Heaps (Keot)

Clever maintaining of partial order in an array $\rightsquigarrow$ an efficient

- implementation of a Priority Queue
- sorting algorithm

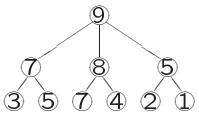
Priority Queue Operations

procedure deleteMax (**modifies** P: PrQueue) **returns** T;

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

Heap-implementation of Priority Queues

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

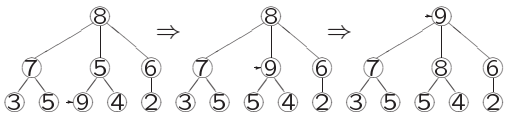


- maximum found directly at the root

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

shifting an incorrectly placed key either **up** or **down** in the structure

procedure shiftUp(v: node)
// if v.key too big wrt its level
while (v has a parent whose key is smaller than v.key) **do**
swap(v.key, v.parent.key);
$v \leftarrow v$.parent;



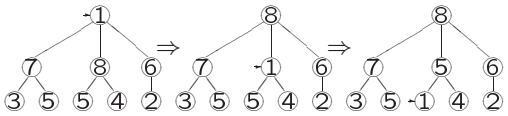
Complexity?

Correctness of shiftUp

Assume that $x = v$.key is too big wrt its level, but the heap is otherwise OK:

Each iteration eliminates one violation of the heap property

procedure shiftDown(v: node)
// if v too small wrt its level
if v has children **then**
    Let l be a child with maximal key;
    if v.key $< l$.key **then**
        swap(v.key,l.key);
        shiftDown(l);



Complexity?

Correctness of shiftDown

Assume that $x = v$.key is too small wrt its level, but the heap is otherwise OK:

Each iteration eliminates one violation of the heap property

PrQueue operations via Heap operations

procedure deleteMax(**modifies** P: PrQueue)
    returns key:
        max $\leftarrow P$.key;
        Let l be a leaf;
        P.key $\leftarrow l$.key;
        Remove leaf l;
        shiftDown(P);
        return max;

procedure insert(a: key; **modifies** P: PrQueue)
        Create a new leaf v into P;
        v.key $\leftarrow a$;
        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 an array $A[1..n]$:
   - root $\rightarrow A[1]$
   - children of $A[i] \rightarrow A[2i], A[2i+1]$
     ($\rightsquigarrow$ parent of $A[j]$ is at $A[\lfloor j/2 \rfloor]$)
2. Maksimum is at $A[1]$
3. The heap is shallow. (See next)

Lemma The height of a 2-heap with n keys is $h = \lfloor \log n \rfloor$

Proof. At depth h there are $1 \ldots 2^h$ keys; Levels at depths 0,1,... $h-1$ are full:

Heapsort (Kekolajittelu)

(Williams & Floyd, 1964)

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

```
P ← new PriorQueue();
for i ← 1 to n do insert(A[i], P);
for i ← n downto 1 do
      A[i] ←deleteMax(P);
```

$O(n)$ heap operations

using a 2-heap each in time $O(\log n)$
$\rightarrow T(n) = O(n \log n)$

Let's concretize, and optimize:

Implementation of shiftDown($A[i]$) in a 2-heap $A[1..j]$:

```
procedure shiftDown(i, j):
// if A[i] too small wrt its level
if 2i ≤ j then //i has a left child ..
    k ← 2i;
    if 2i + 1 ≤ j then //and a right child
        if A[2i + 1] > A[2i] then k ← 2i + 1;
    if A[i] < A[k] then
        Swap A[i] and A[k];
        shiftDown(k, j);
```

```
procedure heapSort(modifies A[1..n]):
// I: Heapify A[1..n]:
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] and A[i];
    shiftDown(1,i − 1);
```



```
procedure heapSort(modifies A[1..n]):
// I: Heapify A[1..n]:
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] and A[i];
    shiftDown(1,i − 1);
```

Invariant I: $A[i..n]$ is in heap order


```
procedure heapSort(modifies A[1..n]):
// I: Heapify A[1..n]:
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] and A[i];
    shiftDown(1,i − 1);
```

Invariant II:
(i) $A[1..i]$ is a heap,
(ii) $A[(i+1)..n]$ is sorted, and
(iii) $A[1], \ldots, A[i] \leq A[(i+1)], \ldots, A[n]$

Heapify $A[1..n]$ takes *linear time only*:

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

Keys at depth i can shift down at most $h - i$ levels ($h = \lfloor \log n \rfloor$)

Total complexity of the *shiftDown* operations is determined by the total length of the paths along which they shift:

$$\begin{aligned}\sum_{i=0}^{h-1} 2^i(h-i) &= h\sum_{i=0}^{h-1} 2^i - \sum_{i=0}^{h-1} i2^i \\ &= h(2^h-1) - ((h-2)2^h+2) \\ &= 2\cdot 2^{\lfloor \log n \rfloor} - \lfloor \log n \rfloor - 2 \\ &< 2\cdot 2^{\log n} = 2n\end{aligned}$$

Comparison of insertion sort, heapsort and quicksort on small arrays

Times milliseconds, from initializing and sorting 100,000 arrays of n integers