UNIVERSITY OF SOUTHAMPTON COMP1009W1

SEMESTER 2 EXAMINATION 2006/2007

DATA STRUCTURES AND ALGORITHMS

Duration: 120 mins

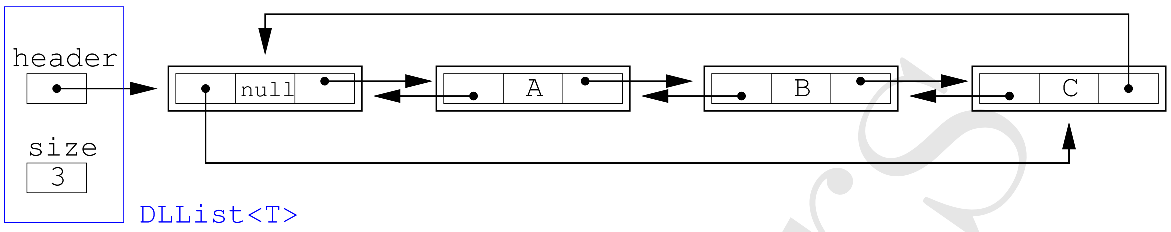
Answer THREE *questions*

This examination is worth 85%. The tutorials were worth 15%.

University approved calculators MAY be used.

Question 1

A doubly linked list consists of nodes that can point in either direction. An efficient implementation uses a dummy node which contains no data. An example of a doubly linked list is shown below.



(a) Write a generic class definition for the doubly linked list described above including the instance variables, a private nested class for the nodes and a default constructor. Do not include any other methods. Approximately correct code is acceptable.

Students have seen doubly linked lists pictorially, but have not been shown any implementation. This tests their understanding and ability to write code.

```
public class DLList<T>
{
    private Node<T> dummy;
    private int cnt;

    private static class Node<T>
    {
        private T element;
        private Node<T> next;
        private Node<T> previous;

        public Node(T o, Node<T> n, Node<T> p) {
            element = o;
            next = n;
            previous =p;
        }
    }

    public DLList()
    {
        dummy = new Node<T>(null, null, null);
        dummy.next = dummy;
        dummy.previous = dummy;
    }
}
```

(10 marks)

(b) Write a `boolean add(T o)` method which adds element `o` to the end of the list (in the example above it would add the element after `C`).

Short but complicated code.

```
boolean add(T o)
{
    dummy.previous = new Node<T>(o, dummy, dummy.previous);
    dummy.previous.previous.next = dummy.previous;
    cnt++;
    return true;
}
```

(5 marks)

(c) Describe the `Iterator<T>` interface of the Java Collect class.

Test knowledge.

The `Iterator<T>` interface specifies three methods.

(i) **`boolean hasNext()` which returns true if there is another element in the collection and false otherwise**

(ii) **`T next()` which returns the next elements and throws an exception if not such element exists**

(iii) **`void remove()` which returns the last element called by `next()` and throws an exception if `next()` hasn't been called.**

(3 marks)

(d) Write an `Iterator<T> iterator()` method which returns a private nested class of type *DLListIterator*

Test knowledge of standard iterator pattern.

```
public Iterator<T> iterator() {return new DDListIterator();}
```

(2 marks)

TURN OVER

(e) Write a private nested class `DDListIterator()` which implements the methods of the `Iterator<T>` interface as well as a constructor method.

This is a hard piece of coding, although they have seen similar code for a singly linked list.

```
private class DDListIterator implements Iterator<T>
{
    private Node<T> current;

    public DDListIterator()
    {
        current = dummy;
    }

    public boolean hasNext()
    {
        return current.next != dummy;
    }

    public T next()
    {
        current = current.next;
        return current.element;
    }

    public void remove()
    {
        if (current==dummy)
            throw new IllegalStateException();
        current.previous.next = current.next;
        current.next.previous = current.previous;
        cnt--;
    }
}
```

(13 marks)

Question 2

(a) What is the time complexity of the following programs (written is pseudo code)

```
prog1(a,b,c)
{
  for (i=1; i<n; i++) {
    for (j=1; j<i; j++) {
       for (k=0; k<100; k++) {
          a(i,j) = b(j)*c(i,k)
       }
    }
  }
}

prog2(a,b,c)
{
  for (i=1; i<n; i++) {
    for (j=1; j<n; j++) {
      if (i-j=1 || i-j=-1)
       for (k=0; k<n; k++) {
          a(i,j) = b(j)*c(i,k)
       }
    }
  }
}
```

Explain your answers.

