HEAPS AND HEAPSORT

Partha P Chakrabarti
Indian Institute of Technology Kharagpur

Overview of Algorithm Design

1. Initial Solution
   a. Recursive Definition – A set of Solutions
   b. Inductive Proof of Correctness
   c. Analysis Using Recurrence Relations
2. Exploration of Possibilities
   a. Decomposition or Unfolding of the Recursion Tree
   b. Examination of Structures formed
   c. Re-composition Properties
3. Choice of Solution & Complexity Analysis
   a. Balancing the Split, Choosing Paths
   b. Identical Sub-problems
4. Data Structures & Complexity Analysis
   a. Remembering Past Computation for Future
   b. Space Complexity
5. Final Algorithm & Complexity Analysis
   a. Traversal of the Recursion Tree
   b. Pruning
6. Implementation
   a. Available Memory, Time, Quality of Solution, etc

1. Core Methods
   a. Divide and Conquer
   b. Greedy Algorithms
   c. Dynamic Programming
   d. Branch-and-Bound
   e. Analysis using Recurrences
   f. Advanced Data Structuring
2. Important Problems to be addressed
   a. Sorting and Searching
   b. Strings and Patterns
   c. Trees and Graphs
   d. Combinatorial Optimization
3. Complexity & Advanced Topics
   a. Time and Space Complexity
   b. Lower Bounds
   c. Polynomial Time, NP-Hard
   d. Parallelizability, Randomization

Problems & Data Structure Requirements

Searching, Sorting, Max, Min, Max & Min, Max & Next Max

Generalized recurrences
↳ Fibonacci (Pingala)

Pattern Matching, Sequence Alignment

Convex Hull, Closest Pair of Points
Matrix Multiplication, Multiplication of n-bit numbers

Coin Selection, Power Line optimal Layout, Activity Selection, Matrix Chain

---

Data Storage, Memoization, Access & Reuse

Operations :-
insert, delete, find, max, min, update key, retrieve in ordered fashion, set operations (∪, ∩, Diff etc)

Data Structures

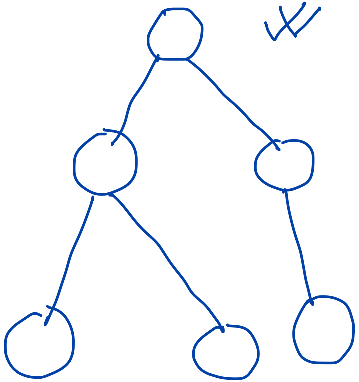
1. Lists, Queues, Stacks, Arrays, etc
2. Trees & Tree-like
   - Tournament, Heap
   - Search Trees, BST
3. Graphs
4. Sets

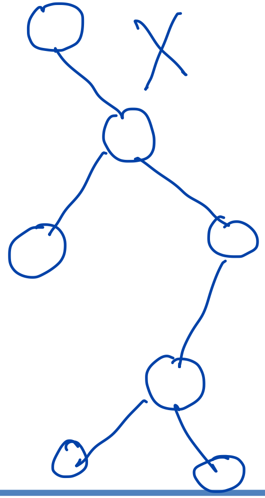
Heap Data Structure: Definition

Max & Min , Max & 2nd Max, Sorting

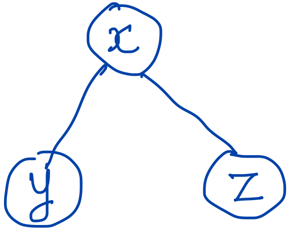
Heap :-

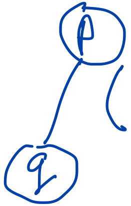
- Complete Binary Tree
- "Heap" Property



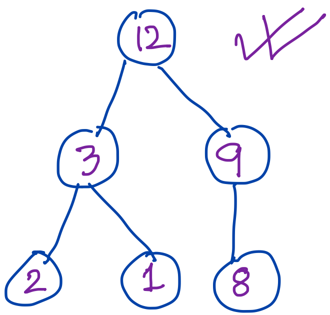


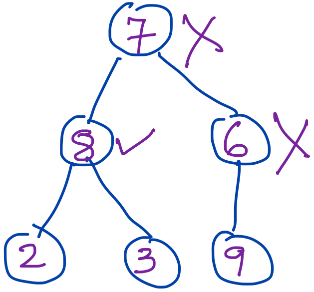
Heap Property (Max Heap)



