DATA STRUCTURES

Heap

By
Zainab Malik

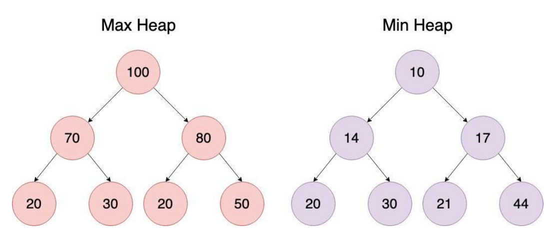
Content

Introduction to Heap

- A heap is binary tree that satisfies the following properties
  - **Shape property:** Heap must be a complete binary tree
  - **Order property:** It must be either Max heap or Min heap

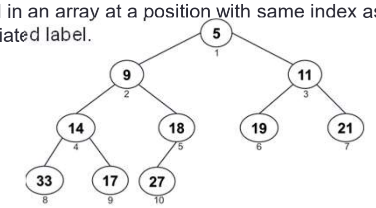
  - Max heap
    - For every node in the heap, the value stored in that node is greater than or equal to the value in each of its children
  - Min heap
    - For every node in the heap, the value stored in that node is less than or equal to the value in each of its children

Introduction to Heap: Max Heap vs. Min Heap



Heap Representation

- Heap is a Complete Binary Tree. This property of Binary Heap makes it suitable to be stored in a linear array.
- Each node is assigned a numeric label and a node is stored in an array at a position with same index as its associated label.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
5	9	11	14	18	19	21	33	17	27					

Operations on Heaps

- On heaps, only two operations are performed
  - Insertion
  - Deletion

Heap Operation: Insertion(item)

- Item is always inserted as last (bottom) child of the original heap.
- After insertion, shape property remain undisturbed, but the order may get violated if a larger item (incase of Max Heap) or smaller item (incase of Min Heap) is inserted.
- To satisfy the order property, heap needs to be readjusted in terms of its structure (reheapifyUpward)

- ReheapifyUpward:
  - It involves moving the items up from the last (bottom) position until either it ends up in a position where the order property satisfied or it hits the root node.

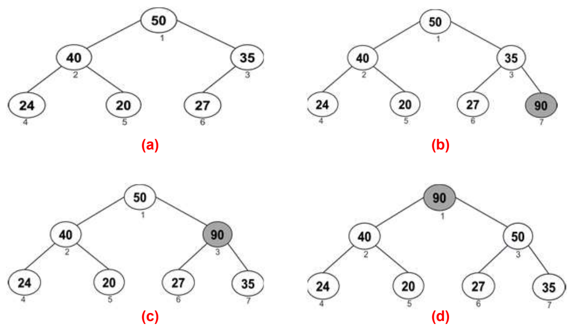
Heap Operation: Insertion(item)

- Insert(item,n,heap):
  1. Set n=n+1
  2. Set heap[n]=item
  3. Call reheapifyUpward(heap n)
  4. Return

- ReheapifyUpward(heap, start)

```
1.  If heap[start] is not a root node then
2.       If(heap[parent]<=heap[start]) then
3.             Set index = index of the child with largest value
4.             Swap heap[parent] and heap[index]
5.             Call repheapifyUpward(heap, parent)
         Endif
    Endif
5.  Return
```

Heap Operation: Insertion(item=90)



Heap Operation: Deletion()

- Element is always deleted from the root of the heap
  - When the element is deleted from the root, it creates a vacant space in the root position
  - As heap must be a complete binary tree, so the vacant space is filled by last (bottom right) element of the heap
  - Like insertion, this replacement ensures the shape property but disturbs the order property that needs to be satisfied by mean of reheapify (reheapifyDownward).
- reheapifyDownward:
  - It involves moving the element down from the root position until either it ends up in a position where order property is satisfied or it hits the leaf node.

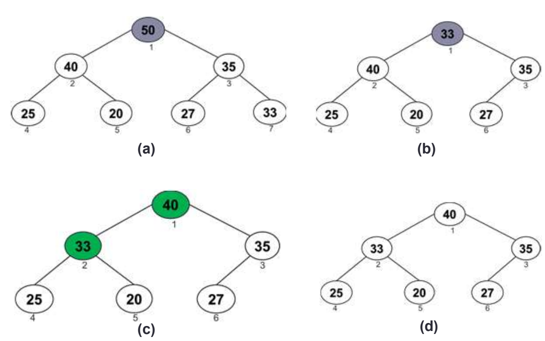
Heap Operation: Deletion()

