II: SORTING & ORDER STATISTICS

Summerizes the running times of the sorting algorithms in this chapters 2 & 6-8. as:

Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$O(n \lg n)$	$\Theta(n \lg n)$
Heap sort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting sort	$\Theta(k+n)$	$\Theta(k+n)$
Radix sort	$\Theta(d(n+k))$	$\Theta(d(n+k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case).

6: Heapsort

→ Heapsort : like merge sort, unlike insertion sort.
→ running time is $O(n \lg n)$.
→ The term "heap" was originally coined in the context of heapsort, but it has since come to refer to "garbage-collected storage" such as programming language Java & Lisp provides

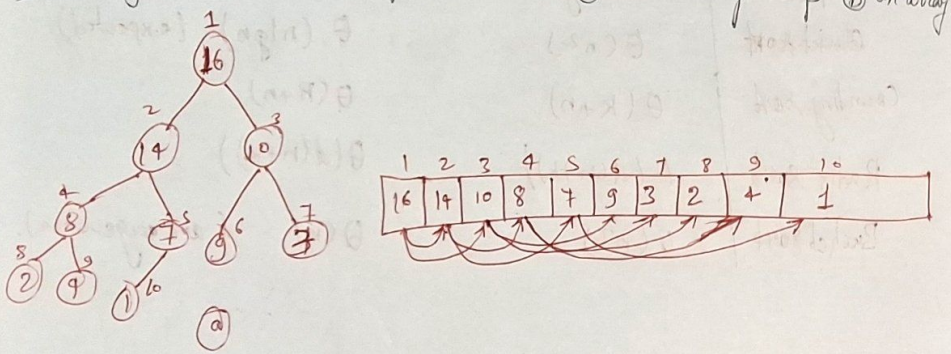
6.1: Heaps (Heap data structure)

→ Heap A (not garbage-collected storage) is a nearly complete binary tree.
→ Height of node = No. of edges on a longest simple path from the node down to a leaf.
→ Height of heap = height of root = $\Theta(\lg n)$

→ A heap can be stored as an array A.

* Root of tree is A[1].
* Parent of A[i] = A[⌊i/2⌋]
* Left child of A[i] = A[2i]
* Right child of A[i] = A[2i+1]
* Computing is fast with binary representation implementation.

Ex:- Figure 6.1: A max-heap viewed as (a) a binary heap (b) an array.



→ Heap property

* Max-heap (largest element at root): ∀ i = nodes excluding the root, A[PARENT(i)] ≥ A[i].
* Min-heap (smallest element at root): ∀ nodes i, excluding the root, A[PARENT(i)] ≤ A[i].

→ By induction & transitivity of ≤, the max-heap property guarantees that the maximum element of a max-heap is at the root. Similar argument for min-heap.

→ Note: In general, heaps can be k-ary tree instead of binary.

Exercises:

6.1-1: What are the minimum & maximum numbers of elements in a heap of height h?

→ Since a heap is an almost-complete binary tree (complete at all levels except possibly the lowest), it has at most $(2^{h+1}-1)$ elements [if it is complete] &

→ At least $2^h - 1 + 1 = 2^h$ elements [if the lowest level has just 1 element & the other levels are complete].

6.1-2: Show that an n-element heap has height $\lfloor \lg n \rfloor$.

Given an n-element heap of height h, we know from Exercise 6.1-1 that

$$2^h \le n \le 2^{h+1} - 1 < 2^{h+1}$$

Thus, $h \le \lg n < h+1$. Since h is an integer, $h = \lfloor \lg n \rfloor$ (by definition of $\lfloor \rfloor$).

6.1-3: Show that in any subtree of a max-heap, the root of the subtree contains the largest value occuring anywhere in the subtree.

If the largest element in the subtree were somewhere other than the root, it has a parent that is in the subtree. So, it is larger than it's parent. So, the heap property is violated at the parent of the maximum element in the subtree. We must have A[PARENT(i)] ≥ A[i].

6.1-5: Is an array that is in sorted order a min-heap?

Yes, for any index i, both LEFT(i) & RIGHT(i) are larger & thus the elements indexed by them are greater or equal to A[i] (because the array is sorted).

6.1-4: Where in a max-heap might the smallest element reside, assuming that all elements are distinct?

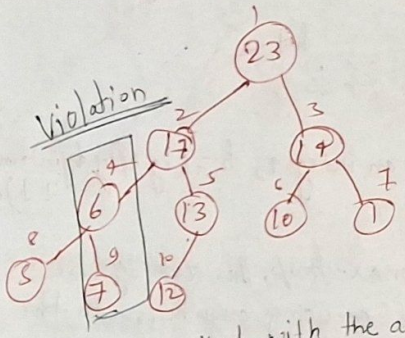
In any of the leaves, that is elements with index $\lfloor n/2 \rfloor + 1$

i.e. the leaves are the nodes indexed by $\lfloor n/2 \rfloor + 1 \to \lfloor n/2 \rfloor + 2 \to \dots \to n$.

i.e. in the second half of the heap array.

6.1-6: Is the array with values $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$ a max-heap?

No, Since PARENT (7) is 6 in the array. This violates the max-heap property.



6.1-7: Show that, with the array representation for storing an n-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$.

Let's take the left child of the node indexed by $\lfloor n/2 \rfloor + 1$.

$$\begin{aligned} \text{LEFT}(\lfloor n/2 \rfloor + 1) &= 2(\lfloor n/2 \rfloor + 1) \\ &> 2(n/2 - 1) + 2 \\ &= n - 2 + 2 \\ &= n. \end{aligned}$$

Since the index of the left child is larger than the number of elements in the heap, the node doesn't have childrens & thus is a leaf. Same goes for all nodes with larger indices.

→ Note: If we take element indexed by $\lfloor n/2 \rfloor$, it will not a leaf. In case of even no of nodes, it will have a left child with index n & in the case of odd no. of nodes it will have a left child with index $n-1$ & a right child with index n.

This makes the no. of leaves in a heap-size n equal to $\lceil n/2 \rceil$.

