University of Glasgow

**Wednesday 26 April 2023**
**14:00–15:30 BST**
**Duration: 1 hour 30 minutes**
**Additional time: 30 minutes**
**Timed exam – fixed start time**

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

# ALGORITHMS AND DATA STRUCTURES 2
# COMPSCI2007

**Answer all 5 questions**

## This examination paper is an open book, online assessment and is worth a total of 60 marks

**1.** Algorithm **F** takes as input an array of positive integers $A[0\,..\,n-1]$. The algorithm is described by the following pseudocode:

```
1:  F(A)
2:    m := n/2        //integer division
3:    for i := 0 to n − 1
4:      c := 0
5:      for j := 0 to n − 1
6:        if A[i] = A[j]
7:          c := c + 1
8:      if c > m
9:        return A[i]
10:   return −1
```

**(a)** Briefly explain what algorithm **F** implements. [2]

> **Solution:** This algorithm finds the *majority element* in array $A$, if it exists (1 point). It returns $-1$ otherwise (1 point). The majority element is the element occurring at least $n/2$ times, with $n$ the total number of elements in $A$.

**(b)** What is the output of $\mathbf{F}(A)$, where $A = [2,1,6,1,1,5,8,1,1]$? Briefly explain why. [2]

> **Solution:** The output is 1 (1 point) because it occurs $5 > 9/2$ times (1 point).

**(c)** What is the running time of $\mathbf{F}(A)$? Briefly explain why by showing the asymptotic number of operations in each line of the pseudocode. Use big-O notation. [3]

> **Solution:** The running time is $O(n^2)$ (1 point). The operations are as follows (2 points)
>
> $$\begin{aligned} T(n) &= O(1) + O(n) + O(n) + O(n^2) + O(n^2) + O(n^2) + O(n) + O(n) + O(1) \\ &= O(n^2) \end{aligned}$$

**(d)** Define algorithm **G**, a more efficient *divide and conquer* version of algorithm **F**. Describe your implementation and briefly explain how the algorithm works. [10]

> **Solution:**
>
> ```
> 1:  G(A, p, r)
> 2:    if p = r        //base case
> 3:      return A[p]
> 4:    q := (p + r)/2
> 5:    left := G(A, p, q)        //recurse on left half
> ```

```
6:    right := G(A, q+1, r)        //recurse on right half
7:    if left = right              //the two halves agree on the majority element
8:       return left
9:    // otherwise, check and return the most frequent of the two
10:   left_count := COUNT(A, left, p, r)
11:   right_count := COUNT(A, right, p, r)
12:   if left_count > right_count
13:      return left
14:   else
15:      return right
```

where **COUNT**$(A, x, p, r)$ returns the number of occurrences of value $x$ in the sub-array $A[p..r]$ in linear time.

1 point for base case (lines 2-3), 1 point for divide step (line 4), 2 points for conquer step (lines 5-6), and 6 points for combine step (lines 7-15).

**(e)** What is the running time of **G**? Explain why by solving its recurrence equation. [3]

**Solution:** The running time of **G** can be described by the following recurrence: $T(n) = 2T(n/2) + 2n = 2T(n/2) + O(n)$ (1 point). This is the same as the recurrence for **MERGE-SORT**, so the solution is $O(n \cdot \log n)$ (2 points).

**2.** **(a)** Is $2^{n+1} = O(2^n)$? Explain why. [2]

**Solution:** Yes (1 point). To show that $2^{n+1} = O(2^n)$, we must find constants $c, n_0 > 0$ such that $0 \leq 2^{n+1} \leq c \cdot 2^n$ for all $n \geq n_0$. Since $2^{n+1} = 2 \cdot 2^n$ for all $n$, we can satisfy the definition with $c = 2$ and $n_0 = 1$. (1 point)

**(b)** Is $2^{2n} = O(2^n)$? Explain why. [2]

**Solution:** No (1 point). To show that $2^{2n} \neq O(2^n)$, assume there exist constants $c, n_0 > 0$ such that $0 \leq 2^{2n} \leq c \cdot 2^n$ for all $n \geq n_0$. Then $2^{2n} = 2^n \cdot 2^n \leq c \cdot 2^n$ which

implies $2^n \leq c$. But no constant is greater than all $2^n$, and so the assumption leads to a contradiction. (1 point)

**(c)** Solve the following recurrence equation $T(n) = 2T(n/4) + \sqrt{n}$. [2]

**Solution:** We can apply the master theorem with $a = 2$, $b = 4$, and $c = 1/2$. Since $\log_4 2 = 1/2$, case 2 applies and $T(n) = \Theta(\sqrt{n} \cdot \log n)$. Other solution methods are possible.