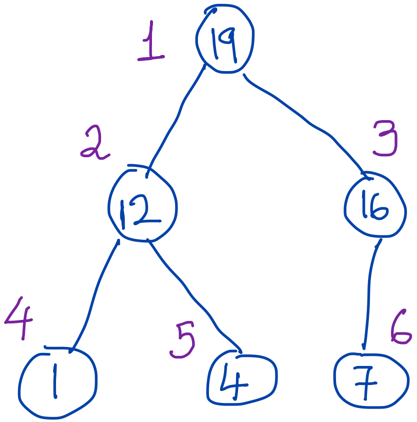


$(x \geq y)$ & $(x \geq z)$





Heap Data Structure: Representation



size(Heap)

1	2	3	4	5	6	7	8
19	12	16	1	4	7		

left(n) → A[2n]
right(n) → A[2n+1]
parent(n) → A[⌊n/2⌋]

- sizeof(A)
- left-child(A, n)
- right child(A, n)
- parent(A, n)
- root(A)
- end(A)

Heap Operations

operations:-

- insert (A, k) [with key = k]
  - ↳ insert a new element and reorder the heap to maintain the heap property
- remove_max (A) /Max-Heap/
  - ↳ remove the largest element from the heap and reorder the heap to maintain the heap property

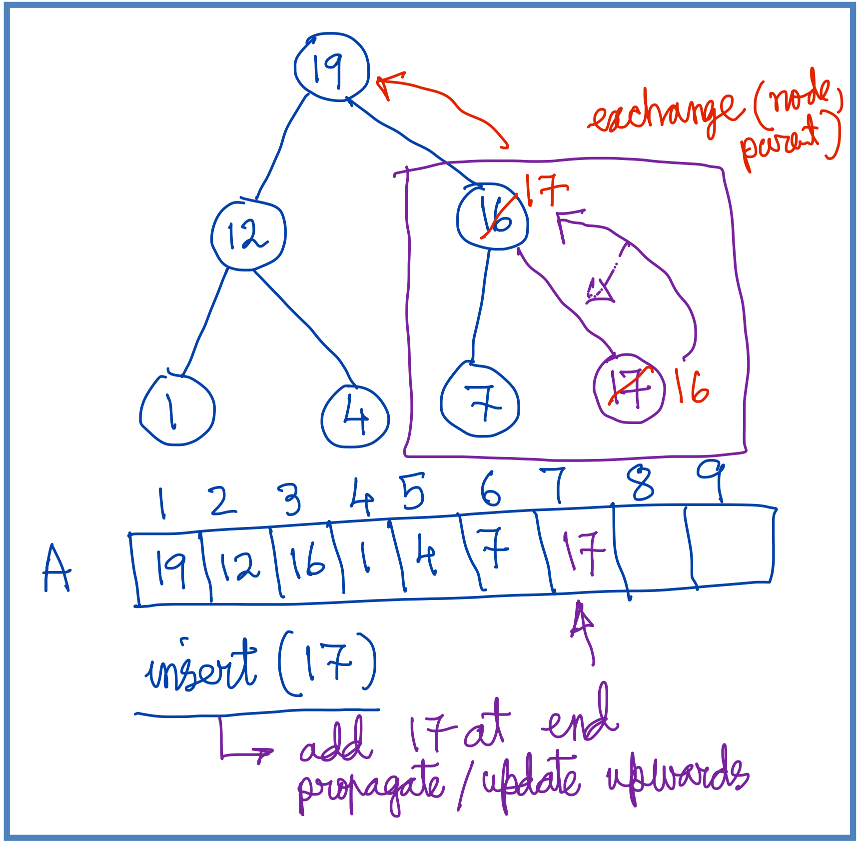
- Build Heap
  - ↳ Given a set of elements, make a heap.

