University of Glasgow

Tuesday 1 May 2018
9.30 am – 11.00 am
(Duration: 1 hour 30 minutes)

DEGREES OF MSci, MEng, BEng, BSc, MA and MA (Social Sciences)

Algorithms and Data Structures 2

(Answer all 40 questions.)

- This examination paper is worth a total of **60 marks**
- This is a **multiple-choice** paper with 4 choices per question
- Wrong answers will incur **a negative mark of -1/3**
- The exam consists of 4 main question topics 1. to 4. with questions labeled (a), (b),...
- Every multiple-choice question has an identifier **Qxx** (Q01, Q02, ...) which corresponds to the identifier on the answer sheet
- Instructions for the answer sheet:
  - Fill in your student number and your name in block letters.
  - Shade the bubble corresponding to your answer
  - If you don't know the answer, do not shade any bubble
  - If you made a mistake, cross out the wrong answer

The use of a calculator is not permitted in this examination

INSTRUCTIONS TO INVIGILATORS

Please collect all exam question papers and exam answer scripts and retain for school to collect. Candidates must not remove exam question papers.

1. A binary search tree is a binary tree T such that each node of T stores an item e. Items stored in the left subtree of T rooted at a node v are less than the item in node v, and items stored in the right subtree of T rooted at a node v are greater than the item in node v. Below is java code for the `BNode` class, and below that code for the `BSTree` class which together implement a binary search tree suitable for storing a set of integers. **[16]**

```
public class BNode {
    private BNode left;
    private int   item;
    private BNode right;

    public BNode(int e){left = null; item = e; right =
null;}

    public int   getItem(){return item;}
    public BNode getLeft(){return left;}
    public BNode getRight(){return right;}
    public void  setLeft(BNode nd){left = nd;}
    public void  setRight(BNode nd){right = nd;}

    public String toString(){
       String s = "(";
       if (left != null) s = s + left.toString();
       s = s +","+ item +",";
       if (right != null) s = s + right.toString();
       s = s +")";
       return s;
    }

}

public class BSTree {
    private BNode root;
    private int size;

    public BSTree(){root = null;}

    public BNode root(){return root;}
    public boolean isEmpty(){return root == null;}
    public int size(){return size;}

    public void insert(int e){
    // ...
    }

    private void insert(int e,BNode nd){
    // ...
    }

    public boolean isPresent(int e){
    // ...
    }

    private boolean isPresent(int e,BNode nd){
    // ...
    }
}
```

(a) What is the correct Java code for the public method `insert(int e)` in class `BSTree`, where the method inserts the integer `e` into the tree rooted at node `nd`. **[2]**

Q01

(A)
```
public void insert(int e){
  if (!isEmpty()) {
        root = new BNode(e);
    } else {
        insert(e,root);
    }
}
```
(B)
```
public void insert(int e){
  if (isEmpty()) {
        insert(e,root);
    } else {
        root = new BNode(e);
    }
}
```
(C)
```
public void insert(int e){
  if (isEmpty()) {
        root = new BNode(e);
    } else {
        insert(e,root);
    }
}
```
(D)
```
public void insert(int e){
  if (isEmpty()) {
        root = new BNode(e);
        insert(e,root);
    }
}
```

(b) What is the correct Java code for the corresponding private method `insert(int e,BNode nd)` in class `BSTree`, where the method inserts the integer `e` into the tree rooted at node `nd`. **[2]**

