CSE221

Lecture 10:
Heaps and Priority Queues

Hyungon Moon

Acknowledgment: The content of this file is based on the slides of the textbook as well as the slides provided by Prof. Won-Ki Jeong.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Outline

- Priority queues
- Heaps

Outline

- Priority queues
- Heaps

Priority Queue

- Queue with priority order for *pop*
  - Not FIFO
- Max priority queue
  - Pop the element with a highest priority first
- Min priority queue
  - Pop the element with a lowest priority first
- Unordered linear list for priority queue
  - O(1) for push, O(n) for top and pop

Priority Queue ADT

- A priority queue stores a collection of entries
- Typically, an entry is a pair (key, value), where the key indicates the priority
- Main methods of the Priority Queue ADT
  - insert(e)
    inserts an entry e
  - removeMin()
    removes the entry with smallest key
- Additional methods
  - min()
    returns, but does not remove, an entry with smallest key
  - size(), empty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

Total Order Relations

- Keys in a priority queue can be arbitrary objects on which an order is defined
- Two distinct entries in a priority queue can have the same key

- Mathematical concept of total order relation $\leq$
  - Reflexive property: $x \leq x$
  - Antisymmetric property: $x \leq y \wedge y \leq x \Rightarrow x = y$
  - Transitive property: $x \leq y \wedge y \leq z \Rightarrow x \leq z$

Comparator ADT

- Implements the boolean function isLess(p,q), which tests whether p < q
- Can derive other relations from this:
  - (p == q) is equivalent to
  - (!isLess(p, q) && !isLess(q, p))
- Can implement in C++ by overloading "()"

Two ways to compare 2D points:

```
class LeftRight { // left-right comparator
public:
    bool operator()(const Point2D& p,
     const Point2D& q) const
    { return p.getX() < q.getX(); }
};
class BottomTop { // bottom-top
public:
    bool operator()(const Point2D& p,
    const Point2D& q) const
    { return p.getY() < q.getY(); }
};
```

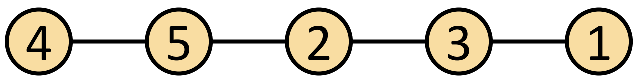
Priority Queue Sorting

- We can use a priority queue to sort a set of comparable elements
  1. Insert the elements one by one with a series of insert operations
  2. Remove the elements in sorted order with a series of 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 elements 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***)

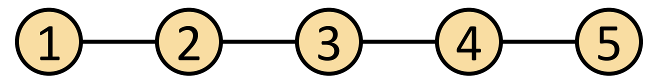
Sequence-based Priority Queue

- 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

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

$$1 + 2 + \ldots + n$$

- Selection-sort runs in $O(n^2)$ 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

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

     $$1 + 2 + \ldots + n$$

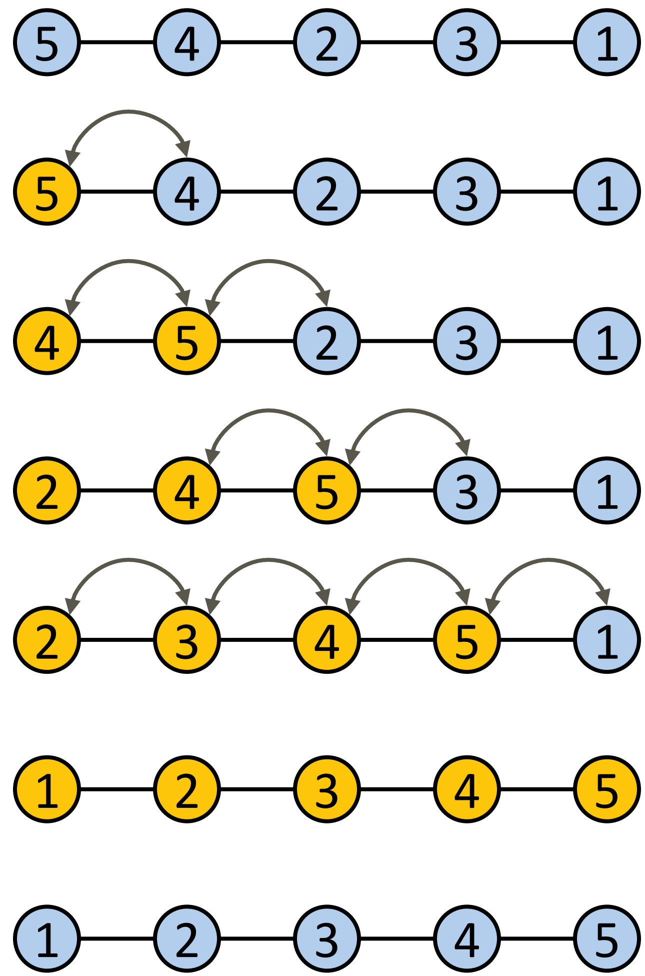
  2. Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes $O(n)$ time
- Insertion-sort runs in $O(n^2)$ time

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)	()

In-place Insertion-Sort

- Instead of using an external data structure, we can implement selection-sort and insertion-sort in-place
- A portion of the input sequence itself serves as the priority queue
- For in-place insertion-sort
  - We keep sorted the initial portion of the sequence
  - We can use swaps instead of modifying the sequence



Outline

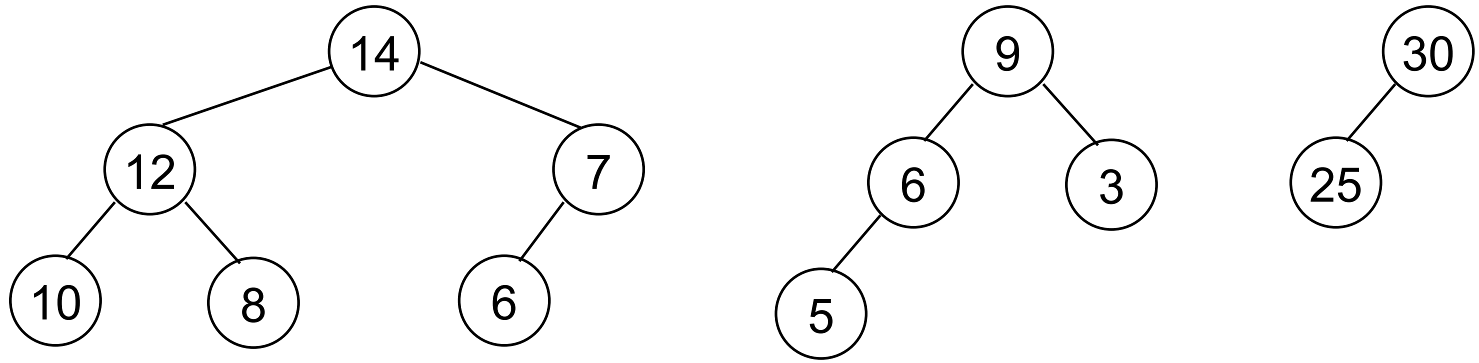
- Priority queues
- Heaps

Heap

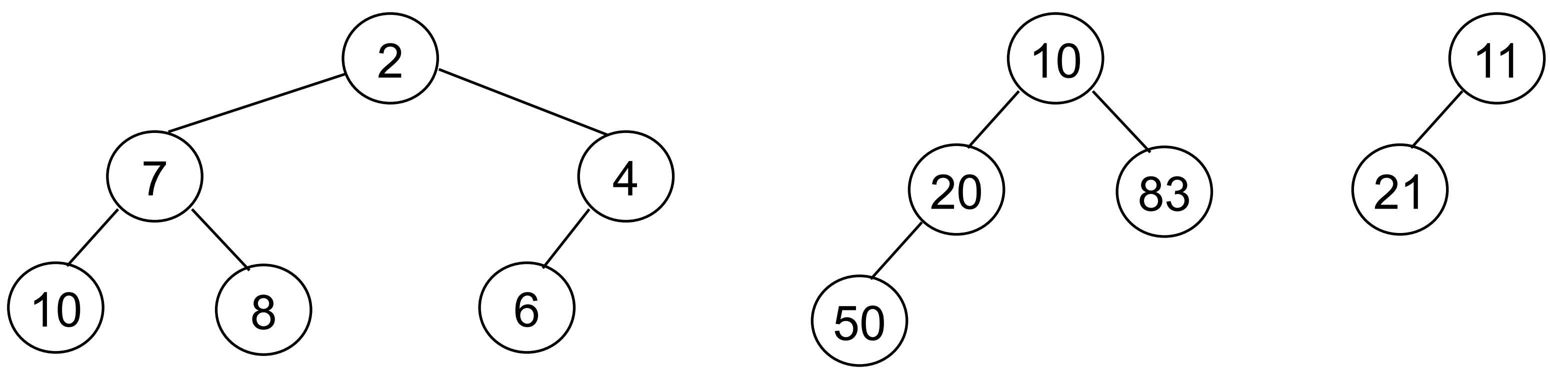
- A heap is a specialized tree-based data structure that satisfies the heap property
  – If A is a parent node of B then key(A) is ordered with respect to key(B) with the same ordering applying across the heap (*wikipedia*)
- Max (min) heap
  – A complete binary tree with the heap property that the key value in each node is no smaller (larger) than the key values in its children

