Heapsort / Treesort (Floyd & Williams)

Definition: Heap

It is best to view a Heap as a binary tree even though it is implemented on an array.
A binary tree is a Heap iff

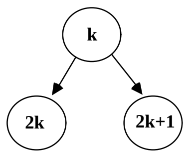
- **It is complete (i.e. Balanced) in the sense that if n = #nodes then height of tree $\leq \lceil \log n \rceil$**
  <u>Note</u>: Height of tree = max level of all nodes in the tree (level of root = 1)
  e.g. n = 10, Height of Heap (tree) $\leq \lceil \log 10 \rceil = 4$.
  In a Heap the lowest or 'leaf' level is filled from left to right.

- **It has the Heap Property, i.e. for each node k, value at k $\geq$ value of each child (if any) of of k.**

Representation of Heap using Arrays

An array A(1..n) is a Heap of n nodes where for any node k, A@k is the value of node k (or A.item(k).key, the associated key) and the children of k are 2k and 2k+1.

k is a leaf iff 2k > n. Parent of node i = i // 2 i.e. $\left\lfloor \frac{i}{2} \right\rfloor$



In a heap, the item A@1 is the largest item in the array. If we replace A@1 with A@n we destroy the heap property at node 1. But the subtrees at A@2 and A@3, i.e. the subtrees of the children of node 1 are still heaps. To restore the heap property we must 'sift down' the value of the node at 1 down to its proper position in the array. This is what the procedure Heapify (the key procedure in Heapsort), in effect, does.

We can describe Heapsort in template form as follows;
Assume we have array attribute A of items of type G with A.lower = 1 and A.upper = n.

```
Heapsort   is—the array A[1..n]
      local
                  ...
      do
            <Build or convert A into a Heap>
            from
                  i := n
            until
                  i = 1
            loop
                  <Exchange A@1 and A@i>
                  i := i-1
                  Heapify(1, i)
            end
      end—Heapsort
```

In initially building a Heap we also use Heapify;

We can express Heapify as,

```
Heapify (i, j : INTEGER) is
        --Heapify the array segment A[i .. j]
        -- i.e. Convert A[i .. j] into a heap
        local...
        do
                if  i is not a leaf and
                        if a child of i contains a larger item than i does then
                                Exchange A@i and A@k  -- k is the largest child
                                Heapify(k,j) -- Heapify the 'subtree' A[k .. j]
                end
        end -- Heapify
```

In detail;

```
Heapify (i, j : INTEGER) is
        --Heapify the array segment A[i .. j]
        local
                k : INTEGER
        do
                k := 2*i
                if  k <= j   then      --  if i is not a leaf in A[i .. j]
                        if k<j and then A.item(k) < A.item(k+1) then
                                k := k+1
                        end
                        if A.item(i) < A.item(k) then
                                Exchange (i,k)
                                Heapify (k,j)      -- Heapify the 'subtree' A[k .. j]
                        end
                end
        end -- Heapify
```

Non-Recursive version of Heapify

With a non-recursive version of Heapify, we can get non-recursive version of Heapsort

```
Heapify (i_val, j :INTEGER) is
    local
        v : G  -- items of type G
        i,k : INTEGER
    do
        i := i_val
        k := 2*i
        if  k < j and then A.item(k) < A.item(k+1) then
            k := k+1
        end -- k is the largest child of i (if any)
        from
            v := A.item(i)
        until
            k > j or else v >= A.item(k)        -- k is a leaf and heap property OK
        loop
            A.put(A.item(k), i)
            i := k
            k := 2*i
            if k<j and then A.item(k) < A.item(k+1) then
                k := k+1
            end
        end loop
        A.put(v,i)
    end -- Heapify (Non-Recursive)
```

Build_Heap

To create a heap in the first place we use Build_Heap. Starting from the 'leaves', we build larger and larger heaps until the whole array is a Heap.

At a typical stage, we are adding node i to Heaps already formed, i.e. the subtrees rooted at 2*i and 2*i+1 will be heaps. In making a Heap at i, we Heapfiy A[i .. n], i.e. we 'sift down' node i through the appropriate subtree of node i. In detail,

```
    Build_Heap is
        local
            k : INTEGER
        do
            from
                k :=  n//2
            until
                k = 0
            loop
                Heapify(k,n)
                k := k-1
            end
    end -- Build_Heap
```

<u>Example</u>:

By using Heapsort, sort (by hand) the following sequence:

44 55 12 42 94 18 06 67 **<u>Solution</u>: [see Handout]**

Performance of HeapSort

Heapsort is an O(n*log n) algorithm, even in the worst case.

Consider the worst case: In creating the heap by Build_Heap

$\left\lceil \frac{n}{2} \right\rceil$ **items are moved down 0 positions down (the leaves)**

$\left\lceil \frac{n}{4} \right\rceil$ **" " " 1 "**

$\left\lceil \frac{n}{2^i} \right\rceil$ **" " (i-1) "**

Total Item moves (in worse case) for Build_Heap

$$\sum_{i=1}^{\lceil \log n \rceil} \frac{n}{2^i} * (i-1)$$

$$= n * \sum_{i=1}^{\lceil \log n \rceil} \frac{i-1}{2^i}$$

$$= k * n$$

<u>Note</u>:

$$\sum_{i=1}^{\infty} \frac{1}{2^i} = 1$$

also

$$\sum_{i=1}^{\infty} \frac{i}{2^i} \quad \textbf{converges}$$

(by Ratio Test, $\lim_{j \to \infty} \frac{a_{j+1}}{a_j} < 1$**)**

tf. Build_Heap is at worst O(n)

After the Heap is created we sort by continually calling Heapify. Each call Heapify(1,i) depends only on the level of i,

i.e. the item at the root (node 1) is sifted down at most ***log i*** **positions. Summing over the loop we get the worst case for the sorting loop is** $\sum_{i=1}^{n} \log i$ **which is O(n*log n) [see handout]**

Therefore, Heapsort is O(n) + O(n*log n) = O(n*log n)

```
class       HEAP_SORTER [G -> COMPARABLE]

feature

    sort (a0: ARRAY [G]; low, high: INTEGER) is
        do
            a := a0;
            base := low - 1;
            n := high - base;
            heapsort
        end ;

feature  {NONE}

    a: ARRAY [G];
    base: INTEGER;
    n: INTEGER;


    heapsort is  -- Sort the array A[low..high] i.e. A[Base+1..Base + n]
        local
            i: INTEGER
        do
            build_heap;
            from
                i := n
            until
                i = 1
            loop
                exchange (base + 1, base + i);
                i := i - 1;
                heapify (base + 1, base + i)
            end
        end ;


    exchange (i, j: INTEGER) is
        local
            it: G
        do
            it := a.item (i);
            a.put (a.item (j), i);
            a.put (it, j)
        end ;
```

```
	build_heap is
		local
			k: INTEGER
		do
			from
				k := n // 2
			until
				k = 0
			loop
				heapify (base + k, base + n);
				k := k - 1
			end
		end ;

	heapify (i_val, j: INTEGER) is
		local
			v: G;
			i, k: INTEGER
		do
			i := i_val;
			k := 2 * i;
			if  k < j and then  a.item (k) < a.item (k + 1) then
				k := k + 1
			end ;
			from
				v := a.item (i)
			until
				k > j or else  v >= a.item (k)
			loop
				a.put (a.item (k), i);
				i := k;
				k := 2 * i;
				if  k < j and then  a.item (k) < a.item (k + 1) then
					k := k + 1
				end
			end ;
			a.put (v, i)
		end ;

end  -- class HEAP_SORTER
```