CSE221

Lecture 9: Heaps and Priority Queues

Fall 2021

Young-ri Choi

Acknowledgment: The content of this file is based on the slides of the textbook as well as the slides provided in former lectures at UNIST.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Outline

- Priority queues
- Heaps

Priority Queue

- Queue with priority order for *pop*
  - –Entries pushed upon their arrivals
  - –But not FIFO (entries ordered by priority)
- Max priority queue
  - –Pop the entry with a highest priority first
- Min priority queue
  - –Pop the entry with a lowest priority first

Priority Queue ADT

- A priority queue stores a collection of entries
- Typically, an entry is a (key, value) pair, where the key indicates the priority

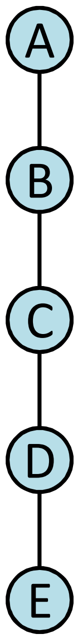
- Main methods
  - insert(e)
    inserts an entry e
  - removeMin() / removeMax()
    removes the entry with smallest / largest key
  - min() / max()
    returns with smallest / largest key, but does not remove the entry
  - size(), empty()

Total Order Relations

- Keys in a priority queue are ALL pairwise comparable and ordered
- When $x \leq y$, we say entry x is related to (or comparable to) entry y
- A binary relation $\leq$ is a total order on all pairs
  - Reflexive property: $\boldsymbol{x} \leq \boldsymbol{x}$
  - Antisymmetric property: $\boldsymbol{x} \leq \boldsymbol{y} \wedge \boldsymbol{y} \leq \boldsymbol{x} \Rightarrow \boldsymbol{x} = \boldsymbol{y}$
  - Transitive property: $\boldsymbol{x} \leq \boldsymbol{y} \wedge \boldsymbol{y} \leq \boldsymbol{z} \Rightarrow \boldsymbol{x} \leq \boldsymbol{z}$
- Connex property: $\boldsymbol{x} \leq \boldsymbol{y}$ or $\boldsymbol{y} \leq \boldsymbol{x}$

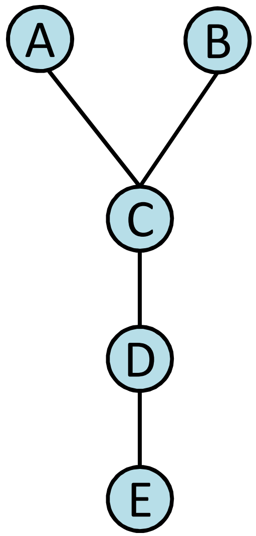
Total Order Example

Total order



Partial order

A and B are not comparable



Priority Queue Sorting

- Use a priority queue to sort a set of comparable entries
  1. Insert the entries one by one with insert operations
  2. Remove the entries in sorted order with removeMin operations
- The running time of this sorting method depends on the priority queue implementation

Algorithm ***PQ-Sort*(*S, C*)**

Input sequence ***S***, comparator ***C*** for the entries of ***S***

Output sequence ***S*** sorted in increasing order according to ***C***

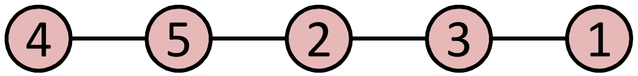
P ← priority queue with comparator ***C***

while !*S.empty* ()
  e ← *S.front*(); *S.eraseFront*()
  P.insert (*e*)

while !*P.empty*()
  e ← *P.min(); P.removeMin*()
  S.insertBack(*e*)

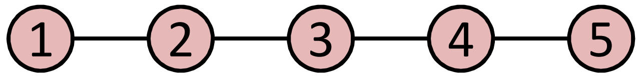
Two Ways of Implementation

- Implementation with an unsorted list



- Performance:
  - insert takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - removeMin and min take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

- Implementation with a sorted list



- Performance:
  - insert takes $O(n)$ time since we have to find the place where to insert the item
  - removeMin and min take $O(1)$ time, since the smallest key is at the beginning

Selection-Sort

- PQ-sorting where the priority queue is implemented with an unsorted sequence
- Running time of Selection-sort:
  1. Inserting the entries with $\boldsymbol{n}$ insert operations takes $\boldsymbol{O}(\boldsymbol{n})$ time
  2. Removing the elements in sorted order with $\boldsymbol{n}$ removeMin operations takes time proportional to

$$\boldsymbol{n} + \boldsymbol{n}\text{-}1 \ldots + 2 + 1$$

- Selection-sort runs in $\boldsymbol{O}(\boldsymbol{n}^2)$ time

Selection-Sort Example

	Sequence S	Priority Queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(7,4)
..	.. ..	
(g)	()	(7,4,8,2,5,3,9)
Phase 2		
(a)	(2)	(7,4,8,5,3,9)
(b)	(2,3)	(7,4,8,5,9)
(c)	(2,3,4)	(7,8,5,9)
(d)	(2,3,4,5)	(7,8,9)
(e)	(2,3,4,5,7)	(8,9)
(f)	(2,3,4,5,7,8)	(9)
(g)	(2,3,4,5,7,8,9)	()

Insertion-Sort

- PQ-sorting where the priority queue is implemented with a sorted sequence
- Running time of Insertion-sort:
  1. Inserting the entries with n insert operations takes time proportional to

     $$1 + 2 \ldots + n\text{-}1 + n$$

  2. Removing the entries in sorted order with n removeMin operations takes $O(n)$ time
- Insertion-sort runs in $O(n^2)$ time

Insertion-Sort Example

	Sequence S	Priority queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(4,7)
(c)	(2,5,3,9)	(4,7,8)
(d)	(5,3,9)	(2,4,7,8)
(e)	(3,9)	(2,4,5,7,8)
(f)	(9)	(2,3,4,5,7,8)
(g)	()	(2,3,4,5,7,8,9)
Phase 2		
(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
..	..	..
(g)	(2,3,4,5,7,8,9)	()

Outline

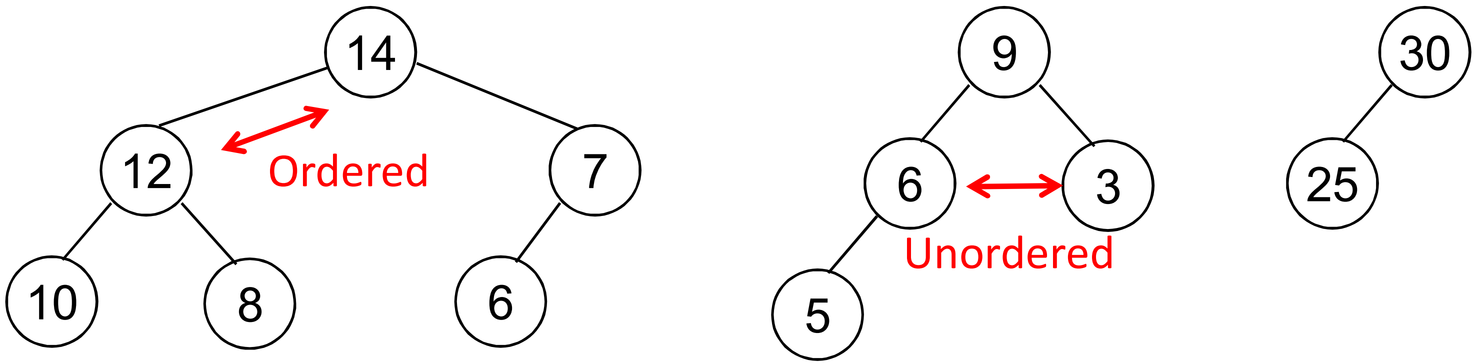
- Priority queues
- Heaps

Heap

- A heap is a tree-based data structure that satisfies the heap property
  - If A is a parent node of B, then key(A) is ordered with respect to key(B)
- Max (min) heap
  - A complete binary tree with the heap property
  - Key in each node is not smaller (not larger) than the keys in its children