- Delete(n,heap):
  1. Set item=heap[root]
  2. Set heap[root]=heap[n]
  3. Set n=n-1
  4. Call reheapifyDownward(heap,root)
  5. Return item

- ReheapifyDownward(heap, start)

```
1. If heap[start] is not a leaf node then
2. Set index = index of the child with largest value
3.    If(heap[start]<=heap[index]) then
4.          Swap heap[index] and heap[start]
5.          Call repheapifyDownward(heap, index)
      Endif
   Endif
6. Return
```


Heap Operation: Deletion()



Applications of Heap

- Priority Queue
- Heap Sort

Priority Queue using Heap

- Each Node in a heap have two types of information i.e. the content and an associated priority
- Heap is build with respect to priority which means that the element with highest priority will be at root node.
- As in heap we always delete from the root therefore, whenever a node will be removed for processing it will be of highest priority

Priority Heap Operation: Insertion(item)

- Insert(item,n,heap)://item must be an object of Element containing both content and priority
  1. Set n=n+1
  2. Set heap[n]=item
  3. Call reheapifyUpward(heap n)
  4. Return

- ReheapifyUpward(heap, start)

```
1. If heap[start] is not a root node then
2.      If(heap[parent].priority<=heap[start].priority) then
3.           Set index = index of the child with largest priority value
4.           Swap heap[parent] and heap[index]
5.           Call repheapifyUpward(heap, parent)
        Endif
   Endif
5. Return
```


Priority Heap Operation: Deletion()

- Delete(n,heap):
  1. Set item=heap[root]
  2. Set heap[root]=heap[n]
  3. Set n=n-1
  4. Call reheapifyDownward(heap,root)
  5. Return item

- ReheapifyDownward(heap, start)

```
1. If heap[start] is not a leaf node then
2. Set index = index of the child with largest priority value
3.      If(heap[start].priority<=heap[index].priority) then
4.           Swap heap[index] and heap[start]
5.           Call repheapifyDownward(heap, index)
        Endif
   Endif
6. Return
```

Applications of Heap

- Priority Queue
- Heap Sort

Heap Sort

- **HeapSort(a,n)** //a is a linear array and n is the last element of a

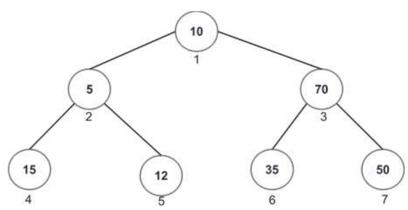
1. Call Heapify(a, n)
2. Repeat Step 3 and 4 For i=n to 1 in steps of -1
3. Swap elements a[1] with a[i]
4. Call ReheapifyDownward(a,1) (see slide # 11)
5. Endfor
6. Return

Heapify(a,n)

1. Set index=Parent of node with index n
2. Repeat step 3 For i=index to 1 in setp of -1
3. Call reheapifyDownward(a,i) (see slide # 8)
4. Endif
5. Return

Heap Sort

Unsorted Array						
1	2	3	4	5	6	7
10	5	70	15	12	35	50



Equivalent Binary Tree

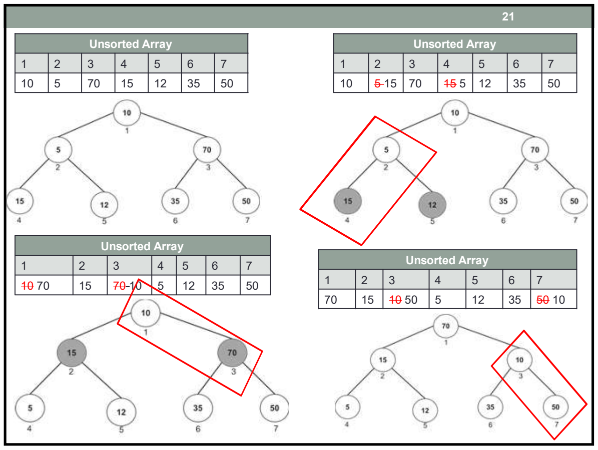
Heap Sort

- **HeapSort(a,n)** //a is a linear array and n is the last element of a

1. **Call Heapify(a, n)**
2. Repeat Step 3 and 4 For i=n to 1 in steps of -1
3. Swap elements a[1] with a[i]
4. Call ReheapifyDownward(a,1) on Heap from 1 to n-1 (see slide # 11)
5. Endfor
6. Return

Heapify(a,n)

1. Set index=Parent of node with index n
2. Repeat step 3 For i=index to 1 in setp of -1
3. Call reheapifyDownward(a,i) (see slide # 8)
4. Endif
5. Return