Q02

```
(A)
    private void insert(int e,BNode nd){
        if (e < nd.getItem() && nd.getLeft() == null) {
            nd.setLeft(new BNode(e));
            size++;
        } else if (e < nd.getItem()) {
            insert(e,nd.getLeft());
        } else if (e > nd.getItem() && nd.getRight() ==
null) {
            nd.setRight(new BNode(e));
            size++;
        } else if (e > nd.getItem()) {
            insert(e,nd.getRight());
        }
    }
(B)
    private void insert(int e,BNode nd){
        if (e < nd.getItem() && nd.getLeft() == null) {
            nd.setLeft(new BNode(e));
            size++;
        } else if (e > nd.getItem()) {
            insert(e,nd.getLeft());
        } else if (e < nd.getItem() && nd.getRight() ==
null) {
            nd.setRight(new BNode(e));
            size++;
        } else if (e < nd.getItem()) {
            insert(e,nd.getRight());
        }
    }
(C)
    private void insert(int e,BNode nd){
        if (e < nd.getItem() && nd.getLeft() == null) {
            nd.setLeft(new BNode(e));
            size++;
        } else if (e < nd.getItem()) {
            insert(e,nd.getLeft());
        } else if (e > nd.getItem() && nd.getRight() ==
null) {
            nd.setRight(new BNode(e));
            size++;
        } else {
            insert(e,nd.getRight());
        }
    }
(D)
    private void insert(int e,BNode nd){
        if (e < nd.getItem() && nd.getLeft() == null) {
            nd.setRight(new BNode(e));
            size++;
        } else if (e < nd.getItem()) {
            insert(e,nd.getLeft());
        } else if (e > nd.getItem() && nd.getRight() ==
null) {
            nd.setLeft(new BNode(e));
            size++;
        } else if (e > nd.getItem()) {
            insert(e,nd.getRight());
        }
    }
```

(c) What is the correct Java code for the public method `isPresent(int e,Bnode nd)`, where the method delivers true if and only if `e` is in the tree rooted at node `nd`. **[2]**

Q03

```
(A)
    public boolean isPresent(int e){return root != null &&
!isPresent(e,root);}
(B)
    public boolean isPresent(int e){return root != null &&
isPresent(e,root);}
(C)
    public boolean isPresent(int e){return root == null &&
isPresent(e,root);}
(D)
    public boolean isPresent(int e){return root == null ||
!isPresent(e,root);}
```

(d) What is the correct Java code for the private method `isPresent(int e,Bnode nd)`, where the method delivers true if and only if `e` is in the tree rooted at node `nd`. **[2]**

Q04

```
(A)
    private boolean isPresent(int e,BNode nd){
   return nd != null &&
       (e == nd.getItem() ||
       (e < nd.getItem() && isPresent(e,nd.getRight()) ||
       (e > nd.getItem() && isPresent(e,nd.getLeft()))));
    }
(B)
    private boolean isPresent(int e,BNode nd){
   return nd != null &&
       (e != nd.getItem() ||
       (e > nd.getItem() && isPresent(e,nd.getLeft()) ||
       (e < nd.getItem() && isPresent(e,nd.getRight()))));
    }
(C)
    private boolean isPresent(int e,BNode nd){
   return nd != null &&
       (e == nd.getItem() ||
       (e < nd.getItem() && isPresent(e,nd.getLeft()) ||
       (e > nd.getItem() && isPresent(e,nd.getRight()))));
    }
(D)
    private boolean isPresent(int e,BNode nd){
   return
       (e == nd.getItem() ||
       (e < nd.getItem() && isPresent(e,nd.getLeft()) ||
       (e > nd.getItem() && isPresent(e,nd.getRight()))));
    }
```

(e) Assume that the following items are inserted into an empty `BSTree` in the following order: 30, 40, 24, 58, 48, 26, 11, 24, 13, 36.

- What is the top node of the tree? **[1]**

Q05

(A) 48

(B) 13

(C) 24

(D) 30

- What is the height of the tree? **[1]**

Q06

(A) 2

(B) 3

(C) 4

(D) 5

- What is the preorder traversal of the tree. **[1]**

Q07

(A) 30,24,26,11,13,40,36,58,48

(B) 30,24,11,13,26,40,36,58,48

(C) 30, 40,36,58,48, 24,11,13,26

(D) 30, 40,36,58,48, 24,26,11,13

- What is the inorder traversals of the tree. **[1]**

Q08

(A) 11,13,24,26,30,36,40,48,58

(B) 58,48,40,36,30,26,24,13,11

(C) 30,24,26,11,13,40,36,58,48

(D) 30, 40,36,58,48, 24,26,11,13

- What is the postorder traversal of the tree. **[1]**

Q09

(A) 13,11,26,24,36,48,58,40,30

(B) 26,13,11,24,48,58,36,40,30

(C) 26,13,11,24,30,48,58,36,40

(D) 48,58,36,40,26,13,11,24,30

(f) How might we modify the `BNode` class such that our binary search tree can represent a multiset? **[3]**

Q10

(A) change the BNode class so the item is a linked list

(B) include an occurrence counter in the BSTree class

(C) make the BSTree root a list of BNode

(D) include an occurrence counter in the BNode class

2. Given a 2-D grid of NxN unsigned integers. We will create a datastructure to hold these values. Each pointer is stored as an unsigned integer as well. You can store up to 8 values in scalar variables (i.e. outside of the datastructure). **[20]**

(a) Assuming the NxN grid is constructed as a double-linked list containing N double-linked lists of size N.