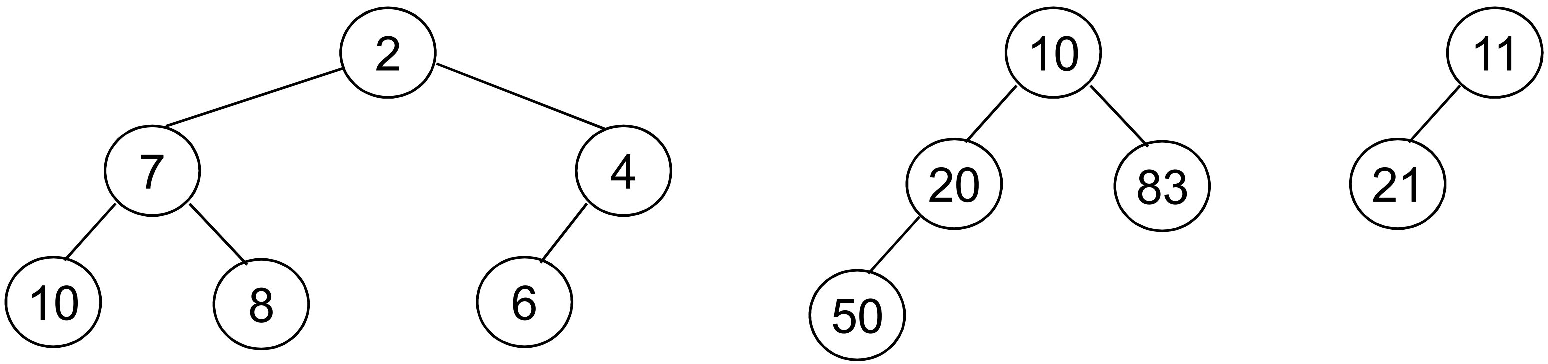
Heap: Complete Binary Tree

Max heap



Complete binary trees!

Min heap

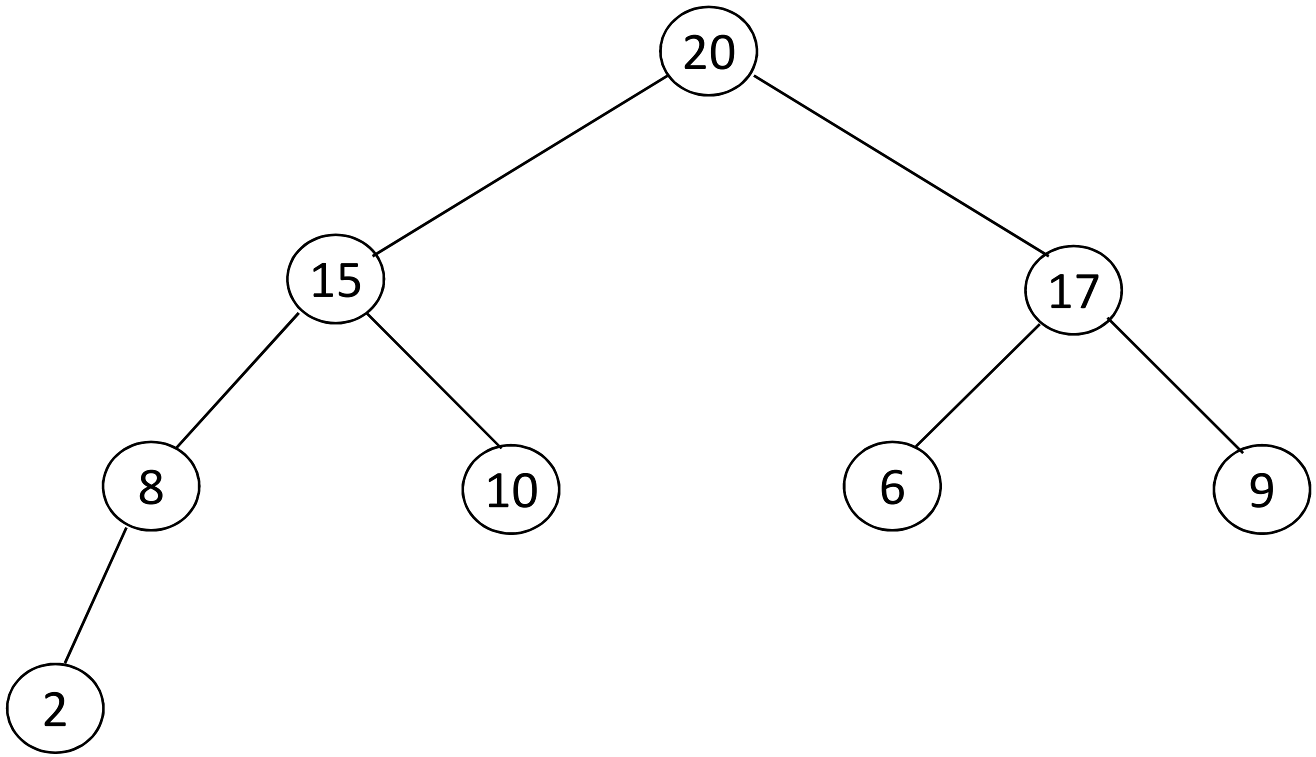


Max Heap

- Push operation
  - Add new element at the end of the tree
  - Bubble up
  - We keep maintaining the max heap property
    - key in each node is not smaller than the keys in its children

