# PDSA Notes

by

Gagneet Kaur

January 31, 202

PAGE NO.
DATE:

# WEEK 6

# Union-Find Data Structure

## KRUSKAL's ALGORITHM FOR MCST

- Process edges in ascending order of cost
- If edge (u,v) does not create a cycle, add it
  - → (u,v) can be added if u and v are in different components
  - → Adding edge (u,v) merges these components
- How can we keep track of components and merge them efficiently?
- Components partition vertices
  - ↳ collection of disjoint sets
- Need data structure to maintain collection of disjoint sets
  - → find (v) - return set containing v
  - → union (u,v) - merge sets of (u,v)

# UNION - FIND DATA STRUCTURE

- A set S partitioned into components $\{C_1, C_2, \ldots, C_k\}$
  Each $s \in S$ belongs to exactly one $C_j$

- Support the following operations

→ MakeUnionFind (S) - setup initial singleton components $\{s\}$ for each $s \in S$

→ Find (s) - return the component containing s

→ Union(s, s') - merges components containing s, s'

# NAIVE IMPLEMENTATION

- Assume $S = \{0, 1, \ldots, n-1\}$
- Setup an array / dictionary 'Component'
- MakeUnionFind (S) - set component[i] = i for each i
- Find (i) - Return Component [i]
- Union (i, j)

```
c_old = Component [i]
c_new = Component [j]
for k in range (n):
    if Component [k] == c_old
       Component [k] == c_new
```

- Complexity

→ MakeUnionFind (S) - O(n)
→ Find(i) - O(1)
→ Union(i, j) - O(n)
→ Sequence of m Union() operations takes time O(mn)

---

# IMPROVED IMPLEMENTATION

- Another array / dictionary 'Members'

- For each component c, Members[c] is a list of its members

- Size[c] = length(Members[c]) is the number of members

- MakeUnionFind (S)
  - → Set Component[i] = i for all i
  - → Set Members[i] = [i], Size[i] = 1 for all i

- Find (i)
  - → Return Component[i]

- Union (i, j)

```
c-old = Component[i]
c_new = Component[j]
for k in Members[c-old]:
    Component[k] = c_new
```

Members [c_new] . append(k)
Size [c_new] = Size [c_new] + 1

# WHY DOES THIS HELP ?

- Members [c_old] allows us to merge Component [i] into Component [j] in time O( size [c_old] ) rather than O(n)

- How can we make use of size [c]
  - → Always merge smaller component into larger one
  - → If Size[c] < size[c'] relabel c as c', else relabel c' as c

- Individual merge operations can still take time O(n)
  - → Both Size[c], Size[c'] could be about n/2.
  - → More careful accounting

- Always merge smaller component into larger one

- For each i, size of Component [i] at least doubles each time it is relabelled

- After m Union() operations, at most 2m elements have been "touched"
  - → Size of Component [i] is at most 2m

- Size of Component [i] grows as 1, 2, 4, ... , so i changes component at most log m times.

PAGE NO.
DATE:

- Over $m$ updates
  - → At most $2m$ elements are relabelled
  - → Each one at most $O(\log m)$ times
- Overall, $m$ Union() operations take time $O(m \log m)$
- Works out to time $O(\log m)$ per Union() operation
- Amortized complexity of Union() is $O(\log m)$

---

# BACK TO KRUSKAL'S ALGORITHM

- Sort $E = \{e_0, e_1, \ldots, e_{m-1}\}$ in ascending order
- MakeUnionFind(V) - each vertex $j$ is in component $j$
- Adding and edge $e_k = (u,v)$ to the tree

→ Check that Find(u) != Find(v)
→ Merge components :
    Union(Component[u], Component[v])

- Tree has $n-1$ edges, so $O(n)$ Union() operations
  → $O(n \log n)$ amortised cost, overall
- Sorting E takes $O(m \log m)$
  Equivalently $O(m \log n)$, since $m \leq n^2$
- Overall time, $O((m+n) \log n)$

# SUMMARY

→ Implement Union-Find using arrays / dictionaries 'Component', 'Member', 'Size'

- MakeUnionFind(S) is O(n)
- Find(i) is O(1)
- Across m operations, amortized complexity of each Union() operation is log m

→ Can also maintain Members[k] as a tree rather than as a list

- Union() becomes O(1)
- With clever updates to the tree, Find() has amortized complexity very close to O(1).

Feb 1, 2022

# Priority Queues

## DEALING WITH PRIORITIES

# Job Scheduler

- A job scheduler maintains a list of pending jobs with their priorities
- When the processor is free, the scheduler picks out the job with maximum priority in the list and schedules it.
- New jobs may join the list at any time
- How should the scheduler maintain the list of pending jobs and their priorities?

