Heapsort

This note presents the essence of heapsort, and illustrates the execution on a list of integers. Data structure used is an array. Unlike certain other sorting schemes, heapsort does not require additional space. In other words, it is an in-place algorithm. The essentials of this scheme are as follows.

1. Read the input list of integers into an array, say $A[0..n-1]$, where $n$ denotes the number of elements.

   **Note:** The array $A$ itself may be viewed as storing a complete binary tree whose root is at $A[0]$ and in which node $A[i]$’s left child (if present) is $A[2i + 1]$ and right child (if present) is $A[2i + 2]$.

2. Transform the complete binary tree into a heap so that an element at an internal node is greater than or equal to each of the elements at its children nodes. In the process, the largest element necessarily pops up at the root that is $A[0]$. Initialize an index variable $last$ to $n – 1$.

3. Swap $A[0]$ and $A[last]$, and transform the (smaller) complete binary tree consisting of the elements $A[0]$ through $A[last – 1]$ into a heap. In the process, the next largest element pops up at the root that is $A[0]$.

4. Decrement $last$, and repeatedly perform Step 3 until $last = 1$.

A piece of pseudocode follows.
heapSort(inout $A$: arrayType, in $n$: integer)  // Sorts $A[0 .. n – 1]$ into ascending order.

// First transform the array so that it satisfies the heap property.
for ($i = n/2$ down to 0)
    heapify($A$, $i$, $n$);    // Assertion: The (sub)tree rooted at $i$ is a heap.

// Assertion: $A[0]$ is the largest item in the heap $A[0 .. n – 1]$.
last = $n – 1$;
// Invariant: $A[0..last]$ is a heap, and the subarray $A[last + 1 .. n – 1]$ is sorted.
for (step = 1 through $n$)
{    // move the largest item, i.e., $A[0]$ to the beginning of the sorted region by swapping.
    swap $A[0]$ and $A[last]$;
    // expand the sorted region and shrink the heap region.
    decrement last;
    // make the heap region a heap again;
    heapify($A$, 0, last);
} // end for

heapify(inout $A$: arrayType, in $root$: integer, in $size$: integer)
// Converts a semiheap rooted at index $root$ into a heap.

// Recursively trickle the item at index $root$ down to its proper position
// by swapping it with its larger child in case the child is larger than the item itself.
// If the item is at a leaf, nothing needs to be done.
if ($root$ is not a leaf)
{ // root must have a left child.
    $child = 2 * root + 1$; // index of the left child
    if (root has a right child)
    {
        $rightChild = child + 1$; // index of the right child
            $child = rightChild$; // index of the right child
    } // end if
    // If the root item is smaller than the larger child item, swap them.
    {
        // transform the semiheap rooted at child into a heap.
        heapify($A$, $child$, $size$); // recursive call
    } // end if
} // end if

Following is a trace of the execution of heapsort on a list of integers.
Tracing the execution of heapsort on a list of integers

Consider the following list of integers: 18, 12, 54, 75, 64, 25, 42, 78, 96. Note that \( n = 9 \).

**Original array \( A \)**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>18</td>
<td>12</td>
<td>54</td>
<td>75</td>
<td>64</td>
<td>25</td>
<td>42</td>
<td>78</td>
<td>96</td>
</tr>
</tbody>
</table>

**Tree representation of the array**

First build the heap.

Start from the rightmost non-leaf node on the level that is just above the lowest level. This happens to be 75 whose index is 3. Invoke heapify(\( A \), 3, 9). Larger of the two children is 96 that is larger than 75 itself, so swapping takes place.

**Array \( A \) after heapify(\( A \), 3, 9)**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>18</td>
<td>12</td>
<td>54</td>
<td>96</td>
<td>64</td>
<td>25</td>
<td>42</td>
<td>78</td>
<td>75</td>
</tr>
</tbody>
</table>

**Tree representation of the array**
Next consider the non-leaf node at index 2 that is 54. Invoke heapify($A$, 2, 9).

Since children of 54 are each smaller than 54, there is no change at this point.