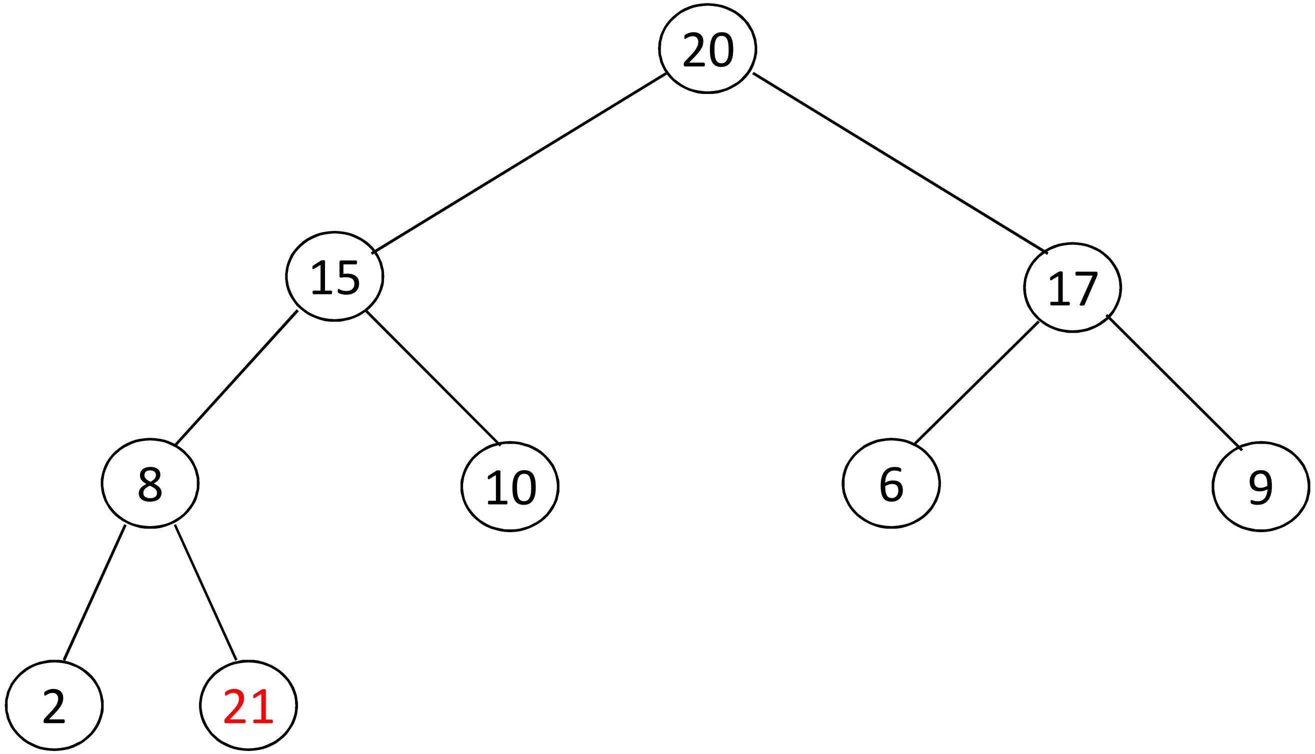
Push Example

- Push 21



Push Example

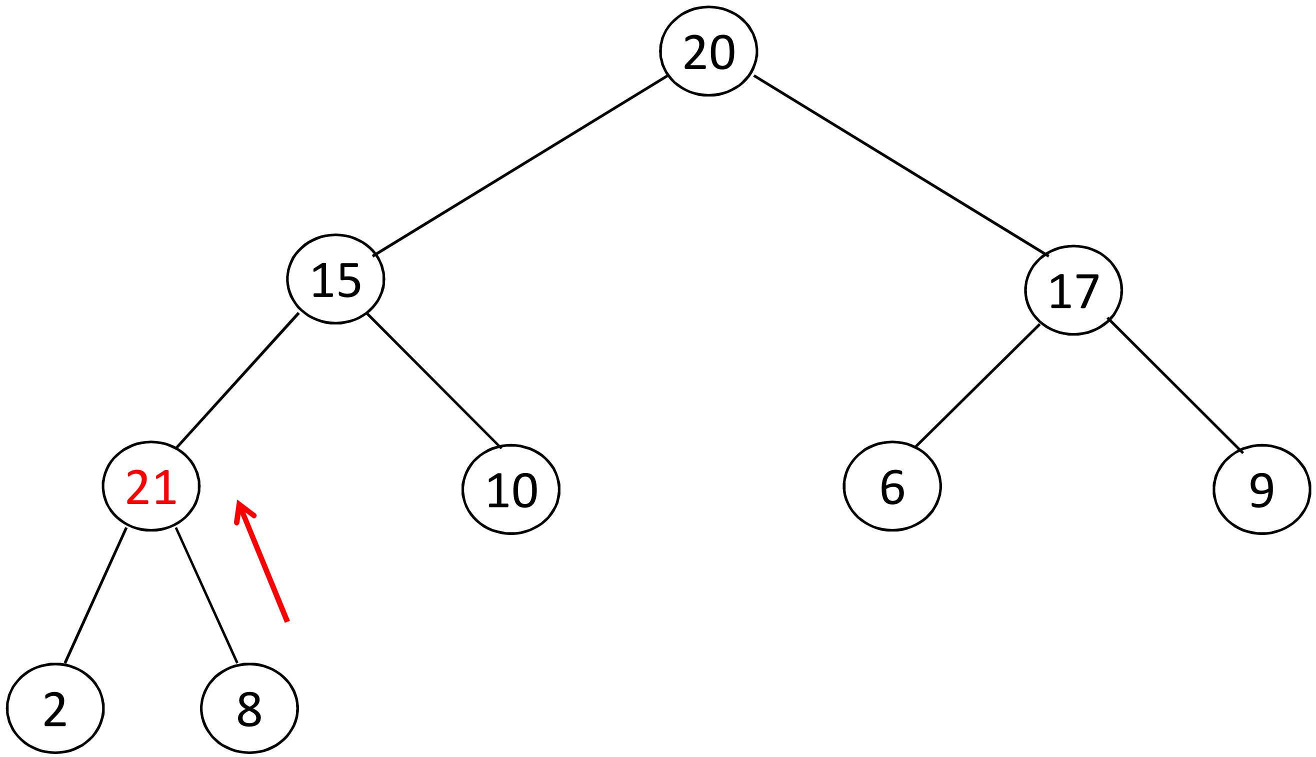
- Push 21



Create node at the end of the heap (complete binary tree)

Push Example

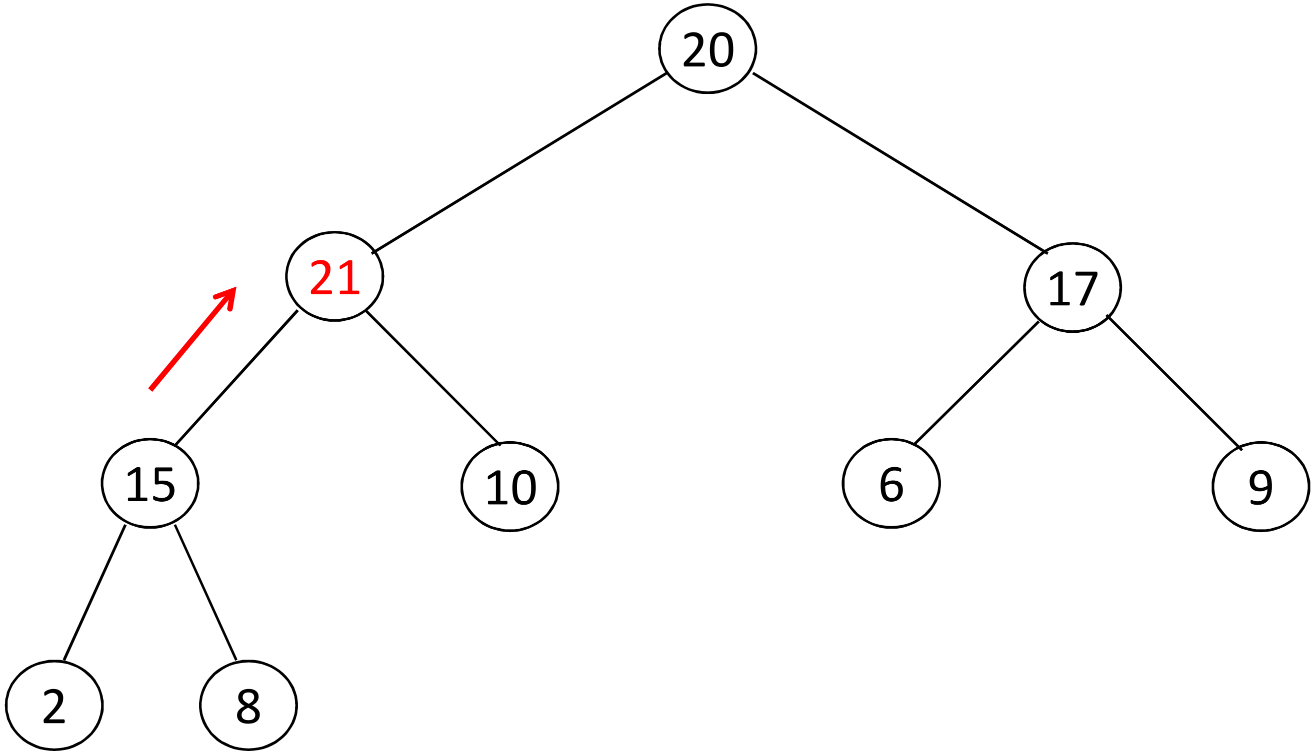
- Push 21



Bubbling up : 8 <-> 21, heap property!