- What is the required storage size for the pointers? **[2]**

Q11
(A) N*N*3
(B) N*(N+1)*3
(C) (N*N+1)*3
(D) N*N

- Given a point at (k,k) on this grid, how man steps are required to calculate the sum of the point and the four neighbours of that point? In pseudo code the operation is:

```
grid(k,k)= grid(k,k)+ grid(k+1,k)+ grid(k-1,k)+ grid(k,k+1)+ grid(k,k-1)[2]
```

Q12

(A) 10*k
(B) 4*k
(C) 6*k
(D) 4*k+2

- What is the complexity of this lookup? **[2]**

Q13
(A) O(1)
(B) O(N)
(C) O(N*N)
(D) O(log N)

(b) If the list was single-linked,

- what would the corresponding size be? **[2]**

Q14
(A) N*N*2
(B) N*(N+1)*2
(C) (N*N+1)*2
(D) N*N

- what would the corresponding number of steps be **[2]**

Q15
(A) 10*k
(B) 4*k
(C) 6*k
(D) 4*k+2

- What is the complexity of this lookup? **[2]**

Q16
(A) O(1)
(B) O(N)
(C) O(N*N)
(D) O(log N)

(c) Now if every element had 2 pointers, one pointing up and one left,

- what is the storage space? **[2]**

Q17
(A) N*N*3
(B) N*(N+1)*3
(C) (N*N+1)*3
(D) N*N

- What would the corresponding number of steps be? **[2]**

Q18
(A) 6*k
(B) 4*k+2
(C) 2*k+4
(D) 2*k

- What is the complexity of this lookup? **[2]**

Q19
(A) O(1)
(B) O(N)
(C) O(N*N)
(D) O(log N)

(d) If the whole grid would be stored as a single 1-D singly linked list, what would be the complexity of the computation of the sum of the neigbours? **[2]**

Q20
(A) O(1)
(B) O(N)
(C) O(N^2)
(D) O(log N)

3. An organisation has a data set of 1 million customers. The information the organisation holds on customers includes their height in centimetres (cm). The organisation wants to sort that data using height as a key, where height is an integer in the range 100cm to 220cm. This might be done using an insertion sort or a merge sort. **[8]**

(a) What is an insertion sort? **[2]**

Q21

(A)
- iterate through the unsorted list
- compare each element (elt) with the next (nelt), swap if elt>nelt
- repeat this process until the first element ends up at the tail

(B)
- create an empty singly linked list
- iterate through the unsorted list
- for every element elt in the unsorted list, insert it before the first element ielt in the new list for which ielt>elt

(C)
- iterate through the unsorted list
- compare each element (elt) with the next (nelt), swap if elt>nelt
- repeat this process until no swaps are necessary

(D)
- create an empty singly linked list
- iterate through the unsorted list
- for every element elt, insert it after the first element ielt in the new list for which ielt<elt
- repeat until all values are sorted

(b) What is its complexity of insertion sort? **[2]**

Q22

(A) O(N.log N)
(B) O(N)
(C) O(N^2)
(D) O(log N)

(c) What is a merge sort? **[2]**

Q23
(A)
- iterate through the unsorted list
- compare each element (elt) with the next (nelt), swap if elt>nelt
- put the ordered pairs in a new list
- repeat this process until all pairs are sorted
- merge the sorted pairs into a single list
(B)
- iterate through the unsorted list
- compare each element (elt) with the next (nelt), swap if elt>nelt
- put the ordered pairs into a new list
- repeat this process until no swaps are necessary
- merge the resulting list of pairs into a single list
(C)
- create an empty singly linked list
- iterate through the unsorted list
- compare each element (elt) with the next (nelt), swap if elt>nelt
- put the ordered pairs in the new list
- compare each pair with the next and swap if necessary
- repeat until all values are sorted
(D)
- create an empty singly linked list
- iterate through the unsorted list
- compare each element (elt) with the next (nelt), swap if elt>nelt
- put the ordered pairs in the new list
- merge subsequent pairs into an ordered list
- repeat until all values are sorted

