1 Heap Sort

1.1 Pseudocode

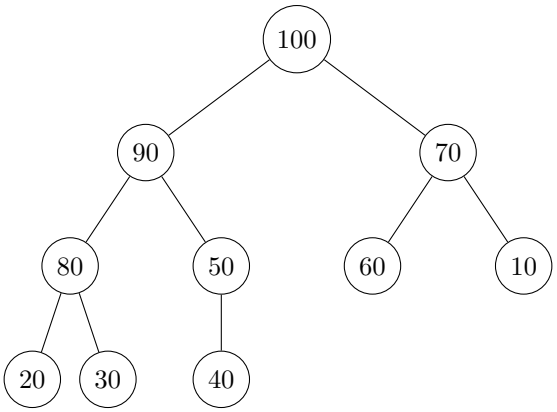
Algorithm 1 Heap Sort

```
1:  procedure HEAPSORT(A, n)                    ▷ A is a list and n is A's size
2:      for r = ⌊n/2⌋ down to 1 do                                  ▷ Create Heap
3:          SIFT(r, n)
4:      end for
5:
6:      for m = n down to 2 do                                      ▷ Finish Sort
7:          A[1] ↔ A[m]
8:          SIFT(1, m − 1)
9:      end for
10: end procedure

11: procedure SIFT(p, m)          ▷ p is the root and m is the size of the list
12:     c ← 2p
13:     while c ≤ m do
14:         if c < m then
15:             if A[c + 1] > A[c] then
16:                 c ← c + 1
17:             end if
18:         end if
19:
20:         if A[c] > A[p] then
21:             A[p] ↔ A[c]
22:             p ← c
23:             c ← 2p
24:         else
25:             exit while loop
26:         end if
27:     end while
28: end procedure
```

1.2 Create Heap

A Heap is a binary tree where every value is larger than its children. Equivalently its descendants. For the purposes of this class will we require that all binary trees are full binary trees.



The traditional way of creating a heap is to insert at the end of the array and sift up. Robert Floyd created a better algorithm for creating the heap. Treat the tree as a recursive heap: each parent is the parent of 2 heaps, and sift from the bottom up. Create heap on left, create heap on right, then sift root down, and move up a level.

Heap Creation Analysis

In a binary tree, most nodes are near the bottom, so when doing the bottom up technique, most of the work is done at the bottom. The number of comparisons can be calculated like so,

$$\frac{n}{2} \cdot 0 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 4 + \cdots = \\ = n \left[\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \cdots \right]$$

Note that $\left[\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \cdots \right]$ can be written as so,

$$\begin{aligned} \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots &= 1 \\ \frac{1}{4} + \frac{1}{8} + \cdots &= \frac{1}{2} \\ \frac{1}{8} + \cdots &= \frac{1}{4} \end{aligned}$$

Which all together becomes

$$1 + \frac{1}{2} + \frac{1}{4} + \cdots = 2$$

So Heap creation does $2n$ comparisons.

1.3 Finish Sort

After heap creation we then sort it,

- Put root node at the bottom of the array (it must be the largest element) so it goes to end of the sorted array.
- Then take the bottom right hand leaf and move to a temporary space.
- Then sift, reordering the tree and put the temporary leaf in its proper spot.
- Repeat until all elements are sorted.

Heap Sort Analysis

Each level has two comparisons, child and temporary. There are $\approx \lg n$ levels, so the total comparisons for a sift is $\approx 2 \lg n$. This is done for each element in the tree so worst case we get $\approx 2n \lg n$. But heap shrinks upon each iteration (it removes an element) so we take the sum from 0 to $n-1$.

$$\begin{aligned}
\sum_{i=0}^{n-1} 2 \lg(i+1) &\approx 2 \sum_{i=1}^{n} \lg i \\
&= 2[\lg 1 + \lg 2 + \lg 3 + \cdots + \lg n] \\
&= 2 \lg(1 \times 2 \times 3 \times \cdots \times n) \\
&= 2 \lg(n!) \\
&= 2 \lg \left[\left(\frac{n}{e} \right)^n \cdot \sqrt{2\pi n} \right] \\
&\approx 2 \left[n \lg \left(\frac{n}{e} \right) + \frac{\lg(2\pi n)}{2} \right] \\
&= 2n \lg n - 2n \lg e + \lg n + \lg(2\pi) \\
&= 2n \lg n + O(n)
\end{aligned}$$

From this we see that even while the tree shrinks, it does not shrink fast enough to make some notable difference. We are still doing $2n \lg n$ comparisons.

1.4 Implementation

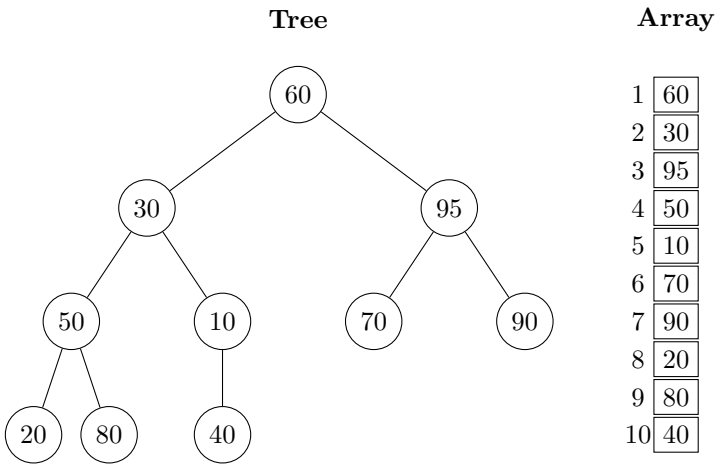


Figure 1: Use an array to implement a tree.
Node has index i, Left child is $2i$, Right child is $2i + 1$, and Parent is $\lfloor \frac{i}{2} \rfloor$.

Create

The first parent is at index $\lfloor \frac{n}{2} \rfloor$. Start there and sift down during heap creation. Siblings can be reached by adding or subtracting 1. The result is a created heap.

Finish

To finish the sort we push bottom into tmp first, then move heap root into the bottom most spot.

1.5 Optimization

As Heapsort stands, the result is worse than merge sort. $\Theta(2n \lg n)$ vs $\Theta(n \lg n)$ shows us that much. We compare tmp against both children and this doubles our total number of comparisons. Instead we can sift the hole left by the root down to the bottom in $\lg n$ comparisons. Then put tmp in the hole and sift it back into position. It follows then that we give up $2n$ comparisons on average. Further optimization can be achieved by binary searching up. This gives Heapsort $n \lg n + n \lg(\lg n) = \Theta(n \lg n)$ performance.