Heapsort

Data Structures and Algorithms
Andrei Bulatov

Heap Property

A heap is a nearly complete binary tree, satisfying an extra condition

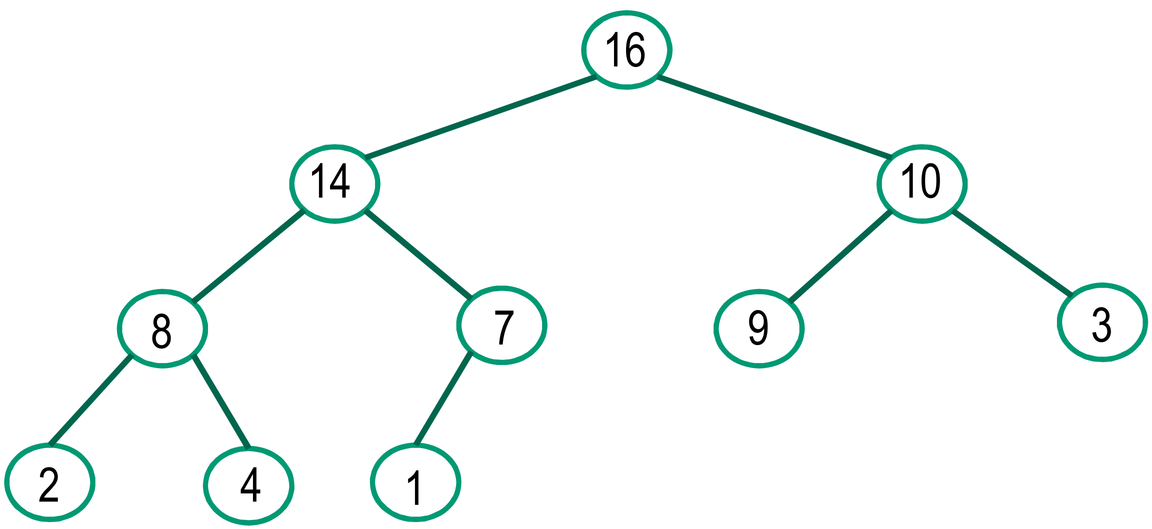
Let Parent(i) denote the parent of the vertex i

Max-Heap Property:

$$\text{Key}(\text{Parent}(i)) \geq \text{Key}(i) \quad \text{for all } i$$

Min-Heap Property:

$$\text{Key}(\text{Parent}(i)) \leq \text{Key}(i) \quad \text{for all } i$$



Heaps

Nearly complete binary tree means that the length of any path from the root to a leaf can vary by at most one

The height of a vertex i is the length of the longest simple downward path from i

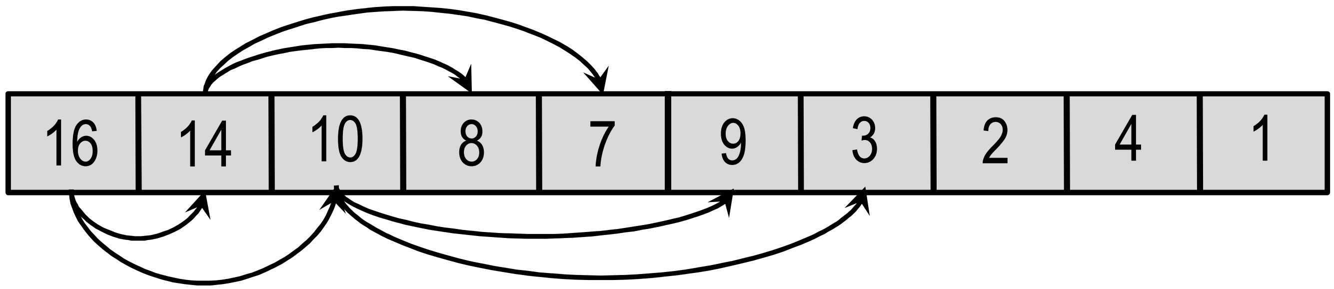
Therefore the height of the root is around log n

Heap Operations

Operation	Goal running time
Creating a max-heap	O(n)
Accessing the maximal element (root)	O(1)
Inserting an element	O(log n)
Deleting an element	O(log n)

Implementing Heaps and Operations

Heap can be implemented by an array



Children:

leftChild(i) = 2i

rightChild(i) = 2i + 1

Parent: parent(i) = $\lfloor i / 2 \rfloor$

Length: length(H) = the number of elements in H

Insertion

```
Insert(H,key)
set n:=length(n),
set H[n+1]:=key
HeapifyUp(H,n+1)
```

H is almost heap with H[i] too big if decreasing H[i] by a certain amount turns H into a heap

---

```
HeapifyUp(H,i)
if i>1 then
   set j:=parent(i)=⌊i/2⌋
   if Key[H[i]]>Key[H[j]] then
      swap array entries H[i] and H[j]
      HeapifyUp(H,j)
   endif
endif
```

HeapifyUp: Soundness

Theorem

The procedure HeapifyUp(H,i) fixes the heap property in O(log i) time, assuming that the array H is almost a heap with the key of H[i] too large.

The running time of Insertion is O(log n)

Proof

Induction on i.

Base Case i = 1 is obvious

Induction Case: Swapping elements takes O(1) time

It remains to observe that after swapping H remains a heap or almost heap

Deletion

Delete(H,i)

```
set n:=length(n),
set H[i]:=H[n]
if Key[H[i]]>Key[H[parent(i)]] then
   HeapifyUp(H,i)
endif
if Key[H[i]]<Key[H[leftChild(i)]] or
  Key[H[i]]<Key[H[rightChild(i)]] then
   HeapifyDown(H,i)
endif
```

H is almost heap with H[i] too small if increasing H[i] by a certain amount turns H into a heap

Deletion (cntd)

```
HeapifyDown(H,i)
set n:=length(H)
if 2i>n then Terminate with H unchanged
else if 2i<n then do
   set left:=2i, right:=2i+1
   let j be the index that minimizes Key[H[left]]
  and Key[H[right]]
else if 2i=n then   set j:=2i
endif
if Key[H[j]]>Key[H[i]] then
    swap array entries H[i] and H[j]
    HeapifyDown(H,j)
endif
```

HeapifyDown: Soundness

Theorem

The procedure HeapifyDown(H,i) fixes the heap property in O(log i) time, assuming that the array H is almost a heap with the key of H[i] too small.

The running time of Deletion is O(log n)

Proof DIY

Building a Heap

```
Build-a-Heap(A)
set n:=length(A)
for i=1 to n do
   set H[i]:=A[i]
   HeapifyUp(H,i)
endfor
```

HeapSort

`HeapSort(A)`

Input: `array A`

Output: `sorted array A`

Method:

```
set H:=Build-a-Heap(A)
set n:=length(H)
for i=n downto 1 do
   set A[i]:=H[1]
   set length(H):=length(H)-1
   Delete(H,1)
endfor
```

Priority Queues

A priority queue is a data structure for maintaining a set S of elements, each with associated value called a key.

Priority Queue operations

Insert(S,x) Insert element x into the set S

Maximum(S) Returns the element of S with the largest key

Extract-Max(S) Removes and returns the element of S with the largest key

Increase-Key(S,x,k) Increases the value of element x's key to the new value k, which is at least as large as x's current key value

Homework

Write pseudocode for procedure

Decrease-Key

that works in the same way as Increase-Key, except that it replaces the key with a smaller value.