Heap

Max heap



Min heap



Max Heap

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

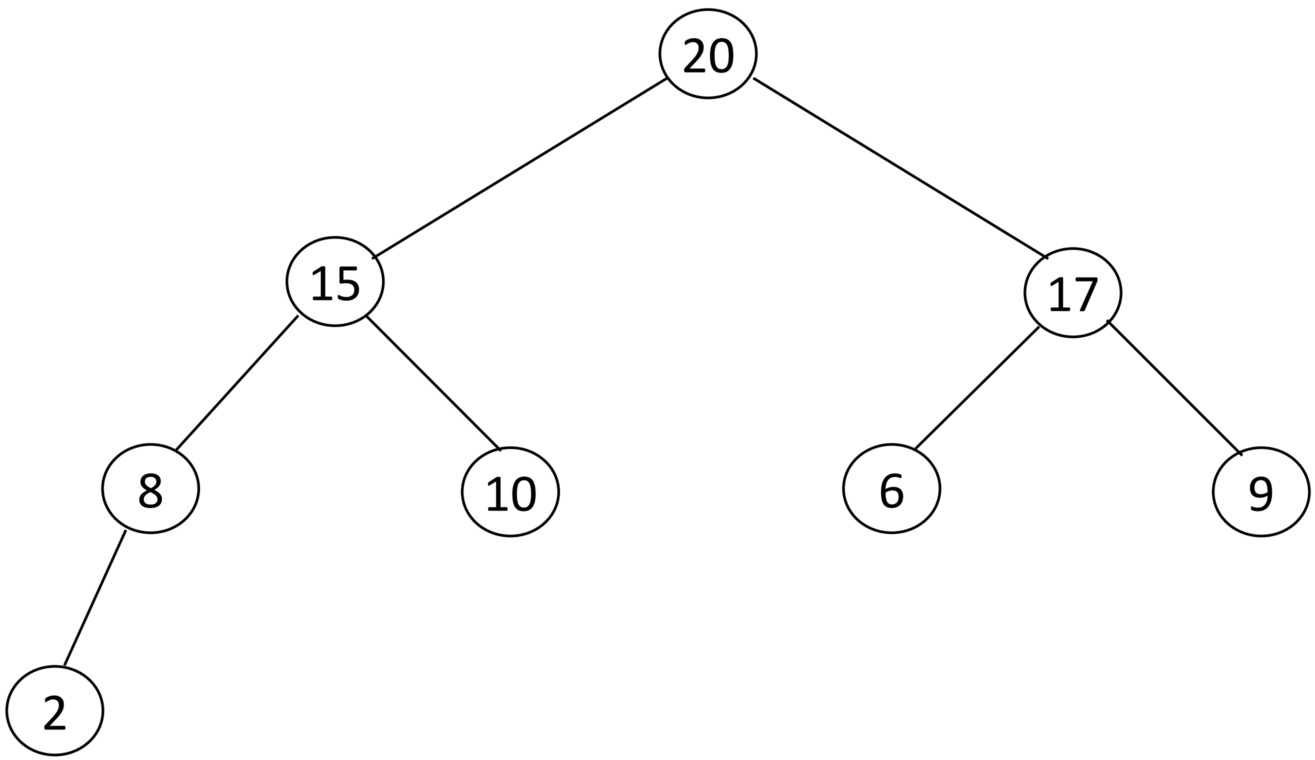
Max Heap

- Push
  - Add new element at the end of the tree
  - Bubbling up

```
template <class Type>
void MaxHeap<Type>::Push(const Element<Type> &x)
{
   if (n==MaxSize) { HeapFull(); return; }
   n++;
   for(int 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;
}
```

