Heaps and Priority Queues

Priority Queue ADT

- A queue where the first element dequeued is the one with the highest priority
- Uses:
  - Simulate real-world systems queues organized by priority
    - Patients in a hospital
    - Files requested from a server
  - A* search (details later)
    - Explore the outcomes of possible moves in a game, with priority given to more promising moves
  - …

Priority Queue ADT

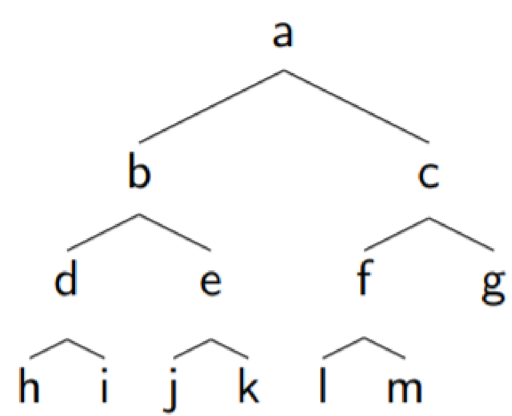
- Insert(S, x): add a new element with priority x to priority queue S
- min(S): return the element with the smallest value from the priority queue
- extract_min(S): remove and return the element with the smallest value from the priority queue

Implementation

- array, linked list
  - O(1) for insert, O(n) for min and extract_min
- Sorted array/linked list
  - O(n) for insert
  - O(1) for min/extract_min

Implementation: Heaps

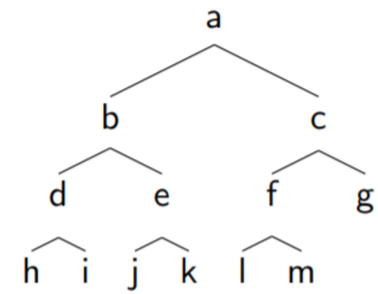
- A tree, with every node having two children, except the "leaves" (nodes at the bottom with no children), and every leaf is as far left as possible on the last level
  - A "complete" tree



  - Heap order property

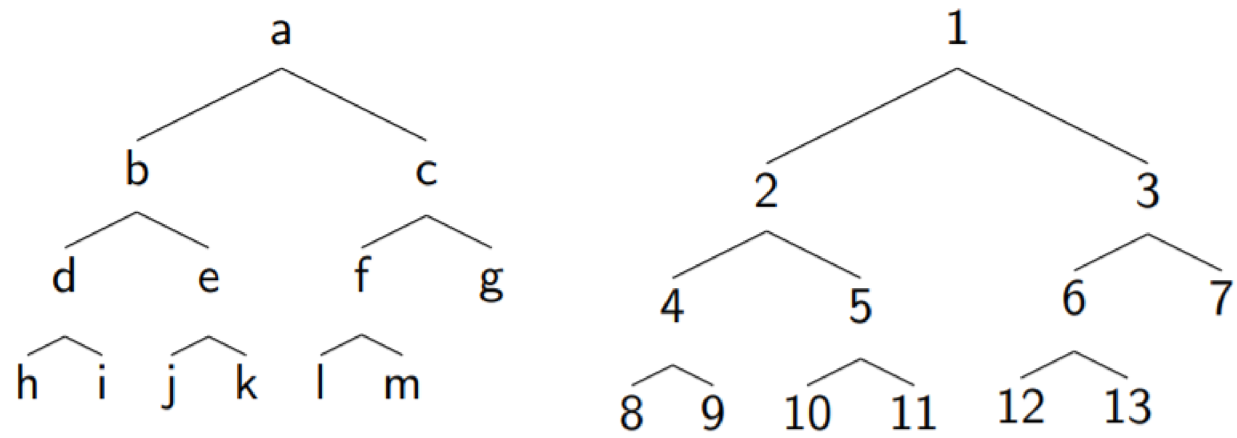

Hyphaene Compressa - Doum Palm
© Shlomit Pinter

Heaps



- Could store similarly to linked lists, with each node having to “children” (instead of one next node)
- Because the tree is complete, we can (and will) store the heap as an array