6.2: Maintaining the heap property

MAX-HEAPIFY is important for manipulating max-heaps. It is used to maintain the max-heap property.

→ Before MAX-HEAPIFY, A[i] may be smaller than its children.
→ Assume left & right subtrees of i are max-heaps.
→ After MAX-HEAPIFY, subtree rooted at i is a max-heap.

MAX-HEAPIFY (A, i, n → heap-size[A])

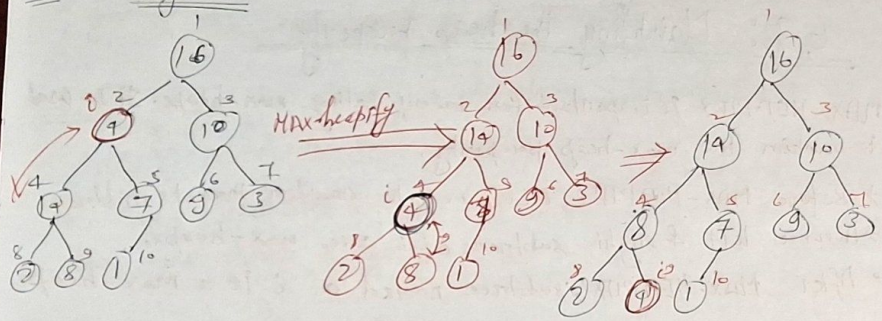
```
① l = LEFT(i)
② r = RIGHT(i)
③ if l ≤ n && A[l] > A[i]
④     then largest = l
⑤     else largest = i
⑥ if r ≤ n && A[r] > A[largest]
⑦     then largest = r
⑧ if largest ≠ i
⑨     then exchange A[i] ⇔ A[largest]
⑩          MAX-HEAPIFY (A, largest, n).
```

The way MAX-HEAPIFY works:

→ Compare A[i], A[LEFT(i)] & A[RIGHT(i)].
→ If necessary, swap A[i] with the larger of the 2 children to preserve heap property.

→ Continue this process of comparing & swapping down the heap, until subtree rooted at i is max-heap. If we hit a leaf, then the subtree rooted at the leaf is trivially a max-heap.

Ex:- Figure 6.2:



Time Analysis:

The running time of MAX-HEAPIFY by the recurrence

$$T(n) \leq T(2n/3) + \Theta(1)$$

Using master theorem,

$a = 1$, $b = 3/2$, $k = 0$, $p = 0$

then

$a \neq b^k$
$a = b^k$
$1 = (3/2)^0$

} case 2: $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$

$$\boxed{T(n) = \Theta(\lg n)}$$ Ans.

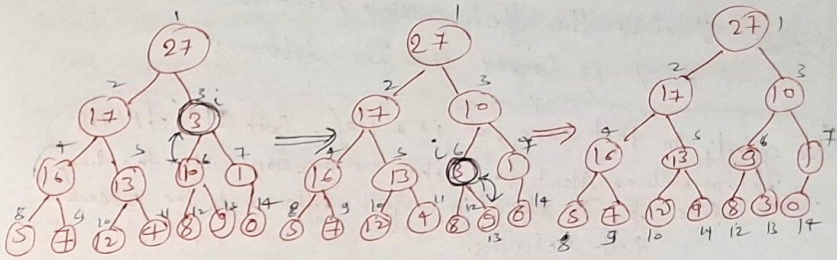
& Worst-Case $T(n) = O(\lg n)$

& height $(h) = O(h)$.

Exercises:

6.2-1: Using figure 6.2 as a model, illustrate the operation of MAX-HEAPIFY $(A, 3)$ on the array

$A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$



6.2-2: Starting with the procedure MAX-HEAPIFY, write pseudocode for the procedure MIN-HEAPIFY (A, i), which performs the corresponding manipulation on a min-heap. How does the running time of MIN-HEAPIFY compare to that of MAX-HEAPIFY?

```
MIN-HEAPIFY (A,i,n)
    l = LEFT(i)
    r = RIGHT(l)
    if l ≤ n && A[l] < A[i]
        smallest = l
    else smallest = i
    if r ≤ n && A[r] < A[smallest]
        smallest = r
    if smallest ≠ i
        exchange A[i] with A[smallest]
        MIN-HEAPIFY (A, smallest, n).
```

Time complexity is the same as $T(n) = \Theta(\lg n)$

6.2-4: What is the effect of calling MAX-HEAPIFY (A, i) for $i > A.heap\text{-}size/2$?

No effect, The comparisions carried out, A[i] is found to be largest & the procedure just returns.

6.2-3: What is the effect calling MAX-HEAPIFY (A, i) when element A[i] is larger than its children?

No effect, In that case, it is a leaf. Both LEFT & RIGHT return values that fail the comparison with the heap size & i is stored in largest. Afterwards the procedure just returns.

6.2-5: The code for MAX-HEAPIFY is quite efficient in terms of constant factors, except possibly for the recursive call in line 10, which might cause some compilers to produce inefficient code. Write an efficient MAX-HEAPIFY that uses an iterative control construct (a loop) instead of recursion.

```
MAX-HEAPIFY (A, i, n)
  while true
    l = LEFT(i)
    r = RIGHT(i)
    if l ≤ n && A[l] > A[i]
        largest = l
    else largest = i
    if r ≤ n && A[r] > A[largest]
        largest = r
    if largest == i
        return
    exchange A[i] with A[largest]
    i = largest.
```

6.2-6: Show that the worst-case running time of MAX-HEAPIFY on a heap of size n is $\Omega(\lg n)$. (Hint: For a heap with n nodes, give node values that cause MAX-HEAPIFY to be called recursively at every node on a simple path from the root down to a leaf.)

Consider the heap resulting from A where A[i] = 1 if A[i] = 2 for $2 \le i \le n$. Since 1 is the smallest element of the heap, it must be swapped through each level of the heap until it is a leaf node.
Since the heap has height $\lfloor \lg n \rfloor$, MAX-HEAPIFY has worst-case time $\Omega(\lg n)$.

