Heap

- The *heap property*:

  each node is more extreme (greater or less) than each of its children

- + Shape property

Binary heap (…) (by default for us)

Binomial heap

a forest of binomial trees satisfying the heap property

Fibonacci heap

a collection of trees satisfying the heap property

Binary heap

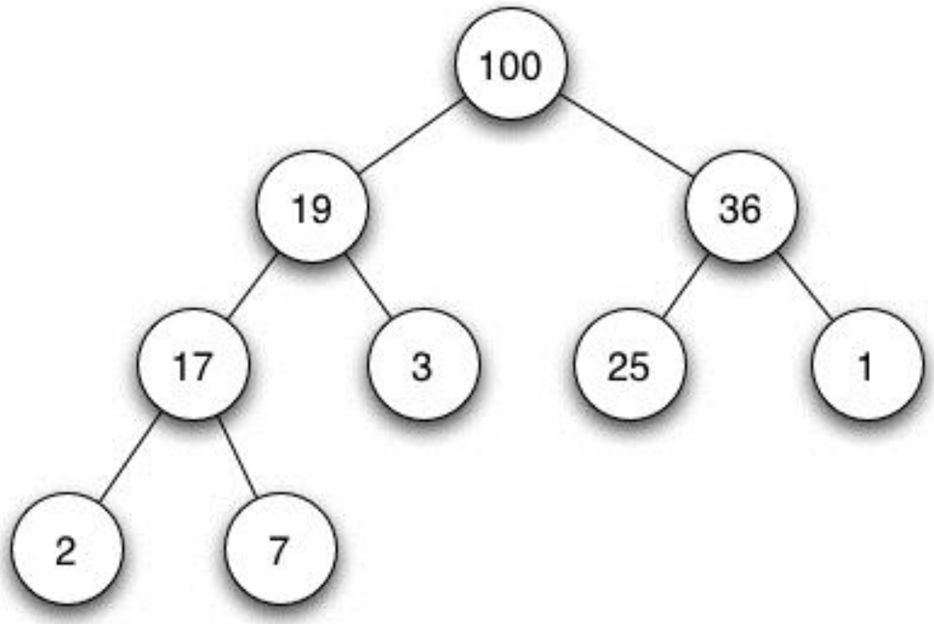
A binary tree with two additional constraints:

- The *shape property*
  (almost) complete
- The *heap property*:
  each node is more extreme (greater or less) than each of its children

or equal

NO ordering of siblings

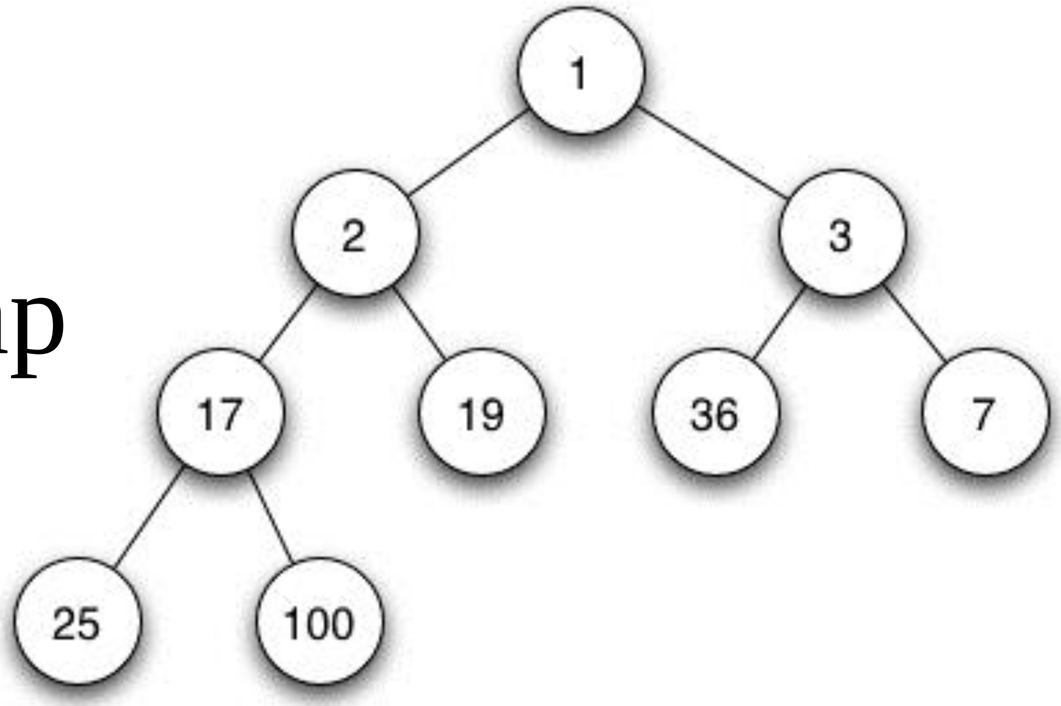
Convention
During these classes, the term heap will refer to max binary heap, when not explicitly specified otherwise.



