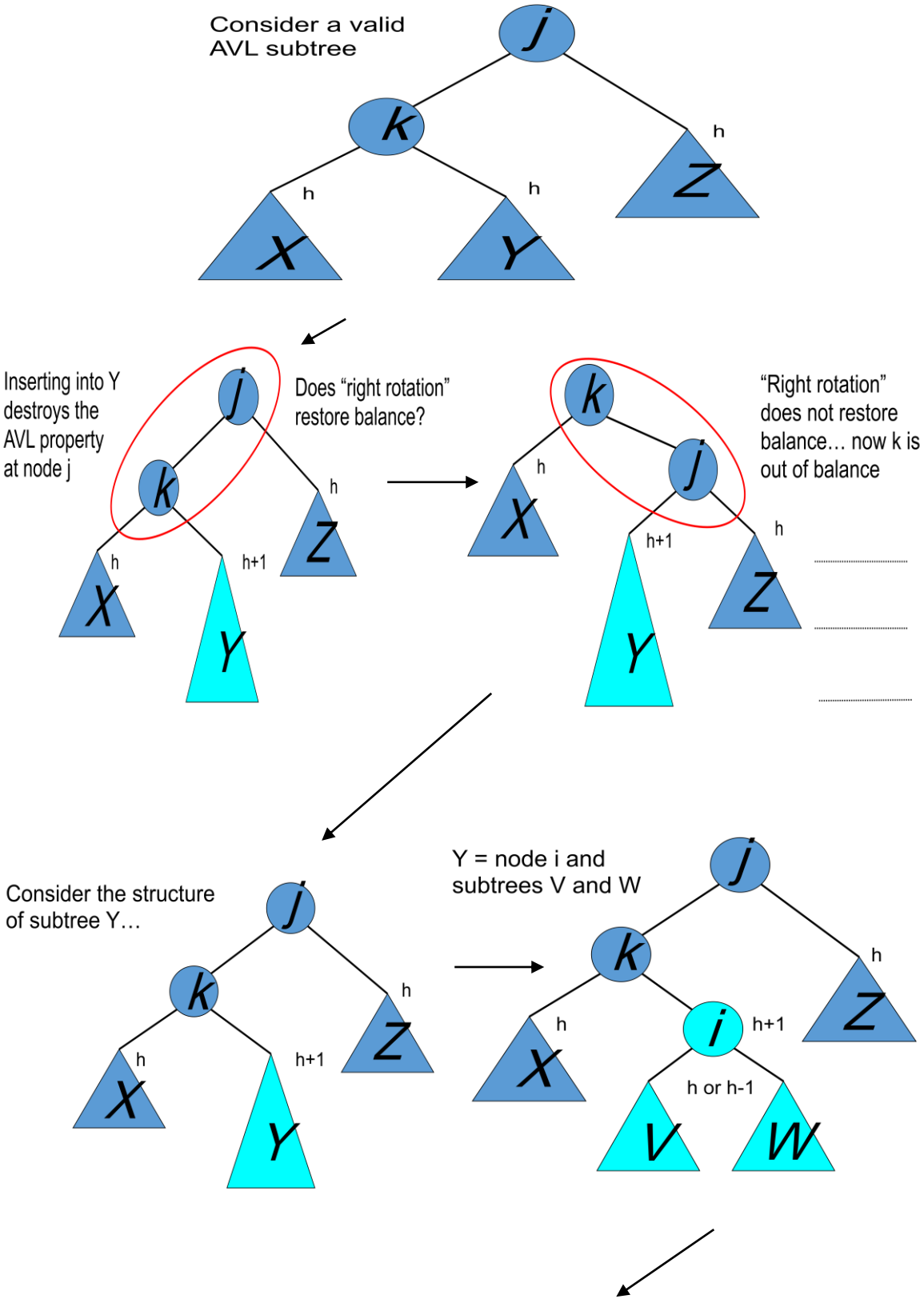
AVL Insertion: Outside Case

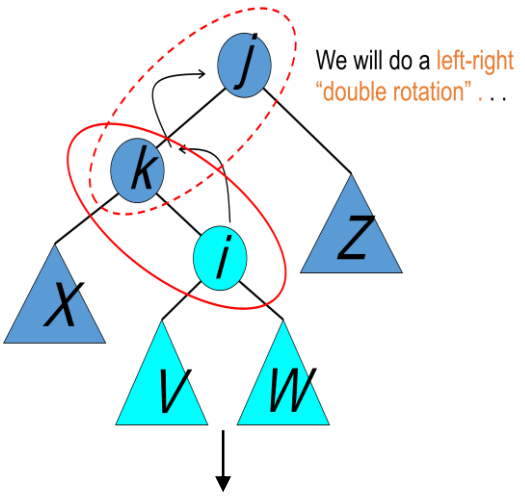


Single right rotation

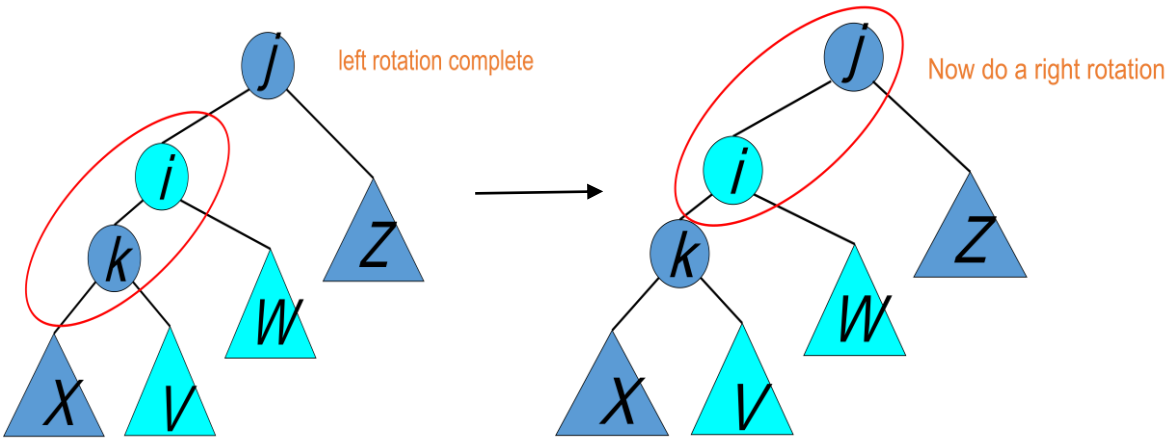


AVL Insertion: Inside Case

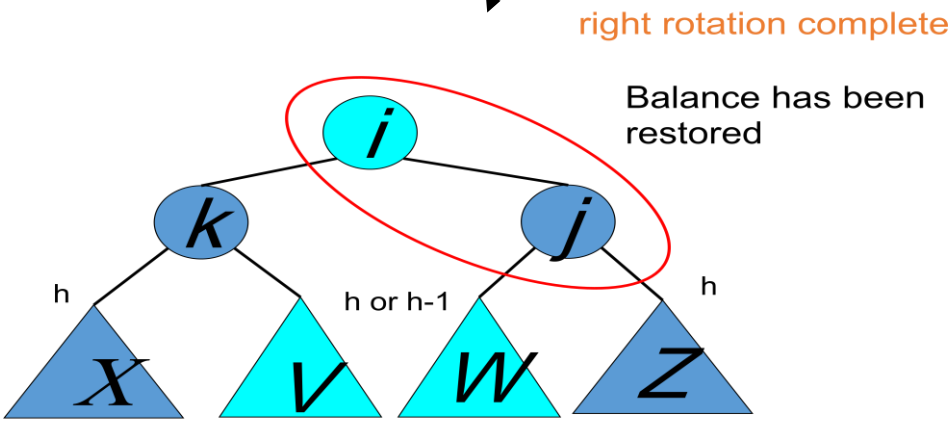




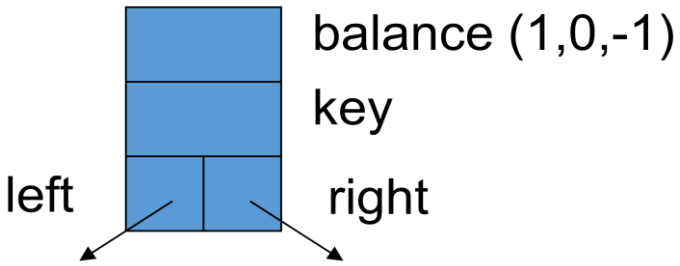
Double rotation : first rotation **second rotation**



