OLLSCOIL NA hÉIREANN
THE NATIONAL UNIVERSITY OF IRELAND, CORK

COLÁISTE NA hOLLSCOILE, CORCAIGH
UNIVERSITY COLLEGE, CORK

SUMMER EXAMINATIONS 2014

CS4092: Special Topics in Computing I
(Algorithms and Linear Data Structures)

Professor Ian Gent
Professor B. O'Sullivan
Dr K. T. Herley

Answer all questions
Total marks available: 100%

1.5 Hours
About 1.11 percent per minute

PLEASE DO NOT TURN THIS PAGE UNTIL INSTRUCTED TO DO SO

ENSURE THAT YOU HAVE THE CORRECT EXAM PAPER

Question 1 [*40%*]

(i) Write a fragment of Java using Map/ArrayBasedMap that declares and creates a map object named nums, populates the map with the numbers from 1 to 100 inclusive (as keys) and then removes all those numbers divisible by 17 (i.e. $17, 34, 51, \cdots$). (*8%*)

(ii) Give a pseudocode fragment that takes the contents of a queue (ADT Queue) and reverses the order of the items contained within it. You may make use of an additional ADT from {Stack, Queue, List Map}, if you wish. (*8%*)

(iii) The incomplete non-recursive binary search shown is intended to return the index within array S that contains the search key k (or "index" -1 if the search key is not present). Complete the algorithm by providing appropriate pseudocode for the placeholders labelled $\boxed{\alpha}$ etc. (*8%*)

```
Algorithm BinarySearch(S, k):
  low ←0
  high ←S.size() − 1
  while [α] do
    mid = [β]
    midKey = S[mid]
    if k = midKey then
      return [γ]
    else
    if k < midKey then
      high ←mid −1
    else
      [δ]
  return −1
```

(iv) Consider an implementation of ADT List using a left-justified array representation based on the following instance variable declarations, where EltType stands for the element type of the items in the list.

```
private EltType elements[];
private int numElts;
```

Give a complete implementation for the operation `remove`. (*8%*)

(v) State the number of comparisons completed during the execution of the following algorithm when applied to an array X of length n. Justify your reasoning carefully. (*8%*)

```
Algorithm BS(X, n):
  s ←1
  while s < n do
    curr = 0
    while curr < n−1 do
      next = curr + 1
      currElt = X[curr]
      nextElt = X[next]
      if currElt > nextElt then
        X[curr] = nextElt
        X[next] = currElt
      curr ←curr + 1
    s ←s +1
```

Question 2 [*30%*] Give a complete Java implementation from scratch for an enhanced version of the traditional queue ADT that includes all the usual ADT operations as well as the following:

> **delete(val):** Remove from the queue all elements that equal the specified value; return the number of deletions made. *Input: Eltype*; *Output: int.*

Your implementation must respect the following conditions:

1. it must be based on the concept of a *doubly-linked list* and
2. it must be capable of housing elements of any (comparable) data type.

You do *not* need to provide code for an interface nor for a node class.

Question 3 [*30%*]

(i) The following recursive algorithm segregates the contents of $A[f..r]$ (the segment of array A between indices f and r inclusive) so that all the values less than or equal to x appear to the left of those greater than x.

```
Algorithm Split(A, x, f, r):
  if r < f then
    return
  else
  if A[f] ≤x then
    Split(A, x, f+1, r)
  else
  if A[r] > x then
    Split(A, x, f, r−1)
  else
    temp = A[f]
    A[f] = A[r]
    A[r]= temp
    Split(A, x, f+1, r−1)
```

Draw a recursion tree to show the execution of Split(X, 5, 0, 7) where the array contains $[3, 8, 4, 1, 6, 5, 2, 7]$ initially. Show the state of the array and the values of f and r at each stage. (*6%*)

(ii) Argue that for any array A of length n, the number of calls to Split arising from Split(A, x, 0, n−1) for any x is at most $n + 1$. (*6%*)

(iii) Show how the algorithm may be modified to produce a variant Split2 so that Split2(A, x, f, r) not only partitions the elements in the manner of Split but also returns the number of values in $A[f..r]$ that are less than or equal to x. (*9%*)

(iv) Based on the Split2 algorithm, write a recursive sorting algorithm (in pseudocode) that takes an interval of an array $A[f..r]$ and that re-arranges the contents so that they appear in increasing order left to right. (*9%*)

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 exceptions, 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>.

[2]No such operation in GT formulation.

ADT Comparator<E>

A comparator provides a means of performing comparions 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.

PLEASE DO NOT TURN THIS PAGE UNTIL INSTRUCTED TO DO SO

THEN ENSURE THAT YOU HAVE THE CORRECT EXAM PAPER