(d) What is its complexity of merge sort? **[2]**
Q24
(A) O(log N)
(B) O(N)
(C) O(N^2)
(D) O(N.log N)

4. In Java, we might represent a set of integers using (i) a BitSet, (ii) a HashSet, (iii) a TreeSet or (iv) a LinkedList. **[16]**

(a) What is the complexity of adding an element to the set when it is represented as a BitSet, a HashSet, a TreeSet or a LinkedList, and the reason for it?

- Adding: Bitset Complexity **[1]**

Q25
(A) O(1)
(B) O(n)
(C) O(n^2)
(D) O(log n)

- Adding: Reason for Bitset Complexity **[1]**

Q26
(A) We need to iterate to find the bit to set
(B) Because it is ordered we can use binary search
(C) We only need to set the bit to 1
(D) Bitset operations are always on the first bit

- Adding: HashSet Complexity **[1]**

Q27
(A) Ideally O(1), worst case O(n)
(B) Ideally O(n), worst case O(n^2)
(C) Always O(n)
(D) Always O(log n)

- Adding: Reason for HashSet Complexity **[1]**

Q28
(A) We need to iterate to find the bucket in which to insert
(B) If there are many collisions, iteration will dominate
(C) Hashing is always constant time
(D) We can do a binary search as the set is fixed size

- Adding: TreeSet Complexity **[1]**

Q29
(A) O(1)
(B) O(n)
(C) O(n^2)
(D) O(log n)

- Adding: Reason for TreeSet Complexity **[1]**

Q30
(A) We need to visit the whole tree to find the item to set
(B) In a balanced tree the number of iterations is the height of the tree
(C) We only need to add one leaf node
(D) Insertion operations are always on the root node

- Adding: LinkedList Complexity **[1]**

Q31
(A) O(1)
(B) O(n)
(C) O(n^2)
(D) O(log n)

- Adding: Reason for LinkedList Complexity **[1]**

Q32
(A) We need to iterate to find the item to set
(B) Because it is ordered we can use binary search
(C) We only need to add the item at the front
(D) We only need to add the item at the back

(b) What is the complexity of removing an element from the set, for each of the four possible representations?

- Removal: Bitset Complexity **[1]**

Q33
(A) O(1)
(B) O(n)
(C) O(n^2)
(D) O(log n)

- Removal: Reason for Bitset Complexity **[1]**

Q34
(A) We need to iterate to find the bit to set
(B) Because it is ordered we can use binary search
(C) We only need to set the bit to 0
(D) Bitset operations are always on the first bit

- Removal: HashSet Complexity **[1]**

Q35
(A) Ideally O(1), worst case O(n)
(B) Ideally O(n), worst case O(n^2)
(C) Always O(n)
(D) Always O(log n)

- Removal: Reason for HashSet Complexity **[1]**

Q36
(A) We need to iterate to find the bucket in which to delete
(B) If there are many collisions, iteration will dominate
(C) Hashing is always constant time
(D) Deletion is not constant time because we need to find the bucket and then the collision chain

- Removal: TreeSet Complexity **[1]**

Q37
(A) O(1)
(B) O(n)
(C) O(n^2)
(D) O(log n)

- Removal: Reason for TreeSet Complexity **[1]**

Q38
(A) We need to visit the whole tree to find the item to delete
(B) In a balanced tree the number of iterations is the height of the tree
(C) We only need to delete one leaf node
(D) Deletion operations are always on the root node

- Removal: LinkedList Complexity **[1]**

Q39
(A) O(1)
(B) O(n)
(C) O(n^2)
(D) O(log n)

- Removal: Reason for LinkedList Complexity **[1]**

Q40
(A) We need to iterate to find the item to set
(B) Deletion is just breaking and restoring pointers
(C) We only need to delete the item at the front
(D) We only need to delete the item at the back