Department of Computer Science & Engineering, Faculty of Engineering
CSC2100B/S: Data Structures **Written Assignment 2**
Spring 2011 Due: *18 Mar, 2011*

Name: ____________________________
SID: ____________________________

Revision 1.1. (28/02)

Section I: Revision Questions (10 points)

Put down your answers to the following simple revision questions.

1. The *link* of linked lists is realized by ______________________________ in C. 1 pt

2. The time complexity for inserting a new node at the beginning of a linked list is ______________ . 1 pt

3. In the Josephus problem, if $N = 8$, $M = 3$ and we start the counting from 1 (downto 8), the person with number ______________ will be elected. 1 pt

4. If the stack ADT is implemented by arrays, the best time complexities of the operations *push* and *pop* is ______________ . 1 pt

5. The string `10 5 + 4 * 2 3 + *` is an example of a/an ______________ expressions. 1 pt

6. The circular array is commonly implemented by the ______________ operator in C. 1 pt

7. For small problems with a small no. of keys within a known range, it is desirable to use a ______________________________ to memorize the data. 1 pt

8. A simple but good hash function for non-negative integral keys k in the range (0, 5000) would be ______________________________ . 1 pt

9. When a hash table of 100 slots with separate chaining has loaded with 75 elements, it is likely to take around ______________ probes to successfully search for a given key. 1 pt

10. The best method of probe sequence generation for open-addressing hash tables discussed in the lecture note is ______________________________ . 1 pt

10 pts

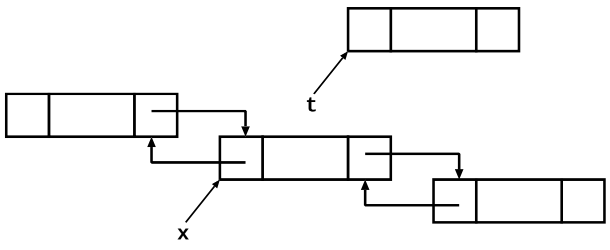
Section II: Short Questions (46 points)
Unless otherwise stated, you should answer the following questions in plain English or math expressions instead of code snippets.
Use the dummy header node convention in singly/doubly linked lists - an empty list is assumed to be a single header node which contains a null link.

1. Explain briefly how to append a singly linked list B to the end of another singly lnked list A. Write down the time complexity for your procedure in terms of the lengths of lists L_A and L_B.

4 pts

2. Suppose you are given a non-empty, *doubly* linked list with N nodes (including the header), extend the following diagram (that only show parts of the list) to illustrate how you insert a new, allocated node t after the node x.

4 pts



What is the time complexity of this insertion?

3. Suggest the sequence of characters popped out from the stack when the input is

`A L G * O * * R I T H * M * * * *`

(Note: character - *push*, asterisk - *pop*)

3 pts

11 pts

4. Suggest the sequence of characters dequeued from the queue when the input is

`A L G * O * * R I * T H * M`

(Note: character - *enqueue*, asterisk - *dequeue*)

3 pts

5. Explain how to implement two stacks using only **ONE** array of size N. Your should also describe a method to check whether all N slots are used up or not.
You may want to use a diagram to explain your answer.

5 pts

6. Postfix to infix conversion:

(a) (5 pts) Design an algorithm that can turn a postfix expressions involving **ONLY** multiplications (`*`) and additions (`+`) into a full parenthesized (`( )`) infix expression.

9 pts

17 pts

(b) (4 pts) Illustrate your algorithm using the following example:

```
6 3 + 2 * 1 8 + * 7 + 4 5 + *
```

7. You are given a hash table of size $M = 5$. The hash function is given as $h(k) = k \mod M$. *Chaining* is used to resolve collisions.
Suppose that the following keys are inserted to the hash table in order:

$$11, 75, 22, 9, 6, 14, 10, 39$$

Draw and show the contents of the hash table after all insertions.

4 pts

8. Consider inserting keys 11, 75, 22, 9, 6, 14, 10, 39 into an empty hash table of size $M = 11$ using open addressing with primary hash function $h'(k) = k \mod M$.
Show the contents of the hash table (as an array) after all insertions, given that the *probing sequence* is generated by

(a) (4 pts) linear probing

8 pts

12 pts

(b) (4 pts) double hashing with $h_2(k) = 1 + (k \mod (M-1))$

9. Propose a data structure that supports the stack *push* and *pop* operations and a third operation *find_min*, which returns the smallest element in the data structure, all in $O(1)$ worst case time. For example, if the input is 4 8 3 $*$, *find_min* will return 4.