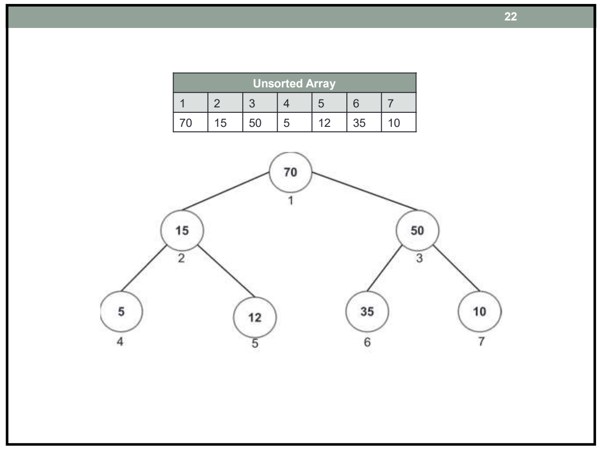
21

Unsorted Array						
1	2	3	4	5	6	7
10	5	70	15	12	35	50

Unsorted Array						
1	2	3	4	5	6	7
10	~~5~~ 15	70	~~15~~ 5	12	35	50

Unsorted Array						
1	2	3	4	5	6	7
~~10~~ 70	15	~~70~~ 10	5	12	35	50

Unsorted Array						
1	2	3	4	5	6	7
70	15	~~10~~ 50	5	12	35	~~50~~ 10



22

Unsorted Array						
1	2	3	4	5	6	7
70	15	50	5	12	35	10

23

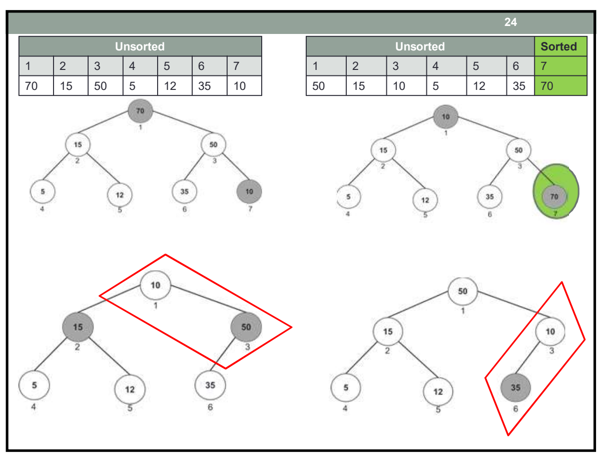
Heap Sort

- **HeapSort(a,n)** //a is a linear array and n is the last element of a

1. **Call Heapify(a, n)** ✓
2. Repeat Step 3 and 4 For i=n to 1 in steps of -1
3. Swap elements a[1] with a[i]
4. Call ReheapifyDownward(a,1) on Heap from 1 to n-1 (see slide # 11)
5. Endfor
6. Return

Heapify(a,n)

1. Set index=Parent of node with index n
2. Repeat step 3 For i=index to 1 in setp of -1
3. Call reheapifyDownwardward(a,i) (see slide # 8)
4. Endif
5. Return



24

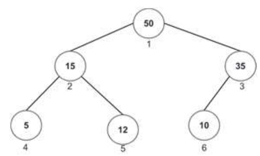
Unsorted						
1	2	3	4	5	6	7
70	15	50	5	12	35	10

Unsorted						Sorted
1	2	3	4	5	6	7
50	15	10	5	12	35	70

After First Iteration of

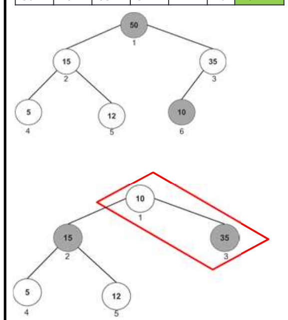
1. Repeat Step 3 and 4 For i=n to 1 in steps of -1
2. Swap elements a[1] with a[i]
3. Call ReheapifyDownward(a,1) on Heap from 1 to n-1 (see slide # 11)
4. Endfor

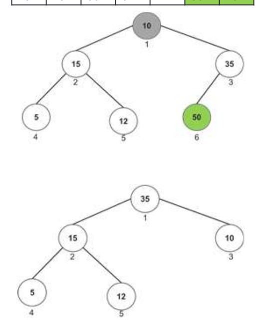
Unsorted						Sorted
1	2	3	4	5	6	7
50	15	35	5	12	10	70



