Massachusetts Institute of Technology — Handout 12
6.046J/18.410J: Introduction to Algorithms — March 13, 2000
Professors Piotr Indyk and Madhu Sudan

Problem Set 4 Solutions

Problem 4-1. Improving Lower Bounds by Using Randomization Suppose you are given an array A of size n that contains either all zeros or $n/2$ zeros and $n/2$ ones, and your problem is to determine whether A contains any ones.

(a) Give a lower bound on the worst-case running time of any deterministic algorithm that solves this problem.

Any deterministic algorithm must examine at least $(n/2) + 1$ elements of the input array before deciding whether the array contains any zeros, in the worst case. To see this, consider a deterministic algorithm that looks at only n/2 elements. Run it on all zeros and note the elements it looks at. Now put ones in all the places it didn't look at and run it again. Since it is deterministic, it must look at the same places and come to the same conclusion. So it must be wrong on one of these inputs, which means that there is no (correct) deterministic algorithm that looks at only n/2 inputs. Since $(n/2)+1$ elements must be examined, the lower bound is $T(n) = \Omega(n)$.

(b) Give a randomized algorithm that runs in $O(1)$ time and gives the right answer with probability at least 3/4, where the probability is taken over the random choices of the algorithm. (That is, your algorithm is allowed to be wrong, but it cannot be wrong with probability greater that 1/4 on *any* input.) Prove your answer.

Randomly select two input elements. If both are 0, return "All zeros". Otherwise return "Half zeros, half ones". The algorithm clearly takes $O(1)$ time. If the input consists of all zeros, the algorithm is always correct. If the input consists of half zeros and half ones, then it makes a mistake only when both of the randomly chosen elements are 0. This occurs with probabilty $(1/2) \cdot (1/2) = 1/4$, assuming that we don't disallow choosing the same element twice. Hence on both inputs, the algorithm succeeds with probability at least 3/4.

(c) Design a randomized algorithm which never makes any errors. Can you do better than the deterministic case? If so, under what circumstances?

The following algorithm will never make any errors and run and $O(1)$ expected time when the input is half ones and half zeros and $O(n)$ expected time when the input is all zeros.

```
Repeat log_2 n times:
    Pick i at random in the range [1, ..., n].
    If A[i] = 1 return "half zeros, half ones"
For i ← 1 to n do
    If A[i] = 1 return "half zeros, half ones"
return "All ones"
```

When all of the input elements are 0, the algorithm will repeat the first loop $\log_2 n$ times, and then will execute the second loop, which has n iterations. In this case the running time will be $O(n)$.

If half of the inputs are 1, then the expected number of times that the algorithm executes the first loop is $O(1)$, since the probability that a randomly chosen element is a one is $1/2$. Let X be the number of iterations of the first loop. Then $E(X)$ is given by the recurrence $E(X) \leq 1 \cdot (1/2) + (1 + E(X)) \cdot (1/2)$, which solves to $E(X) \leq 2$. The reasoning here is that with probability $1/2$ there is only one iteration (because the first element seen is a one). If the first element is not a one, then after the first iteration we are starting over, and can again use $E(X)$ to indicate the number of remaining iterations. Another solution is to write out the formula for the expectation: $E(X) = \sum_{i=1}^{\log n} 1/2^i \leq 2$. If no ones are seen after $\log n$ iterations of the first loop, then the second loop is executed. This happens with probability $1/2^{\log n} = 1/n$. The total contribution of the second loop to the expected running time is thus $(1/n) \cdot O(n) = O(1)$.

There are lots of variations on this algorithm. For example, no harm is done by performing more iterations of the first loop, up to $O(n)$ iterations total. Another idea is to guarantee that no element is examined more than once. In this case, there is no need for a second loop. After $(n/2)+1$ elements have been examined, the right answer can be determined immediately. As in the first solution, in the case of half zeros and half ones, in each iteration of the first loop the probability of examining a one is at least $1/2$. One way to ensure that no element is seen twice is to swap each randomly selected element with the last element in the array, and then to decrease the length of the array.

Problem 4-2. Selection in linear time

The algorithm will work if the input elements are divided into groups of 7. The recursion is

$$T(n) \leq T(\left\lceil \frac{n}{7} \right\rceil) + T(\frac{5n}{7} + 8) + O(n) .$$

We wish to prove this by substitution. We can simplify our life by showing that the $\left\lceil \frac{n}{7} \right\rceil$ and the $\frac{5n}{7} + 8$ do not pose any problems. For $n \geq 70$, $\left\lceil \frac{n}{7} \right\rceil \leq \frac{1.1n}{7}$ and $\frac{5n}{7} + 8 \leq \frac{5.8n}{7}$.

