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 __**pointers**__ in C. [1 pt]

2. The time complexity for inserting a new node at the beginning of a linked list is __$O(1)$__ . [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 __**7**__ 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 __$O(1)$__ . [1 pt]

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

6. The circular array is commonly implemented by the __**modulus**__ 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 __**direct-address table**__ 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 __$k \mod M$**, where** $M < 5000$__ . [1 pt]

9. When a hash table of 100 slots with separate chaining has loaded with 75 elements, it is likely to take around __**3**__ 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 __**double hashing**__ . [1 pt]

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.

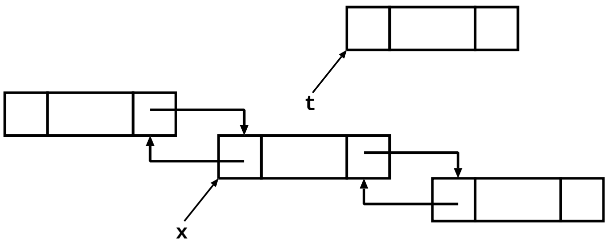
1. Explain briefly how to append a singly linked list B to the end of another singly lnked list A. Write down the time complexity for your procedure in terms of the lengths of lists L_A and L_B.

4 pts

Solution: Traverse to the end e of A in $O(L_A)$ time.
Set next pointer of e to link the first node of B (the node pointed by the header) - $O(1)$ time. Total complexity = $O(L_A)$.

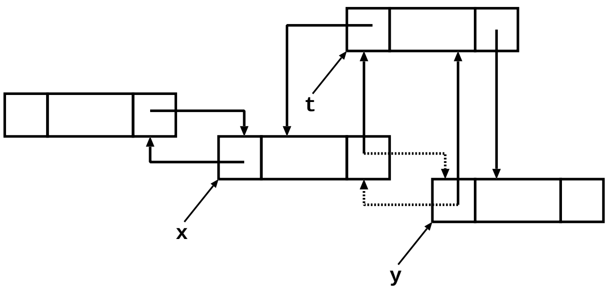
2. Suppose you are given a non-empty, *doubly* linked list with N nodes (including the header), extend the following diagram (that only show parts of the list) to illustrate how you insert a new, allocated node t after the node x.

4 pts



What is the time complexity of this insertion?

Solution: Denote the original next node of x to be y. Then update as follows:
$next(x) = t$, $prev(t) = x$, $next(t) = y$ and $prev(y) = t$, as shown.



It takes constant time to insert the new node - $O(1)$.

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

3 pts

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

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

Solution:

G O L H M T I R

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

3 pts

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

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

Solution:

A L G O R

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

Solution: Two stacks can be implemented in an array by having one grow from the low end (0), and the other from the high end ($N-1$).
Initialization: set `tos1` to -1 and `tos2` to N.
To push to stack 1, increment `tos1` and write to `a[tos1]`.
To push to stack 2, decrement `tos2` and write to `a[tos2]`.
All N slots are used up when `tos1` equals `tos2`.

6. Postfix to infix conversion:

9 pts

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

Solution: Keep a stack S for operands.
Process the postfix expression from left to right. Whenever we see an operand, push it to S; whenever we see an operator, form a new infix expression with parenthesized in the following form:

$$(pop(S)\ op\ pop(S))$$

Push the new expression back to S and continue until the whole postfix expression is processed.

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

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

Solution:

```
Input      Stack
  6        6
  3        6 | 3
  +        (6 + 3)
  2        (6 + 3) | 2
  *        ((6 + 3) * 2)
  1        ((6 + 3) * 2) | 1
  8        ((6 + 3) * 2) | 1 | 8
  +        ((6 + 3) * 2) | (1 + 8)
  *        (((6 + 3) * 2) * (1 + 8))
  7        (((6 + 3) * 2) * (1 + 8)) | 7
  +        ((((6 + 3) * 2) * (1 + 8)) + 7)
  4        ((((6 + 3) * 2) * (1 + 8)) + 7) | 4
  5        ((((6 + 3) * 2) * (1 + 8)) + 7) | 4 | 5
  +        ((((6 + 3) * 2) * (1 + 8)) + 7) | (4 + 5)
  *        (((((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: 4 pts

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

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

```
0: [   |-]-> [10 |-]-> [75 |-]-> NULL
1: [   |-]-> [ 6 |-]-> [11 |-]-> NULL
2: [   |-]-> [22 |-]-> NULL
3: [   |-]-> NULL
4: [   |-]-> [39 |-]-> [14 |-]-> [ 9 |-]-> NULL
```

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 8 pts

(a) (4 pts) linear probing

Solution:

```
   0    1    2    3    4    5    6    7    8    9   10
[ 11 ][ 22 ][ 10 ][ 14 ][  x ][  x ][  6 ][ 39 ][  x ][ 75 ][  9 ]
```

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

Solution:

```
   0    1    2    3    4    5    6    7    8    9   10
[ 11 ][  x ][ 14 ][ 22 ][  x ][ 39 ][  6 ][  x ][  9 ][ 75 ][ 10 ]
```

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. (You may assume all elements are distinct.) For example, if the input is 4 8 3 $*$, *find_min* will return 4.

6 pts

Solution: Let E be our extended stack. E will be implemented with two stacks. One of them S is used to keep track of *push* and *pop* and the other M keeps track of the minimum.

To implement $push(X, E)$, we perform $push(X, S)$. If X is smaller than or equal to the top element in M, perform also $push(X, M)$. To implement *pop*(E), we perform *pop*(S) and if the return element is equal to the top element in M, perform also *pop*(M). $find\_min(E)$ is realized by examining the top of M.

All operations are in $O(1)$.

Section III: Long Question / Code Study (44 points)

Unless otherwise stated, you should answer the following questions in plain English instead of code snippets.

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

16 pts

```c
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:

```c
 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.

Solution: Finding the minimum element among all nodes on the list.
$O(N)$

(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.

Solution:

```c
void swap_node(node *a, node *b){
    node *ap = a->prev, *an = a->next;
    node *bp = b->prev, *bn = b->next;

    assert(a && b && ap && an && bp && bn);

    ap->next = b;
    bn->prev = a;
    a->next = bn;
    b->prev = ap;
    if (an == b){
        b->next = a;
        a->prev = b;
    }
    else {
        an->prev = b;
        bp->next = a;
        a->prev = bp;
        b->next = an;
    }
}
```

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

Solution: Selection sort. Time complexity = $O(N^2)$

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

Solution: It is not stable.
It can be improved to stable if (i) lines 6 and 7 is modified to:

```
if (m->e > t->e)
    m = t;
```

and (ii) line 9 is modified to (insert m after x):

```
m->next = x->next;
m->prev = x;
x->next->prev = m;
x->next = m;
```

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

Solution:

```
void dll_bubble(node *list){
    node *end, *t;

    for (end = list; end->prev != list; end = end->prev){
        for (t = list->next; t->next != end; ){
            if (t->e > t->next->e)
                swap_node(t, t->next);
            else
                t = t->next;
        }
    }
}
```

2. The Josephus problem can be simluated using arrays as well. The following code snippet is one of the possible implementations: 16 pts

```
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\n.", p + 1);
    }
}
```

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

Solution: Every entry has to be written zero's.

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

Solution:

```
Persion #3 is eliminated.
Persion #2 is eliminated.
Persion #4 is eliminated.
Persion #1 is eliminated.
```

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

Solution: Queue. FIFO.

(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.

Solution: For loop on line 3 clearly executes for N times. The inner loop on line 4 in the worse case can iterate MN times. The no. of iterations $= O(MN^2)$.

(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`.)

Solution:

```c
void josephus2(int a[], int N, int M){
    int i, p = 0;

    for (i = 0; i < N; i++)
        a[i] = i + 1;

    do {
        p = (p + M - 1) % N;
        printf("Person #%d is eliminated.\n", a[p]);
        for (i = p; i < N - 1; i++) /* shift down */
            a[i] = a[i + 1];
    } while (--N);
}
```

Time complexity $= O(N^2)$

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

12 pts

```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 $4, 5, 6, 7, \ldots$, when we delete the item at slot 5, the item at slot 6 will be moved to slot 5, the item at slot 7 will be moved to slot 6, 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.

Solution:

```c
void hash_delete(hashtbl_t *ht, const char *key){
    int i, j, k, p = ht->m;

    if ((p = hash_find_insert(ht, key)) < 0) return;

    for (i = p; i != ht->m; i++){
        j = oahash(key, i, ht->m);
        k = oahash(key, i + 1, ht->m);
        if (ht->slots[k] != NULL)
            ht->slots[j] = ht->slots[k];
        else {
            ht->slots[j] = NULL;
            break;
        }
    }
}
```

(b) (3 pts) How many items do you have to move at most when the hash table is half-full? Explain briefly.

Solution: When the load factor is 0.5, the no. of search miss is around 1.5, this would be the number of probes until you have an empty slot. Therefore on average 0.5 item is moved (in worse), given that the key is found in the first probe.

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

Solution: Mark the slot to a predefined value `HASH_DELETED`.
Modify search such that it continues when `HASH_DELETED` is encountered and stops only when the slot is found empty.
Modify insert such that new item will be added whenever the slot is either empty (`NULL`) or deleted previously (`HASH_DELETED`).