6.3: Building a heap

BUILD-MAX-HEAP (A, n)

① for $i = \lfloor n/2 \rfloor$ down to 1

② do MAX-HEAPIFY (A, i, n)

Correctness:

Loop Invariant: At start of every iteration of for loop, each node $i+1, i+2, \ldots, n$ is root of MAX-heap.

→ Initialization: We know that each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf, which is the root of a trivial max-heap. Since $i = \lfloor n/2 \rfloor$ before the first iteration of for loop, the invariant is initially true.

→ Maintenance: children of node i are indexed higher than i, so by the loop invariant, they are both roots of max-heaps. Correctly assuming that $i+1, i+2, \ldots, n$ are all roots of max-heap, MAX-HEAPIFY makes node i a max-heap root. Decreasing i reestablishes the loop invariant at each iteration.

→ Termination: When $i=0$, the loop terminates. By the loop invariant, each node, notably node 1, is the root of a max-heap.

Analysis:

→ Simple bound: $O(n)$ calls to MAX-HEAPIFY, each of which takes $O(\lg n)$ time $\Rightarrow O(n \lg n)$.

→ Tighter Bound Analysis: Observation: Time to run MAX-HEAPIFY is linear in the height of the node it's run on, & most nodes have small heights. Have $\leq \lceil n/2^{h+1} \rceil$ nodes of height h & height of heap is $\lfloor \lg n \rfloor$. The time required by MAX-HEAPIFY when called on a node of height h is $O(h)$, so the total cost of BUILD-MAX-HEAP is

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

Evaluate the last summation by substituting $x = 1/2$ in the formula (A.8). $\sum_{k=0}^{\infty} k x^k$, which yields

$$\sum_{h=0}^{\infty} h/2^h = \frac{1/2}{(1-1/2)^2} = 2.$$

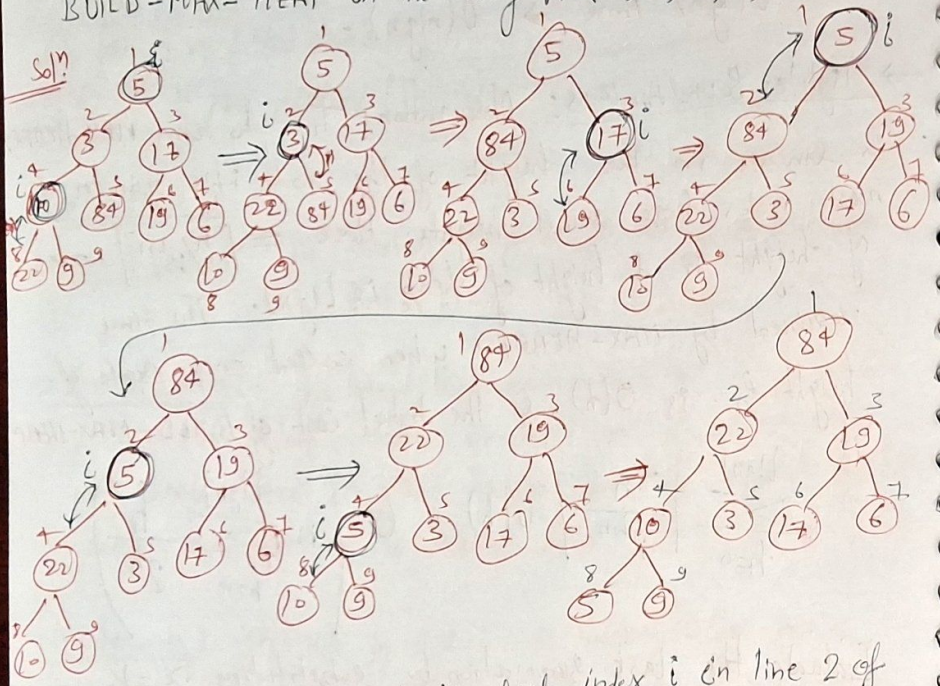
Thus, the running time of BUILD-MAX-HEAP is $O(n)$.

→ Building a min-heap from an unordered array can be done by calling MIN-HEAPIFY instead of MAX-HEAPIFY, also taking linear time.

Exercises

6.3-1: Using Figure 6.3 as a model, illustrate the operation of BUILD-MAX-HEAP on the array $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

Soln:



6.3-2: Why do we want the loop index i in line 2 of BUILD-MAX-HEAP to decrease from $\lfloor A.length/2 \rfloor$ to 1 rather than increase from 1 to $\lfloor A.length/2 \rfloor$?

Otherwise we won't be allowed to call MAX-HEAPIFY, since it will fail the condition of having the subtrees be max-heaps. That is, if we start with 1, there is no guarantee that A[2] & A[3] are roots of max-heaps.

6.3-3: Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.

We know that the leaves of a heap are the nodes indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.

Note that these elements corresponds to the second half of the heap array (plus the middle element if n is odd). Thus, the number of leaves in any heap of size n is $\lceil n/2 \rceil$. Let's prove by induction. Let n_h denote the number of nodes at height h. The upper bound holds for the base since $n_0 = \lceil n/2 \rceil$ is exactly the number of leaves in a heap of size n.

Now assume it holds for $h-1$. We have prove that it also holds for h. Note that if n_{h-1} is even each node at height h has exactly two children, which implies $n_h = n_{h-1}/2 = \lfloor n_{h-1}/2 \rfloor$. If n_{h-1} is odd, one node at height h has one child & the remaining has 2 children, which also implies $n_h = \lfloor n_{h-1}/2 \rfloor + 1 = \lceil n_{h-1}/2 \rceil$. Thus, we have,

$$n_h = \left\lceil \frac{n_{h-1}}{2} \right\rceil$$

$$\leq \left\lceil \frac{1}{2} \cdot \left\lceil \frac{n}{2^{(h-1)+1}} \right\rceil \right\rceil$$

$$= \left\lceil \frac{1}{2} \cdot \left\lceil \frac{n}{2^h} \right\rceil \right\rceil$$

$$= \left\lceil \frac{n}{2^{h+1}} \right\rceil .$$

{OR}

An alternative solution relies on four facts:

① Every node not on the unique simple path from the last leaf to the root is the root of a complete binary subtree.

