

Solutions to a few of the review problems:

Problem 1 (i) Write the Java interface for the ADT Stack. (Just write an interface that specifies methods — **do not** implement the methods.)

Answer:

```
interface Stack {
    public Object pop();
    public void push(Object x);
    public int size();
    public boolean empty();
}
```

(ii) Write the Java interface for the ADT Iterator. (Just write an interface that specifies methods — **do not** implement the methods.)

Answer:

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

(iii) Write a client function with title line:

```
public static int count(Stack s, Object x)
```

The function returns a count of the number of times the object x is found on the Stack s . When the method ends the Stack should store exactly the same data as at the beginning of the method. However your function will need to alter the Stack data as it runs.

Answer:

```
public static int count(Stack s, Object x) {
    Stack temp = new Stack();
    int ans = 0;

    while (!s.empty()) {
        Object y = s.pop();
        if (y.equals(x)) ans++;
        temp.push(y);
    }
    while (!temp.empty()) s.push(temp.pop());
    return ans;
}
```

(iv) Write a client function with title line:

```
public static int count(Iterator i, Object x)
```

The function returns a count of the number of times the object x is found on the Iterator i . This function can and should empty out the Iterator and should not restore it to its original state.

Answer:

```
public static int count(Iterator i, Object x) {
    int ans = 0;
    while (i.hasNext()) {
        Object y = i.next();
        if (y.equals(x)) ans++;
    }
    return ans;
}
```

Problem 2 Give useful O -estimates of the run times of the following methods:

(a) The method *addHead* for a singly linked list that has size n .

Answer: $O(1)$

(b) An efficient method to calculate the power x^n (consider the run time as a function of n , the time should be considered as being proportional to the total number of additions, subtractions, multiplications, and divisions performed).

Answer: $O(\log(n))$

(c) An efficient method to sort an array of n numbers into order.

Answer: $O(n\log(n))$

For (d) and (e), consider the following recursive function, in which A represents an integer constant:

```
int f(int n) {
    if (n <= 0) return 1;
    int ans = f(n/2) * 2;
    for (int i = 1; i<= n; i++)
        for (int j = 1; j <= n; j++)
            ans += i / j;
    for (int k = 1; k < A; k++)
        ans -= f(n/2 - k);
    return ans;
}
```

(d) In the case where $A = 3$ estimate the run time of $f(n)$.

Answer: $O(n^2)$

(e) In the case where $A = 4$ estimate the run time of $f(n)$.

Answer: $O(n^2\log(n))$

Problem 3 The class Queue is to be programmed with an array based implementation. A partial version of the implementation follows. The two most important methods have been omitted.

```
class Queue {
    private Object data[];
    private int front, rear, size;
    public Queue() {
        data = new Object[100]; front = rear = size = 0; }
    public int size() { return size; }
    public boolean empty() { return size == 0; }
    // methods omitted here
}
```

(a) **Identify the two missing methods.** For each give the name, parameters and return type.

Answer:

```
public Object dequeue();
public void enqueue(Object x);
```

(b) **Give a complete implementation of ONE** of the two missing Queue methods. (You can choose either of them.)

Answer:

```
public void enqueue(Object x) {
    if (size() == 100)
        throw new RuntimeException("Queue Full");
    data[rear++] = x;
    if (rear == 100) rear = 0;
    size++;
}
```

Answer:

```
public Object dequeue() {
    if (empty()) throw new RuntimeException("Queue Empty");
    Object answer = data[front++];
    if (front == 100) front = 0;
    size--;
    return answer;
}
```

Problem 4 Give useful O -estimates (in terms of the quantity n) for the run times of the following methods:

(a) The method *removeTail* for a singly linked list that has size n .

Answer: $O(n)$

(b) An efficient method to sort an array that contains n items of data.

