OLLSCOIL NA hÉIREANN
THE NATIONAL UNIVERSITY OF IRELAND, CORK

COLÁISTE NA hOLLSCOILE, CORCAIGH
UNIVERSITY COLLEGE, CORK

Autumn Examinations 2010

CS2504: Algorithms and Linear Data Structures

Dr C. Shankland
Professor J. Bowen
Dr K. T. Herley

Answer all three questions
Total marks: 80

1.5 Hours

Question 1 [*40 marks*] *Answer all eight parts.*

(i) Write a fragment of Java code that writes out the numbers from one to 100 inclusive with ten numbers per line. (*5 marks*)

(ii) Give a series of Java statements that do the following: (a) declare a variable named `stk` to refer to a stack (ADT Stack) of integer values; (b) create a stack (ArrayBasedStack) and make `stk` refer to it; (c) populate the stack by adding the numbers from one to ten inclusive; (d) remove and print the top three numbers from the stack. (*5 marks*)

(iii) Give a complete pseudocode algorithm that takes a list object (ADT List) named `lst` containing integer elements and that deletes all odd numbers from the list. Your algorithm may manipulate `lst` by means of the operations of ADT List only. (*5 marks*)

(iv) Describe succinctly, using words and/or diagrams as appropriate, how an array might be used to represent an object of type ADT Queue. Indicate clearly how operations enqueue and dequeue can be implemented efficiently. (*5 marks*)

(v) Consider the pseudocode shown below of a simplified array-based implementation of ADT Stack. Translate this code into a detailed, compilable Javacode. You may assume, for now, that the stack items are of type Integer, but you must include appropriate instance/class variable declarations, a constructor and implementations of the methods push and pop.

Representation:
S: array of length N = 100
t: integer variable

Algorithm push(o):
t ←t+1
S[t] ←o

Algorithm pop():
e ←S[t]
t ←t−1
return e

(*5 marks*)

(vi) Indicate briefly the modifications required to **Part (v)** to make the implementation generic *i.e.* capable of accommodating stack items of any type. (*5 marks*)

(vii) Given below is a flawed version of a non-recursive version of the well-known binary search algorithm that contains a number of logical errors. Modify the code so that BinarySearch(S, k) returns the index within S that houses k, if k is present and so that it returns -1, if k is not present.

```
Algorithm BinarySearch(S, k)
    low ←0
    high ←S.size()
    while low ≥high do
        mid = (low + high)/2
        midKey = key(mid)
        if k = midKey then
            return mid
        else
        if k > midKey then
            high ←mid − 1
        else
            return −1
```

(5 marks)

(viii) Explain the concept of a *comparator* and the role this concept plays in the implementation of ADTs. *(5 marks)*

Question 2 [*20 marks*]

Give a Java implementation of ADT List. Include both an interface `List.java` and an implementation `ArrayBasedList.java`. For full marks your interface/implementation must

- be as complete as possible (though you may ignore the iterator and listIterator operations completely);
- include implementations for operations size, isEmpty, get, set, add (both variants) and remove;
- be based on the concept of a left-justified array;
- be generic *i.e.* capable of accommodating any element type.

Partial marks will be awarded for sensible, coherent answers that meet some but not all of the requirements listed.

Question 3 [*20 marks*]

(i) Give an algorithm in pseudocode that takes two lists L1 and L2 of elements with elements arranged in increasing order from front to back within the list, and that merges the two lists in to a single list L (in increasing order). *(4 marks)*

(ii) Give a pseudocode implementation of Merge-sort based on the above that takes a list of elements and that rearranges those elements in increasing order. Your algorithm may either be recursive or non-recursive. *(4 marks)*

(iii) Sketch a proof that your Merge-sort algorithm does actually sort the elements of any list given as input. (*3 marks*)

(iv) Analyze the worst-case running time of your Merge algorithm. Assume the running time is captured by the number of comparisons performed. Comment on the efficiency of Merge-sort in relation to other well-known sorting algorithms. (*3 marks*)

(v) Suppose that we have an application that requires the manipulation of a large number of student records. For each student we have his names (first, middle, last), id number, date of birth among other data. To forestall possible exam-time confusion, the Registrar's Office wants a list of groups of students with similar names. Indicate how efficiently to produce a list of names (first name, last name combinations) shared by more than one student and for each such "shared name", generate a list of the details of those students that bear that name.

(*6 marks*)

cs2504 ADT Summary

General Notes

1. All of the "container" ADTs (Stack, Queue, List, Map, Priority Queue and Set) support the following operations.

   size(): Return number of items in the container. *Input:* None; *Output:* int.
   isEmpty(): Return boolean indicating if the container is empty. *Input:* None; *Output:* boolean.

