Heap and Priority Queue

Li Yin[1]

February 6, 2019

[1]www.liyinscience.com

In this chapter, we introduce heap data structures which is essentially an array object but it can be viewed as a nearly complete binary tree. The concept of the data structures in this chapter is between liner and non-linear, that is using linear data structures to mimic the non-linear data structures and its behavior for higher efficiency under certain context.

0.1 Heap

Heap is a tree based data structures that satisfies **heap property** but implemented as an array data structure. There are two kinds of heaps: **max-heaps** and **min-heaps**. In both kinds, the values in the nodes satisfy a heap property. For max-heap, the property states as for each subtree rooted at x, items on all of its children subtrees of x are smaller than x. Normally, heap is based on binary tree, which makes it a binary heap. Fig. 1 show a binary max-heap and how it looks like in a binary tree data structure. In the following content, we default our heap is a binary heap. Thus, the largest element in a max-heap is stored at the root. For a heap of n elements the height is $\log n$).

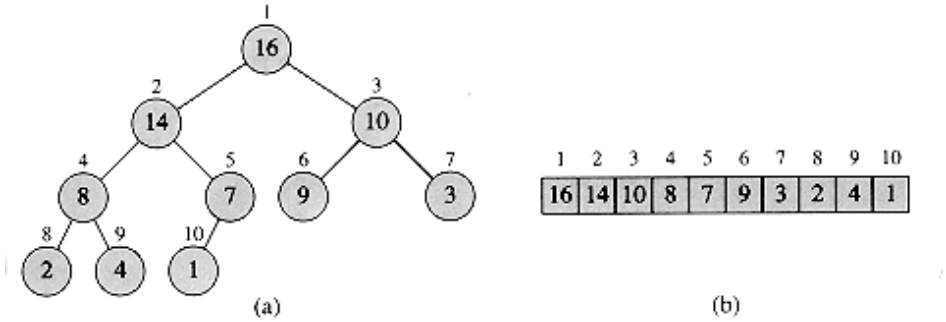

Figure 1: Max-heap be visualized with binary tree structure on the left, and implemnted with Array on the right.

As we can see we can implement heap as an array due to the fact that the tree is complete. A complete binary tree is one in which each level must be fully filled before starting to fill the next level. Array-based heap is more space efficient compared with tree based due to the non-existence of the child pointers for each node. To make the math easy, we iterate node in the tree starting from root in the order of level by level and from left to right with beginning index as 1 (shown in Fig. 1). According to such assigning rule, the node in the tree is mapped and saved in the array by the assigned index (shown in Fig. 1). In heap, we can traverse the imaginary binary tree in two directions: **root-to-leaf** and **leaf-to-root**. Given a parent node with p as index, the left child of can be found in position $2p$ in the array. Similarly, the right child of the parent is at position $2p + 1$ in the list. To find the parent of any node in the tree, we can simply use $\lfloor p/2 \rfloor$. In Python3, use integer division $n//2$. *Note: we can start index with 0 as used in* ***heapq*** *library introduced later in this section. Given a node x, the left and right child will be $2*x+1$, $2*x+2$, and the parent node will have index $(x-1)//2$.*

The common application of heap data structure include:

- Implementing a priority-queue data structure which will be detailed in the next section so that insertion and deletion can be implemented in $O(\log n)$; Priority Queue is an important component in algorithms like Kruskal's for minimum spanning tree (MST) problem and Dijkstra's for single-source shortest paths (SSSP) problem.

- Implementing heapsort algorithm,

Normally, there is usually no notion of 'search' in heap, but only insertion and deletion, which can be done by traversing a $O(\log n)$ leaf-to-root or root-to-leaf path.

0.1.1 Basic Implementation

The basic method that a heap supports are **insert** and **pop**. Additionally, given an array, to convert it to a heap, we need operation called **heapify**.