**3.** **(a)** Explain with an example why **SELECTION-SORT** is not stable. [2]

**Solution:** **SELECTION-SORT** was covered in class. The relative order of equal elements is not preserved. Consider input array $A = [3,3,1]$. When sorting $A$, the first 3 (in red) gets swapped to the right of the second 3 as shown by the following sequence of operations.

**Iteration 0** : $A = [1,3,3]$ swap minimum 1 with 3

**Iteration 1** : $A = [1,3,3]$ swap minimum 3 (left-most occurrence) with itself

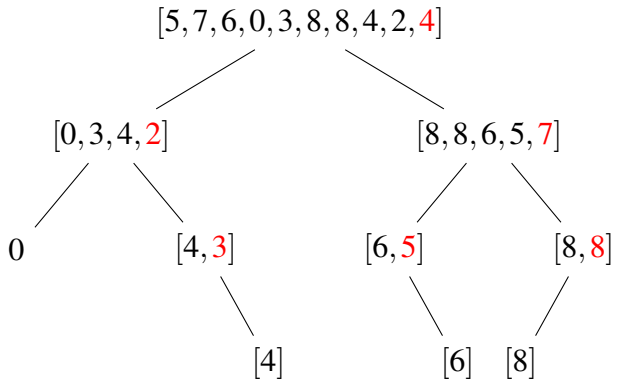
**Iteration 2** : $A = [1,3,3]$ swap minimum 3 with itself

**(b)** What modifications do you need to do to make **SELECTION-SORT** stable? [3]

**Solution:** First, when finding the minimum element, always choose the leftmost entry (1 point). Second, instead of moving the minimum to the front with one swap, move all elements to its left one position to the right, like in **INSERTION-SORT** (2 points). Other solutions are possible.

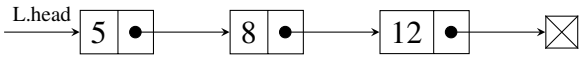
**(c)** Draw the recursion tree computed when sorting array $A = [5,7,6,0,3,8,8,4,2,4]$ with **QUICKSORT**$(A,0,9)$. Assume the right-most element is selected as pivot when partitioning. [5]

**Solution:**




3 points for correct tree structure, 2 points for correct pivot selections.

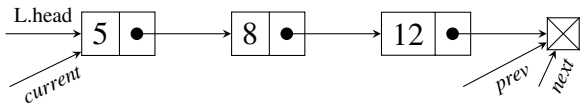
**4.** **(a)** Given the head of a singly linked list, write an algorithm that reverses the list in place, *i.e.* only by modifying the pointer attributes and without using auxiliary data structures. Provide the pseudocode and illustrate how the algorithm operates on the following input:



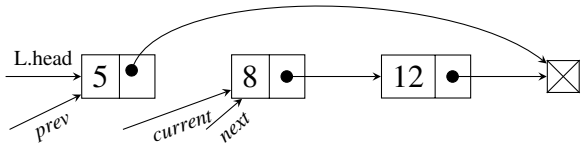

[5]

**Solution:**

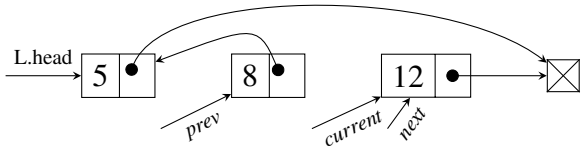
```
1:  REV(L)
2:      current := L.head
3:      prev := NIL
4:      next := NIL
5:      while current ≠ NIL
6:          next := current.next
7:          current.next := prev
8:          prev := current
9:          current := next
10:     L.head := prev
```

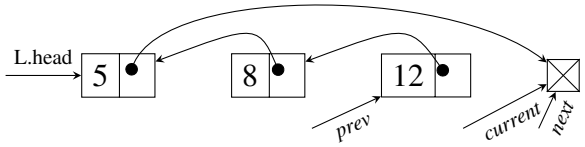


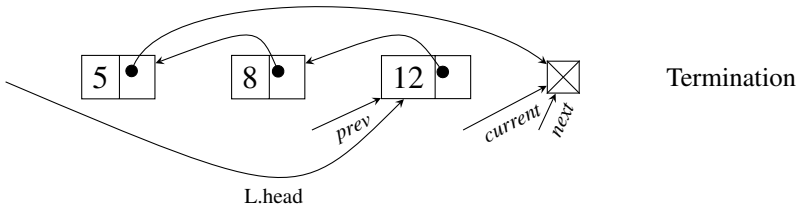

Initialisation




Loop






3 points for the pseudocode, 2 points for correct diagrams.

