Mastering Heap

March 18, 2017

1 Practice working with Heap Data Structures and Algorithms from CLRS and Geeks for Geeks. Org

1.1 CLRS

Question: What are the minimum and maximum number of elements in a heap of height h?

Answer: Minimum : 2^h , Maximum: $2^{h+1} - 1$

Explanation: Minimum number of elements in a heap of height h is size of a binary tree of height (h-1) + 1. In short it's (size(h-1) + 1) and maximum number of elements in a heap of height h is size of a binary tree of height h - 1. In short that is size(h)

To calculate size(h-1) :
size(0) = 2^0 = 1
size(1) = 2^0 + 2^1 = 3
...
...
size(h-1) = 2^0 + 2^1 + 2^2 + 2^{h-1} = $2^0(1 - 2^h)/(1 - 2)$ = $2^h - 1$
so minumum = $2^h - 1 + 1$ = 2^h
and maximum = $2^{h+1} - 1$

2 Build Max-Heap

We need to know that in a heap data structure the leaves start from (len(A)/2 + 1) for an array indexed at 1 and len(A)/2 for an array indexed at 0. Because if you want to get the child of this leaf you would get 2*(len(A) / 2 + 1) = len(A) + 2, which is not possible. Even the last element of the array is the child of the array-index len(A)/2

We also need the procedure MAX-HEAPIFY(A, i)

```
In [57]: def max_heapify(A, i):
             l = 2 * i + 1
             r = 2 * i + 2
             if l >= len(A) and r >= len(A):
                 return
             if l < len(A) and A[l] >= A[i]:
                 largest = l
             else:
                 largest = i
```

```
        if r < len(A) and A[r] >= A[largest]:
            largest = r
        if largest != i:
            A[largest], A[i] = A[i], A[largest]
            max_heapify(A, largest)
```

3 Non recursive implementation of max-heapify

```
In [63]: def max_heapify_non_rec(A, i):
             while i < (len(A) / 2):
                 l = 2 * i + 1
                 r = 2 * i + 2
                 if l >= len(A):
                     break
                 if l < len(A) and A[l] >= A[i]:
                     largest = l
                 else:
                     largest = i
                 if r < len(A) and A[r] >= A[largest]:
                     largest = r
                 if largest != i:
                     A[largest], A[i] = A[i], A[largest]
                     i = largest
```

```
In [66]: def build_max_heap(A):
             i = len(A) / 2 - 1
             while i >= 0:
                 max_heapify(A, i)
                 i = i - 1
             return A
```

```
In [67]: def build_max_heap_non_rec(A):
             i = len(A) / 2 - 1
             while i >= 0:
                 max_heapify_non_rec(A, i)
                 i = i - 1
             return A
```

```
In [68]: build_max_heap([1, 2, 3, 4, 9, 16, 7])
```

```
Out[68]: [16, 9, 7, 4, 2, 3, 1]
```

```
In [69]: build_max_heap_non_rec([1, 2, 3, 4, 9, 16, 7])
```

```
Out[69]: [16, 9, 7, 4, 2, 3, 1]
```

```
In [60]: def min_heapify(A, i):
             l = 2*i + 1
```

```
        r = 2*i + 2
        if l >= len(A) and r >= len(A):
            return
        if l < len(A) and A[l] <= A[i]:
            smallest = l
        else:
            smallest = i
        if r < len(A) and A[r] <= A[smallest]:
            smallest= r
        if smallest != i:
            A[smallest], A[i] = A[i], A[smallest]
            min_heapify(A, smallest)
```

```
In [61]: def build_min_heap(A):
             i = len(A) / 2
             while i >= 0:
                 min_heapify(A, i)
                 i = i - 1
             return A
```

```
In [62]: build_min_heap([12, 3, 4, 5, 2, 1])
```

```
Out[62]: [1, 2, 4, 5, 3, 12]
```

Build heap operation takes $\mathcal{O}(n)$

4 Heapsort

```
In [88]: def heapsort_descending(A):
             build_max_heap(A)
             i = len(A) - 1
             while i > 0:
                 # B is an alias for a sub array of A and B doesn't get allocated a new set of m
                 B = A[:i+1]
                 B[1], B[i] = B[i], B[1]
                 i = i - 1
                 max_heapify(B, 1)
             print A
```

```
In [85]: def heapsort(A):
             build_min_heap(A)
             i = len(A) - 1
             while i > 0:
                 B = A[:i+1]
                 B[1], B[i] = B[i], B[1]
                 i = i - 1
                 min_heapify(B, 1)
             print A
```

```
In [86]: heapsort([12, 3, 4, 5, 2, 1])
```

```
[1, 2, 4, 5, 3, 12]
```

```
In [87]: heapsort_descending([12, 3, 4, 5, 2, 1])
```

```
[12, 5, 4, 3, 2, 1]
```

5 Priority Queue

Priority queue is a **data structure for maintaining a set S of elements**, each with an associated value called **key**.