Double rotation : second rotation



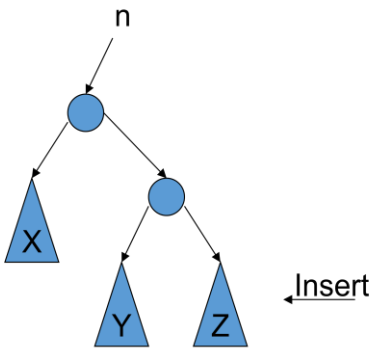
Implementation



No need to keep the height; just the difference in height, i.e. the balance factor; this has to be modified on the path of insertion even if you don't perform rotations. Once you have performed a rotation (single or double) you won't need to go back up the tree

Single Rotation

```
RotateFromRight(n : reference node pointer)
{
        p : node pointer;
        p := n.right;
        n.right := p.left;
        p.left := n;
        n := p
}
```

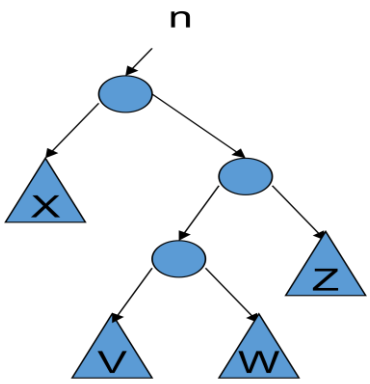


Notes: You also need to modify the heights or balance factors of n and p

Double Rotation

Implement Double Rotation in two lines.

```
DoubleRotateFromRight(n : reference node pointer)
{
        RotateFromLeft(n.right);
        RotateFromRight(n);
}
```



Insertion in AVL Trees

- Insert at the leaf (as for all BST)
  - › only nodes on the path from insertion point to root node have possibly changed in height.
  - › So after the Insert, go back up to the root node by node, updating heights.
  - › If a new balance factor (the difference hleft-hright) is 2 or –2, adjust tree by rotation around the node.

```
Insert(T : reference tree pointer, x : element)
{
        if T = null then
        {
                T := new tree; T.data := x; height := 0; return;
        }
        case
                T.data = x : return ; //Duplicate do nothing
                T.data > x : Insert(T.left, x);
                                if ((height(T.left)- height(T.right)) = 2)
                                {
                                    if (T.left.data > x ) then //outside case
                                            T = RotatefromLeft (T);
                                     else                   //inside case
                                            T = DoubleRotatefromLeft (T);}
                T.data < x :  Insert(T.right, x);
                                code similar to the left case
        Endcase
        T.height := max(height(T.left),height(T.right)) +1;
        return;
}
```

Pros and Cons of AVL Trees

Arguments for AVL trees:

1. Search is O(log N) since AVL trees are always balanced.
2. Insertion and deletions are also O(logn)
3. The height balancing adds no more than a constant factor to the speed of insertion.

Arguments against using AVL trees:

1. Difficult to program & debug; more space for balance factor.
2. Asymptotically faster but rebalancing costs time.
3. Most large searches are done in database systems on disk and use other structures (e.g. B-trees).
4. May be OK to have O(N) for a single operation if total run time for many consecutive operations is fast (e.g. Splay trees).

Exercises:

Question 1

Implement the AVL tree ADT. You must implement the tree class, including *insert()* and *print()* methods, as well as the node class. The tree must maintain the AVL property. The *insert()* method inserts an item into the AVL tree (using the normal binary search tree insert procedure), and then re-balances the trees, using rotations (if necessary). The *print()* method outputs the AVL tree to the console, using an in-order traversal. The output should include the balance for each node (right subtree height – left subtree height). Sample output from *print()* is shown below:

80 (-1)

70 (0)

60 (1)

50 (0)

40 (1)

30 (0)

20 (0)

10 (0)

Test the AVL tree using the driver class provided.

Question 2

Create a method to check if a binary tree is an AVL tree without using the height method.

Lab 9: Hash Tables

AIM: To write a program that implement the basic operations of a Hash table.

Overview:

A hash table is a data structure that offers very fast insertion and searching. When you first hear about them, hash tables sound almost too good to be true. No matter how many data items there are, insertion and searching (and sometimes deletion) can take close to constant time: O(1) in Big O notation. In practice this is just a few machine instructions.

For a human user of a hash table this is essentially instantaneous. It's so fast that computer programs typically use hash tables when they need to look up tens of thousands of items in less than a second (as in spelling checkers). Hash tables are significantly faster than trees, which, as we learned in the preceding chapters, operate in relatively fast O(logN) time. Not only are they fast, hash tables are relatively easy to program.

Hash tables do have several disadvantages. They're based on arrays, and arrays are difficult to expand once they've been created. For some kinds of hash tables, performance may degrade catastrophically when the table becomes too full, so the programmer needs to have a fairly accurate idea of how many data items will need to be stored (or be prepared to periodically transfer data to a larger hash table, a time consuming process).

Also, there's no convenient way to visit the items in a hash table in any kind of order (such as from smallest to largest). If you need this capability, you'll need to look elsewhere.

However, if you don't need to visit items in order, and you can predict in advance the size of your database, hash tables are unparalleled in speed and convenience.

Introduction to Hashing

In this lab we'll introduce hash tables and hashing. One important concept is how a range of key values is transformed into a range of array index values. In a hash table this is accomplished with a hash function. However, for certain kinds of keys, no has function is necessary; the key values can be used directly as array indices. We'll look at this simpler situation first and then go on to show how hash functions can be used when keys aren't distributed in such an orderly fashion.

Hash Functions

A hash function is a function which transforms a key into a natural number called a hash value, i.e. f: K $\rightarrow$ H, where K is the set of keys and H is a set of natural numbers. Function f is a many-to-one function. If two different keys, say k_1 and k_2, have the same hash value, i.e. $f(k_1) = f(k_2)$ then the two keys are said to collide and the corresponding records are called synonyms. Two restrictions are imposed on f:

- For any k $\in$ K the value should be obtained as fast as possible.
- It must minimize the number of collisions.

An example of hash function is: f(k) = γ (k) modulus B, where γ is a function which maps a key to a natural number, and B is a natural number, possibly prime. The expression of function γ depends on the keys. If the keys are numeric, the γ (k) = k. The simplest function γ on alphanumeric keys is the sum of the (ASCII) codes for each character of the key.

```
int f (char *key)
{
        Int i, sum;
        sum = 0;
        for ( i = 0; i < strlen (key ) ; i++ )
                sum += key[i];
        return (sum % B);
}
```

A Dictionary

Let's say we want to store a 50,000 word English language dictionary in main memory.

You would like every word to occupy its own cell in a 50,000-cell array, so you can access the word using an index number. This will make access very fast. But what's the relationship of these index numbers to the words? Given the word morphosis, for example, how do we find its index number?

Converting Words to Numbers

What we need is a system for turning a word into an appropriate index number. To begin, we know that computers use various schemes for representing individual characters as numbers. One such scheme is the ASCII code, in which a is 97, b is 98, and so on, up to 122 for z.