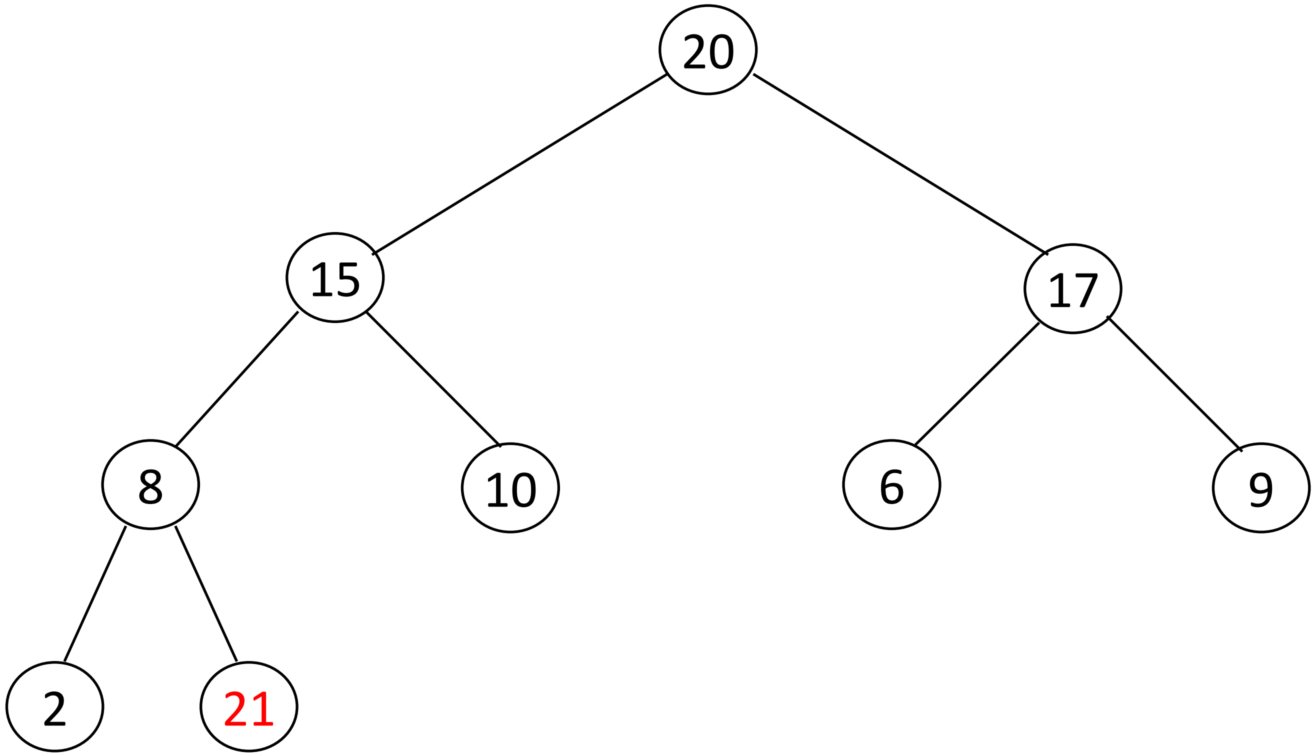
Push Example

- Push 21



Push Example

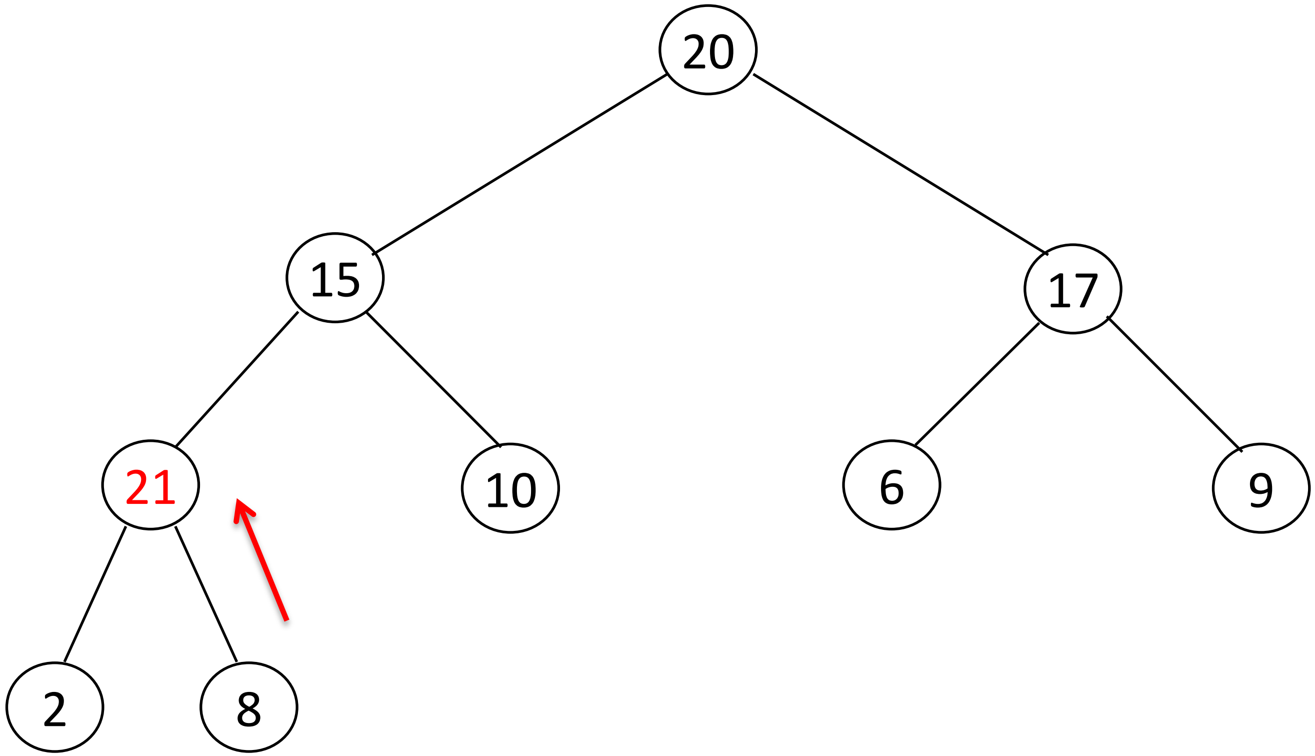
- Push 21



Create node at the end of the heap

Push Example

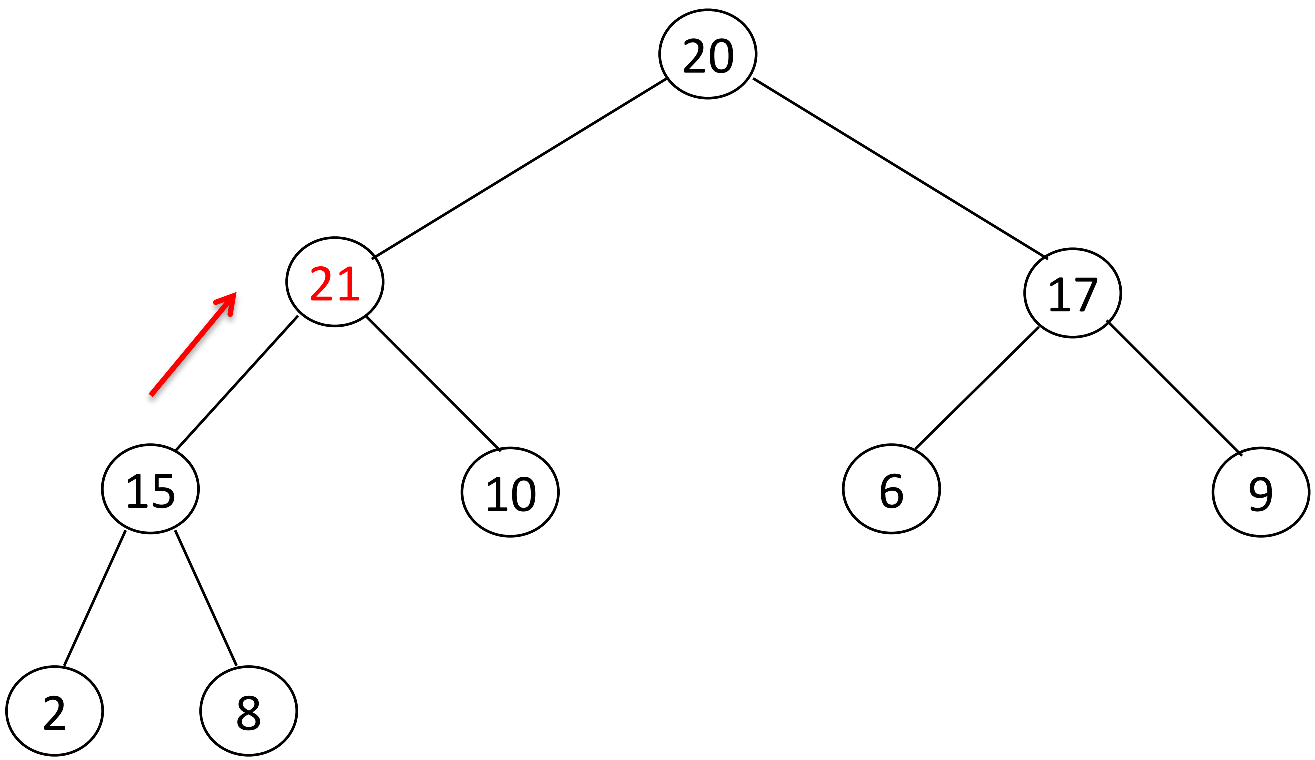
- Push 21



Bubbling up : 8 <-> 21

Push Example

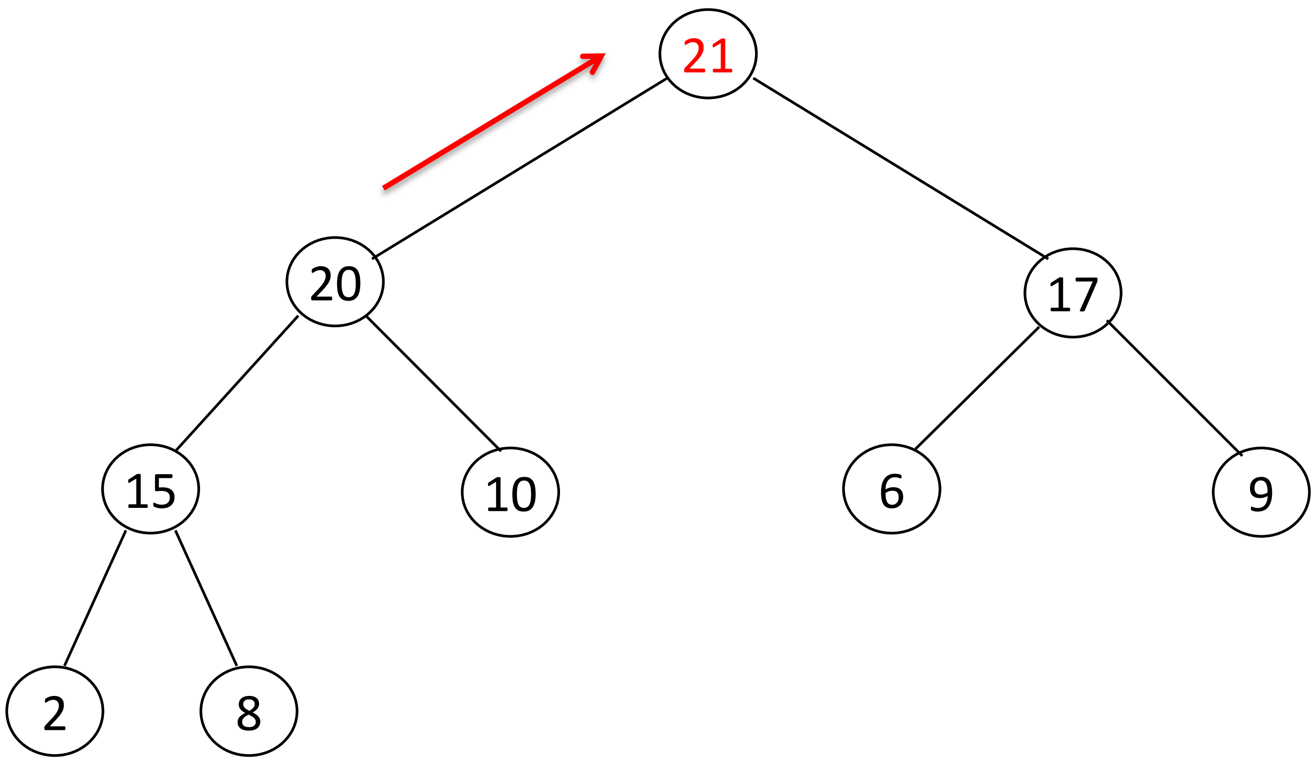
- Push 21



Bubbling up : 15 <-> 21