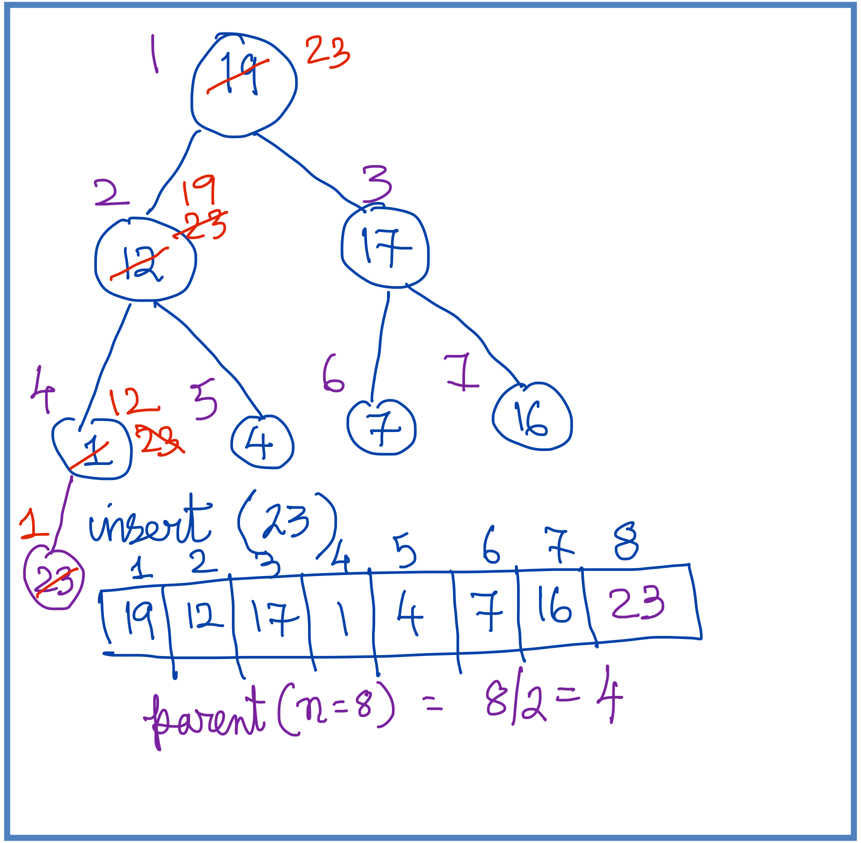
---

auxiliary operations

- update_key (A, n, k)
  - ↳ Given a new key value k to a node n, it will update the key value and reorder the heap to maintain heap property
- delete (A, n)
- find (A, k) // Difficult for heap //

Heap Operation: Insert_new





Heap Operation: Insertion Algorithm

```
insert (A, k)
{ n = add (A, k) / adds to end of A /
     update_up (A, n)
}

update_up (A, node)
{  if (node ≠ root)
      { if (key (node) > key (parent (node)))
          { exchange (node, parent)
             update_up (A, parent)
          }
      }
}
```

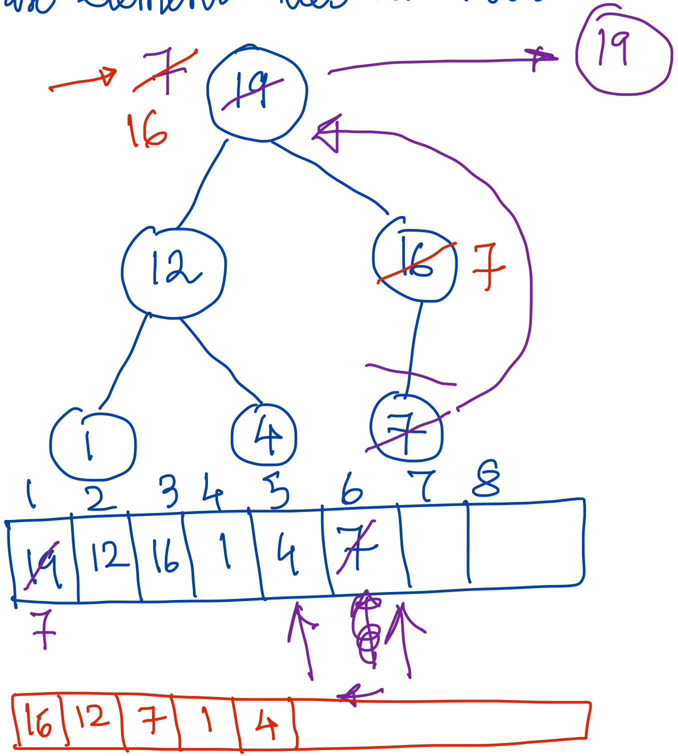
Complexity

→ Height of the Heap (form of a Binary Tree)