However, the ASCII code runs from 0 to 255, to accommodate capitals, punctuation, and so on. There are really only 26 letters in English words, so let's devise our own code—a simpler one that can potentially save memory space. Let's say a is 1, b is 2, c is 3, and so on up to 26 for z. We'll also say a blank is 0, so we have 27 characters. (Uppercase letters aren't used in this dictionary.)

How do we combine the digits from individual letters into a number that represents an entire word? There are all sorts of approaches. We'll look at two representative ones, and their advantages and disadvantages.

Add the Digits

A simple approach to converting a word to a number might be to simply add the code numbers for each character. Say we want to convert the word cats to a number. First convert the characters to digits using our homemade code:

c = 3

a = 1

t = 20

s = 19

Then we add them:

3 + 1 + 20 + 19 = 43

Thus in our dictionary the word cats would be stored in the array cell with index 43. All the other English words would likewise be assigned an array index calculated by this process.

How well would this work? For the sake of argument, let's restrict ourselves to 10-letter words. Then (remembering that a blank is 0), the first word in the dictionary, a, would be coded by

0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 1 = 1

The last potential word in the dictionary would be zzzzzzzzzz (ten Zs). Our code obtained by adding its letters would be

26 + 26 + 26 + 26 + 26 + 26 + 26 + 26 + 26 + 26 = 260

Thus the total range of word codes is from 1 to 260. Unfortunately, there are 50,000 words in the dictionary, so there aren't enough index numbers to go around. Each array element will need to hold about 192 words (50,000 divided by 260).

Clearly this presents problems if we're thinking in terms of our one word-per-array element scheme. Maybe we could put a subarray or linked list of words at each array element. However, this would seriously degrade the access speed. It would be quick to access the array element, but slow to search through the 192 words to find the one we wanted.

So our first attempt at converting words to numbers leaves something to be desired. Too many words have the same index. (For example, was, tin, give, tend, moan, tick, bails, dredge, and hundreds of other words add to 43, as cats does.) We conclude that this approach doesn't discriminate enough, so the resulting array has too few elements. We need to spread out the range of possible indices.

Multiply by Powers

Let's try a different way to map words to numbers. If our array was too small before, let's make sure it's big enough. What would happen if we created an array in which every word, in fact every potential word, from a to zzzzzzzzzz, was guaranteed to occupy its own unique array element?

To do this, we need to be sure that every character in a word contributes in a unique way to the final number.

We'll begin by thinking about an analogous situation with numbers instead of words.

Recall that in an ordinary multi-digit number, each digit position represents a value 10 times as big as the position to its right. Thus 7,546 really means

7*1000 + 5*100 + 4*10 + 6*1

Or, writing the multipliers as powers of 10:

$7*10^3 + 5*10^2 + 4*10^1 + 6*10^0$

(An input routine in a computer program performs a similar series of multiplications and additions to convert a sequence of digits, entered at the keyboard, into a number stored in memory.)

In this system we break a number into its digits, multiply them by appropriate powers of 10 (because there are 10 possible digits), and add the products.

In a similar way we can decompose a word into its letters, convert the letters to their numerical equivalents, multiply them by appropriate powers of 27 (because there are 27 possible characters, including the blank), and add the results. This gives a unique number for every word.

Say we want to convert the word cats to a number. We convert the digits to numbers as shown earlier. Then we multiply each number by the appropriate power of 27, and add the results:

$$3*27^3 + 1*27^2 + 20*27^1 + 19*27^0$$

Calculating the powers gives

$$3*19{,}683 + 1*729 + 20*27 + 19*1$$

And multiplying the letter codes times the powers yields

$$59{,}049 + 729 + 540 + 19$$

Which sums to 60,337.

This process does indeed generate a unique number for every potential word. We just calculated a four-letter word. What happens with larger words? Unfortunately the range of numbers becomes rather large. The largest 10-letter word, zzzzzzzzzz, translates into

$$26*27^9 + 26*27^8 + 26*27^7 + 26*27^6 + 26*27^5 + 26*27^4 + 26*27^3 + 26*27^2 + 26*27^1 + 26*27^0$$

Just by itself, 27^9 is more than 7,000,000,000,000, so you can see that the sum will be huge. An array stored in memory can't possibly have this many elements.

The problem is that this scheme assigns an array element to every potential word, whether it's an actual English word or not. Thus there are cells for aaaaaaaaaa, aaaaaaaaab, aaaaaaaaac, and so on, up to zzzzzzzzzz. Only a small fraction of these are necessary for real words, so most array cells are empty.

Our first scheme—adding the numbers—generated too few indices. This latest scheme—adding the numbers times powers of 27—generates too many.

Hashing

What we need is a way to compress the huge range of numbers we obtain from the numbers-multiplied-by-powers system into a range that matches a reasonably sized array.

How big an array are we talking about for our English dictionary? If we only have 50,000 words, you might assume our array should have approximately this many elements. However, it turns out we're going to need an array with about twice this many cells. (It will become clear later why this is so.) So we need an array with 100,000 elements.

Thus we look for a way to squeeze a range of 0 to more than 7,000,000,000,000 into the range 0 to 100,000. A simple approach is to use the modulo operator (%), which finds the remainder when one number is divided by another.

To see how this works, let's look at a smaller and more comprehensible range. Suppose we squeeze numbers in the range 0 to 199 (we'll represent them by the variable largeNumber) into the range 0 to 9 (the variable smallNumber). There are 10 numbers in the range of small numbers, so we'll say that a variable smallRange has the value 10.

It doesn't really matter what the large range is (unless it overflows the program's variable size). The Java expression for the conversion is

```
smallNumber = largeNumber % smallRange;
```

The remainders when any number is divided by 10 are always in the range 0 to 9; for example, 13%10 gives 3, and 157%10 is 7.

A similar expression can be used to compress the really huge numbers that uniquely represent every English word into index numbers that fit in our dictionary array:

```
arrayIndex = hugeNumber % arraySize;
```

This is an example of a hash function. It hashes (converts) a number in a large range into a number in a smaller range. This smaller range corresponds to the index numbers in an array. An array into which data is inserted using a hash function is called a hash table.

Collisions

We pay a price for squeezing a large range into a small one. There's no longer a guarantee that two words won't hash to the same array index.

This is similar to what happened when we added the letter codes, but the situation is nowhere near as bad. When we added the letters, there were only 260 possible results (for words up to 10 letters). Now we're spreading this out into 50,000 possible results.

Even so, it's impossible to avoid hashing several different words into the same array location, at least occasionally. We'd hoped that we could have one data item per index number, but this turns out not to be possible. The best we can do is hope that not too many words will hash to the same index.

Perhaps you want to insert the word melioration into the array. You hash the word to obtain its index number, but find that the cell at that number is already occupied by the word demystify, which happens to hash to the exact same number (for a certain size array).

It may appear that the possibility of collisions renders the hashing scheme impractical, but in fact we can work around the problem in a variety of ways.

Remember that we've specified an array with twice as many cells as data items. Thus perhaps half the cells are empty. One approach, when a collision occurs, is to search the array in some systematic way for an empty cell, and insert the new item there, instead of at the index specified by the hash function. This approach is called open addressing. If cats hashes to 5,421, but this location is already occupied by parsnip, then we might try to insert cats in 5,422, for example.

A second approach (mentioned earlier) is to create an array that consists of linked lists of words instead of the words themselves. Then when a collision occurs, the new item is simply inserted in the list at that index. This is called separate chaining.

In the balance of this chapter we'll discuss open addressing and separate chaining, and then return to the question of hash functions.

