Max-Min Heap

Mengxi Wu (mw4355@nyu.edu)

March 7, 2020

1 Find maximum and minimum

```
Find-Maximum($A$)
1 return $A[1]$
```

According to the definition, root at level 0 which is an even level, so it will be the maximum among the keys of the whole tree. The worst-case running time is $O(1)$.

```
Find-Minimum($A$)
1 if $A[2] < A[3]$ then
2     return $A[2]$
3 return $A[3]$
```

According to the definition, $A[2]$ and $A[3]$ at level 1 which is an odd level, so $A[2]$, $A[3]$ will be the minimum of the subtree tree rooted at them respectively. The minimum of the whole tree will be the minimum among $A[2]$ and $A[3]$. Every step in the Find-Minimum algorithm takes constant time, so the worst-case running time is $O(1)$.

2 Insert

First, we build a helper function call IS-VIOLATE to check whether the position of the element at index i is proper. The IS-VIOLATE takes 4 inputs.
A : the heap
i : the index of the inserted key
j: the index of root of the subtree where the key is in
$level$: the level in the heap where the root with index j is at
The algorithm compares the key with root to check whether the max-min heap property is violated. If the property is violated, it returns true; otherwise, returns false.

```
IS-Violated($A, i, j, level$)
1 if $level$ is even then
2     if $A[i] > A[j]$ then
3         return $True$
4 if $level$ is odd then
5     if $A[i] < A[j]$ then
6         return $True$
7 return $False$
```

Then, we build INSERT(A, key). Let n denote the size of the heap. We first append the key at the position $n+1$. Second, we use IS-VOILATED function to check whether the key and the root of the subtree that contain the key violate the max-min heap property. If they violate, we exchange their values, and update the index of the key. Repeat the second step for all the roots of the subtrees that contain the key.

```
INSERT($A, key$)
1  $A[n+1] = key$
2  $i = n+1$
3  $j = \text{PARENT}(i)$
4  $level = \lfloor \log(n+1) \rfloor - 1$
5  while $level \geq 0$ do
6      if IS-VIOLATED$(A, i, j, level)$ then
7          exchange $A[i]$ and $A[j]$
8          $i = j$
9      $j = \text{PARENT}(j)$
10     $level = level - 1$
```

Steps 1-4 takes constant times. The while loop repeats $\lfloor \log(n+1) \rfloor - 1$ times. IS-VOILATED and steps 7-10 takes constant time. Therefore, the worst-case running time of INSERT(A, key) is $O(\log n)$.

3 Heapify

First, we build two helper functions, SWAP-WITH-CHILDREN and SWAP-WITH-GRANDCHILDREN. SWAP-WITH-CHILDREN checks whether the root and its children violate the property. If they violate, then rearrange the positions. Similarly, SWAP-WITH-GRANDCHILDREN checks whether the root and its grandchildren violate the property. If they violate, then rearrange the positions.

```
SWAP-WITH-CHILDREN($A, i$)
1  $level = \lfloor \log i \rfloor$
2  if $level$ is even then
3      $max$ = index of the largest child of $i$
4      if $A[max] > A[i]$ then
5          exchange $A[max]$ with $A[i]$
6  if $level$ is odd then
7      min = index of the smallest child of $i$
8      if $A[min] < A[i]$ then
9          exchange $A[min]$ with $A[i]$
10 return $A$
```

```
SWAP-WITH-GRANDCHILDREN$(A, i)$
 1  $level = \lfloor \log i \rfloor$
 2  if $level$ is even then
 3      $max$ = index of the largest grandchild of $i$
 4      if $A[max] > A[i]$ then
 5          exchange $A[max]$ with $A[i]$
 6          $newRoot = max$
 7  if $level$ is odd then
 8      min = index of the smallest grandchild of $i$
 9      if $A[min] < A[i]$ then
10          exchange $A[min]$ with $A[i]$
11          $newRoot = min$
12  return $A$, $newRoot$
```