# Priority Queue

- Need to maintain a collection of items with priorities to optimise the following operations
- delete_max()
  - → Identify and remove item with highest priority
  - → Need not be unique

# Maintaining as a list incurs cost $O(N^2)$ across N inserts and deletions

# Using a $\sqrt{N} \times \sqrt{N}$ array reduces the cost to $O(\sqrt{N})$ per operations ; $O(N\sqrt{N})$ across N inserts and deletion



- insert ( )
  - → add a new item to the collection

# IMPLEMENTING PRIORITY QUEUES

## # With One Dimensional Structures

- Unsorted List
  - → insert() is $O(1)$
  - → delete_max () is $O(n)$

- Sorted List
  - → delete_max() is $O(1)$
  - → insert () is $O(n)$

- Processing n items requires $O(n^2)$

## # Moving to 2-dimensions

→ FIRST ATTEMPT

- Assume N processes enter/leave the queue
- Maintain a $\sqrt{N} \times \sqrt{N}$ array
- Each row is in sorted order

insert ( )

- → keep track of the size of each row
- → insert into the first row that
  - · use size of row to determine

- Insert 15
- Takes time $O(\sqrt{N})$
  - Scan size column to locate row to insert, $O(\sqrt{N})$
  - Insert into the first row with free space, $O(\sqrt{N})$

delete_max ()

- maximum in each row is the last element
- position is available through size column
- identify the maximum amongst these
- delete it
- Again $O(\sqrt{N})$
  - find the maximum among last entries, $O(\sqrt{N})$
  - delete it, $O(1)$

---

## SUMMARY

- 2D $\sqrt{N} \times \sqrt{N}$ array with sorted rows
  - insert() is $O(\sqrt{N})$
  - delete_max() is $O(\sqrt{N})$
  - Processing N items is $O(N\sqrt{N})$
- Can we do better ?
- Maintain a special binary tree - heap
  - Height $O(\log N)$
  - insert() is $O(\log N)$
  - delete_max() is $O(\log N)$
  - processing N items is $O(N \log N)$
- Flexible - need not fix N in advance

Feb 2, 2022

PAGE NO.
DATE:

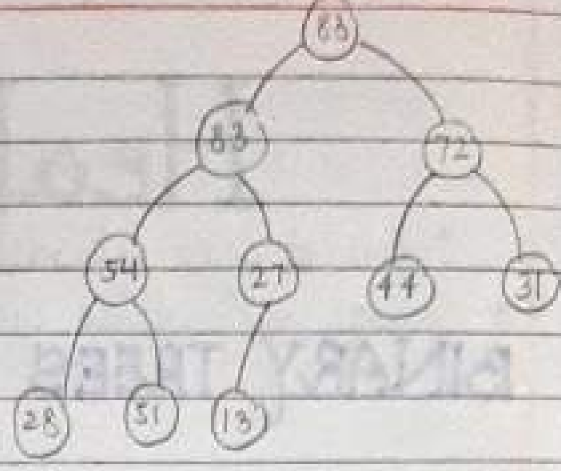
# Heaps

## BINARY TREES

- Values are stored as nodes in a rooted tree
- Each node has up to two children
  - → left child & right child
  - → order is important
- Other than the root, each node has a unique parent
- Leaf node - no children
- Size - number of nodes
- Height - number of levels

## HEAP

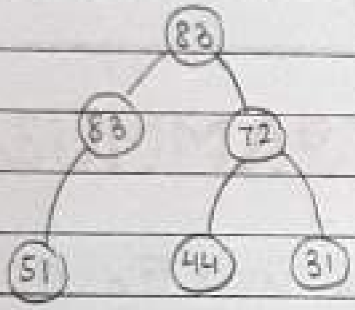
- Binary tree filled level by level, left to right
- The value at each node is at least as big the values of its children
  - → max-heap



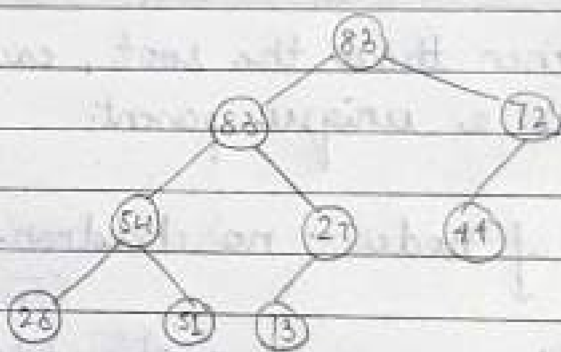

- Binary tree on the right is an example of a heap
- Root always has the largest value
  → By induction, because of max-heap property

