MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

29 Heaps Continued and Sorting

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

April 4, 2022

Admin

- I will publish the Assignment 3 after this lecture.
  - It is related to extending the ADT of Binary Search Tree with extra functions
  - I will provide a base code that you can extend it
  - I will provide a detailed video that describes the code
  - Ask questions in the QA channel
    - If your question is related to the base code, make sure that you have watched the video
- The marking of Assignment 2 is almost done. I will release the grades soon.
- Besides tomorrow's tutorial on program correctness, I will make a lecture note for the previous lecture. I hope both the lecture notes and the tutorial content clarify the concepts.

Review

- Priority Queue
  - Data structure for storing a collection of prioritized elements
    - Priority is defined based on keys
  - Supporting arbitrary element insertion
  - Supporting removal of elements in order of priority
  - Sorting based on Priority Queue:
    - keys stored in a **Sequence S**
    - create a **Priority Queue PQ**
    - While the **S** is not empty
      - get a key from sequence and insert it into the **PQ**
    - While **PQ** is not empty
      - remove the min of **PQ**
      - insert it back to **S**

Sequence-based Priority Queue (Review)

- 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

Priority Queue Sorting (Review)

- 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 depends on the priority queue implementation:
  - **Unsorted sequence gives selection-sort: $O(n^2)$ time**
  - **Sorted sequence gives insertion-sort: $O(n^2)$ time**

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***()

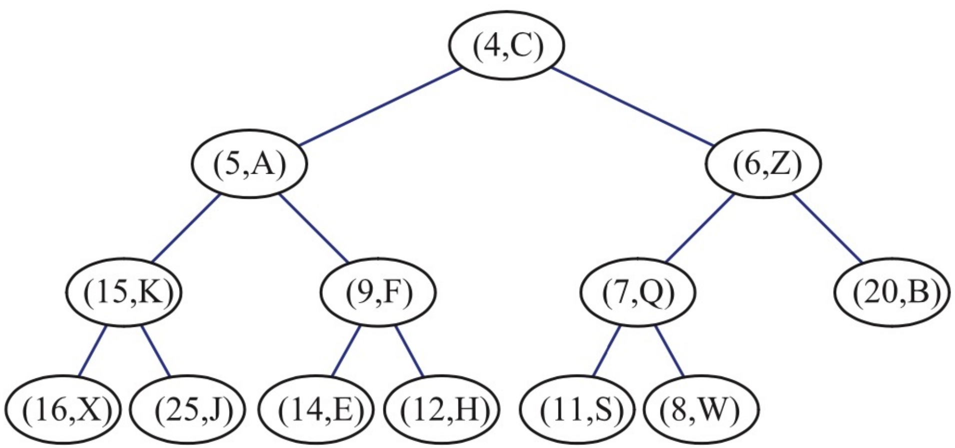
  e ← ***P.removeMin***()

  S.insertBack(***e***)

Review

Method	PQ Unsorted List	PQ Sorted List	Heap
insert	O(1)	O(n)	O(logn)
removeMin	O(n)	O(1)	O(logn)

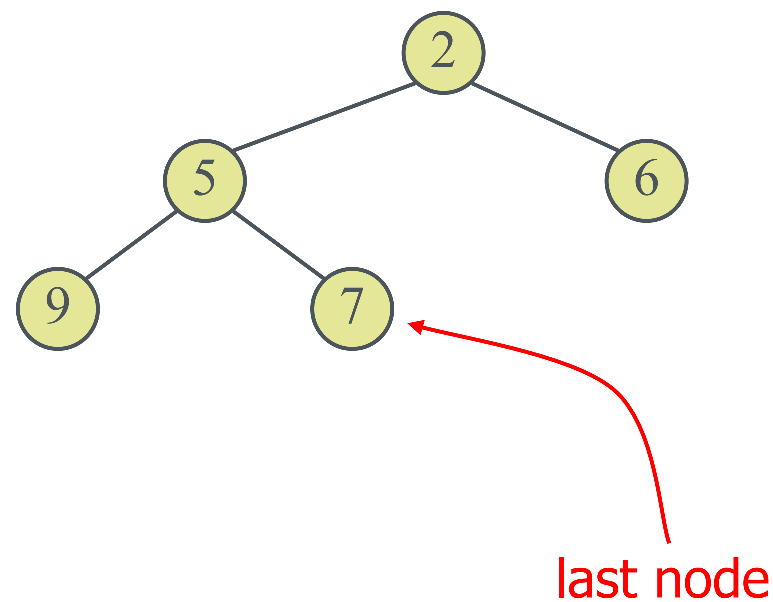
- Can we do better?
  - We can definitely get the better of both the worlds using the **heap** data structure.



Heaps (Review)

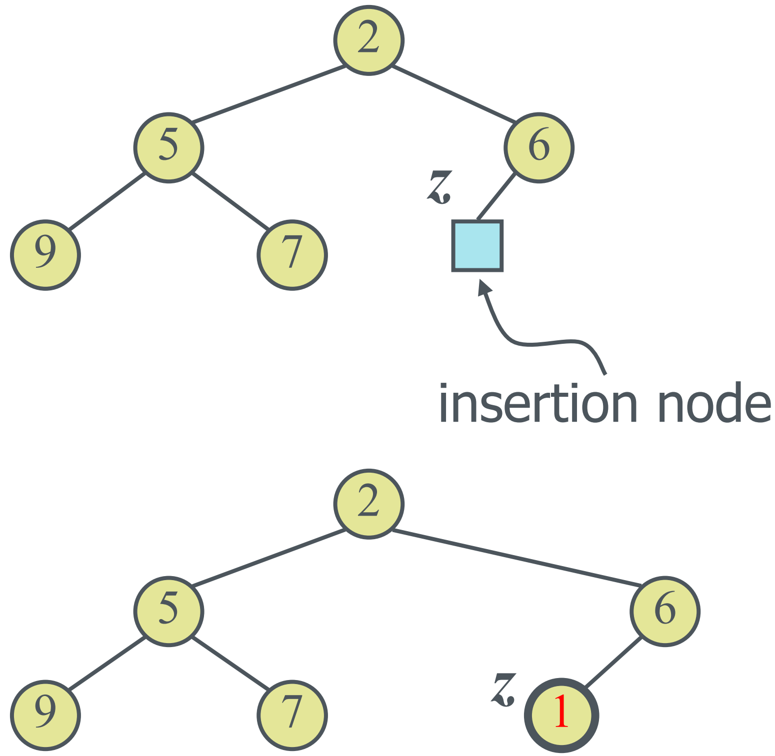
- A heap is a binary tree storing keys at its nodes and satisfying the following properties:
- Heap-Order: for every internal node v other than the root, $key(v) \geq key(parent(v))$
- Complete Binary Tree: let h be the height of the heap
  - for $i = 0, \ldots, h - 1$, there are 2^i nodes of depth i
  - at depth $h - 1$, the internal nodes are to the left of the external nodes

The last node of a heap is the rightmost node of maximum depth



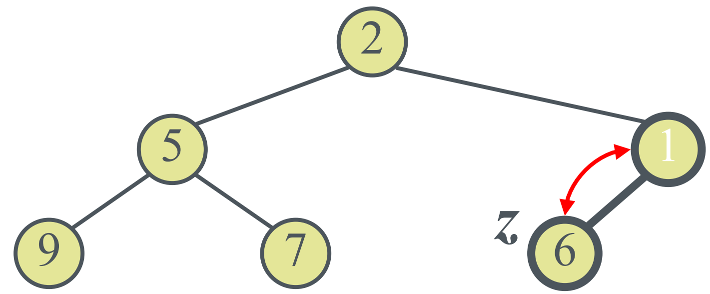
Insertion into a Heap

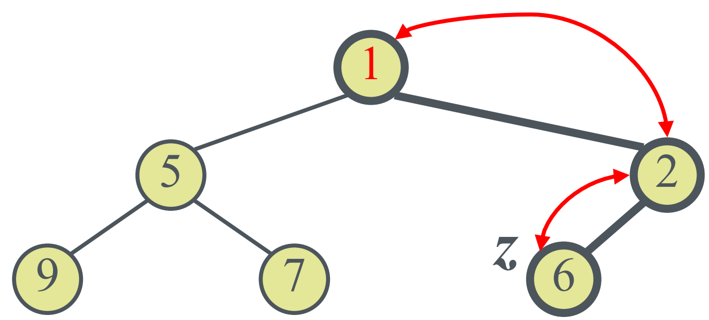
- Method insert of the priority queue ADT corresponds to the insertion of a key k to the heap
- The insertion algorithm consists of three steps
  - Find the insertion node z (the new last node)
  - Store k at z
  - Restore the heap-order property (discussed next)



Upheap

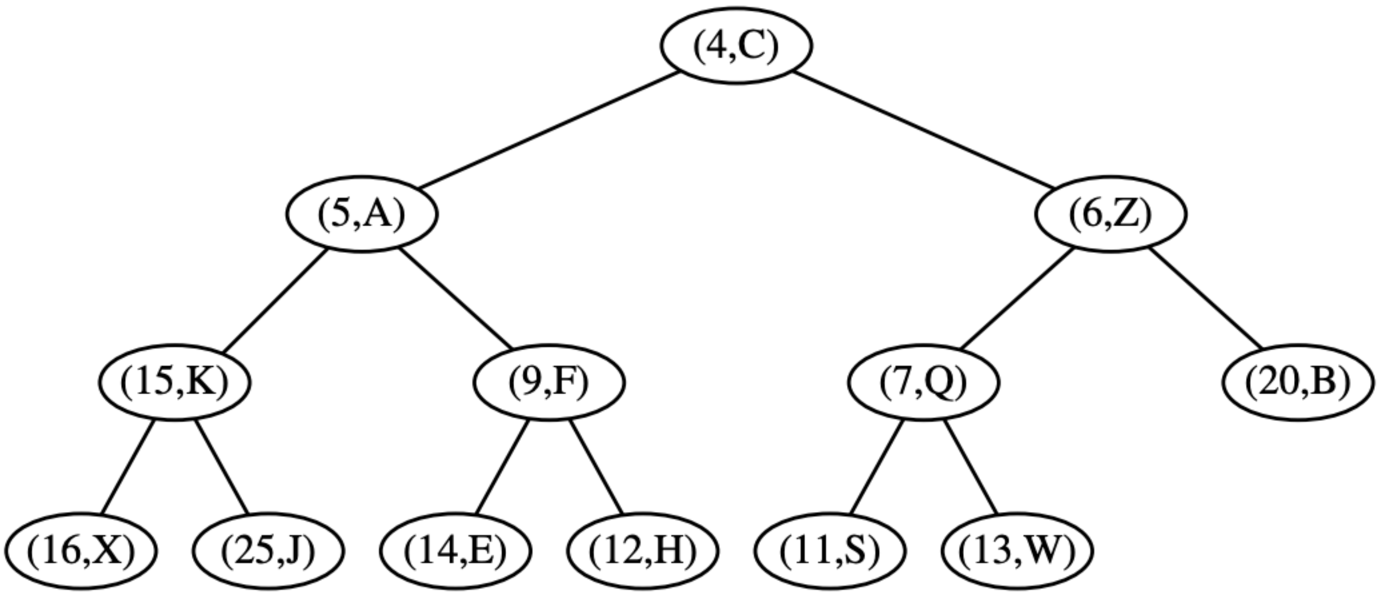
- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time





