Priority Queues, Heaps, Heapsort

Textbook Reading:

Chapter 4, Section 4.6, pp.169-192.

Priority Queues, Heaps, Heapsort

- A priority queue has the same operations as a queue except the dequeue operation involves dequeuing the element have the highest priority.
- The highest priority can be the smallest or largest elements depending on the applications.
- In practice, the priority is a key in a record with multiple fields.

- Priority queues is an important ADT with myriad applications
- We will apply priority queues in algorithm design. In particular, when using the greedy method a priority queue can be used to obtain the next smallest or next largest element of the base set
- In algorithms like Prim's algorithm for finding a minimum spanning tree and Dijkstra's algorithm for finding shortest paths, we will use a hybrid version of a priority queue that allows us to change the priority of an element

Implementations of a priority queue

Three main implementation of a priority queue are:

- List
- Sorted List
- Heap

ADT	Enqueue	Dequeue	Change Priority
List	$O(1)$	$O(n)$	$O(1)$
Sorted List	$O(n)$	$O(1)$	$O(n)$
Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$

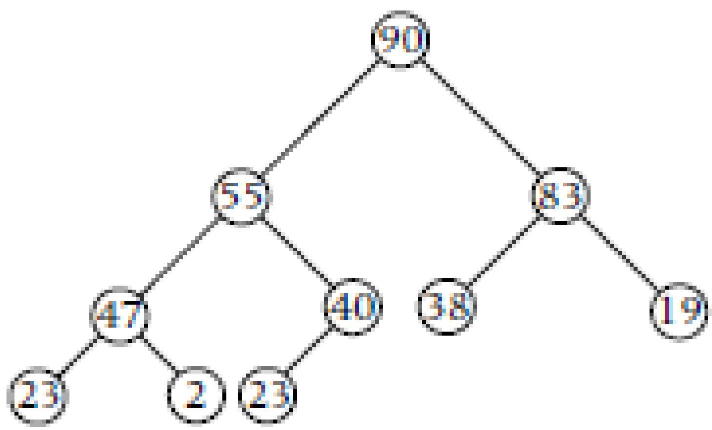
Heap – Max-Heap and Min-Heap

A **max-heap** is a complete binary tree where the key of any node is **greater than** or equal to the keys of its children.

A **min-heap** is a complete binary tree where the key of any node is **less than** or equal to the keys of its children.

