OLLSCOIL NA hÉIREANN
THE NATIONAL UNIVERSITY OF IRELAND, CORK

COLÁISTE NA hOLLSCOILE, CORCAIGH
UNIVERSITY COLLEGE, CORK

Summer Examinations 2012

CS2504: Algorithms and Linear Data Structures

Professor I. P. Gent
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) Give appropriate Java statements to declare and create a queue object of Integer items (using Queue/ArrayBasedQueue), to add the values one to six inclusive to the queue and finally to remove and print one item at a time until the queue is empty. (*5 marks*)

(ii) Sketch, using diagrams or words as appropriate, how ADT Stack may be represented by means of a linked list and indicate how operations push and pop may be carried out. (*5 marks*)

(iii) Sketch how ADT Stack may be represented by means of a left-justified array and give pseudo-code for operations push and pop. (*5 marks*)

(iv) In relation to the following recursive algorithm

```
Algorithm h(n, x, y, z):
  if n > 0 then
    return h(n−1, x, z, y) + h(n−1, z, y, x)
  else
    return 1
```

give a complete recursion tree for the computation $h(3, 1, 3, 2)$ and calculate the value returned. (*5 marks*)

(v) Give a pseudo-code algorithm that takes a list of numbers (List<Integer>) and removes any duplicate values. In other words the first occurrence of each distinct value should be retained, any subsequent re-occurrences should be removed. (*5 marks*)

(vi) Give an expression in terms of the value of j for the worst-case number of comparisons involved in a single execution of lines $(5) - (8)$ in the algorithm given below. Also give an expression for the total worst-case number of comparisons involved in the execution of the whole algorithm, expressed in terms of n, the length of the array A.

```
1   Algorithm X(A, n):
2     j ← 1
3     while j < n do
4       k ← A[j]
5       i ← j − 1
6       while i ≥ 0 and A[i] < k do
7         A[i+1] ← A[i]
8         i ← i − 1
9
10      A[i+1] ← k
11      j ← j + 1
```

(*5 marks*)

(vii) Give pseudo-code for a binary search algorithm (recursive or non-recursive) that takes an array $A[0..n]$ of integer values in non-decreasing order together with a search key k and that returns true if A contains k and false otherwise. (*5 marks*)

(viii) Suppose that X is a map (ADT Map) with Integer keys and values that are lists of strings (List<String>). Write a Java fragment that outputs the contents of the map one entry per line in the following format: first the key, then a colon, then the

contents of the value as a comma-separated list of strings each enclosed in quotes. (*5 marks*)

Question 2 [*20 marks*] Give an interface and implementation of ADT Map based on the doubly-linked list representation that fully supports all the features of the ADT apart from the method named iterator.

Your implementation must be capable of handling keys of any (comparable) type and values of any type and should make appropriate use of comparators. State any assumptions you make about ancillary classes *etc.*

Question 3 [*20 marks*]

In a bid to save money, *University College Kroc (UCK)* has decided to implement its own in-house, simplified CAO-like system to handle admissions.

Each course has a distinct alphanumeric code (KC401 etc.) and a maximum number of applicants (quota) that can be accommodated on that course. Each applicant has a name, a unique id number, a Leaving Cert point score and three courses listed in order of preference (1 highest, 3 lowest).

Places are awarded on courses purely on the basis of applicants' points; there are no minimum points requirements, no special subject requirements, no sub-quotas for mature students and only a single round of offers. A place may not be offered to a candidate on a course unless all the applicants with higher points are also accommodated either on that course or another for which they expressed a higher preference. Subject to this principle, where a number of applicants having equal points compete for a smaller number of places on a course, those places may be awarded among those applicants by lots (*i.e.* at random).

Sketch an application that takes a list of courses and applicants and that allocates the available places on the courses consistent with the above requirements. The application should output a list of applicants and the courses (at most one) offered to each and for each course a list (in descending order of points) of the applicants to offered places on that course.

You are not expected to write a complete Java program, but your description, in words, diagrams or pseudo-code as appropriate should indicate the main algorithmic concepts and data structures that such a program might embody.

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 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.

PLEASE DO NOT TURN THIS PAGE
UNTIL INSTRUCTED TO DO SO

THEN ENSURE THAT YOU HAVE THE
CORRECT EXAM PAPER