2. The GT and Java Collections formulations make use of exeptions to signal the occurance of an ADT error such as the attempt to pop from an empty stack. Our formulation makes no use of exeptions, but simply aborts program execution when such an error is encountered.

3. See the sheet entitled "ADT Comparison Table" for a more detailed comparison of our ADTs and their GT and Java Collections counterparts.

ADT Stack<E>

A stack is a container capable of holding a number of objects subject to a LIFO (last-in, first-out) discipline. It supports the following operations.

push(*o*): Insert object *o* at top of stack. *Input:* E; *Output:* None.
pop(): Remove and return top object on stack; illegal if stack is empty[1]. *Input:* None; *Output:* E.
top(): Return the object at the top of the stack, but do not remove it; illegal if stack is empty[1]. *Input:* None; *Output:* E.

ADT Queue<E>

A queue is a container capable of holding a number of objects subject to a FIFO (first-in, first-out) discipline. It supports the following operations.

enqueue(*o*): Insert object *o* at rear of queue. *Input:* Object; *Output:* None.
dequeue(): Remove and return object at front of queue; illegal if queue is empty[1]. *Input:* None; *Output:* E.
front(): Return the object at the front of the queue, but do not remove it; illegal if queue is empty[1]. *Input:* None; *Output:* E.

Iterator<E>

An iterator provides the ability to "move forwards" through a collection of items one by one. One can think of a "cursor" that indicates the current position. This cursor is initially positioned before the first item and advances one item for each invocation of operation next.

hasNext(): Return true if there are one or more elements in front of the cursor. *Input:* None; *Output:* boolean.
next(): Return the element immediately in front of the cursor and advance the cursor past this item. Illegal if cursor is at the end of the collection[1]. *Input:* None; *Output:* E.

[1]GT counterpart throws exception.

ListIterator<E>

This ADT extends ADT Iterator and applies to List objects only. A list iterator provides the ability to "move" back and fourth over the elements of a list.

hasPrevious(): Return true if there are one or more elements before the cursor. *Input:* None; *Output:* boolean.
nextIndex(): Return the index of the element that would be returned by a call to next. Illegal if no such item[1]. *Input:* None; *Output:* int.
previous(): Return the element immediately before the cursor and move cursor in front of element. Illegal if no such item[1]. *Input:* None; *Output:* E.
previousIndex(): Return the index of the element that would be returned by a call to previous. Illegal if no such item[1].
Input: None; *Output:* int.
add(o): Add element *o* to the list at the current cursor position, *i.e.* immediately after the current cursor position. *Input:* E; *Output:* None.
set(o): Replace the element most recently returned (by next or previous) with o. *Input:* E; *Output:* None.
remove(): Remove from underlying list the element most recently returned (by next or previous). *Input:* None; *Output:* None.

Note: It is legal to have several iterators over the same list object. However, if one iterator has modified the list (using operation remove, say), all other iterators for that list become invalid. Similarly, if the underlying list is modified (using List operation add, for example), then all iterators defined on that list become invalid.

List<E>

A list is a container capable of holding an ordered arrangement of elements. The *index* of an element is the number of elements that preceed it in the list.

get(inx): Return the element at specified index. Illegal if no such index exists[1]. *Input:* int; *Output:* E.
set(inx, newElt): Replace the element at specified index with newElt. Return the old element at that index. Illegal if no such index exists[1]. *Input:* int, E; *Output:* E.
add(newElt): Add element newElt at the end of the list.[2] *Input:* E; *Output:* None.
add(inx, newElt): Add element newElt to the list at index inx. Illegal if inx is negative or greater than current list size[1]. *Input:* int, E; *Output:* None.
remove(inx): Remove the element at the specified index from the list and return it. Illegal if no such index exists[1]. *Input:* int; *Output:* E.
iterator(): Return an iterator of the elements of this list. *Input:* None; *Output:* Iterator<E>.
listIterator(): Return a list iterator of the elements in this list [2]. *Input:* None; *Output:* ListIterator<E>.

ADT Comparator<E>

A comparator provides a means of performing comparions

[2]No such operation in GT formulation.

between objects of a particular type. It supports the following operation.

compare(a, b): Return an integer i such that $i < 0$ if $a < b$, $i = 0$ if $a = b$ and $i > 0$ if $a > b$. Illegal if a and b cannot be compared[1]. *Input:* E, E; *Output:* int.

ADT Entry<K,V>

An entry encapsulates a *key* and *value*, both of type Object. It supports the following operations.

getKey(): Return the key contained in this entry. *Input:* None; *Output:* K.

getValue(): Return the value contained in this entry. *Input:* None; *Output:* V.