Open Addressing

In open addressing, when a data item can't be placed at the index calculated by the hash function, another location in the array is sought. We'll explore three methods of open addressing, which vary in the method used to find the next vacant cell. These methods are linear probing, quadratic probing, and double hashing.

Linear Probing

In linear probing we search sequentially for vacant cells. If 5,421 is occupied when we try to insert cats there, we go to 5,422, then 5,423, and so on, incrementing the index until we find an empty cell. This is called linear probing because it steps sequentially along the line of cells.

Java Code for a Linear Probe Hash Table

It's not hard to create methods to handle search, insertion, and deletion with linear-probe hash tables. We'll show the Java code for these methods, and then a complete hash.java program that puts them in context.

The find() Method

The find() method first calls hashFunc() to hash the search key to obtain the index number hashVal. The hashFunc() method applies the % operator to the search key and the array size, as we've seen before.

Next, in a while condition, find() checks if the item at this index is empty (null). If not, it checks if the item contains the search key. If it does, it returns the item. If it doesn't, find() increments hashVal and goes back to the top of the while loop to check if the next cell is occupied. Here's the code for find():

```
public DataItem find(int key)          // find item with key  (assumes table not full)
{
        int hashVal = hashFunc(key);          // hash the key
        while (hashArray[hashVal] != null)    // until empty cell,
        {                                     // found the key?
                If (hashArray[hashVal].iData == key)
                        return hashArray[hashVal]; // yes, return item
                ++hashVal;                    // go to next cell
                hashVal %= arraySize;         // wrap around if necessary
        }
        return null;                          // can't find item
}
```

The insert() Method

The insert() method uses about the same algorithm as find() to locate where a data item should go. However, it's looking for an empty cell or a deleted item (key –1), rather than a specific item. Once this empty cell has been located, insert() places the new item into it.

```
public void insert(DataItem item)      // insert a DataItem (assumes table not full)
{
        int key = item.iData;              // extract key
        int hashVal = hashFunc(key); // hash the key until empty cell or -1,
        while(hashArray[hashVal] != null && hashArray[hashVal].iData != -1)
        {
                ++hashVal;                      // go to next cell
                hashVal %= arraySize;           // wrap around if necessary
        }
        hashArray[hashVal] = item; // insert item
}        // end insert()
```

The delete() Method

The delete () method finds an existing item using code similar to find (). Once the item is found, delete () writes over it with the special data item nonItem, which is predefined with a key of -1.

```
public DataItem delete(int key)                 // delete a DataItem
{
        int hashVal = hashFunc(key);            // hash the key
        while(hashArray[hashVal] != null)       // until empty cell,
        {                                       // found the key?
                if(hashArray[hashVal].iData == key)
                {
                        DataItem temp = hashArray[hashVal]; // save item
                        hashArray[hashVal] = nonItem; // delete item
                        return temp;            // return item
                }
```

```
            ++hashVal;                  // go to next cell
            hashVal %= arraySize;       // wrap around if necessary
        }
        return null;    // can't find item
}       // end delete()
```

Quadratic Probing

We've seen that clusters can occur in the linear probe approach to open addressing.

Once a cluster forms, it tends to grow larger. Items that hash to any value in the range of the cluster will step along and insert themselves at the end of the cluster, thus making it even bigger. The bigger the cluster gets, the faster it grows.

It's like the crowd that gathers when someone faints at the shopping mall. The first arrivals come because they saw the victim fall; later arrivals gather because they wondered what everyone else was looking at. The larger the crowd grows, the more people are attracted to it.

The ratio of the number of items in a table, to the table's size, is called the load factor. A table with 10,000 cells and 6,667 items has a load factor of 2/3.

```
loadFactor = nItems / arraySize;
```

Clusters can form even when the load factor isn't high. Parts of the hash table may consist of big clusters, while others are sparsely inhabited. Clusters reduce performance.

Quadratic probing is an attempt to keep clusters from forming. The idea is to probe more widely separated cells, instead of those adjacent to the primary hash site.

The Step Is the Square of the Step Number

In a linear probe, if the primary hash index is x, subsequent probes go to x+1, x+2, x+3, and so on. In quadratic probing, probes go to x+1, x+4, x+9, x+16, x+25, and so on. The distance from the initial probe is the square of the step number: $x+1^2$, $x+2^2$, $x+3^2$, $x+4^2$, $x+5^2$, and so on.

Double Hashing

To eliminate secondary clustering as well as primary clustering, another approach can be used: double hashing. Secondary clustering occurs because the algorithm that generates the

sequence of steps in the quadratic probe always generates the same steps: 1, 4, 9, 16, and so on.

What we need is a way to generate probe sequences that depend on the key instead of being the same for every key. Then numbers with different keys that hash to the same index will use different probe sequences.

The solution is to hash the key a second time, using a different hash function, and use the result as the step size. For a given key the step size remains constant throughout a probe, but it's different for different keys.

Experience has shown that this secondary hash function must have certain characteristics:

- It must not be the same as the primary hash function.
- It must never output a 0 (otherwise there would be no step; every probe would land on the same cell, and the algorithm would go into an endless loop).

Experts have discovered that functions of the following form work well:

```
stepSize = constant - (key % constant);
```

where constant is prime and smaller than the array size. For example,

```
stepSize = 5 - (key % 5);
```

This is the secondary hash function used in the Workshop applet. For any given key all the steps will be the same size, but different keys generate different step sizes. With this hash function the step sizes are all in the range 1 to 5.

Table Size a Prime Number

Double hashing requires that the size of the hash table is a prime number. To see why, imagine a situation where the table size is not a prime number. For example, suppose the array size is 15 (indices from 0 to 14), and that a particular key hashes to an initial index of 0 and a step size of 5. The probe sequence will be 0, 5, 10, 0, 5, 10, and so on, repeating endlessly. Only these three cells are ever examined, so the algorithm will never find the empty cells that might be waiting at 1, 2, 3, and so on. The algorithm will crash and burn.

If the array size were 13, which is prime, the probe sequence eventually visits every cell.

It's 0, 5, 10, 2, 7, 12, 4, 9, 1, 6, 11, 3, and so on and on. If there is even one empty cell, the probe will find it. Using a prime number as the array size makes it impossible for any number to divide it evenly, so the probe sequence will eventually check every cell.

A similar effect occurs using the quadratic probe. In that case, however, the step size gets larger with each step, and will eventually overflow the variable holding it, thus preventing an endless loop.

In general, double hashing is the probe sequence of choice when open addressing is used.

Exercises:

Question 1

Write a program to manage a hash table, using open addressing, where the keys are book ISBNs. Your code should provide create, insert, find and delete operations on that table.
Use a hash function suitable for character strings and motivate your answer in a comment stored in the heading area of your hash table processing routines. Output for find should be yes, followed by the table index if found or no if not found.

Question 2

Write a program to manage a hash table, using open addressing, where the keys are student full names. Your code should provide create, insert, find and delete operations on that table.
Use a hash function suitable for character strings and motivate your answer in a comment stored in the heading area of your hash table processing routines. Output for find should be yes, followed by the table index if found or no if not found.

Question 3

Write a program to manage a hash table, using open addressing, with numeric long integer keys, where the hash function should be selectable before each run. The methods you should use in building your hash functions are linear, quadratic and double hashing. Your code should provide create, insert, find and delete operations on that table.
Output for find should be yes, followed by the table index if found or no if not found.