## NON-EXAMPLES

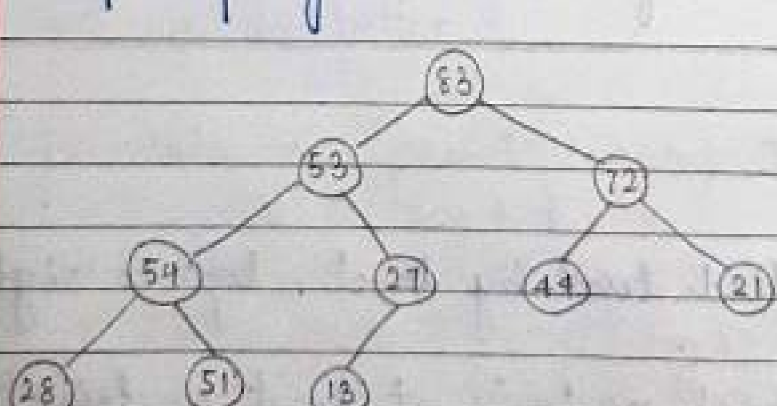
(1) No "holes" allowed




(2) Cannot leave a level incomplete




(3) Heap Property is violated



# insert ( )

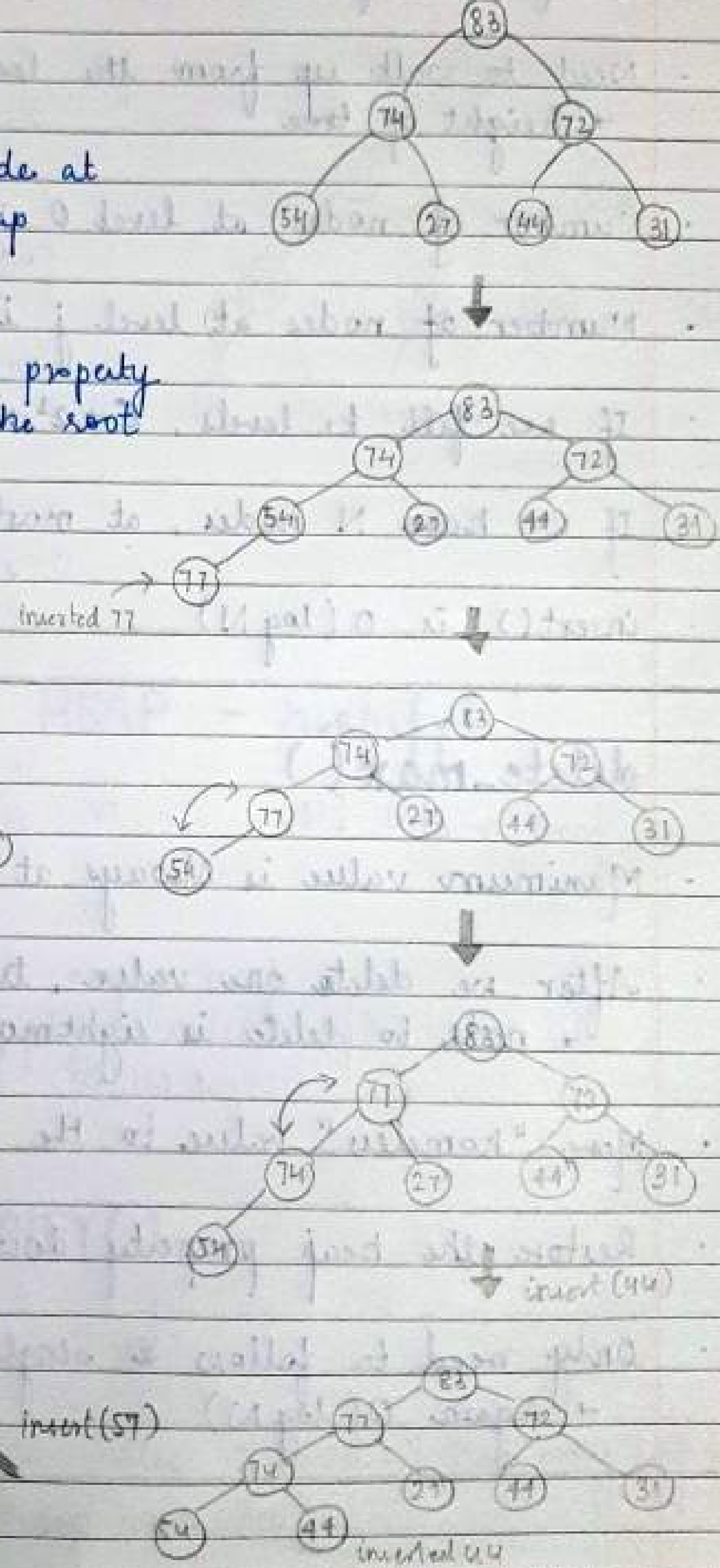
→ insert (77)

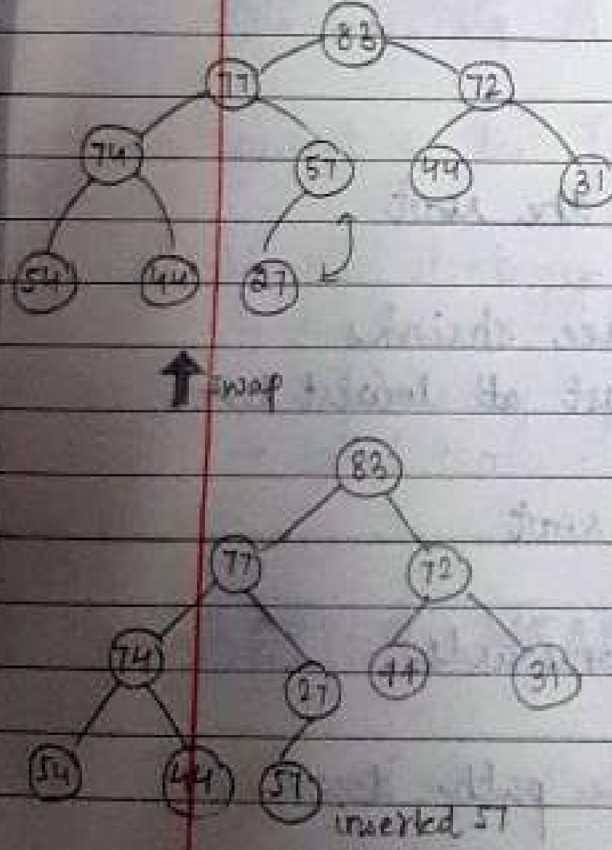
→ Add a new node at dictated by heap structure

