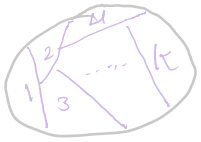
Week 6

Thursday, July 17, 2025 3:01 PM

Union find Data Structure:

Set S has components $\{C_1, \cdots C_k\}$



Has 3 steps → i) Make union find (S) ⇒ Make singleton components for each $s \in S$.

ii) Find(s) ⇒ Return components having s.

iii) Union(s,s') => Merge components having s and s'.

For the Basic implementation; the complexities are →

→ Make union find(S) ⇒ $O(n)$

→ find(i) ⇒ $O(1)$

→ Union(i,j) ⇒ $O(n)$

→ for sequence of m values, Union operation takes time of ⇒ $O(m \times n)$

For the improved implementation, the complexities are:

→ Make union find (S) ⇒ $O(n)$

→ Find(i) ⇒ $O(1)$

→ Union (i,j) ⇒ $O(\log n)$

★ This new data structure is used for improving Kruskal's Algorithm.

Complexity

- Tree has $n-1$ edges, so $O(n)$ Union() operations
- $O(nlogn)$ amortized cost, overall
- Sorting E takes $O(mlogm)$
  - Equivalently $O(mlogn)$, since $m \leq n^2$
- Overall time, $O((m+n)logn)$

When Kruskal's Algorithm is implemented using [Make Union Find] as data-structure, the overall complexity is reduced.

Make Union Find naive / basic code → Aim is to find the group/component an element belongs to, and also group elements into sets.

```
class MakeUnionFind:
    def __init__(self):
        self.components = {}
        self.size = 0
    def make_union_find(self,vertices):
        self.size = vertices
        for vertex in range(vertices):
            self.components[vertex] = vertex
    def find(self,vertex):
        return self.components[vertex]
    def union(self,u,v):
        c_old = self.components[u]
        c_new = self.components[v]
        for k in range(self.size):
            if self.components[k] == c_old:
                self.components[k] = c_new
```

Initial Constructor → size will keep track of vertices present (in number)

Initialization
- size is number of vertices
- every vertex is its own parent.

Find function → Tells which component a vertex belongs to.

Merge operation

Here, we are checking if an element belongs to the old group (u), then it will be sent to the new group (v).

Trying to merge (union) group(u) to group (v).
- group (v) → location of v (c_new)
- group (u) → location of u (c_old)

Say for 4 vertices,
self.components = {
0 : 0,
1 : 1,
2 : 2,
3 : 3
}

Make Union Find (Improved Code): The aim is to make the algorithm faster.

```
class MakeUnionFind:
    def __init__(self):
        self.components = {}
        self.members = {}
        self.size = {}
    def make_union_find(self,vertices):
        for vertex in range(vertices):
            self.components[vertex] = vertex
            self.members[vertex] = [vertex]
            self.size[vertex] = 1
    def find(self,vertex):
        return self.components[vertex]
```

Defining class

self.components → tells the group a node belongs to.

self.members → list of members of each group.

self.size → size of group.

every vertex is it's own parent.

every group has only 1 member, the number itself.

Which group the vertex currently belongs to.

```
    def find(self,vertex):
        return self.components[vertex]
```
→ Which group the vertex currently belongs to.

```
    def union(self,u,v):
        c_old = self.components[u]
        c_new = self.components[v]
        # Always add member in components which have greater size
        if self.size[c_new] >= self.size[c_old]:
            for x in self.members[c_old]:
                self.components[x] = c_new
                self.members[c_new].append(x)
                self.size[c_new] += 1
        else:
            for x in self.members[c_new]:
                self.components[x] = c_old
                self.members[c_old].append(x)
                self.size[c_old] += 1
```
{ merge (union) u with v (u in c_old & v in c_new)

{ merge based on size

{ if size(v) >= size(u), move every element of u in v and update accordingly.

{ if size(u) > size(v), move every element of v in u and update accordingly.

⇒ called Union by size.

⇒ Faster, as complexity of Union: $O(\log n)$.

Improved Kruskal's Algorithm:

```
def kruskal(WList):
    (edges,TE) = ([],[])
    for u in WList.keys():
        edges.extend([(d,u,v) for (v,d) in WList[u]])
    edges.sort()
```
→ edges are the graph edges and TE stores edges of spanning tree.

→ edges list sorted by weight. (Least wt. first)

```
    mf = MakeUnionFind()
    mf.make_union_find(len(WList))
```
→ union find for every vertex. To check if there's a cycle.

```
    for (d,u,v) in edges:
        if mf.components[u] != mf.components[v]:
            mf.union(u,v)
            TE.append((u,v,d))
```
→ if u & v are in different groups we can connect them.

