CSE326 – Data Structures, Winter 2004
Dry assignment #2 - Solutions

Problem 1.
The first interpretation one could make from reading the text of the problem is that IDs are distributed to customers ahead of time and they don't necessarily reflect the order of arrival of customers. This is why `client(k)` is a needed function.

The data structures to use are 2 arrays and 2 associated variables, one for each array. Specifically:

```
var customerSequence[N]: array of integer;
   /* the element with index k holds the ID of the k-th arriving customer */
var numCustomersToDate: integer;
   /* holds the length of the array customerSequence[] */
var customerTotalPurchased[N]: array of integer;
   /* holds the total value of goods purchased by a customer with a given ID */
var numPotentialCustomers: integer;
   /* holds the value that init() passes in; it could be less than N */
```

The space complexity of the data structure is indeed O(N) – the maximum number of elements each array can hold. As far as time complexities, let us examine each function separately:
- `init(N)` – initializes `numPotentialCustomers` to `N`, the starting length `numCustomersToDate` of the `customerSequence[]` array to 0, and initializes all `N` elements of the `customerTotalPurchased[]` array to 0 (since customers have not bought anything yet). The time complexity of this operation is determined by the length of the arrays and is therefore O(`N`).
- `purchase(i,x)` – adds `x` to the value stored in `customerTotalPurchased[i]`, and if it was 0 before, increments by 1 the value of `numCustomersToDate`, and sets the value of `customerSequence[numCustomersToDate]` to the ID `i` of the customer. The time complexity of this operations is O(1) – only assignments, comparisons, and conditionals are executed.
- `sum(i)` – returns the value of `customerTotalPurchased[i]`. The time complexity here is clearly O(1).
- `client(k)` – returns 0 if `numCustomersToDate` is 0 or if `k` is larger than `numCustomersToDate`. Otherwise, returns `customerSequence[k]`. The time complexity here is O(1) too – only a comparison and a conditional are executed.

One important assumption we made above is that array indices begin from 1, not 0. (An adjustment of indices would be necessary if it were the other way round.)

Note: It is necessary to have the numCustomersToDate counter for the purchase() function to be O(1) time in the case the customer is new.

Problem 2.

a) Here is how the adaptive linked list looks like after each step.

Initial state:	header $\rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$
After `find(3)`:	header $\rightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 5$
After `insert(6)`:	header $\rightarrow 6 \rightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 5$
After `delete(2)`:	header $\rightarrow 6 \rightarrow 3 \rightarrow 1 \rightarrow 4 \rightarrow 5$
After `find(4)`:	header $\rightarrow 4 \rightarrow 6 \rightarrow 3 \rightarrow 1 \rightarrow 5$

b) It is typical that linked list-based implementations of `insert()` or `delete()` use the operation `find()` as a core "building block." However, here it is convenient to do it the other way round, since `find()` needs to not only find the element, but move it from its current position to the front of the list. This moving to the front can

be accomplished either as a deletion from the old place in the list followed by an insertion to the front (which we will show in pseudocode below), or it can be done independently of `insert()` and `delete()`, essentially duplicating much of the functionality of these two routines. The latter alternative is slightly more efficient in practice (i.e., the time complexity order remains the same, but the constants improve), but its drawback is that it fails to reuse functions that have already been built – often a highly desirable objective.

```
insert (x: integer; lst: list pointer): void
{
   var t: node;
   t = new node;
   t.elem = x;
   t.next = lst.next;
   lst.next = t;
}

delete (x: integer; lst: list pointer): integer
{
   var status: integer;
   status = NOT_FOUND;
   while (NULL != lst.next)
   {
      if (lst.next.elem != x)  /* look for the element x; move on if not found */
         lst = lst.next;
      else                     /* ... or exit the loop if found               */
         break;
   }
   if (NULL != lst.next) /* if x was found, adjust pointers and free up memory */
   {
      var temp: node;
      temp = lst.next;
      lst.next = temp.next;
      temp.next = NULL;
      delete(temp);  /* frees the dynamically alloc'd memory that temp points to */
      status = FOUND;
   }
   return (status);  /* this status is necessary in the find() method */
}

find (x: integer; lst: list pointer): integer
{
   var status: integer;
   status = delete(x, lst); /* first delete the element if it's in the list...  */
   if (FOUND == status)
      insert(x, lst);       /* ... then insert it, if it was in the list before */
   return (status);
}
```

Note: Many of you did not do any error checking, e.g., accessing the contents of what a pointer points to without checking if it is NULL.