**(b)** Design a data structure that implements the following operations, all in constant time: PUSH, POP, MIN. Note, no pseudocode is required in the answer. [5]

**Solution:** Maintain two stacks (any implementation with $O(1)$ operations will work), one containing all the items and another containing the minima (2 points). To push an item, push it on the first stack; if it is smaller than the topmost item on the second stack, push it on the second stack as well (1 point). To pop an item, pop it from the first stack; if it is the top item on the second stack, pop it from the second stack as well (1 point). To find the minimum, return the top item on the second stack (1 point). Other solutions are possible.

**5.** **(a)** Give five orderings of the keys 1, 12, 4, 10, 5, 9, 7 that, when inserted into an initially empty BST, produce the best-case tree. [2]

**Solution:** Any sequence that inserts 7 first; 4 before 1 and 5; 10 before 12 and 9. For example:

$$7,10,9,12,4,5,1$$
$$7,10,12,9,4,5,1$$
$$7,10,12,9,4,1,5$$
$$7,4,1,5,10,12,9$$
$$7,4,5,1,10,12,9$$
$$\ldots$$

**(b)** Algorithm **S** takes as input an array of positive integers $A[0\,..\,n-1]$. The algorithm is described by the following pseudocode:

```
1: S(A)
2:     T := EMPTY_BST
3:     for i := 0 to n − 1
4:         INSERT_BST(T, A[i])
5:     INORDER(T)
```

Briefly explain what algorithm **S** implements. Also comment on the worst and best case running times. [4]

**Solution:** Algorithm **S** sorts array $A$ by first building a BST containing $A$'s elements (using **INSERT_BST** repeatedly to insert the integers one by one) and then printing the numbers by an inorder tree traversal (2 points).

In the worst case the running time is $O(n^2)$. This occurs when a linear chain of nodes results from the repeated **INSERT_BST** operations (1 point). We have a $O(n \cdot \log n)$ running time in the best case when the repeated **INSERT_BST** operations yield a balanced tree (1 point).

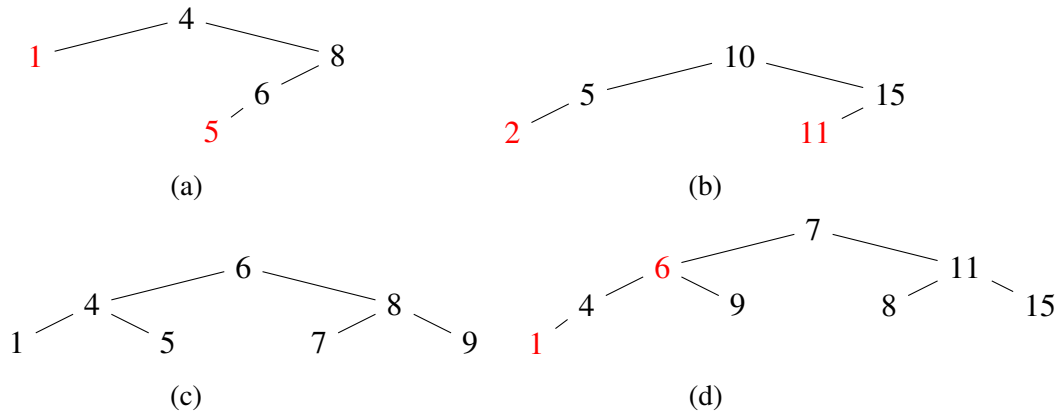
**(c)** Which of the following are valid red-black trees?
Briefly explain your answers. [4]

**Solution:**

(a) Not balanced: black-height 1 in left subtree, 3 in right subtree.

(b) Valid red black BST: black-height 2.

(c) Valid red black BST: black-height 3.




