Nafis Abeer

EC330 HW 5

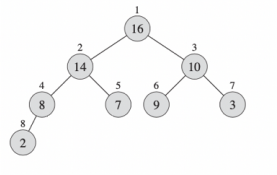
```
PARENT(i)
1  return ⌊i/2⌋

LEFT(i)
1  return 2i

RIGHT(i)
1  return 2i + 1
```

```
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]
```

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

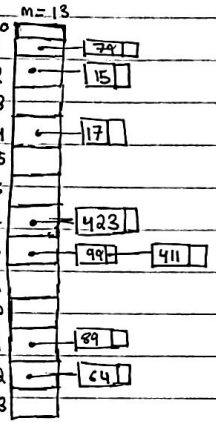


1) Max-heap: the value of each node is less than or equal to the value of its parent, with the maximum value element at the root.
    - I will use the above pseudo-code to describe how we can build a heap out of 8 numbers using only 8 comparisons. For the purposes of the images, I am replacing *{n1, n2, … n8}* with *A[1], A[2]… A[8]*. The idea is still the same, an array with 8 indices, starting with 1, holding 8 numbers.
    - The *Max-Heapify* function takes an index (parent) and sets twice its value as its left child and twice its value plus 1 as its right child. If the index that we are working on is 3, its left child is index 6 and its right child is index 7. The function then performs (1) comparison between the left child and the parent. If the left child's value is bigger it sets a new variable, *largest*, as the number of the left index. *Largest* only holds the number of the index not the value it contains. If the left child's value happens to be smaller, *largest* will be set to the original index itself. The function then performs another (2) comparison between the right child and the index that contains the larger value, out of the parent and the left child. If the right child's value is bigger than the value contained by index *largest*, right child is declared the new *largest* index. The function then checks if the original index is the same as the *largest* index. If it is, nothing happens; or else the value contained by *largest* is swapped with the value contained by the original index. At the conclusion of *Max-Heapify* (*A, i*)*, A[i]* contains the largest value out of *A[i], A[i*2]* and *A[i*2 + 1]* with the help of (2) comparisons.
    - For our purposes since we are working with an array with a length of 8, the *for loop* in line 2 of *Build-Max-Heap* will run from 4 down to 1. That means we will *Max-Heapify A[4]* followed by *A[3]* then *A[2]* and finally *A[1]* (also known as the root of the heap).
    - After the first iteration of the *for loop* in line 2 of *Build-Max-Heap,* the parent *A[4]* contains the larger value out of *A[4]* and *A[8]*. Two comparisons have taken place. After the second iteration, parent *A[3]* contains the larger value out of *A[3], A[6]* and *A[7]*. Two more comparisons have taken place. After the third iteration parent *A[2]* contains the larger value out of *A[2], A[4]* and *A[5]*. *A[4]* by now is already larger than *A[8]*. Two additional comparisons have taken place. Finally, after the final iteration of the for loop, parent *A[1]* contains the larger value out of *A[1], A[2]* and *A[3]*. Two final comparisons have taken place totaling to 8 comparisons. Each parent index is now bigger than its children and array *A* of length 8 is now a *max heap.*

2) Given a sequence of inputs, 17, 423, 64, 79, 411, 89, 99, 15

$h'(x) = x \bmod 13$.

(a) Chaining m = 13



17 mod 13 = 4

423 mod 13 = 7

64 mod 13 = 12

79 mod 13 = 1

411 mod 13 = 8

89 mod 13 = 11

99 mod 13 = 8 ← collision

15 mod 13 = 2

used for both these parts as well

(b) linear probing hash function: ~~[illegible]~~ $h(k,i) = (h'(k) + i) \bmod m$

0	
1	79
2	15
3	
4	17
5	
6	
7	423
8	411
9	99
10	
11	89
12	64

(99+1) mod 13 = 9 ↑ ← collision

(c) quadratic probing, hash function $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$

0		
1	79	
2	15	
3		
4	17	
5		
6		
7	423	
8	411	99 collision
9		new position:
10	99	$(99 + 1^2 + 1) \bmod 13$
11	89	$= 10$
12	64	

(c) double hashing $h_2(x) = 7 - (x \bmod 7)$

0	99		99 mod 13 = 8 collision
1	79	79 mod 13 = 1	7 − 99 mod 7 = 6 start probing
2	15	15 mod 13 = 2	8 + 1·6 = 14
3			14 % 13 = 1 taken
4	17	17 mod 13 = 4	8 + 2·6 = 20
5			20 % 13 = 7 taken
6			8 + 3·6 = 26
7	423	423 mod 13 = 7	26 % 13 = 0 not taken
8	411	411 mod 13 = 8	
9			
10			
11	89	89 mod 13 = 11	
12	64	64 mod 13 = 12	