**CSE548, AMS542: Analysis of Algorithms, Spring 2015** **Date:** Mar 18

# Homework #2

**( Due: Apr 2 )**

**Task 1.** [ **70 Points** ] **Crisscrossed**

There are $n > 0$ locations on each side of a river running straight from north to south. Suppose the locations on the west side of the river are numbered from $w_1$ to $w_n$, and those on the east side from $e_1$ to $e_n$ as they are encountered from north to south. For $1 \le i, j \le n$, each $w_i$ is connected to exactly one $e_j$ using a straight bridge over the river, and vice versa. Thus there are exactly $n$ bridges, and for $1 \le k \le n$, each bridge $b_k$ is given by its two endpoints $(w_i, e_j)$. We say that two bridges $b_k = (w_i, e_j)$ and $b_{k'} = (w_{i'}, e_{j'})$ with $k \ne k'$ cross provided either $(i < i') \wedge (j > j')$ or $(i > i') \wedge (j < j')$ holds. In the example shown in Figure 1 bridge $(w_1, e_3)$ crosses bridge $(w_2, e_1)$, but it does not cross bridge $(w_3, e_5)$. That example has 8 bridge crossings in total.

Given $n > 0$ bridges $b_1, b_2, \ldots, b_n$, function Print-Crossings prints all crossings between bridges in the input.

This task asks you to compute the number of times the ***print*** instruction in line 6 of Print-Crossings is executed averaged over all possible inputs of size $n$ (i.e., all possible ways $n$ bridges can be constructed).

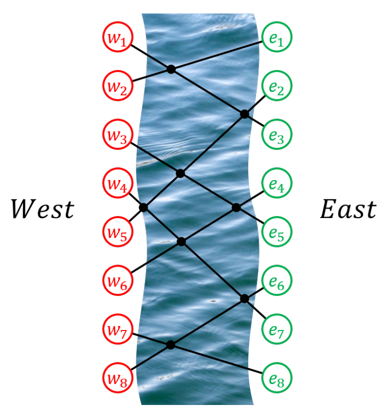


Figure 1: A river running straight from north to south with 8 specific locations on each side. A bridge connects a location on the west bank to a location on the east bank. Every location is connected by exactly one bridge. So there are exactly 8 bridges. Also there are 8 bridge crossings.

Print-Crossings( $b_1,\ b_2, \ldots, b_n$ )

(Inputs are $n$ bridges $b_k$, $1 \le k \le n$. Each $b_k$ is given by its two end points $(w_i, e_j)$, $1 \le i, j \le n$. For every pair of bridges $b_k = (w_i, e_j)$ and $b_{k'} = (w_{i'}, e_{j'})$ the following holds: $k \ne k' \implies (i \ne i') \wedge (j \ne j')$. We say that two bridges $b_k = (w_i, e_j)$ and $b_{k'} = (w_{i'}, e_{j'})$ with $k \ne k'$ cross provided either $(i < i') \wedge (j > j')$ or $(i > i') \wedge (j < j')$ holds. This function prints all such crossings in the input.)

1. ***for*** $k \leftarrow 1$ ***to*** $n$ ***do***
2. $\quad (w_i, e_j) \leftarrow b_k$
3. $\quad$ ***for*** $k' \leftarrow 1$ ***to*** $k - 1$ ***do***
4. $\quad\quad (w_{i'}, e_{j'}) \leftarrow b_{k'}$
5. $\quad\quad$ ***if*** $(i < i') \wedge (j > j')$ ***or*** $(i > i') \wedge (j < j')$ ***then***
6. $\quad\quad\quad$ ***print*** $k,\ k'$ $\quad$ {*bridges* $b_k$ *and* $b_{k'}$ *cross*}
7. ***return***

Figure 2: Print all bridge crossings.

Let $s_{n,k}$ = number of inputs of size $n$ for which line 6 is executed exactly $k$ times, and also let $f_{n,k} = \frac{s_{n,k}}{n!}$ be the fraction of all possible inputs if size $n$ each of which results in precisely $k$

executions of line 6. Then clearly, our required average $A_n$ and its variance $V_n$ are given by the following expressions.

$$A_n = \sum_k k f_{n,k} \quad \text{and} \quad V_n = \sum_k k^2 f_{n,k} - A_n^2$$

(*a*) [ **15 Points** ] Prove that for $n > 0$, $s_{n,k}$ can be described using the following recurrence relation.

$$s_{n,k} = \begin{cases} 0 & \text{if } k < 0 \ \vee \ k > \binom{n}{2}, \\ 1 & \text{if } k = 0 \ \vee \ k = \binom{n}{2}, \\ \sum_{i=0}^{n-1} s_{n-1,k-i} & \text{otherwise.} \end{cases}$$

(*b*) [ **15 Points** ] Consider the following generating function for $s_{n,k}$'s with $n > 0$.

$$S_n(z) = s_{n,0} + s_{n,1}z + s_{n,2}z^2 + \ldots + s_{n,k}z^k + \ldots + s_{n,\binom{n}{2}} z^{\binom{n}{2}}$$

Use results from part (*a*) to show that for $n > 0$,

$$S_n(z) = \begin{cases} 1 & \text{if } n = 1, \\ \left(1 + z + z^2 + \ldots + z^{n-1}\right) S_{n-1}(z) & \text{otherwise.} \end{cases}$$

(*c*) [ **10 Points** ] Solve the recurrence from part (*b*) to show that $S_n(z) = \frac{1}{(1-z)^n} \prod_{k=1}^{n} \left(1 - z^k\right)$.