→ $\lceil \log(n) \rceil$

Heap Operation: Remove_Max

Max element lies at root

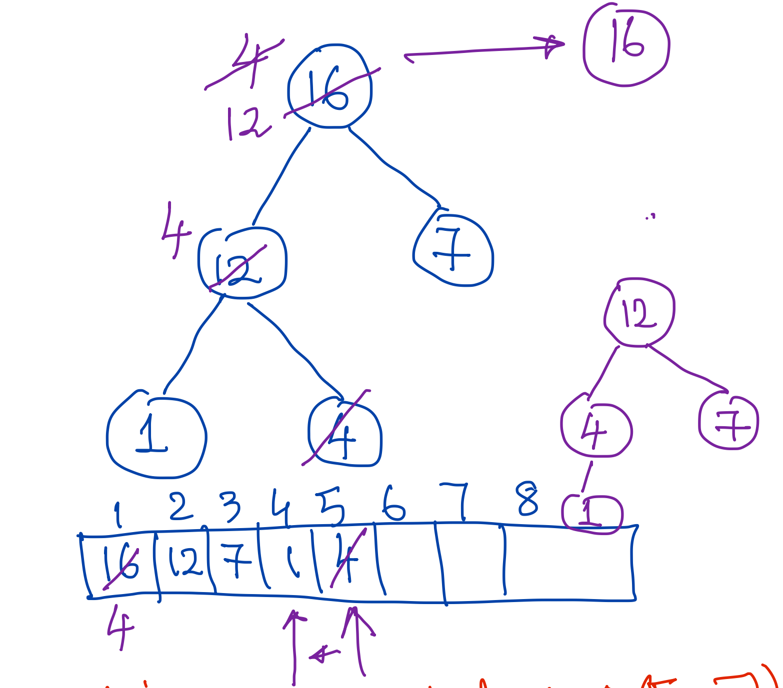


1. Remove the root → A[1]
2. Replace A[1] by A[size(A)]
3. Reduce size(A) by 1.
4. update_down (A, node)
   ↑ root

Heapify (A, node)
└→ update_down

Heap Operation: Heapify Algorithm

```
update-down (A, node)
{   l ← left(node)
    r ← right(node)
    large = largest-key (node, l, r)
    if (large ≠ node)
      { exchange (node, large)
        update-down (A, large)
      }
}
```



Complexity : Height of heap O(⌈log n⌉)

Building Initial Heap

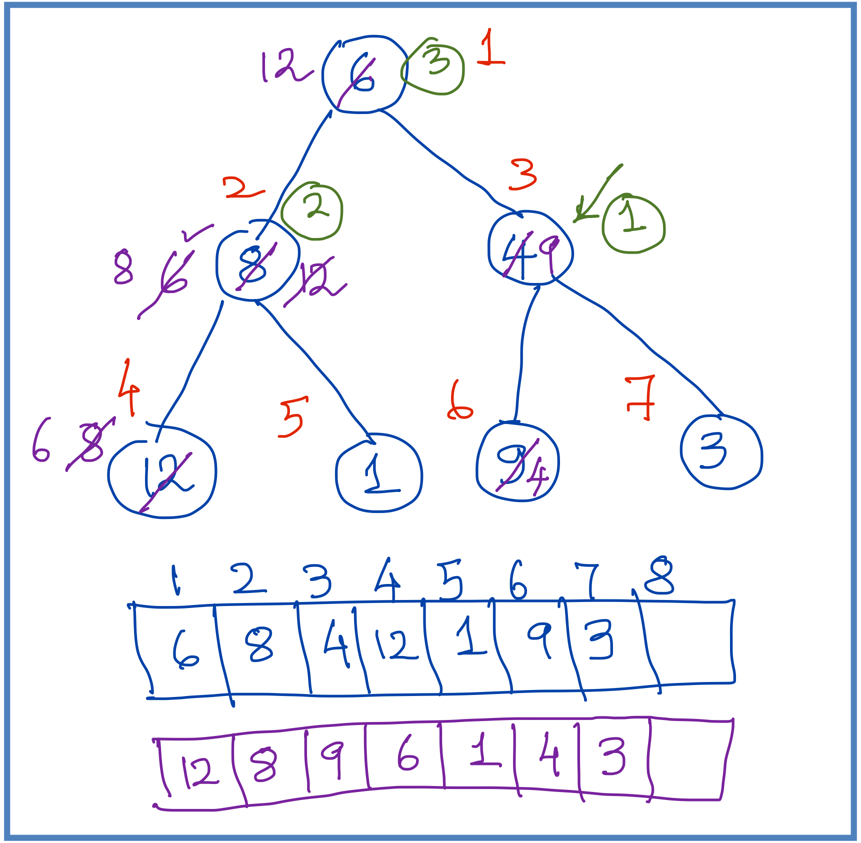
option 1: insert the n elements one after another