Push Example

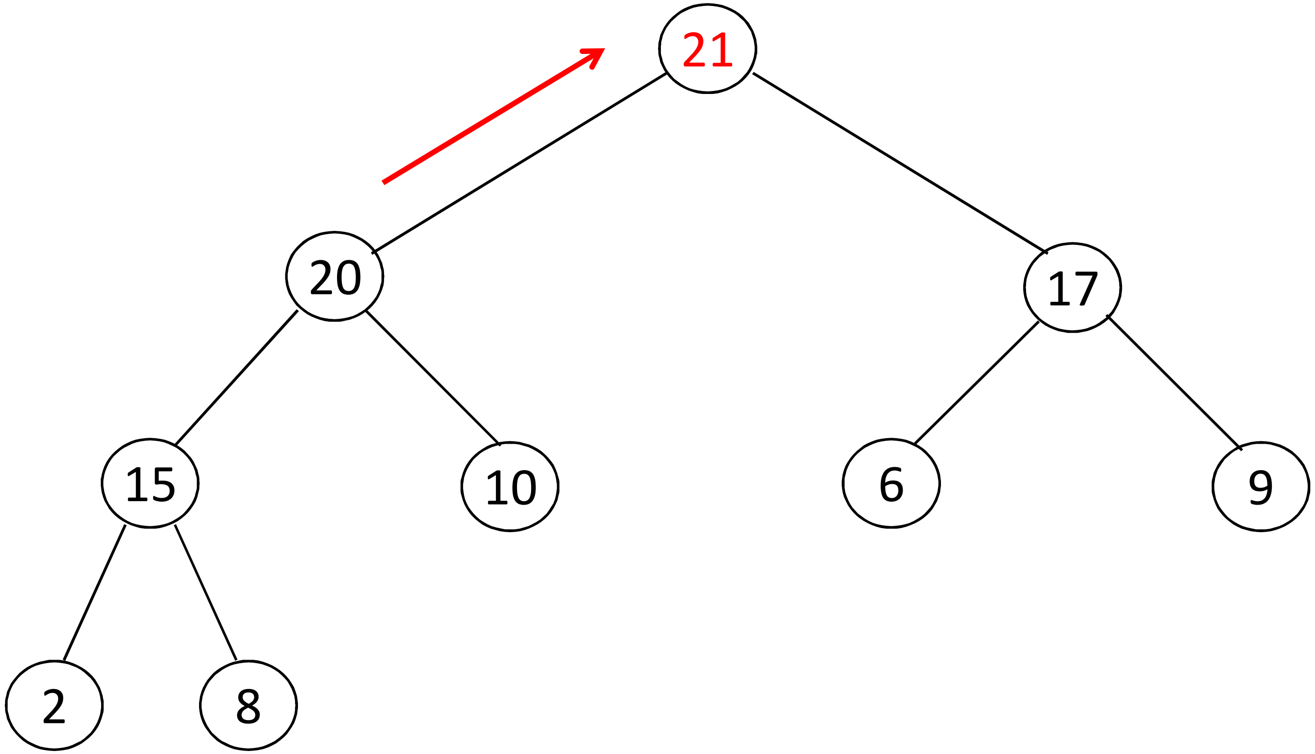
- Push 21



Bubbling up : 15 <-> 21

Push Example

- Push 21



Bubbling up : 20 <-> 21, done!

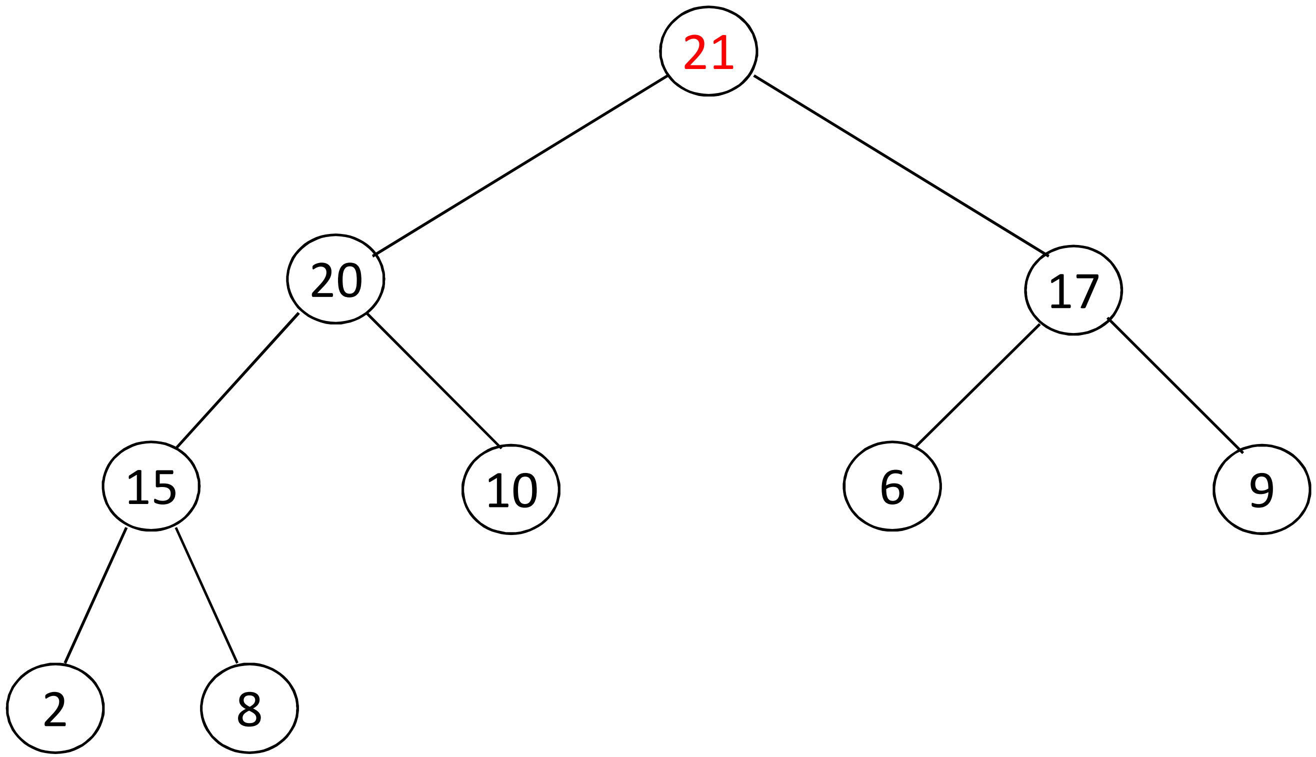
ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Max Heap

- Pop operation
  - Remove the largest value at root
  - Overwrite root with the last element
  - Remove last element
  - Trickle down while maintaining the max heap property