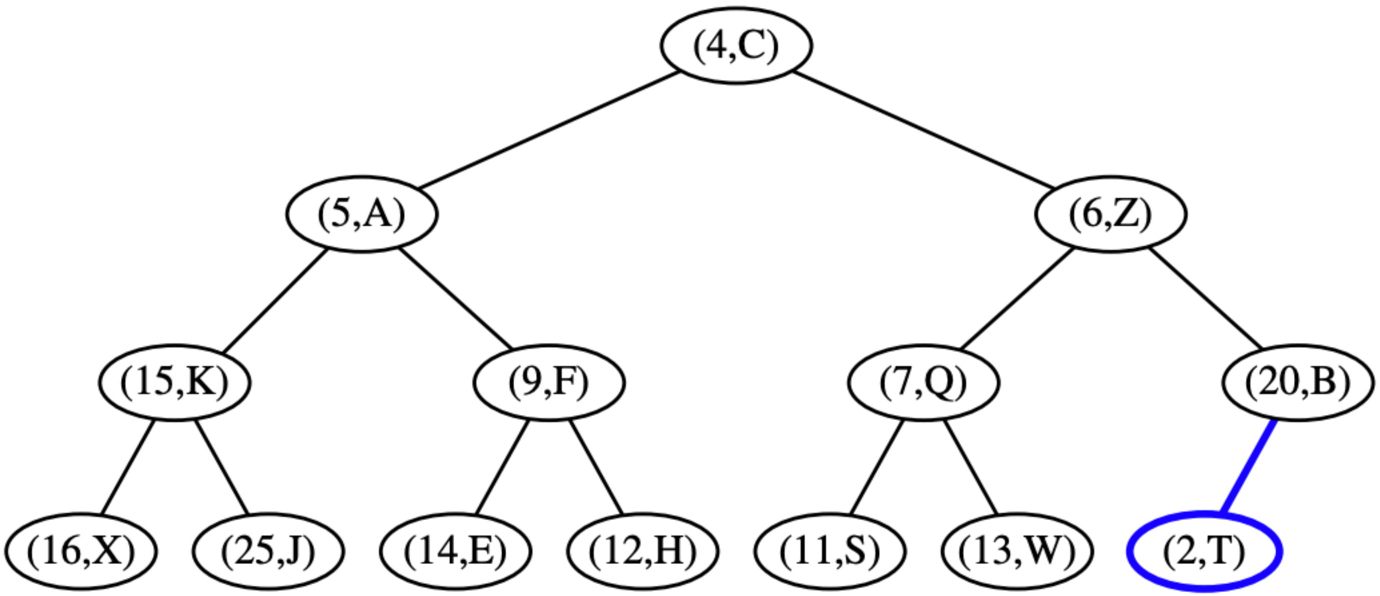
Upheap - Another Example

- insert (2, T)



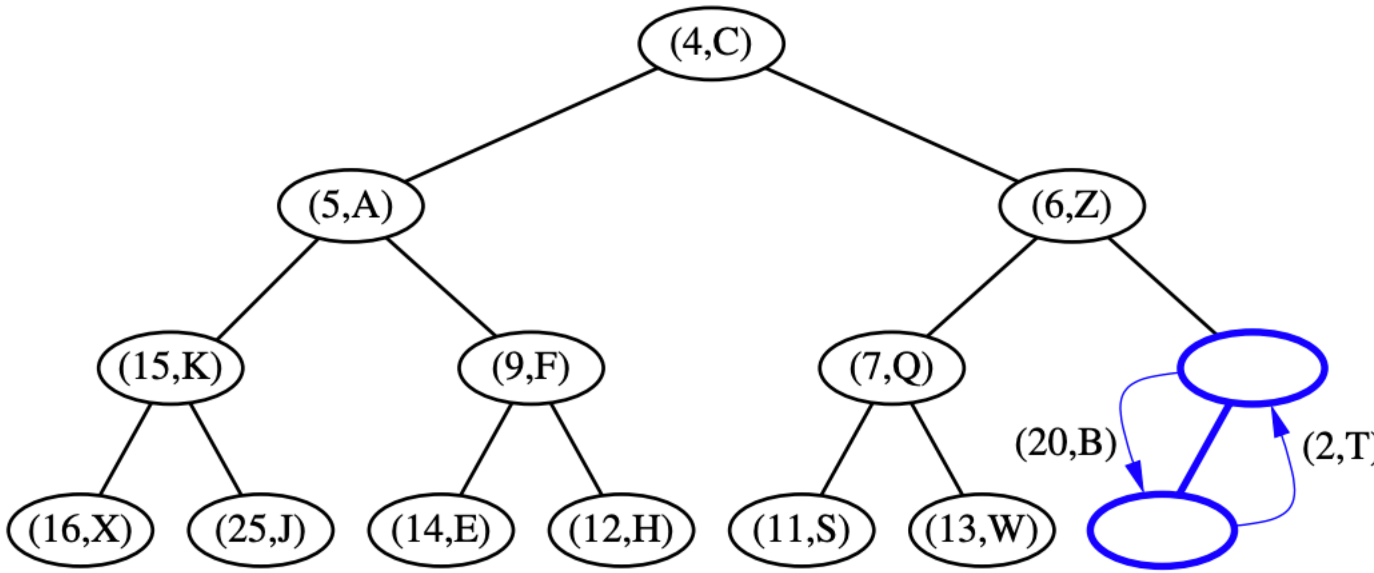
Upheap - Another Example

- insert (2, T)



Upheap - Another Example

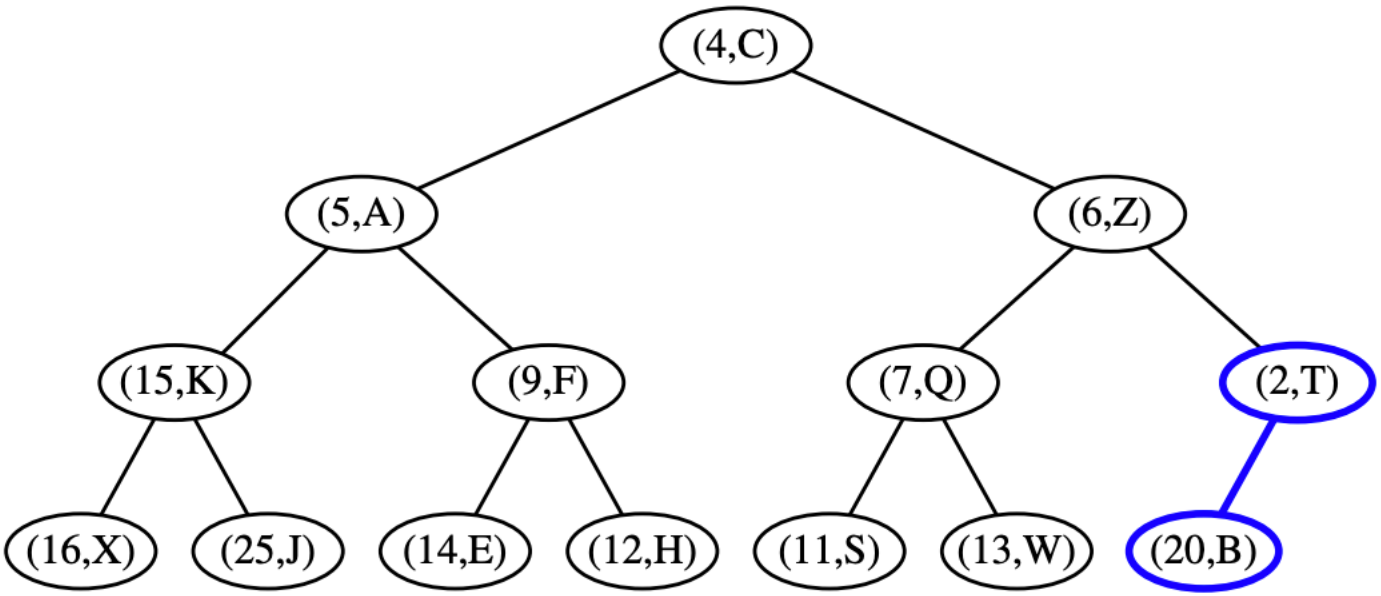
- insert (2, T)