Let's implement a heap class using list. Because the first element of the heap is actually empty, we define our class as follows:

```
class Heap:
    def __init__(self):
        self.heap = [None]
        self.size = 0
    def __str__(self):
        out = ''
        for i in range(1, self.size + 1):
            out += str(self.heap[i]) + ' '
        return out
```

Insert with Floating When we insert an item, we put it at the end of the heap (array) first and increase the size by one. After this, we need to traverse from the last node leaf-to-root, and compare each leaf and parent pair to decide is a swap operation is needed to force the heap property. For example, in the min-heap. The time complexity is the same as the height of the complete tree, which is $O(\log n)$.

```
    def _float(self, index): # enforce min-heap, leaf-to-root
        while index // 2: # while parent exist
            p_index = index // 2
            if self.heap[index] < self.heap[p_index]:
                # swap
                self.heap[index], self.heap[p_index] = self.heap
    [p_index], self.heap[index]
            index = p_index # move up the node
    def insert(self, val):
        self.heap.append(val)
        self.size += 1
        self._float(index = self.size)
```

Pop with Sinking When we pop an item, we first save the root node's value in order to get either the maximum or minimum item in the heap. Then we simply use the last item to fill in the root node. After this, we need to traverse from the root node root-to-leaf, and compare each parent with left and right child. In a min-heap, if the parent is larger than its smaller child node, we swap the parent with the smaller child. This process is called sinking. Same as the insert, $O(\log n)$.

```
    def _sink(self, index): # enforce min-heap, root-to-leaf
        while 2 * index < self.size:
            li = 2 * index
            ri = li + 1
            mi = li if self.heap[li] < self.heap[ri] else ri
            if self.heap[index] > self.heap[mi]:
                # swap index with mi
                self.heap[index], self.heap[mi] = self.heap[mi],
     self.heap[index]
            index = mi
    def pop(self):
        val = self.heap[1]
        self.heap[1] = self. heap.pop()
        self.size -= 1
        self._sink(index = 1)
        return val
```

Now, let us run an example:

```
h = Heap()
lst = [21, 1, 45, 78, 3, 5]
for v in lst:
    h.insert(v)
print('heapify with insertion: ', h)
```

```
h.pop()
print('after pop(): ', h)
```

The output is listed as:

```
heapify with insertion: 1 3 5 78 21 45
after pop(): 3 21 5 78 45
```

Heapify with Bottom-up Floating Heapify is a procedure that convert a list to a heap data structure. Similar to the operation insertion, it uses floating. Differently, if we iterating through the list and use insertion operation, each time we try to float, the previous list is already a heap. However, given an unordered array, we can only get the minimum/maximum node floating starting from the bottom. Therefore, we call floating process for the elements in the list in reverse order as a bottom-up manner. The upper bound time complexity is $O(n \log n)$ because we have n call of the float which has an upper bound $O(\log n)$. However, we can prove it actually has a tighter bound by observing that the running time depends on the height of the tree. The proving process is shown on geeksforgeeks [1].

```
    def heapify(self, lst):
        self.heap = [None] + lst
        self.size = len(lst)
        for i in range(self.size, 0, -1):
            self._float(i)
```

Now, run the following code:

```
h = Heap()
h.heapify(lst)
print('heapify with heapify:', h)
```

Out put is:

```
heapify with heapify: 1 5 21 78 3 45
```

Which way is more efficient building a heap from a list?
Using insertion or heapify? What is the efficiency of each method? The experimental result can be seen in the code.

When we are solving a problem, unless specifically required for implementation, we can always use an existent Python module/package. Here, we introduce one Python module: heapq that implements heap data structure for us.

[1] https://www.geeksforgeeks.org/time-complexity-of-building-a-heap/

0.1.2 Python Built-in Library: heapq

heapq: heapq is a built-in library in Python that implements relevant functions to carry out various operations on heap data structure. These functions are listed and described in Table 1. *To note that heapq is not a data type like queue.Queue() or collections.deque(), it is a library (or class) that can do operations like it is on a heap.* heapq has some other