Tests understanding of time complexity. Answers need to be justified.

`prog1` is $\Theta(n^2)$. The outer loop is order n, the middle loop is also order n even though it only runs up to i. The inner loop is order 1. (In fact, the innermost loop runs $100n(n-1)/2$ times).

`porg2` is also $\Theta(n^2)$. The first loop is order n, the middle loop is also order n, but the inner loop only runs, at most, twice for each time the middle loop runs. Thus the if statement is called n^2 times and the inner statement `a(i,j) = b(j)*c(i,k)` runs $2(n-1)n$ times.

TURN OVER

(6 marks)

(b) If a program takes 1s on an input of size $n = 1000$, how long will take on an input of size $n = 10\,000$ if the time complexity is

(i) logarithmic, i.e. $\Theta(\log(n))$

(ii) log-linear, i.e. $\Theta(n \log(n))$

(iii) quadratic, i.e. $\Theta(n^2)$

(iv) cubic, i.e. $\Theta(n^3)$

Show your working.

Test understanding of time complexity

We denote the time (in seconds) to solve a problem of size n by $T(n)$ (I ignore sub-dominant terms)

(i) Logarithmic implies $T(n) = c\log_{10}(n)$ (I'm free to choose the base of the logarithm). Thus $T(100) = 1 = 3c$ or $c = 1/3$. $T(1000) = \log(1000)/3 = 4/3 \approx 1.33$.

(ii) Log-linear scaling implies $T(n) = cn\log_{10}(n)$ so $c = 1/3000$ and $T(1000) = 40/3 \approx 13.3$

(iii) Quadratic scaling implies $T(n) = cn^2$ so $c = 10^{-6}$ and $T(1000) = 10^{-6} \times 10^8 = 100$

(iv) Cubic scaling implies $T(n) = cn^3$ so $c = 10^{-9}$ and $T(1000) = 10^{-9} \times 10^{12} = 1000$

(8 marks)

(c) Which of the following statements is true?

(i) A $\Theta(n \log(n))$ algorithm will always beat a $\Theta(n)$ algorithm

(ii) An $O(n^2)$ algorithm will always run faster than an $O(n^3)$ algorithm for sufficiently large n

(iii) An $O(n^2)$ algorithm will run faster than an $\Omega(n^2 \log(n))$ algorithm for sufficiently large n

Explain your answer.

Test understanding of big-O and friends. Answers need an explanation.

(i) This statement is wrong because for small n some log-linear algorithms can be faster than linear time algorithms. It all depends on the constant and possibly non-dominant terms.

(ii) This statement is also wrong as big-O is an upper bound. Thus an $O(n^3)$ algorithm may in fact be a $\Theta(n)$ algorithm while the $O(n^2)$ may be a $\Theta(n^2)$ algorithm, in which case the statement would be wrong.

(iii) This statement is correct. The $\Omega(n^2 \log(n))$ algorithm is a lower bound while the $O(n^2)$ algorithm is an upper bound which will be strictly less than the lower bound for large enough n.

(6 marks)

(d) The Fibonacci number, f_n, is generated from the recursion relation $f_n = f_{n-1} + f_{n-2}$ with $f_1 = f_2 = 1$ write an **efficient** program to calculate f_n. What is the time complexity of your code?

Fairly easy test of unwrapping a recursive relation.

Here is some pseudo code to solve this

```
Fibonacci(n)
{
  f = fn = 1
  for (i=3; i≤ n: i++) {
    tmp = f + fn
    f = fn
    fn = tmp
  }
  return fn
}
```

which is a linear time algorithm.

(4 marks)

(e) Consider the program below for computing Fibonacci numbers

```
Fibonacci(n)
{
  if (n<3) return 1;
  return Fibonacci(n-1) + Fibonacci(n-2)
}
```

Let $T(n)$ be the number of additions needed for the above program to compute f_n. Write a recursion relation for $T(n)$ and provide appropriate base cases.

TURN OVER

Simple test of writing a recursion relation for computing the run time of a recursive algorithm.

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) + 1 \\ T(1) = T(2) &= 1 \end{aligned}$$

(4 marks)

(f) Show that $T(n) \in \Theta(\phi^n)$, by substitution, $c\phi^n$ into the recursion formula and ignoring non-leading terms (i.e. terms small compared to ϕ^n). Compute, ϕ. Asymptotically how much longer does it take to compute $T(n+10)$ compared to $T(n)$?

This is very challenging.