(*d*) [ **30 Points** ] Let $F_n(z) = \frac{1}{n!} S_n(z)$. Use your results from part (*c*) to show that

$$A_n = F_n'(1) = \frac{1}{2}\binom{n}{2}$$

$$\text{and} \quad V_n = F_n''(1) + F_n'(1) - \left(F_n'(1)\right)^2 = \frac{2n+5}{36}\binom{n}{2}.$$

**Task 2. [ 110 Points ] The Sheap**

A *Scanning Heap* or *Sheap* is a priority queue that supports *Insert*, *Delete* and *Extract-Min* operations. An *Insert*$(x, k_x)$ operation inserts the item $x$ with key $k_x$ into the queue assuming that $x$ does not already exist in the queue. A *Delete*$(x)$ operation deletes item $x$ from the queue if it exists, and an *Extract-Min*() operation retrieves and deletes an item with the minimum key from the queue. We assume for simplicity that all keys in the data structure are distinct.

A sheap on $N$ items consists of $r = 1 + \lceil \log_2 N \rceil$ levels. For $0 \le i \le r - 1$, level $i$ consists of a *data buffer* $D_i$ and an *operations buffer* $O_i$. Each item in $D_i$ is of the form $(x, k_x)$, where $x$ is the item id and $k_x$ is its key. Each operation in $O_i$ is augmented with a time stamp indicating the time of its insertion into the data structure.

At any time, the following invariants are maintained.

INVARIANT 1

(*a*) *Each* $D_i$ $(0 \le i < r)$ *contains at most* $2^i$ *items.*

INVARIANT 2

(*a*) *Key of every item in* $D_i$ $(0 \le i < r-1)$ *is no larger than the key of any item in* $D_{i+1}$.

(*b*) *All operations applicable to* $D_i$ $(0 \le i < r-1)$ *that are not yet applied, reside in* $O_0, O_1, \ldots, O_i$.

INVARIANT 3

(*a*) *Items in each* $D_i$ *are kept sorted in ascending order by item id.*

(*b*) *Operations in* $O_0$ *are kept sorted in ascending order by time stamp.*

(*c*) *For* $0 < i < r$, *operations in each* $O_i$ *are divided into (a constant number of) segments with updates in each segment sorted in ascending order by item id and time stamp.*

All buffers are initially empty.

The pseudocodes for all data structural operations are given in Figures 3 and 4.

A *Insert*( $x$, $k_x$ ) operation is performed by the INSERT function which inserts the entry $\langle$ *Insert*, $x$, $k_x$, $t$ $\rangle$ into $O_0$, where $t$ is the current time stamp. A *Delete*( $x$ ) operation is performed similarly by the DELETE function which inserts the entry $\langle$ *Delete*, $x$, $t$ $\rangle$ into $O_0$. In both cases further processing is deferred to the next *Extract-Min* operation.

The EXTRACT-MIN function executes an *Extract-Min* operation by first calling the FIND-MIN function to find the item with the minimum key in the data structure, and then calling the DELETE function to delete this item.

The FIND-MIN function first sorts the operations in $O_0$ by item id (primary) and time stamp (secondary). Then it finds the shallowest data buffer $D_k$ that is left non-empty after applying the operations in $O_k$ (by calling EXECUTE-OPS). The elements left in $D_k$ are distributed to the shallowest data buffers by calling REDISTRIBUTE.

When the EXECUTE-OPS function is called with parameter $i$, it applies the operations in $O_i$ on the items of $D_i$, and empties $O_i$ by moving the operations from $O_i$ to $O_{i+1}$. It also moves any overflowing items from $D_i$ to $O_{i+1}$ as *Insert* operations.

Now your task is to answer the following questions.

(*a*) [ **15 Points** ] Prove that the *Extract-Min* function correctly returns the item with smallest key in the data structure at the time of its execution.

(*b*) [ **15 Points** ] Explain how to implement the REDISTRIBUTE( $i$ ) function to run in $\Theta(|D_i|)$ time.

(*c*) [ **15 Points** ] For a sequence of $N$ *Insert*, *Delete* and *Extract-Min* operations performed on a sheap, find the worst-case cost of each of those three types of operations.

(*d*) [ **15 Points** ] Prove that for $1 \le i \le r-1$, every empty $O_i$ receives batches of operations at most a constant number of times before $O_i$ is applied on $D_i$ and emptied again.

(*e*) [ **50 Points** ] Show that a sheap supports each operation (i.e., *Insert*, *Delete* and *Extract-Min*) in $\mathcal{O}(\log N)$ amortized time, where $N$ is the total number of operations performed on the sheap. Results from parts (*b*) and (*d*) will be useful in proving this part.