→ Restore the heap property along path to the root

→ insert (44)

→ insert (57)



PAGE NO.
DATE:

# COMPLEXITY OF INSERT()

- Need to walk up from the leaf to the root
  → height of tree
- Number of nodes at level 0 is $2^0 = 1$
- Number of nodes at level j is $2^j$
- If we fill k levels, $2^0 + 2^1 + \dots + 2^{k-1} = 2^k - 1$ nodes
- If we have N nodes, at most $1 + \log N$ levels
- insert() is $O(\log N)$

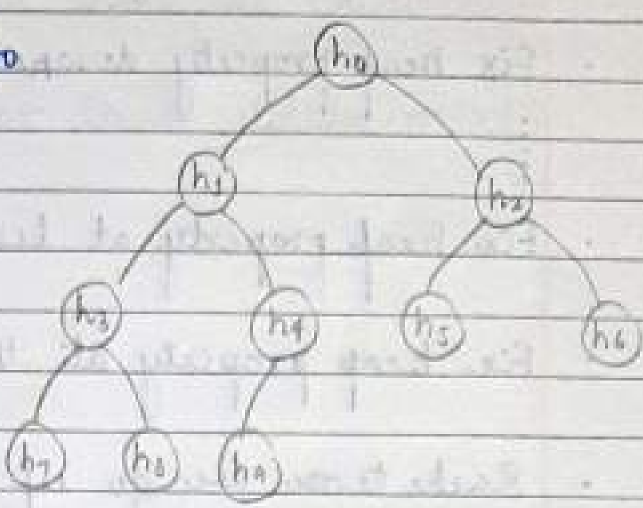
---

## delete_max()

- Maximum value is always at the root
- After we delete one value, tree shrinks
  → node to delete is rightmost at lowest level
- Move "homeless" value to the root
- Restore the heap property downwards
- Only need to follow a single path down
  → Again $O(\log N)$

# IMPLEMENTATION

- Number the nodes top to bottom left right




- Store as a list
  H = [h0, h1, ..., h9]

- Children of H[i] are at
  H[2*i+1], H[2*i+2]

- Parent of H[i] is at
  H[(i-1)//2] for i>0

---

# BUILDING A HEAP – heapify()

- Convert a list [v0, v1, ... , vN] into a heap

- Simple strategy
  → start with an empty heap
  → repeatedly apply insert(vj)
  → total time is O(N logN)

---

# BETTER HEAPIFY()

- List L = [v0, v1, ...., vN]

- mid = len(L) // 2, slice L[mid:] has only leaf nodes (already satisfy heap condition)

- Fix heap property downwards for second last level
- Fix heap property downwards for third last level

⋮

- Fix heap property at level 1
- Fix heap property at the root
- Each time we go up one level, one extra step per node to fix heap property
- However, number of nodes to fix halves
- Second last level, $n/4 \times 1$ setps
- Third last level, $n/8 \times 2$ steps
- Fourth last level, $n/16 \times 3$ steps

⋮

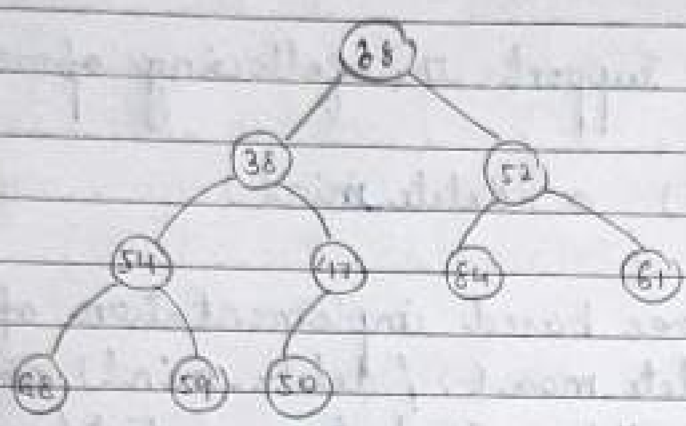
- Cost turns out to be $O(n)$.

---

## → SUMMARY ←

- Heaps are a tree implementation of priority queues
  - → insert () is $O(\log N)$
  - → delete_max () is $O(\log N)$
  - → heapify() builds a heap in $O(N)$

- Can invert the heap condition
  - → Each node is smaller than its children
  - → min-heap
  - → delete-min() rather than delete-max().



February 3, 2022

PAGE NO.
DATE:

# Using Heaps In Algo.

## PRIORITY QUEUES & HEAPS

- Priority queues support the following operations
  - → insert()
  - → delete_max() or delete_min()

- Heaps are a tree based implementation of priority queu
  - → insert(), delete_max()/delete_min() are both O(logN
  - → heapify() builds a heap from a list/array in time O(N)

- Heap can be represented as a list/array
  - → simple index arithmetic to find parent & children of a node

- What more do we need to use a heap in an algo?

