OLLSCOIL NA hÉIREANN
THE NATIONAL UNIVERSITY OF IRELAND, CORK

COLÁISTE NA hOLLSCOILE, CORCAIGH
UNIVERSITY COLLEGE, CORK

SUMMER EXAMINATIONS 2011

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 pseudo-code algorithm that reads a sequence of non-negative integer values and that prints out all of the odd numbers in the sequence first and then all of the even ones. The appearance of a negative number signifies the end of the sequence. Your algorithm may use any of the ADTs listed in the ADT Summary appended to the end of this paper, but no other data structures. Use a method named nextInt to read and return integer values. *(5 marks)*

(ii) Describe, in words or diagrams as appropriate, how ADT Queue may be represented efficiently by means of a "circular array" concept. *(5 marks)*

(iii) Give pseudo-code algorithms for Queue operations enqueue, dequeue and size in terms of the queue representation described above. *(5 marks)*

(iv) Modify the binary search algorithm shown below to produce an algorithm named IsPresent that searches in an array A arranged in *decreasing order* left to right and that returns the boolean value true if the search key k is present in the array and false otherwise.

```
Algorithm BinarySearch(A, k)
  low ←0
  high ←A.length − 1
  while low ≤high do
    mid = (low + high)/2
    midKey = A[mid]
    if k = midKey then
      return mid
    else
    if k < midKey then
      high ←mid −1
    else
      low ←mid+1
  return −1
```

(5 marks)

(v) Describe how ADT Map may be represented by means of a doubly-linked list and illustrate your answer by means of a detailed sketch showing both an empty map and a non-empty map of size three. *(5 marks)*

(vi) Describe what is meant by the worst-case running time of an algorithm and explain how it relates to the efficiency of the algorithm. For the sorting problem, name an efficient algorithm and quote its worst-case running time. *(5 marks)*

(vii) Suppose that we wish to eliminate all duplicate values from a large array A of integer quantities. Describe briefly an efficient approach to completing this task, naming any ADTs or algorithms you rely on and justifying the efficiency of your approach. *(5 marks)*

(viii) Indicate how many calls to method X occur during the execution of X(5, 5). Justify your reasoning.

```
Algorithm X(k, n)
  if k > 0 then
    move(k−1, k∗n)
    move(k−1, 2∗n)
```

(5 marks)

Question 2 [*20 marks*] ADT Priority Queue is a container abstraction. A priority queue holds *entries*: each entry consists of (i) a key (also referred to as a priority) and (ii) a value. The keys can be of any "comparable" type on which an order $x \leq y$ may be defined and keys need not be distinct; the values can be of any type. The ADT supports the following operations. as well as the standard size and isEmpty operations.

- insert(k, e): Insert a new entry with key k and value e into the priority queue and return the new entry. Input: Key type, Value type; Output: Entry.
- min(): Return, but do not remove, an entry in the priority queue with the smallest key. Illegal if priority queue is empty. Input: None; Output: Entry.
- removeMin(): Remove and return an entry in the priority queue with the smallest key. Illegal if priority queue is empty. Input: None; Output: Entry.

Using either a representation based on the concept of a straightforward left-justified array or one based on a doubly-linked list, give a complete Java implementation for this ADT. State any assumptions you make about any ancillary classes you employ. For full marks your implementation must be able to accommodate any key type and any value type.

Question 3 [*20 marks*]

Suppose that we wish to develop a computer application to analyze the frequency of occurrence of words in a collection $\mathcal{C}$ of English-language documents. The objective is to identify both the 1000 most common words (those with the greatest number of occurrences across the complete collection) and the least common words (those occurring in fewer than 5% of the documents). The common words are to be listed in decreasing order of frequency. Each of the uncommon words is to be listed togther with a list of the names of the documents in which it appears.

Sketch an approach to this problem. It is not expected that you provide a complete program, but your sketch, using diagrams, pseudo-code or prose as appropriate, should address the main conceptual issues involved so that a competent Java programmer could complete the program based on your blueprint without any significant conceptual insight or ingenuity on his part.

Describe how the individual documents are processed, how the common and uncommon words are identified and how the required output is produced. Describe carefully the main data structures you use.

Assume that $\mathcal{C}$ is presented in the form of a list of filenames (List<String>), that the files are plain text files and that file names are distinct. Ignore the distinction between upper and lower-case and disregard all non-words (numbers, punctuation *etc.*) appearing in the documents. Treat words sharing a common stem, such as singular and plural forms of nouns, as distinct words. State any other assumptions that you rely upon.

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 non-leaf 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.