② A node that is root of a complete binary subtree & has height h is the ancestor of 2^h leaves.

③ By Exercise 6.1-7, an n-element heap has $\lceil n/2 \rceil$ leaves.

④ For non-negative reals a & b, we have $\lceil a \rceil \cdot b \geq \lceil ab \rceil$.

* The proof is by contradiction. Assume that an n-element heap contains at least $\lceil n/2^{h+1} \rceil + 1$ nodes of height h.
* Exactly one node of height h is on the unique simple path from the last leaf to the root, & the subtree rooted at this node has at least one leaf (that being the last leaf).
* All other hands, nodes of height h, of which the heap contains at least $\lceil n/2^{h+1} \rceil$, are the roots of complete binary subtrees, & each node is the root of a subtree with 2^h leaves.
* Moreover, each subtree whose root is at height h is disjoint.

Therefore, the number of leaves in the entire heap is at least

$$\left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot 2^h + 1 \geq \left\lceil \frac{n}{2^{h+1}} \cdot 2^h \right\rceil + 1$$

$$= \left\lceil \frac{n}{2} \right\rceil + 1.$$

which contradicts the property that an n-element heap has $\lceil n/2 \rceil$ leaves.

6.4: The heapsort Algorithm

HEAPSORT(A, n)

① BUILD-MAX-HEAP(A, n) ⟶ $O(n)$

② for $i = n$ downto 2 ⟶ $(n-1)$ times

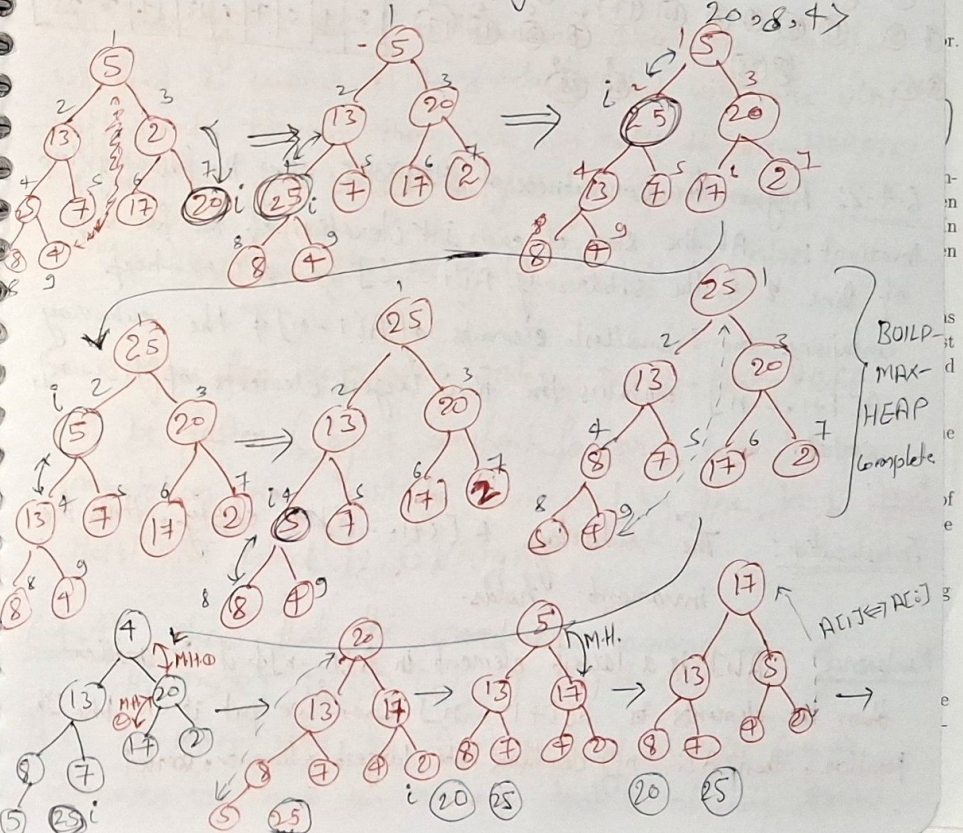
③ do exchange $A[1] \Leftrightarrow A[i]$ ⟶ $O(1)$ times

