ESO207A: Data Structures and Algorithms

Practice Set *2: Heaps and Stacks* –

1. A Max-Priority Queue supports the operation INCREASE-KEY(A, i, v) that increases $A[i]$ to v. Can you write the operation DECREASE-KEY(A, i, v) that decreases $A[i]$ to v to work in worst-case $O(\log n)$ time.

Solution. Suppose we implement a Max-Priority Queue as a Max-Heap. The value of node i is decreased to v. Thus, the new value continues to be no larger than its parent and i is in the correct relation with respect to PARENT(i), that is, $A[i] \leq A[\text{PARENT}(i)]$. Hence, we may try to float the value down in the heap, until it finds its correct position.

```
DECREASE-KEY(A, n, i, v) // A is a max-heap with n elements, A[i] is decreased to v.
1.  done = FALSE
2.  A[i] = v
3.  while i is not a leaf node and not done
4.          // find the largest among i, LEFT(i) and RIGHT(i)
5.      largest = i
6.      if A[i] < A[LEFT(i)]
7.          largest = LEFT(i)
8.      if RIGHT(i) exists in the heap and A[largest] < A[RIGHT(i)]
9.          largest= RIGHT(i)
10.     if largest == i
11.         done = false // i continues to be locally the largest, so we are done
12.     else            // else exchange A[i] with largest child and continue
13.         Exchange A[i] with A[largest]
14.         i = largest
```

2. Design a data structure that supports the following operations: INSERT, MEDIAN and EXTRACT-MEDIAN of a dynamic set. Each of the operations should run in time $O(\log n)$ worst-case time. (*Hint:* Use two heaps).
Can you extend the above data structure to additionally support the operations MAXIMUM and EXTRACT-MAX in worst-case $O(\log n)$ time.

Solution. This is a classic question involving heaps. We wish to design a data structure that supports INSERT, MEDIAN and EXTRACT-MEDIAN. We know that in a max-heap, we can support INSERT, MEDIAN and EXTRACT-MAX. So, one way to approach this problem is to try to make the median as the maximum element of a heap. This gives us the following idea. We will maintain the following invariant. Suppose the dynamic set at any time has n elements.

1. Keep the $\lceil n/2 \rceil$ minimum elements in a max-heap, call it L.
2. Keep the remaining $\lfloor n/2 \rfloor$ maximum elements in a min-heap, call it R.

This way, the median is always the maximum element of the max-heap L. The advantage is that the next-largest element is available as the minimum element of R.

How can we implement insertion of a value v? Suppose n is odd. If v is larger than the correct median, then it is inserted into the heap of larger elements R. The number of elements increases from odd to even, and the invariant is maintained. If however, v is smaller than the correct median, then, the median changes. We now do the following.

1. Extract-Max from L which gives us the old median. Insert this into the right heap R.
2. Insert v into L.

We have deleted the old median and inserted v into L. Thus, L has the same number of elements as before, that is, $\lceil n/2 \rceil = (n+1)/2 = \lceil (n+1)/2 \rceil$, since n is odd. We have added the old median to R, thereby increasing its size by 1, as desired. The new median will float up to the top of the heap L as a result of the ExtractMax followed by Insert.

Now suppose n is even. If v is larger than the median value, then, v must be inserted into the second half, namely R. After this insertion, there is an imbalance in the number of elements in the first and second half, as the first half only has $n/2 = \lfloor (n+1)/2 \rfloor$ elements. To fix this, we shift the minimum element of the min-heap R and insert it into the max-heap L, which will float to the top. This maintains the invariant.

The operation Extract-Median can be implemented in a similar fashion. If n is odd, call Extract-Max on L and return. If n is even, after Extract-Max(L) is called, it only has $n/2 - 1 = \lfloor (n-1)/2 \rfloor$ elements, and so is short by 1. We compensate this by shifting the minimum element of R into L.

```
global variables $n$, $L$ and $R$
Shift-L-to-R //Shifts the max element of $L$ to $R$
1.  $x =$Extract-Max($L$)
2.  Insert($R, x$)

Shift-R-to-L { Shifts the min element of $R$ into $L$
1.  $x =$ Extract-Min($R$)
2.  Insert($L, x$)

Extract-Median
1.  if $n$ is odd
2.      $n = n - 1$
3.      return Extract-Max($L$)
4.  else
5.      $m =$ Extract-Max($L$)
6.      Shift-R-to-L
7.      $n = n - 1$
8.      return $x$

Insert($v$)
1.  $m =$Maximum($L$)
2.  if $n$ is odd
3.      if $m \leq v$
```

```
4.             INSERT($R, v$)
5.         else
6.             SHIFT-L-TO-R
7.             INSERT ($L, v$)
8.     else // $n$ is even
9.         if $v \leq m$
10.            INSERT($L, v$)
11.        else
12.            INSERT($R, v$)
13.            SHIFT-R-TO-L
14.    $n = n + 1$
```

