CS409: HW2 Solutions

Feb, 2001

Bo graded the first five questions; Don graded the last four.

11.1-6 The top pointer of Stack h simulates the head pointer of the resulting queue, while the top pointer of Stack t simulates the tail pointer of the resulting queue:

```
ENQUEUE(x)
    t.PUSH(x);

DEQUEUE(x)
    if (t.EMPTY() && h.EMPTY())
        error "underflow"
    if (h.EMPTY())
        while (!t.EMPTY())
            h.Push(t.POP());
    return h.POP();
```

Comments: Most people get the "correct but inefficient" comment. (I took one point off for this; perhaps it was asking a little too much for now . . .) Some people actually thought about efficiency, for example, in ENQUEUE, if the last action performed is ENQUEUE, the new element is just pushed in, while if the last action performed is DEQUEUE, they move everything back to this stack and then push the new element in, similar with DEQUEUE, so if either ENQUEUE or DEQUEUE is performed repeatedly, this will be more efficient than the most naive algorithm, and it is actually quite close to what I have above. You only need to realize that once those elements are in the other stack ready to be DEQUEUEed, you can just leave them there, you don't need to move them back before entering the new element. The above algorithm has an O(1) amortized complexity (which will be covered in class soon).

11.2-2

```
PUSH(k)
    x.key = k;
    x.next = L.head;
    L.head = x;
```

```
POP()
    if STACK-EMPTY()
        error "underflow";
    x = L.head;
    L.head = x.next;
    return x.key;

STACK-EMPTY()
    return (L.head == NIL);
```

Comments: Almost everyone had the right idea, but:

1. One point was taken off for those who didn't mention STACK-EMPTY.
2. This is supposed to be implemented with a singly linked list, but some of you are setting the pointer *pre*, probably because you figured out this should be using LIST-INSERT and then just copied from the textbook without much thinking. But note that Insertion and Deletion for singly linked list is quite different from that for doubly linked list. I'm not sure that those of you who used LIST-INSERT and LIST-DELETE instead of code were aware of this.
3. For POP, the idea is LIST-DELETE, however you should be returning something, and you shouldn't put the code for deletion after the return line, since codes after return (to be more exact, if they come after return in any possible path) won't be executed...
4. Most people supposed that the parameter for PUSH is an element (or node, with the fields: key and next), and return an element instead of the key for POP. This is actually not very reasonable. The user of the Stack shouldn't need to know that it's actually implemented with a List, so when they use PUSH, they will just give you the *Key* they want to push in, and you should be creating the node with corresponding key to be inserted into the List, similar for POP. No points were taken off for this, but you should note that there is a distinction here.

11.2-5

```
UNION($S_1$, $S_2$)
    if ($S_1$.head == NIL) return $S_2$;
    if ($S_2$.head == NIL) return $S_1$;
    $S_1$.tail.next = $S_2$.head;
    $S_1$.tail = $S_2$.tail;
    return $S_1$
```

Comments:

1. Remember that the code to link the two lists should actually return a list! Many people forgot this, and the grading was pretty lenient in the sense that we tried to figure out which list was intended (normally the first one). Please make sure not to make this kind of mistake again, because you will definitely lose more points.

2. Try to think through your solutions and see if all the pointers make sense in the final result. In some cases, this wasn't true.

3. Many people didn't consider the cases that S_1 or S_2 can be empty lists. Again the grading was quite lenient, but if you were writing real codes, you might be getting exceptions!

11.4.2

```
PRINT-TREE(p)
    if (p == NIL) return;
    PRINT (p.key);
    PRINT-TREE(p.left-child);
    PRINT-TREE(p.right-child);

PRINT-TREE(T.root);
```

Comments: It's really better to decide whether p is NIL at the beginning of the function. Mnany people made the assumption that p is not NIL the first time PRINT-TREE is called. But actually $T.root$ might be NIL, so you'll have to write:

```
if (T.root != NIL) PRINT-TREE(T.root);
```

Actually, most of you only gave the code for the recursive function without specifying how it should be called, so I have no idea whether you are aware of this or not. Again, grading was lenient on this.

12.2-4 Keeping the list in sorted order shouldn't affect successful searches. An unsuccessful search takes time about half the length of the list on average; we can quit once we've gone past the element we're searching for, and on average, the element we're searching for will be around the middle of the list. Insertion will take longer: on average, it will take time about half the length of the list (since we have to run along the list to find the right place to do insertion), whereas insertion at the head takes time $O(1)$ independent of the length of the list. Note that for deletion, we're given a key, not a pointer (unlike the standard deletion in linked lists). If the key we want to delete is actually in the list, on average, we have to search through about half the list to find it, whether or not the list is in sorted order. On the other hand, if the key is not in the list, having the list sorted saves a factor of two on average, just as with unsuccessul search. Is it worth sorting? Probably not, unless the lists are long and we expect to have quite a few unsuccessful searches in our application. [Grading: an informal argument as I've done it is OK for this one.]

12.3-1: The hint in this problem is "long strings". Since it is possible that several strings might have much in common with the target string, the string-to-string comparisons might become lengthy. One could use the hash values merely by replacing, within the algorithm, "Compare target to string in the record at current location within list" with: "If h(target) equals the hash value stored at current record, then compare target to string at current location; else, move on to next record..."

Asuming uniform hashing, the probability of having to compare two strings that would turn out to be different would be reduced to $1/m$, where m is the number of hash slots.

Common mistakes:

1. Some people just said "compare hash values instead of strings". While those people might indeed understand that same hash values do not imply same strings, you are essentially asked to describe a modification to an algorithm, and such imprecision is unacceptable. Generally, no credit was given for this response.

2. Many people prescribed sweeping changes to the data structure to take advantage of the hash values. The problem stated that the hash values are stored with the strings in a linked list. While this is not the best structure for searching, it is clear that you should live within it, and describe an alteration to the linked-list searching algorithm. Depending on the level of radicality of the proposed changes, one point or no points was given.

12.3-4: Fairly straightforward:
h(61) = 700
h(62) = 318
h(63) = 936
h(64) = 554
h(65) = 172
Note that "mod 1" is interpreted as "fractional part".

12.4-1: Here is what the final table looks like:

slot	0	1	2	3	4	5	6	7	8	9	10
linear probing	22	88	-	-	4	15	28	17	59	31	10
quadratic probing	22	-	88	17	4	-	28	59	15	31	10
double hashing	22	-	59	17	4	15	28	88	-	31	10

Comments: There were few notable mistakes other than cases where one miscalculation causes a several-key "pile-up" by causing false hits or forestalling detection of legitimate hits.

12.4-4: Theorem 12.5 gives that on average the maximum number of probes needed for an unsuccessful search would be $\frac{1}{1-\alpha}$. For $\alpha = \frac{1}{2}, \frac{3}{4}$, and $\frac{7}{8}$, we have, respectively, 2, 4, and 8. For a successful search, we are given a maximum of $\frac{1}{\alpha} + \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$. This gives us, respectively, 3.386, 3.182 and 3.519. There were few errors (other than misreading the problem and giving only numbers for unsuccessful searches).