Table 1: Methods of **heapq**

Method	Description
heappush(h, x)	Push the value item onto the heap, maintaining the heap invariant.
heappop(h)	Pop and return the *smallest* item from the heap, maintaining the heap invariant. If the heap is empty, IndexError is raised.
heappushpop(h, x)	Push item on the heap, then pop and return the smallest item from the heap. The combined action runs more efficiently than heappush() followed by a separate call to heappop().
heapify(x)	Transform list x into a heap, in-place, in linear time.
heapreplace(h, x)	Pop and return the smallest item from the heap, and also push the new item. The heap size doesn't change. If the heap is empty, IndexError is raised. This is more efficient than heappop() followed by heappush(), and can be more appropriate when using a fixed-size heap.
nlargest(k, iterable, key = fun)	This function is used to return the k largest elements from the iterable specified and satisfying the key if mentioned.
nsmallest(k, iterable, key = fun)	This function is used to return the k smallest elements from the iterable specified and satisfying the key if mentioned.

functions like merge(), nlargest(), nsmallest() that we can use. Check out `https://docs.python.org/3.0/library/heapq.html` for more detials.

Now, let us try to heapify the same examplary list as used in the last section, [21, 1, 45, 78, 3, 5], we use need to call the function heapify(). The time complexity of heapify is $O(n)$

```
'''implementing with heapq'''
from heapq import heappush, heappop, heapify
h = [21, 1, 45, 78, 3, 5]
heapify(h) # inplace
print('heapify with heapq: ', h)
```

The print out is:

```
heapify with heapq:  [1, 3, 5, 78, 21, 45]
```

As we can see the default heap implemented in the heapq library is forcing the heap property of the min-heap. What is we want a max-heap instead? In heapq library, it does offer us function, but it is intentionally hided from users. It can be accessed like: heapq._[function]_max(). Now, let us implement a max-heap instead.

```
# implement a max-heap
h = [21, 1, 45, 78, 3, 5]
heapq._heapify_max(h) # inplace
print('heapify max-heap with heapq: ', h)
```

The print out is:

```
heapify max-heap with heapq:  [78, 21, 45, 1, 3, 5]
```

Also, in practise, a simple hack for the max-heap is to save data as negative. Also, in the priority queue.

0.2 Priority Queue

A priority queue is an extension of queue with properties: (1) additionally each item has a priority associated with it. (2) In a priority queue, an item with high priority is served (dequeued) before an item with low priority. (3) If two items have the same priority, they are served according to their order in the queue.

Heap is generally preferred for priority queue implementation because of its better performance compared with arrays or linked list. Also, in Python queue module, we have PriorityQueue() class that provided us the implementation. Beside, we can use heapq library too. These contents will be covered in the next two subsection.

Applications of Priority Queue:

1. CPU Scheduling
2. Graph algorithms like Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc
3. All queue applications where priority is involved.

Implement with heapq Library The core function is the ones used to implement the heap: heapify(), push(), and pop(). Here we demonstrate how to use function nlargest() and nsmallest() if getting the first n largest or smallest is what we need, we do not need to heapify() the list as we needed in the heap and pop out the smallest. The step of heapify is built in these two functions.

```
''' use heapq to get nlargest and nsmallest'''
li1 = [21, 1, 45, 78, 3, 5]
# using nlargest to print 3 largest numbers
print("The 3 largest numbers in list are : ", end="")
print(heapq.nlargest(3, li1))

# using nsmallest to print 3 smallest numbers
print("The 3 smallest numbers in list are : ", end="")
print(heapq.nsmallest(3, li1))
```

The print out is:

```
The 3 largest numbers in list are : [78, 45, 21]
The 3 smallest numbers in list are : [1, 3, 5]
```

Implement with PriorityQueue class Class PriorityQueue() is the same as Queue(), LifoQueue(), they have same member functions as shown in Table **??**. Therefore, we skip the semantic introduction. PriorityQueue() normally thinks that the smaller the value is the higher the priority is, such as the following example:

```
import queue #Python3 needs to use queue, others Queue
q = queue.PriorityQueue()
q.put(3)
q.put(10)
q.put(1)

while not q.empty():
    next_job = q.get()
    print('Processing job:', next_job)
```

The output is:

```
Processing job: 1
Processing job: 3
Processing job: 10
```

We can also pass by any data type that supports '<' comparison oeprator, such as tuple, it will use the first item in the tuple as the key.

