The Bubble-Down Algorithm for Constructing Heaps

Logan Mayfield

April 1, 2014

1 Introduction

The bubble-down algorithm takes a semi-heap, a binary tree in which all nodes but the root satisfy the heap property, and produces a heap. It is from this outcome that bubble-down derives its other common name, *heapify*. The algorithm works By systematically swapping, or bubbling, the non-heap root value down the tree until a heap is formed and provides a $O(\lg n)$ means of turning an n node semi-heap into a proper heap.

2 The *bubble-down* Algorithm

Let T be a heap or semi-heap with root value $T.v$ and left and right sub-heaps $T.l$ and $T.r$. When the root value $T.v$ is less than or equal to the roots values of $T.l$ and $T.r$ then T is heap. If, however $T.v$ is greater than the root of one of it's child-heaps, then T is a semi-heap and bubble-down will swap $T.v$ with the smallest child root value. In doing so, that sub-heap might in turn because a semi-heap. To fix this, we can simply recursively bubble-down on that sub-tree. This leads to the following pseudocode for bubble-down

```
BUBBLE-DOWN(T)
1  min = T
2  for c = each child of T
3       if c.V > T.v
4            min = c
5  if min ≠ T
6       swap(T.v, min, v.)
7       BUBBLE-DOWN(min)
```

Figure 1: Pseudo-code for the *bubble-down* algorithm

When the heap/semi-heap T has no children, then we're dealing with the base case of *bubble-down* and no change is made to T. Otherwise we see that on line 2 we check each available child of T to see if it's root value is less than T and if it is, mark it as the *min*. If T is not the min, then we'll swap and recurse on line 5, otherwise T was the min and the algorithm can terminate without further recursion.

3 Analysis

Bubble-down works because the initial tree was a semi-heap, it's left and right sub-trees were heaps. Thus, in the case where T is smaller than the root value of its children, we can infer that it is smaller than all the nodes in the tree and the tree is, therefore, a heap. If we assume the standard array implementation of a heap, then the tree is always complete and balanced and access to the subtrees takes $O(1)$ time. In the worst case, bubble-down must swap the initial root all the way down to a leaf spot. This takes $O(lgn)$ time as finding the swap node and carrying out the swap takes $O(1)$ time.

4 Conclusions

Bubble-down is an efficient process for transforming a semi-heap into a heap. The presentation here focused on min-heaps, but it should be clear that the algorithm can easily be adapted for max-heaps. Given the easy array-based, balanced tree implementation of a heap, it's safe to assume the $O(\lg n)$ upper-bound in practice.

An efficient bubble-down procedure is a critical element of efficient heap construction and the heap-sort algorithm. By applying bubble-down from the deepest non-leaf nodes up to the root of an arbitrary balanced binary tree, one can construct a heap in $O(n)$ time. If the root values of that heap are then repeatedly removed, then the contents are the tree are given in sorted order and you've carried out heap-sort. So, bubble-down is not only useful in the creation and management of heaps, but it serves as the basis for efficient sorting as well.

5 Appendix: Iterative Bubble-Down

Here we provide an iterative, loop-based implementation of bubble-down. The looped code is nearly identical to the body of the recursive implementation. The main loop is driven by a boolean flag that indicates whether or not a swap occurred on the last iteration.

```
BUBBLE-DOWN(T)
 1  cur = T
 2  swapped =true
 3  while swapped
 4        swapped = false
 5        min = cur
 6        for c = each child of cur
 7              if c.V > T.v
 8                    min = c
 9        if min ≠ c
10              swap(T.v, min, v.)
11              swapped = true
```

Figure 2: An Iterative version of the *bubble-down* algorithm

When the tree T is empty or is a singleton tree node, then the inner loop will not execute, no swaps will occur and the algorithm will terminate without changing T. This is equivalent to the base-case of the recursion. By setting the flag *swapped* to false at the start of each loop and only setting it to true if a swap occurs, we ensure that in the case where the tree at *cur* is a proper heap, the algorithm will terminate. Thus, we see the underlying logic is equivalent to that of the recursive implementation. If follows then that the efficiency analysis is the same and that the iterative algorithm is also $O(\lg n)$