ADT Map<K, V>

A map is a container capable of holding a number of entries. Each entry is a key-value pair. Key values must be distinct. It supports the following operations.

get(k): If map contains an entry with key equal to k, then return the value of that entry, else return null. *Input:* K; *Output:* V.

put(k, v): If the map does not have an entry with key equal to k, add entry (k, e) and return null, else, replace with v the existing value of the entry and return its old value. *Input:* K, V; *Output:* V.

remove(k): Remove from the map the entry with key equal to k and return its value; if there is no such entry, return null. *Input:* K; *Output:* V.

iterator(): Return an iterator of the entries stored in the map[3]. *Input:* None; *Output:* Iterator<Entry<K, V>>.

ADT Position<E>

A position represents a "place" within a tree (*i.e.* a node); it contains an *element* (of type E) and supports the following operation.

element(): Return the element stored at this position. *Input:* None; *Output:* E.

ADT Tree<E>

A tree is a container capable of holding a number of positions (nodes) on which a parent-child relationship is defined. It supports the following operations.

root(): Return the root of T; illegal if T empty[1]. *Input:* None; *Output:* Position<E>.

parent(v): Return the parent of node v; illegal if v is root[1]. *Input:* Position<E>; *Output:* Position<E>.

children(v): Return an iterator of the children of node v. *Input:* Position<E>; *Output:* Iterator<Position<E>>.

isInternal(v): Return boolean indicating if node v is internal. *Input:* Position<E>; *Output:* boolean.

isExternal(v): Return boolean indicating if node v is a leaf. *Input:* Position<E>; *Output:* boolean.

isRoot(v): Return boolean indicating if node v is the root. *Input:* Position<E>; *Output:* boolean.

iterator(): Return an iterator of the positions(nodes) of T[3]. *Input:* None; *Output:* Iterator<Position<E>>.

replace(v, e): Replace the element stored at node v with e and return the old element. *Input:* Position<E>, E; *Output:* E.

ADT Binary Tree<E>

A binary tree is an extension of a tree in which each node has has at most two children. Objects of type ADT Binary Tree support the operations of the latter type plus the following additional operations.

left(v): Return the left child of v; illegal if v has no left child[1]. *Input:* Position<E>; *Output:* Position<E>.

right(v): Return the right child of v; illegal if v has no right child[1]. *Input:* Position<E>; *Output:* Position<E>.

hasLeft(v): Return true if v has a left child, false otherwise. *Input:* Position<E>; *Output:* boolean.

hasRight(v): Return true if v has a right child, false otherwise. *Input:* Position<E>; *Output:* boolean.

ADT Priority Queue<K,V>

A priority queue is a container capable of holding a number of entries. Each entry is a key-value pair; keys need not be distinct. It supports the following operations.

insert(k, e): Insert a new entry with key k and value e into the priority queue and return the new entry. *Input:* K, V; *Output:* Entry.

min(): Return, but do not remove, an entry in the priority queue with the smallest key. Illegal if priority queue is empty[1]. *Input:* None; *Output:* Entry.

removeMin(): Remove and return an entry in the priority queue with the smallest key. Illegal if priority queue is empty[1]. *Input:* None; *Output:* Entry.

Set<E>

add(newElement): Add the specified element to this set if it is not already present. If this set already contains the specified element, the call leaves this set unchanged *Input:* E; *Output:* None.

contains(checkElement): Return true if this set contains the specified element i.e. if checkElement is a member of this set. *Input:* E; *Output:* boolean.

remove(remElement): Remove the specified element from this set if it is present. *Input:* E; *Output:* None.

addAll(addSet): Add all of the elements in the set addSet to this set if the are not already present. The addAll operation effectively modifies this set so that its new value is the union of the two sets. *Input:* Set<E>; *Output:* None.

containsAll(checkSet): Return true if this set contains all of the elements of the specified set i.e. returns true if checkSet is a subset of this set. *Input:* Set<E>; *Output:* boolean.

removeAll(remSet): Remove from this set all of its elements that are contained in the specified set. This operation effectively modifies this set so that its new value is the asymmetric set difference of the two sets. *Input:* Set<E>; *Output:* None.

retainAll(retSet): Retain only the elements in this set that are contained in the specified set. This operation effectively modifies this set so that its new value is the intersection of the two sets. *Input:* Set<E>; *Output:* None.

iterator(): Return an iterator of the elements in this set. The elements are returned in no particular order. *Input:* None; *Output:* Iterator<E>.

[3]Operation differs from counterpart in GT formulation.