Heaps (Keot)

Example of transformation (of an array) ...

~ an efficient

- implementation of a Priority Queue
- sorting algorithm

Operations of a Priority Queue

procedure deleteMax (modifies P; 
   PrQueue) returns T;

procedure insert(x: T; modifies P; 
   PrQueue);

procedure shiftUp(v: node)
   if v.parent ≠ null and 
      v.key > v.parent.key then 
         swap(v.key, v.parent.key);
         shiftUp(v.parent);

procedure shiftDown(v: node)
   if v has children then 
      Let l be a child with largest l.key;
      if v.key < l.key then 
         swap(v.key, l.key);
         shiftDown(l);

Heap-implementation of Priority Queue

Heap: tree, where max key of any subtree is at its root

- maximum found directly at the root

Insertion of a key and deleteMax require restoration of the heap:

shifting an incorrectly placed key either up or down in the structure
PrQueue operations using Heap operations:

**procedure** deleteMax(modifies $P$: PrQueue)
**returns** key:

max := $P$.key;
Let $l$ be a leaf;
$P$.key := $l$.key;
Remove leaf $l$;
shiftDown($P$);
**return** max;

**procedure** insert($a$: key; modifies $P$: PrQueue)
Create a new leaf $v$ into $P$;
$v$.key := $a$;
shiftUp($v$);

---

2-heap is a *binary* tree whose keys have

- a heap order

which is

- *full*: filled level-wise,
  from left to right

(Generalization: $d$-heap, $d \geq 2$)

---

Properties:

1. Implementation as array $A[1..n]$:
   - root $\rightarrow A[1]$

2. Maximum is at the root $A[1]$

3. The heap is shallow:

---

**Lemma** The height of a 2-heap with $n$ keys is $h = \lceil \log n \rceil$

**Proof.** At depth $h$ there are $1 \ldots 2^h$ keys;
Levels at smaller depths are full:
Heapsort (Kekolajittelu)
(Williams & Floyd, 1964)

Idea: Sort $A[1..n]$ using a Priority Queue:

$$P := \text{new PriorQueue}();$$  
$$\text{for } i := 1 \text{ to } n \text{ do insert}(A[i], P);$$  
$$\text{for } i := n \text{ downto } 1 \text{ do}$$  
$$\hspace{1cm} A[i] := \text{deleteMax}(P);$$

$O(n)$ heap operations  
implementation using 2-heap  
$\rightarrow T(n) = O(n \log n)$

Let's concretize, and optimize:

\begin{verbatim}
  procedure heapSort(modifies A[1..n]): 
    // I: transform A[1..n] into a heap: 
    for i := [n/2] downto 1 do shiftDown(i, n); 
    // II: pick the keys in decreasing order: 
    for i := n downto 2 do 
      // A[i] ← deleteMax(A[1..i]): 
      swap(A, 1, i); 
      shiftDown(1, i - 1);

  procedure shiftDown(i, j): 
    if 2i ≤ j then // i has a left child .. 
      k := 2i; 
    if 2i + 1 ≤ j then // and a right child 
    if A[i] < A[k] then 
      swap(A, i, k); 
      shiftDown(k, j);
\end{verbatim}

Invariant I: $A[1..n]$ is in heap order
procedure heapSort(modifies A[1..n]):
   // I: transform A[1..n] into a heap:
   for i := ⌊n/2⌋ downto 1 do shiftDown(i, n);
   // II: pick the keys in decreasing order:
   for i := n downto 2 do
      // A[i] ← deleteMax(A[1..i]):
      swap(A, 1, i);
      shiftDown(1, i - 1);

Invariant II:
(i) A[(i + 1)..n] is sorted,
(ii) A[1] = maxA[1..i], and
(iii) A[1], ..., A[i] ≤ A[(i + 1)], ..., A[n]

Construction of the heap (I) takes linear time only:

Height of the heap $h = \lfloor \log n \rfloor$

There are $2^i$ keys at depth $i = 0, \ldots, h - 1$

From depth $i$, we can shift down at most $h - i$ levels

Complexity is determined by the total length of the shiftDown paths, which is at most:

$$\sum_{i=0}^{h-1} 2^i(h - i) = h \sum_{i=0}^{h-1} 2^i - \sum_{i=0}^{h-1} i2^i$$

$$= h(2^h - 1) - (h - 2)2^h + 2$$

$$= 2 \cdot 2^{\lfloor \log n \rfloor} - \lfloor \log n \rfloor - 2$$

$$< 2 \cdot 2^{\log n} = 2n$$