6 pts

6 pts

Section III: Long Question / Code Study (44 points)
Unless otherwise stated, you should answer the following questions in plain English instead of code snippets.

16 pts

1. Consider the following basic C structure declaration for the circular doubly linked lists:

```
typedef struct node_s node;
struct node_s {
    int e;
    node *prev;
    node *next;
};
```

The first node of the list will be a dummy header node with the previous link points to the last element. The end of the list is marked by a next link back to the header node. The following is a correct implementation of a certain sorting algorithm:

```
 1  void dll_sort(node *list){
 2      node *x, *t, *m;
 3
 4      for (x = list->next; x != list; x = m->next){
 5          for (m = t = x; t != list; t = t->next)
 6              if (m->e >= t->e)
 7                  m = t;
 8
 9          swap_node(x, m);
10      }
11  }
```

(a) (2 pts) Briefly explain what the for-loop on lines 5-7 is trying to do. Put down the time complexity for this loop in terms of N, the length of the input list.

(b) (4 pts) It is given that the function `swap_node(node *a, node *b)` exchanges the positions of nodes pointed by `a` and `b` respectively. Write an implementation of the function within 20 lines of C code. You can only modify the `prev` and `next` pointers but not the data `e` stored at the node.

(c) (2 pts) Name the sorting algorithm used.
Put down the time complexity of the whole algorithm as implemented above.

16 pts

(d) (3 pts) Is the implementation stable? If yes, explain why; otherwise, discuss how to modify and make it stable.

(e) (5 pts) Using the same structure declaration and functions, write down C code for an stable implementation of the *bubble sort*.

2. The Josephus problem can be simluated using arrays as well. The following code snippet is one of the possible implementations:

16 pts

```c
void josephus(int a[], int N, int M) {
    int i, c, p;
    for (c = 1, p = -1; c <= N; c++){
        for (i = 0; i != M; i += (a[p] == 0))
            p = (p + 1) % N;
        a[p] = c;
        printf("Person #%d is eliminated.", p + 1);
    }
}
```

(a) (2 pts) What should be the correct initialization to the array `a`?

16 pts

(b) (3 pts) Put down the output of the function when it is executed as `josephus(a, 4, 3)`.

(c) (2 pts) If we want to store the order of eliminations for later use (at line 7), what data structure should be used?

(d) (3 pts) What is the **worse case** time complexity of this algorithm? Explain breifly using the O-notation in terms of N and M.

(e) (6 pts) Suggest another array-based C implementation of the problem that generates the same output as the program above. Put down the time complexity of your algorithm in terms of N and M.
(Note: do not forget to write a suitable initialization for `a`.)

0 pts

12 pts

3. Consider the implementation of an open-addressing hash table storing C strings as declared below:

```c
typedef struct hashtbl_s {
    int m;           /* size of the hash table */
    char *slots[];   /* array of (char *) */
} hashtbl_t;
```

Suppose that the hash table is properly initialized (i.e. all slots to NULL pointers) and the function `oahash(key, i, M)` gives the probe sequence based on *linear probing*, the search/insertion subroutine is implemented as follows:

```c
int hash_find_insert(hashtbl_t *ht, const char *key){
    int i, j;
    for (i = 0; i != ht->m; i++){
        j = oahash(key, i, ht->m);
        if (ht->slots[j] == NULL){
            ht->slots[j] = strdup(key);
            return -1;
        }
        if (strcmp(ht->slots[j], key) == 0)
            return j;
    }
    return -2;
}
```

NOTE: YOU MAY IGNORE ERROR CHECKING IN THIS QUESTION.

(a) (6 pts) A method to delete an entry in the hash table is to move all occupied slots in the probe sequence of key k to its previous slot, so that the succeeding searches can locate the items (strings in this question) as usual.
For example, if the probe sequence is $4, 5, 2, 6, \ldots$, when we delete the item at slot 5, the item at slot 2 will be moved to slot 5, the item at slot 6 will be moved to slot 2, ad so on, until no item is found.
Write a C function `void hash_delete(hashtbl_t *ht, const char *key)` that realizes the above algorithm.
You may assume that the given key exists in the hash table.

12 pts

(b) (3 pts) How many items do you expect to move *on average* when the hash table is half-full? Explain briefly.

(c) (3 pts) Describe another method for deletion in open-addressing hash tables.

0 pts