$$[\_, a, b, c, d, e, f, g, h, i, j, k, l, m, ??, ??]$$



$$[\_, a, b, c, d, e, f, g, h, i, j, k, l, m, ??, ??]$$

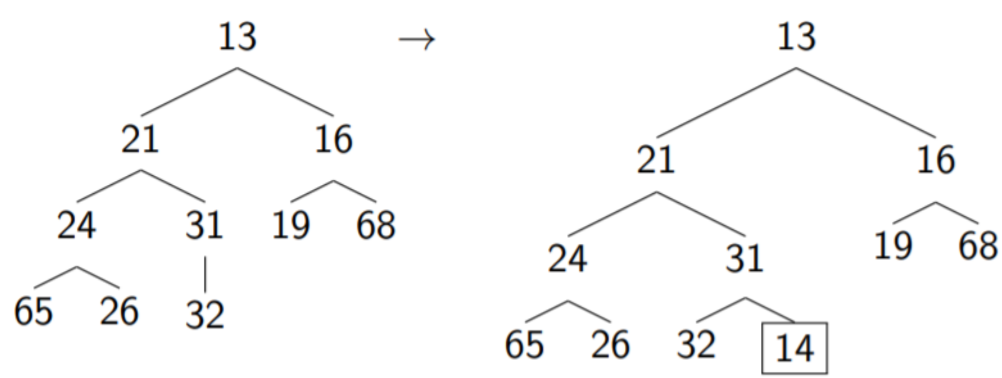
- Given a node at index i, we can get the “parent” and the “children”:
  - parent(i) = $i/2$
  - left(i) = $2 \ast i$
  - right(i) = $2 \ast i + 1$