McMaster University

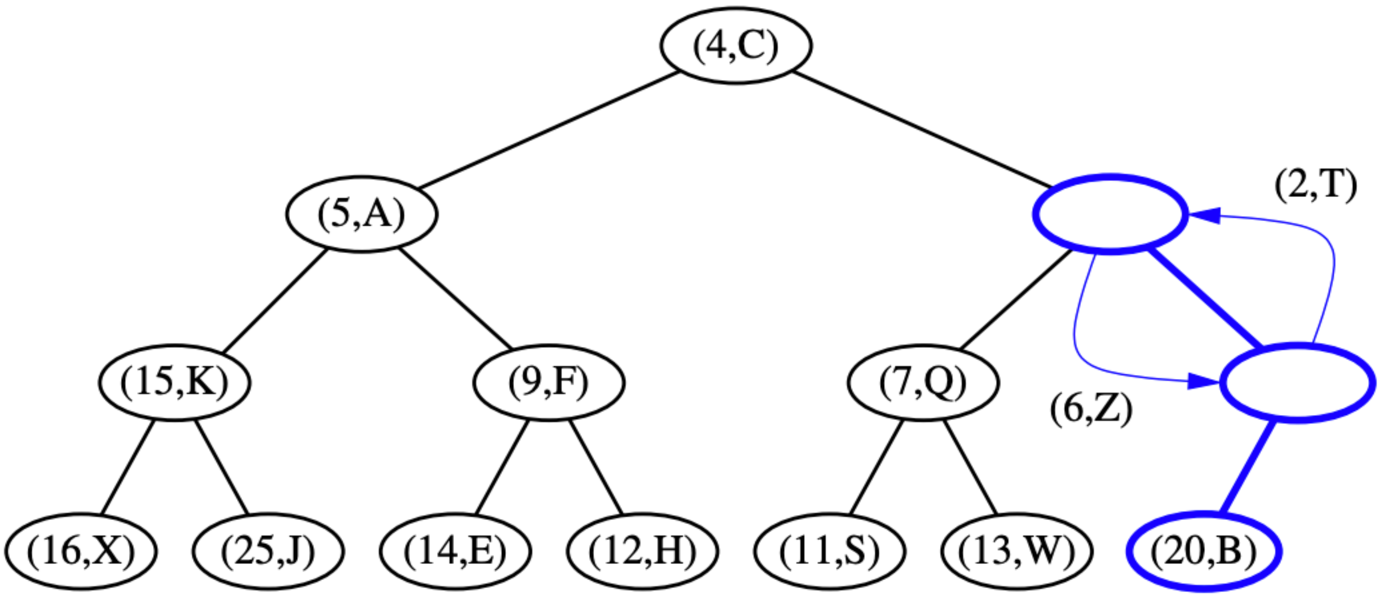
Upheap - Another Example

- insert (2, T)



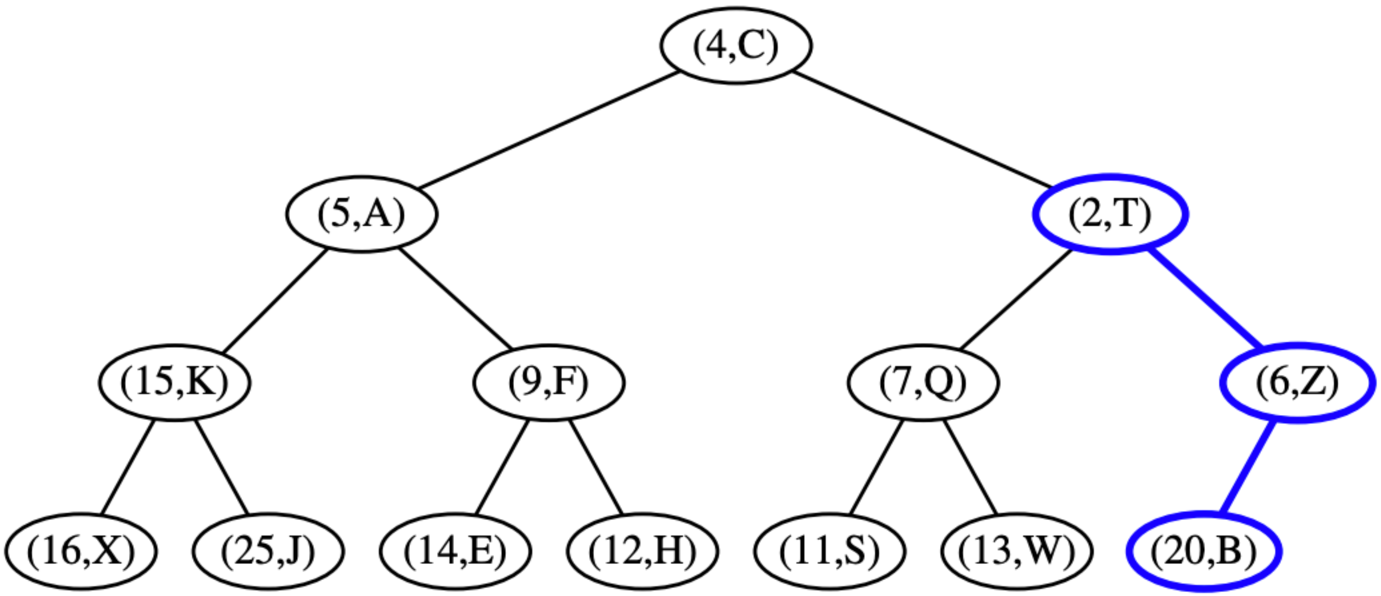
Upheap - Another Example

- insert (2, T)



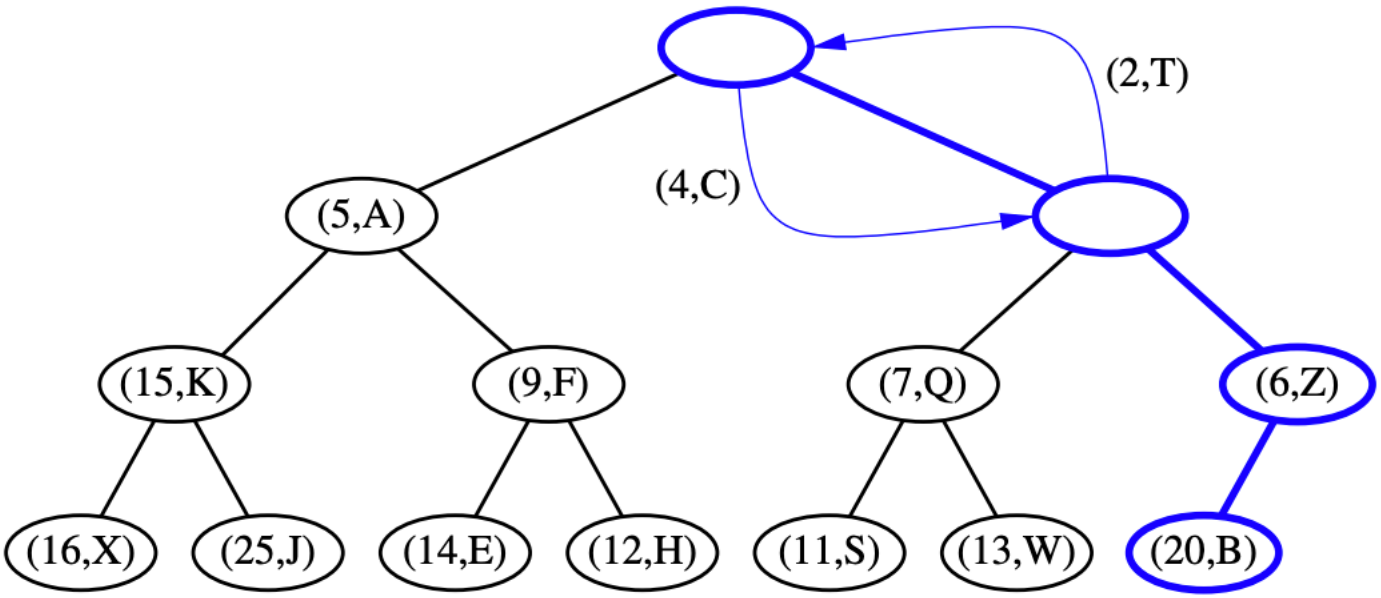
Upheap - Another Example

- insert (2, T)



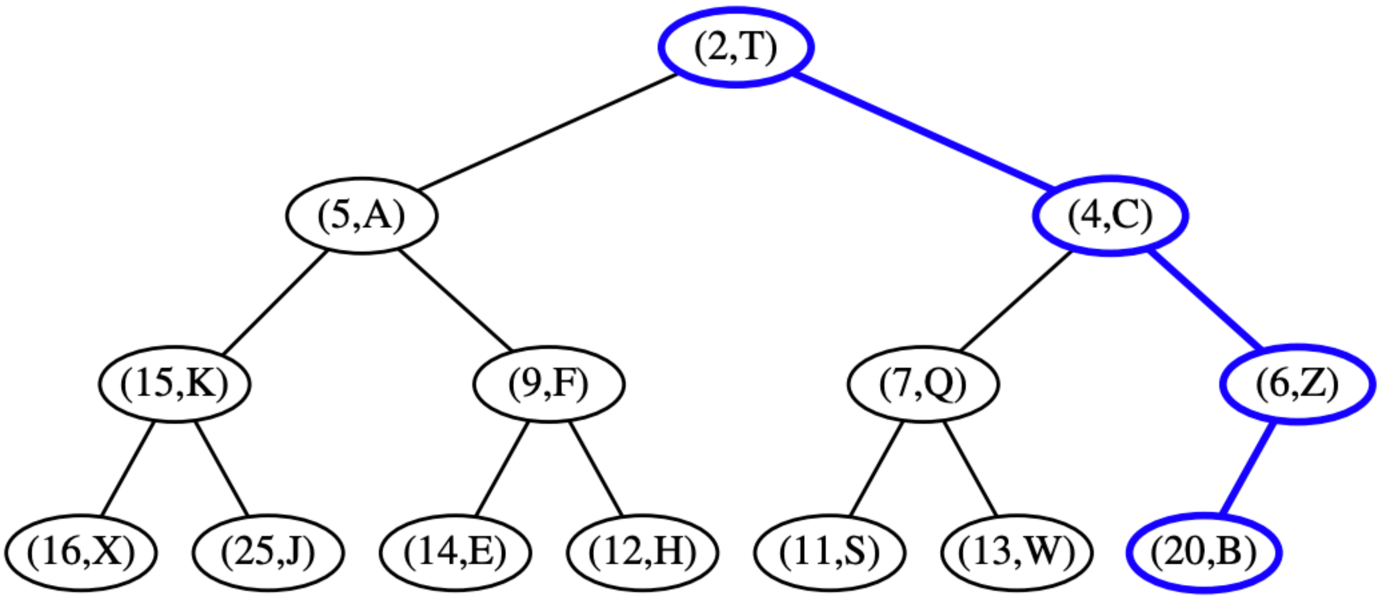
Upheap - Another Example

- insert (2, T)



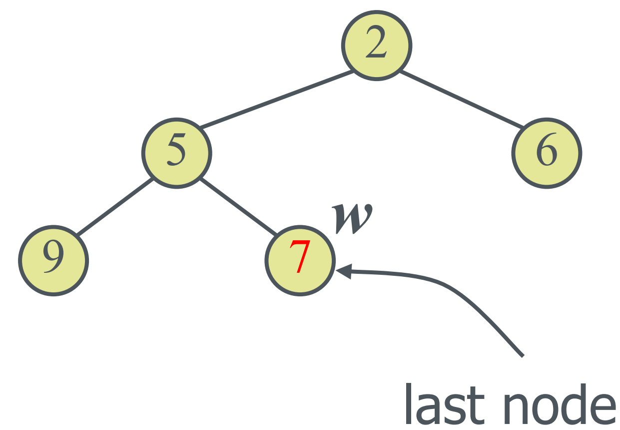
Upheap - Another Example

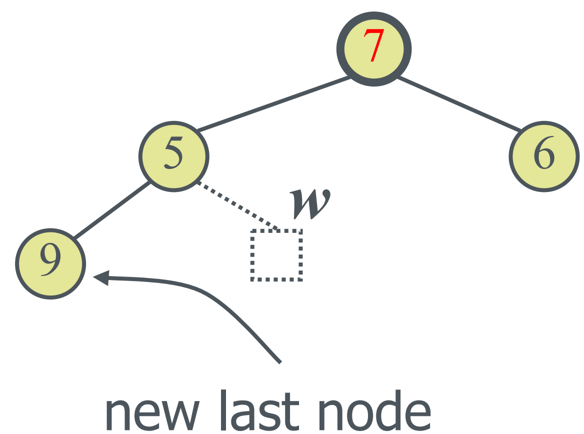
- insert (2, T)



Removal from a Heap

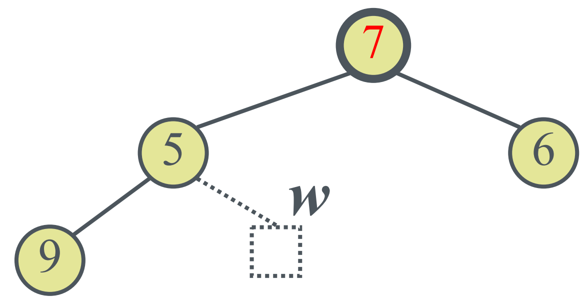
- Method removeMin of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node w
  - Remove w
  - Restore the heap-order property (discussed next)

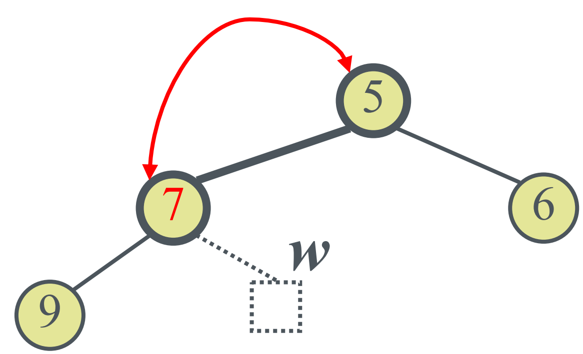




Downheap

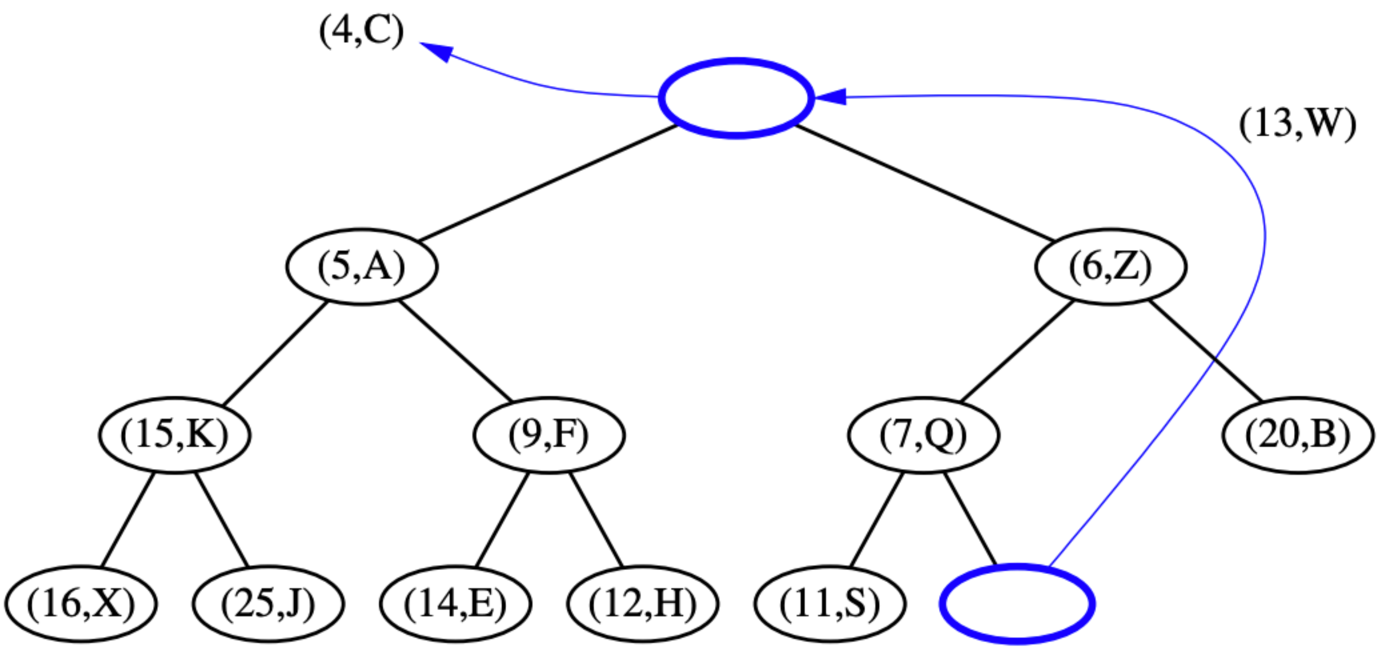
- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key k along a downward path from the root
- Upheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time





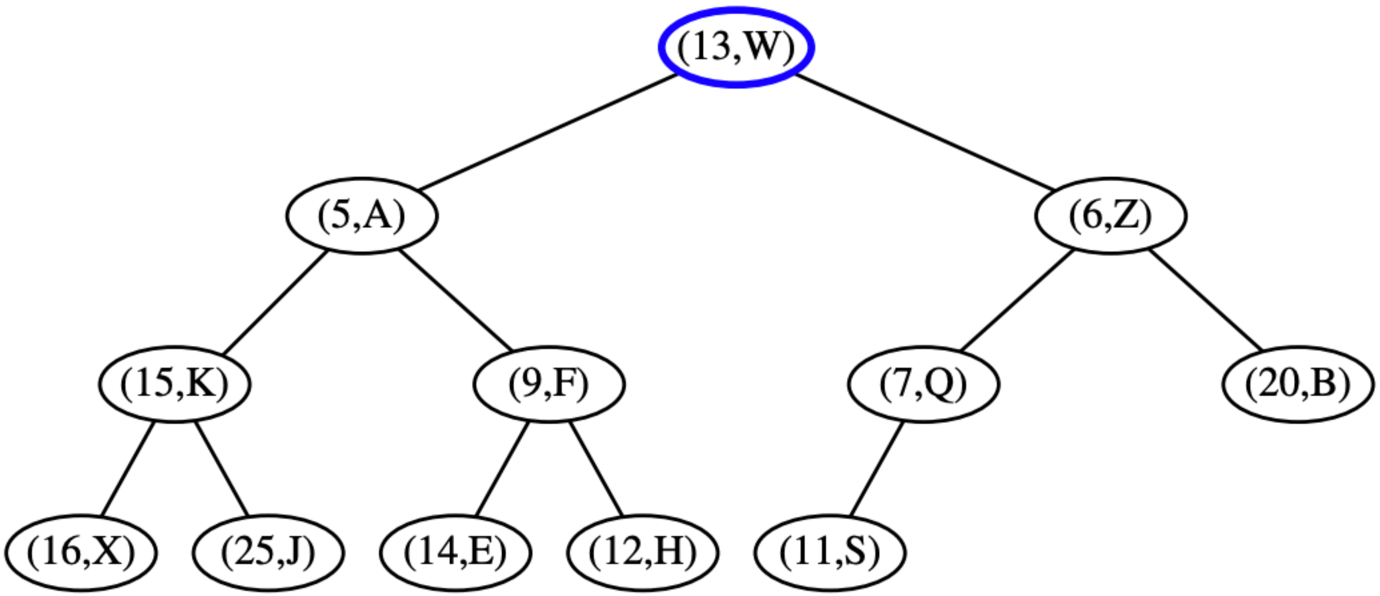
Downheap - Another Example

- removeMin



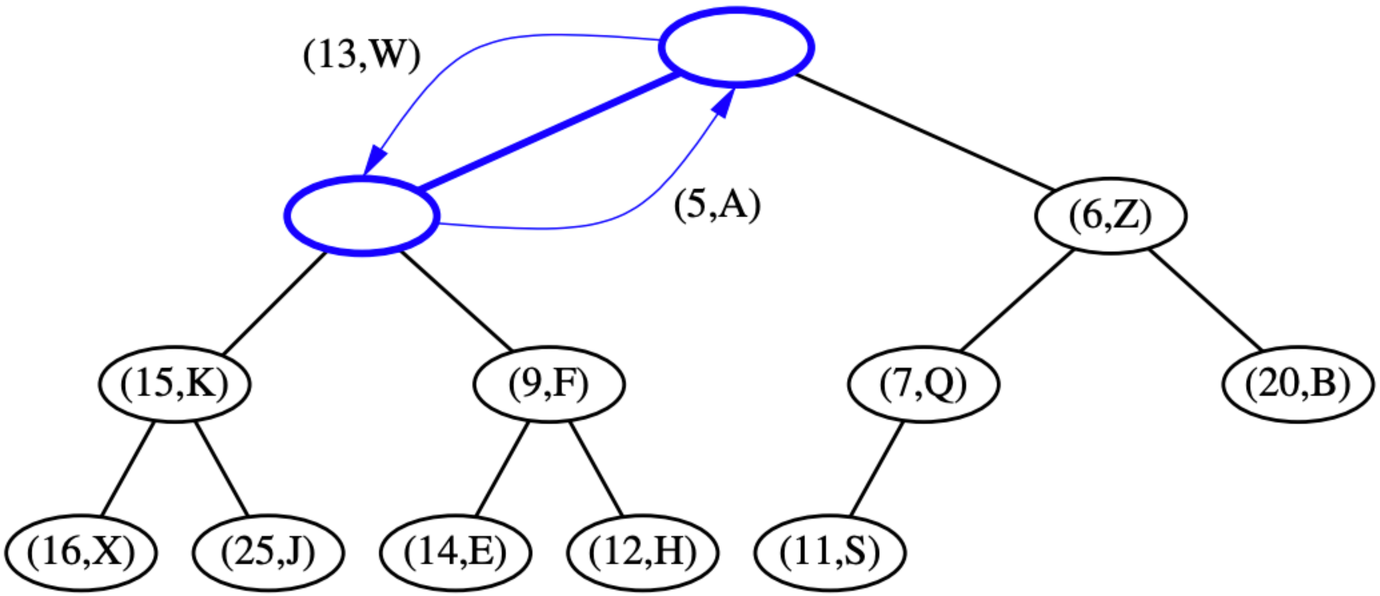
Downheap - Another Example

- removeMin



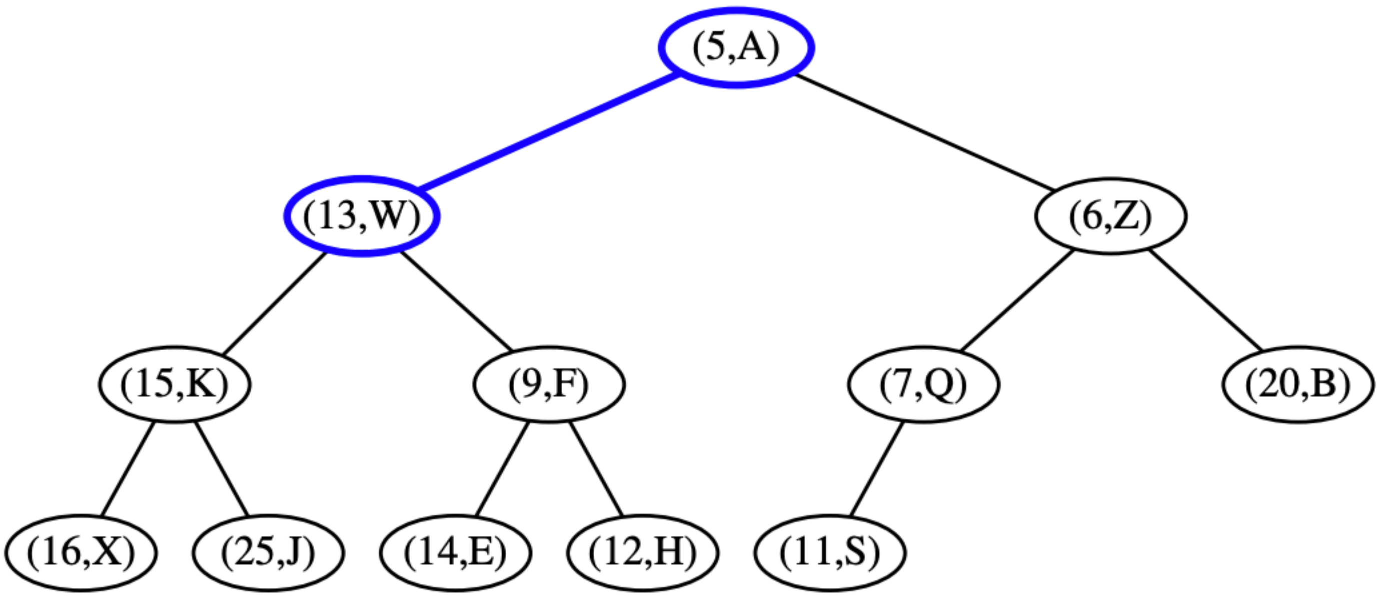
Downheap - Another Example

- removeMin



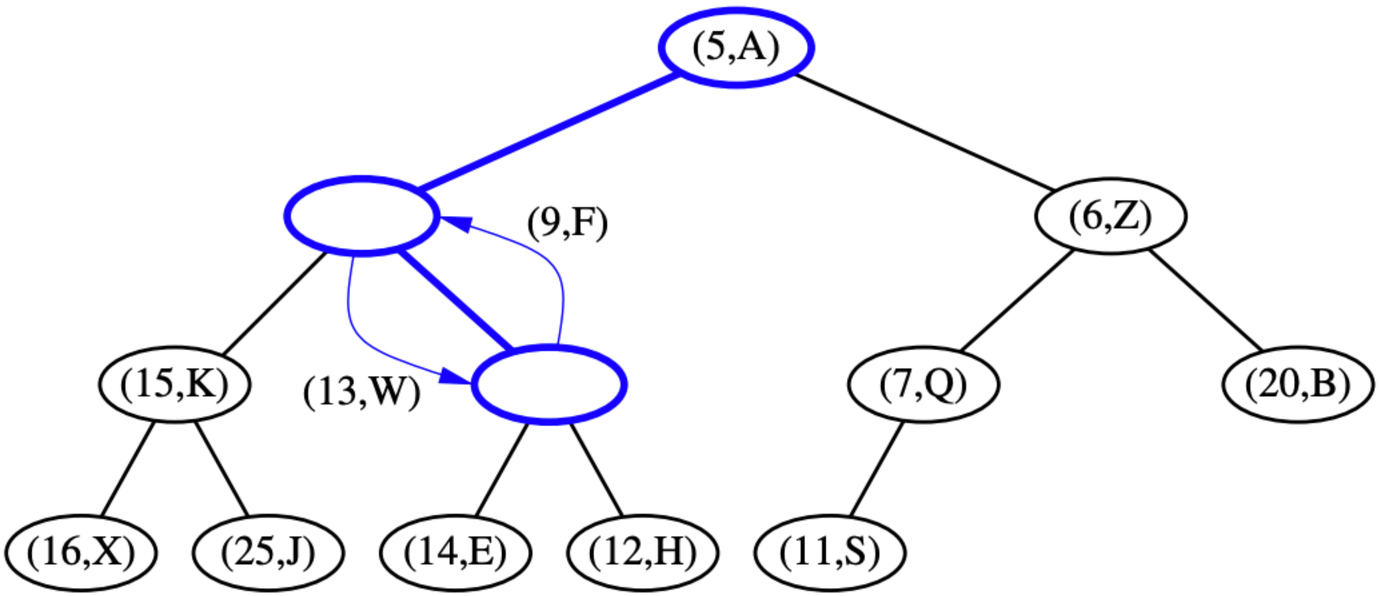
Downheap - Another Example

- removeMin



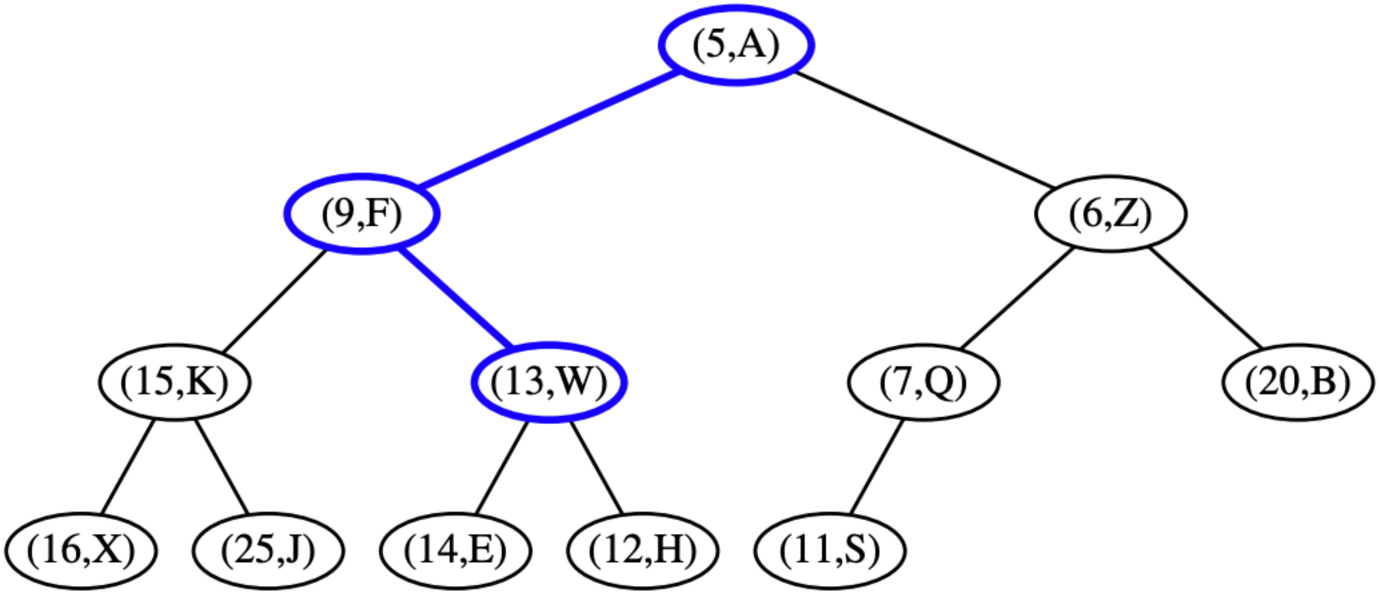
Downheap - Another Example

- removeMin



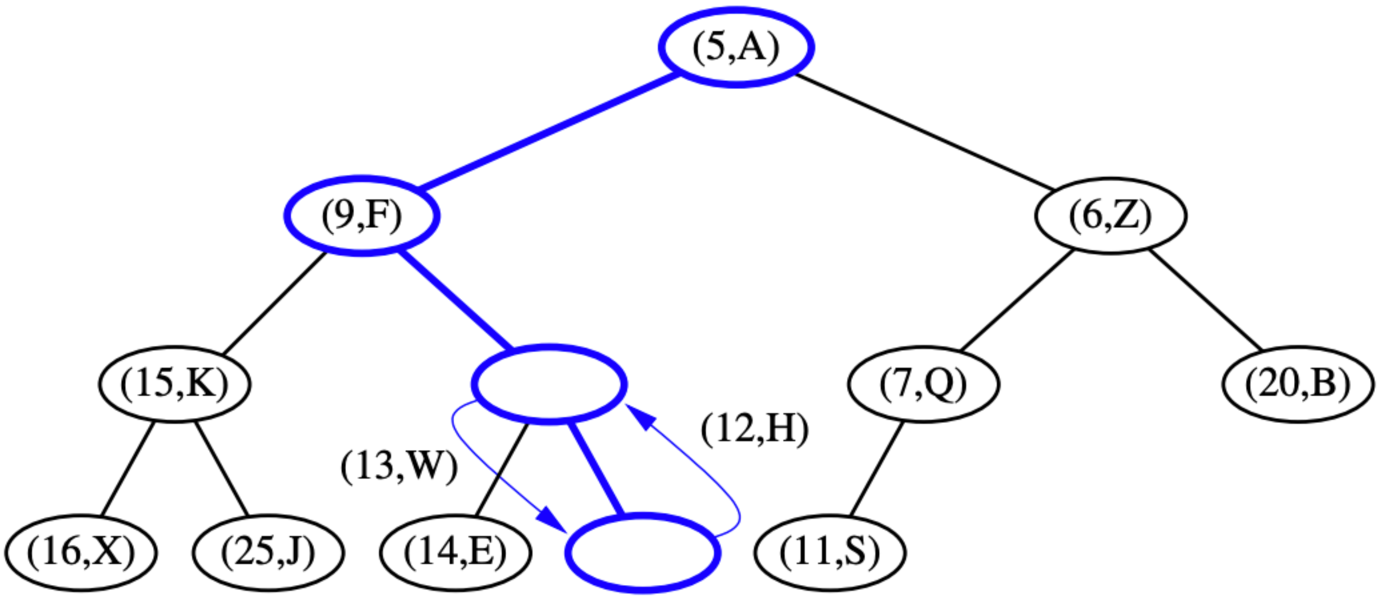
Downheap - Another Example

- removeMin



Downheap - Another Example

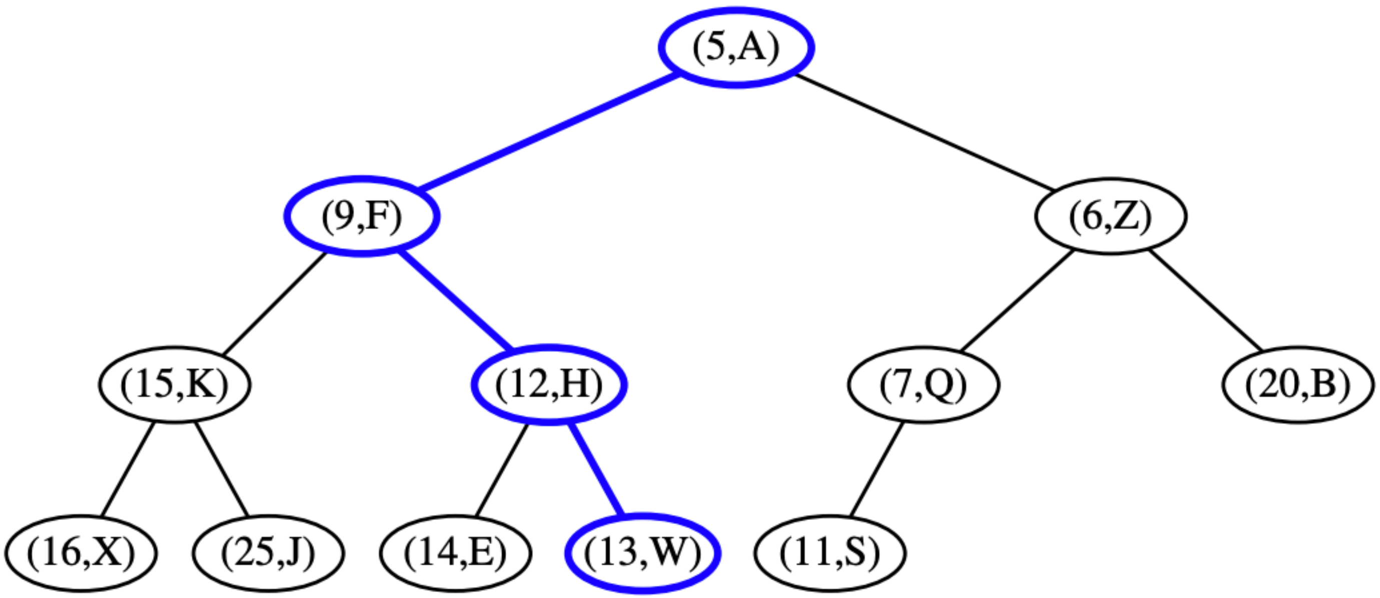
- removeMin



McMaster University

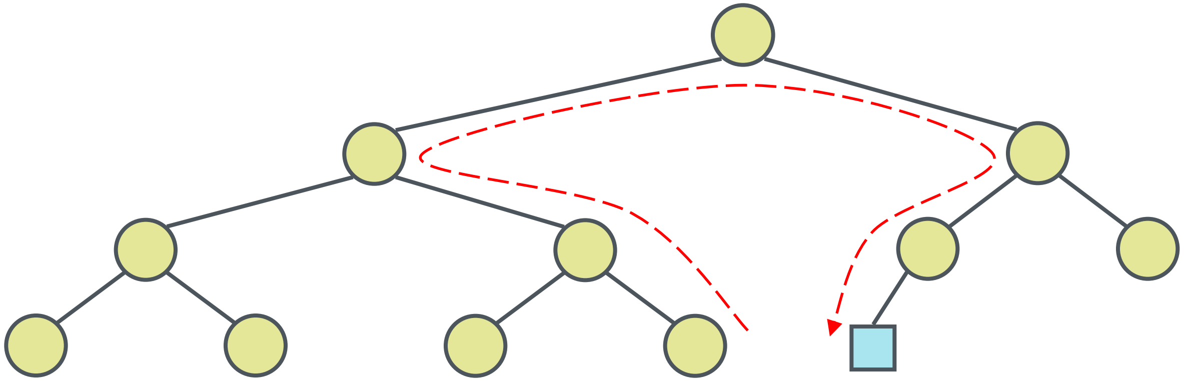
Downheap - Another Example

- removeMin



Updating the Last Node

- The insertion node can be found by traversing a path of $O(\log n)$ nodes
  - Go up until a left child or the root is reached
  - If a left child is reached, go to the right child
  - Go down left until a leaf is reached
- Similar algorithm for updating the last node after a removal



Priority Queue Sorting (Recall again!)

- 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 depends on the priority queue implementation:
  - **Unsorted sequence gives selection-sort: O(n^2) time**
  - **Sorted sequence gives insertion-sort: O(n^2) time**

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.removeMin***()

  S.insertBack(***e***)

Sorting using a PQ mimics two Sorting Algorithms

- **Selection sort** algorithm sorts an array by repeatedly finding the minimum element from unsorted part and putting it at the beginning.
- **Selection sort** is the variation of PQ-sort where the priority queue is implemented with an **unsorted sequence.**
- **Runs in $O(n^2)$ time.**

		List L	*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)	()

Sorting using a PQ mimics two Sorting Algorithms

- **Insertion sort** is the variation of PQ-sort where the priority queue is implemented with a **sorted sequence.**
- **Runs in O(n^2) time.**

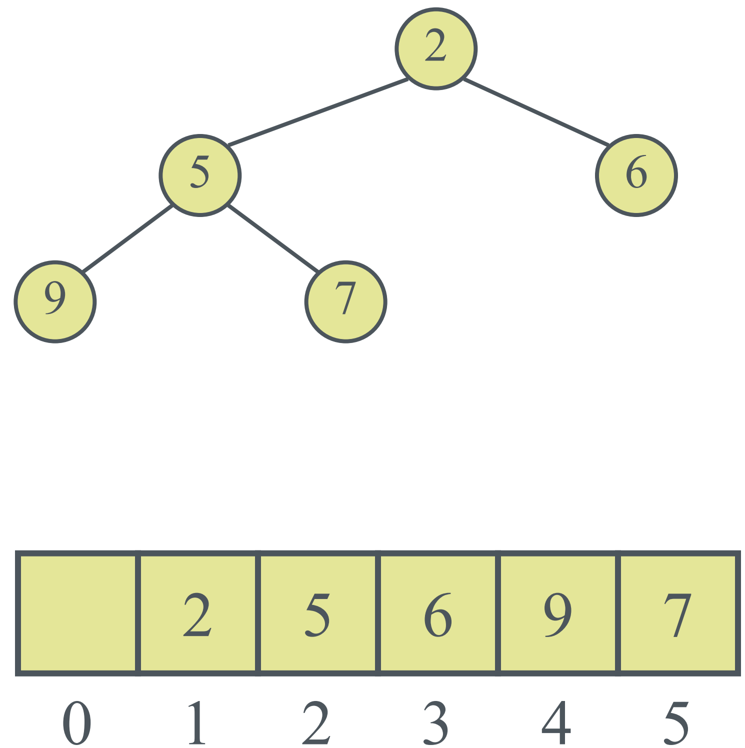
		List L	*Priority Queue P*
Input		(7,4,8,2,5,3,9)	()
Phase 1	(a)	(4,8,2,5,3,9)	(7)
	(b)	(8,2,5,3,9)	(4,7)
	(c)	(2,5,3,9)	(4,7,8)
	(d)	(5,3,9)	(2,4,7,8)
	(e)	(3,9)	(2,4,5,7,8)
	(f)	(9)	(2,3,4,5,7,8)
	(g)	()	(2,3,4,5,7,8,9)
Phase 2	(a)	(2)	(3,4,5,7,8,9)
	(b)	(2,3)	(4,5,7,8,9)
	⋮	⋮	⋮
	(g)	(2,3,4,5,7,8,9)	()

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.

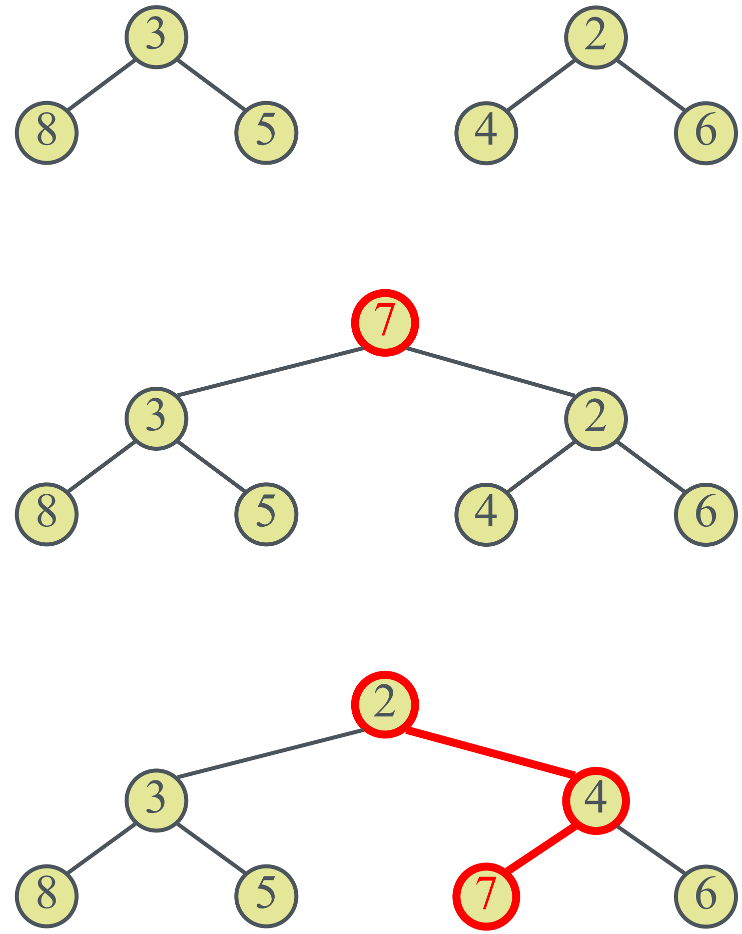
Heap: Array-based representation

- 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

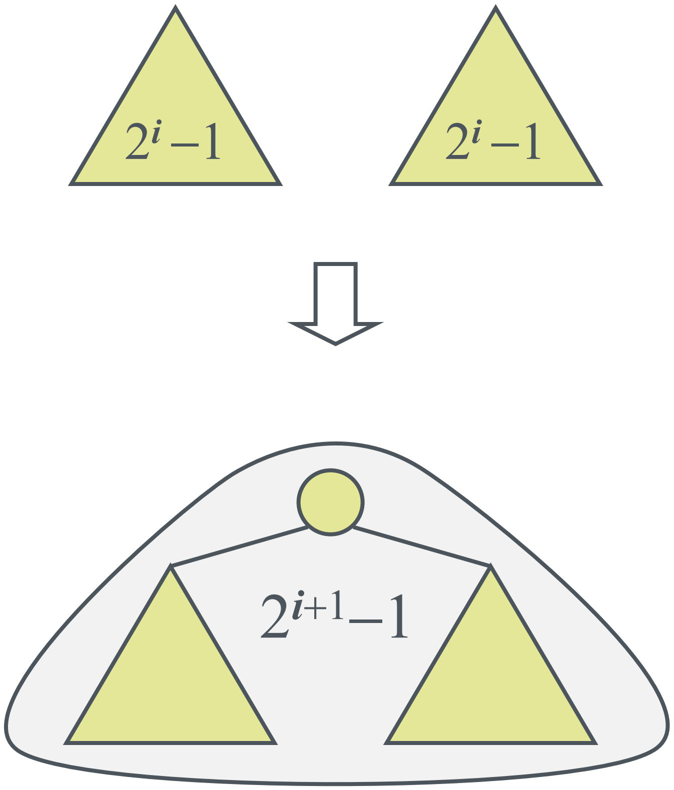
- We are given two two heaps and a key k
- We create a new heap with the root node storing k and with the two heaps as subtrees
- We perform downheap to restore the heap-order property



