HEAP AND PRIORITY QUEUE and Applications (BRANCH-and-BOUND)

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)

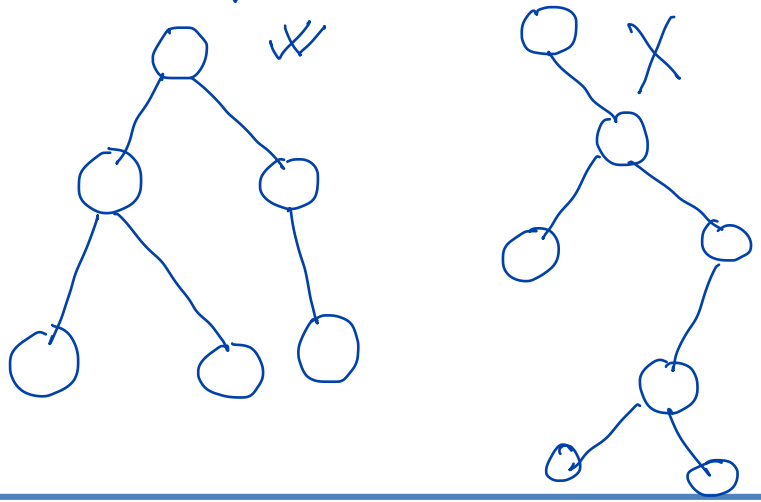
Data Structures
1. Lists, Queues, Stacks, Arrays, etc
2. Trees & Tree-like
   - Tournament, Heap
   - Search Trees, BST
3. Graphs
4. Sets

Heap Data Structure: Definition

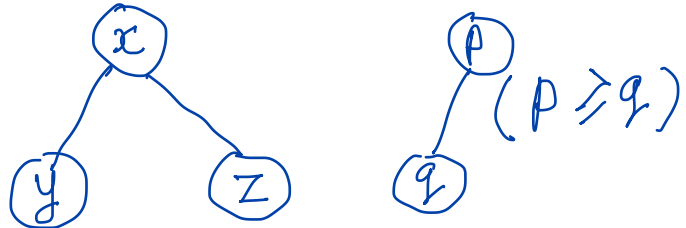
Max & Min , Max & 2nd Max, Sorting

Heap :-

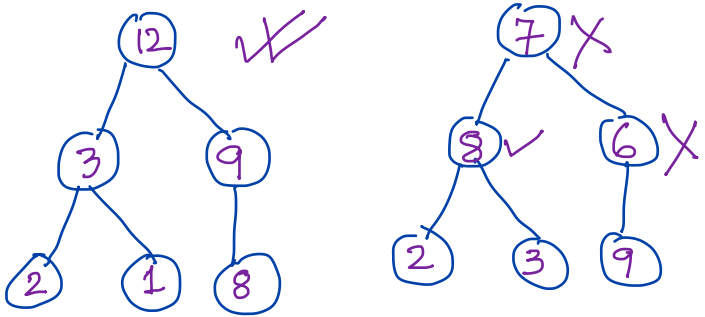
- Complete Binary Tree
- "Heap" Property



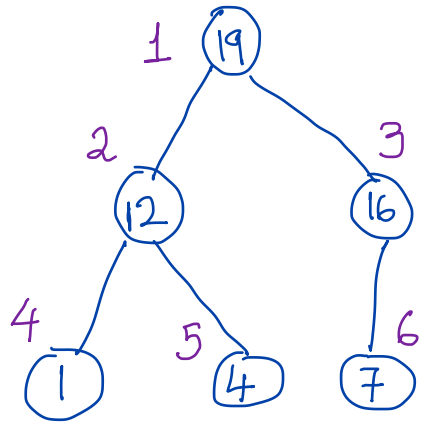
Heap Property (Max Heap)



(x ≥ y) & (x ≥ 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

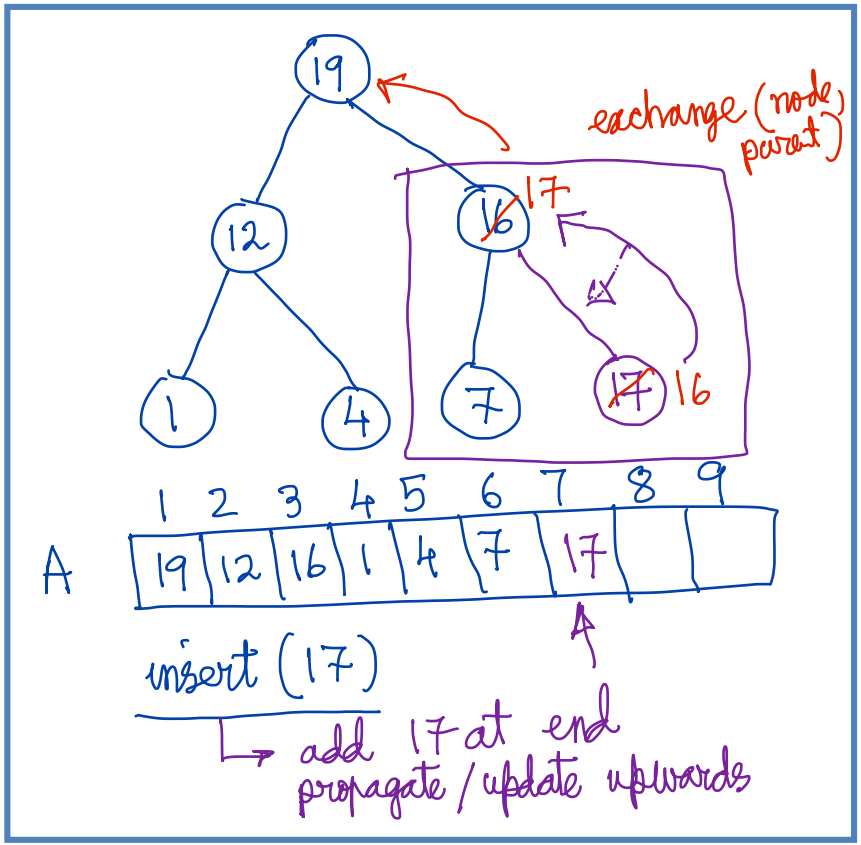
- Build Heap
  - ↳ Given a set of elements, make a heap.