If we want to give the number with larger value as higher priority, a simple hack is to pass by negative value. Another more professional way is to pass by a customized object and rewrite the comparison operator: $<$ and $==$ in the class with ___lt___() and ___eq___(). In the following code, we show how to use higher value as higher priority.

```
class Job(object):
    def __init__(self, priority, description):
        self.priority = priority
        self.description = description
        print('New job:', description)
        return
    # def __cmp__(self, other):
    #     return cmp(self.priority, other.priority)
    '''customize the comparison operators '''
    def __lt__(self, other): # <
        try:
            return self.priority > other.priority
        except AttributeError:
            return NotImplemented
    def __eq__(self, other): # ==
        try:
            return self.priority == other.priority
        except AttributeError:
```

```
            return NotImplemented

q = Queue.PriorityQueue()

q.put( Job(3, 'Mid-level job') )
q.put( Job(10, 'Low-level job') )
q.put( Job(1, 'Important job') )

while not q.empty():
    next_job = q.get()
    print('Processing job:', next_job.priority)
```

The print out is:

```
Processing job: 10
Processing job: 3
Processing job: 1
```

In single thread programming, is heapq or PriorityQueue more efficient?

In fact, the PriorityQueue implementation uses heapq under the hood to do all prioritisation work, with the base Queue class providing the locking to make it thread-safe. While heapq module offers no locking, and operates on standard list objects. This makes the heapq module faster; there is no locking overhead. In addition, you are free to use the various heapq functions in different, noval ways, while the PriorityQueue only offers the straight-up queueing functionality.

0.3 Bonus

Let us take these knowledge into practice with a LeetCode Problem:

0.1 **347. Top K Frequent Elements (medium).** Given a non-empty array of integers, return the k most frequent elements.

```
Example 1:

Input: nums = [1,1,1,2,2,3], k = 2
Output: [1,2]

Example 2:

Input: nums = [1], k = 1
Output: [1]
```

Analysis: to solve this problem, we need to first using a hashmap to get information as: item and its freqency. Then, we need to obtain the top frequent elements. The second step can be down with sorting, or using heap we learned.

Solution 1: Use Counter(). Counter() has a function most_common(k) that will return the top k most frequent items. However, its complexity will be $O(n \log n)$.

```
from collections import Counter
def topKFrequent(self, nums, k):
    return [x for x, _ in Counter(nums).most_common(k)]
```

Solution 2: Use dict and heapq.nlargest(). The complexity should be better than $O(n \log n)$.

```
from collections import Counter
import heapq
def topKFrequent(self, nums, k):
    count = collections.Counter(nums)
    return heapq.nlargest(k, count.keys(), key=count.get)
```

We can also use PriorityQueue().

```
from queue import PriorityQueue
class Solution:
def topKFrequent(self, nums, k):
    h = PriorityQueue()

    # build a hashmap (element, frequency)
    temp = {}
    for n in nums:
        if n not in temp:
            temp[n] = 1
        else:
            temp[n] += 1
    # put them as (-frequency, element) in the queue or
    heap
    for key, item in temp.items():
        h.put((-item, key))

    # get the top k frequent ones
    ans = [None]*k
    for i in range(k):
        _, ans[i] = h.get()
    return ans
```

Fibonacci heap With fibonacc heap, insert() and getHighestPriority() can be implemented in O(1) amortized time and deleteHighestPriority() can be implemented in O(Logn) amortized time.

0.4 LeetCode Problems

selection with key word: kth. These problems can be solved by sorting, using heap, or use quickselect

1. 703. Kth Largest Element in a Stream (easy)

2. 215. Kth Largest Element in an Array (medium)
3. 347. Top K Frequent Elements (medium)
4. 373. Find K Pairs with Smallest Sums (Medium
5. 378. Kth Smallest Element in a Sorted Matrix (medium)

priority queue or quicksort, quickselect

1. 23. Merge k Sorted Lists (hard)
2. 253. Meeting Rooms II (medium)
3. 621. Task Scheduler (medium)