Question 4

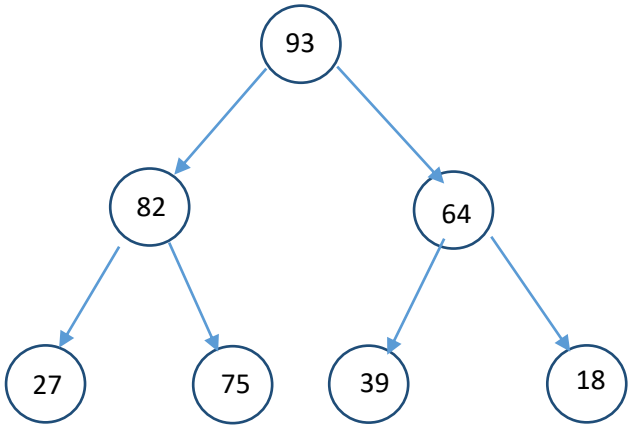
Write a program to manage a hash table, using open addressing, with character string keys, where the hash function should be selectable before each run. The methods you should use in building your hash functions are linear, quadratic and double hashing. Your code should provide create, insert, find and delete operations on that table.
Output for find should be yes, followed by the table index if found or no if not found.

Lab 10: Heaps

AIM: To write a program that implement the basic operations of the heap.

Overview:

Linked structures are not the only way in which you can represent trees. If you take the binary tree shown below and copy its contents into an array in level order, you produce the following array.



Index	Entry
0	93
1	82
2	64
3	27
4	75
5	39
6	18

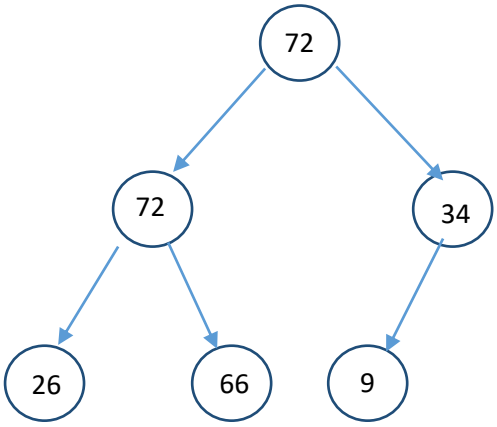
Examining the relationship between positions in the tree and entries in the array, you see that if an element is stored in entry N in the array, then the element's left child is stored in entry

2N + 1, its right child is stored in entry 2N + 2, and its parent is stored in entry (N _ 1) / 2. These mappings make it easy to move through the tree stepping from parent to child (or vice versa).

A heap is a binary tree that meets the following conditions.

- The tree is complete. That is, every level in the tree is filled, except possibly the bottom level. If the bottom level is not filled, then all the missing elements occur on the right.
- Each element in the tree has a corresponding value. For each element E, all of E's descendants have values that are less than or equal to E's value. Therefore, the root stores the maximum of all values in the tree. (Note: In this laboratory we are using a max-heap. There is another heap variant called a min-heap. In a min-heap, all of E's descendants have values that are greater than or equal to E's value.)

The tree shown at the beginning of this laboratory is a heap, as is the tree shown below.



The fact that the tree is complete means that a heap can be stored in level order in an array without introducing gaps (unused areas) in the middle. The result is a compact representation in which you can easily move up and down the branches in a heap.

Clearly, the relationship between the priorities (or values) of the various elements in a heap is not strong enough to support an efficient search process. Because the relationship is simple, however, you can quickly restructure a heap after removing the highest priority (root) element or after inserting a new element. As a result, you can rapidly process the elements in a heap in descending order based on priority. This property combined with the compact array representation forms the basis for an efficient sorting algorithm called heap sort and makes a heap an ideal representation for a priority queue.

Heap ADT

Elements

The elements in a heap are of generic type HeapData defined in the file HeapData.java. Each element has a priority that is used to determine the relative position of the element within the heap. Elements usually include additional data. Note that priorities are not unique-it is quite likely that several elements have the same priority. These objects must support the six basic relational operators, as well as a method called pty( ) that returns an element's priority.

Structure

The elements form a complete binary tree. This is a max-heap. For each element E in the tree, all of E's descendants have priorities that are less than or equal to E's priority.

Constructors and Methods

Heap ( )

Precondition:

None.

Post condition:

Default Constructor. Calls setup, which creates an empty heap. Allocates enough memory for a heap containing DEF_MAX_HEAP_SIZE (a constant value) elements.

Heap (int maxNumber)

Precondition:

maxNumber > 0.

Post condition:

Constructor. Calls setup, which creates an empty heap. Allocates enough memory for a heap containing maxNumber elements.

void setup (int maxNumber)

Precondition:

maxNumber > 0. A helper method for the constructors. Is declared private since only heap constructors should call this method.

Post condition:

Creates an empty heap. Allocates enough memory for a heap containing maxNumber elements.

void insert (HeapData newElement)

Precondition:

Heap is not full.

Post condition:

Inserts newElement into a heap. Inserts this element as the bottom rightmost element in the heap and moves it upward until the properties that define a heap are restored. Note that repeatedly swapping array elements is not an efficient way of positioning the newElement in the heap.

HeapData removeMax ( )

Precondition:

Heap is not empty.

Post condition:

Removes the element with the highest priority (the root) from a heap and returns it.

Replaces the root element with the bottom rightmost element and moves this element downward until the properties that define a heap are restored. Note that (as in insert above) repeatedly swapping array elements is not an efficient method for restoring the heap.

void clear ( )

Precondition:

None.

Post condition:

Removes all the elements in a heap.

boolean isEmpty ( )

Precondition:

None.

Post condition:

Returns true if a heap is empty. Otherwise, returns false.

boolean isFull ( )

Precondition:

None.

Post condition:

Returns true if a heap is full. Otherwise, returns false.

Exercises:
Question 1

Step 1: Implement the operations in Heap ADT using an array representation of a heap. Heaps can be different sizes; therefore you need to store the actual number of elements in the heap (size), along with the heap elements themselves (element). Remember that in Java the size of the array is held in a constant called length in the array object. Therefore, in Java a separate variable (such as *maxSize*) is not necessary, since the maximum number of elements our heap can hold can be determined by referencing length - more specifically in our case, *element.length.*

You are to fill in the Java code for each of the constructors and methods where only the method headers are given. Each method header appears on a line by itself and does not contain a semicolon. This is not an interface file, so a semicolon should not appear at the end of a method header. Each of these methods needs to be fully implemented by writing the body of code for implementing that particular method and enclosing the body of that method in braces.

```
public class Heap
{
        // Constant
        private static final int DEF_MAX_HEAP_SIZE = 10; // Default maximum heap size
        // Data members
        private int size;                   // Actual number of elements in the heap
        private HeapData [ ] element; // Array containing the heap elements
        // ___The following are Method Headers ONLY ___ //
        // each of these methods needs to be fully implemented
        // Constructors and helper method setup
        public Heap ( )                     // Constructor: default size
        public Heap ( int maxNumber ) // Constructor: specific size
        // Class methods
        private void setup ( int maxNumber ) // Called by constructors only
        // Heap manipulation methods
        public void insert ( HeapData newElement ) // Insert element
```

```
    public HeapData removeMax ( ) // Remove max pty element
    public void clear ( ) // Clear heap
    // Heap status methods
    public boolean isEmpty ( )      // Heap is empty
    public boolean isFull ( )       // Heap is full
    // Output the heap structure - used in testing/debugging
    public void showStructure ( )
    // Recursive partner of the showStructure() method
    private void showSubtree ( int index, int level )
}       // class Heap
```

Step 2: Save your implementation of the Heap ADT in the file Heap.java. Be sure to document your code.

Question 2

After removing the root element, the *removeMax* operation inserts a new element at the root and moves this element downward until a heap is produced. The following method performs a similar task, except that the heap it is building is rooted at array entry root and occupies only a portion of the array.

void moveDown (HeapData [ ] element, int root, int size)

Precondition:

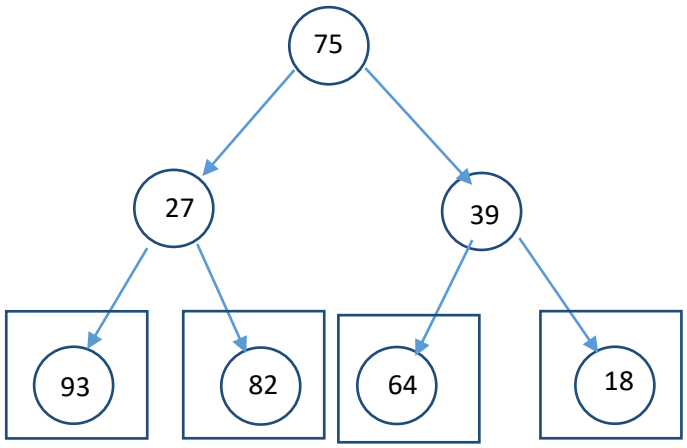
The left and right subtrees of the binary tree rooted at root are heaps.

Post condition:

Restores the binary tree rooted at root to a heap by moving element [root] downward until the tree satisfies the heap property. Parameter size is the number of elements in the array. Remember that repeatedly swapping array elements is not an efficient method for restoring the heap.

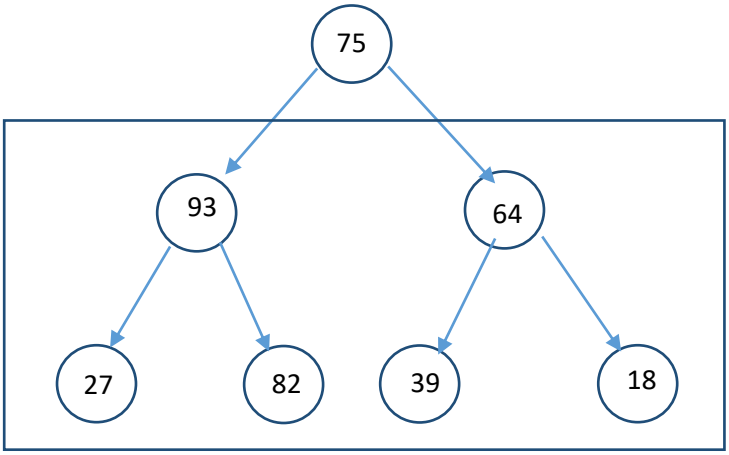
In this exercise, you implement an efficient sorting algorithm called heap sort using the *moveDown( )* method. You first use this method to transform an array into a heap. You then remove elements one-by-one from the heap (from the highest priority element to the lowest) until you produce a sorted array.

Let's begin by examining how you transform an unsorted array into a heap. Each leaf of any binary tree is a one-element heap. You can build a heap containing three elements from a pair of sibling leaves by applying the *moveDown( )* method to that pair's parent. The four *singleelement* heaps (leaf nodes) in the following tree



Index	Entry
0	75
1	27
2	39
3	93
4	82
5	64
6	18

Are transformed by the calls *moveDown*(sample, 1, 7) and *moveDown*(sample, 2, 7) into a pair of three-element heaps:

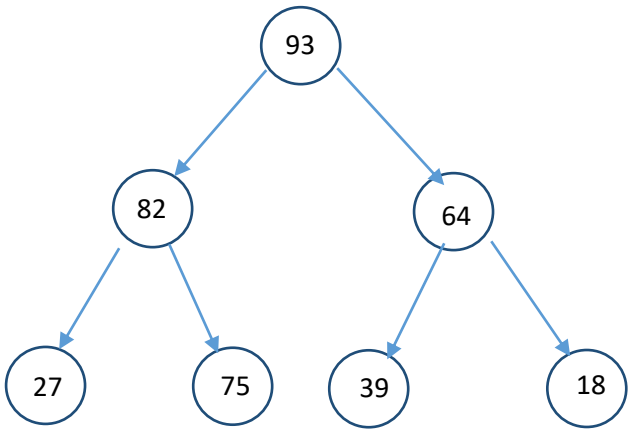


Index	Entry
0	75
1	93
2	64
3	27
4	82
5	39
6	18

By repeating this process, you build larger and larger heaps, until you transform the entire tree (array) into a heap.

```
// Build successively larger heaps within the array until the
// entire array is a heap.
for ( j = (size - 1) / 2; j >= 0; j-- )
        moveDown( element, j, size );
```

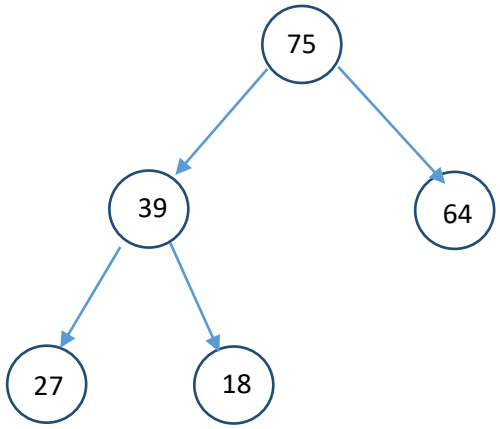
Combining the pair of three-element heaps shown previously using the call moveDown(sample, 0, 7), for instance, produces the following heap.



Index	Entry
0	93
1	82
2	64
3	27
4	75
5	39
6	18

Now that you have a heap, you remove elements of decreasing priority from the heap and gradually construct an array that is sorted in ascending order. The root of the heap contains the highest priority element. If you swap the root with the element at the end of the array and use moveDown( ) to form a new heap, you end up with a heap containing six elements and a sorted array containing one element. Performing this process a second time yields a heap containing five elements and a sorted array containing two elements.

Heap



Sorted array

Index	Entry
0	75
1	39
2	64
3	27
4	18
5	82
6	93

You repeat this process until the heap is gone and a sorted array remains.

```
// Swap the root element from each successively smaller heap with
// the last unsorted element in the array. Restore the heap after
// each exchange.
for ( j = size - 1; j > 0; j-- )
{
        temp = element[ j ];
        element[ j ] = element[ 0 ];
        element[ 0 ] = temp;
        moveDown( element, 0, j );
}
```

A shell containing a heapSort( ) method comprised of the two loops shown above is given in the file TestHeapSort.jshl.

Step 1: Using your implementation of the removeMax operation as a basis, create an implementation of the moveDown( ) method.

Step 2: Before testing the resulting heapSort( ) method using the test program in the file TestHeapSort.java, prepare a test plan for the heapSort( ) method that covers arrays of different lengths containing a variety of priority values. Be sure to include arrays that have multiple elements with the same priority. A test plan form follows.

Question 3

A priority queue is a linear data structure in which the elements are maintained in descending order based on priority. You can only access the element at the front of the queue-that is, the element with the highest priority-and examining this element entails removing (dequeuing) it from the queue.

You can easily and efficiently implement a priority queue as a heap by using the Heap ADT insert operation to enqueue elements and the removeMax operation to dequeue elements. The following incomplete definitions derive a class called PtyQueue from the Heap class. In Java the keyword extends is used to specify inheritance (class PtyQueue extends Heap means PtyQueue inherits from Heap). Thus, PtyQueue is the subclass and Heap is the superclass. The subclass inherits all of the public and protected instance variables and methods defined by the superclass and adds its own, unique elements as needed.

```
class PtyQueue extends Heap
{
        // Constructor
        public PtyQueue ( ) // Constructor: default size
        { }
        public PtyQueue ( int size ) // Constructor: specific size
        { }
        // Queue manipulation methods
        public void enqueue ( HeapData newElement ) // Enqueue element
        { }
        public HeapData dequeue ( ) // Dequeue element
        { }
} // class PtyQueue
```

Implementations of the Priority Queue ADT constructor, enqueue, and dequeue operations are given in the file PtyQueue.java. These implementations are very short, reflecting the close relationship between the Heap ADT and the Priority Queue ADT. Note that you inherit the remaining operations in the Priority Queue ADT from the Heap class. You may use the file TestPtyQueue.java to test the Priority Queue implementation.