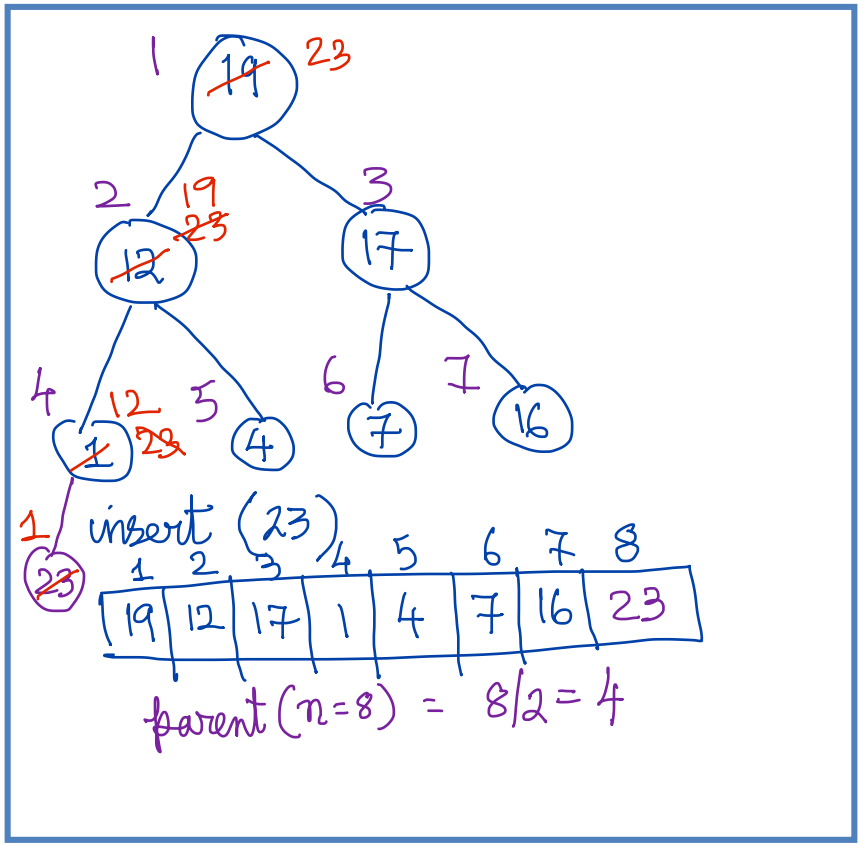
---

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

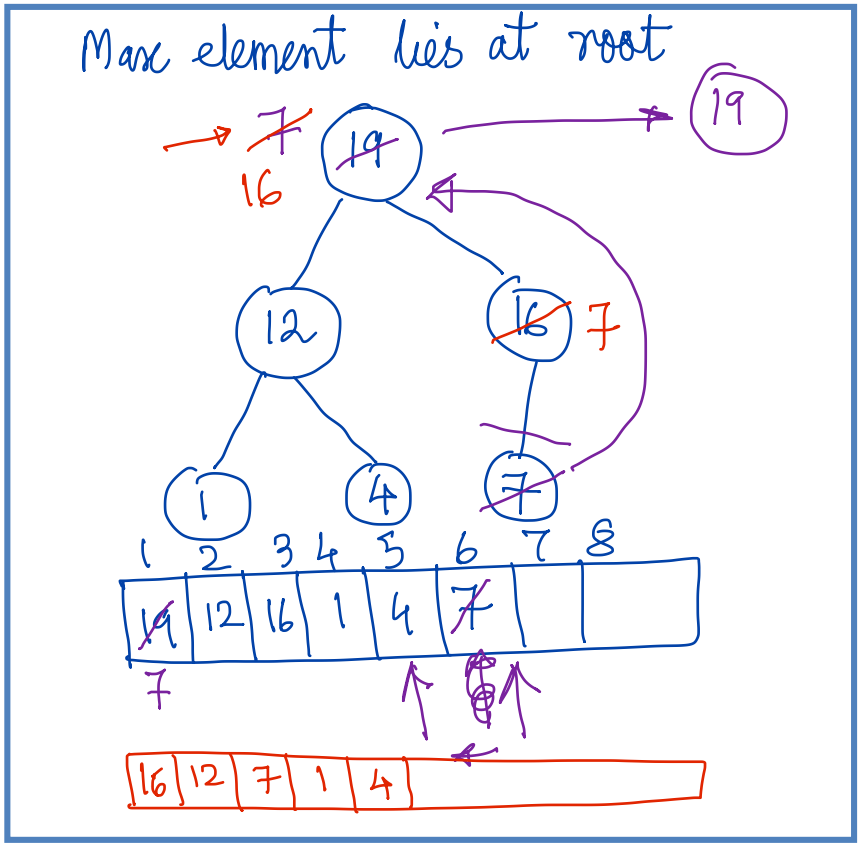
```
insert (A, k)
{ n = add (A, k) / adds to end of A /
    update_up (A, n)
}

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

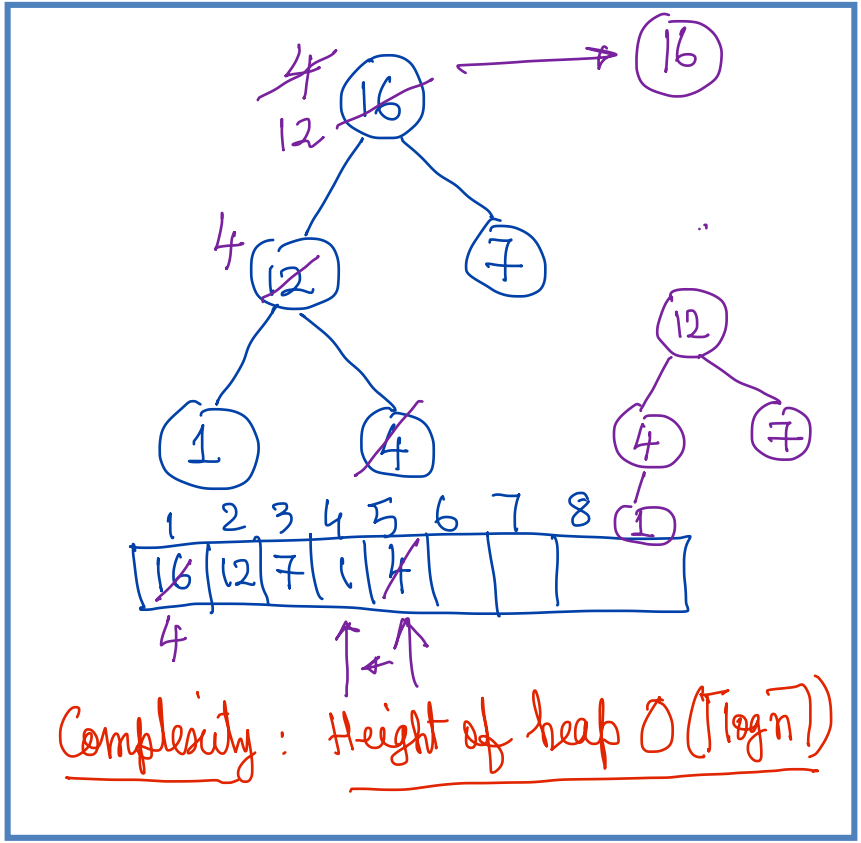


1. Remove the root → A[1]
2. Replace A[1] by A[size(A)]
3. Reduce size(A) by 1.
4. update-down (A, node)
   ↑ root

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)
     }
}
```



Building Initial Heap

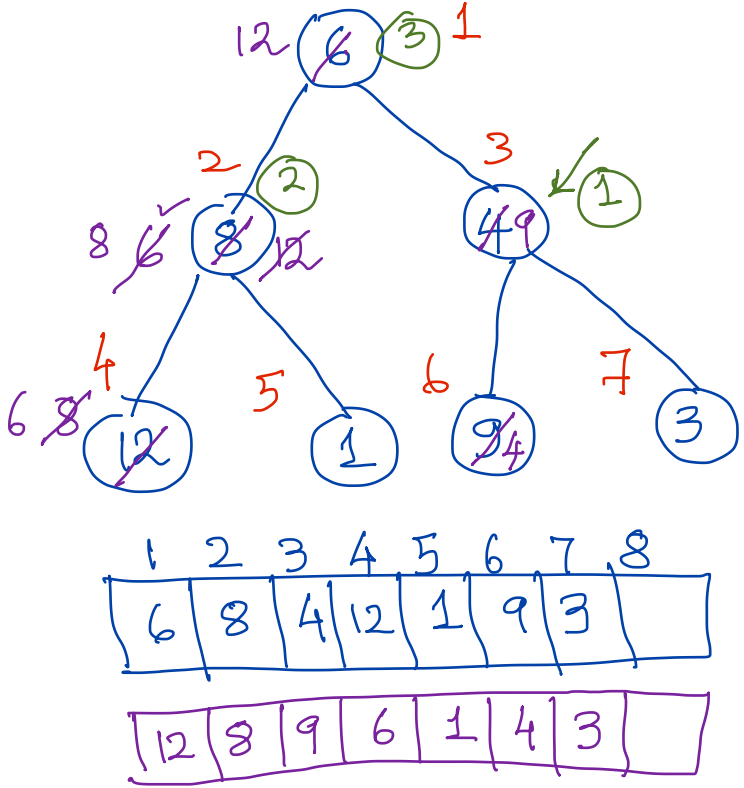
Option 1: insert the n elements one after another