The recurrence can now be simplified to

$$T(n) \leq T(\frac{1.1n}{7}) + T(\frac{5.8n}{7}) + O(n) .$$

Substitute $T(n) = cn$.

$$T(n) \leq \frac{1.1cn}{7} + \frac{5.8cn}{7} + O(n) .$$

$$T(n) \leq \frac{6.9cn}{7} + O(n) \ .$$

$$T(n) \leq cn \ .$$

The algorithm will not work if the input elements are divided into groups of 3. The recursion is

$$T(n) \leq T(\left\lceil \frac{n}{3} \right\rceil) + T(\frac{2n}{3} + 4) + O(n) \ .$$

For this recursion, if we draw a recursion tree, we will see that the first $\lfloor \log_3 n \rfloor$ levels of the recursion tree is full, and each of the first $\lfloor \log_3 n \rfloor$ level has $\Theta(n)$ work. Therefore, the total running time would be at least $\Omega(n \lg n)$.

Problem 4-3. Data Structures for Order Statistics

A data structure which will do what we what is defined as follows:

```
ELEMENT
    value /* the value of the data */
    Data /* pointer to the data structure below */
```

```
DATA
    max /* the maximum element, initialize to null */
    min /* the minimum element, initialize to null */
    A[1,...,n] /* an array from 1 to n initialized to zero */
```

To insert an element, we check that our element is not the new max or min, then we create a new element and increment the counter in our array from 1 to n.

INSERT$(a, Data)$
if $Data.max = null$
then $Data.max \leftarrow a$
if $Data.min = null$
then $Data.min \leftarrow a$
if $a > Data.max$
then $Data.max \leftarrow a$
else if $a < Data.min$
then $Data.min \leftarrow a$
$E \leftarrow$ new $Element$
$E.value \leftarrow a$
$Data.array[a] + +$
$E.Data \leftarrow Data$
$p \leftarrow ptr(E)$
return p

Each line in the algorithm can be executed in constant time and there are no loops so it will be $O(1)$.

In order to decrement a pointer we can subtract one from the value field in our element data structure. We also need to decrement the counter in the array at the old value and increment the counter for the value one less than that. The rest of it is preserving max and min. We check if the decremented value is less than min. If so, decrement min. If the undecremented value is greater than max and there are no more elements of that size, then decrement the max value as well.

DECREMENT(ptr)
if $ptr- > value \leq 1$
then return
$tmp \leftarrow ptr- > value$
$ptr- > value - -$
$ptr- > Data.Array[tmp] - -$
$ptr- > Data.Array[ptr- > value] + +$
if $tmp = ptr- > Data.min$
then $ptr- > Data.min - -$
if $tmp = ptr- > Data.max$
then if $ptr- > Data[tmp] \leq 0$
    $ptr- > Data.max - -$

Each line in the algorithm can be executed in constant time and there are no loops so it will be $O(1)$.

Due to the fact that we preserve the max and min properties in the data structure during Insert and Decrement, we can simply return their values in $O(1)$ time.

GETMAX(*Data*)
return *Data.max*

GETMIN(*Data*)
return *Data.min*

Sketch of correctness:

GetMin will always return the minimum value in the array because when an element is inserted it is compared to the minimum value and every time an item is decremented it is compared to the minimum value.

GetMax will always return the maximum value in the array because when an element is inserted it is compared to the maximum value and when the maximum item is decremented either there were other elements which had that value (in which case max remains the same) or there weren't (in which case max is decremented as well.

Alternately, this can be done as an array of size n of doubly linked lists. Each element of the list contains the value.

ELEMENT
value /* the value of the data */
next /* pointer to the next element (starts null) */
prev /* pointer to the previous element (starts null) */

Then there is an array A from 1 to n of elements whose next and prev values start null and whose values are 1 to n. There should also be pointers to max and min.

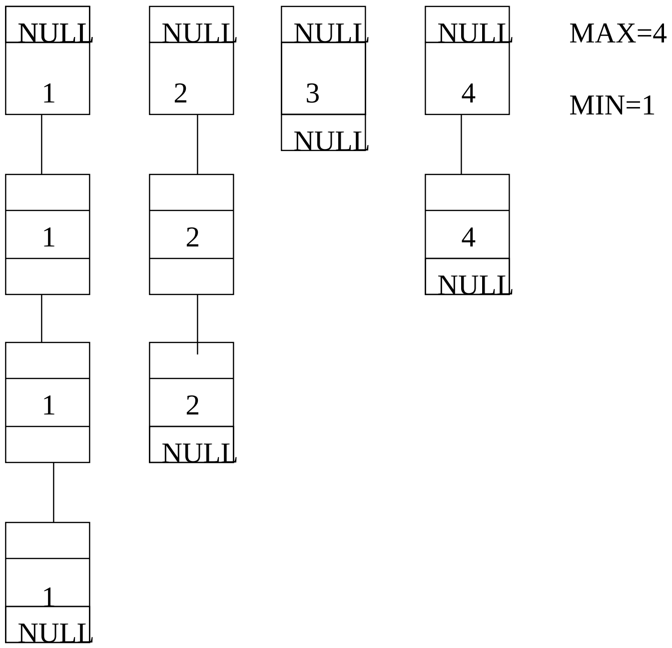

Figure 1: Doubly linked list structure example

INSERT(a, A)

if $A.max = null$
then $A.max \leftarrow a$
if $A.min = null$
then $A.min \leftarrow a$
if $a > A.max$
then $A.max \leftarrow a$
else if $a < A.min$
then $A.min \leftarrow a$
$E \leftarrow$ new $Element$
$E.value \leftarrow a$
$E.next \leftarrow A.next$
$E.prev \leftarrow A[a]$
return ptr(E);