Then, we can build HEALPIFY. The idea is first comparing the root with its two children. If the property is violated, we rearrange the positions. After this operation, the left subtree and right subtree are still valid max-min heaps and the relation between root and its children is also valid. Second, we compare the root with its 4 grandchildren. If the root at a min level, its grandchildren will also be at a min level. The minimum among the grandchildren will be the minimum value in the left and right subtrees. It is similar for root at a max level. Thus, checking whether the root and its grandchildren violate the property and rearranging the positions if it is violated will let root and its grandchildren satisfy the property. After the first and second operations we can ensure the root, its children and its grandchildren all satisfy the property. The node index by the *newRoot*, however, now has the original value $A[i]$, and thus the subtree rooted at *newRoot* might violate the property. Consequently, we call HEAPIFY recursively on that subtree.

```
HEAPIFY$(A, i)$
1  $A$ = SWAP-WITH-CHILDREN$(A, i)$
2  $A$, $newRoot$ = SWAP-WITH-GRANDCHILDREN$(A, i)$
3  HEAPIFY$(A, newRoot)$
```

In SWAP-WITH-CHILDREN, since each node has at most 2 children, getting the index of the largest or smallest children takes constant time. The other steps also takes constant time, so SWAP-WITH-CHILDREN takes $\Theta(1)$. In SWAP-WITH-GRANDCHILDREN, since each node has at most 4 grandchildren, getting the index of the largest or smallest grandchildren takes constant time. The other steps also takes constant time, so SWAP-WITH-GRANDCHILDREN takes $\Theta(1)$ as well. For a heap at root i with n elements, the children's subtrees each have size at most $\frac{2n}{3}$ nodes (the worst case occurs when the bottom level of the tree is exactly half full). Thus, let $T(n)$ denote the running time of the HEAPIFY; we have $T(n) = T(\frac{2n}{3}) + \Theta(1)$. By the second case of master theorem, we have the running time of HEAPIFY $T(n) = O(\log n)$.

4 Build Heap

We can use HEAPIFY in a bottom-up manner to convert the $A[1...n]$, where n =size of the heap. 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. Thus, from index $\lfloor \frac{n}{2} \rfloor + 1$ to n are all leaves. All the leaves are valid heaps, so we just need to apply HEAPIFY from the first parent node.

```
BUILD-HEAP(A, n)
1  for $i = \lfloor \frac{n}{2} \rfloor$ downto 1 do
2      HEAPIFY(A, i)
```

Our running time 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 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, $n_{h+1} = \lfloor \frac{n_h}{2} \rfloor + 1 = \lceil \frac{n_h}{2} \rceil$.

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

. Let $T(n)$ donate the running time for BUILD-HEAP.

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

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

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

Therefore, the running time for BUILD-HEAP is $T(n) = O(2n) = O(n)$.

5 Extract maximum and minimum

The algorithms are shown as below. Let n denote the size of the heap in running time analysis.

```
EXTRACT-MAX(A)
1  $max = A[1]$
2  $A[1] = A[A.$heap-size$]$
3  $A.$heap-size $= A.$heap-size - 1
4  HEAPIFY(A, 1)
5  return $max$
```

Steps 1-3 take $O(1)$. HEAPIFY takes $O(\log n)$. The total running time for EXTRACT-MAX is $O(\log n)$.

```
EXTRACT-MIN($A$)
 1  if $A[2] < A[3]$ then
 2      $min = A[2]$
 3      $A[2] = A[A.\text{heap-size}]$
 4      $newRoot = 2$
 5  else
 6      $min = A[3]$
 7      $A[3] = A[A.\text{heap-size}]$
 8      $newRoot = 3$
 9  $A$.heap-size = $A$.heap-size - 1
10  HEAPIFY($A, newRoot$)
11  return $min$
```

Steps 1-9 take $O(1)$. HEAPIFY takes $O(\log n)$. The total running time for EXTRACT-MIN is $O(\log n)$.