MATH 182: Homework #5

Professor Wotao Yin

Eric Chuu

UID: 604406828

Assignment: 2.4 # 2, 7, 16, 20, 26

February 22, 2017

Question 2.4.2

Criticize the following idea: To implement find the maximum in constant time, why not use a stack or queue, but keep track of the maximum value inserted so far, then return that value for find the maximum?

Solution

This implementation only performs well until the first time the maximum value is removed. Since we have only kept track of the maximum value and have not maintained the order of any of the other elements in the stack/queue, we would have to traverse the inserted elements to find the new maximum after removing the maximum. □

Question 2.4.7

The largest item in a heap must appear in position 1, and the second largest must be in position 2 or position 3. Give the list of positions in a heap of size 31 where the largest kth largest (i) can appear, and (ii) cannot appear, for $k = 2, 3, 4$ (assuming the values to be distinct).

Solution

Case $k = 2$. The second largest element can appear in position 2 or 3 of the heap, both of which are on the level directly beneath the root (position 1). This is because there is no specified order defined between nodes on the same level. It follows that the second largest element cannot appear on any level lower than the level below the root, so it cannot appear in position 1 or positions in the set $[4, 31]$.

Case $k = 3$. The third largest element can appear in the 1st level of the tree (root is at level 0), in position 2 or 3. As mentioned in the previous case, there is no strict ordering between nodes of the same level, so values in positions 4 and 5 can be greater than the value in position 3. Likewise, the values in positions 6 and 7 can be greater than the value in position 2. Thus the 3rd largest element can appear in positions $[2, 7]$, but it cannot appear in position 1 or positions in the set $[8, 31]$.

Case $k = 4$. The fourth largest element can appear in the 1st level, in positions 2 or 3, if we consider the heap represented by the array: $[0, 10, 9, \mathbf{6}, 8, 5, 4, 3]$, where 0 in position 0 of the array is not part of the heap. 6 is the fourth largest element in the heap, and it is in position 3. The fourth largest element can appear in the second level, in positions 4 through 7. We can apply the same logic as before and note that nodes in the third level (positions 8 to 15) can have values that are greater than nodes in the previous level, i.e., the value in positions 8, 9 can be greater than the values in positions 5, 6, 7. Thus, the fourth largest element can appear in positions in the set [2, 15], but cannot appear in position 1 or in the set $[16, 31]$. □

Question 2.4.16
For $n = 32$, give arrays of items that make heapsort use as many and as few compares as possible.

Solution
The case that would allow heapsort to use as few compares as possible is an input array with identical elements in every index of the array. In this case, when `sink()` is called, the `break` statement is guaranteed to be executed on the first iteration because the elements are all equal. This prevents traversing deeper down the tree and minimizes the number of compares.

The case that causes more compares is an input with arrays that are partially ordered, or sorted in increasing order: [0, 1, 2, 3, ..., 32], where element in position 0 is not part of the heap. This makes intuitive sense because the order of the array ensures that compares are made at every level of the tree since elements at the bottom of the tree are greater than those at the previous level. □

Problem 2.4.20
Prove that sink-based heap construction uses at most $2n$ compares and n exchanges.

Solution
We first show that sink-based heap construction uses at most n exchanges, where n is the number of elements in the heap. Suppose that the tree is perfect, with the last level of the tree completely populated, and has height k. The assumption that the tree is perfect is safe to make, since a tree that is fully populated on the last level naturally results in more compares. Then at each level $i \leq k$, there are 2^i nodes. Moreover, it's clear that for a node on the i-th level, heap construction uses $(k-i)$ exchanges in the worst case, i.e, the number of nodes away from the bottom of the tree it is. This happens when it needs to exchange with every single one of its descendants. Then we can express the number of exchanges in the worst case with the following sum

$$\sum_{i=0}^{k} 2^i(k-i) = k \cdot \sum_{i=0}^{k} 2^i - \sum_{i=0}^{k} i \cdot 2^i \tag{1}$$

In order to simplify this expression we first show that the following holds

$$\sum_{i=0}^{k} i \cdot a^i = \frac{a - (k+1) \cdot a^{k+1} + k \cdot a^{k+2}}{(a-1)^2} \tag{2}$$

We differentiate both sides with respect to a to get

$$\begin{aligned}
\text{LHS} &= \sum_{i=0}^{n} i \cdot a^{i-1} = \frac{1}{a} \sum_{i=0}^{n} i \cdot a^i \\
\text{RHS} &= \frac{(n+1)a^n(a-1) - (a^{n+1}-1)}{(a-1)^2} \\
\Rightarrow \sum_{i=0}^{n} i \cdot a^i &= \frac{(n+1)a^{n+1}(a-1) - (a^{n+2}-a)}{(a-1)^2} \\
&= \frac{(n+1)a^{n+2} - (n+1)a^{n+1} - a^{n+2} + a}{(a-1)^2} \\
&= \frac{na^{n+2} - (n+1)a^{n+1} + a}{(a-1)^2}
\end{aligned}$$

Thus, the identity in (2) holds, and we let $a = 2$, and plug the result back into equation (1) above,

$$\begin{aligned}
k \cdot \sum_{i=0}^{k} 2^i - \sum_{i=0}^{k} i \cdot 2^i &= k \cdot (2^{k+1} - 1) - (2 - (k+1)2^{k+1} + k \cdot 2^{k+2}) \\
&= k \cdot 2^{k+1} - k - 2 + (k+1) \cdot 2^{k+1} - k \cdot 2^{k+2} \\
&= 2^{k+1} - 2 - k \\
&\leq 2^{k+1} - 1 \\
&= n
\end{aligned}$$

Thus, it follows that the in the worst case, the number of exchanges used in sink-based heap construction is bounded by the n, the number of items in the heap.

To show that sink-based heap construction uses at most $2n$ compares, we consider the code used during heap-construction. At most 2 compares are made before each exchange in the `sink()` call. Then it's clear that since we use most n exchanges, then we also use at most $2n$ compares. □

Problem 2.4.26

Heap without exchanges. Because the `exchange()` primitive is used in the `sink` and `swim` operations, the items are loaded and stored twice as often as necessary. Give more efficient implementations that avoid this implementation, a la insertion sort.

Solution

We address the inefficiency that `exchange()` causes by storing the value that we want to sink and comparing it to its descendants (children, children of its children) until we find a value that is less than the original value, all the while shifting values up the heap. We finally put the original value into the tree as a child of the last value that was greater than it. This prevents unnecessary swaps before determining the final position of the value we are sinking. The code is given below. □

```
public class HeapSort{

   public static void sort(int[] a) {
      int n = a.length - 1;           // exclude element at position 0
      for (int k = n / 2; k >= 1; k--) { // heap-construction using sink
         betterSink(a, k, n);
      }
      while (n > 1) {                 // sort down
         exch(a, 1, n--);
         betterSink(a, 1, n);
      }
   }

   private static void betterSink(int[] a, int k, int n) {
      int target = a[k];   // store item to be sunk
      while (2 * k <= n) {
         int j = 2 * k;   // index of left child
         if (j < n && less(a, j, j + 1))
            j++;
         if (!(target < a[j]))
            break;       // if target >= child, then break -> heap-ordered
         a[k] = a[j];
         k = j;
      }
      a[k] = target;      // put target into final position
   }

   // swap elements in positions i, j
   private static void exch(int[] a, int i, int j) {
      int tmp = a[i];
      a[i] = a[j];
      a[j] = tmp;
   }

   // returns true if element at pos i less than element at pos j
   private static boolean less(int[] a, int i, int j) {
      return a[i] < a[j];
   }
}
```