Push Example

- Push 21



Bubbling up : 20 <-> 21, done!

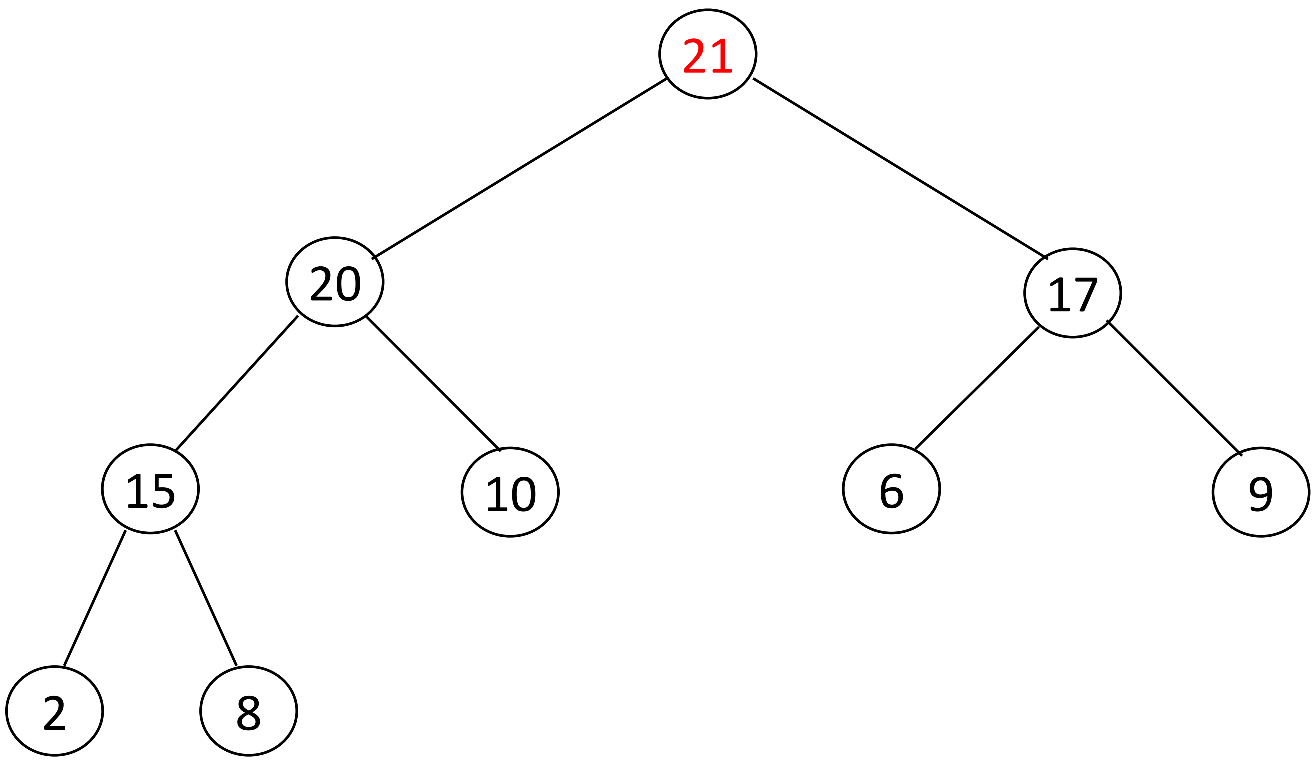
Max Heap

- Pop
  - Overwrite root with the last element
  - Remove last element
  - Trickle down

```
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--;
   // i : current node, j : child
   for (int 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;
}
```