→ add edge to spanning tree.

```
        # We can stop the process if the size becomes equal to the total number of vertices
        # Which represent that a spanning tree is completed

        if mf.size[mf.components[u]]>= mf.size[mf.components[v]]:
            if mf.size[mf.components[u]] == len(WList):
                break
        else:
            if mf.size[mf.components[v]] == len(WList):
                break
    return(TE)
```
Early stopping criteria.

{ checking if entire graph is connected or not. (one component must have all nodes)

→ return the minimum spanning tree. (edges of the MST)

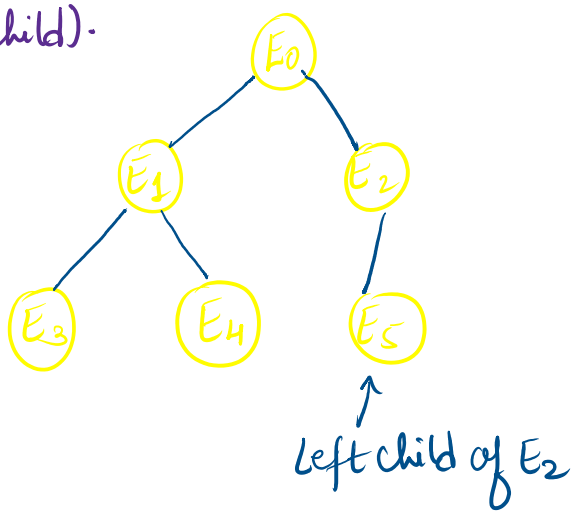
Priority Queue: Need to maintain collection of items with priority for optimizing operations like delete max() & insert().

Priority Queue: Need to maintain collection of items [illegible] optimizing operations like delete max() & insert().

delete max(): Identifying & removing the element with the highest priority.

insert(): Adding a new item to the list.

Binary tree: A data structure where each node can contain atmost 2 children. (Left child & right child).



HEAP: A binary tree filled level by level, from left to right. (Implementation of Priority Queue)

Two types ⇒ (1) Max heap: For any node other than leaf node, the value must be greater than or equal to child nodes.

left child of H[i] = H[2 * i + 1]

Right child of H[i] = H[2 * i + 2]

Parent of H[i] = H[(i-1) // 2], for i > 0

(2) Min heap: For any node other than leaf node, the value must be lesser than or equal to child nodes.

Heap has 2 methods of being created.

(1) Insert the keys one-by-one. [O(n log n)]

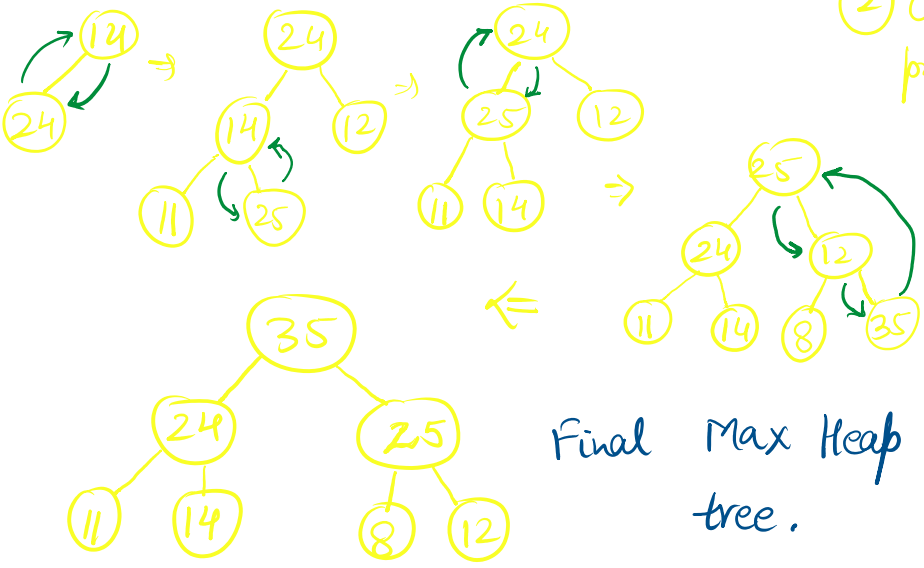
(2) Heapify method [O(n)]

Create Heap → (1) Creating a max heap using insertion method.

Qns. Keys = 14, 24, 12, 11, 25, 8, 35

(1) Insert level by level, left to right [O(1)]

Qus. Keys = 14, 24, 12, 11, 25, 8, 35



① Insert level by level, left to right [O(1)]

② Compare and swap to ensure the properties of max heap are followed. [O(n log n)]

for n elements ↓ for comparisions

Final Max Heap tree.

