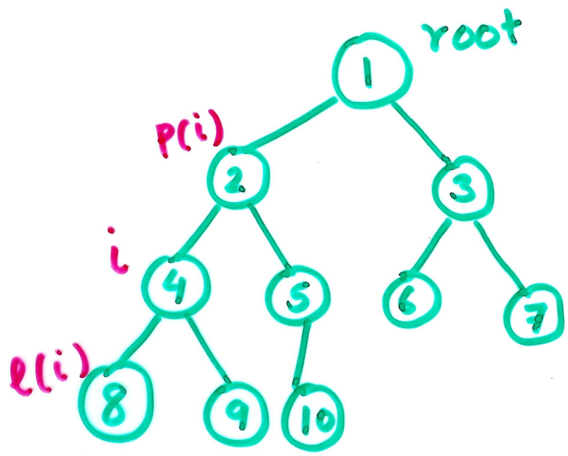
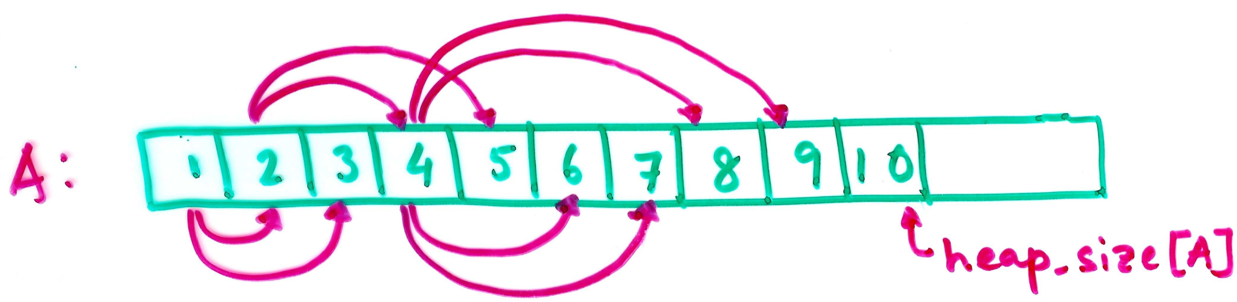
Heapsort

1. Heaps.



Heap property: for every node i, the value in i is less or equal the value in $p(i)$.



The embedding is defined by

$$\begin{aligned} root &:= 1; \\ \ell(i) &:= 2i \\ r(i) &:= 2i+1 \\ p(i) &:= \left\lfloor \frac{i}{2} \right\rfloor \end{aligned}$$

Heap property is: $A[i] \le A[p(i)]$ for all i

Largest element is in root $A[1]$.

The <u>height</u> of a node is the number of edges on the longest downward path starting at the node.

The <u>height</u> of a heap is the height of its root.

<u>Claim</u>. $\log(n+1)-1 \le h \le \log n$

<u>proof</u>. $\sum_{i=0}^{h-1} 2^i + 1 \le n \le \sum_{i=0}^{h} 2^i$

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

$$\log(n+1)-1 \le h \le \log n$$

Maintaining Heap Property:

Downheap extends the heap-property by one more node.

```
Procedure Downheap(i);
  max := i; if l(i) ≤ heap-size[A] and A[max] < A[l(i)]
                 then max := l(i)
            endif
            if r(i) ≤ heap-size[A] and A[max] < A[r(i)]
                 then max := r(i)
            endif
    if max ≠ i then A[i] ⟷ A[max];
         Downheap(max)
    endif
```

cost is $O(h)$

An iterative version is:

```
Procedure Downheap(i);
 repeat max:=i;
  if l(i) ≤ heap-size(A) and A[max] < A[l(i)]
        then max:= l(i)
  endif
  if r(i) ≤ heap-size(A) and A[max] < A[r(i)]
        then max:= r(i)
  endif
  A[i] <--> A[max]; i <--> max
 until i:=max
```

Building a heap

The idea is to construct it from bottom up.

```
Procedure Buildheap(n);
    for i:= n downto 1 do Downheap(i) endfor
```

(Assumption: A is global; n = heap-size[A])

It is easy to show that this takes $O(n \log n)$ time. But, a tighter analysis is possible.

The amount of time to build the heap is at most

$$\sum_{i=0}^{h} 2^i \cdot O(h-i) = O\left(h\sum_{i=0}^{h} 2^i - \sum_{i=0}^{h} 2^i \cdot i\right)$$
$$= O\left(h \cdot 2^{h+1} - h - (h+1)2^{h+1} + 2^{h+2} - 2\right)$$
$$= O(n)$$

[used the fact: $\sum_{i=0}^{h} i\,2^i = (h+1)2^{h+1} - 2^{h+2} + 2$.]

HeapSort algorithm

The input is an unsorted array $A[1 \ldots n]$, $n = \text{length}[A]$. After the construction of heap the maximum is repeatedly moved while shrinking it.

```
Procedure HeapSort(n);
 Buildheap(n);   heap-size[A] := n;
 for i := n downto 2  do  A[1] <--> A[i];
                          heap-size[A] := i-1
                          Downheap(1)
 endfor
```

Complexity is $O(n\log n)$

Heap as Priority Queue

A priority queue stores a multiset S of keys and supports operations

Insert(x): $S := S \cup \{x\}$

Delete(i): remove element at location i

Max: return the largest key

ExtractMax: return the largest key and remove it.

```
Procedure Insert(x);
  heap-size[A] := heap-size(A)+1; i:= heap-size(A);
   A[i] := x; upheap(i)

Procedure Upheap(i);
  while i>1 and A[i] > A[p(i)] do
        A[i] <--> A[p(i)] ; i:=p(i)
  endwhile

Procedure Delete(i);
  A[i] := A[heap-size(A)]; heap-size(A):=heap-size(A)-1;
  if A[i] < A[p(i)] then Downheap(i)
        else upheap(i)
  endif

function ExtractMax;
  ExtractMax := A[1]; Delete(1)
```

All take $O(\log n)$ time.