Pop Example

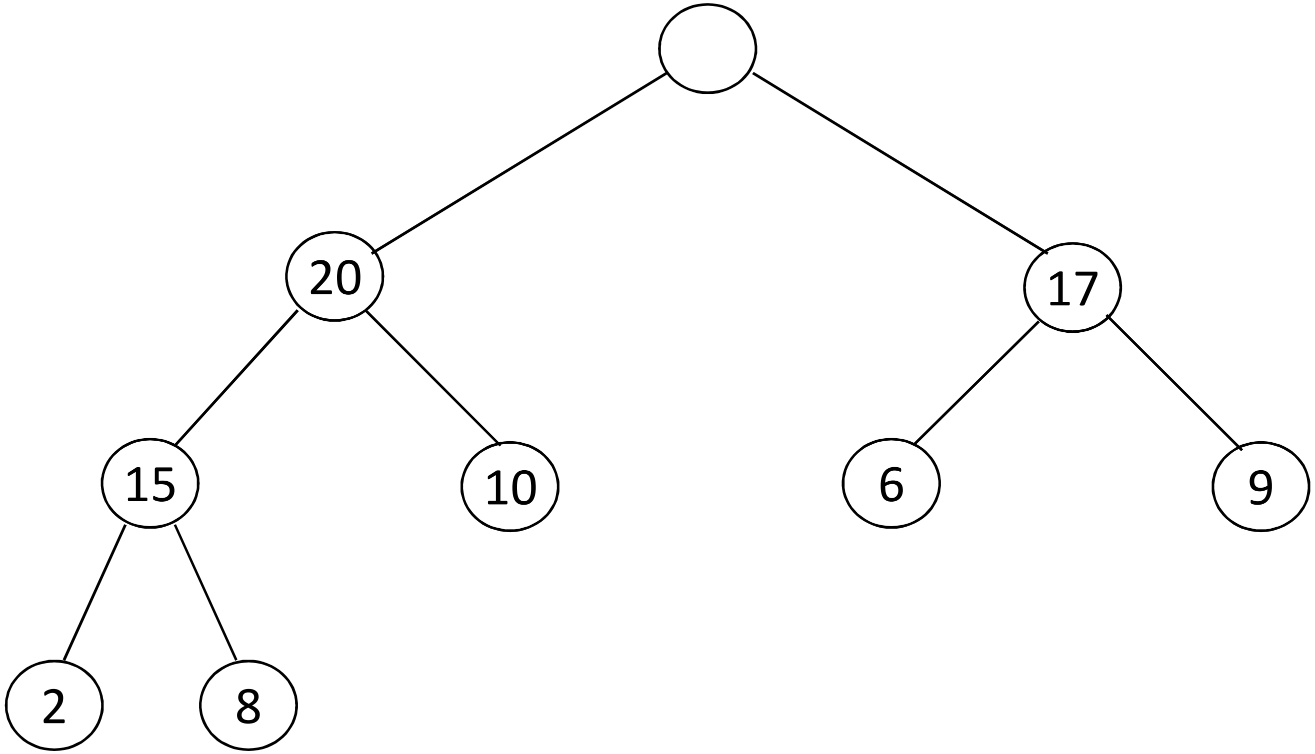
- Pop



ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Pop Example

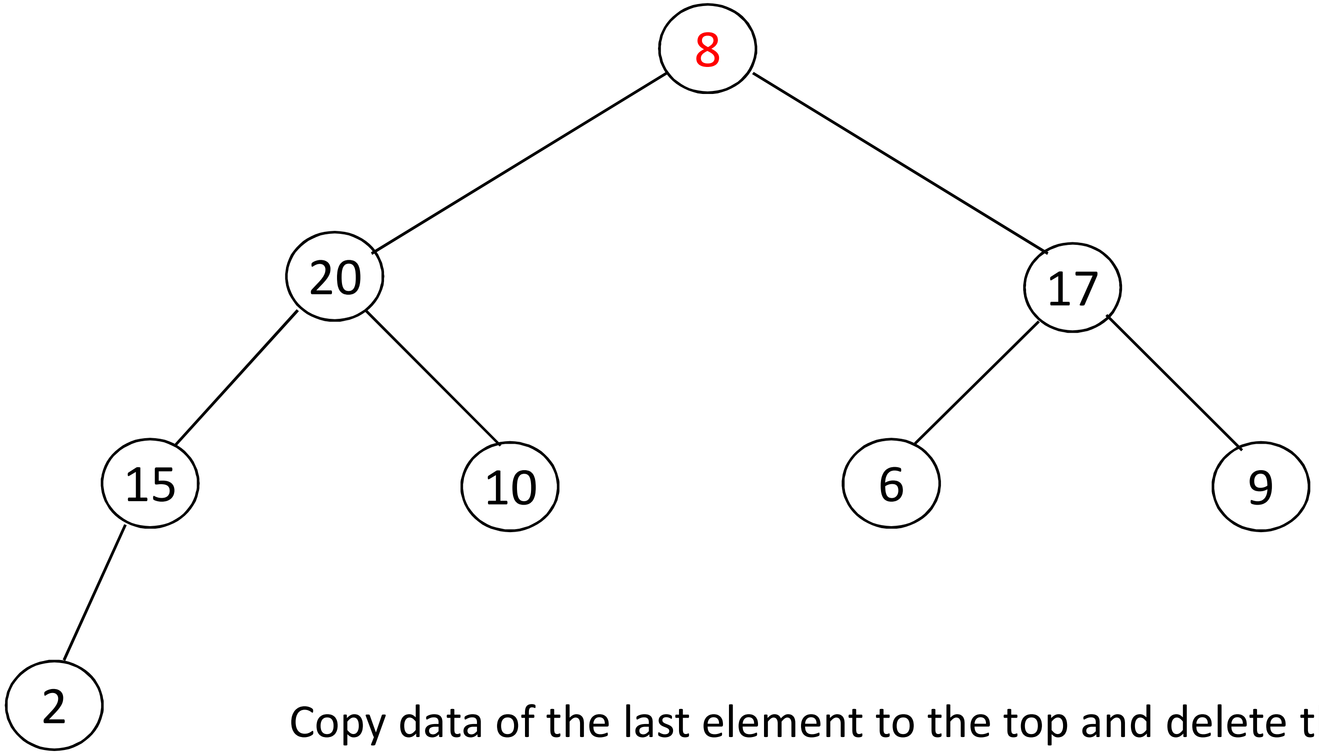
- Pop



Delete 21

Pop Example

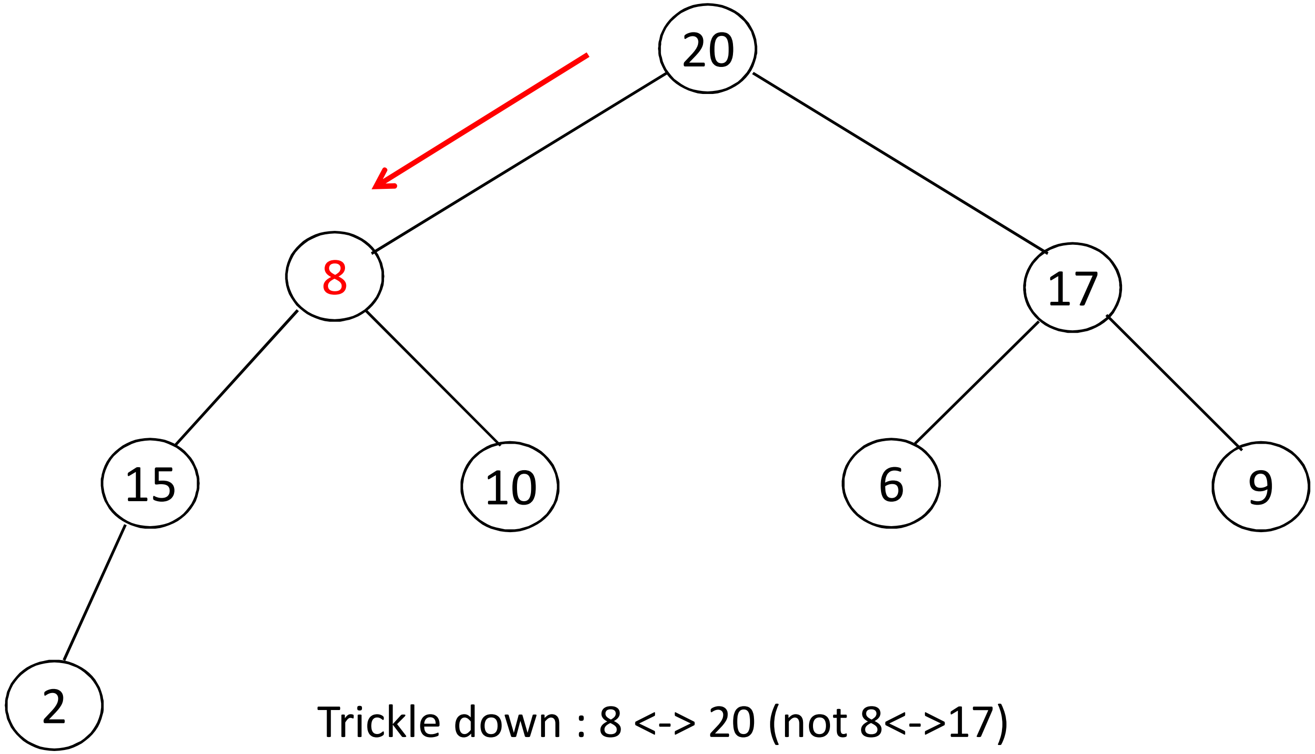
- Pop



Copy data of the last element to the top and delete the last node

Pop Example

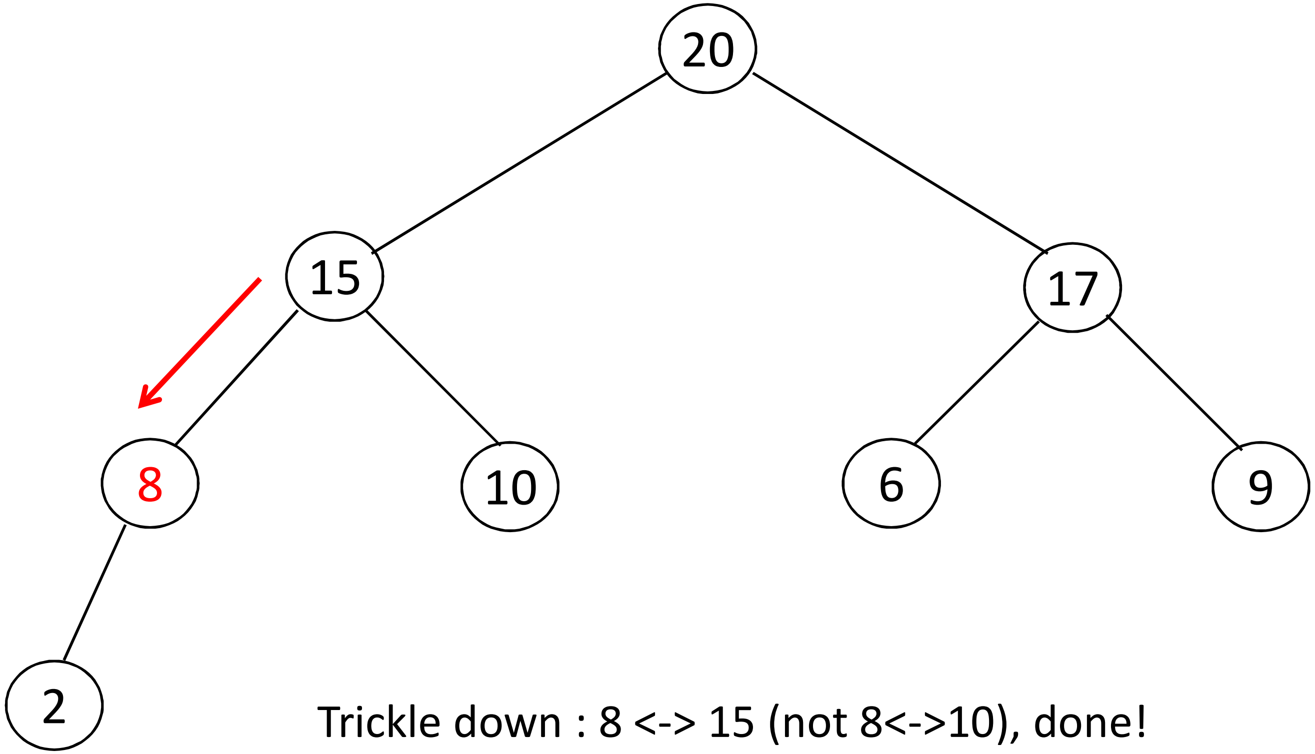
- Pop



Trickle down : 8 <-> 20 (not 8<->17)

Pop Example

- Pop



Trickle down : 8 <-> 15 (not 8<->10), done!

Heap-Sort

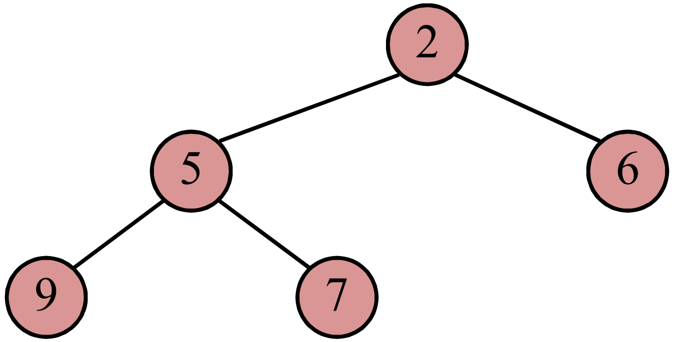
- Consider a priority queue with n items implemented by means of a min heap
  - the space used is $O(n)$
  - methods insert and removeMin take $O(\log n)$ time
  - methods size, empty, and min take time $O(1)$ time
- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort

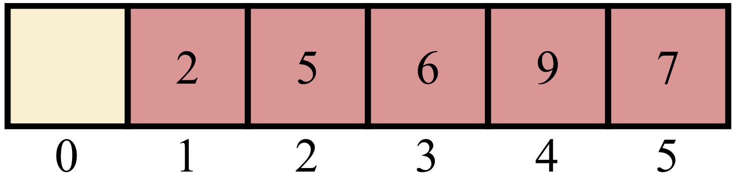
Implementation

- Can be implemented using array
  - Why? Complete binary tree
- Parent / children can be efficiently accessed
  - Parent : floor(i/2)
  - Left child : 2*i
  - Right child : 2*i + 1

Array-based Heap Implementation

- ***n*** keys on an array of length ***n* + 1**
- The cell of at index ***0*** is not used
- For the node at index ***i***
  - the left child is at index **2*i***
  - the right child is at index **2*i* + 1**
- Links between nodes are not explicitly stored
- insert corresponds to inserting at index ***n* + 1**