A max-priority queue supports the following operations:

1. INSERT(S, x): inserts x into set S.
2. MAXIMUM(S): returns the element with the maximum key from the set S.
3. EXTRACT-MAX(S): removes and returns the element with max key in set S.
4. INCREASE-KEY(S, x, k): increases the x's key to k. It's assumed that k is at least as large as x's current key.

6 Heap Class

We need to create Heap class because we need to store the properties of the heap. The most important property being the heap-size.

```
In [87]: class Heap_(object):
             #The heap class
             def __init__(self, arr = None):
                 if arr == None:
                     self.arr = list()
                 else:
                     self.arr = arr
                 self.HEAP_SIZE = len(self.arr)
             def set_heap_size(self, size):
                 self.HEAP_SIZE = size
             def get_heap_size(self):
                 return self.HEAP_SIZE
             def get_heap(self):
                 return self.arr
             def max_heapify(self, index):
                 left = index * 2 + 1
                 right = index * 2 + 2
                 arr = self.get_heap()
                 len_heap_arr = self.get_heap_size()
                 if left >= len_heap_arr:
                     return
```

```
        if left < len_heap_arr and arr[left] >= arr[index]:
            largest = left
        else:
            largest = index
        if right < len_heap_arr and arr[right] >= arr[largest]:
            largest = right
        if largest != index:
            arr[largest], arr[index] = arr[index], arr[largest]
            self.max_heapify(largest)
    def build_heap(self):
        arr= self.get_heap()
        len_heap_arr = self.get_heap_size()
        index = len_heap_arr / 2
        while index >= 0:
            self.max_heapify(index)
            index = index - 1
    def extract_max(self):
        arr = self.get_heap()
        max_ = arr[0]
        arr[0], arr[self.get_heap_size() - 1] = arr[self.get_heap_size() - 1], arr[0]
        self.set_heap_size(self.get_heap_size() - 1)
        self.max_heapify(0)
        return max_
    def increase_key(self, i, key):
        if i >= self.get_heap_size():
            print "invalid index"
            return
        arr = self.get_heap()
        if arr[i] > key:
            print "Error: entered key is smaller than current key"
        arr[i] = key
        while i > 0:
            parent = i / 2
            if arr[parent] < arr[i]:
                arr[parent], arr[i] = arr[i], arr[parent]
                i = parent
        return self.get_heap()
    def insert_key(self, key):
        heap_size = self.get_heap_size()
        arr = self.get_heap()
        if heap_size == len(arr):
            arr.append(-1000)
        elif heap_size < len(arr):
            arr[heap_size] = -1000
        self.set_heap_size(heap_size + 1)
        self.increase_key(heap_size, key)
        return arr
```

```
In [88]: heap = Heap_([1, 2, 3, 4, 9, 16, 7])

In [89]: heap.build_heap()

In [90]: heap.get_heap()

Out[90]: [16, 9, 7, 4, 2, 3, 1]

In [91]: heap.increase_key(1, 32)

Out[91]: [32, 16, 7, 4, 2, 3, 1]

In [92]: heap.extract_max()

Out[92]: 32

In [93]: heap.get_heap()[:heap.get_heap_size()]

Out[93]: [16, 4, 7, 1, 2, 3]

In [94]: heap.extract_max()

Out[94]: 16

In [95]: heap.get_heap()[:heap.get_heap_size()]

Out[95]: [7, 4, 3, 1, 2]

In [96]: heap.extract_max()

Out[96]: 7

In [97]: heap.get_heap()[:heap.get_heap_size()]

Out[97]: [4, 2, 3, 1]

In [98]: heap.increase_key(2, 32)

Out[98]: [32, 4, 2, 1, 7, 16, 32]

In [99]: heap.insert_key(64)

Out[99]: [64, 32, 4, 1, 2, 16, 32]

In [100]: heap.get_heap_size()

Out[100]: 5
```

6.1 Problems on Heap from CLRS continued...