Substituting $c\phi^n$ into the recursion formula we get

$$\begin{aligned} c\phi^n &= c\phi^{n-1} + c\phi^{n-2} + 1 \\ &\approx c\phi^{n-1} + c\phi^{n-2} \end{aligned}$$

Or

$$\phi^2 - \phi - 1 = 0$$

Solving the quadratic equation and retaining the physical (positive) solution we get

$$\phi = \frac{\sqrt{5}+1}{2}$$

Thus the time to compute $T(n+10)$ is

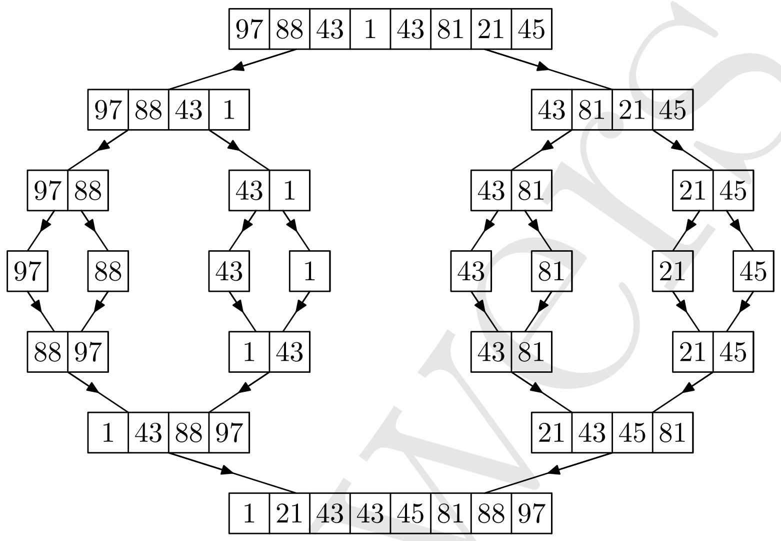
$$T(n+10) \approx c\phi^{n+10} \approx \phi^{10}\, T(n) \approx 123\, T(n).$$

(5 marks)

Question 3

(a) Show how Merge Sort would sort the numbers 97, 88, 43, 1, 43, 81, 21, 45.

Test practical knowledge of Merge Sort



(8 marks)

(b) What does it mean for a sort algorithm to be stable? Why is it desirable for a sort algorithm to be stable? Is Merge Sort stable?

Test theoretical understanding.

There are three elements to this question

- **A stable sort algorithm will keep elements with the same key value in the same order as in the input array.**
- **This is desirable when sorting complex data that may have been sorted on some other key.**
- **Merge short is stable.**

(6 marks)

(c) Derive an expression for the time complexity of Merge Sort.

TURN OVER

Relatively simple complexity analysis.

Merge sort consists of two steps, dividing the array up recursively into arrays of half the size. For an array of length n this can be done $\lceil \log_2(n) \rceil$ times. Each step involves copy all n components to a new array (although this can be avoided). Thus the time complexity of this first step is $\Theta(n \log(n))$. The second step involves merging the arrays into a new array. This involves the same number of outer loops as the first step (i.e.$\lceil \log_2(n) \rceil$). The merging algorithm is a linear time algorithm. This follows because the arrays to be merged are already in order. Thus to construct the new array involves a binary decision, it will be the smallest element left in one of the two arrays being merged. Thus this step is also takes $\Theta(n \log(n))$ steps. Thus the total time complexity is $\Theta(n \log(n))$.

(8 marks)

(d) Explain why the choice of the pivot is crucial to the success of Quick Sort. What problem occurs if we use the first element as the pivot?

Test understanding of Quick Sort algorithm.

Quick Sort uses a divide and conquer algorithm where the division is around the pivot. The efficiency of quick sort depends critically on how even the division is. If the division is approximately even then the number of divisions necessary to reach a trivial sized array is $\Theta(\log_2(n))$. On the other hand, if the division is very uneven the number of divisions necessary to reach a small array can be $\Theta(n)$. This would make Quick Sort a $\Theta(n^2)$ algorithm.

Choosing the first element as the pivot leads to the highly undesirable property that Quick Sort would run as a $\Theta(n^2)$ algorithm for arrays that are already sorted (or in reverse order) as the pivot would divide the array as unevenly as possible.

(6 marks)

(e) What is the worst case time complexity of Merge Sort and Quick Sort? Which sorting algorithm is preferred in practice and why?

Testing understanding of an apparent paradox.