- removeMin corresponds to removing at index ***1***
- Yields in-place heap-sort





ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Push Algorithm

```
template <class Type>
void MaxHeap<Type>::Push(const Element<Type> &x)
{
   if (n==MaxSize) { HeapFull(); return; }
   n++;
   int i; // i : current node
   for(i=n; 1; ) {
       if (i==1) break; // Root reached
       if (x.key <= heap[i/2].key) break;
       // move parent down
       heap[i] = heap[i/2];
       i /= 2;
   }
   heap[i] = x;
}
```

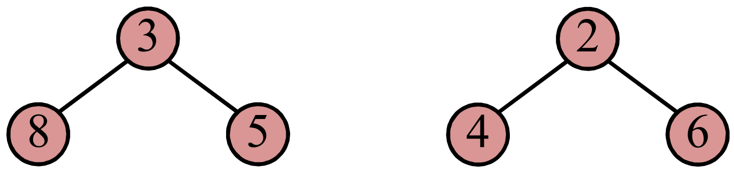
Pop Algorithm

```
template <class Type>
Element<Type> *MaxHeap<Type>::Pop(Element<Type> &x)
{
   if (!n) { HeapEmpty(); return 0;}
   x = heap[1];
   Element<Type> k = heap[n];
   n--;
   int i, j; // i : current node, j : child
   for (i=1, j=2; j<=n; )
   {
      if (j<n)
         if (heap[j].key < heap[j+1].key) j++;
      // j is the larger child
      if (k.key >= heap[j].key) break;
      heap[i] = heap[j]; // move the child up
      i = j; j *= 2; // move down a level
   }
   heap[i] = k;
   return &x;
}
```

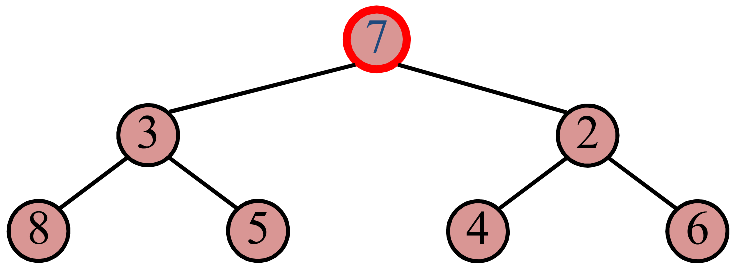
Push/Pop time complexity : O(log n)

Merging Two Heaps

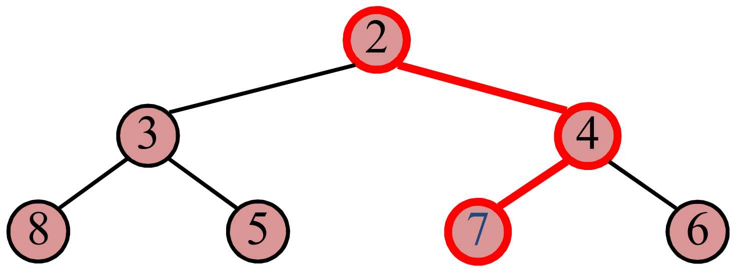
- We are given two heaps and a key ***k***



- We create a new heap with the root node storing ***k*** and with the two heaps as subtrees