Answer: $O(n \log(n))$

For (c) and (d), consider the following recursive method, in which *pow* calculates either the square (in part (c)) or the cube (in part (d)) of its parameter.

```
int f(int n) {
    if (n <= 0) return 1;
    int ans = 0;
    for (int i = 1; i <= pow(n); i++)
        ans += i * i;
    return ans - f(n/2) + f(n/2 - 1) - f(n/2 - 2) + f(n/2 - 3);
}
```

(c) In the case where *pow*(n) calculates n^2 , estimate the run time of $f(n)$.

Answer: $O(n^2 \log(n))$

(d) In the case where *pow*(n) calculates n^3 , estimate the run time of $f(n)$.

Answer: $O(n^3)$

(e) Give a O -estimate for the run time of the following method. Here, the parameter n represents the number of array elements to be considered:

```
int estimateMedian(int a[], int n, int startAt) {
    if (n == 1) return a[startAt];
    int half = n / 2;
    int med1 = estimateMedian(a, half, startAt);
    int med2 = estimateMedian(a, n - half, startAt + half);
    return (med1 + med2) / 2;
}
```

Answer: $O(n)$

Problem 5 Suppose that a singly linked list is implemented as a class *LinkedList* that has a single instance variable *head* of class *Node*. No sentinels are used, so that *head.data* is the first object in the list, and the last node is indicated by a *next* link with value *null*.

Write a method *deFred*() of the class *LinkedList* that removes all entries (and their nodes) that store data equal to the string *Freddy*. For example, if the list l stores the data: $A, B, C, D, E, \textit{Freddy}, G, \textit{Freddy}, H$ a call of the method $l.\textit{deFred}()$ would leave it as: A, B, C, D, E, G, H .

Answer:

```
public void deFred() {
    Node cursor = head;
    while (cursor != null && cursor.getNext() != null) {
        if (cursor.getNext().getData().equals("Freddy"))
            cursor.setNext(cursor.getNext().getNext());
    }
}
```

```

        else cursor = cursor.getNext();
    }
    if (head != null && head.getData().equals("Freddy"))
        head = head.getNext();
}

```

Problem 6 A partial array based implementation of a Deque (with capacity 100) follows. The four most important methods have been omitted.

```

class Deque
{
    private Object data[];
    private int front, rear, size;

    public ArrayDeque() {
        data = new Object[100];
        front = size = 0;
        rear = 1;
    }
    public int size() { return size; }
    public boolean empty() { return size == 0; }

    // methods omitted
}

```

(a) **Identify the four missing methods.** For each give a reasonable name, parameters and return type.

Answer:

```

public Object removeFront();
public Object removeRear();
public void addFront(Object x);
public void addRear(Object x);

```

(b) **Give a complete implementation of ONE** of the four missing Deque methods. (You can choose any one of them.)

Answer:

```

public void addFront(Object x) {
    if (size() == 100)
        throw new RuntimeException("Deque Full");
    data[front--] = x;
    if (front < 0) front = 99;
    size ++;
}

public void addRear(Object x) {
    if (size() == 100)
        throw new RuntimeException("Deque Full");
    data[rear++] = x;
    if (rear == 100) rear = 0;
    size ++;
}

public Object removeFront() {
    if (empty()) throw new RuntimeException("Deque Empty");
    front = front + 1;
    if (front == capacity) front = 0;
    size --;
}

```

```

        return data[front];
    }

    public Object removeRear() {
        if (empty()) throw new RuntimeException("Deque Empty");
        rear = rear - 1;
        if (rear == -1) rear = capacity - 1;
        size--;
        return data[rear];
    }

```

(c) What additions would need to be made to the Java implementation to make the Deque iterable. (Identify one change to the title line for the class and one change within the body of the class.)

Answer:

The title line should be changed to:

```
class Deque implements Iterable { ...
```

and a method with title

```
public Iterator iterator() { ....
```

should be added to the body of the class

(**Extra credit**) Implement the changes that you specified in (c), and give an implementation of an appropriate Iterator class, if this is needed. (Write your implementation on the facing page.)