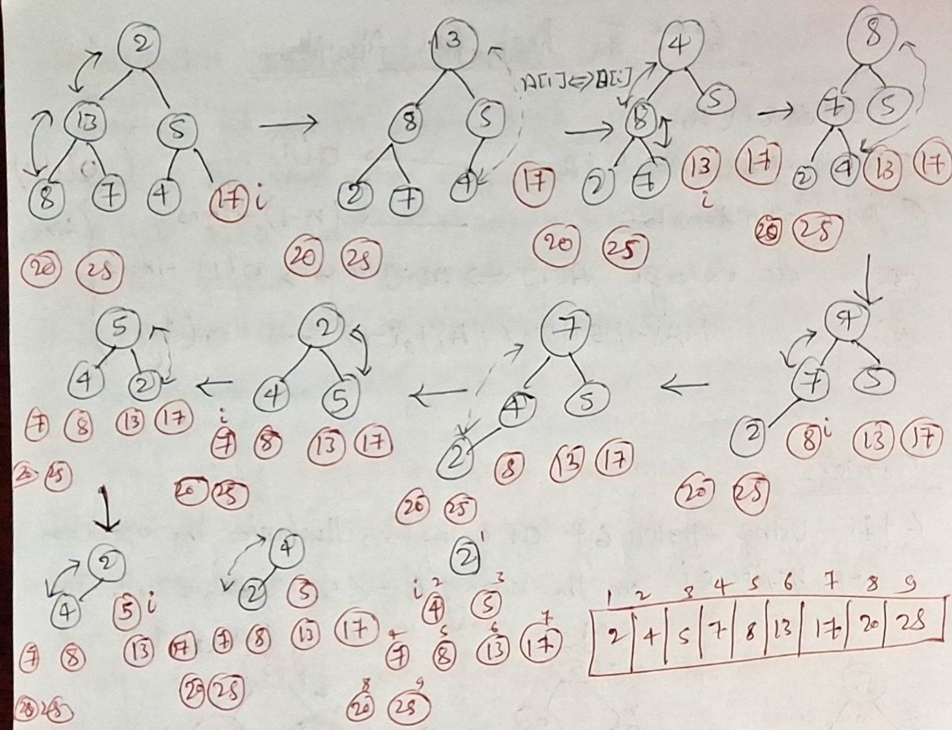
④ MAX-HEAPIFY(A, 1, i-1) ⟶ $O(\lg n)$

③ & ④: $O(1) + O(\lg n)$ times, $O(n) + O(n \lg n)$

Exercises

6.4-1: Using figure 6.4 as a model, illustrate the operation of HEAPSORT on the array $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$





6.4-2: Argue the correctness of HEAPSORT using the following loop invariant: At the start of each ~~int~~ iteration of the for loop of line 2-5, the subarray A[1 ‥ i] is a max-heap containing the i smallest elements of A[1 ‥ n] & the subarray A[i+1 ‥ n] contains the n-i largest elements of A[1 ‥ n], sorted.

Proof:

Initialization: The subarray A[i+1 ‥ n] is empty, thus the invariant holds.

Maintanance: A[1] is a largest element in A[1 ‥ n] & it is smaller than the elements in A[i+1 ‥ n]. When we put it in the ith position, then A[i ‥ n] contains the largest elements, sorted.

Decreasing the heap size & calling MAX-HEAPIFY turns A[1 ‥ i-1] into a max-heap. Decreamenting i sets up the invariants for the next iteration.

Termination: After the loop i=1. This means that A[2 ‥ n] is sorted & A[1] is the smallest element in the array, which makes the array sorted.

6.4-3: What is the running time of HEAPSORT on an array A of length n that is already sorted in increasing order? what about decreasing order?

→ Both of them are $\Theta(n \lg n)$.

→ If the array is sorted in increasing order, the algorithm will need to convert it to a heap that will take $O(n)$,

→ Afterwards, however, there are n-1 calls to MAX-HEAPIFY & each one will perform the full $\lg k$ operations.

Since $\sum_{k=1}^{n-1} \lg k = \lg((n-1)!) = \Theta(n \lg n)$

Same goes for decreasing order. BUILD-MAX-HEAP will be faster (by a constant factor), but the computation time will be dominated by the loop in HEAPSORT, which is $\Theta(n \lg n)$.

6.4-4: Show that the worst-case running time of HEAPSORT is $\Omega(n \lg n)$

This is essentially the first part of exercise 6.4-3. Whenever we have an array that is already sorted,

we take linear time to convert it to a max-heap & then $n \lg n$ time to sort it.

*6.4-5: Show that when all elements are distinct, the best case running time of HEAPSORT is $\Omega(n \lg n)$.

This proved to be quite tricky. My initial solution was wrong.

→ Also, heapsort appeared in 1964, but the lower bound was proved by Schaffer & Sedgewick in 1992.

→ Let's assume that the heap is a full binary tree with $n = 2^k - 1$. There are 2^{k-1} leaves & $2^{k-1} - 1$ inner nodes.

→ Let's look at sorting the first 2^{k-1} elements of the heap. Let's consider their arrangement in the heap & color the leaves to be red & inner nodes to be blue. The colored nodes are a subtree of the heap (otherwise there would be contradiction). Since there are 2^{k-1} colored nodes, at most 2^{k-2} are red, which means that at least $2^{k-2} - 1$ are blue.

→ While the red nodes can jump directly to the root, the blue nodes need to travel up before they get removed. Let's count to the no. of swaps to move the blue nodes to ~~travel up before~~ root. The minimal case of swaps is when

① there are $2^{k-2} - 1$ blue nodes &

② they are arranged in a binary tree.

If there are d such blue nodes, then there would be $i = \lg d$ levels, each containing 2^i nodes with length i.

Thus the no. of swaps is

$$\sum_{i=0}^{\lg d} i 2^i = 2 + (\lg d - 2) 2^{\lg d} = \Omega(d \lg d).$$

And now for a lazy (but cute) trick, we'll have figured out a tight bound on sorting half of the heap. We have following recurrence:

$$T(n) = T(n/2) + \Omega(n \lg n)$$

Apply master theorem,

$a = 1, b = 2$

$a < b^k$ } case 3: $\boxed{T(n) = \Omega(n \lg n)}$ Ans

Because, $f(n) = \Omega(n \lg n n) = \Omega(n^{\log_b a + \epsilon})$ with $\epsilon > 0$.

6.5: Heap implementation of priority queue

Priority Queue

→ Maintains a dynamic set S of elements.
→ Each set elements has a key - an associated value
→ Max-priority queue supports dynamic-set operations:
  - INSERT(S, x): insert element x into set S
  - MAXIMUM(S): returns element of S with largest key.
  - EXTRACT-MAX(S): removes & returns elements of S with largest key.
  - INCREASE-KEY(S, x, k): increases values of element x's key to k. Assume k ≥ x's current key value.

→ Applicat": schedule jobs on shared computer. max-queue
→ Min-priority queue supports similar operations. (k ≤ x's).
→ Applicat" of min-priority queue: event-driven simulation.

Finding the maximum element

```
HEAP-MAXIMUM(A)
return A[1]
```

Time: Θ(1).

Extracting max element

```
HEAP-EXTRACT-MAX (A,n)
if n<1
  then error "heap underflow"
max = A[1]
A[1] = A[n]
MAX-HEPIFY(A, 1, n-1) ⟶ O(lgn)
return max
```

} O(lgn)

Increasing key value

```
HEAP-INCREASE-KEY (A,i,key)
if key < A[i]
   then error "new key is smaller than current key"
A[i] = key
While i>1 && A[PARENT(i)] < A[i]
      do exchange A[i] ⟷ A[PARENT(i)]
         i = PARENT(i)
```

Analysis: Upward path from node i has length O(lgn) in an n-element heap.