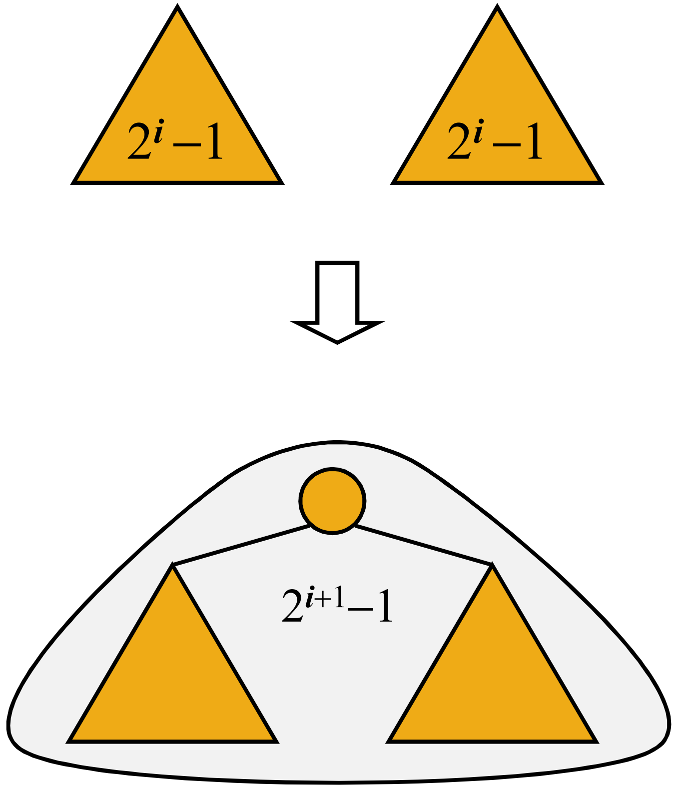


- We trickle down to restore the heap-order property

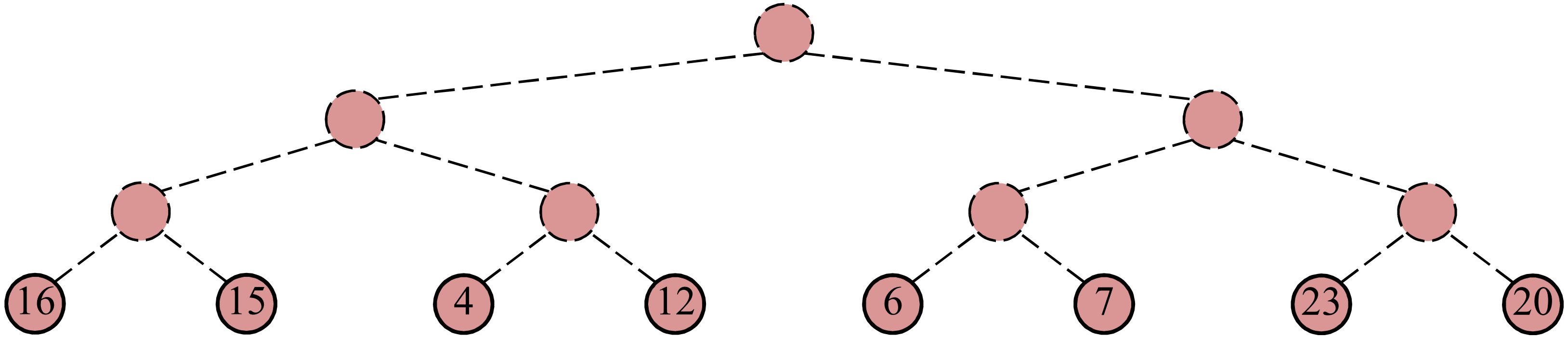


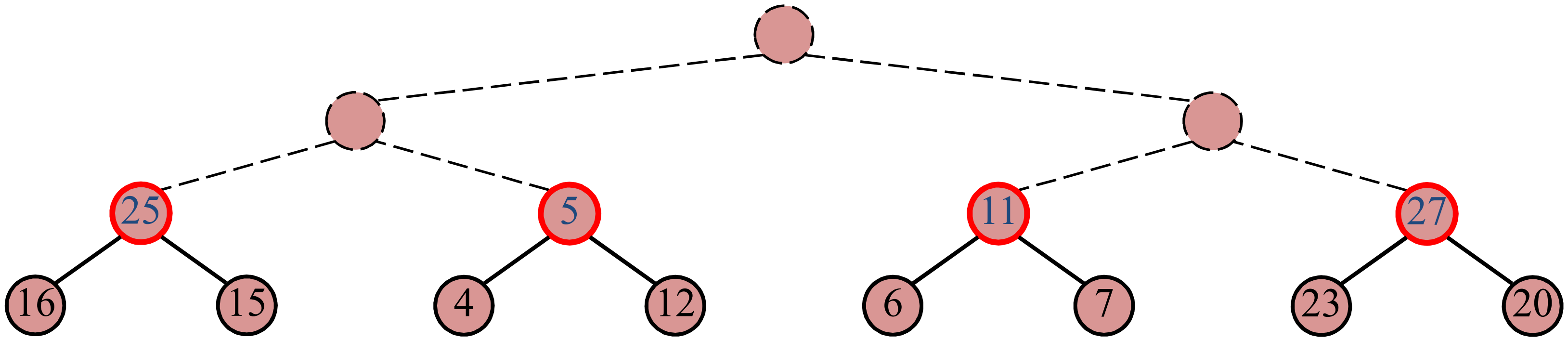
Bottom-up Heap Construction

- We can construct a heap storing ***n*** given keys in using a bottom-up construction with log ***n*** phases
- In phase ***i***, pairs of heaps with 2^i -1 keys are merged into heaps with 2^{i+1}-1 keys
- Higher opportunity for parallel execution

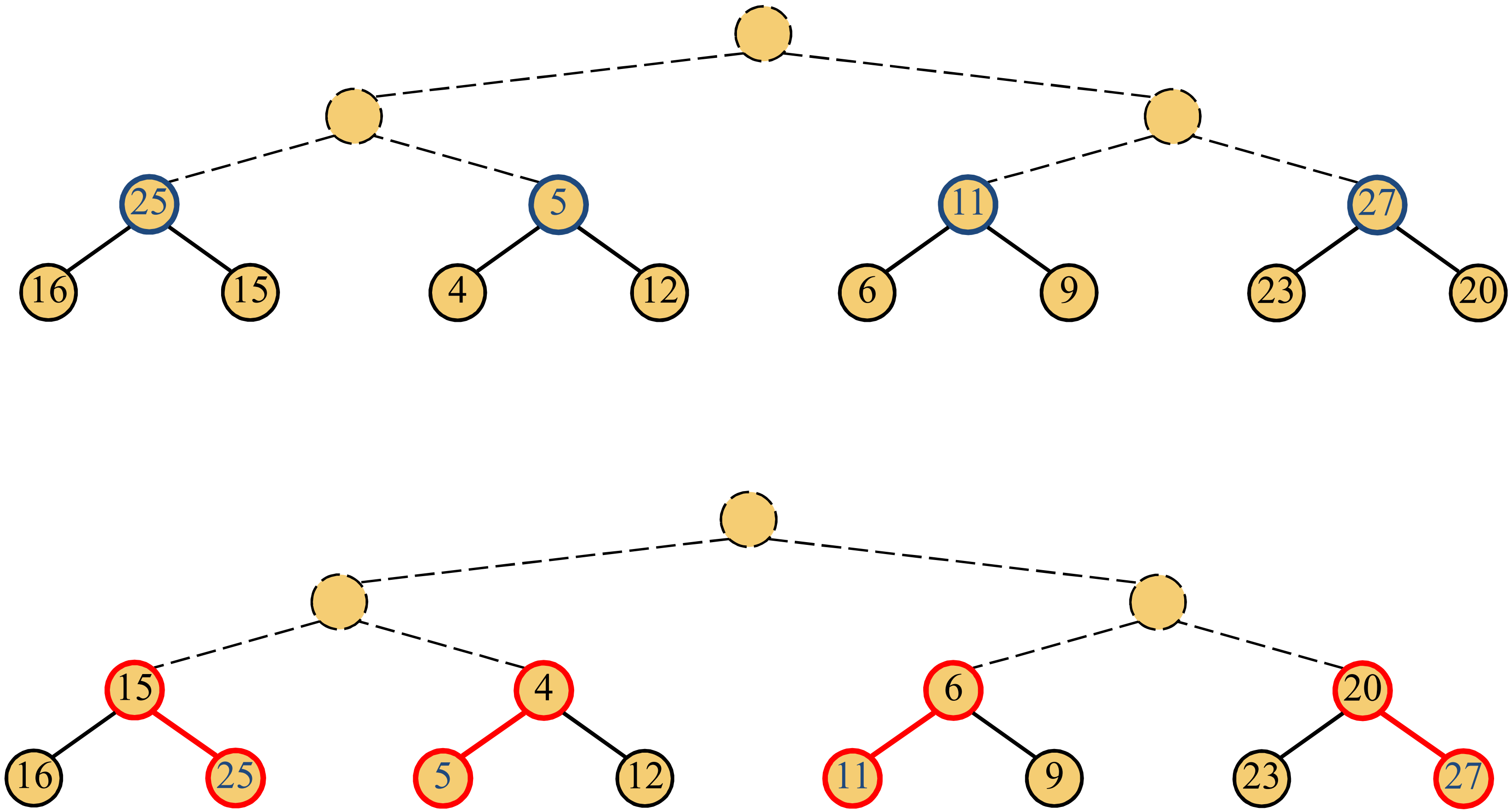


Example (Min Heap)