Answer:

```

    public Iterator iterator() {
        Vector v = new Vector();
        int cursor = front;
        for (int i = 0; i < size; i++) {
            cursor++;
            if (cursor == capacity) cursor = 0;
            v.add(data[cursor]);
        }
        return v.iterator();
    }

```

Problem 7 Write a complete DList adapter implementation of the interface *Deque*. (Do not supply any implementation for the classes *DNode* and *DList*.)

Assume that a doubly linked list is available and has been implemented as the class *DList*. The implementation of class *DList* uses an auxiliary class *DNode* and provides a constructor and the methods *size*, *isEmpty*, *getFirst*, *getLast*, *getBefore*, *getAfter*, *addFirst*, *addLast*, *addBefore*, *addAfter*, and *remove* as discussed in class.

Answer:

```

class DListAdapterDeque implements Deque
{
    private DList l;

    public DListAdapterDeque() {
        l = new DList();
    }

    public int size() {
        return l.size();
    }
}

```

```

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

public void addFront(Object x) {
    l.addFirst(x);
}

public void addRear(Object x) {
    l.addLast(x);
}

public Object removeFront() {
    return l.remove(l.getFirst());
}

public Object removeRear() {
    return l.remove(l.getLast());
}
}

```

Problem 8 Give useful O -estimates of the run times of the following methods:

(a) The method *addTail* for a singly linked list that has size n .

Answer: $O(1)$.

(b) An efficient method to calculate the power x^n (consider the run time as a function of n).

Answer: $O(\log n)$.

You could make a case that the answer should be $O(n)$, and this will be accepted.

(c) An efficient method to calculate the n^{th} Fibonacci number.

Answer: $O(n)$.

You could make a case that the answer should be $O(\log n)$, and this will be accepted.

For (d) and (e), consider the following recursive function, in which A represents an integer constant:

```

int f(int n) {
    if (n <= 0) return 1;
    int ans = 0;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= i; j++)
            ans += i * j;
    for (int k = 1; k <= A; k++)
        ans -= f(n/2 - k);
    return ans;
}

```

(d) In the case where $A = 1$ estimate the run time of $f(n)$.

Answer: Overhead $h(n) = O(n^2)$, Effective recursive cost $r(n) = n^{\log_2(1)} = n^0$. By the Master theorem, the runtime $t(n)$ satisfies $t(n) = O(n^2)$.

(e) In the case where $A = 4$ estimate the run time of $f(n)$.

Answer: Overhead $h(n) = O(n^2)$, Effective recursive cost $r(n) = n^{\log_2(4)} = n^2$. By the Master theorem, the runtime $t(n)$ satisfies $t(n) = O(n^2 \log n)$.

Problem 9 Consider the following partial implementation of a circular list of singly linked nodes.

```

public class CircularList {
    private Node cursor;
    public CircularList() { cursor = null; }
}

```

```

public boolean isEmpty() {return cursor == null;}
public void advance() { cursor = cursor.getNext(); }
public void addAfter(Object d) { CODE OMITTED TO SAVE SPACE }
public void addBefore(Object d) {
    addAfter(d);
    swapData(cursor, cursor.getNext());
    cursor = cursor.getNext();
}
private void swapData(Node n, Node m) { // helper method for addBefore and remove
    Object temp = n.getData();
    n.setData(m.getData()); m.setData(temp); }
public Object remove() { CODE OMITTED HERE }
public String toString() { CODE OMITTED HERE }
}

```

Supply an implementation for the missing method *toString*. The output from your method should match the following format (which indicates a circular list with size 3, stored data A, B, C and the cursor at item A):

A -> B -> C ->

Answer:

```

public String toString() {
    if (cursor == null) return "";
    String ans = cursor.getData() + " ->";
    Node n = cursor.getNext();
    while (n != cursor) {
        ans += n.getData();
        ans += " ->";
        n = n.getNext();
    }
    return ans;
}

```

(Extra credit) Write an implementation of the method *remove*. (Hint: Use the *swapData()* method, and apply the trick used in *addBefore()*.)