Operating systems commonly use priority queues to regulate access to system resources such as printers, memory, disks, software, and so forth. Each time a task requests access to a system resource, the task is placed on the priority queue associated with that resource. When the task is dequeued, it is granted access to the resource to print, store data, and so on.

Suppose you wish to model the flow of tasks through a priority queue having the following properties:

- One task is dequeued every minute (assuming that there is at least one task waiting to be dequeued during that minute).
- From zero to two tasks are enqueued every minute, where there is a 50% chance that no tasks are enqueued, a 25% percent chance that one task is enqueued, and a 25% chance that two tasks are enqueued.
- Each task has a priority value of zero (low) or one (high), where there is an equal chance of a task having either of these values.

You can simulate the flow of tasks through the queue during a time period n minutes long using the following algorithm.

```
Initialize the queue to empty.
for ( minute = 0 ; minute < n ; ++minute )
{
        If the queue is not empty, then remove the task at the front of the queue.
        Compute a random integer k between 0 and 3.
        If k is 1, then add one task to the queue. If k is 2, then add two tasks.
        Otherwise (if k is 0 or 3), do not add any tasks to the queue.
        Compute the priority of each task by generating a random value of 0 or 1
        (assuming here are only 2 priority levels).
}
```

Lab 11: Sorting 1

AIM: In this lab session we will experiment with a number of sorting algorithms. Specifically: insertion sort, radix sort and merge sort.

Overview:

To sort means to arrange a sequence of elements, like a_0, $a_1 \ldots a_{n-1}$i in ascending or descending order. In practice sorting is used to arrange records based on a key – the key is a field in the record.

Insertion Sort

Insertion sort takes a sorted sequence $a_0 < a_1 < a_2 < \ldots < a_{j-1}$ and inserts a new element, a_j, in its rightful position by comparing it with a_{j-1}, a_{j-2} . . . till it finds an element $a_i < a_j$. Then, it inserts a_j in the position following a_i. To make room for the element to insert, all the elements in positions $j - 1$, $j - 2 \ldots i + 1$, i.e. a_{j-1}, a_{j-2} . . . a_{i+1} are shifted one position right in the sequence. This method is called direct insertion.

Binary insertion implies executing a binary search for finding the position where a_j is to be inserted, based on the fact that the sequence (a_0, a_1. . . a_{j-1}) is arranged in ascending order.

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

Algorithm: (INSERTION SORT) INSERTION (A, N)

[Where A is an array and N is the number of values in the array]

1. Repeat steps 2 to 4 for K=1, 2, 3 . . . N-1:
2. Set TEMP = A[K] and i =K-1.
3. Repeat while i >= 0 and TEMP < A[i]

    a) Set A[i+1] = A[i]. [Moves element forward.]

    b) Set i = i -1.

    [End of loop.]

4. Set A[i+1] =TEMP. [Insert element in proper place.]

   [End of Step 2 loop.]

5. Return.

Radix Sort

Here 'A' is Linear Array and N is the number of values stored in A and R is the Radix/Base (10) of Number System. Here, recursive function is going to use for RADIX SORT. For detail/explanation about this algorithm, see the lecture notes.

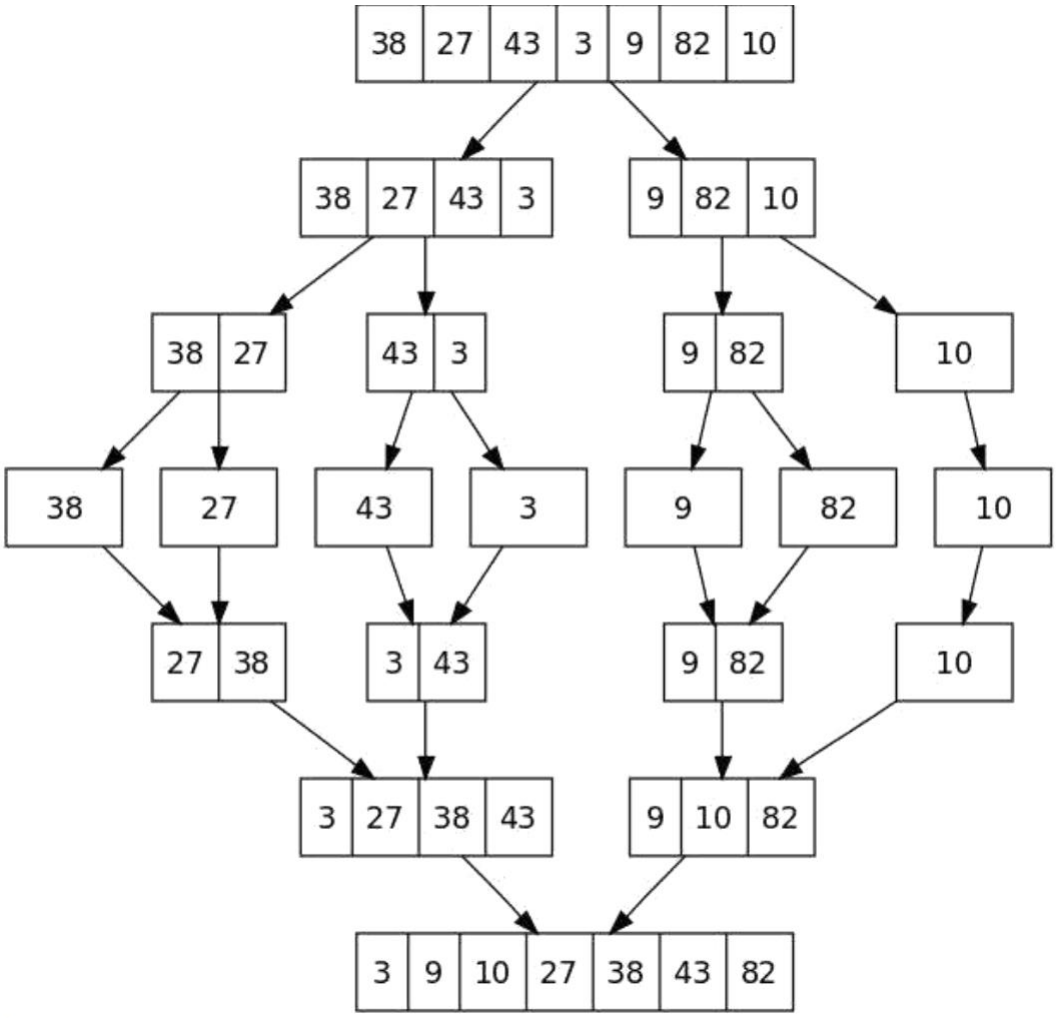
Algorithm: RADIXSORT (A, N, R)

1. If N=1, then: Return.
2. Set D=R/10, P=-1.
3. Repeat for i=0, 1, 2 . . . . (N-1)
    Repeat for j=0, 1, 2 . . . 9
        Set C [i] [j]:= -1.
4. Repeat for i=0, 1, 2 . . . . (N-1)
    i. Set X=A [i] % R.
    ii. Set m=X/D.
    iii. If m>0, then Set Z=1.
    iv. Set C [i] [m] = A [i].
5. Repeat for j=0, 1, 2 9
    i. Repeat for j=0, 1, 2 . . . . (N-1)
        1. If C [i] [j] ≠ -1, then:
            a. Set P=P+1.
            b. Set A [P] =C [i] [j].
6. If Z=1, then
    RADIXSORT (A, N, R*10).
7. Return.