$O(n \log n)$

option 2:

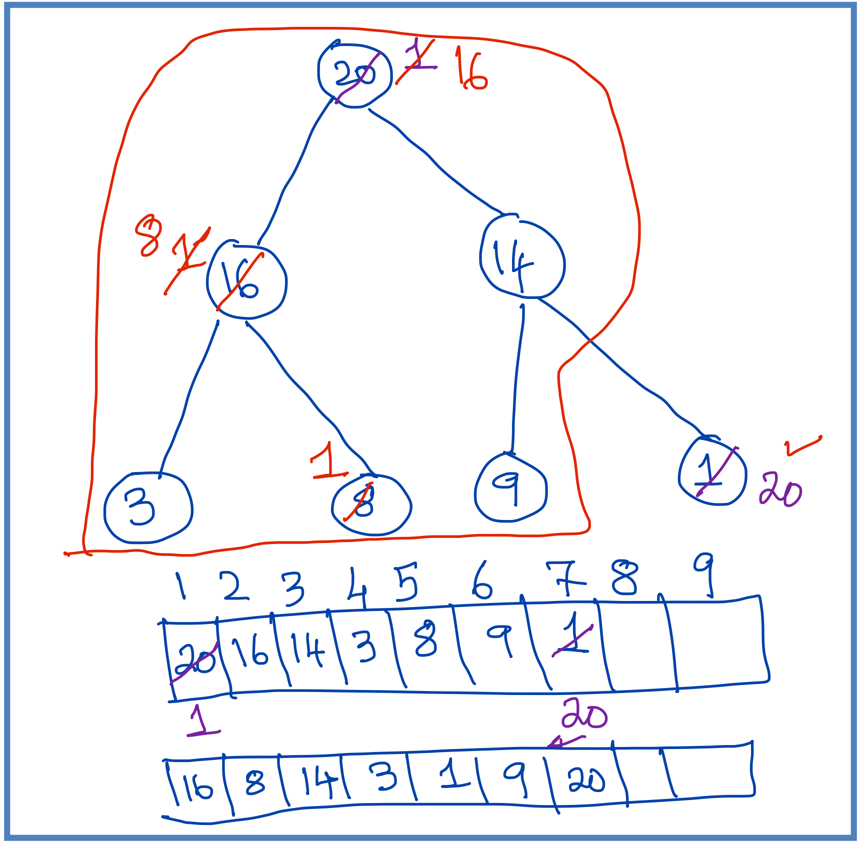
```
Build_heap(A)
{ initialize A randomly as per input.
  for i = A[⌊size/2⌋] to 1
  { update_down(A, i)
}
```