The worst case time complexity of Merge Sort is $\Theta(n \log_2(n))$ while that for Quick Sort is $\Theta(n^2)$. In practice Quick Sort is usually preferred because it is found to be quicker. That is, its average case complexity is lower than Merge Sort. The probability of the worst case complexity occurring is usually so small that people do not worry about it.

(5 marks)

Question 4

(a) Describe Prim's algorithm for computing the minimum spanning tree. Explain how this implemented efficiently.

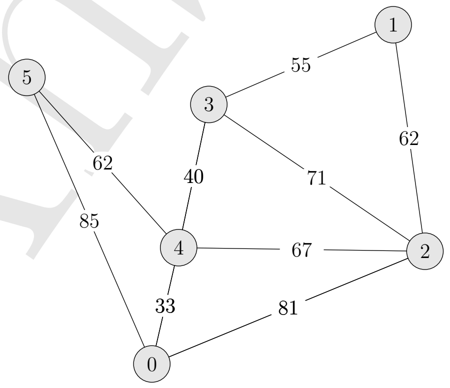
Test knowledge of basic graph algorithm.

Prim's algorithm starts at an arbitrary node. It then builds the minimum spanning tree by moving to the nearest unconnected node to any of the nodes already included in the tree.

The implementation uses a priority tree to collect a list of nearest nodes. It also keeps an array of shortest distances to a node which is initially set to infinity. Nodes added to the minimum spanning tree have their minimum distance set to zero. As we connect a new node into the spanning tree all neighbours of the node are examined and if they are a shorter distance than the current shortest distance they are added to the priority queue with the priority being the distance. Also the list of shortest distances is amended. To obtain the next member of the minimum spanning tree the minimum entry in the priority queue is popped. If this node has not already been added to the minimum spanning tree (i.e. it has a non-zero minimum distance) then it is added to the minimum spanning tree, otherwise it is thrown away and the next element in the priority queue is examined. The process terminates when $n-1$ edges have been added to the minimum spanning tree.

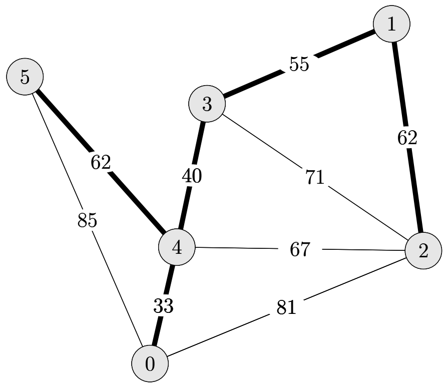
(8 marks)

(b) Compute the minimum spanning tree for the graph shown below.



Starting from node 0, show the order in which the minimum spanning tree is constructed using Prim's algorithm.

Test the student can apply Prim's algorithm.



Starting from node 0, the minimum spanning tree is constructed in the following order (0,4), (4,3), (3,1), (4,5), (1,2). The last two could be in inverse order although in any sensible implementation of a priority queue this won't happen.

(8 marks)

(c) Describe a possible application of minimum spanning trees.

Test knowledge about applications

Minimum spanning trees could be used in finding the shortest (or lowest cost) route for a network joining together cities. For example, if you wanted to construct a fibre optic network which connects all users.

(2 marks)

(d) What do we mean by an "efficient" algorithm? Give two examples of efficient graph algorithms (not including minimum spanning tree). Describe two graph problems for which there are no known efficient algorithms.

Test general knowledge about hardness.

Efficient algorithms are usually taken to be those which are solvable in polynomial time.

Examples of efficient graph algorithms are Dijkstra's algorithm for find minimum length paths and the max-flow algorithm for finding the maximum flow through a network.

Examples of problems with no known efficient algorithms include general TSP, graph-colouring (or vertex colouring...).

(6 marks)

(e) Describe in outline how branch and bound works. Is branch and bound an "efficient algorithm"? Under what conditions is branch and bound useful?

Test knowledge of advanced heuristic.

Branch and bound is a modified exhaustive search algorithm where solutions are systemically constructed in a search tree. Partial solutions are built and evaluated. It differs from exhaustive search in that the best solution found so far is used as a bound on the quality of the solution. For any partial solution which reaches the bound it is known that the complete solution will be no better than the solution already found. It is therefore unnecessary to complete the search beyond that point. This prunes large parts of the search tree, significantly reducing the number of states that need to be searched.

Branch and bound is not generally an efficient algorithm in that there is no polynomial upper bound on it time complexity. In practice, it significantly outperfroms exhaustive search by many orders of magnitude. However, it is usually only practical for fairly small problem sizes.

(9 marks)

END OF PAPER