Algorithms I – supervision 5

James Wood

February 22, 2015

1 Hash tables

(a)

hash	values		
0	35	10	5
1	6		
2	2		
3	18	3	8
4			

(b)

hash	value
0	10
1	
2	2
3	3
4	
5	35
6	6
7	5
8	18
9	8

(c) (i) Every item in the hash table is of the form either `(key, value, next)` or `(prev, next)`, where `prev` and `next` are indices into the table. This can be represented by the algebraic data type `Entry k v i`, where `k` is the type of keys, `v` is the type of the values and `i` is the type of the hash function's output:

```
data Entry k v i = Item k v (Maybe i)
                 | Free (Maybe i) (Maybe i)
```

Then, given `table`, an array of this type; and `firstFree`, the index in `table` of the first element in the free list, `set` is implemented as follows:

```
def set(key, value):
    h = hash(key)
    case table[h]:
        Item k v i:
            if k == key:
                # Replace the existing value
                table[h] = Item k value i
            else:
                # Put the new item in the place of the old
                    one,
                # and put the old item in the first free slot
                table[h] = Item key value (Just firstFree)

                # Assuming we have more free spaces
                Free _ (Just n) = table[firstFree]
                table[firstFree] = Item k v i
                firstFree = n
        Free p n:
            # First entry for this hash value
            table[h] = Entry key value Nothing

            # Clean up free list
            case p:
                Just pi:
                    table[pi].next = n
                    case n:
                        Just ni:
                            table[ni].prev = p
                        Nothing:
                            # Do nothing
                Nothing:
                    # h == firstFree
                    # Assuming we have more free space
                    Just ni = n
                    firstFree = ni

                    table[ni].prev = Nothing
```

(ii)

hash	0	1	2	3	4	`prev`	`next`	`firstFree`
0			2, C	2, C	2, C		1	
1				12, T	12, T			
2		2, A	2, A	2, A	2, A		0	
3								←
4					5, Z			

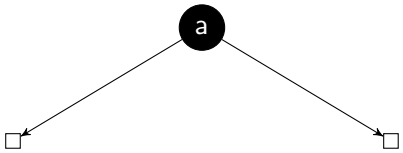
2 Red-black trees

(a)

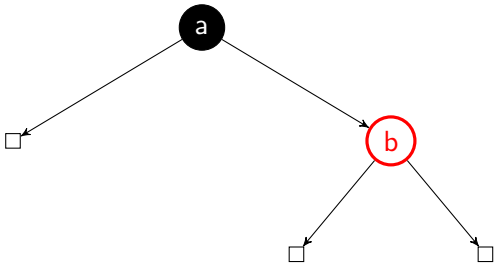
- Each node is either red or black.
- The root is black.
- All (empty) leaves are black.
- The child of any red node is black.
- There exists a number b such that any path from the root to a leaf contains exactly b black nodes.

The main advantage of a red-black tree over a binary search tree is that no path from root to leaf can be more than twice as long as any other path from root to leaf. This means that red-black trees can't become too unbalanced, so searching is guaranteed to run in logarithmic, rather than linear, time. The main disadvantages are that red-black trees are more complex to implement and have more overhead not measured by asymptotic complexity.

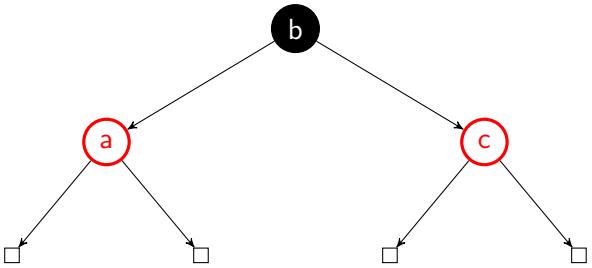
(b) 2-node $[a]$:



3-node $[a, b]$



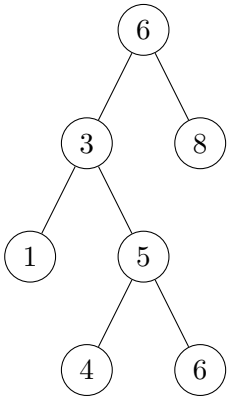
4-node $[a, b, c]$



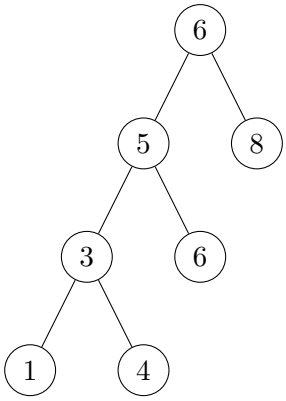
(c) The maximum possible number of nodes is achieved with a perfect tree in which all nodes are black. This has $2^{h+1} - 1$ nodes.

For $h = 2 \cdot k$, the red-black tree with the minimum number of nodes is achieved by having one root-to-leaf path consisting of alternating black and red nodes, with black nodes elsewhere. The alternating path has perfect black trees of decreasing height connected to it, giving a total of $\sum_{i \in [0..k]} 2 \cdot \left(1 + 2^i - 1\right)$, which evaluates to $2 \cdot \left(2^{k+1} - 1\right)$ as a geometric series.

(d) A rotation about a node of a binary search tree is an operation that moves one of the children of the node into its parent's place, moving the parent into the subtree opposite to the one the child came from. Then, the three subtrees of these two nodes are rettached in order. Consider this example:



To perform a leftward rotation on the 3 node, the 5 node is moved in place of the 3 node, and the 3 node and its left subtree are rettached under the 5 node. Then, the 4 subtree is moved to become the other subtree of the 3 node, and the 6 subtree becomes the right subtree of the 5 node again. This yields:



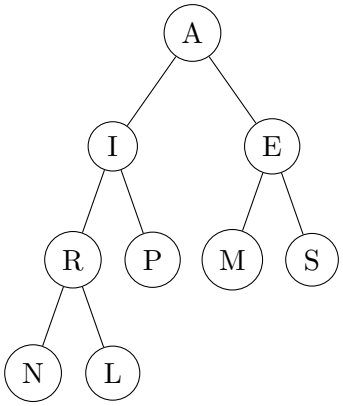
(e) To transform BST A into BST B, find the position of B's root's key in A, then rotate this node so that it becomes the root of A. At this point, the left (resp right) subtree of A will have the same keys as the left (resp right) subtree of B, by the BST property. So, apply this procedure to the two subtrees recursively, with leaves (or singleton trees) already the same and acting as a base case.

3 Heaps

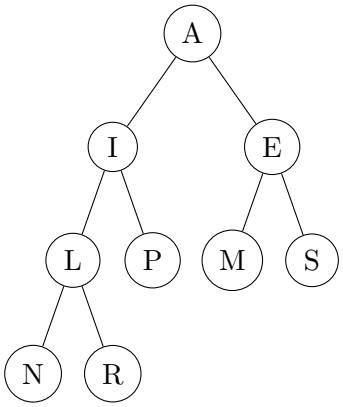
(a) In a min-heap, each node is less or equal to both of its children. To convert from tree to array representation, read off the elements breadth first, with some null placeholder in place of subtrees missing from the hypothetical full heap. The root has index 0, and a node with index i has children at $2 \cdot i + 1$ and $2 \cdot i + 2$.

(b) When an array is sorted into ascending order, every element is less than or equal to every following element. When the array is interpreted as a tree, any element's children always follow it, and thus in a sorted array, any element's children are less or equal to it. So the heap property is satisfied.

(c) `N` and `L` are children of `R`, but are smaller, so the array does not represent a min-heap. It is turned into a min-heap as follows:

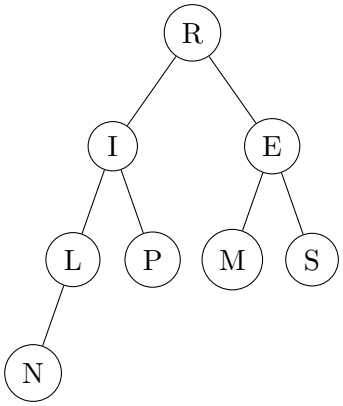


Start with the last element with children. If it is larger than any of its children, swap it with its smallest child, then repeat the comparison and possible swap in its new location.

