CECS 274: Data Structures
Heaps

Oscar Morales Ponce

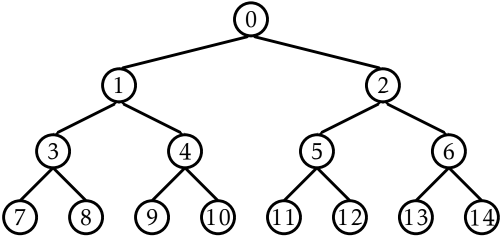
California State University Long Beach

Example

- Suppose you search for song in a list of songs with a prefix, ex ”you”. Which songs should you show first?
- We add to each song a rank (popularity, hotness) so we can show the songs with the highest rank first.
- How can be done efficiently?

BinaryHeap: The Basics

- BinaryHeap
  - State variables:
    - n: Number of elements in the heap.
    - a: Backing array simulating a complete binary search where $a[i]$ stores the element i.
  - Invariant:
    - $a[i] > a[parent(i)]$
    - $n \leq length(a) < 3n$



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Heaps: An Implicit Binary Tree

- We use an array to simulate a complete binary tree

$\text{left}(i)$
  return $2 \cdot i + 1$

$\text{right}(i)$
  return $2 \cdot (i + 1)$

$\text{parent}(i)$
  return $(i - 1) div 2$

BinaryHeap: $add(x)$

- $add(x)$: Insert the element x in the heap

add(x)
  Check Invariants (size)
  Insert x at the position $a[n]$
  Increase n by one
  Check Invariants (heap)

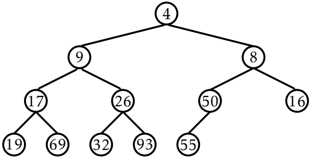
BinaryHeap: $add(x)$

```
add(x)
    if length(a) = n then
        resize()
    a[n] ← x
    n ← n + 1
    bubble_up(n − 1)
    return true
```

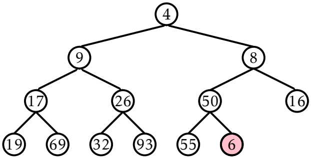
BinaryHeap: $bubble\_up(i)$

bubble_up(i)
  if $i < 0$ **or** $i \geq n$ **then** Exception
  $p \leftarrow$ parent(i)
  while $i > 0$ **and** $a[i] < a[p]$ **do**
    $a[i], a[p] \leftarrow a[p], a[i]$
    $i \leftarrow p$
    $p \leftarrow$ parent(i)

BinaryHeap: $add(x)$ example

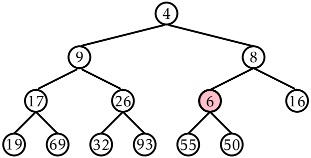


4	9	8	17	26	50	16	19	69	32	93	55			

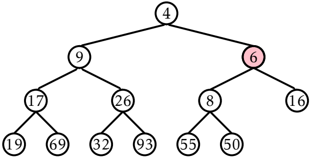


4	9	8	17	26	50	16	19	69	32	93	55	6		

BinaryHeap: $add(x)$ example



4	9	8	17	26	6	16	19	69	32	93	55	50		



4	9	6	17	26	8	16	19	69	32	93	55	50		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

BinaryHeap: $remove()$

- $remove()$: Remove the smallest element in the heap

```
remove()
    Check Preconditions
    Let x be the value of a[i]
    Exchange a[0] with a[n − 1]
    Decrease n by one
    Check Invariants (heap and size)
```

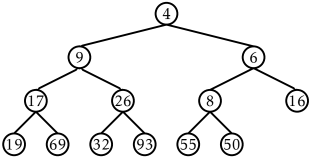
BinaryHeap: $remove()$

```
remove()
   if n = 0 then Exception
   x ← a[0]
   a[0] ← a[n − 1]
   n ← n − 1
   trickle_down(0)
   if 3 · n < length(a) then
      resize()
   return x
```

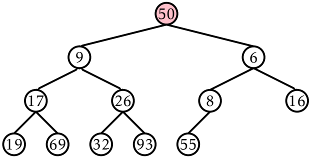
BinaryHeap: $trickle\_down(i)$

```
trickle_down(i)
    while i ≥ 0 do
        j ← −1
        r ← right(i)
        if r < n and a[r] < a[i] then
            ℓ ← left(i)
            if a[ℓ] < a[r] then
                j ← ℓ
            else
                j ← r
        else
            ℓ ← left(i)
            if ℓ < n and a[ℓ] < a[i] then
                j ← ℓ
        if j ≥ 0 then
            a[j], a[i] ←    a[i], a[j]
        i ← j
```

BinaryHeap: $remove(i)$ example

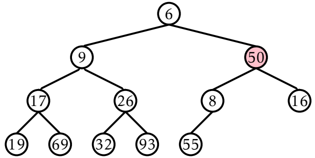


4	9	6	17	26	8	16	19	69	32	93	55	50		

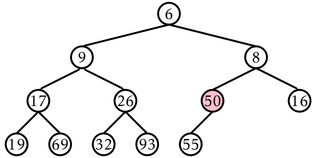


50	9	6	17	26	8	16	19	69	32	93	55			

BinaryHeap: $remove(i)$ example



6	9	50	17	26	8	16	19	69	32	93	55			



6	9	8	17	26	50	16	19	69	32	93	55			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Summary

Theorem

A BinaryHeap implements the Priority Queue interface. Ignoring the cost of calls to $\mathrm{resize}()$*, a BinaryHeap supports the operations* $\mathrm{add}(x)$ *and* $\mathrm{remove}()$ *in* $O(\log n)$ *time per operation. Furthermore, beginning with an empty BinaryHeap, any sequence of* m $\mathrm{add}(x)$ *and* $\mathrm{remove}()$ *operations results in a total of* $O(m)$ *time spent during all calls to* $\mathrm{resize}()$*.*