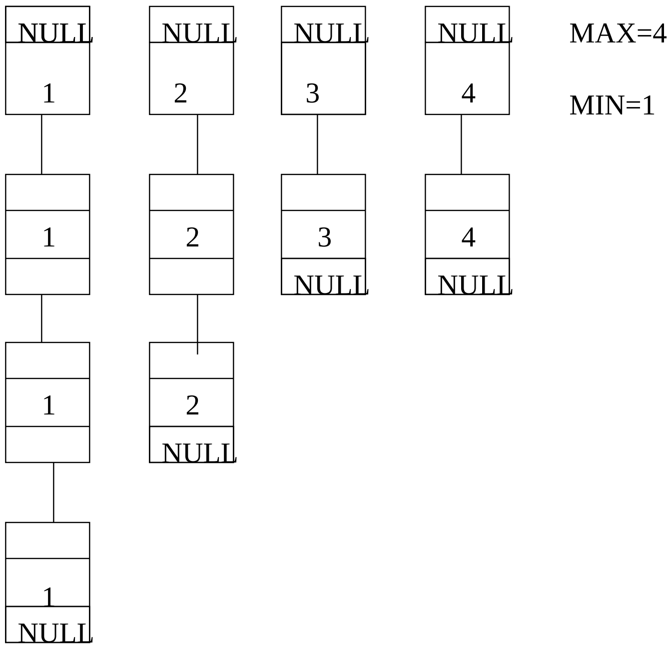

Figure 2: Insert(3,A)

DECREMENT(ptr)

if $ptr- > value \leq 1$
then return
$tmp \leftarrow ptr$
$tmp- > prev- > next \leftarrow temp- > next$
$tmp- > next- > prev \leftarrow temp- > prev$
$ptr- > value - -$
if $tmp- > value = ptr- > A.min$
then $ptr- > A.min - -$
if $tmp- > value = ptr- > A.max$
then if $A[tmp- > value]- > next = null$
    $ptr- > A.max - -$
Insert$(ptr- > value, A)$

GETMAX(A)

return $A.max$

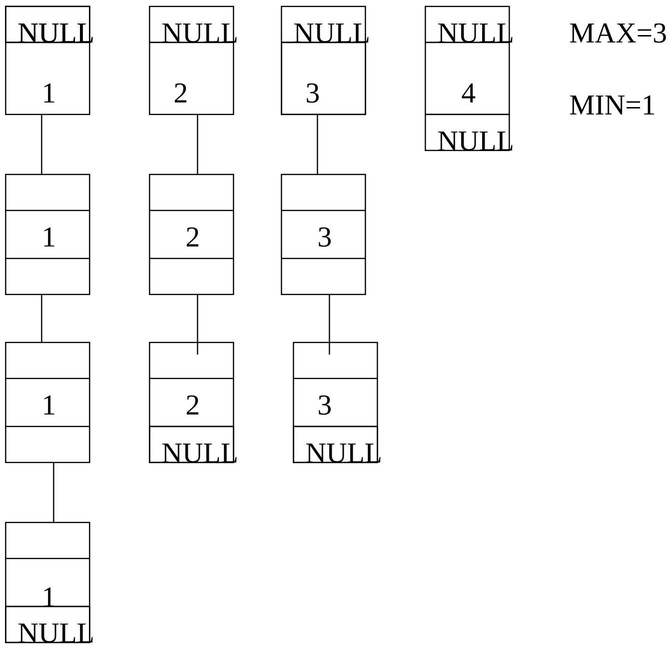

Figure 3: Decrement(ptr) where ptr points to the element at A[4]

GETMIN(A)
1 return $A.min$

Problem 4-4. Uniform Families of Hash Functions

Let $\mathcal{H}$ be a family of hash functions $h : X \mapsto \{0, \cdots, m-1\}$ such that for any $x \in X$ and $i \in \{0, \cdots, m-1\}$, the probability that $h(x) = i$ when h is taken randomly from $\mathcal{H}$ is exactly $1/m$. Is $\mathcal{H}$ a universal family? Explain.

No. Consider the following family $\mathcal{H}$ with the above properties:
$\mathcal{H} = \{h_1, \cdots, h_m\}$ where h_i maps all of X to the ith slot. Clearly, for any x and i, when we take h randomly the probability that $h(x) = i$ is exactly $1/m$. However, $\mathcal{H}$ is not universal as any two elements will always collide.