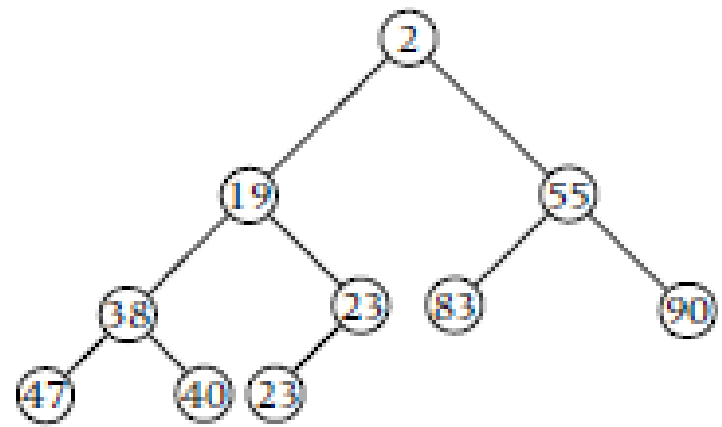
Sample Max-Heap and Min-Heap

Max-Heap



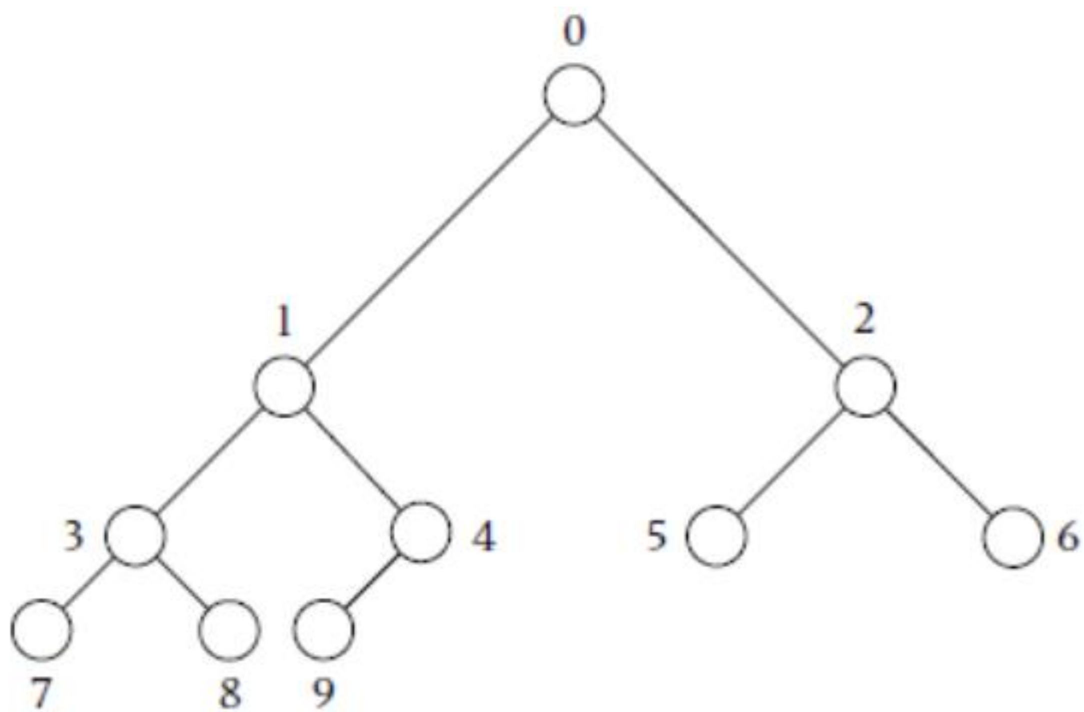
(a)

Min-Heap

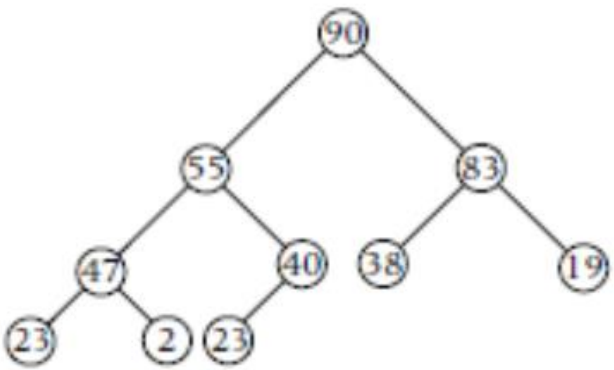


(b)

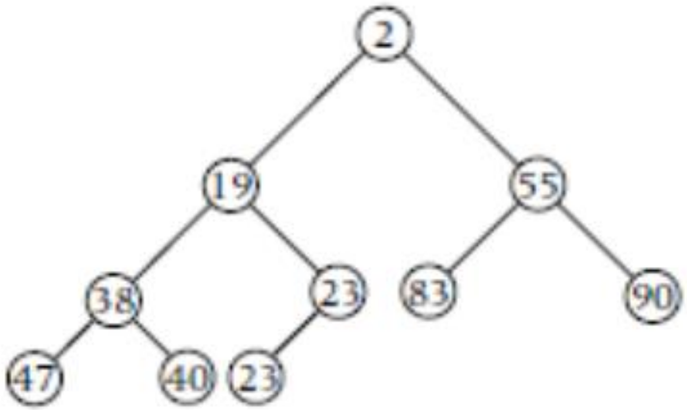
Association of array indices with nodes of Heap



Implementation of a Heap using an Array

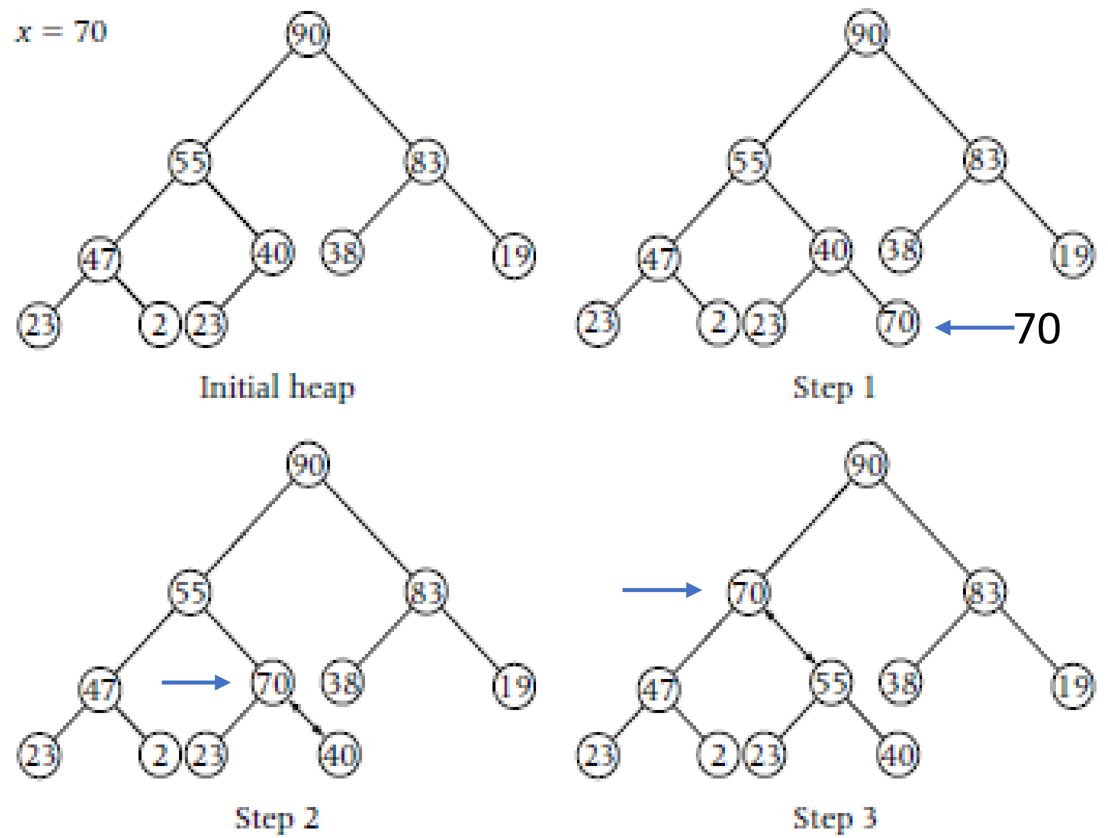


0	1	2	3	4	5	6	7	8	9
90	55	83	47	40	38	19	23	2	23



0	1	2	3	4	5	6	7	8	9
2	19	55	38	23	83	90	47	40	23

Insertion into Sample Max-Heap



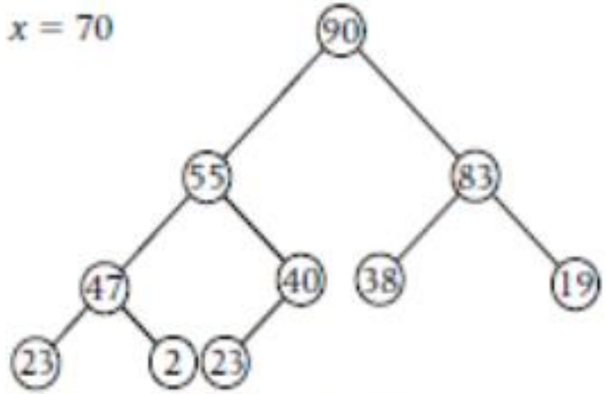
Fast Access of Parent and Children

Given the index i of a node in a complete tree:

Index of Parent: $(i - 1)/2$ (integer division)

Index of Children: $2*i + 1$ and $2*i + 2$

PSN. Show action of array for inserting 70 into sample max-heap from previous slide.



Initial heap

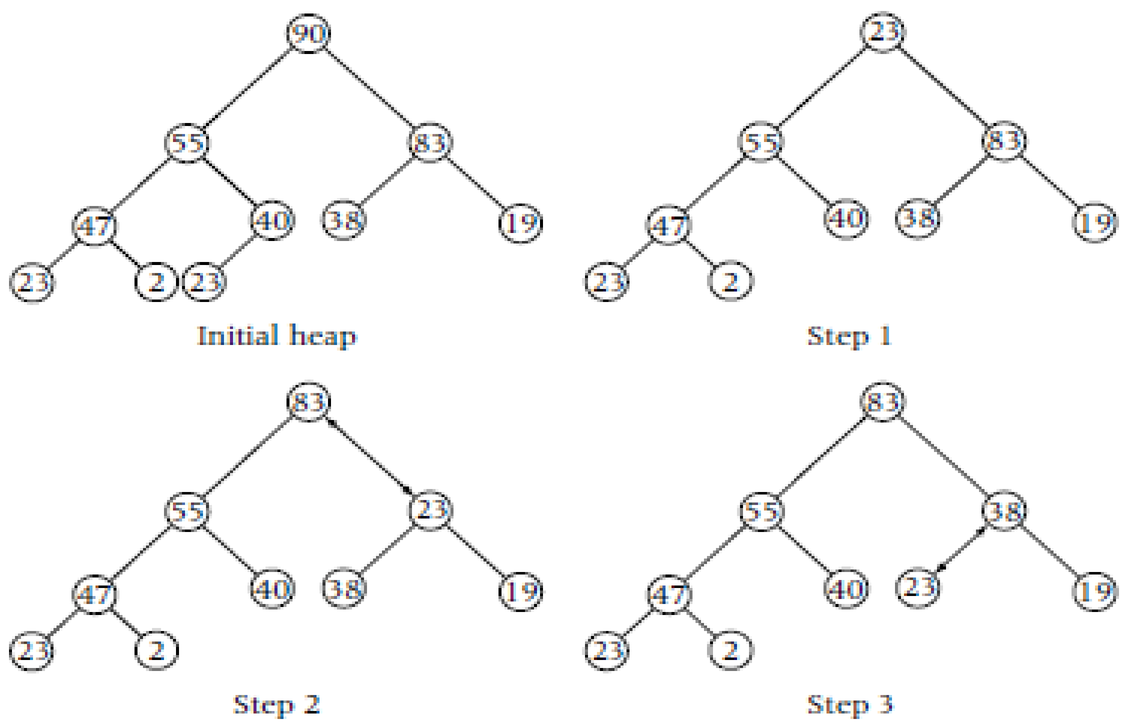
Pseudocode for Insert Operation

procedure *InsertMaxHeap*($A[0{:}m-1]$, *HeapSize*, x)
Input: $A[0{:}m-1]$ (an array containing a heap in positions 0, . . . , *HeapSize* – 1)
HeapSize (the number of elements in the heap)
x (a value to be inserted into heap)
Output: $A[0{:}m-1]$ (array altered by addition of x and heap maintained)
HeapSize (input *HeapSize* + 1)

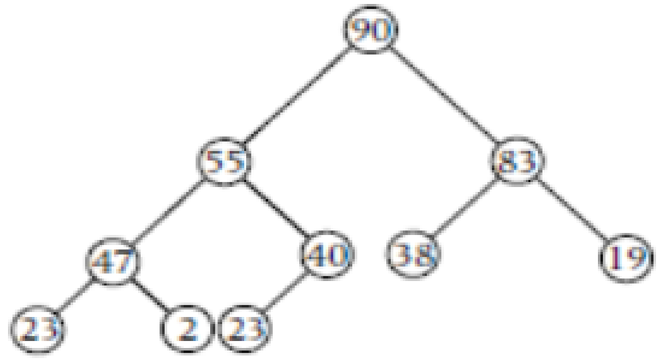
```
    i ← HeapSize
    j ← ⌊(i – 1)/2⌋
    //traverse path to root until proper position for inserting x is found
    while j ≥ 0 .and. A[j] < x do
        A[i] ← A[j]      //move parent down one position in path
        i ← j            //move x up one position in path
        j ← ⌊(j – 1)/2⌋
    endwhile
    //i is now a proper position for x
    A[i] ← x
    HeapSize ← HeapSize + 1
end InsertMaxHeap
```

Dequeue – Deletion from Sample Max-Heap

Move last element to root and adjust heap (if necessary) to correct violation of heap property at root.



PSN. Show action of deleting an element from a the sample max-heap from previous slide.



Psuedocode to Adjust Max-Heap for Violation of Heap Property at Root

```
procedure AdjustMaxHeap(A[0:m – 1],n,i)
Input: A[0:m – 1] (an array)
       n (consider A[0:n – 1] as complete binary tree, n ≤ m)
       i (index where subtrees of A[0:n – 1] rooted at 2i + 1and 2i + 2 are heaps)
Output: A[0:m – 1] (subtree of A[0:n – 1] rooted at A[i], adjusted so that it
                    becomes a heap)
       Temp ← A[i]
       //traverse down a path until a proper position for Temp = A[i] is found
       Found ← .false.        //Found signals when a proper position is found
       j ← 2*i + 1     //j is the path finder. At completion of loop,
                       //⌊(j –1)/2⌋ is a proper position for Temp = A[i]
       while j ≤ n .and. .not. Found do
           if j < n – 1.and. A[j] < A[j + 1] then   //then move to right child j + 1
               j ← j + 1                 //path finder updated to right child
           endif
           if Temp ≥ A[j] then
               Found ← .true.
           else
               A[⌊(j – 1)/2⌋] ← A[j]   //move larger child up one position in path
               j ← 2*j + 1  //move path finder to next possible position for Temp
           endif
       endwhile
       A[⌊(j – 1)/2⌋] ← Temp
end AdjustMaxHeap
```

Changing Priority

- In some applications of priority queues i.e., in Prim's and Dijkstra's algorithms, which we will see later for computing a minimum spanning tree and shortest path tree, it is necessary to change the priority of an element.
- After the priority (i.e., value) is changed, to restore the max-heap property there are two cases:
  - Priority was increased. Restore max-heap property using similar action to insertion.
  - Priority was decreased. Restore max-heap property using similar action to deletion, i.e., perform adjust operations at node where priority was changed.

Complexity Analysis

Since the depth of a complete tree is approximately $\log_2 n$, the operations of insertion, deletion and changing the priority for a max-heap (or min-heap) all can be done in time $O(\log n)$.

Heap Sort

Heap Sort is identical to Selection Sort, except instead of using function Max() to find the index of the largest element, a max-heap is used.

1. Create a max-heap for array $L[0{:}n-1]$ of list elements. This can be done using *MaxMinHeap1*, which simply performs a sequence of insertions, in time $O(n \log n)$. It can be done in time $O(n)$ using *MaxMinHeap2* by performing a sequence of adjust operations.

2. Swap $L[0]$ and $L[n-1]$. In a max-heap the largest element is in position $L[0]$. Adjust the heap at node 0.

3. Repeat Step 2 for sublist $L[0{:}i]$, i, $n-2$, $n-3$, ..., 2.

Pseudocode for Heap Sort

procedure *HeapSort*(A[0:n – 1])
Input: A[0:n – 1] (a list of size n)
Output: A[0:n – 1] sorted in nondecreasing order
    MakeMaxHeap2(A[0:n – 1],n)
    for $i \leftarrow 1$ **to** $n - 1$ **do**
        //interchange A[0] with A[n – i]}
        interchange (A[0],A[n – i])
        AdjustMaxHeap(A[0:n – 1],$n - i - 1$,0)
    endfor
end *HeapSort*

procedure *MakeMaxHeap2*(A[0:m – 1],n)
Input: A[0:m – 1] (an array)
    n (values in A[0:n – 1] considered as complete binary tree, $n \leq m$)
Output: A[0:m – 1] (A[0:n – 1] made into a heap)
    for $i \leftarrow \lfloor (n-1)/2 \rfloor$ **down to** 0 **do**
        AdjustMaxHeap(A[0:m – 1],n,i)
    endfor
end *MakeMaxHeap2*

Complexity Analysis of Heapsort

- Creating a heap takes time $O(n)$ using *MaxMinHeap2* or $O(n \log n)$ using *MaxMinHeap1*.
- Each call to *AdjustMaxHeap* takes time $O(n \log n)$.
- $n - 1$ calls are made to *AdjustMaxHeap.*
- Total is $O(n \log n)$.

What's a horse's top priority when voting?

A stable economy.