Heap

December 5, 2017

0.1 Heap

- It is basically a binary tree
- Tow main binary heap types: min and max heap
- In a max heap, the keys of parent nodes are always greater than or equal to those of the children -> the highest key is in the root node.
- In a min heap, the keys of parent nodes are less than or equals to those of the children -> the lowest key is in the root node
- It is complete: it cannot be unbalanced !!! we insert every new item to the next available place
- Applications: Dijkstra's algorithm, Prims algorithm
- The heap is one maximally efficient implementation of a priority queue abstract data type
- It has nothing to do with the pool of memory from which dynamically allocated memory is allocated.

0.2 Heap properties

1) Complate -> we construct the heap from left to right across each row // of course the last row may not be completely full. There is no missing node from left to right in a layer

2) In a binary heap every node can have 2 children, left child and right child

3)

- Min heap -> the parent is always smaller than the values of the children
- Max heap -> the parent is always greater
- So: the root node will the smallest/greatest value in the heap. // O(1) access !!!
- It is an **O(n)** process to construct a heap
- W ehave to reconstruct it if the heap properties are viloated but it takes **O(logN)** time --> O(N) + O(logN) - O(N)
- inserting an item to the heap is just adding the data to the array with incremented index!!! **O(1)**

0.3 Deleting an item

- We just get rid of the item we want to delete. But there will be a hold in the tree. So we put the last item there, and make sure the heap properties are valid // with reconstructions!!!
- So: we have managed to get rid of the root node and to make some reconstructions in order to end up with a valid heap again!!!
- Operation: deleting the root node **O(1)** + reconstruction **O(logN) = O(logN)**
- Operation: deleting the arbitrary node **O(N)** + reconstruction **O(logN) = O(N)**!!!

0.4 Heapsort

- Comparison-based sorting algorithm
- Use Heap data structure rather than a linear-time search to find the maximum
- A bit slower in practice on most machines than a well-implemented quicksort, it has the advantage of a more favorable worst-case **O(N logN)** runtime
- It is an in-place algorithm, but it is not a stable sort
- **Does NOT NEED ADDITIONAL MEMORY**
- Problem: first we have to construct the heap itself from the numbers we want to sort -> **O(N)** time complexity!!!

0.5 Running time

- Running time: we have to consider **N** items + have to make some swappings if necessary **O(N*logN)**
- Memory complexity: we have **N** items we want to store in the heap. We have to allocate memory for an array with size **N** --> **O(N)** memory complexity
- Find the minimum /maximum: **O(1)** very fast. Because in a heap the highest priority item is at the root node, it is easy heapArray[0] will be the item we are looking for
- Insert new items: we can insert at the next avilable place, so increment the array index and insert it -> **O(1)** fast. But we have to make sure the heap propperties are met.. it may take **O(logN)** time
- **O(log_2 N)** why? bacause a node has at most **log_2 N** parents so at most **Log_2 N** swaps are needed
- Remove item: We ususlay remove the root node. Removing it is quite fast: just delete it in **O(1)** time. But we have to make sure we met the heap properties **O(logN)** time to reconstruct the heap!!!

```
In [13]: class Heap(object):
             HEAP_SIZE = 10

             def __init__(self):
               self.heap = [0]* Heap.HEAP_SIZE
               self.currentPosition = -1

             def insert(self, item):
```

```
    if self.isFull():
      print("Heap is full...")
      return

    self.currentPosition = self.currentPosition + 1
    self.heap[self.currentPosition] = item
    self.fixUp(self.currentPosition)

  def fixUp(self, index):
    parentIndex = int((index - 1)/2)

    while parentIndex >= 0 and self.heap[parentIndex] < self.heap[index]:
      temp = self.heap[index]
      self.heap[index] = self.heap[parentIndex]
      self.heap[parentIndex] = temp;
      parentIndex = (int)((index-1)/2)

  def heapsort(self):

    for i in range(0, self.currentPosition + 1):
      temp = self.heap[0]
      print("%d " % temp)
      self.heap[0] = self.heap[self.currentPosition - i]
      self.heap[self.currentPosition - i] = temp
      self.fixDown(0, self.currentPosition - i - 1)

  def fixDown(self, index, upto):

    while index <= upto:
      leftChild = 2*index  + 1
      rightChild = 2*index + 2

      if leftChild < upto:
        childToSwap = None

        if rightChild > upto:
          childToSwap = leftChild
        else:
          if self.heap[leftChild] > self.heap[rightChild]:
            childToSway = leftChild
          else:
            childToSway = rightChild

        if self.heap[index] < self.heap[childToSwap]:
          temp = self.heap[index]
          self.heap[index] = self.heap[childToSway]
          self.heap[childToSwap] = temp
        else:
```

```
                break

              index = childToSwap

            else:
              break


        def isFull(self):
          if self.currentPosition == Heap.HEAP_SIZE:
            return True
          else:
            return False
```

```
In [14]: heap = Heap()
         heap.insert(10)
         heap.insert(-20)
         heap.insert(0)
         heap.insert(2)

         heap.heapsort()
```

```
10
```

```
        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-14-458ada994925> in <module>()
          5 heap.insert(2)
          6
    ----> 7 heap.heapsort()


        <ipython-input-13-f70e9428940e> in heapsort(self)
         31       self.heap[0] = self.heap[self.currentPosition - i]
         32       self.heap[self.currentPosition - i] = temp
    ---> 33       self.fixDown(0, self.currentPosition - i - 1)
         34
         35   def fixDown(self, index, upto):


        <ipython-input-13-f70e9428940e> in fixDown(self, index, upto)
         50             childToSway = rightChild
         51
    ---> 52         if self.heap[index] < self.heap[childToSwap]:
```

```
53            temp = self.heap[index]
54            self.heap[index] = self.heap[childToSway]
```

```
TypeError: list indices must be integers or slices, not NoneType
```

In [ ]: