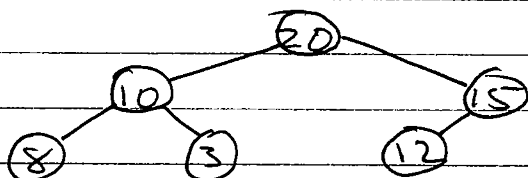
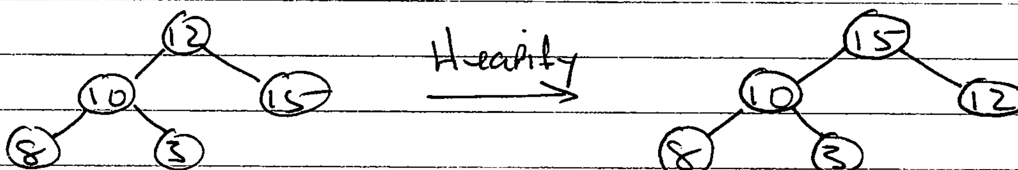
Ex. AFTER BUILD-HEAP (A) :

20 10 15 8 3 12 |

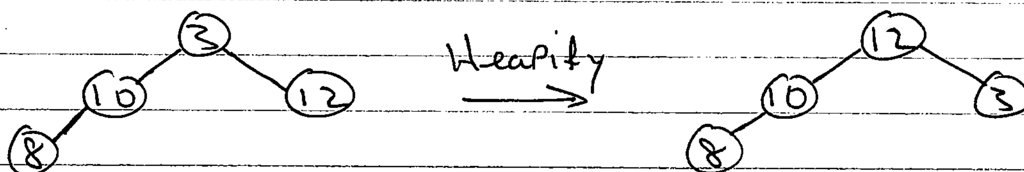


12 10 15 8 3 | 20



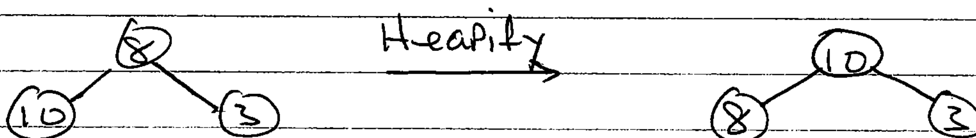
15 10 12 8 3 | 20

3 10 12 8 | 15 20



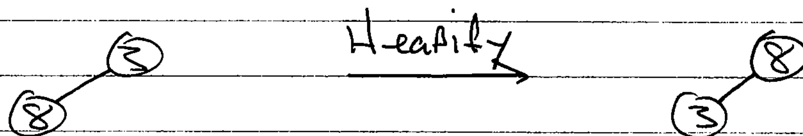
12 10 3 8 | 15 20

8 10 3 | 12 15 20



10 8 3 | 12 15 20

3 8 | 10 12 15 20



8	3 |	10	12	15	20
3 |	8	10	12	15	20

Run Time

The call to Build-Heap takes $\Theta(n)$ time in worst case. Each of the $n-1$ calls to Heapify takes $\Theta(\lg n)$ time in worst case.

Therefore the worst case run time of Heapsort is

$$T(n) = \Theta(n) + (n-1)\Theta(\lg n) = \Theta(n \lg n).$$

Note this is as good as MergeSort (asymptotically speaking), but Heapsort sorts in-place, while MergeSort requires extra memory.

6.5 Priority Queues

A **Priority Queue** is an ADT which maintains a finite set S of elements, each with an associated value called a **key.**

$$x = (\underbrace{\cdot, \cdot, \cdot,}_{\text{satellite data}} \text{key}) \in S$$

$$S = \{ \cdot, \cdot, \cdot, x, \cdot, \cdot, \cdot \}$$

The attribute $key[x]$ defines the "priority" of x in S.

A priority queue supports the following operations

Insert(S,x): insert new element x into S.

Maximum(S): Return element with largest key

ExtractMax(S): Return and delete max element.

IncreaseKey(S,x,k): change $key[x]$ to $k \geq key[x]$.

MORE GENERALLY THE ABOVE ADT WOULD BE CALLED A MAX PRIORITY QUEUE. THE DUAL NOTION OF A MIN PRIORITY QUEUE WOULD HAVE OPERATIONS Insert(S, x) Minimum(S), ExtractMin(S), DecreaseKey(S, x, K).

NOTE: FROM OUR POINT OF VIEW REGARDING ADTs, ExtractMax IS A HYBRID OPERATION - WE COULD DEFINE A PURE MANIPULATION PROCEDURE CALLED DeleteMax(S), WHICH JUST DELETES THE MAXIMUM ELEMENT. ExtractMax(S) IS THEN THE SEQUENCE Maximum(S), DeleteMax(S).

A MAX (RESP. MIN) PRIORITY QUEUE CAN BE IMPLEMENTED EFFICIENTLY AS A MAX (RESP. MIN) HEAP.

HeapMaximum(A)

```
1.) if heap-size[A] < 1
2.)     error "heap underflow"
3.) return A[1]
```

RUN TIME $\Theta(1)$.

HeapExtractMax(A)

```
1.) if heap-size[A] < 1
2.)     error 'heap underflow'
3.) max ← A[1]
4.) A[1] ← A[heap-size[A]]
5.) heap-size[A] ← (heap-size[A] - 1)
6.) Heapify(A, 1)
7.) return max
```

TO OBTAIN THE PURE MANIPULATION PROCEDURE HeapDeleteMax(A), JUST LEAVE OUT LINES (3) AND (7).

THE RUN TIME IS $\Theta(\lg n)$, SINCE ALL LINES TAKE CONSTANT TIME EXCEPT LINE 6, WHICH TAKES TIME $\Theta(\lg n)$.

HeapIncreaseKey(A, i, K)

```
1.) if K ≥ A[i]
2.)     A[i] ← K
3.)     while i > 1 & A[Parent(i)] < A[i]
4.)         A[i] ↔ A[Parent(i)]
5.)         i ← Parent(i)
```

HeapIncreaseKey(A, i, k) INCREASES THE PRIORITY OF A[i] TO k, THEN TRAVERSES AN UPWARD PATH FROM PARENT TO PARENT, REPAIRING THE HEAP PROPERTY ALONG THE WAY. SINCE THIS PATH CAN HAVE LENGTH AT MOST $\lg n$, WHERE n = heap-size[A], THE RUN TIME OF HeapIncreaseKey(A, i, k) IS, IN WORST CASE, $\Theta(\lg n)$.

HeapInsert(A, k)

1.) heap-size[A] ← heap-size[A] + 1
2.) A[heap-size[A]] ← $-\infty$
3.) HeapIncreaseKey(A, heap-size[A], k)

RUN TIME $\Theta(\lg n)$

EXCERCISE

IMPLEMENT A PRIORITY QUEUE IN BOTH C AND Java. ADD APPROPRIATE CONSTRUCTORS, DESTRUCTORS AS APPROPRIATE. ALSO ADD isFull AND isEmpty AS WELL.

Rmk:

A HEAP IS A DATA STRUCTURE WHILE A PRIORITY QUEUE IS AN ADT. CAN A PRIORITY QUEUE BE IMPLEMENTED IN OTHER WAYS?