$O(n \log n)$

Option 2:

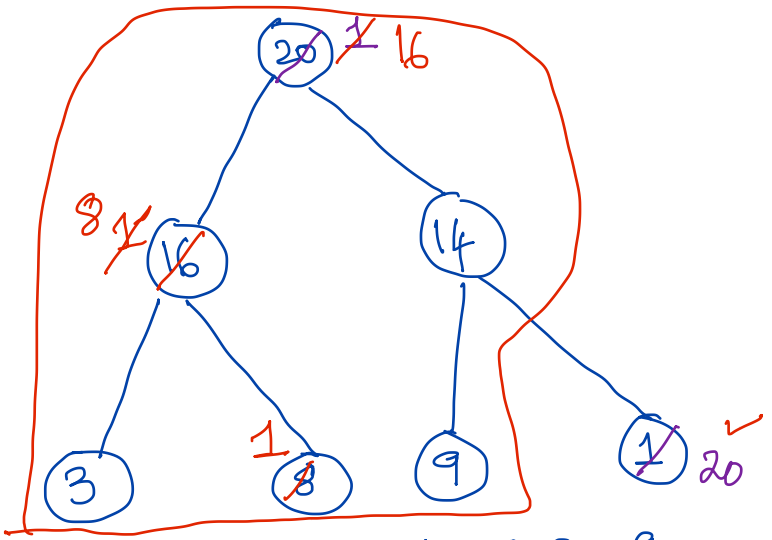
```
Build_heap (A)
{ Initialize A randomly as per input
  for i = A[⌊size/2⌋] to 1
    { update_down (A, i)
  }
```

$O(n)$ ✓

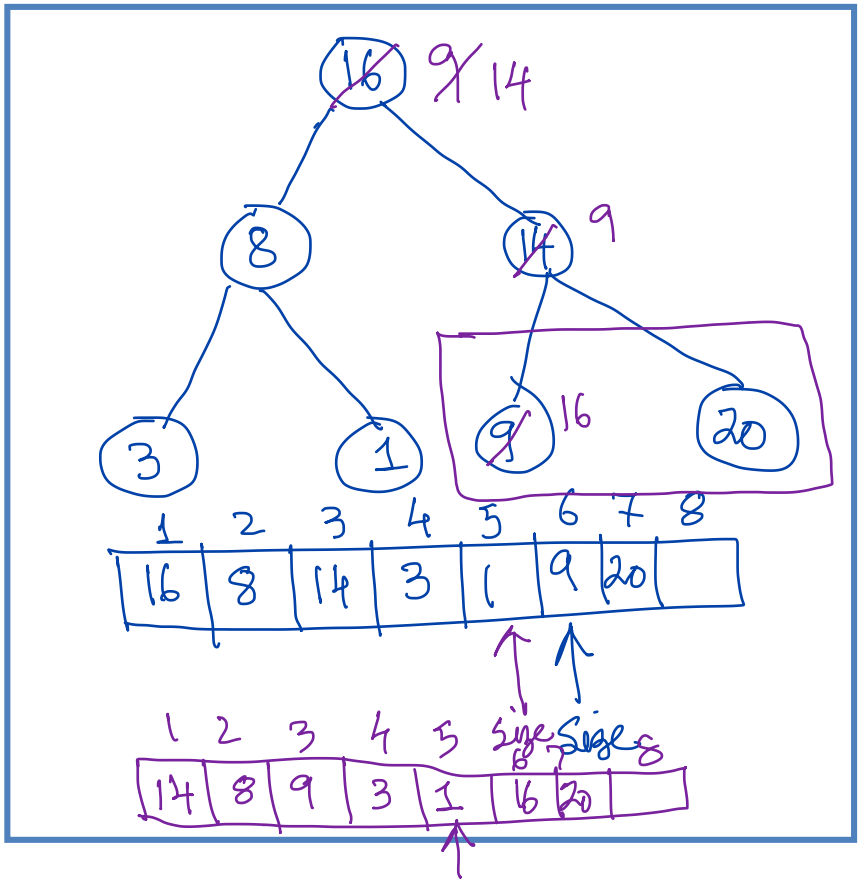


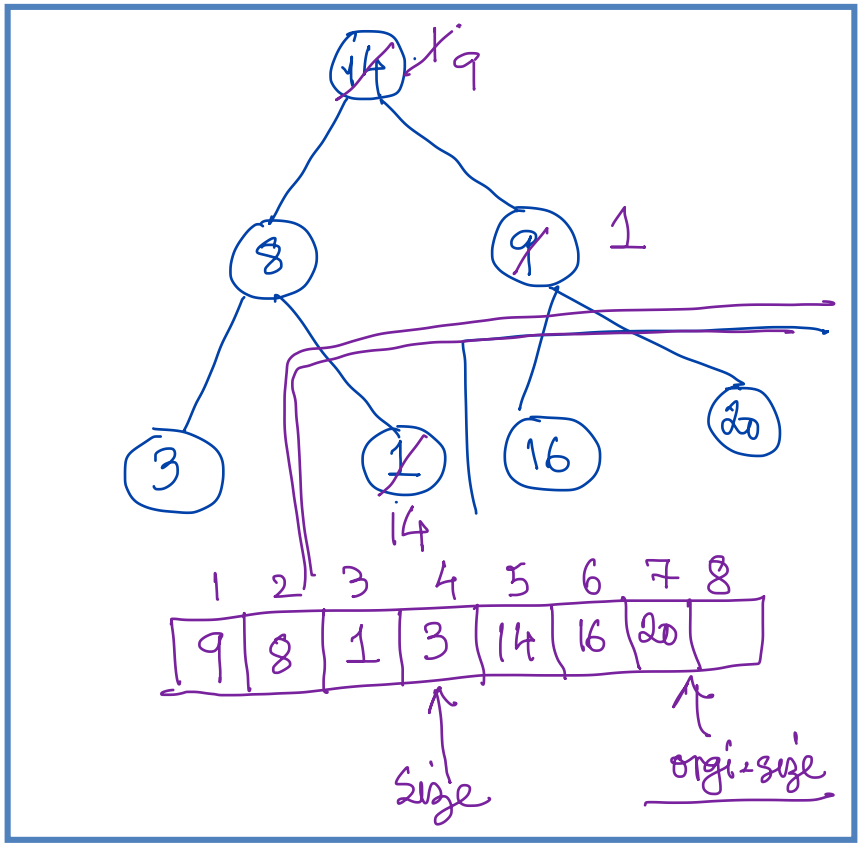
Heap Sort

```
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

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

Stacks
Queues
Lists
Arrays
Heap, BST
Priority Queue

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

Priority Queue: Operations & Applications

Queue : FIFO

→ Elements have key values which could have an ordering

- Sorting, searching, optimization problems, max-min, max-next max
- Pole Wiring Problem
- Coins
- Activity Selection

Operations

- insert
- delete Max / Min

Heap

- update
- delete element

BST

- find

static :-

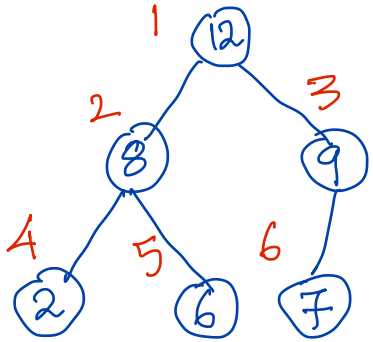
dynamic :-

Priority Queue: Heap Implementation

Build Heap
Insert-new-element
Remove Min / Max

update element → provided we have pointer or index of node
delete element

1. Complete Binary Tree
2. Heap property



1	2	3	4	5	6	7
12	8	9	2	6	7	

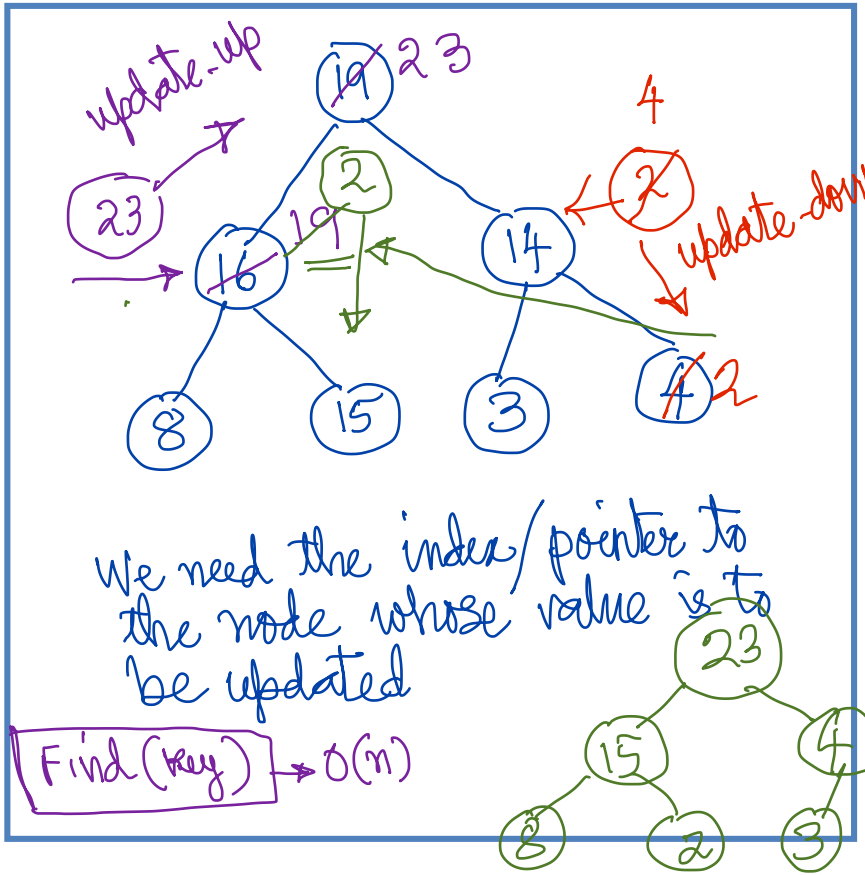
insert-new
- inserts at the end
- update-up (A, node) O(log n)

remove-max
- remove to root or first element
- replace root by last element
- update-down (A, node) O(log n)

Heapify

Build-heap (Bottom up) update-down iteratively O(n)

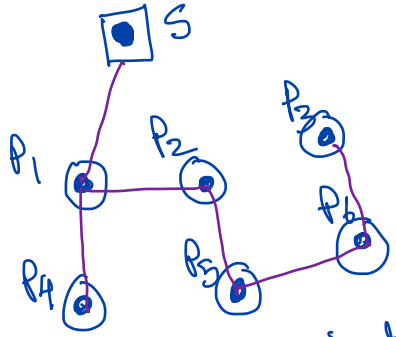
Heap Priority Queue: Update / Revise / Delete



update (A, node)
{ check whether the key (node) is greater than parent
↳ if so → update_up (A, node)
else
O (log n) ↳ update - down (A, node)

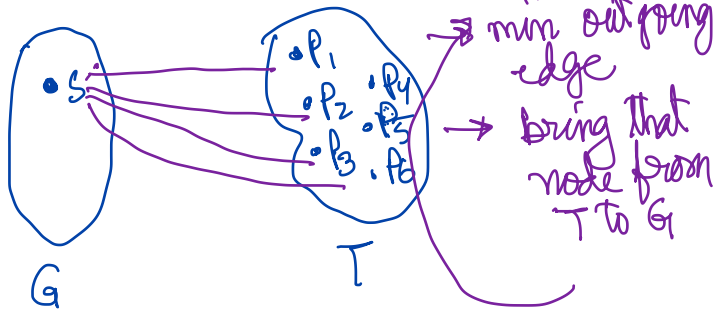
delete (A, node)
{ - remove (A, node)
O (log n) - replace by the last element of heap; decrement size
- update (A, node)
}

Electrical Pole Placement Problem



Minimum cost spanning Tree

Greedy optimization Solution



what data structure to use?

operations :- PRIORITY QUEUE

R: Set of edges between G & T

- insert-new
- delete
- remove-min

→ Edges (e)

→ Nodes (n)

each operation will be of $O(e)$

$O(n \log e + e \log e)$

$= O(e \log e)$

$= O(n^2 \log n)$

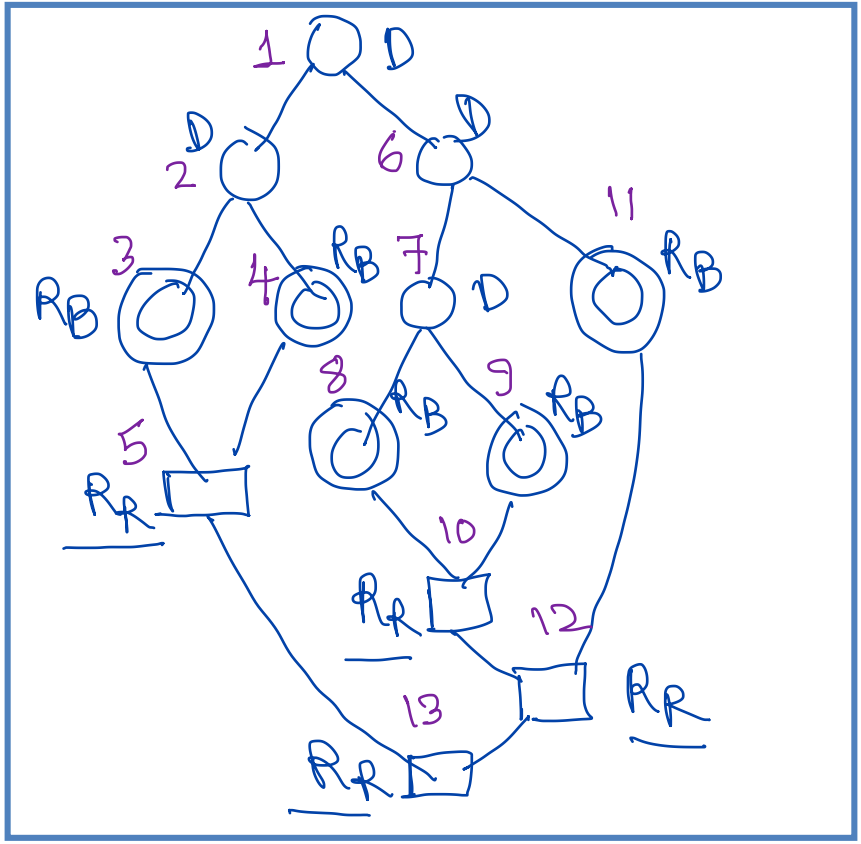
$e = O(n^2)$

Evaluation of Recursion Tree

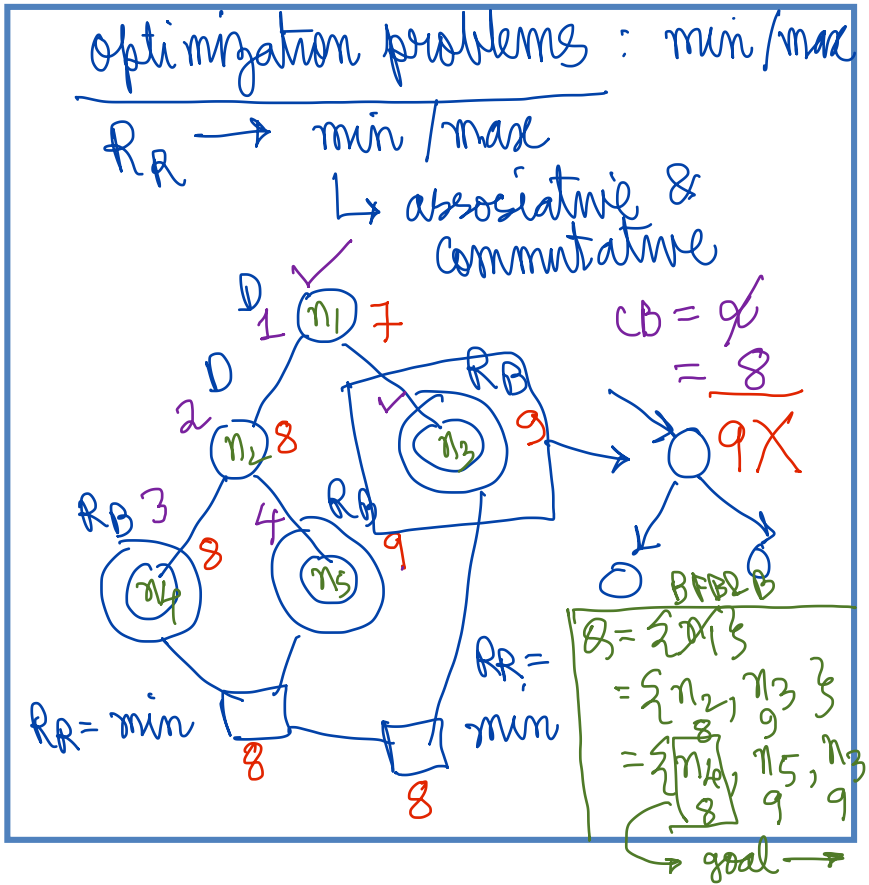
$$f(X) = R_B(X) \text{ if } B(X)$$