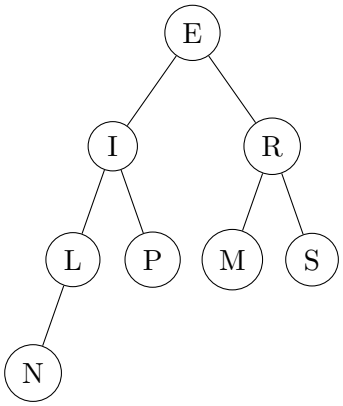


Then, move onto the previous element. In this case, all previous elements are in a suitable place, so there are no more intermediate steps to show.

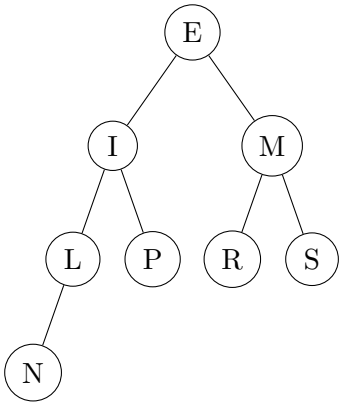
(d) First, the root node is removed and replaced by the last node.



Then, the node that is currently the root is swapped with its smallest child until it is smaller than both of its children.



`R` needs to be moved again.



The min-heap property is restored.

(e) Heapsort needs linear time to check that the array it is given forms a min-heap. Then, the element promoted to the top on the `extractMin` steps will always be the largest element, and thus will always need to go to the bottom of the heap, taking logarithmic time on each call. Hence, the algorithm will run in $\Theta\ (n \cdot \log n)$ time.

(f) `heapify` is known to to run in linear time. Then, when an element is promoted as part of `extractMin`,

4 Sorting and searching

(a) (i) Put all of the items into a priority queue, where the priority of each element is its value. Then, dequeue k elements.

Using the heap implementation of the priority queue, building the priority queue takes $O\, n$ time. Then, dequeueing takes $O\ (k \cdot \log n)$ time, since it involves m oving k elements down the height of the tree. This gives worst case complexity $O\ (n + k \cdot \log n)$

(ii) Assume that $n \geq k$. If $n = k$, return the given array. Otherwise, compare all of the elements of the array to a pivot element, putting lesser elements to the left, the pivot and equal elements after them and greater elements after those. Let l be the number of lesser elements, e be the number of elements that are or are equal to the pivot and g be the number of greater elements, so that $l + e + g = n$. If $k \in [0..l]$, repeat the process on the lesser elements. If $k \in (l..l + e]$, return the first k elements currently in the array. Otherwise, in which case $k \in (l + e..n]$, return the lesser and equal elements along with the smallest $k - l - e$ larger elements.

The worst cases occur when the pivot at each step is an extreme value. Then, the complexity is $O\ (\max(k, n - k))$. However, as with calculating order statistics, the pivot is usually not extreme, giving us the better average case of $\Theta\ (\log n)$.

(b) Sum the length fields, then pick a random number in the range $[0..s)$, where s is the sum (using `random` appropriately). Then, set t to s and iterate through the slots of the hash table, taking the length of the slot off t each time. When the length exceeds t, pick the tth item in the chain of the current slot. This takes $O\ (m + L)$ time, since it relies on traversing the slots of the hash table, then traversing a chain.

5 Bogosort

(a) In the best case, the list is already sorted, so the loop terminates as soon as `is_sorted` returns. This takes Θn time.

Any shuffling of the list has a probability of order $1/n!$ of being sorted. Each call to `shuffle` and `is_sorted` takes linear time. Hence, on average, bogosort will take $\Theta\ (n! \cdot n)$, or simply $\Theta\ (n!)$, time.

Bogosort is not guaranteed to terminate, so has undefined worst case complexity.

(b) Bogosort is unstable because there are no guarantees based on the order of the input list on the order of the output list.