OLLSCOIL NA hÉIREANN
THE NATIONAL UNIVERSITY OF IRELAND, CORK

COLÁISTE NA hOLLSCOILE, CORCAIGH
UNIVERSITY COLLEGE, CORK

Autumn 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) What output does the following code produce? Assume that s, q and m are instances of ADT Stack, Queue and Map, respectively and that all are initially empty.

```
for i ←0 to 5 do
    s.push(2*i)
    q.enqueue(2*i + 1)
    m.put(2*i, 0)

while not s.isEmpty() do
    q.enqueue(p.pop())

while not q.isEmpty() do
    t ←q.dequeue()
    if m.get(t) ≠ null then
        print(t)
```

(*5 marks*)

(ii) Explain what is meant by the term *left-justified array* and describe how this concept may be used to represent ADT Stack. (*5 marks*)

(iii) In terms of a left-justified representation of ADT Stack, give algorithms in pseudo-code for operations push and pop. (*5 marks*)

(iv) Describe, in words, diagrams or pseudo-code as appropriate, a suitable representation for ADT Map. (*5 marks*)

(v) In terms of your chosen representation for ADT Map, give a complete pseudo-code algorithm for operation remove. (*5 marks*)

(vi) Explain what a comparator is and why it is useful in the context of the Java implementation of ADT Map. (*5 marks*)

(vii) Compare and contrast the linear search and binary search algorithms and comment on their respective worst-case running times. (*5 marks*)

(viii) Describe succinctly what a merging algorithm does when applied to two lists (ADT List). Give a complete pseudo-code implementation for the MergeSort sorting algorithm. (*5 marks*)

Question 2 [*20 marks*] ADT Deque is a queue-like ADT that supports insertions and deletions at both the front and the rear of the "queue". It supports the following main operations:

- insertFirst(e): Insert a new element *e* at the beginning of the deque. Input: EltType; Output: None.
- insertLast(e): Insert a new element *e* at the end of the deque. Input: EltType; Output: None.
- removeFirst(): Remove and return the element at the beginning of the deque. Illegal if the deque is empty. Input: None; Output: EltType.
- removeLast(): Remove and return the element at the end of the deque. Illegal if the deque is empty. Input: None; Output: EltType.
- size(): Return the number of elements in the deque. Input: None; Output: int.
- isEmpty(): Return true if the dequeue is empty and false otherwise Input: None; Output: boolean.

Give a suitable Java interface for ADT Deque. (*4 marks*)

Give suitable Java implementation of this ADT. For full marks your implementation must

- be based on a linked-list representation representation of the ADT;
- be capable of handing any types of objects as elements;
- include suitable Java code for each of the following
  - instance variables (*4 marks*)
  - a constructor (*4 marks*)
  - implementations of operations insertFirst and insertLast (*4 marks*)
  - implementation of operations removeFirst and removeLast. (*4 marks*)

Question 3 [*20 marks*]

Suppose we wish to write an application to analyze babies' names with a view to determining the most common choices. Each baby is represented by a `Baby` object that has the following members: `firstName`, `lastName`, `ppsNumber` and `sex` (all of type String), together with getters and setters for these. (A baby's sex is encoded as one of the strings "male" or "female").

(i) Give a Java fragment that takes an object babyList of type List<Baby> and that writes out the names (first name and last name) of each baby in the list. (*4 marks*)

(ii) Explain carefully how ADT Map might be used to record how many babies bear a particular name (first name). (*4 marks*)

(iii) Write Java algorithm that analyzes the contents of babyList and prints out whichever name (first name) is the most common (ignoring the possibility of ties). (*8 marks*)

(iv) Describe briefly how to determine the *ten* most common boys' names. (*4 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 exceptions 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>.

Page 4 of 5

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 of 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.