Time: O(lgn).

Inserting into the heap

```
MAX-HEP-INSERT (A, key, n)
A[n+1] = -∞
HEAP-INCREASE-KEY (A, n+1, key)
```

Analysis: Constant time assignments + time for HEAP-INCRESE-KEY.

Time: O(lgn).

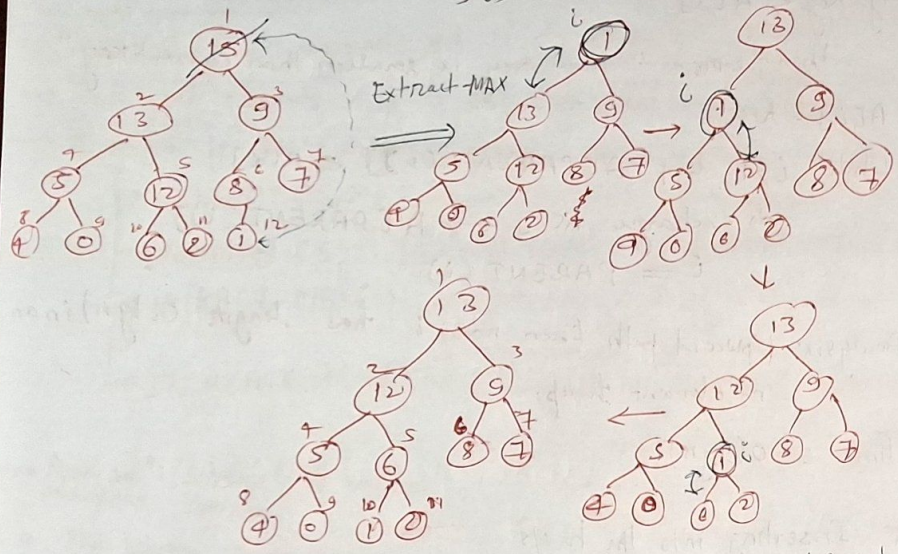
* Min-priority queue operations are implemented similarly with min-heaps.

Exercises

6.5-1:

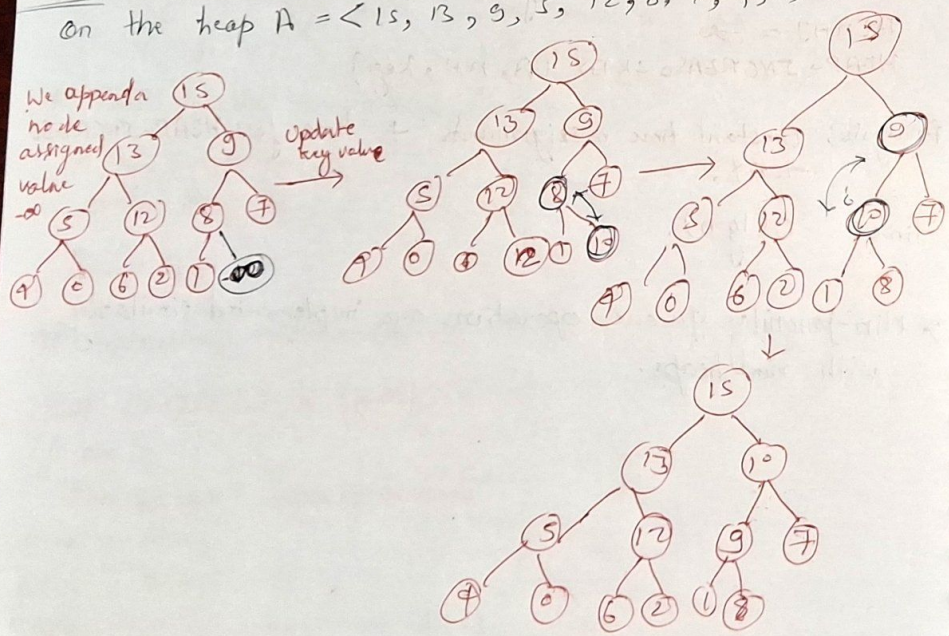
Illustrate the operation HEAP-EXTRACT-MAX on the heap

$A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$



6.5-2:

Illustrate the operation MAX-HEAP-INSERT (A, 10) on the heap $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$



6.5-3:

Write pseudocode for the procedure HEAP-MINIMUM, HEAP-EXTRACT-MIN, HEAP-DECEASE-KEY & MIN-HEAP-INSERT that implement a min-priority queue with a min-heap.

```
# HEAP-MINIMUM

HEAP-MINIMUM(A)
    return A[1].

# HEAP-EXTRACT-MIN(A)
    if n < 1                          // n = A.heap-size.
    then error "heap underflow".
    min = A[1]
    A[1] = A[n]
    A[n] = A[n]-1
    MIN-HEAPIFY(A, 1)
    return min

# HEA-DECREASE-KEY(A, i, key)
    if key > A[i]
        then error "new key is larger than current key"
    A[i] = key
    while i > 1 && A[PARENT(i)] > A[i]
        exchange A[i] with A[PARENT(i)]
        i = PARENT(i)

# MIN-HEAP-INSERT(A, key)
    A[n] = A[n]+1                     // n = n+1
    A[n] = ∞                          // n = ∞
    HEAP-DECREASE-KEY(A, n, key).
```

6.5-4: Why do we bother setting the key of the inserted node to $-\infty$ in line 2 of MAX-HEAP-INSERT when the next thing we do is increase its key to the desired value?

In orders to pass the guard clause. Otherwise we have to drop the check if key $\leq$ A[i].

6.5-5: Argue the correctness of HEAP-INCREASE-KEY using the following loop invariant:

At the start of each iteration of the while loop of lines 4-6, the subarray A[1..n] satisfies the max-heap property, except that there may be one violation: A[i] may be large than A[PARENT(i)]. [n = A.heapsize]

You May assume that the subarray A[1..n] satisfies the max-heap property at the time HEAP-INCREASE-KEY is called.