Unsorted						Sorted
1	2	3	4	5	6	7
50	15	35	5	12	10	70

Unsorted					Sorted	
1	2	3	4	5	6	7
10	15	35	5	12	50	70

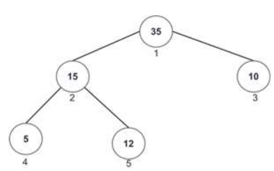




After second Iteration of

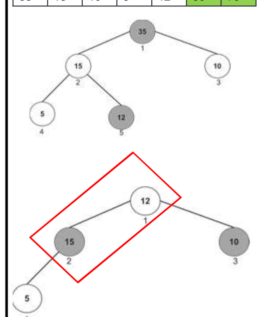
1. Repeat Step 3 and 4 For i=n to 1 in steps of -1
2. Swap elements a[1] with a[i]
3. Call ReheapifyDownward(a,1) on Heap from 1 to n-1 (see slide # 11)
4. Endfor

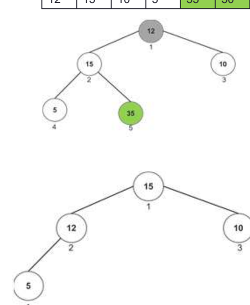
Unsorted					Sorted	
1	2	3	4	5	6	7
35	15	10	5	12	50	70



Unsorted					Sorted	
1	2	3	4	5	6	7
35	15	10	5	12	50	70

Unsorted				Sorted		
1	2	3	4	5	6	7
12	15	10	5	35	50	70

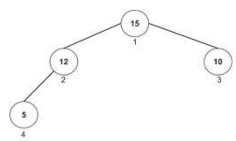




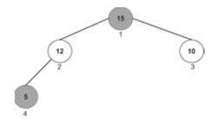
After Third Iteration of

1. Repeat Step 3 and 4 For i=n to 1 in steps of -1
2. Swap elements a[1] with a[i]
3. Call ReheapifyDownward(a,1) (see slide # 11)
4. Endfor

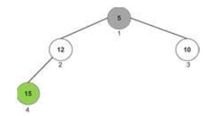
Unsorted				Sorted		
1	2	3	4	5	6	7
15	12	10	5	35	50	70

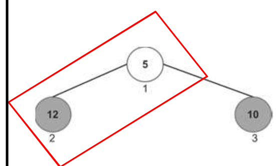


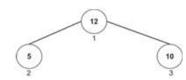
Unsorted Array				Sorted		
1	2	3	4	5	6	7
15	12	10	5	35	50	70



Unsorted Array			Sorted			
1	2	3	4	5	6	7
5	12	10	15	35	50	70



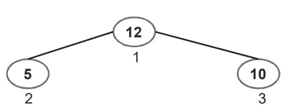




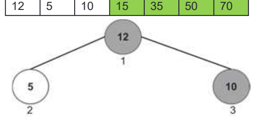
After 4th Iteration of

1. Repeat Step 3 and 4 For i=n to 1 in steps of -1
2. Swap elements a[1] with a[i]
3. Call ReheapifyDownward(a,1) (see slide # 11)
4. Endfor

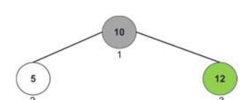
Unsorted			Sorted			
1	2	3	4	5	6	7
12	5	10	15	35	50	70



Unsorted			Sorted			
1	2	3	4	5	6	7
12	5	10	15	35	50	70



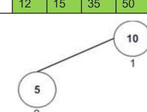
Unsorted		Sorted				
1	2	3	4	5	6	7
10	5	12	15	35	50	70



Unsorted		Sorted				
1	2	3	4	5	6	7
10	5	12	15	35	50	70

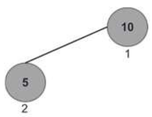
After 5th Iteration of

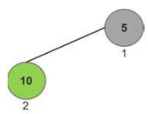
1. Repeat Step 3 and 4 For i=n to 2 in steps of -1
2. Swap elements a[1] with a[i]
3. Call ReheapifyDownward(a,1,i-1) (see slide # 11)
4. Endfor



Unsorted		Sorted				
1	2	3	4	5	6	7
10	5	12	15	35	50	70

Unsorted	Sorted					
1	2	3	4	5	6	7
5	10	12	15	35	50	70





After 6th Iteration of

1. Repeat Step 3 and 4 For i=n to 1 in steps of -1
2. Swap elements a[1] with a[i]
3. Call ReheapifyDownward(a,1) (see slide # 11)
4. Endfor

Sorted Array						
1	2	3	4	5	6	7
5	10	12	15	35	50	70

Another Example of Heapsort

https://www.cs.usfca.edu/~galles/visualization/HeapSort.html

Thank You