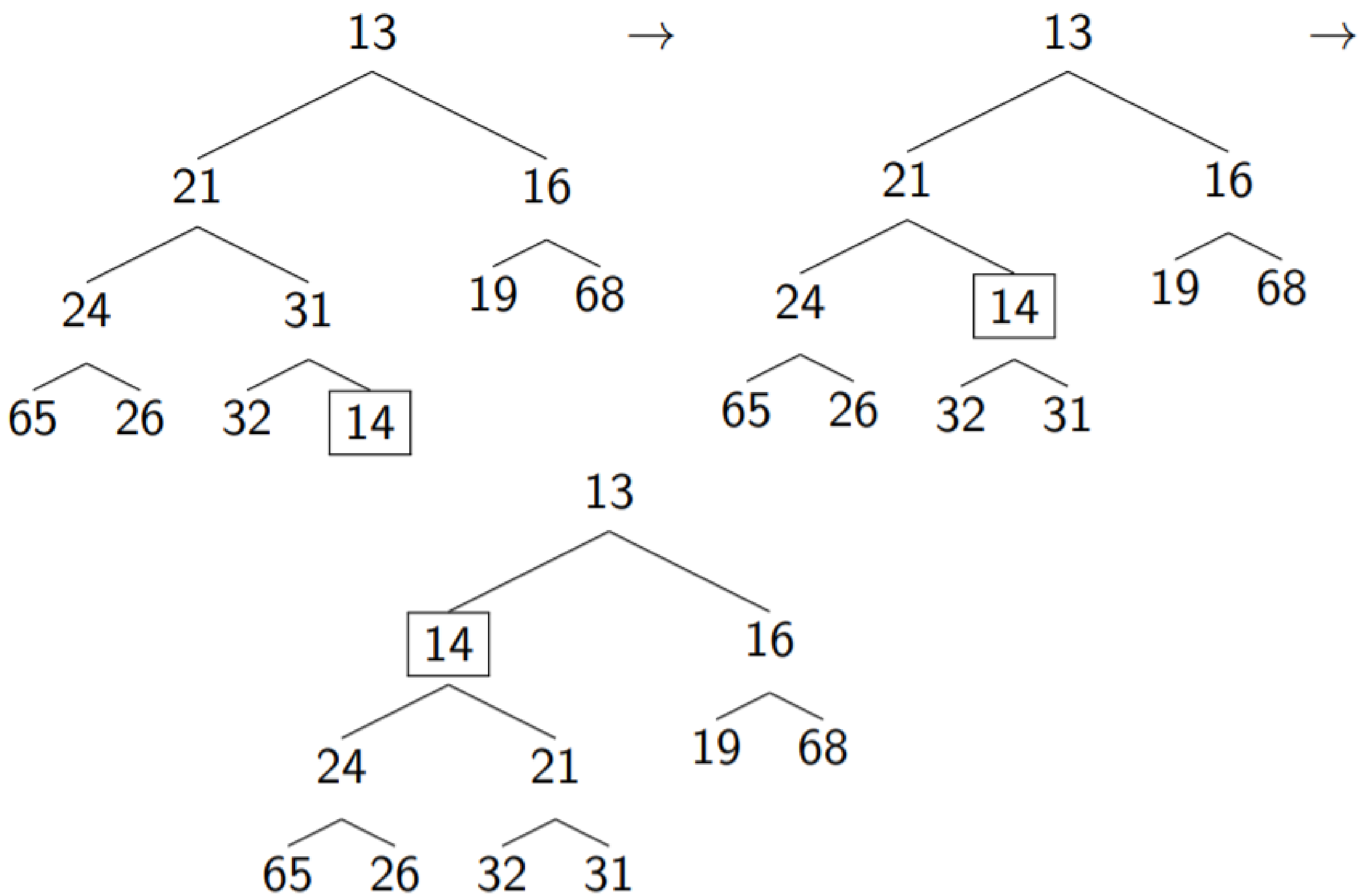
Heap order property

- For each node n other than the “root” (top node) in a binary heap, the value stored in the parent must be less than or equal to the value stored in n
  - The minimum element is at the root

Heap Operations: Insert

- Initially place the new value at the leftmost empty space in the bottom level of the heap
  - Heap remains a complete tree
  - Might break the heap order property
- To fix this, *percolate* the value up the tree until it is in an appropriate spot

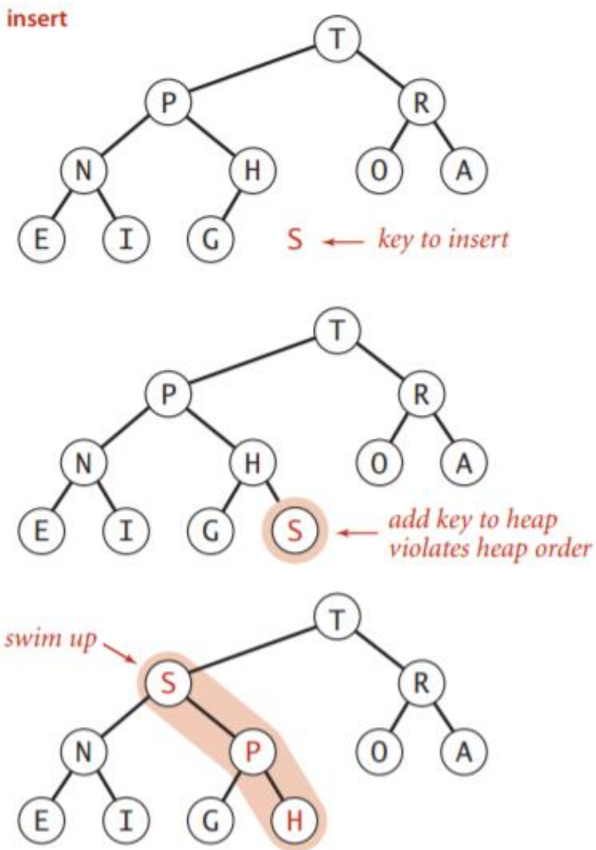



13
21
16
24
31
19
68
65
26
32
14
→
13
21
16
24
14
19
68
65
26
32
31
→
13
14
16
24
21
19
68
65
26
32
31