(d) Not balanced (black-height 2 in left subtree, 3 in right subtree) and not a valid BST ($9 > 7$ should be in the right subtree).

1 point for each correct answer.

**(d)** Suppose you want to attack the hashing function for strings defined by the following pseudocode:

```
1: H(S)
2:     hash := 0
3:     for i := 0 to S.length
4:         hash := (hash * 31) + S[i]
5:     return hash
```

where $S[i]$ returns the numerical value (*i.e.* ASCII code, see Appendix A) of the $i$-th character in string $S$. Describe your approach to generate multiple collisions. [4]

**Solution:** We begin by identifying strings of length 2 yielding the same hash code:

| | |
|---|---|
| `Aa` | $65 * 31 + 97 = 2112$ |
| `BB` | $66 * 31 + 66 = 2112$ |
| `C#` | $67 * 31 + 35 = 2112$ |

Then we exploit the fact that any string of length $2N$ that is formed by concatenating these three strings together in any order (*e.g.* `BBC#Aa`, `BBBBBB`, `AaAaAa`, `BBAaC#`, `AaBBBBAa`, ...) will hash to the same value. Other solutions are possible.

# A ASCII table

Dec = Decimal Value
Char = Character

'5' has the int value 53
if we write '5'-'0' it evaluates to 53-48, or the int 5
if we write char c = 'B'+32; then c stores 'b'

| Dec | Char | Dec | Char | Dec | Char | Dec | Char |
|---|---|---|---|---|---|---|---|
| 0 | NUL (null) | 32 | SPACE | 64 | @ | 96 | ` |
| 1 | SOH (start of heading) | 33 | ! | 65 | A | 97 | a |
| 2 | STX (start of text) | 34 | " | 66 | B | 98 | b |
| 3 | ETX (end of text) | 35 | # | 67 | C | 99 | c |
| 4 | EOT (end of transmission) | 36 | $ | 68 | D | 100 | d |
| 5 | ENQ (enquiry) | 37 | % | 69 | E | 101 | e |
| 6 | ACK (acknowledge) | 38 | & | 70 | F | 102 | f |
| 7 | BEL (bell) | 39 | ' | 71 | G | 103 | g |
| 8 | BS (backspace) | 40 | ( | 72 | H | 104 | h |
| 9 | TAB (horizontal tab) | 41 | ) | 73 | I | 105 | i |
| 10 | LF (NL line feed, new line) | 42 | * | 74 | J | 106 | j |
| 11 | VT (vertical tab) | 43 | + | 75 | K | 107 | k |
| 12 | FF (NP form feed, new page) | 44 | , | 76 | L | 108 | l |
| 13 | CR (carriage return) | 45 | - | 77 | M | 109 | m |
| 14 | SO (shift out) | 46 | . | 78 | N | 110 | n |
| 15 | SI (shift in) | 47 | / | 79 | O | 111 | o |
| 16 | DLE (data link escape) | 48 | 0 | 80 | P | 112 | p |
| 17 | DC1 (device control 1) | 49 | 1 | 81 | Q | 113 | q |
| 18 | DC2 (device control 2) | 50 | 2 | 82 | R | 114 | r |
| 19 | DC3 (device control 3) | 51 | 3 | 83 | S | 115 | s |
| 20 | DC4 (device control 4) | 52 | 4 | 84 | T | 116 | t |
| 21 | NAK (negative acknowledge) | 53 | 5 | 85 | U | 117 | u |
| 22 | SYN (synchronous idle) | 54 | 6 | 86 | V | 118 | v |
| 23 | ETB (end of trans. block) | 55 | 7 | 87 | W | 119 | w |
| 24 | CAN (cancel) | 56 | 8 | 88 | X | 120 | x |
| 25 | EM (end of medium) | 57 | 9 | 89 | Y | 121 | y |
| 26 | SUB (substitute) | 58 | : | 90 | Z | 122 | z |
| 27 | ESC (escape) | 59 | ; | 91 | [ | 123 | { |
| 28 | FS (file separator) | 60 | < | 92 | \ | 124 | \| |
| 29 | GS (group separator) | 61 | = | 93 | ] | 125 | } |
| 30 | RS (record separator) | 62 | > | 94 | ^ | 126 | ~ |
| 31 | US (unit separator) | 63 | ? | 95 | _ | 127 | DEL |