3. You are given an arithmetic expression that can contain three types of brackets, left and right parenthesis, left and right braces and left and right square brackets, namely, '(' and ')', '{' and '}' and '[' and ']'. Write a program that checks whether the given expression has properly nested and matching parenthesis of all types that occur in it. For example, $\{2 + (3 + [5\%7] * 10)/3\} - 9$ is a well-formed expression with matching parenthesis. To test well-formedness in terms of properly nested parenthesis of all types, one can ignore the operators and operands completely and remove them from the expression. This leaves a string consisting only of left and right parenthesis of the three kinds, which can be tested. Examples:

> Ignoring the operands and operators, the following expression is well formed: []([{()}]{()}).
>
> Ignoring operators and operands, the following expression is not well-formed: [({)}]

Solution. This problem typifies one of the many among a class of applications where a stack is used to recognize well-formed strings according to a formal notation. Here, a string is well-formed if the multiple parenthesis types are properly nested, and otherwise not.

The algorithm is as follows. Keep a stack for the left parenthesis (of all types). Design a tokenizer that classifies inputs into operators, operands and parentheses. Operators and operands are ignored as they appear on the input stream. If the next token is a left parenthesis, push it on the stack. If the next token is a right parenthesis, check its type ']', '}' or ')'. The corresponding left parenthesis must be on the top of the stack. If so, pop the stack and continue. Otherwise, the expression is malformed.

4 You are given an array (of numbers or characters) $A[1 \ldots n]$. For $1 \leq i \leq n$, let $g(i)$ be the smallest index with the smallest number or character in $A[i \ldots n]$, that is, $A[g(i)] = \min_{j=i}^{n} A[j]$. Compute $g[1 \ldots n]$ in linear-time.

Solution.

```
COMPUTEG($A, n$)
1.  $g[n] = A[n]$
2.  for  $i = n - 1$ downto 1
3.      if  $A[i] \leq A[g[i+1]]$
4.          $g[i] = i$
5.      else $g[i] = g[i+1]$
```

The loop maintains the invariant that $g(i+1)\ldots g(n)$ are correctly computed in $g[i+1\ldots n]$. Initially, for n, this is correct. Inductively, $g(i)$ is i if $A[i] \leq \min_{j=i+1}^{n} A[j] = A[g(i+1)]$. This is exactly the test. *Time*: The program clearly runs in linear time.

4. Given an array of numbers $A[1\ldots n]$, for each i, $1 \leq i \leq n$, $g(i)$ is defined as the nearest index $j \geq i$ such that $A[j] > A[i]$. Let $g(n) = \infty$ and if $A[i]$ is not smaller than any element in $A[i+1\ldots n]$, then we set $g(i) = \infty$. (You may assume that the numbers in the array are all distinct, if needed, though this is not necessary). For example, consider the array $A = \{10, 7, 6, 8, 9, 12, 1\}$. Then, $g(1) = 6$, $g(2) = g(3) = 4$, $g(4) = 5$, $g(5) = 6$, $g(6) = g(7) = \infty$. Give a linear-time algorithm to compute $g[1\ldots n]$.

The program maintains the following invariants.

1. The stack S contains indices from top of stack to bottom say $s_1, s_2, \ldots, s_k$, where, s_1 is the index on top of stack and s_k is the index at the bottom of the stack.
2. $s_1 > s_2 > \ldots > s_k$ and moreover, $A[s_1] \leq A[s_2] \leq \ldots \leq A[s_k]$.
3. Consider the state of the program just prior to one execution of the for loop. Then, $g[1,\ldots,i-1]$ has been determined, except for $g(s_1), g(s_2), \ldots, g(s_k)$. Moreover, for $s_{j-1} < r < s_j$, $g(r) \leq s_j$.

```
NEAREST_GT($A, n$) // returns an array $g[1\ldots n]$
1.  STACK $S$; // $S$ is a stack of indices
2.  MAKE_EMPTY($S$) // Make $S$ to be an empty stack
3.  for $i = 1$ to $n$
4.      $curr = A[i]$ // stores the value of $A$ at current index $i$
5.      while not ISEMPTY($S$) and $curr > A[\text{TOP}(S)]$
6.          $s = \text{POP}(S)$
7.          $g[s] = i$
8.      PUSH($S, i$)
9.  while not ISEMPTY($S$)
10.     $g[\text{TOP}(S)] = \infty$
11.     POP($S$)
12. return $g$
```

5. You are playing a game that allows the following moves. Given an initial string of characters s and empty strings t and u initially. The moves are :

- Remove a character from the front of s and append it to the end of t.
- Remove a character from the end of t and append it to the end of u.

The game ends when s is empty and t is empty and the answer of the game is u. Design a program that makes u to be lexicographically the smallest string possible. For example, suppose the input is `cab`. Then, consider the following moves.

s	t	u
cab		
ab	c	
b	ca	
b	c	a
	cb	a
	c	ab
		abc

Clearly, `abc` is lexicographically the smallest string possible for u.

Example 2. Consider the input string $s =$ `acdb`.

s	t	u
acdb		
cdb	a	
cdb		a
db	c	a
b	cd	a
	cdb	a
	cd	ab
	c	abd
		abdc

The smallest string in lexicographic order possible is `abdc`.