Intro Algo Chapter 6

Mengxi Wu (mw4355@nyu.edu)

March 7, 2020

1 Heap

Heap data structure is an array object we can view as a complete binary tree. A complete binary tree is a binary tree that is completely filled except possibly the lowest level, which is filled from the left. It is represented by an array A. The array has two attributes: $A.length$ and $A.heap-size$. Only $0 \leq A.heap-size \leq A.length$ are valid elements of the heap. Given index i in the array: $PARENT(i) = \lfloor \frac{i}{2} \rfloor$, $LEFT(i) = 2i$, $RIGHT(i) = 2i+1$. A heap holds heap property. In max-heap, for every node i other than the root $A[PARENT(i)] \geq A[i]$. For the heap sort algorithm, we use max-heaps.

The height of a node in a heap is the number of edges on the longest path from the node to a leaf. The height of the heap is the height of its root. A heap with n elements, its height is $\Theta(\log n)$. Suppose the heap has height h. The number of nodes from height 1 to height h is $2^h - 1$. The number of nodes in height 0 is at least 1 and at most 2^h.

$$2^h \leq n \leq 2^{h+1} - 1$$

$$\log 2^h \leq \log n \leq \log 2^{h+1} - 1 < \log 2^{h+1}$$

$$h \leq \log n < h + 1$$

The height must be an integer, so the height is $\lfloor \log n \rfloor$.

2 Max Heapify

To maintain heap's property we use a function call MAX-HEAPIFY. It takes inputs an array A and index i. It assumes Left-Subtree(i) and Right-Subtree(i) are valid max-heap. $A[i]$ may violate the max-heap property. MAX-HEAPIFY lets $A[i]$ find its proper position in the heap, so that the subtree roots at index i can be a valid max-heap.

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4       largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7       largest = r
 8  if largest ≠ i
 9       exchange A[i] with A[largest]
10       MAX-HEAPIFY(A, largest)
```

The procedure of the algorithm is quite obvious. Step 1-7 determines the largest element among root i, its left and right children. It stores the index of the largest one in $largest$. If $largest \neq i$, rearranging the positions is needed. We swap $A[largest]$ and $A[i]$. Now, the subtree roots at $largest$ contains value $A[i]$.

It may violates the max-heap property so we need to call MAX-HEAPIFY on subtree roots at *largest*.

The runtime for step 1-7 is $O(1)$. The nodes in the subtree roots in the subtree roots at *largest* is bounded by $\frac{2n}{3}$. It can be proved as follow. First, thinking about the worst case. We want the left subtree contains as many nodes as possible. The worst case is the last level is half full (all nodes in last level is on the left subtree). n= 1 + number of nodes in left subtree + number of nodes in right subtree. Suppose right tree has height h, then the left subtree has height $h+1$. Number of nodes in right subtree $= 2^{h+1} - 1$. Number of nodes in the left subtree $= 2^{h+2} - 1$.

$$n = 1 + 2^{h+2} - 1 + 2^{h+1} - 1$$

$$n = 3 \cdot 2^{h+1} - 1$$

$$2^{h+1} = \frac{n+1}{3}$$

$$2^{h+2} - 1 = \frac{2n+2}{3} - 1 = \frac{2n-1}{3} < \frac{2n}{3}$$

Then, the runtime $T(n)$ for MAX-HEAPIFY is $T(n) \leq T(\frac{2n}{3}) + \Theta(1)$. $T(n) = 1 \cdot T(\frac{n}{\frac{3}{2}}) + \Theta(1)$. By Master Theorem, $\Theta(n^{\log_{\frac{3}{2}} 1}) = \Theta(1)$, then $T(n) = O(\log n)$ or more strictly $T(n) = \Theta(\log n)$.

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.

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

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

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

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

```
HEAP-MAXIMUM(A)
1  return A[1]
```

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

```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2      error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

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.

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2      error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5      exchange A[i] with A[PARENT(i)]
6      i = PARENT(i)
```

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 $O(\log n)$.

```
MAX-HEAP-INSERT(A, key)
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = −∞
3  HEAP-INCREASE-KEY(A, A.heap-size, key)
```

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