Data Structures

Mengxi Wu (mw4355@nyu.edu)

September 27, 2020

1 Binary Search Tree

inary Search Tree is a node-based binary tree data structure which has the following properties:
The left subtree of a node contains only nodes with keys lesser than the node's key.
The right subtree of a node contains only nodes with keys greater than the node's key.
The left and right subtree each must also be a binary search tree.

2 AVL Tree

AVL Tree Deletion
http://www.mathcs.emory.edu/ cheung/Courses/323/Syllabus/Trees/AVL-delete.html

3 Build Max Heap

We can use MAX-HEAPIFY in a bottom up manner to covert array $A[1...n]$ into a max-heap, where $n = A.length$. First we need to prove a prerequisite: $A[\lfloor \frac{n}{2} \rfloor + 1...n]$ are all leaf nodes. According to the parent and child index relation, the last element has child has index $\lfloor \frac{n}{2} \rfloor$. Why not $\lceil \frac{n}{2} \rceil$? When n is odd, $\lceil \frac{n}{2} \rceil = \frac{n+1}{2}$, its parent index exceeds n. From index $\lfloor \frac{n}{2} \rfloor + 1$ to n are all leaves.

Initialization: Prior to the loop, for node with index $\lfloor \frac{n}{2} \rfloor + 1$, $\lfloor \frac{n}{2} \rfloor + 2...n$, the node is leaf so it is a trivial valid max-heap, which satisfy the input requirement of MAX-HEAPIFY.
Maintenance: The children of node i are larger than i. By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call MAX-HEAPIFY(A, i) to make node i a max-heap root. Moreover, the MAX-HEAPIFY call preserves the property that nodes i+1, i+2...n are all roots of max-heaps. Decrementing i in the for loop update reestablishes the loop invariant for the next iteration.
Termination: At termination, i = 0. By the loop invariant, each node 1,2...n is the root of a max-heap.

The runtime for step 1 is $O(1)$. For a lose upper bound, the for iteration happens $\lfloor \frac{n}{2} \rfloor$ times and each time calls MAX-HEAPIFY once. MAX-HEAPIFY takes $O(\log n)$. Thus, a lose upper bound is $O(n \log n)$. However, we can do better. Our tighter analysis relies on the properties that an n-element heap has height $\lfloor \log n \rfloor$ and at most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes at height h. The second fact can be proved by induction. Let n_h denote the number of nodes at height h. When $h = 0$, from previous proof, we from nodes from index $\lfloor \frac{n}{2} \rfloor + 1$ to n are leaves. Thus, at $h = 0$, we have $n_0 = \lceil \frac{n}{2} \rceil$ nodes. The base case holds. Suppose it holds for h. We know nodes at height h is the children of node at height $h+1$. if n_h is even, $n_{h+1} = \frac{n_h}{2} = \lceil \frac{n_h}{2} \rceil$. If n_h is odd, $n_{h+1} = \lfloor \frac{n_h}{2} \rfloor + 1 = \lceil \frac{n_h}{2} \rceil$.

$$n_{h+1} = \lceil \frac{n_h}{2} \rceil \leq \lceil \frac{1}{2} \cdot \lceil \frac{n}{2^{h+1}} \rceil \rceil = \lceil \frac{n}{2^{h+2}} \rceil$$

. Thus, we can have,

$$T(n) = \sum_{h=1}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil \cdot O(h) \leq \sum_{h=1}^{\lfloor \log n \rfloor} \frac{n}{2^h} \cdot O(h)$$

$$T(n) = O(n \cdot \sum_{h=1}^{\lfloor \log n \rfloor} \frac{h}{2^h}) = O(n \cdot \sum_{h=0}^{\infty} \frac{h}{2^h})$$

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2$$

Therefore, the tighter bound for BUILD-MAX-HEAP is $T(n) = O(2n) = O(n)$.

4 Heap Sort

The heapsort algorithm starts by using BUILD-MAX-HEAP to build a max-heap on the input array A[]1...n], where $n = A.length$. Since the maximum element of the array is stored at the root $A[1]$, we can put it into its correct final position by exchanging it with $A[n]$. Then, we discard nth node from the heap since it is at the right position. We can do so by simply decrementing A.heap-size-1. The new root might violate the max-heap property though its children remain the max-heap, so we can call MAX-HEAPIFY(A,1), which makes a $A[1...n-1]$ a valid max-heap. The heapsort algorithm then repeats this process for the max-heap of size n - 1 down to a heap of size 2. Build-MAX-HEAP takes $O(n)$. The for iterations take $n-1$ times and each time calls MAX-HEAPIFY. Thus, the runtime $T(n) = O(n) + O(n\log(n)) = O(n \log n)$.

5 Priority Queue

A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key. A max-priority queue supports the following operations:
INSERT(S, x): inserts the element x into the set S
MAXIMUM(S): returns the element of S with the largest key
EXTRACT-MAX(S): removes and returns the element of S with the largest key
INCREASE-KEY(S, x, k): increases the value of element x's key to the new value k

The runtime for HEAP-MAXIMUM is $O(1)$.

Swap the first (largest element) with the element at the end of the array. Then decrementing the heap-size. The call MAX-HEAPIFY to keep the heap property. The running time of HEAP-EXTRACT-MAX is $O(\log n)$, since it performs only a constant amount of work on top of the $O(\log n)$ for MAX-HEAPIFY.

The procedure first updates the key of element $A[i]$ to its new value. Because increasing the key of $A[i]$ might violate the max-heap property. Then, we compare $A[i]$ with its parent to check whether violate. If it violates, swap the key of parent with $A[i]$ and update i until we find a proper place for the new key. The running time of HEAP-INCREASE-KEY on an n-element heap is $O(\log n)$, since the path traced from the node updated in line 3 to the root has length $O(\log n)$.

The running time of MAX-HEAP-INSERT on an n-element heap is $O(\log n)$.