$O(n)$ ✓

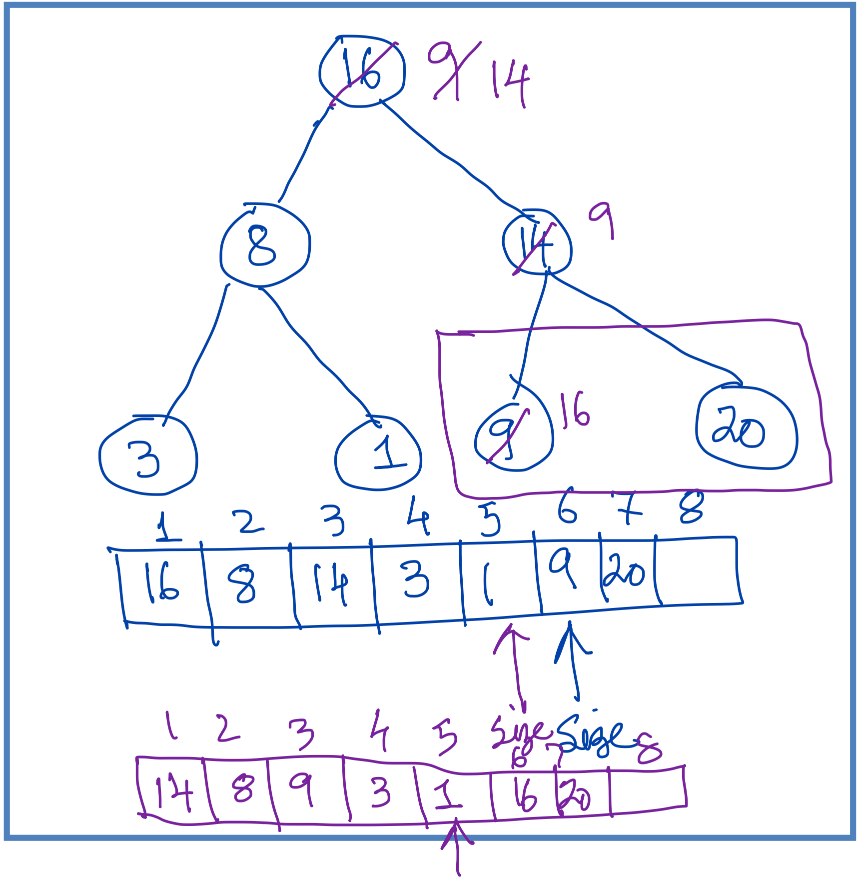


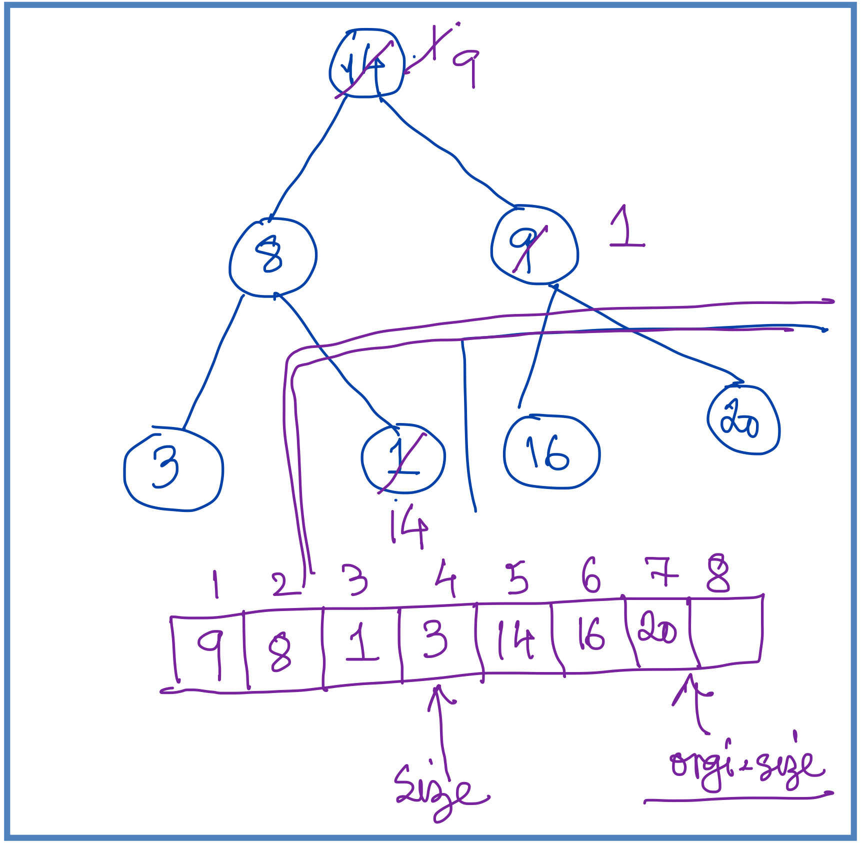
Heap Sort

```
Heapsort (A)
{ Buildheap (A)
    for i = size to 2 do
    { swap ( A[1], A[i] )
      size = size - 1
      update_down (A, 1)
    }
}
```



Heap Sort: Example





Heap Sort: Analysis

Build Heap: O(n)

Sorting: –

- Remove & Exchange O(1)
- update-down — O(log n)

→ O(n) times

Heapsort: O(n log n)

Compare with [Both Space & Time]

Quicksort

Mergesort

other Sorts

Summary and Extensions

→ update_key
→ delete

dynamic series of data structure operations →

Priority Queue

Thank you

Any Questions?