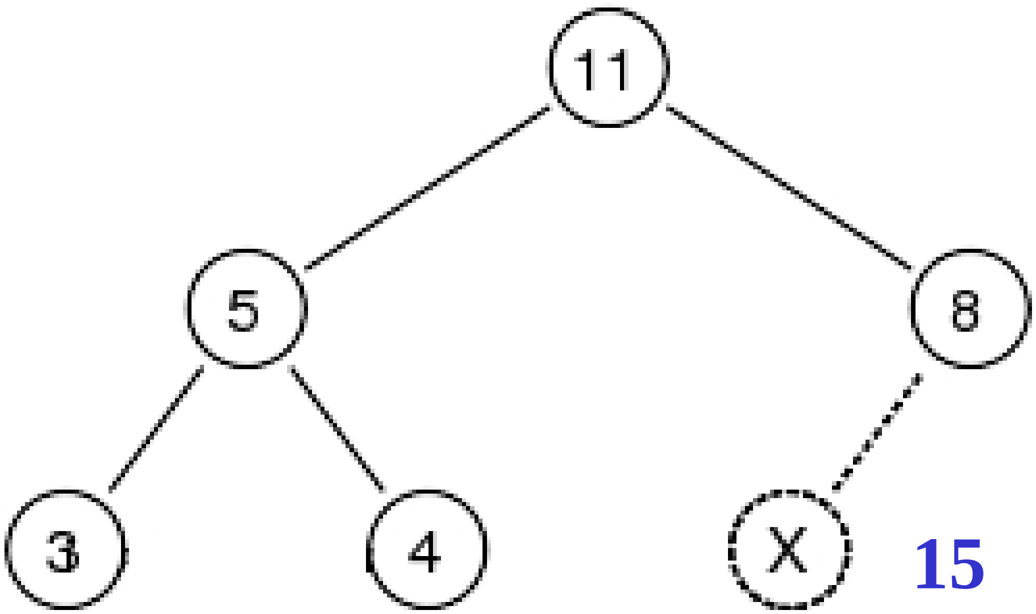
Max binary heap

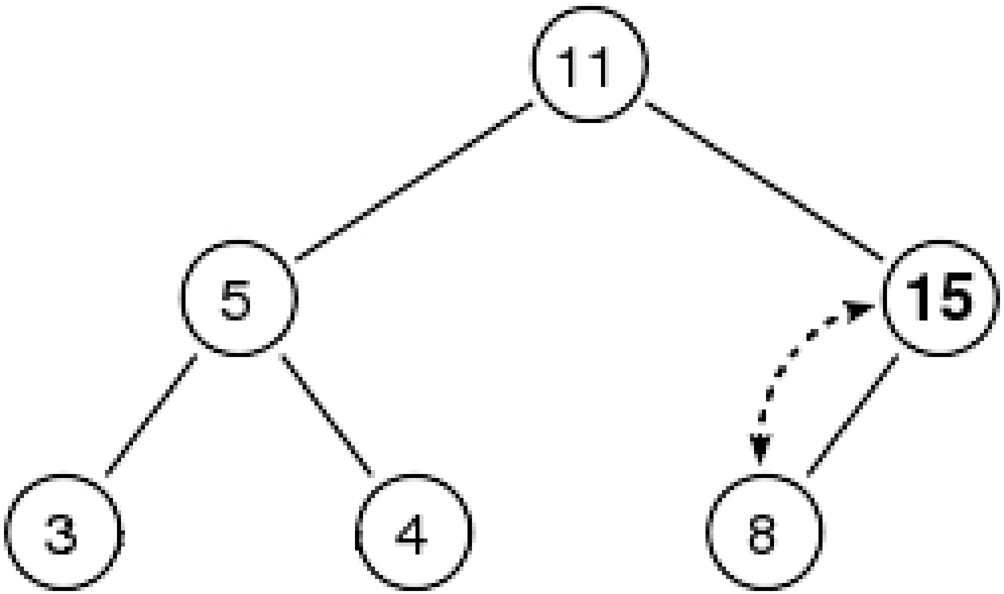
by default, for us

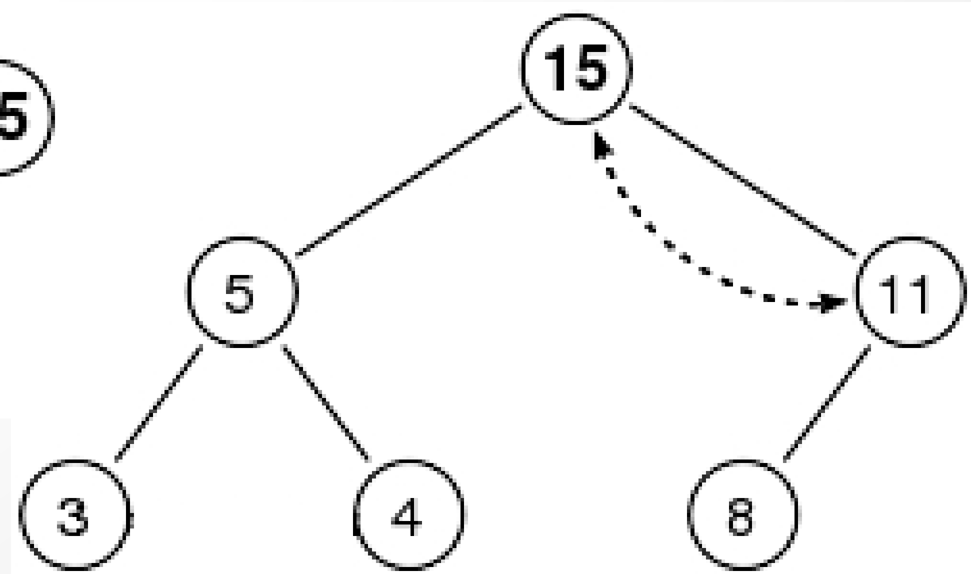
Min binary heap



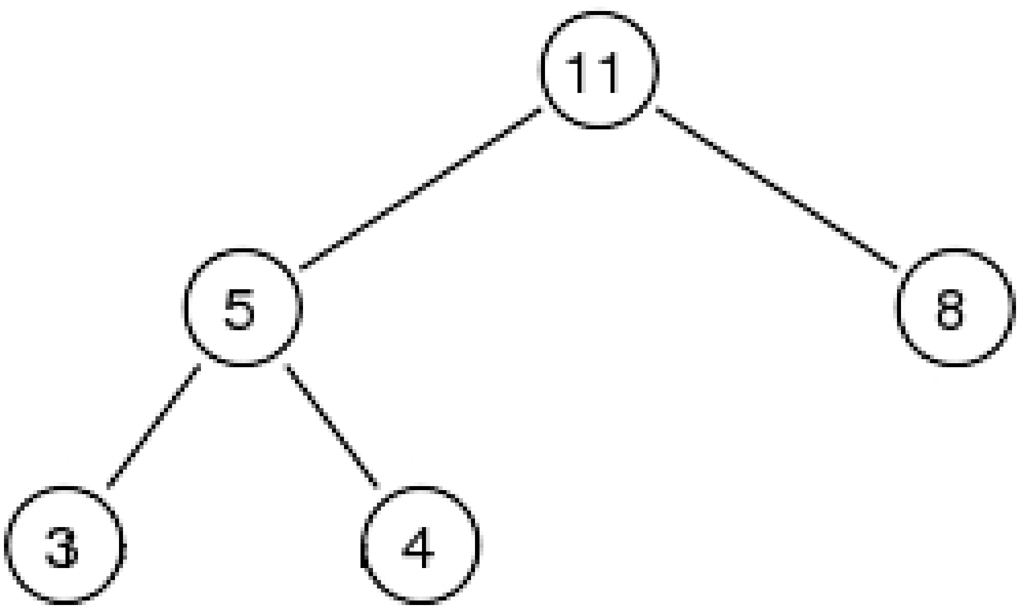
Heap - insertion

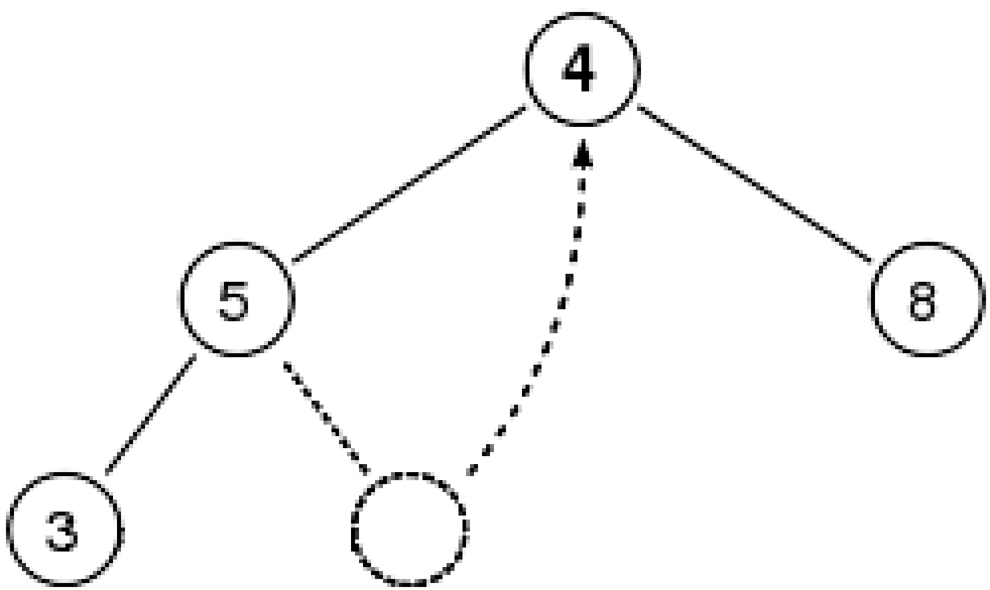


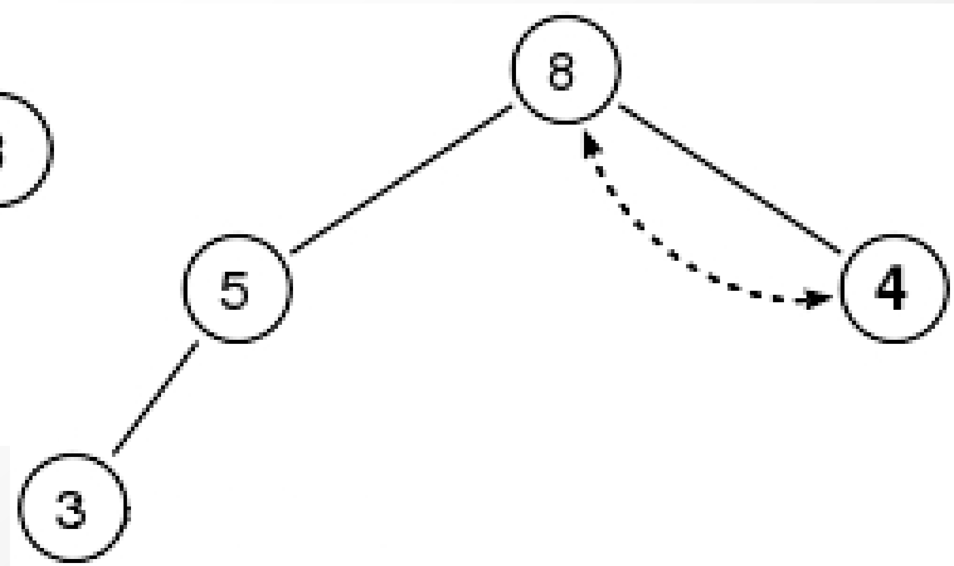




Heap – delete root







Binary heap

Max binary heap

getMax	O(1)
insert	O(log n)
deleteMax	O(log n)

The height of binary heap: O(log n)

Heap – stored in array

tree root item has index 1

n tree elements: $a[1]$ .. $a[n]$

element $a[i]$

children: $a[2i]$ and $a[2i+1]$

parent $a[\text{ floor } (i/2)]$

tree root item has index 0

n tree elements: $a[0]$ .. $a[n-1]$

element $a[i]$

children: $a[2i+1]$ and $a[2i+2]$

parent $a[\text{ floor } ((i-1)/2)]$

```
Heap: record
   n: Integer
   els: array [1..MAX] of TComparable
end