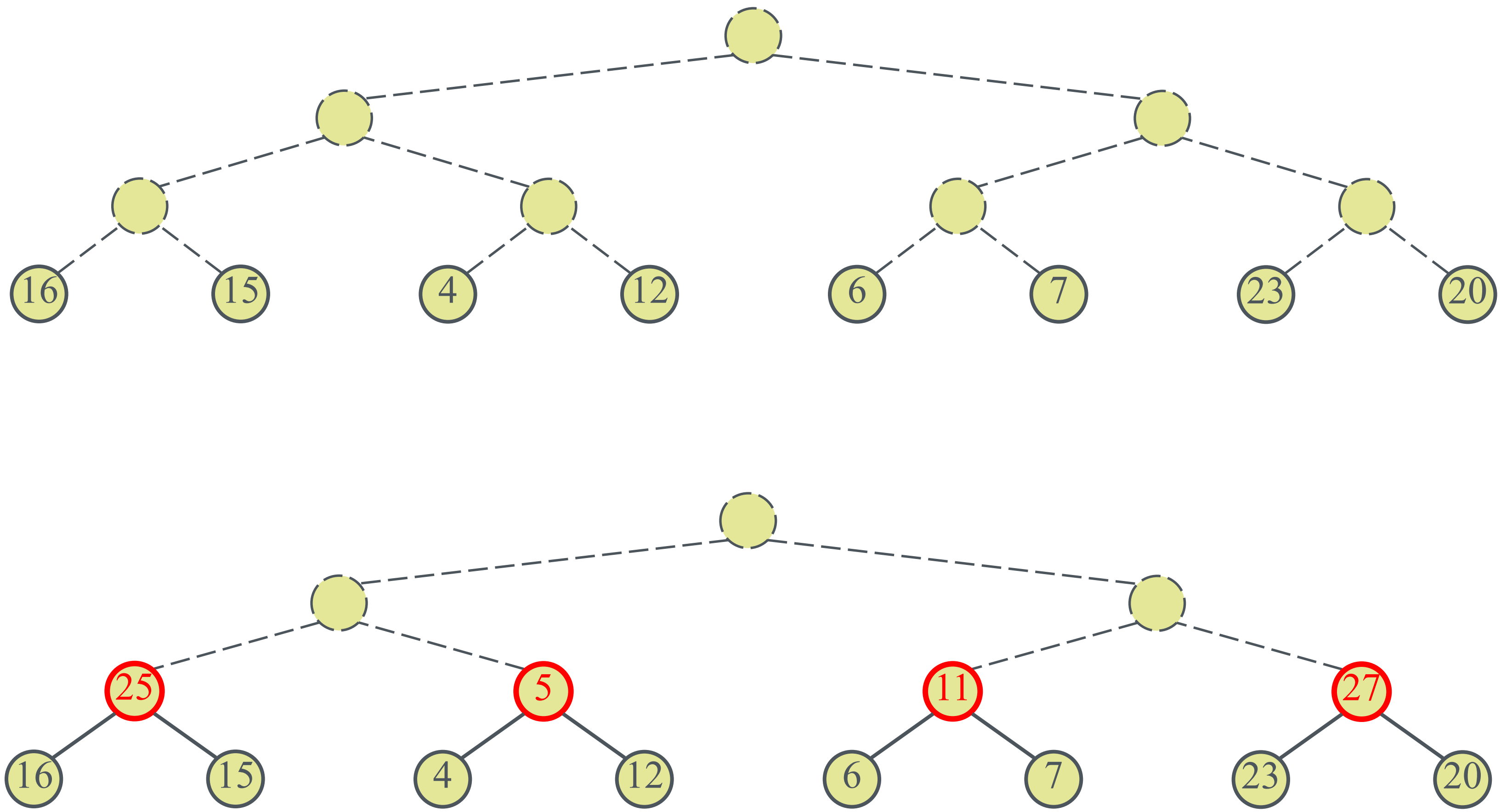
McMaster University

Bottom-up Heap Construction

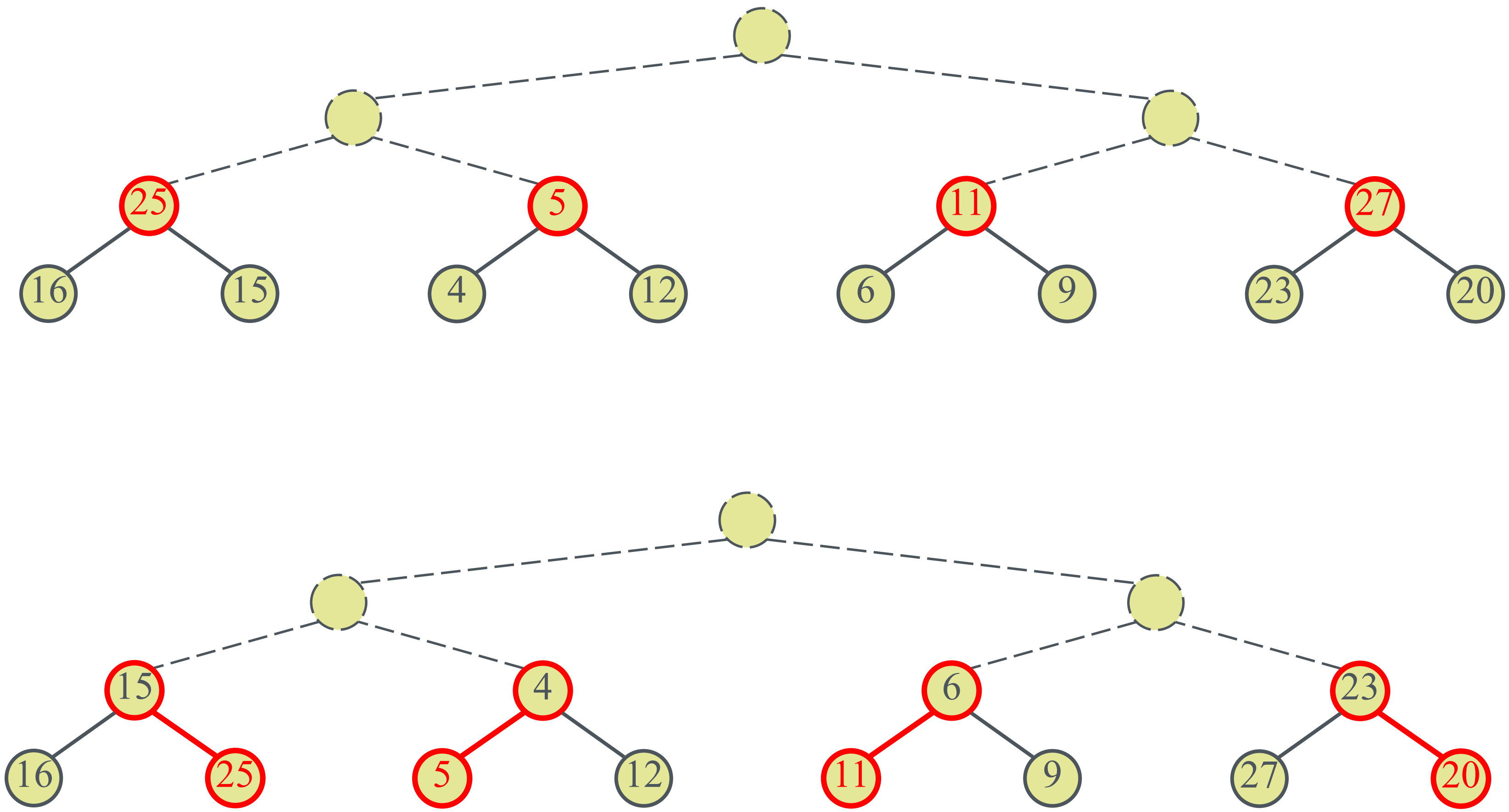
- We can construct a heap storing n given keys in using a bottom-up construction with $\log n$ phases
- In phase i, pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys
  - we construct (n+1)/2 elementary heaps storing one entry each.
  - we form (n + 1)/4 heaps, each storing three entries, by joining pairs of elementary heaps and adding a new entry.
  - …



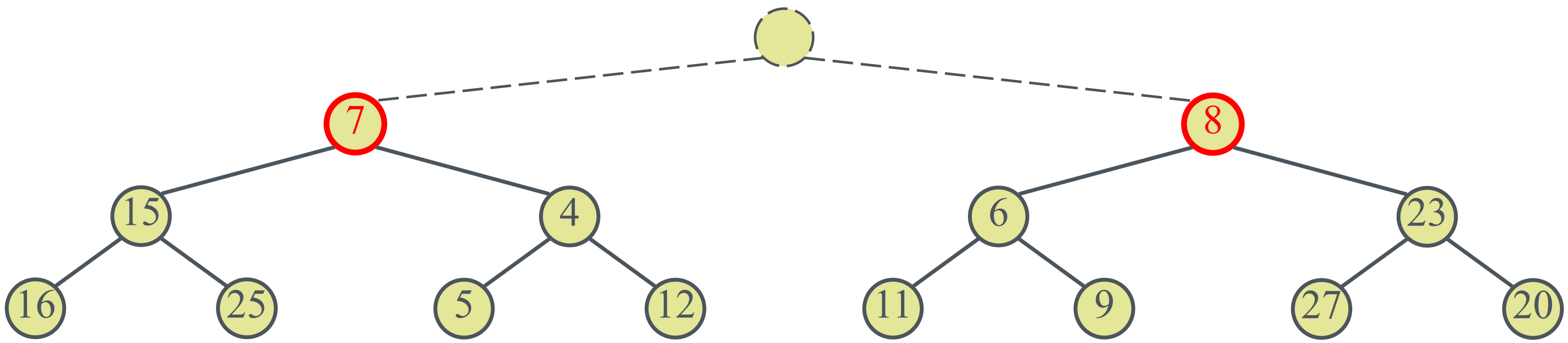
Bottom-up Heap Construction - Example

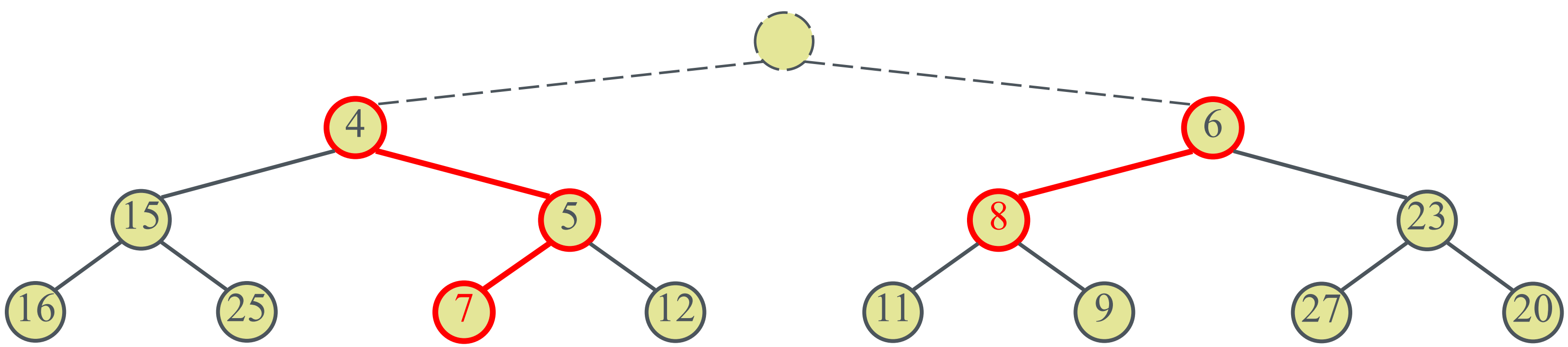


Bottom-up Heap Construction - Example

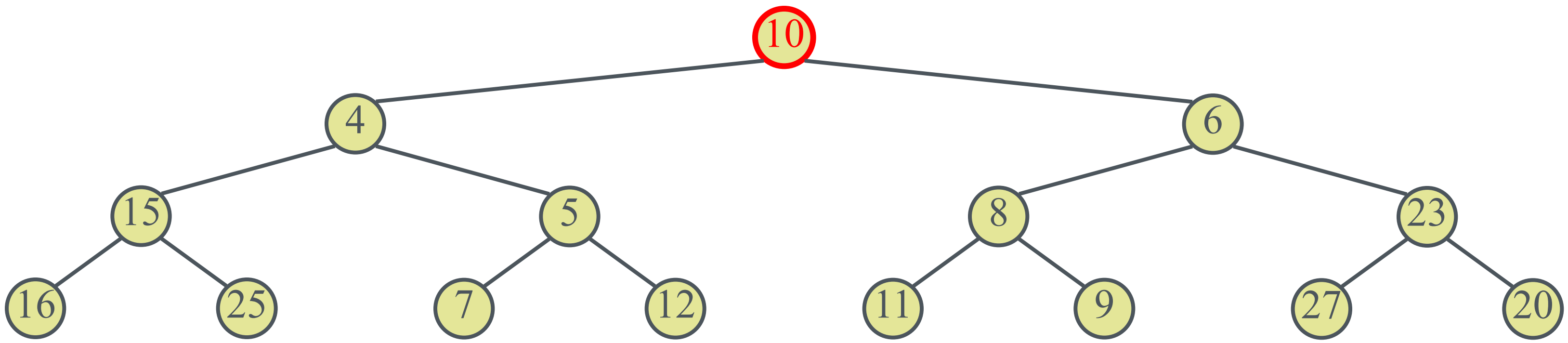


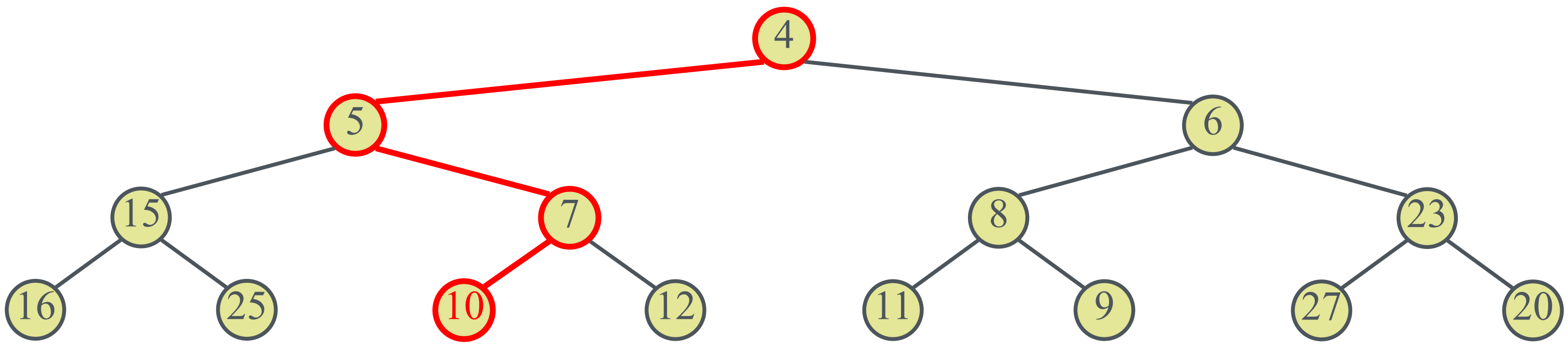
Bottom-up Heap Construction - Example





Bottom-up Heap Construction - Example





Questions?

Please evaluate this course!
Thank you