Answer:

```

public Object remove() {
    if (cursor == null) throw new RuntimeException("Empty");
    Object answer = cursor.getData();
    if (cursor.getNext() == cursor) {
        cursor = null;
        return answer;
    }
    swapData(cursor, cursor.getNext());
    cursor.setNext(cursor.getNext().getNext());
    return answer;
}

```

Problem 10 Write Java interfaces for the following ADTs. (For each part, just write an interface that specifies methods — do not implement these methods.)

(i) Stack

Answer:

```

interface Stack {
    public Object pop();
    public void push(Object x);
}

```

(ii) Queue

Answer:

```
interface Queue{
    public Object remove();
    public void add(Object x);
}
```

(iii) Deque

Answer:

```
interface Deque{
    public Object removeFirst();
    public Object removeLast();
    public void addFirst(Object x);
    public void addLast(Object x);
}
```

Problem 11 Write a Java implementation for the following problem. You can make use of whichever of the ADTs Stack, Queue, and Deque that you need. Assume that these ADTs are already implemented, compiled and available in files Stack.class, Queue.class, Deque.class.

Input is read from the terminal (*System.in*). All input Strings are read and stored. The input Strings BEGIN, END, and PRINT have special meanings. Whenever a String that matches PRINT is read, all stored strings that appear after the last stored BEGIN string are printed (in the same order that they were entered). These printed strings together with their enclosing BEGIN and PRINT strings are removed from storage. If there is no stored BEGIN string an Exception should be thrown.

When the string END is read, the program terminates.

For example, if the input is:

```
BEGIN is This PRINT BEGIN easy. PRINT END
```

the output would be:

```
is This easy.
```

Or if the input is:

```
BEGIN is This BEGIN easy. PRINT PRINT ignore END
```

the output would be

```
easy. is This
```

Answer:

```
import java.util.*;

public class Prob2 {
    public static void main(String args[]) {
        Stack s = new Stack();
        Scanner i = new Scanner(System.in);
        String w;

        while (i.hasNext()) {
            w = i.next();
            if (w.equals("END")) System.exit(0);
            if (!w.equals("PRINT")) s.push(w);
            else { // process a PRINT instruction
```



```
public static int factorial(int n) {
    if (n <= 0) return 1;
    return n * factorial(n - 1);
}
```

(c)

```
public static double power(double x, int n) {
    double ans = 1.0; int index = 0;
    while (index++ < n) ans = ans * x;
    return ans;
}
```

(d) Which of these methods is inefficient? Give an efficient implementation of this method.

Problem 14 Write a Java implementation for the following problem. You can make use of whichever of the ADTs Stack, Queue, and Deque that you need. Assume that these ADTs are already implemented, compiled and available in files Stack.class, Queue.class, Deque.class.

Input is read from the terminal (*System.in*). All input Strings are read and stored. The input String PRINT has a special meaning. Whenever a String that matches PRINT is read, the first stored string and the last 5 stored strings are printed. If there are not enough strings, the program should terminate as soon as it runs out of strings.

For example, if the input is:

```
1 2 3 4 5 6 7 8 PRINT 9 PRINT 10 11 12 13 PRINT
```

the output would be:

```
1 8 7 6 5 4 2 9 3
```

Problem 15 Suppose that a singly linked list is implemented as a class *LinkedList* that has a single instance variable *head* of class *Node*. No sentinels are used, so that *head.data* is the first object in the list, and the last node is indicated by a *next* link with value *null*.

Write a method of the class called *oddEntries* that removes all entries at even positions of the list. For example, if the list starts as 1, 2, 3, 4, 5 the method would turn it into 1, 3, 5.