$$= \{ 1.\ (Y_1, Y_2, \ldots, Y_K) = D(X)$$
$$2.\ R_R(f(Y_1), f(Y_2), \ldots, f(Y_K)) \}$$

Evaluation of the Recursion Tree

- Depth-first → Stack
- Breadth-first → Queue
- Iterative Deepening



Pruning & Branch & Bound



pruning

DFB&B :— Maintain a current best (initialized to ∞)

- whenever we find a solution we update current best to the better solution
- Any node which has got cost ≥ current best is pruned

Best first B&B

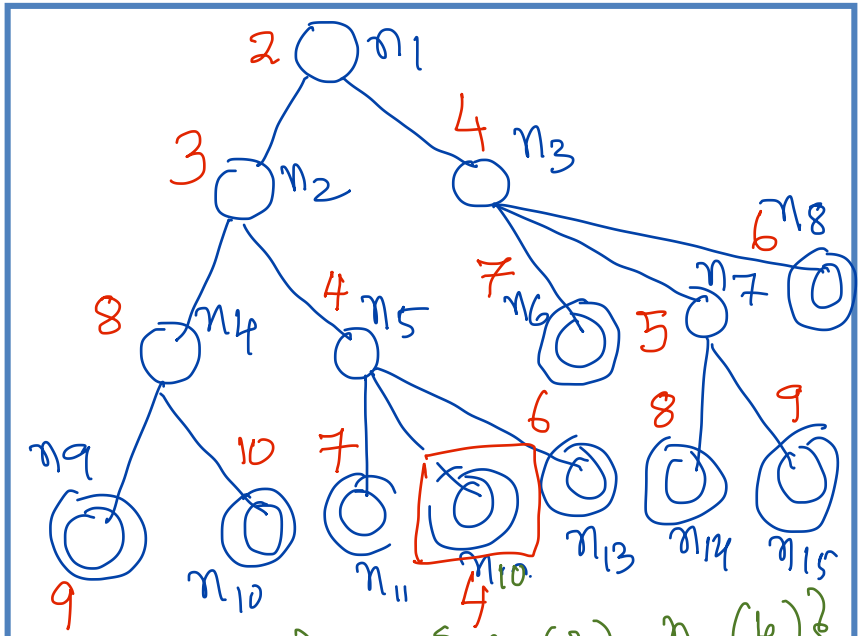
- ordered search using a priority queue of costs :
  - Remove-Min, insert-children, stop when goal is reached.

Best First Search using Priority Queues

MINIMIZATION

1. Initialize : $Q = \{start\}$
2. Remove from Q to node n with lowest cost. If Q is empty → fail
3. If n is goal → stop with solution
4. Generate successors of node n using the Decomposition Rule Let them be $n_1, n_2, \ldots, n_k$
5. If $n_i \notin Q$ insert n_i in Q
6. If $n_i \in Q$ update n_i in Q
7. Goto Step 2.

Q: Priority Queue
1. Insert
2. Delete Min
3. Update



$Q = \{n_1(2)\} \rightarrow \{n_2(3), n_3(4)\}$