Initialization: A is a heap except that A[i] might be larger that it's parent, because it has been modified. A[i] is larger than its children, because otherwise the guard clause would fail & the loop will not be entered (the new value is larger than the old value & the old value is larger than the children).

Maintenance: When we exchange A[i] with its parent, the MAX-heap property is satisfied except that now A[PARENT(i)] might be larger than its parent. Changing i to its parent maintains the invariant.

Termination: The loop terminates whenever the heap is exhausted or the max-heap property for A[i] & its parent is preserved. At the loop termination, A is a max-heap.

6.5-6: Each exchange operation on line-5 of HEAP-INCREASE-KEY typically requires three assignments. Show how to use the idea of the inner loop of INSERTION-SORT to reduce the three assignments down to just one assignment.

```
HEAP-INCREASE-KEY (A, i, key)
    if key > A[i]
        then error " new key is smaller than the
                     current key"
    While i > 1 && A[PARENT(i)] < key
        A[i] = A[PARENT(i)]
        i = PARENT(i)
    A[i] = key
```

6.5-7: Show how to implement a first-in, first-out queue with ~~queue~~ a priority queue. Show how to implement a stack with a priority queue. (Queue & stacks are defined in section 10.1).

→ Both are simple, For stack we keep adding elements in increasing priority, while in a queue we add them in decreasing priority.

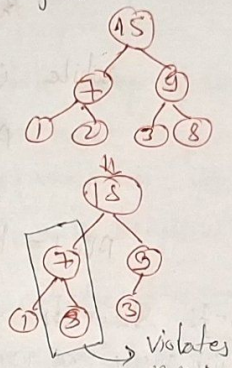
→ For stack we can set the new priority to HEAP-MAXIMUM(A)+1. For queue we need to keep track of it & decrease it on every insertion.

→ Both are not very sufficient, Furthermore, if the priority can overflow or underflow. so will eventually need to reassign priorities.

6.5-8: The operation HEAP-DELETE (A,i) deletes the item in node i from heap A. Give an implementation of HEAP-DELETE that runs in $O(\lg n)$ time for an n-element max-heap.

```
HEA-DELETE (A,i)
  if A[i] > A[n]
      A[i] = A[n]
      MAX-HEAPIFY (A,i)
  else
      HEAP-INCREASE-KEY (A, i, A[n])
  n = n - 1
```

Note: The following algorithm is wrong/fail. For example, a given an array A = <15, 7, 9, 1, 2, 3, 8>
Delete A[5] = 2, then it will fail.



Now,

```
HEAP-DELETE (A,i)
  A[i] = A[n]
  A[n] = A[n] - 1
  MAX-HEAPIFY (A,i)
```

6.5-9: Give an $O(n \lg k)$-time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the I/P lists. (Hint: Use a min-heap for k-way merging.)

We take one element of each list & put it in a min-heap. Along with each element we have to track which list we took it from. When merging, we take a minimum element from the heap & insert another element off the list it came from (unless the list is empty). We continue until we empty the heap.

We have n steps & at each step we're doing an insertion into the heap, which is $\lg k$.

Suppose that sorted lists on I/P are all nonempty, we have following pseudocode:

```
def MERGE-SORTED-LISTS (lists)
    n = lists.length
    let lowest-from-each be an empty array
    for i = 1 to n
        add (lists[i][0], i) to lowest-from-each
        delete lists[i][0].
A = MIN-HEAP(lowest-from-each)
let merged-lists be an empty array
while not A.EMPTY()
        element-value, index-of-list = HEAP-EXTRACT-
                                       MIN(A)
        add element-value to merged-lists
        if lists[index-of-list].length > 0
            MIN-HEAP-INSERT (A, (lists[index-of-list][0],
                                 index-of-list))
            delete lists[index-of-list][0]
return merged-list.
```

6-Problems

6-1: Building a heap using insertion

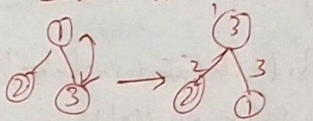
We can build a heap by repeatedly calling MAX-HEAP-INSERT to insert the elements into the heap. Consider the following variation of the BUILD-MAX-HEAP procedure:

```
BUILD-MAX-HEAP'(A)
    A.heapsize = 1
    for i = 2 to A.length
        MAX-HEAP-INSERT(A, A[i]).
```

(a) Do the procedure BUILD-MAX-HEAP & BUILD-MAX-HEAP' always create the same heap when run on the same I/p array? Prove that they do, or provide a counterexample.

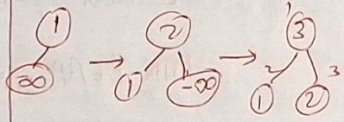
Ex: A = ⟨1, 2, 3⟩

BUILD-MAX-HEAP



A = [3 | 2 | 1]

BUILD-MAX-HEAP'



A = [3 | 1 | 2]

(b) Show that in the worst case, BUILD-MAX-HEAP' requires $\Theta(n \lg n)$ time to build a n-element heap.

Each insert step takes at most $\Theta(\lg n)$, since we are doing it n times, we get a bound on the runtime of $O(n \lg n)$.

$$T = \sum_{i=2}^{n} \lg i = \lg n! \overset{\text{worst-case}}{=} \Theta(n \lg n)$$

$$\geq n/2 \cdot \Theta(\lg n) = \Omega(n \lg n).$$

$$\sum_{i=1}^{n} \Theta(\lfloor \lg i \rfloor) \geq \sum_{i=\lceil n/2 \rceil}^{n} \Theta(\lfloor \lg \lceil n/2 \rceil \rfloor) \geq \sum_{i=\lceil n/2 \rceil}^{n} \Theta(\lfloor \lg(n/2) \rfloor) = \sum_{i=\lceil n/2 \rceil}^{n} \Theta(\lfloor \lg n - 1 \rfloor)$$

6-2 Analysis of d-ary heaps

A d-ary heap is like a binary heap, but (with one possible exception) non-leaf nodes have d children instead of 2 children.

