

# CS 14 - Summer 2003 - Quiz 1

July 2, 2003

## 1 True/False

Circle *T* or *F* as appropriate. Running-time questions are all in terms of input size  $n$

1. **T F**  $n^3$  grows faster than  $n^2 \log_{42} n$  - Divide both by  $n^2$ ,  $n$  is obviously faster than  $\log n$  (remember that the base of a logarithm doesn't matter!)
2. **T F**  $n$  grows faster than  $n \log m$  -  $\log m$  is a constant.
3. **T F**  $2^{2n}$  grows faster than  $3^n$  -  $2^{2n} = (2^2)^n = 4^n$
4. **T F** A *stack* is a FIFO. - FIFO stands for First In First Out. A stack is a LIFO, since the more recent object you place in it is the first thing that will be removed.
5. **T F** A *stack* built using ADT List adds to a different end of the list than it removes from. - A stack adds and removes from the same end of a List, because it is a LIFO.
6. **T F** A *stack* using a linked-list can be implemented so that both the *push* and *pop* operations can be done in  $O(1)$  time. - Insertion and Removal at the head of a Linked-List are both  $O(1)$  operations.
7. **T F** A *queue* is a FIFO.
8. **T F** A *queue* can be efficiently ( $O(1)$ ) implemented as a singly-linked list. - A queue must add and remove from opposite ends of the list, so this sounds infeasible. However, since we know that there will be no removals from one end of the list and no additions to the other, this helps some. We know we can do removal in  $O(1)$  at the head of the list, can we do the same for insertion at the end? The answer is actually "Yes," if we keep a pointer to the tail. (Since there are no removals from this end, we can efficiently keep a pointer to the tail and simply move it forward when we perform an insertion.) - Since I believe I misspoke when describing this in class, and most people answered what I had said in lecture, this question was not counted in the final tally.

## 2 Multiple Choice

Circle the letter of the correct answer. Running-time questions are all in terms of input size  $n$

1. For ADT List, which of these is not part of the basic set of operations?

- (a) Size
  - (b) *Insert At Tail* ADT List has Insert, but Insert At Tail is not builtin, it must be created from Size() and Insert()
  - (c) isEmpty
  - (d) Delete
2. A *linked-list* can never:
- (a) Have links pointing forward and backward
  - (b) *Print its contents in O(1)* How could we possibly print  $n$  elements in less than  $n$  time?
  - (c) Be sorted
  - (d) Be empty
3. A simple sorting algorithm like *selection sort* or *bubble sort* has a worst-case of
- (a)  $O(1)$  time because all lists take the same amount of time to sort
  - (b)  $O(n)$  time because it has to perform  $n$  swaps to order the list
  - (c)  $O(n^2)$  time because sorting 1 element takes  $O(n)$  time - After 1 pass through the list, either of these algorithms can guarantee that 1 element is sorted.
  - (d)  $O(n^3)$  time, because the worst case has really random input which takes longer to sort

### 3 Short Answer

*If you want the option to request a regrade available, your answers to these questions must be made in pen.*

1. (a) What are the worst-case Big-O running times for insertion, removal, and isEmpty for an array implementation of ADT List?
 

$O(n)$  for insertion and removal (we need to expand or compress the array to adjust for the change),  $O(1)$  for isEmpty (we simply check if the number of used elements in the array is 0)
- (b) For a singly-linked list implementation of ADT List?
 

$O(n)$  for insertion and removal (we need to traverse the list to find where the insertion or removal should take place),  $O(1)$  for isEmpty (we simply check if the head of the list is NULL)

2. Write the function `int size(Node* n)` for a singly-linked list of *Nodes* whose head is *n*

```
int size(Node* n)
{
    if (n == NULL) return 0;
    return 1 + size(n->next);
}
```

3. When using only the ADT List interface with no Iterators, what is the running time of a function that prints out the values of a singly-linked list and why?

To print each element of the list, we must get the value using *Retrieve*. For a linked-list, *Retrieve* has a worst-case (and average-case) of  $O(n)$ . There are  $n$  elements in the list, so this results in  $O(n^2)$ .

4. What is the Big-O running time of this code:

```
void order(vector<int>& nums)
{
    for (unsigned int i = 0; i < nums.size(); i++)
    {
        int s = nums[i];
        int sInd = i;
        for (unsigned int j = i + 1; j < nums.size(); j++)
        {
            if (nums[j] < s)
            {
                sInd = j;
                s = nums[j];
            }
        }
        swap(nums[i], nums[sInd]); // This is a constant-time op
    }
}
```

The running time is  $O(n^2)$ , since we have two loops of  $O(n)$ , and everything else (including the if statement) executes in constant time, regardless of the size of the vector *nums*. This code is actually selectionSort.