---

## DIJKSTRA'S ALGORITHM

- Maintain two dictionaries with vertices as keys
  - → <u>visited</u>, initially <u>False</u> for all <u>v</u>
  - → <u>distance</u>, initially <u>infinity</u> for all <u>v</u>

- Set distance[s] to 0

- Repeat, until all reachable vertices are visited

→ Find unvisited vertex nextv with minimum distance
→ Set visited [nextv] to True
→ Recompute distance [v] for every neighbour v of nextv

**Code :**

```
def dijkstra(WMat, s):
    (rows, cols, x) = WMat.shape
    infinity = np.max(WMat) * rows + 1
    (visited, distance) = ({}, {})
    for v in range(rows):
        (visited[v], distance[v]) = (False, infinity)
    distance[s] = 0
    for u in range(rows):
        nextd = min([distance[v] for v in range(rows)
                          if not visited[v]])
        nextvlist = [v for v in range(rows)
                        if (not visited[v]) and
                           distance[v] == nextd]

        if nextvlist == []:
            break
        nextv = min(nextvlist)
        visited[nextv] = True
        for v in range(cols):
            if WMat[nextv, v, 0] == 1 and (not visited[v]):
                distance[v] = min(distance[v], distance[nextv]
                                  + WMat[nextv, v, 1])

    return (distance)
```

# Bottleneck

→ Find unvisited vertex j with minimum distance
  · Naive implementation requires an O(n) scan

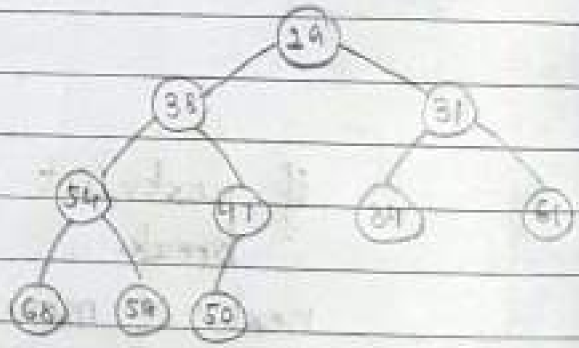
→ Maintain unvisited vertices as a min-heap
  · delete_min() in O(log n) time

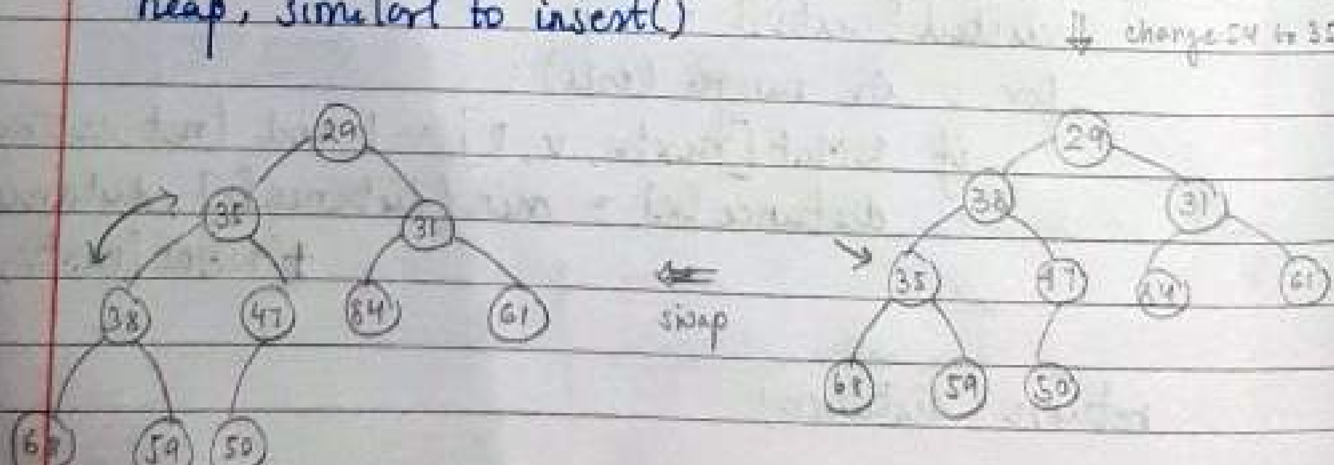
→ But also need to update distances of neighbours
  · Unvisited neighbours' distances are inside the min-heap
  · Updating a value is not a basic heap operation

---

## UPDATING VALUES IN A MIN-HEAP

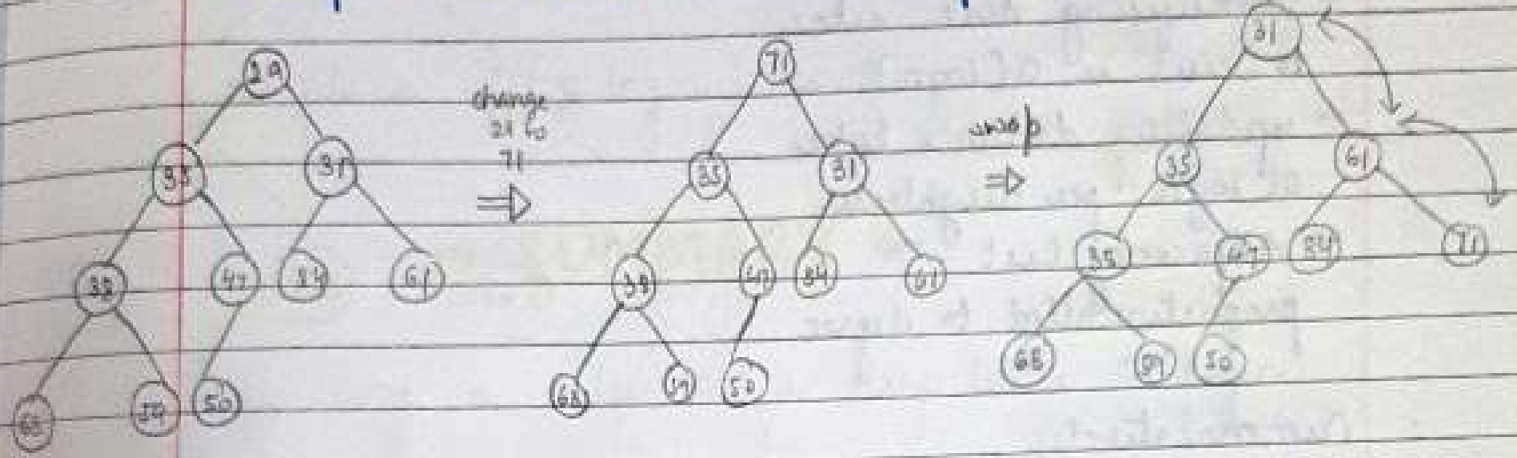
· Changing 54 to 35
→ reducing a value can create violation with parent
→ swap upwards to restore heap, similar to insert()





PAGE NO.
DATE:

- Changing 29 to 71
  → Increasing a value can create a violation with child
  → swap downwards to restore heap, similar to delete_min()




- Both updates are O(log n)!
  → Are we done?

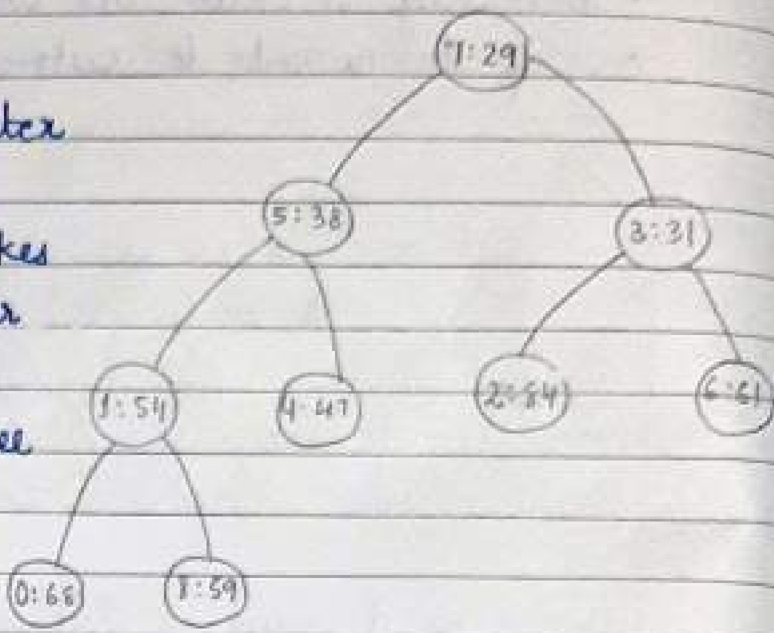
- Locate the node to update?

- Maintain 2 additional dictionaries
  → Vertices are {0, 1, ...., n-1}
  → Heap positions are {0, 1, ...., n-1}
  → VtoH maps vertices to heap positions
  → HtoV maps heap positions to vertices

- Update node 1 to 35

- Update VtoH and HtoV each time we swap values in the heap

# DIJKSTRA's ALGORITHM




- Using min-heaps
  - → identifying next vertex to visit is $O(\log n)$
  - → updating distance takes $O(\log n)$ per neighbour
  - → Adjacency List - proportionally to degree
- Cummulatively
  - → $O(n \log n)$ to identify vertices to visit across n iterations.
  - → $O(m \log n)$ distance updates overall

V to H

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 7 | 3 | 5 | 2 | 4 | 1 | 6 | 0 | 8 |

H to V

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 7 | 5 | 3 | 1 | 4 | 2 | 6 | 0 | 8 |

- Overall $O((m+n) \log n)$

# HEAP SORT

- Start with an unordered list
- Build a heap - $O(n)$
- Call delete_max() n times to extract elements in descending order - $O(n \log n)$

- After each delete_max(), heap shrinks by 1
- Store maximum value at the end of current heap
- In place $O(n \log n)$ sort

## → SUMMARY ←

- Updating a value in a heap takes $O(\log n)$
- Need to maintain additional pointers to map values to heap positions and vice-versa
- With this extended notion of heap, Dijkstra's algo complexity improves from $O(n^2)$ to $O((m+n) \log n)$
- In a similar way, improve Prim's algorithm to $O((m+n) \log n)$
- Heaps can also be used to sort a list in place in $O(n \log n)$