```

Extract maximum (root)

```
Funct. extractMax (H)          //if size(H)>=1
   extractMax :=H.els [1]
   H. els[1]:=H. els[H.n]
   H.n := H.n -1
   downHeap(H,1)
end_extractMax
```

subalg. downHeap(H,poz)

```
el:=H.Element[poz];
p:=poz; ch:=2*poz
while ch<=H.n do
    if ch<H.n then
                    if H. els[ch]<H. els[ch+1] then
                              ch:=ch+1
    endif           endif
    if H. els[ch]< el then break;
    else            H. els[p]:=H. els[ch]
                    p:=ch; ch:=2*ch
    endif
endwhile
H. els[p]:=el
```

end_downHeap

add

```
subalg. add (H,el)
H.n := H.n +1
H. els[H.n] := el
upHeap (H, H.n)
end_add
```

```
subalg. upHeap (H, i)
el := H. els[i]
ch:=i
p:=ch div 2
while (p>=1) and (H. els[p]<el) do
    H. els[ch] := H. els[p]
    ch:=p
    p:=p div 2
endwhile
H. els[ch]:=el
end_upHeap
```

build heap - complexity

- A heap could be built by successive insertions. $O(n \log_2 n)$
- optimal method:
  - starts by randomly putting the elements
  - then: build the *heap property*

// build the heap property

```
Subalg. buildHeapProp(H)
   for i:=[H.n / 2] , 1 , step = -1 do
        downHeap (H,i)
   endfor
endbuildHeapProp
```

$$nrNodes_h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

build heap - complexity

- obvious: complexity $\in O(n*\log_2(n))$
- not obvious, but proved: complexity $\in O(n)$

proof ideas

- nr. nodes of height h $\quad nrNodes_h \leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$
- complexity (*nr. of oper.*)

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right)$$

$$\leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right)$$

$$= O(n)$$

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

HeapSort

- build a heap => O(n)
- repeatedly extract maximum => n*O(log(n))

=> O(n*log(n)) (even in the worse case)

Heap - usage

- used in the sorting algorithm

  heapsort

  one of the best sorting methods

  with no quadratic worst case scenarios

- used to implement priority queues

  Java util: Priority Queue

  based on a priority heap

  head of this queue is the ***least*** *element*

C++ STL .

Standard Template Library: Algorithms

Heap:

push_heap

pop_heap

make_heap …

(uses RandomAccessIterator)

sort_heap …

C++ STL

priority queue

Priority queues are implemented as container adaptors

The underlying container

- accessible through random access iterators
- operations:
  - front()
  - push_back()
  - pop_back()

- random access iterators is required to keep a heap structure internally
- container adaptor call make_heap, push_heap and pop_heap