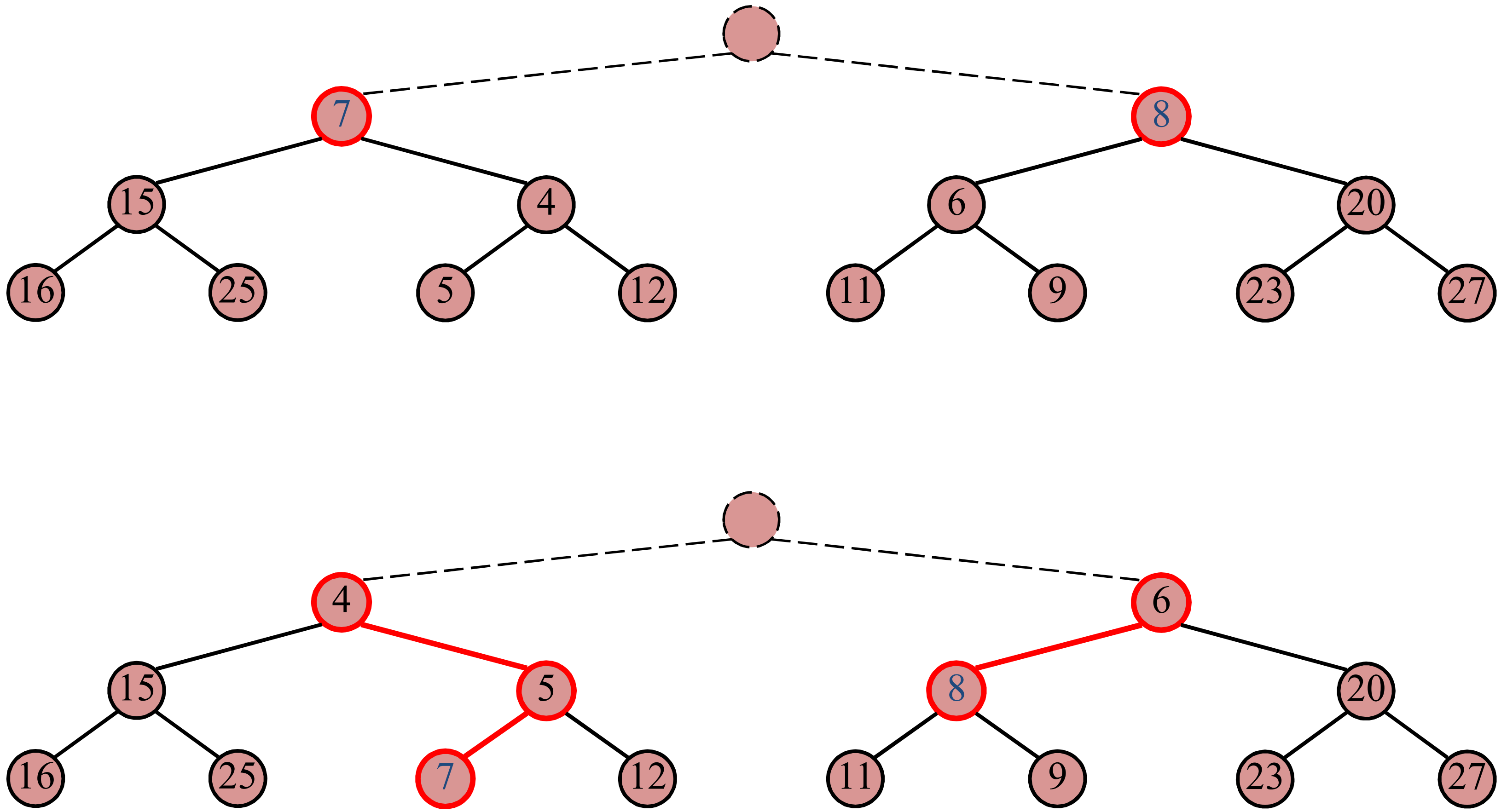




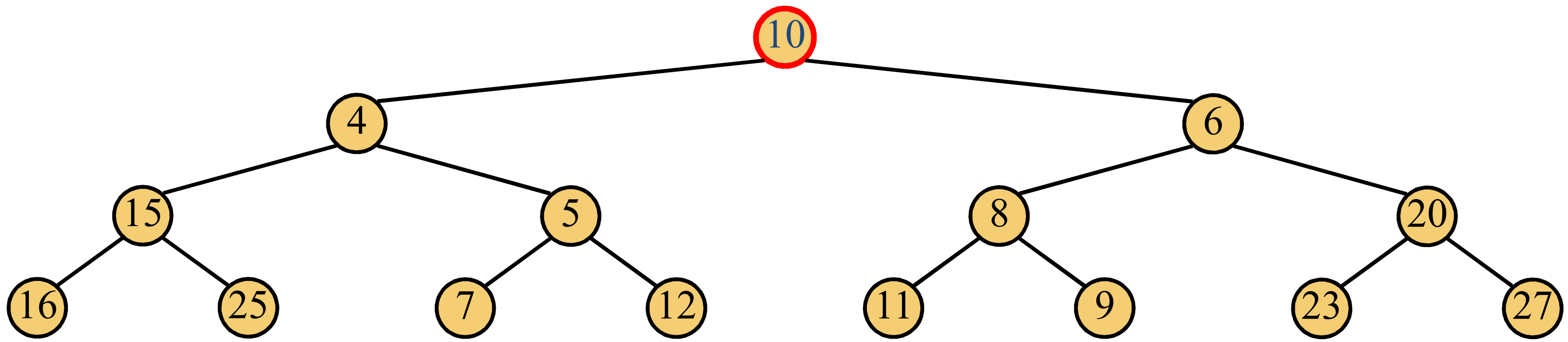
Example (Min Heap)

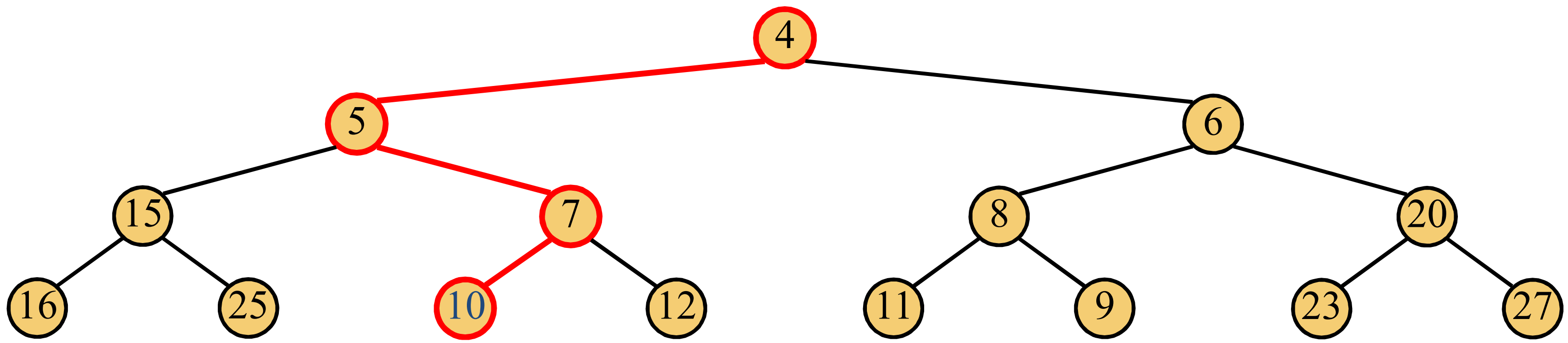


Example (Min Heap)



Example (Min Heap)





Questions?