$\{n_4(8), n_5(4), \ldots\} \leftarrow \{n_3(4), n_4(8), n_5(4), \ldots\}$

$\{n_6(7), n_7(5), n_8(6)\} \rightarrow \{\ldots n_{10}(4) \ldots\}$

SOLUTION

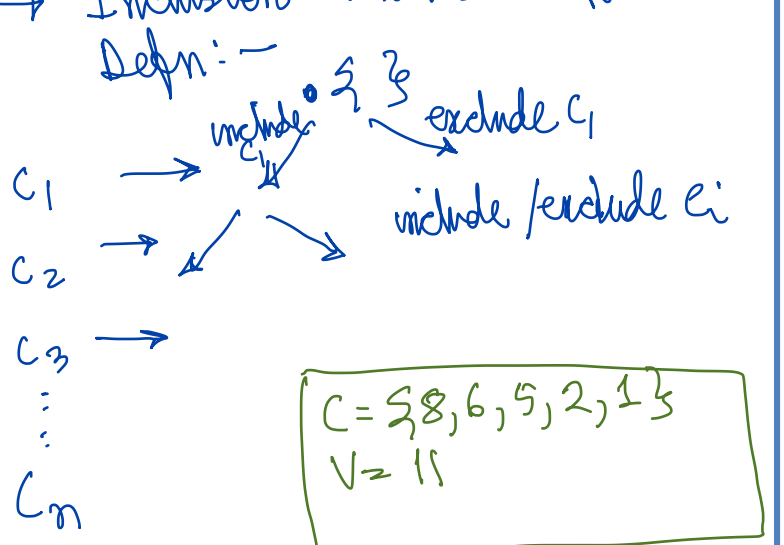
Coin Selection Problem

Coins = $\{C_1, C_2, \ldots, C_n\}$, V

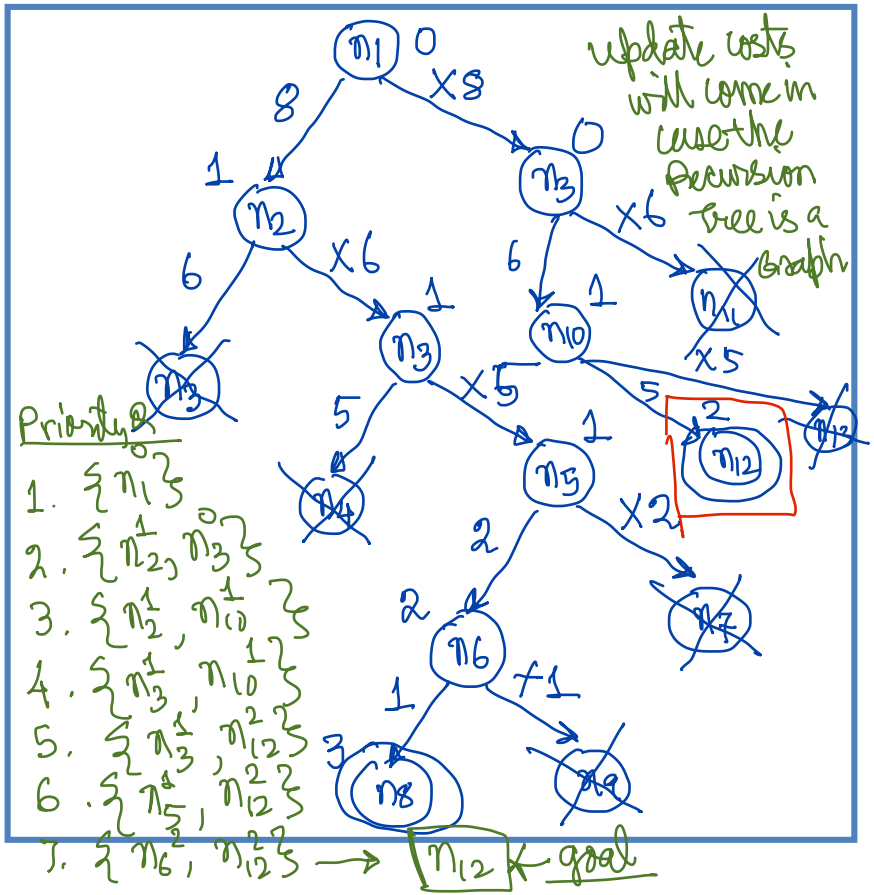
↳ Find the minimum no. of coins to get the exact value of V

→ Inclusion - Exclusion Recursive

Defn:-



C = {8, 6, 5, 2, 1}
V = 11



Related Data Structures

- Binary Search Trees (BSTs)
  - Balanced BSTs
    - AVL Trees
    - B-Trees, etc
- Advanced Heaps
  - Binomial Heaps
    Fibonacci Heaps
    etc

- Weighted BSTs
  - frequency of access
    (DYNAMIC PROGRAMMING METHOD)
- Data Structures using Advanced operations
  - Union, Intersection, Set Difference, etc.

Summary

- insert
- remove-min/max
- update/delete

↳ Priority Queue operations

↳ Implemented by a Heap Data Structure

In case we have
Find ← BST
Union/Intersection, etc ← ??

Operations could be
- provided before the algo. (static)
- online

Large class of optimization problems → Greedy, B&B

Thank you

Any Questions?