Insert: algorithm

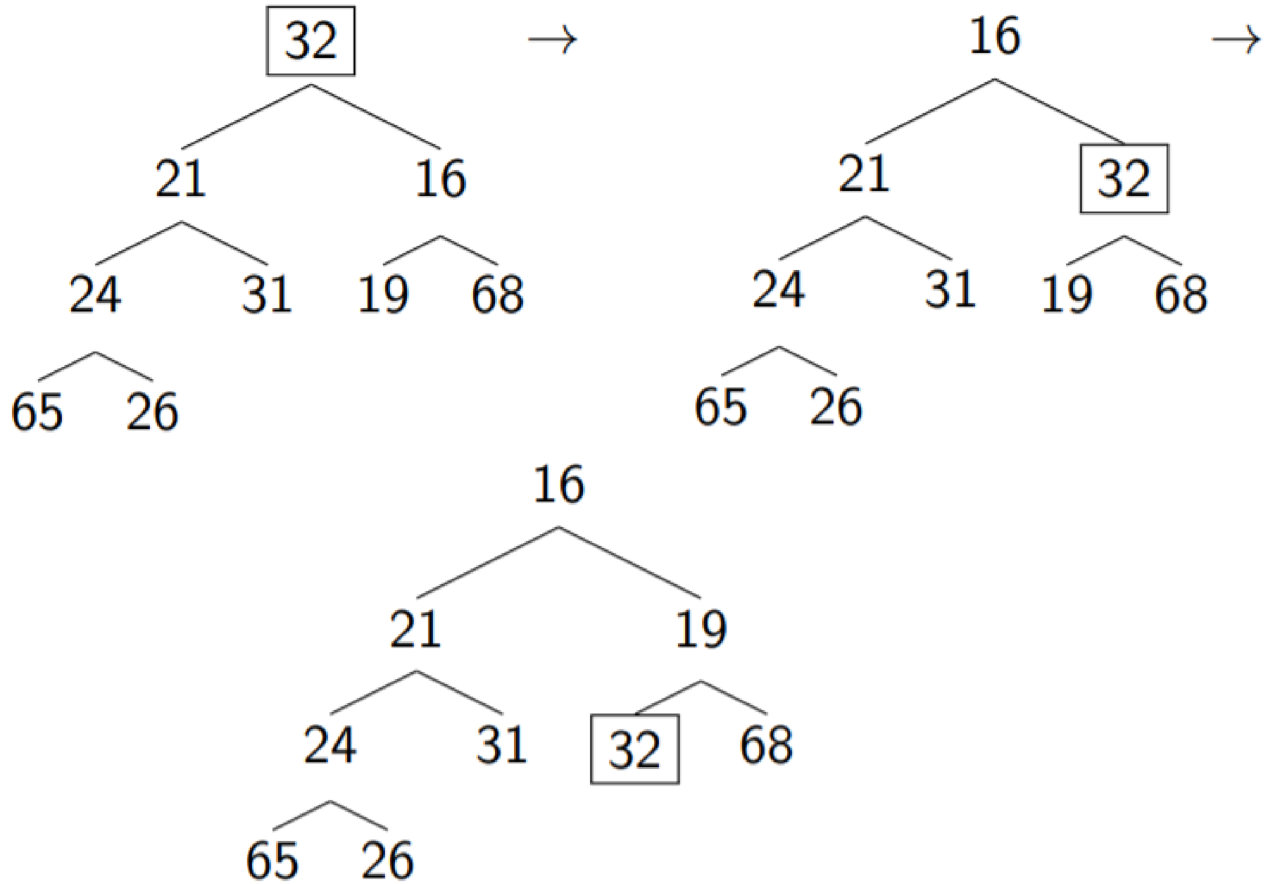
n: size of the heap
pq: array storing the heap

```
Insert(x)
  k = n + 1
  pq[k] = x
  while (k > 1 and pq[k/2] > pq[k])
    swap(pq[k], pq[k/2])
    k = k / 2
```