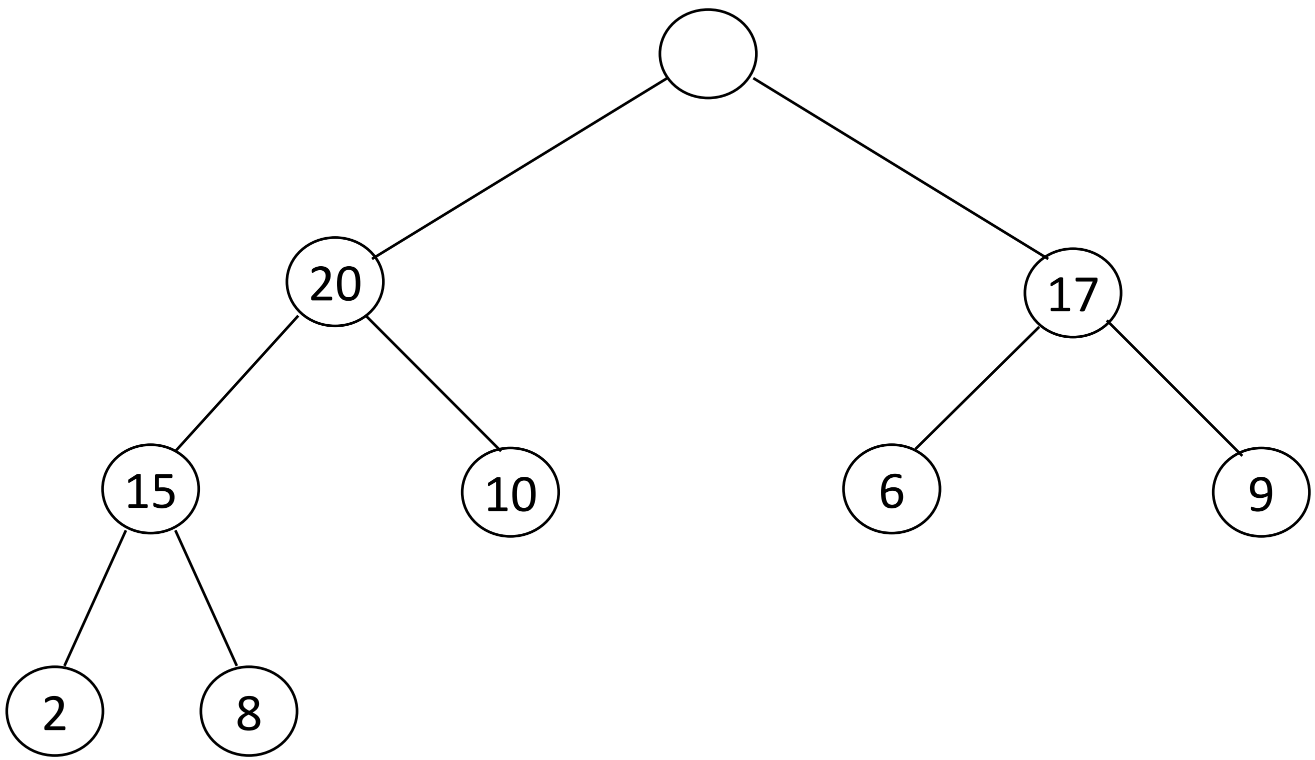
Pop Example

- Pop



Pop Example

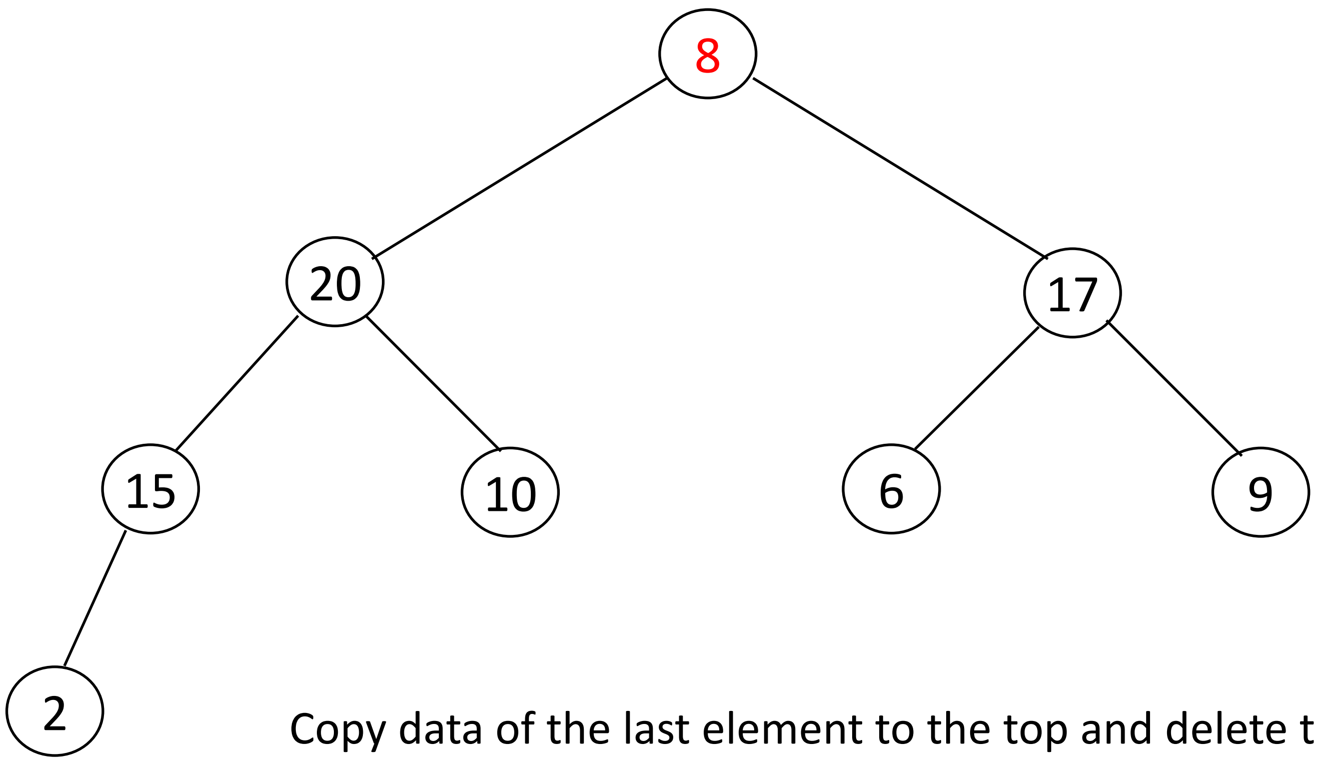
- Pop



Delete 21

Pop Example

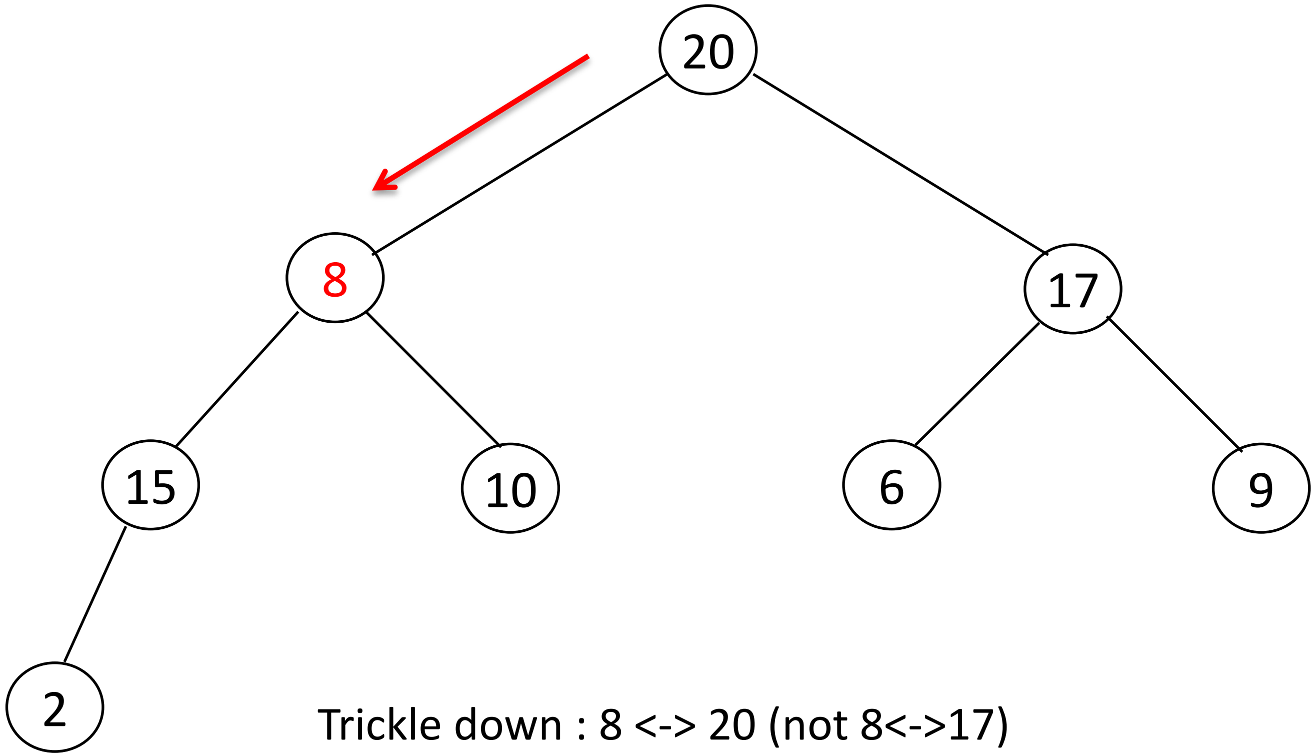
- Pop



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

Pop Example

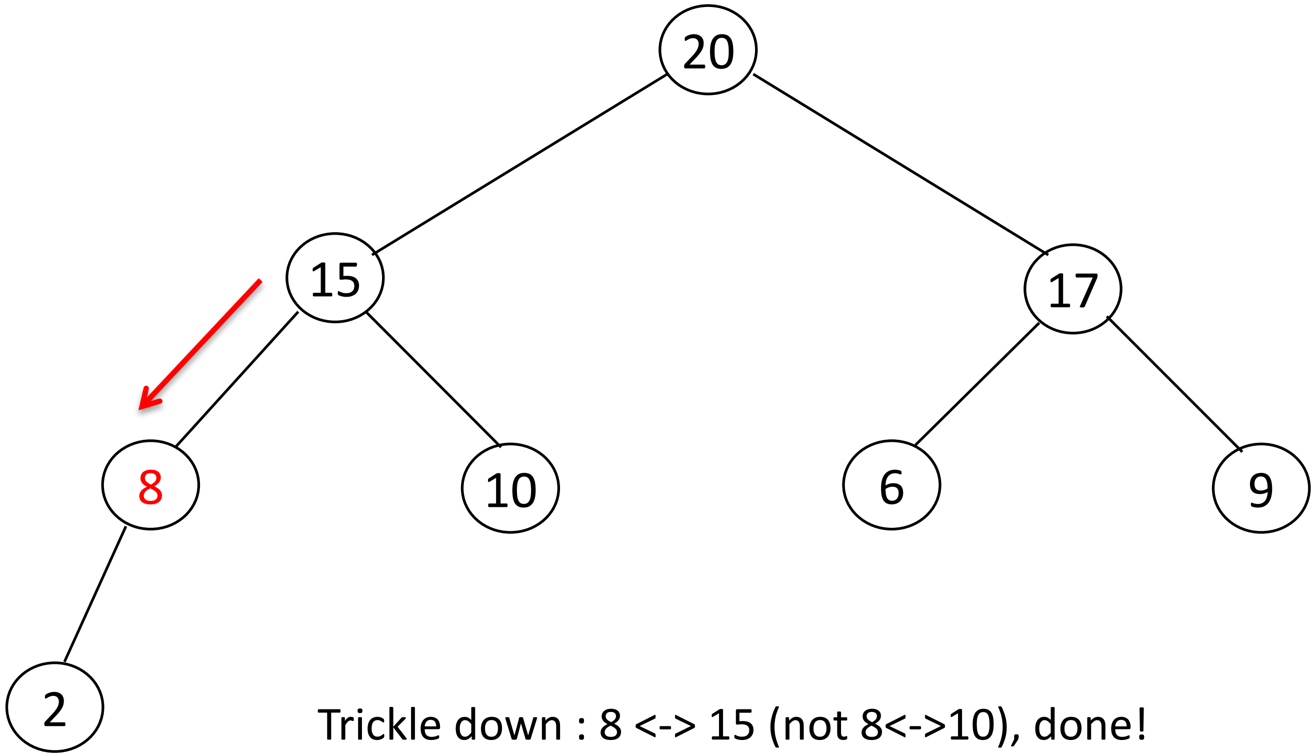
- Pop



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

Pop Example

- Pop



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

Note

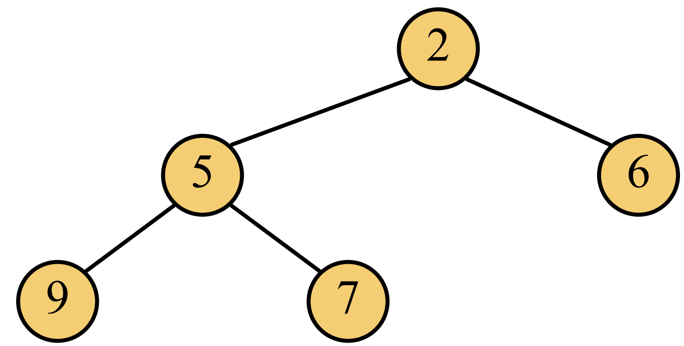
- Push/pop time complexity : O(log n)
  - Heap sort?
- Siblings are not ordered
  - No guarantee left < right
    - Only parent >= children is guaranteed
  - Find the larger child to trickle down
- When bubbling up / trickling down, find the final position by shifting parents / children
  - No swapping