Feb 4, 2022

# Search Trees

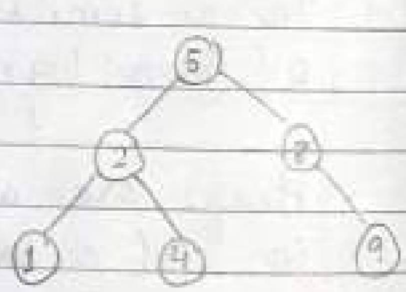
## DYNAMIC SORTED DATA

- Sorting is useful for efficent searching
- What if the data is changing dynamically?
  - → Items are periodically inserted & deleted
- Insert / delete in a sorted list takes time $O(n)$
- Move to a tree structure, like heaps for priority queues.

---

## BINARY SEARCH TREE

- For each node with value v
  - → All values in the left subtree are $< v$
  - → All values in the right subtree $> v$




- No duplicate values

<u>Implementing a Binary Search Tree</u>

- Each node has a value and pointers to its children
- Add a frontier with empty nodes, all fields –

PAGE NO.
DATE:

→ Empty tree is a single empty node
→ Leaf node to points to empty nodes

- Easier to implement operations recursively

## The class Tree

- Three local fields, value, left, right
- Value None for empty value-
- Empty true has all fields None
- Leaf has a nonempty value and empty left and right

**Code** :

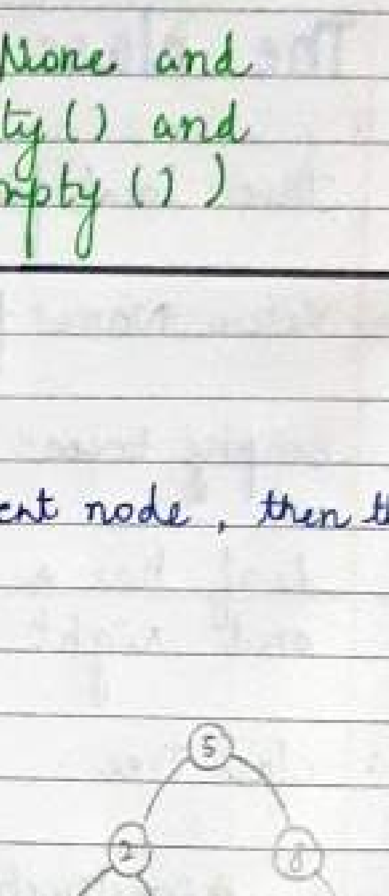
```
class Tree :

    # Constructor
    def __init__ (self, initval = None) :
        self.value = initval
        if self.value :
            self.left = Tree()
            self.right = Tree()
        else :
            self.left = None
            self.right = None

        return
```

```
# Only empty node has value None
def isempty (self):
    return (self.value == None)

# Leaf nodes have both children empty
def isleaf(self):
    return ( self.value != None and
             self.left.isempty() and
             self.right.isempty() )
```

---

# Inorder Traversal

- List the left subtree, then the current node, then the right subtree
- Lists values in sorted order
- Use to print the tree




```
class Tree:
    .....
    # Inorder Traversal
    def inorder (self):
        if self.isempty():
            return ([])
        else:
            return (self.left.inorder() + [self.value] +
                    self.right.inorder())
```

```
# Display Tree as a string
def __str__ (self):
    return (str(self.inorder()))
```

# Find A Value v

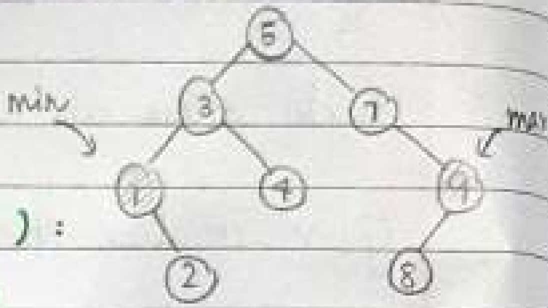
- check value at current node
- If v smaller than current node, go left
- If v ~~smaller~~ greater than current node, go right
- Natural generalization of binary search

```
class Tree:
    ....
    # check if value v occurs in tree
    def find(self, v):
        if self.isempty():
            return(False)

        if self.value == v:
            return(True)

        if v < self.value:
            return(self.left.find(v))

        if v > self.value:
            return(self.right.find(v))
```

# Minimum And Maximum

- Minimum is left most node in the tree
- Maximum is right most node in the tree

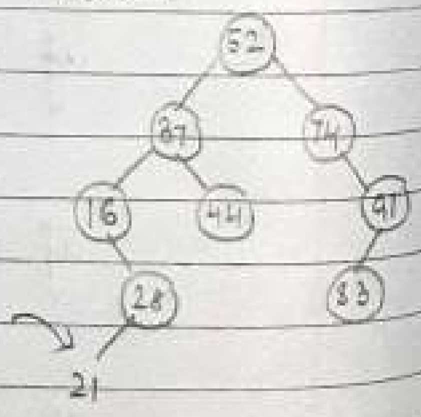
```
class Tree :
    ....
    def minval (self) :
        if self. left. isempty ( ) :
            return (self. value)
        else :
            return ( self. left. minval ( ))

    def maxval (self) :
        if self. right. isempty () :
            return (self. value)
        else :
            return ( self. right. maxval ( ))
```




---

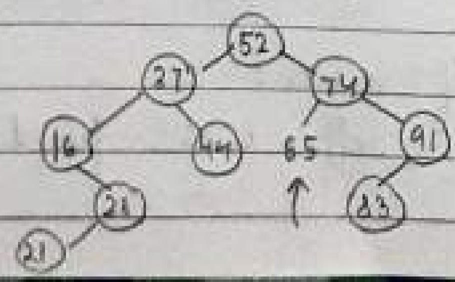
# Insert a value v

Insert 21




- Try to find v
- Insert at the position where <u>find</u> fails.

Insert 65



PAGE NO.
DATE:

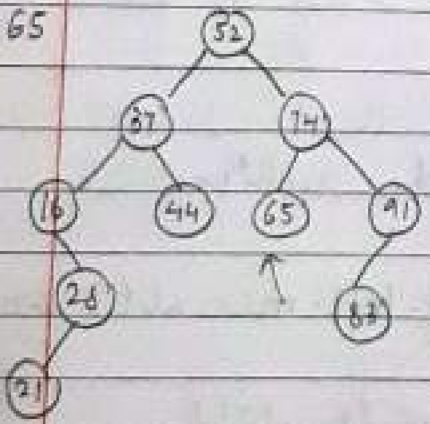
```
class Tree :
    ....
    def insert (self, v) :
        if self.isempty () :
            self.value = v
            self.left = Tree()
            self.right = Tree()

        if self.value == v :
            return

        if v < self.value :
            self.left.insert (v)
            return

        if v > self.value :
            self.right.insert (v)
            return
```

# Delete a value v

- If v is present, delete
- Leaf node ? No problem
- If only one child, promote the subtree
- Otherwise, replace v with self.left.maxval() and delete self.left.maxval()
  → self.left.maxval() has no right child

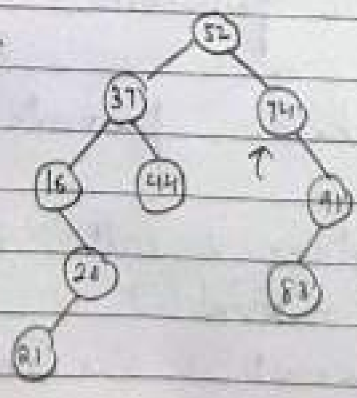
```
class Tree :
    ....
    def delete (self, v) :
        if self. isempty( ):
            return
        if v < self. value :
            self. left. delete (v)
            return
        if v > self. value :
            self. right . delete (v)
            return
        if v == self. value :
            if self. isleaf ( ):
                self. makeempty ( )
            elif self. left . isempty ( ) :
                self. copyright ( )
            elif self. right. isempty ( ) :
                self. copyleft ( )
            else :
                self. value = self. left . maxval ( )
                self. left. delete (self . left. maxval ( ))
        return
```

Delete 65

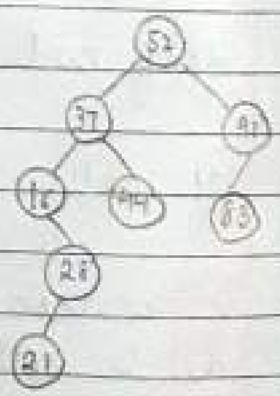



⇒

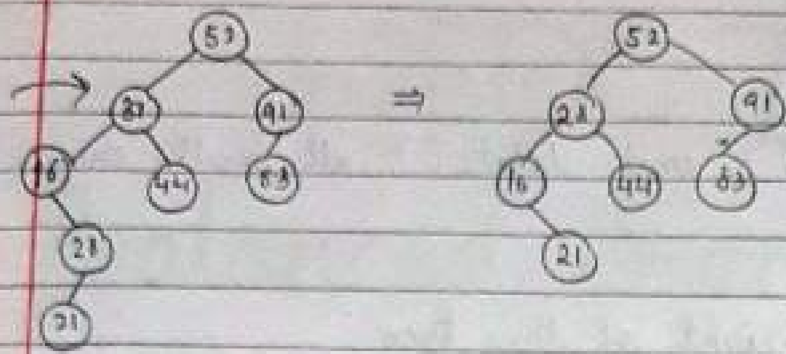
Delete 74




⇒



Delete 37




```
# Convert leaf node to empty node
def makeempty(self):
    self.value = None
    self.left = None
    self.right = None
    return
```

```
# Promote left child
def copyleft(self):
    self.value = self.left.value
    self.right = self.left.right
    self.left = self.left.left
    return
```

```
# Promote right child
def copyright(self):
    self.value = self.right.value
    self.left = self.right.left
    self.right = self.right.right
    return
```

# Complexity

- find(), insert() and delete() all walk down a single path
- Worst-case : height of the tree
- An unbalanced tree with n nodes may have height O(n)
- Balanced trees have height O(log n)
- Will see how to keep a tree balanced to ensure all operations remain O(log n).