Merge Sort

Here 'A' is Linear Array and N is the number of values stored in A. Here, recursive function is going to use for MERGE SORT. For detail about this algorithm, see the lecture notes.

Trace of Merge Sort



Algorithm: MergeSort (A, BEG, END)

[Here A is an unsorted array. BEG is the lower bound and END is the upper bound.]

1. If (BEG < END) Then
    a) Set MID = (BEG + END) / 2
    b) Call MergeSort (A, BEG, MID)
    c) Call MergeSort (A, MID + 1, END)
    d) Call MERGE (A, BEG, MID, END)

    [End of If of step-1]

2. Return

Algorithm: MERGE (A, BEG, MID, END)

[Here A is an unsorted array. BEG is the lower bound, END is the upper bound and MID is the middle value of array. B is an empty array.]

1. Repeat for I = BEG to END
   Set B[I] = A[I] [Assign array A to B]
   [End of For Loop]
2. Set I = BEG, J = MID + 1, K = BEG
3. Repeat step-4 While (I <= MID) and (J <= END)
4. If (B[I] <= B[J]) Then [Assign smaller value to A]
   a) Set A[K] = B[I]
   b) Set I = I + 1 and K = K + 1
   Else
   a) Set A[K] = B[J]
   b) Set J = J + 1 and K = K + 1
   [End of If]

[End of While Loop of step-3]

5. If (I <= MID) Then [Check whether first half has finished or not]
   a) Repeat While (I <= MID)
      i. Set A[K] = B[I]
      ii. Set I = I + 1 and K = K + 1
   [End of While Loop step-5(a)]
   Else
   b) Repeat While (J <= END)
      i. Set A[K] = B[J]
      ii. Set J = J + 1 and K = K + 1
   [End of While Loop step-5(b)]

[End of If of step-5]

6. End

Exercises:

Question 1

Running an actual program, count the number of compares needed to sort n values using insertion sort, where n varies (e.g., powers of 2).

Question 2

Running an actual program, and using the millisecond timer, System.currentTimeMillis, measure the length of time needed to sort arrays of data of various sizes using a sort of your choice.

Question 3

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

Lab 12: Sorting 2

AIM: In this lab session we will experiment with a number of sorting algorithms. Specifically: Quick Sort, shell sort and Binary Tree Sort.

Quick Sort

Here 'A' is Linear Array and N is the number of values stored in A. Here, recursive function is going to use for QUICK SORT. For detail about this algorithm, see the lecture notes.

Quick sort is an algorithm of the divide-and-conquer type. That is, the problem of sorting a set is reduced to the problem of sorting two smaller sets.

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		

Algorithm: QUICKSORT (A, LEFT, RIGHT)

1. Low = Left
2. High = right
3. Key = A[(Left + Right)/2]
4. Repeat step 9 while (Low <= High)
5. Repeat step 6 while (a[Low] < Key)
6. Low = Low + 1
7. Repeat step 8 while (a[High] > key)
8. High = High – 1
9. If (Left <= High)
    a. Temp = A[Low]
    b. A[Low] = A[High]
    c. A[High] = temp
    d. Low = Low + 1
    e. High = High + 1
10. If (Left < High) QUICKSORT (A, Left, High)
11. If (Right > Low) QUICKSORT (A, Low, Right)
12. Exit

Shell sort

Another insertion sort method is known as shell sort. To understand how this works, we will use the concept of h–sorting. h– sorting means direct insertion of the sequences:

$<h_0, a_h, a_{2h} \ldots>$

$<h_1, a_{1+h}, a_{1+2h} \ldots>$

...

$<a_h, a_{2h}, a_{3h} \ldots>$

h is called an increment. Then, shell sort involves the selection of k increments, in descending order, i.e.

$h_1 > h_2 > h_3 > \ldots > h_k = 1$

And performing a h1–sort, then a h2–sort, and so on, and finally a 1–sort.

The performance of the method is tightly connected to the selection of the increments.

Trace of Shell Sort

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
The Array	80	93	60	12	42	30	68	85	10

N = 9

Gap = 4

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
The Array	10	30	60	12	42	93	68	85	80

N = 9

Gap = 2

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
The Array	10	12	42	30	60	85	68	93	80

N = 9

Gap = 1

Algorithm

1. n = length of the list, increment = n/2
2. Do the following until increment>0
    i = increment until i<n do the following
        Store the value at index i in a temporary variable(temp)
        j = i until j>= increment do the following
            - If temp is less than the value at index j-increment
                Replace the value at index j with the value at index j-increment and decrease j by increment.
            - Else break out of the j loop.
        Replace the value at index j with temp and increase i by 1.
    Divide increment by 2.

Property:

1. Best-Case Complexity- O(n) when array is already sorted.
2. Worst-Case Complexity-It depends on gap sequence; best known is $O(n(\log n)^2)$ and occurs when array is sorted in reverse order.
3. Average-Case Complexity- It also depends on gap sequence.
4. O(1) extra space.
5. It's only efficient for medium size lists. it is a complex algorithm and it's not nearly as efficient as the merge, heap, and quick sorts

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.

Algorithm

```
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();
    }
}
```

Exercises:

Question 1

Modify the partition method used by quicksort so that the pivot is randomly selected.

Question 2

Generate a 100-element list containing integers in the range 0-10. Sort the list with shell sort and with quick sort.

1. Which is faster? Why?
2. Try this again, but with 100,000 elements. Note the relative difference> Why does it exists?

The material included in this manual has been entirely adopted from the following references:

- http://it.yu.edu.jo/index.php?option=com_docman&task=doc_download&gid=174&Itemid=171
- http://www.cin.ufpe.br/~grm/downloads/Data_Structures_and_Algorithms_in_Java.pdf
- https://mrajacse.files.wordpress.com/2012/08/data-structures-lab-manual.pdf
- http://www.nitkkr.ac.in/clientFiles/FILE_REPO/2014/DEC/8/1418017571401/Advanced_Data_Structure_IT-411.pdf
- http://csiflabs.cs.ucdavis.edu/~ssdavis/60/CursorList.pdf
- http://jnec.org/Lab-manuals/ECT/SE/Data%20Structure.pdf
- http://iwlbooks.com/12450-data-structures-in-java-a-laboratory-course-sandra.html
- http://www.cin.ufpe.br/~grm/downloads/Data_Structures_and_Algorithms_in_Java.pdf
- http://mca.griet.ac.in/ppt/DS%20LAB.pdf
- http://www.scribd.com/doc/213944198/DS-CSC-214-LAB-Mannual-for-Students#scribd
- http://www.wctmgurgaon.com/wctm/dsa%20lab-it-labmanual.pdf
- http://users.utcluj.ro/~jim/DSA/Resources/LabCode/DSALab.pdf
- https://fcsiba.wikispaces.com/file/view/DS-Lab+manual.docx
- http://ggnindia.dronacharya.info/CSEDept/Downloads/Labmanuals/Aug09-Dec09/CSE%20&%20IT/III%20sem/DSLab_CSE-III.pdf
- www.rajalakshmi.org/Dept/EEE/EE2209-LM.doc
- http://www.srmuniv.ac.in/sites/default/files/files/It0501Data%20Structures&Algorithms%20Lab.pdf
- http://enos.itcollege.ee/~jpoial/allalaadimised/js.pdf
- http://www.kau.edu.sa/GetFile.aspx?id=164699&fn=Lab8-cpcs-204-v3-%28Searching%20&%20Sorting%29.docx
- http://annamalaiuniversity.ac.in/studport/download/engg/cse/labmanual/MCA-I%20SEM%20-%20MCA1710-Prog%20Lab-II-Data%20Structures%20using%20C-Lab%20manual.pdf