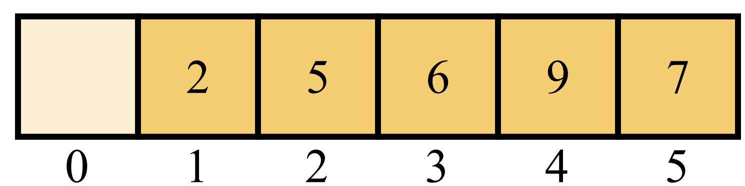
Heap-Sort

- Consider a priority queue with n items implemented by means of a 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

Vector-based Heap Implementation

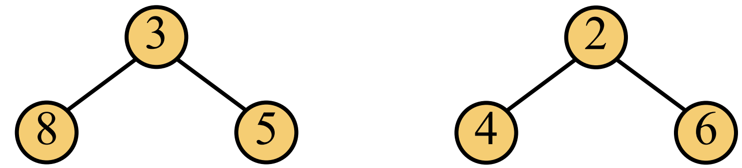
- We can represent a heap with n keys by means of a vector of length $n + 1$
- For the node at rank i
  - the left child is at rank $2i$
  - the right child is at rank $2i + 1$
- Links between nodes are not explicitly stored
- The cell of at rank 0 is not used
- Operation insert corresponds to inserting at rank $n + 1$
- Operation removeMin corresponds to removing at rank n
- Yields in-place heap-sort



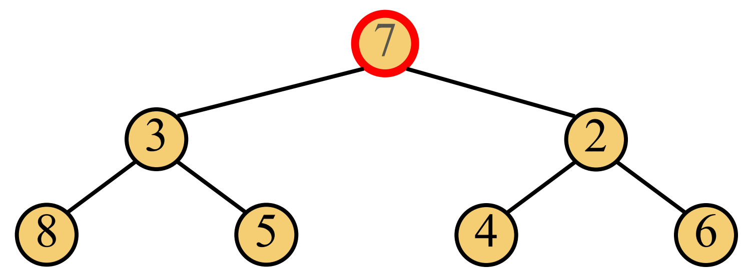


Merging Two Heaps

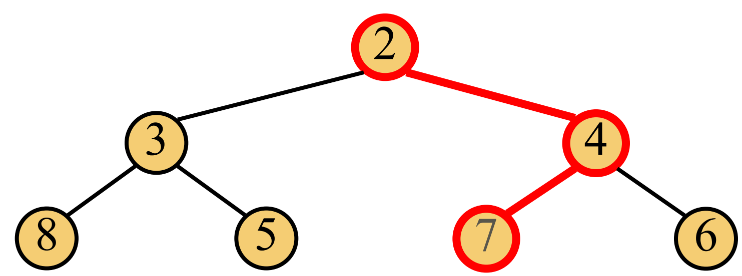
- We are given two two heaps and a key k



