Solutions for Homework 2

Problem 1: (Grade 2.5 + 5 pts) You have seen the implementation of a stack using an array. In the implementation shown in class, you add the first element at position 0, next element at position 1, etc. You always add and delete elements from the end of the list. Consider an array of size n. Design a stack implementation where you insert the first element at position n-1, the second at position n-2 etc. From which end do you add and delete elements?

Associated with each stack is *TopOfStack*, which is N for an empty stack (this is how an empty stack is initialized). To push some element X onto the stack, we decrement *TopOfStack* and then set *Stack*[*TopOfStack*]=X, where *Stack* is the array representing the actual stack. To pop, we set the return value to *Stack*[*TopOfStack*] and then increment *TopOfStack*.

Implement two stacks using one array of size n. You should be able to push and pop from both the stacks in constant time. Your stack overflows if total size of both stacks exceed n. Your algorithm should detect overflow when it happens and subsequently terminate.

The previous question gives us a hint. The first stack will add the first element at postion 0, second element at position 1 etc. The second stack will have the same model as the one described in the previous question. We denote *T1* as top for the first stack and *T2* as top for the second stack.

```
The algorithm is like this:

Init() {
 T1=-1;
 T2=n;
}

The following  procedure checks for stack overflow:
int isFull() {
  return (T2-T1==1);
}
```

Problem 2: Grade 5 pts You have a character string as input to your program. You need to find out whether the string is of the form a C b. Here C is the letter C, a and b are sequences of As and Bs, and a and b must be mirror images of each other, e.g., a = ABAAB, b = BAABA. For full grade you must give an O(N) algorithm where N is the number of letters in the input string.

Keep two variables T1 and T2. T1 is an index to the first character of the string and T2 is an index to the last character of the string. Compare the characters that correspond to positions T1 and T2. If they do not match, then the input is not of the desired form. If they match, then you increment T1 and decrement T2 and follow the same rule. The algorithm terminates when both T1 and T2 find character C.
Since this scheme traverses the whole input string, is $O(n)$.

Problem 3: Grade 10 pts You have two sorted lists, L_1, L_2. You know the lengths of each list, L_1 has length N_1 and L_2 has length N_2. Design algorithms to output a sorted list $L_1 \cap L_2$. You need to give two algorithms. One should have complexity $O(N_1 + N_2)$ and another should have complexity $O\left(\min(N_1, N_2)\log\left(\max(N_1, N_2)\right)\right)$. Which (if any) is faster? (There are three possibilities here, first algorithm is faster for all N_1, N_2, second algorithm is faster for all N_1, N_2, or the answer depends on the values of N_1 N_2.) Justify your answer.

```
Algorithm 1:

Intersection(L1,L2) {

int T1=1,T2=1;

 while ((T1<=N1) && (T2<=N2)) {
   if (L1[T1]<L2[T2])
T1++;
   else if (L1[T1]==L2[T2]) {
  printf(L1[T1]);
          T1++;
  T2++;
        }
        else  T2++;
 }

}
```

Every iteration of the while loop moves by one array element(in L1 or L2 or both). As there are a total of $N1 + N2$ elements in both arrays, and the algorithm terminates when all these array elements are scanned, the time taken is $O(N1 + N2)$.

```
 {
Algorithm 2:

We can search if every element of the smallest list min(N1,N2) is present in
the other list max(N1,N2). The search can be a binary search because we are
given sorted lists.
Without loss of generality, assume that N1 is smaller than N2.
BinarySearch code can be found in page 30 of the textbook, so it is omited here.

Intersection2(L1,L2) {

int i=1;

  while (i<=N1) {
   if (BinarySearch(L2,L1[i],N2)
       printf(L1[T1]);
   i++;
  }
 }
```

The binary search for a particular number takes time $O(\log(max(Ni, N2)))$, and it is invoked $min(N1, N2)$ times. Hence, total time is:

$$O(min(N1, N2) \log(max(Ni, N2)))$$

If $N1 = \Theta(N2)$ then the first algorithm is $O(N1)$, and the second is $O(N1 \log N1)$. So, in this case the first is faster.

If $N1 = o(N2)$ then the first algorithm takes $O(N2)$, and the second takes $O(N1 \log N2)$. So, in this case the second is faster.

Problem 4: Grade 7.5 + 10 pts This problem requires you to implement a special stack. The special stack does the usual push and pop in constant times. In addition, it must find the minimum value of elements currently in stack in constant time (find-min operation).

There are different solutions. One of them is the following:
Use an implementation of a stack along with an auxiliary stack to keep track of the minimum element in the stack at each point. Updating of the auxiliary stack may either be done at every operation or only when the minimum in the stack changes. Let's take the implementation that keeps both stacks at the same height.
Under this implementation, when doing a **push**, we compare the element x to be pushed with the top of the auxiliary stack y (representing the current minimum). If $x < y$ we push x on to both stacks. Otherwise we push x on to the regular stack and y on to the auxiliary stack.

When performing a **pop** simply pop both stacks and return the value popped from the regular stack.

When performing a find-min return the value at the top of the auxiliary stack (without popping it).