Next consider the non-leaf node at index 1 that is 12. Invoke heapify($A$, 1, 9). Larger of the two children is 96 that is larger than 12 itself, so swapping takes place.

The resulting tree follows.
Because of the swapping having taken place at this point, the present execution of heapify($A$, 1, 9) induces a recursive invocation of heapify($A$, 3, 9). Larger of the two children of 12 is 78 that is larger than 12 itself, so swapping takes place.

Next consider the non-leaf node at index 0 that is 18. Invoke heapify($A$, 0, 9). Larger of the two children is 96 that is larger than 18 itself, so swapping takes place.
The resulting tree follows.

Because of the swapping having taken place at this point, the present execution of heapify($A$, 0, 9) induces a recursive invocation of heapify($A$, 1, 9). Larger of the two children of 18 is 78 that is larger than 18, so swapping takes place.

The resulting tree follows.
Again, because of the swapping having taken place at this point, the present execution of `heapify(A, 1, 9)` induces a recursive invocation of `heapify(A, 3, 9)`. Larger of the two children of 18 is 75 that is larger than 18, so swapping takes place.

The resulting tree follows.

The present chain of recursive invocations now terminates, since element 18 now occupies a leaf node. Also, the process of building initial heap itself is now complete.
Here starts the actual sorting.

Set \( \text{last} = n - 1 \), so \( \text{last} = 8 \). Swap \( A[0] \) with \( A[\text{last}] \).

Array \( A \) after swapping \( A[0] \) and \( A[\text{last}] \)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>78</td>
<td>54</td>
<td>75</td>
<td>64</td>
<td>25</td>
<td>42</td>
<td>12</td>
<td>96</td>
</tr>
</tbody>
</table>

Tree representation of the array

Decrement \( \text{last} \), so \( \text{last} = 7 \), and perform heapify(\( A, 0, \text{last} \)).
This step consists of swapping 18 and 78 and a recursive invocation of heapify(\( A, 1, \text{last} \)),
which in turn invokes heapify(\( A, 3, \text{last} \)). The successive snapshots of the tree are as follows.
Decrement last, so last = 6, and perform heapify($A, 0, last$).
This step consists of swapping 12 and 75 and a recursive invocation of heapify($A, 1, last$),
which in turn invokes heapify($A, 4, last$). The successive snapshots of the tree are as follows.

Array $A$ after swapping $A[0]$ and $A[last]$

Decrement last, so last = 5, and perform heapify($A, 0, last$).
This step consists of swapping 42 and 64 and a recursive invocation of heapify($A, 1, last$).
The successive snapshots of the tree are as follows.
Decrement last, so last = 4, and perform heapify(A, 0, last).
This step consists of swapping 25 and 54 and a recursive invocation of heapify(A, 2, last).
The successive snapshots of the tree are as follows.
Decrement last, so last = 3, and perform heapify(A[0], last).
This step consists of swapping 12 and 42 and a recursive invocation of heapify(A[1], last),
which in turn invokes heapify(A[3], last). The successive snapshots of the tree follow.

Array A after swapping A[0] and A[last]

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>18</td>
<td>25</td>
<td>42</td>
<td>54</td>
<td>64</td>
<td>75</td>
<td>78</td>
<td>96</td>
</tr>
</tbody>
</table>

Heap Sorted

Tree representation of the array

Decrement last, so last = 2, and perform heapify(A[0], last).
This step consists of swapping 12 and 25 and a recursive invocation of heapify(A[2], last).
The successive snapshots of the tree are as follows.
Decrement last, so last = 1, and perform heapify(A, 0, last).
This step consists of swapping 12 and 18 and a recursive invocation of heapify(A, 1, last).
The successive snapshots of the tree are as follows.