Problem 16 The class Queue is to be implemented as a doubly linked list of DNode objects. Each DNode is a standard doubly linked node, with 3 instance variables *Object data*, *DNode next*, and *DNode prev*. As usual, DNode has 3 get methods and 3 set methods (one for each instance variable) and a 3 argument constructor.

A partial implementation of the class Queue follows. The two most important methods have been omitted.

```
class Queue {
    private DNode front, rear; private int size;
    public Queue() {front = rear = null; size = 0;}
    public int size() { return size;}
    public boolean empty() { return size == 0; }
}
```

(a) **Identify the two missing methods.** For each give the name, parameters and return type.

Answer:

```
public Object dequeue()
public void enqueue(Object x)
```

(b) **Give a complete implementation of ONE** of the two missing Queue methods. (You can choose either of them.)

Answer:

```

public Object dequeue() {
    if (empty()) throw new RuntimeException();
    Object answer = front.getData();
    front = front.getNext();
    size--;
    if (size != 0) front.setPrev(null);
    else rear = null;
    return answer;
}
public void enqueue(Object x) {
    DNode newRear = new DNode(x, rear, null); // data, prev, next
    if (rear != null) rear.setNext(newRear);
    else front = newRear;
    rear = newRear;
    size++;
}
}

```

Problem 17 Write a Java program to solve the following problem. You can make use of whichever of the ADTs Stack, Queue, and Deque that you need. Assume that these ADTs are already implemented, compiled and available in files Stack.class, Queue.class, Deque.class.

Input is read, one String at a time, from a file *input.txt*. Input Strings are stored as they are read. The input Strings !R and !F have special meanings. Whenever a String that matches !R is read, all stored strings are removed from storage and printed in reverse order of input. Whenever a String that matches !F is read, every second stored string is printed in order of input, and the program terminates. The program also terminates without further output when the input file ends.

For example, if the input is:

Reverse these !R and half these words are printed forward !F but this is lost !R

the output would be:

these Reverse
and these are forward

Answer:

```

import java.util.*;
import java.io.*;

public class Prob2 {
    public static void main(String args[]) {
        try {
            Deque d = new Deque();
            Scanner in = new Scanner(new File("input.txt"));
            String s;

            while (in.hasNext()) {
                s = in.next();
                if (s.equals("!R")) {
                    while (!d.empty()) {
                        s = (String) d.removeRear();
                        System.out.print(s + " ");
                    }
                    System.out.println();
                }
                else if (s.equals("!F")) {
                    while (!d.empty()) {
                        s = (String) d.removeFront();
                        System.out.print(s + " ");
                    }
                }
            }
        }
    }
}

```

```

        if (!d.empty()) d.removeFront();
    }
    System.out.println();
    break;
}
else d.addRear(s);
}
} catch (Exception e) {}
}
}

```

Problem 18 A linked stack is implemented using a standard Node class as follows:

```

import java.util.*;

class stack implements Iterable {
    private Node top;
    private int size;
    public stack() { top = null; size = 0;}
    public Object pop() {
        if (size == 0) throw new RuntimeException("");
        Object answer = top.getData();
        top = top.getNext();
        size--;
        return answer;
    }
    public void push(Object x) {
        Node newNode = new Node(x, top);
        top = newNode;
        size++;
    }

    // the iterator method is missing
}

```

Write a class *StackIterator* to implement objects that can be returned by the stack iterator. **Also write** the missing stack method called *iterator*. You can decide whether the iterator will run through the data in the stack in LIFO or FIFO order (one choice is much easier).

Answer:

(a) The StackIterator class:

```

class StackIterator implements Iterator {
    private Node current;
    public StackIterator(Node c) {
        current = c;
    }
    public Object next() {
        Object answer = current.getData();
        current = current.getNext();
        return answer;
    }
    public boolean hasNext() {
        return current != null;
    }
    public void remove() {}
}

```

(b) The iterator method in the stack class:

```
public Iterator iterator() {  
    return new StackIterator(top);  
}
```