(a) How would you represent a d-ary heap in array?

We can use those 2 following functions to retrieve parent of ith element & jth children of ith element.

d-ARY-PARENT(i)	d-ARY-CHILD(i,j)
return floor $(\lfloor (i-2)/d \rfloor + 1)$	return $d(i-1)+j+1$

obviously $1 \leq j \leq d$. You can verify these functions checking that d-ARY-PARENT(d-ARY-CHILD(i,j)) = i.

Also easy to see is that binary heap is special type of d-ary heap where d=2. If you substitute d with 2, then you will see that they match functions PARENT, LEFT, RIGHT.

(b) What is the height of a d-ary heap of n-elements in terms of n & d?

Since each node has d children, the height of a d-ary heap with n nodes is $\Theta(\log_d n)$.

(c) Give an efficient implementation of EXTRACT-MAX in a d-ary max-heap. Analyze its running time in terms of d & n.

d-ARY-HEAP-EXTRACT-MAX(A) consists of constant time operations followed by a call to d-ARY-MAX-HEAPIFY(A,i).

The number of times this recursively calls itself is bounded

by the height of the d-ary heap, so that the running time is $O(d \log_d n)$.

```
d-ARY-HEAP-EXTRACT-MAX(A)
    if A.heapsize < 1
        error "heap underflow"
    max = A[1]
    A[1] = A[A.heapsize]
    A.heapsize = A.heapsize - 1
    d-ARY-MAX-HEAPIFY(A, 1)
    return max
```

```
d-ARY-MAX-HEAPIFY(A, i)
    largest = i
    for k = 1 to d
        if d-ARY-CHILD(i,k) ≤ A.heapsize &&
           A[d-ARY-CHILD(i,k)] > A[largest]
            largest = d-ARY-CHILD(i,k)
    if largest != i
        exchange A[i] with A[largest]
        d-ARY-MAX-HEAPIFY(A, largest).
```

(c) Give an efficient implementation of INCREASE-KEY(A,i,k) which flags an error if $k < A[i]$, but otherwise sets $A[i] = k$ & then updates the d-ary max-heap structure appropriately. Analyze its running time in terms of d & n.

The runtime is $O(\log_d n)$ since the while loop runs at most as many times as the height of the d-ary array.

```
d-ARY-INCREASE-KEY(A, i, key)
    if key < A[i]
        error "new key is smaller than current key"
    A[i] = key
    while i > 1 && A[d-ARY-PARENT(i)] < A[i]
        exchange A[i] with A[d-ARY-PARENT(i)]
        i = d-ARY-PARENT(i)
```

(d) Give an efficient implementation of INSERT in a d-ary max-heap. Analyze the running time in terms of d & n.

```
d-ARY-MAX-HEAP-INSERT(A, key)
    A.heapsize = A.heapsize + 1
    A[A.heapsize] = key
    i = A.heapsize
    while i > 1 && A[d-ARY-PARENT(i)] < A[i]
        exchange A[i] with A[d-ARY-PARENT(i)]
        i = d-ARY-PARENT(i)
```

Note:

Height : $\log_d n = \left(\frac{\log_2 n}{\log_2 d}\right)$

Complexity : EXTRACT-MAX : $d \log_d n$

INSERT : $\log_d n$

INCREASE-KEY : $\log_d n$

d-ARY-PARENT(i) → $\lfloor (i-2)/d + 1 \rfloor$

d-ary-child(i,j) → $d(i-j) + j + 1$

d-A-PARENT(d-ary-child(i,j)) = i

$\forall\ 1 \le j \le d.$

by the height of the d-ary heap, so that the running time is $O(d \log_d n)$.

```
# d-ARY-HEAP-EXTRACT-MAX(A)
    if A.heapsize < 1
        error "heap underflow"
    max = A[1]
    A[1] = A[A.heapsize]
    A.heapsize = A.heapsize - 1
    d-ARY-MAX-HEAPIFY(A,1)
    return max
# d-ARY-MAX-HEAPIFY(A,i)
    largest = i
    for k = 1 to d
        if d-ARY-CHILD(i,k) ≤ A.heapsize &&
           A[d-ARY-CHILD(i,k)] > A[largest]
            largest = d-ARY-CHILD(i,k)
    if largest != i
        exchange A[i] with A[largest]
        d-ARY-MAX-HEAPIFY(A, largest).
```

(c) Give an efficient implementation of INCREASE-KEY(A,i,k) which flags an error if $k < A[i]$, but otherwise sets $A[i] = k$ & then updates the d-ary max-heap structure appropriately. Analyze its running time in terms of d & n.

The runtime is $O(\log_d n)$ since the while loop runs at most as many times as the height of the d-ary array.

```
d-ARY-INCREASE-KEY(A,i,key)
    if key < A[i]
        error "new key is smaller than current key"
    A[i] = key
    while i > 1 && A[d-ARY-PARENT(i)] < A[i]
        exchange A[i] with A[d-ARY-PARENT(i)]
        i = d-ARY-PARENT(i)
```

(d) Give an efficient implementation of INSERT in a d-ary max-heap. Analyze the running time in terms of d & n.

```
d-ARY-MAX-HEAP-INSERT(A,key)
    A.heap-size = A.heap-size + 1
    A[A.heapsize] = key
    i = A.heap-size
    while i > 1 && A[d-ARY-PARENT(i)] < A[i]
        exchange A[i] with A[d-ARY-PARENT(i)]
        i = d-ARY-PARENT(i)
```

Note:

Height : $\log_d n = \left(\frac{\log_2 n}{\log_2 d}\right)$

Complexity : EXTRACT-MAX : $d \log_d n$

INSERT : $\log_d n$

INCREASE-KEY : $\log_d n$

* d-A-PARENT(i) → $\lfloor (i-2)/d + 1 \rfloor$
* d-ary-child(i,j) → $d(i-j)+j+1$
* d-A-PARENT(d-ary-child(i,j)) = i $\forall\ 1 \le j \le d$.