Insert( $x$, $k_x$ )

(Inserts an item $x$ with key $k_x$ into the data structure assuming that the data structure does not already contain $x$.)

append the operation to $O_0$ augmented with current time stamp maintaining invariant 3(*b*)

Delete( $x$ )

(Delete the item $x$ from the data structure.)

append the operation to $O_0$ augmented with current time stamp maintaining invariant 3(*b*)

Extract-Min( )

(Extract the item with the smallest key from the data structure.)

sort the operations in $O_0$ in increasing order of item id (primary) and time stamp (secondary)
$i \leftarrow -1$
***repeat***
    $i \leftarrow i+1$
    Execute-Ops( $i$ ) $\{$*apply the operations in* $O_i$ *on the items in* $D_i\}$
***until*** ( $|D_i| > 0$ ) $\vee$ ( $i = r-1$ )
***if*** $|D_i| = 0$ ***then*** $\{$*the data structure has become empty*$\}$
    $(x, k_x) \leftarrow (\_\,, +\infty)$, $r \leftarrow 1$ $\{$*will return* $+\infty$ *as the minimum key*$\}$
***else*** $\{D_i$ *has the item with the smallest key in the entire data structure*$\}$
    $(x, k_x) \leftarrow$ the item with the smallest key in $D_i$
    remove $(x, k_x)$ from $D_i$ $\{D_i$ *now has at most* $2^i - 1$ *items*$\}$
    Redistribute( $i$ ) $\{$*redistribute the items in* $D_i$ *to the shallowest possible data buffers*$\}$
    $D_i \leftarrow \emptyset$ $\{D_i$ *has become empty*$\}$
***return*** $(x, k_x)$

Figure 3: Sheap operations.

EXECUTE-OPS( $i$ )

(Applies the operations in $O_i$ on the items in $D_i$, move remaining operations from $O_i$ to $O_{i+1}$ if $i < r-1$, and after executing the operations moves overflowing items from $D_i$ to $O_{i+1}$ as *Insert*s.

**Preconditions:** All invariants hold, and for $0 \leq j < i$, all $O_j$ are empty.

**Postconditions:** All invariants hold, and for $0 \leq j < i+1$, all $O_j$ are empty.)

1. merge the segments of $O_i$
2. ***if*** ( $|D_i| = 0$ ) $\wedge$ ( $i < r-1$ ) ***then*** {*if $i$ is not the last level and $D_i$ is empty*}
3.     empty $O_i$ by moving the contents of $O_i$ as a new segment of $O_{i+1}$
4. ***else***
5.     ***if*** $i = r-1$ ***then*** $k \leftarrow +\infty$
6.     ***else*** $k \leftarrow$ largest key in $D_i$
7.     scan $D_i$ and $O_i$ simultaneously, and for each *op* $\in O_i$ do: {*apply the operations in $O_i$ on $D_i$*}
8.         ***if*** *op* $=$ *Delete*( $x$ ) ***then*** remove any item $(x, k_x)$ from $D_i$, if exists
9.         ***if*** *op* $=$ *Insert*( $x,\ k_x$ ) $\wedge$ $k_x \leq k$ ***then*** copy $(x, k_x)$ to $D_i$
10.     ***if*** $i < r-1$ ***then*** {*move appropriate operations from $O_i$ to $O_{i+1}$*}
11.         copy each operation in $O_i$ that was not applied in steps 7–9 to $O_{i+1}$
12.     ***if*** $|D_i| > 2^i$ ***then*** {*restore invariant $1(a)$, if violated*}
13.         ***if*** $i = r-1$ ***then*** $r \leftarrow r+1$
14.         keep the $2^i$ items with the smallest $2^i$ keys in $D_i$,
            and move each remaining item $(x, k_x)$ to $O_{i+1}$ as *Insert*( $x,\ k_x$ )
15.         $O_i \leftarrow \emptyset$

REDISTRIBUTE( $i$ )

(Distributes the elements in $D_i$ to the shallowest data buffers maintaining invariants $1(a)$, $2(a)$ and $3(a)$. Let $D'$ denote the input $D_i$.

**Preconditions:** All invariants hold. All $D_j$ and $O_j$ with $0 \leq j \leq k$ are empty, where $k$ is the smallest integer such that $2^{k+1} - 1 \geq |D'|$. No key value in the data structure is smaller than any key value in $D'$.

**Postconditions:** All invariants hold. All operations buffers remain unchanged, but $\bigcup_{j=0}^{k} D_j = D'$.)

1. copy all items in $D_i$ to an initially empty temporary buffer $D'$ leaving $D_i$ empty
2. $j \leftarrow$ largest integer such that $2^j - 1 < |D'|$
3. ***while*** $j \geq 0$ ***do***
4.     move $|D'| - 2^j + 1$ items with the largest $|D'| - 2^j + 1$ keys from $D'$ to $D_j$ maintaining invariant $3(a)$
5.     $j \leftarrow j - 1$

Figure 4: Sheap helper functions.