c) We will show two similar ideas below.
Idea 1: Let's assume that `find(z)` occurs on operations op_1, op_2, …, $op_{k/4}$, where $\{op_i\}_{i=1..k/4}$ is an ordered subset of the set $\{i\}_{i=1..k}$. (For instance, $op_1 = 1$, $op_2 = 5$, $op_3 = 6$, $op_{k/4} = k\text{-}3$ is one such subset.)
- The cost (i.e., the time complexity) of doing `find(z)` the very first time will be
  $cost_{find(z),1} = O(N)$,
since the element z will need to be found in a linked list of N elements, and moved to the front of the list.
- For j>1, the cost of doing `find(z)` for the j^{th} time (i.e., on operation op_j) will be

$cost_{\texttt{find(z)},j} \le (op_j - op_{(j-1)})$,

since z was moved to the front of the list during the previous find (on operation $op_{(j-1)}$), so it could have been pushed back by at most $(op_j - op_{(j-1)})$ different elements that have moved to the front between operations $op_{(j-1)}$ and op_j. Hence, one needs to traverse at most $(op_j - op_{(j-1)})$ elements to find z on operation op_j.

As the total time dedicated to `find(z)` across all k `find()` operations will be the sum of the times that `find(z)` takes for each individual operation op_i, from here we can write

$total\_cost_{\texttt{find(z)}} = cost_{\texttt{find(z)},1} + \Sigma_{j=2..k/4}\ cost_{\texttt{find(z)},j} \le c_1.N + \Sigma_{j=2..k/4}\ (op_j - op_{(j-1)}) =$

$= c_1.N + (op_2 - op_1) + (op_3 - op_2) + \ldots + (op_{k/4} - op_{(k/4\ -\ 1)}) = c_1.N + (op_{k/4} - op_1) \le c_1.N + k \le$

$\le c_2.k$,

since $k = \Omega(N)$. Therefore,

$average\_cost_{\texttt{find(z)}} = total\_cost_{\texttt{find(z)}} / (k/4) \le c_2.k / (k/4) = 4c_2 = O(1)$.

Idea 2: A slightly more elegant, though perhaps more abstract, solution rests on the following observations (most of them already explained above):

the first `find(z)` operation will take time O(N);

each `find(y)` operation (for elements $y \neq z$) pushes the z element by at most one position behind in the list;

there are a total of 3k/4 `find(y)` operations, so the z element will have to "travel" at most a total of 3k/4 positions back to the top of the list, regardless of how many steps at a time that may take.

From here, the total time for `find(z)` will be at most (N + 3k/4). Therefore, the average time becomes

$(N + 3k/4) / k = N/k + 3/4 = O(1)$,

since $k = \Omega(N)$.

Problem 3.

a. A Stack is an appropriate data structure. According to the rule (iii), the end symbols must be closed in reverse order. This rule reminds one of the Last-In-First-Out rule of stack. In addition, by using stack the validation-checking algorithm can be done not only in linear time, but in a single pass through the input.

b.

We will use the following variables/ADTs

Stack: stack ADT

paren_ctr, square_ctr, brace_ctr: integers which are initialized to zero

We do the following on reading each input.

'(': increment paren_ctr, push '(' onto Stack

'[': increment square_ctr, push '[' onto Stack

'{': if square_ctr or paren_ctr is nonzero, report error (v), otherwise increment brace_ctr and push '{' onto Stack

')': if paren_ctr is zero, report error (ii). If Stack.Pop() is not '(', report error (iii). Otherwise decrement parent_ctr.

']': if square_ctr is zero, report error (ii). If Stack.Pop() is not '[', report error (iii). Otherwise decrement square_ctr.

'}': if brace_ctr is zero, report error (ii). if Stack.Pop() is not '{', report error (iii). Otherwise decrement brace_ctr.

At the end of the input, if Stack is not empty, report error (i).

c. For rule (i), if all the symbols paired up, the stack should be empty at the end of the input.
For rule (ii), if there is a right symbol comes up with no left symbol in the previous positions, which means the counter of that symbol is zero, then the string is illegal.

For rule (iii), if the symbols are placed in the string with the right order, which means the right symbols are in the reverse order of left symbols, then the right symbol will be paired up with the popped left symbol. Otherwise it is illegal, if it is not against rule (ii) then it is against rule(iii). Rule (iv) is implied by rule (iii).
For rule (v), if the '{' comes up with no other symbol in front of it except itself, then the counter of the other two symbols should be zeros. Otherwise the string is illegal.

In the assignment it was not necessary to distinguish between rules (ii) and (iii). There was some confusion about this when grading, and announcement will be made to the class list.