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

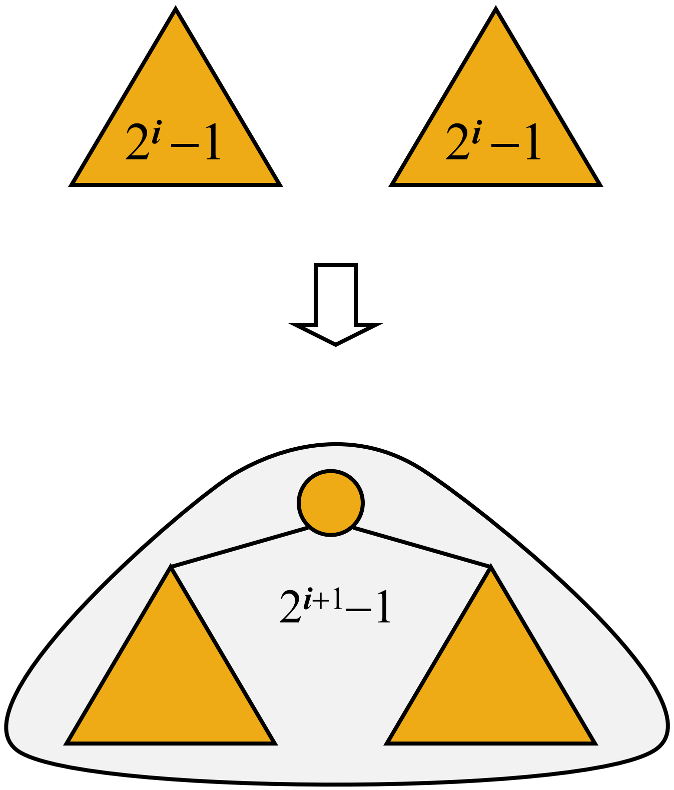


- We perform downheap to restore the heap-order property

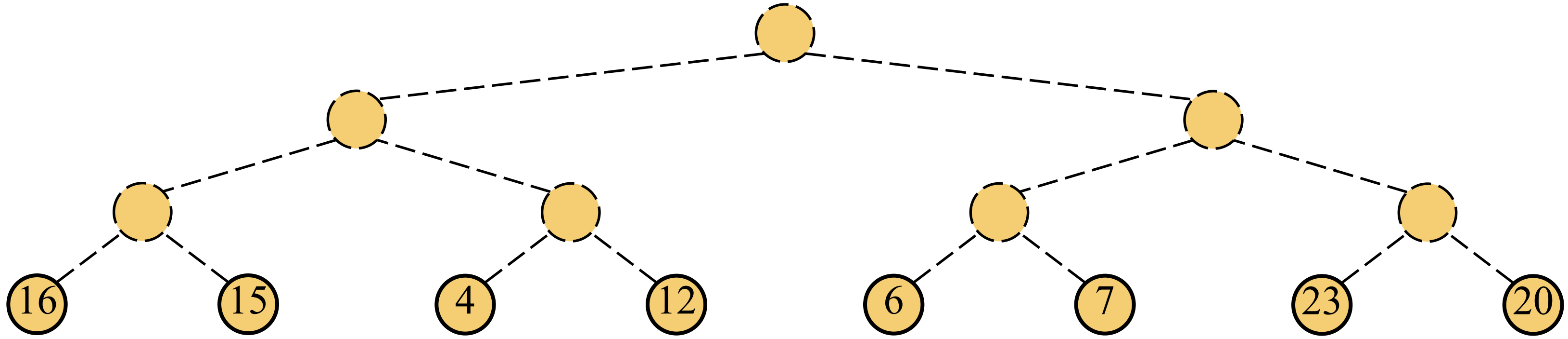


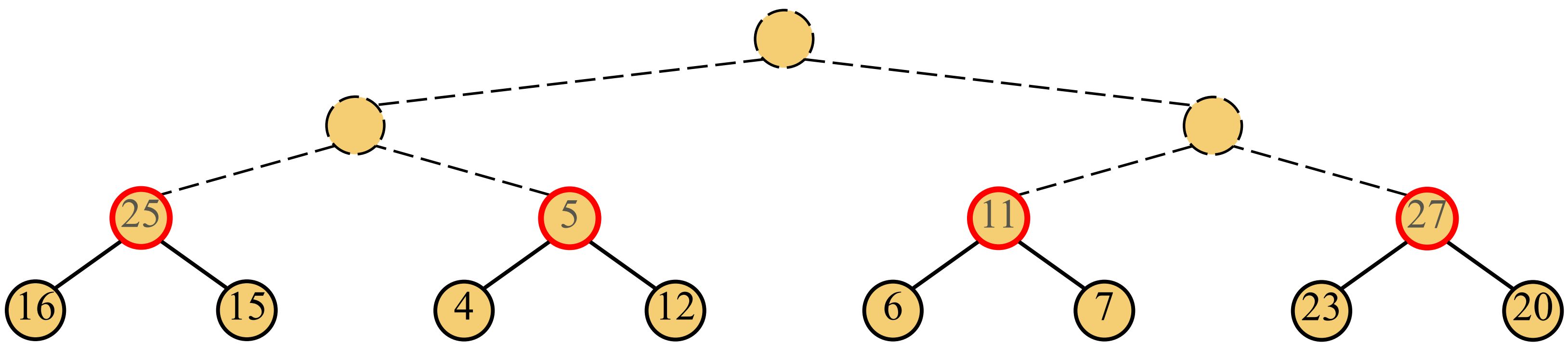
Bottom-up Heap Construction

- 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

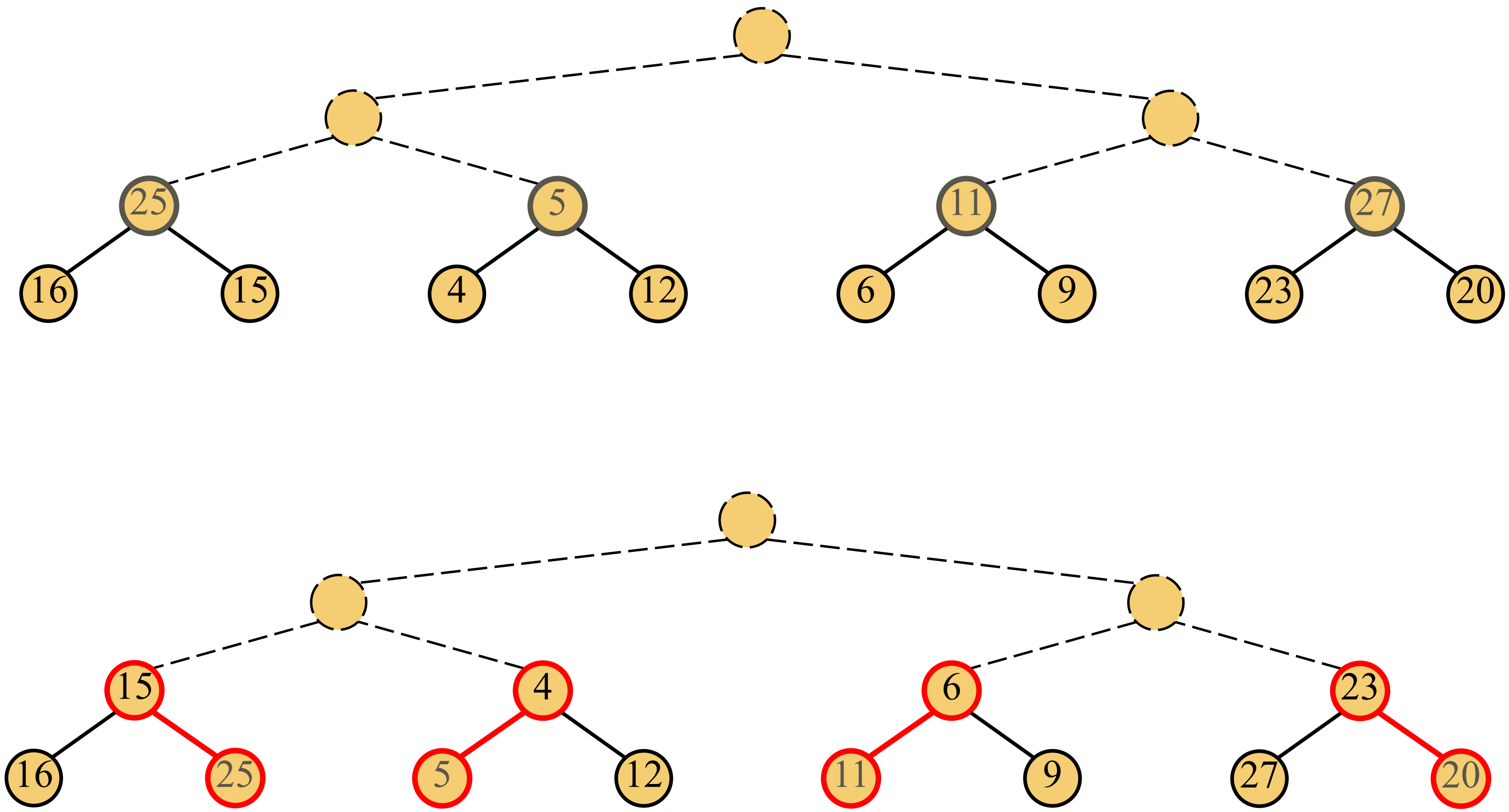


Example

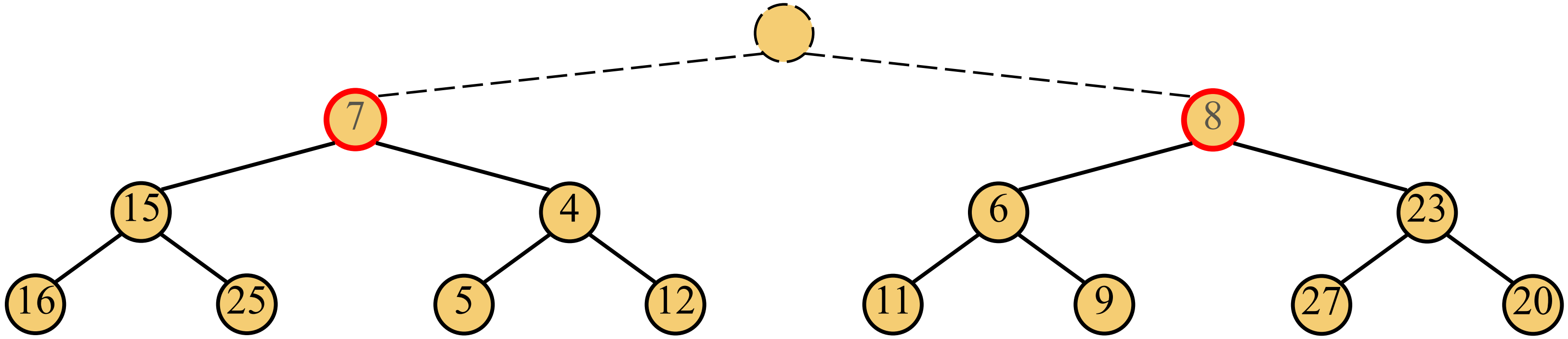


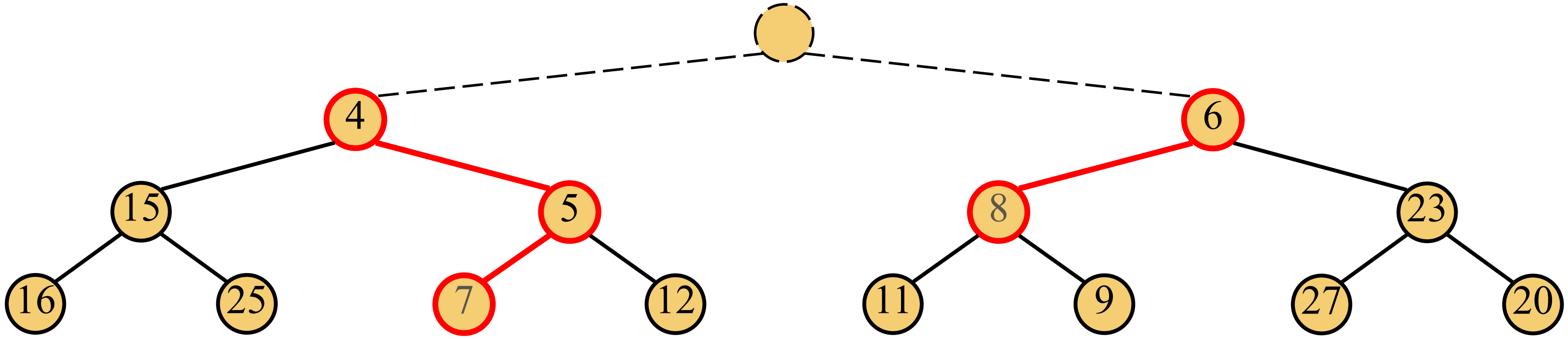


Example (contd.)

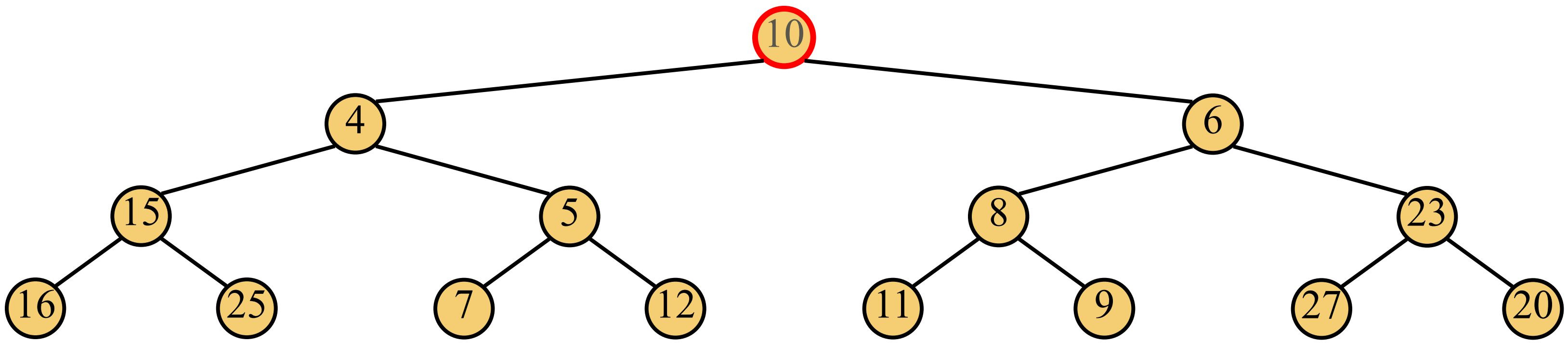


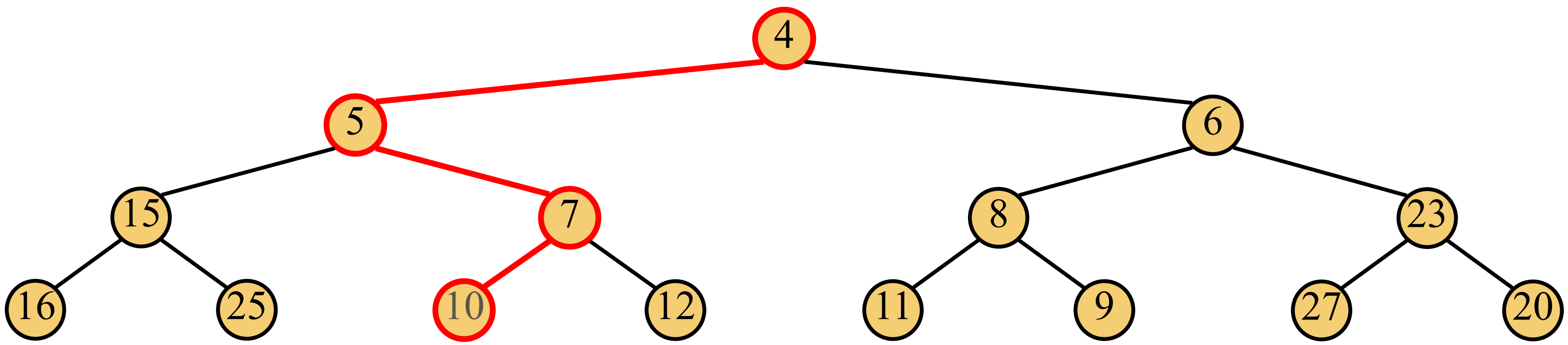