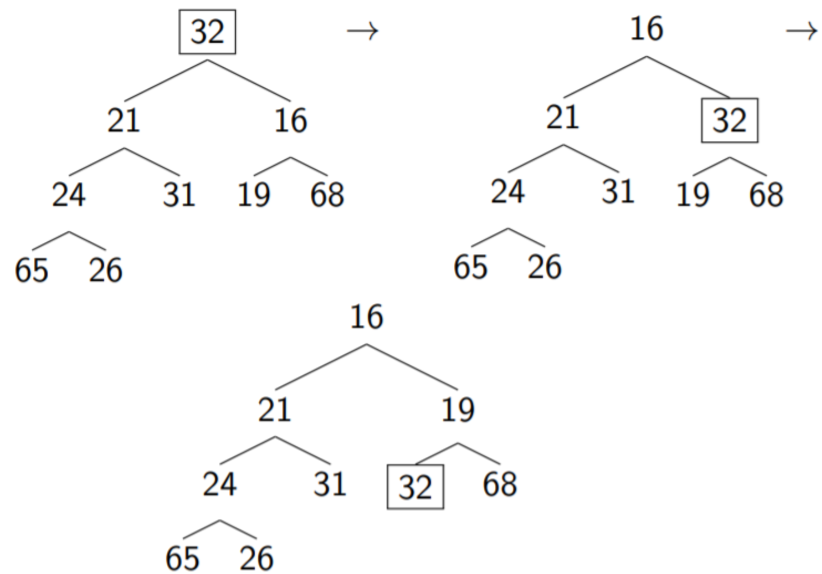
Heap operations: extract_min

- Replace the minimum element (save it first) with the element at the end of the array
- Percolate the element now at index 1 down the array until the heap-order property is satisfied



32
21
16
24
31
19
68
65
26
→
16
21
32
24
31
19
68
65
26
→
16
21
19
24
31
32
68
65
26

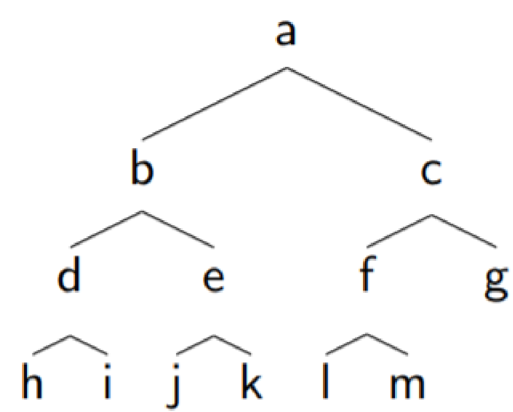
extract_min




```
extract_min()
  min = pq[1]
  swap(pq[1], pq[n])
  n = n-1
  k = n
  while (2*k <= n)
    j = 2*k
    if (j < n and pq[j] > pq[j+1])
      j  = j+1 //want to exchange with the smaller child since the smaller
             //child can be a parent of the larger one
    if (pq[k] <= pq[j])
       break
    swap(pq[k], pq[j])
    k = j
  return min
```

Complexity of insert and extract_min

- Height of a tree: the longest path from node to leaf
- $n \leq 2^0 + 2^1 + 2^2 + \cdots + 2^h = 2^{h+1} - 1$
- $n > 2^0 + 2^1 + \cdots + 2^{h-1} = 2^h - 1$
- $h - 1 \leq \log_2(2^h - 1) \leq \log_2 n \leq \log_2(2^{h+1} - 1) < h + 1$
- Insert and extract_min need at most h swaps
- O(h) = O(log(n))



h = 3
n = 13

Implementation

- array, linked list
  - O(1) for insert, O(n) for min and extract_min
- Sorted array/linked list
  - O(n) for insert
  - O(1) for min/extract_min
- Heap
  - O(log(n)) insert
  - O(log(n)) extract_min
  - O(1) min