This Class: Heap

Our goal with introducing the “Heap” data structure is listed below:

1. Dealing with the “Insert” operation;
2. Finding the maximum or minimum (depends on the property of the heap we use) element in O(1) time even under the worst case.
3. Also, we need the operation like deleting the maximum or minimum element.

Definition of Heap:
- A data structure for finding minimum or maximum element in a much faster way.
- Array A with property, such that A[i] ≥ A[2i] and A[i] ≥ A[2i+1], for any i=1,2,3,…, where i represents the index of any parent nodes for any sub-trees, 2i and 2i+1 represent the indices of children nodes rooted at i. The two inequality formulas imply that parent node is always larger than its children nodes within that sub-tree. The following graph is a specific example of a heap.

In the graph, we can see that element 15, 3 and 8 are all parent nodes for their own sub-trees rooted at them.

Also, we can see that for element 3, it is only larger than both of its own children within that sub-tree (dash box). It won’t be larger than element 6 which is the children of element 8.

Now, let’s represent the heap in the array representation, which is A:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>3</td>
<td>8</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>


And note that usually, we use a variable “heapsize” to denote the size of an array. So the above heap has a size of 6.

Here, we are going to show how to find “parent” and “children” nodes.

%This is a function finding the parent node for particular i.
Function [indexP] = parent (i)

Return \( \left\lfloor \frac{i}{2} \right\rfloor \)
%This is a function finding the left children for particular parent node i. 
Function \[\text{left}(i)\] = left (i)

\[
\text{Return } 2i
\]

%This is a function finding the right children for particular parent node i. 
Function \[\text{right}(i)\] = right (i)

\[
\text{Return } 2i+1
\]

**Operation:** function = Insert(x)

Now, let’s consider how to insert an element into a heap.

Process:
Initially, we put the element x that we want to insert at the last position of the array, and then check if \(x > \text{parent (indeX)}\) (here we use indeX to denote the index of element x). If the condition is satisfied, we swap(x, parent (indeX)).

In the following, we need to recursively call this procedure until all the elements are satisfied with the property of a heap. The following is an example in which we show this procedure.

Example: let’s assume that we want to add an element 16 (the black circle) into the current heap.

Operation: function = Delete_maximum(T)

Now, let’s consider another operation of heap: function Delete_maximum(T)

Process:
- put the current last element to root and we get a new root element \(R’\);
- then compare \(R’\) with the children of previous root node \(R\);
- if one of children element of \(R\) is larger than \(R’\), swap \((R’, \text{children of } R)\);
- follow \(R’\) (now is in one of the children position of \(R\)) down the tree until this element reach to a point that all the elements in the array can be satisfied with the property of heap. A specific example is in the following graph.

Example: let’s assume we have deleted the maximum element 16 in the heap.
In our example, since element 8 is larger than its current children, which is 6, we are already reach a point that all the elements in the heap are satisfied with the property of heap. However, perhaps in other cases, after element 8 goes down to its previous children’s position as shown in the right sub-graph above, we still need to follow it to continue going down until we reach that point.

Pseudo code for the above procedure:

\[
\text{heapify} \ (A, \ i) \\
\text{L} \leftarrow \text{left} \ (i) \\
\text{R} \leftarrow \text{right} \ (i) \\
\text{If } \text{L} \leq \text{heapsize} \ (A) \text{ and } A(L) > A \ (i) \\
\text{largest} \leftarrow L \\
\text{else} \\
\text{largest} \leftarrow i \\
\text{end} \\
\text{if } \text{R} \leq \text{heapsize} \ (A) \text{ and } A(R) > A \ (\text{largest}) \\
\text{largest} \leftarrow R \\
\text{end} \\
\text{if } \text{largest} \neq i \\
\text{swap} \ (A(i), \ A(\text{largest})) \\
\text{heapify} \ (A, \ \text{largest})
\]

Thus, when we first move the last element to the root, we can call this function like heapify(A, 1).

The cost of the function of heapify (A, i) would be O(depth(T)), since the depth of the heap Tree is depth(T).

**Building a heap**
- Given an array
- Want to reorganize the elements to have heap.
- Elements \(A[\left\lfloor \frac{n}{2} \right\rfloor + 1 : n] \) are leaves, the following graphs show this.
Idea:

- Bottom-up repair, the graph on the right shows this.
  From the \( \odot 1 \), \( \odot 2 \), \( \odot 3 \), \( \odot 4 \) and so on.

Pseudo code for this procedure:

```
Build-Heap (A)

heapsize ← length(A)
for i ← \left\lfloor \frac{heapsize}{2} \right\rfloor : - 1 : 1
  heapify (A, i)
end
```

Cost Analysis:

\[
\sum_{k=0}^{\log n} \sum_{v \text{ at level } k} (\log n - k) \leq \sum_{k=0}^{\log n} (2^k \log n - k2^k) \leq \log n \sum_{k=0}^{\log n} 2^k - \sum_{k=0}^{\log n} k2^k
\]

\[
= \log n \sum_{k=0}^{\log n} 2^k - 2 \sum_{k=0}^{\log n} k2^{k-1}
\]

\[
= \log n (2^{\log n+1} - 1) - 2 \sum_{k=0}^{\log n} k2^{k-1}
\]

\[
= \log n (2^{\log n+1} - 1) - 2 \left( \frac{x^{\log n+1} - 1}{x-1} \right) \bigg|_{x=2}
\]

\[
= \log n (2^{\log n+1} - 1) - 2 \left( \frac{x^{\log n+1} - 1}{(x-1)^2} \right) \bigg|_{x=2}
\]

\[
= \log n (2^{\log n+1} - 1) - 2(2^{\log n+1} \log n + 1 - 2^{\log n+1} + 1)
\]
= \log n \ (2n - 1) - 2((\log n + 1) n - 2n + 1)

= 2n \ \log n - 2n \ \log n - 2n + 4n - 2

= 2n - 2 = \Theta(n)

From the result above, we can see that it just takes linear time to reorganize an array to a heap.