Example (contd.)





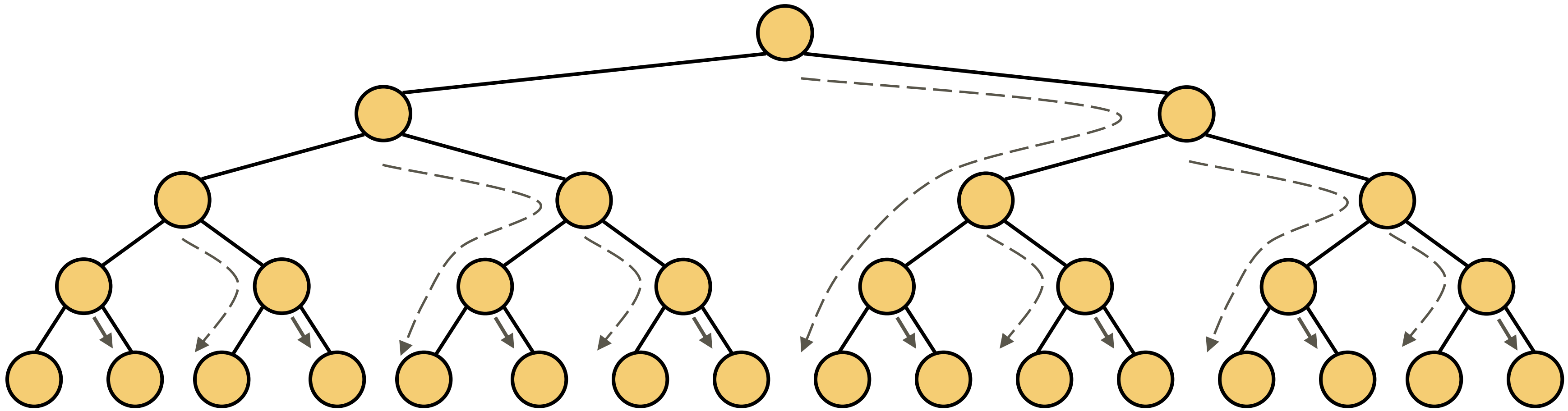
Example (end)





Analysis

- We visualize the worst-case time of a downheap with a proxy path that goes first right and then repeatedly goes left until the bottom of the heap (this path may differ from the actual downheap path)
- Since each node is traversed by at most two proxy paths, the total number of nodes of the proxy paths is $O(n)$
- Thus, bottom-up heap construction runs in $O(n)$ time
- Bottom-up heap construction is faster than n successive insertions and speeds up the first phase of heap-sort



Questions?