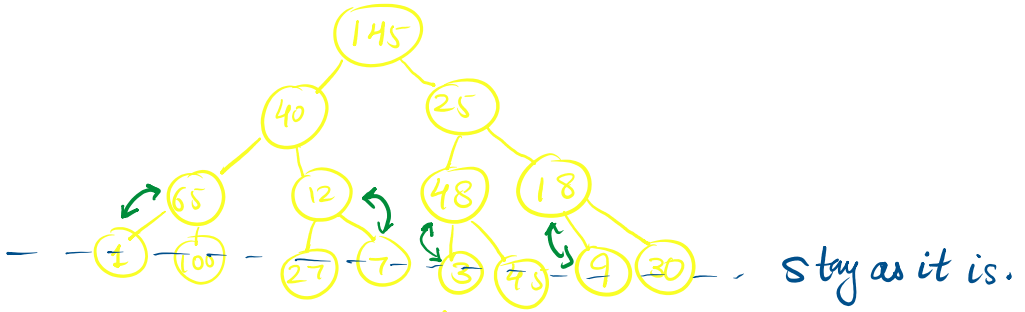
Qus. Using Heapify method. Keys = 145, 40, 25, 65, 12, 48, 18, 1, 100, 27, 7, 3, 45, 9, 30 create a min-heap.

① Create a heap with given keys. Level by level left to right.

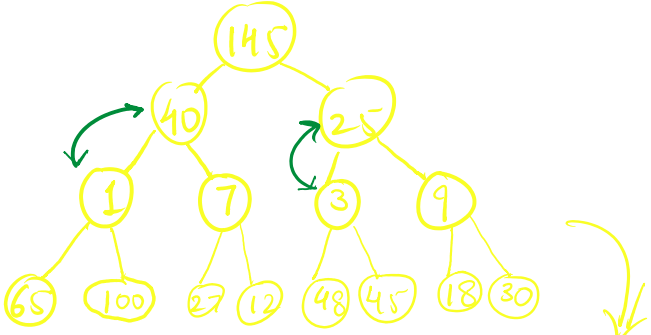
② Last row stays as it is.

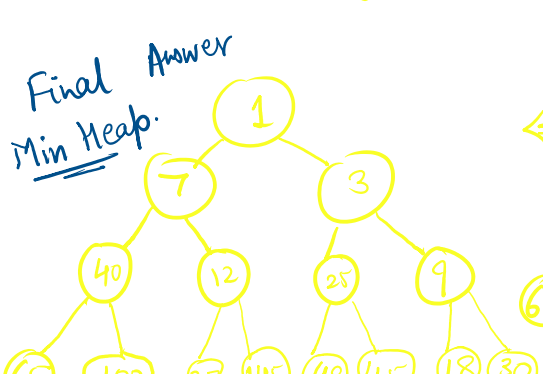
For others, start doing swapping to get min element at parent node. Keep doing until you have min heap

⇒ For a node at level n, the possible no. of swaps is log n.

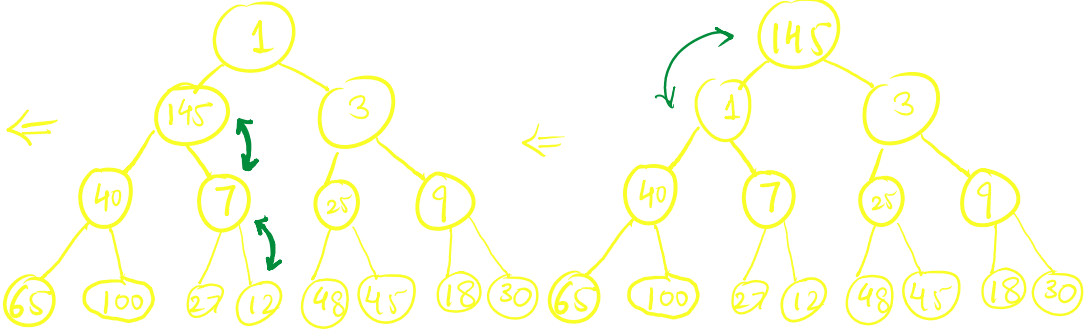


stay as it is.





Final Answer
Min Heap.



Max Heap Implementation:

→ Creating class

```
class maxheap:
    def __init__(self):
        self.A = []
```
→ Initializing constructor (a list)

```
    def max_heapify(self,k):
        l = 2 * k + 1
        r = 2 * k + 2
        largest = k
        if l < len(self.A) and self.A[l] > self.A[largest]:
            largest = l
        if r < len(self.A) and self.A[r] > self.A[largest]:
            largest = r
        if largest != k:
            self.A[k], self.A[largest] = self.A[largest], self.A[k]
            self.max_heapify(largest)
```
→ left child index
→ right child index
→ largest node
→ updating largest

Comparing left and right child value to the largest one (if left & right child exist)

if child > percent, make the swap.
recursive heapify on subtree.

```
    def build_max_heap(self,L):
        self.A = []
        for i in L:
            self.A.append(i)
        n = int((len(self.A)//2)-1)
        for k in range(n, -1, -1):
            self.max_heapify(k)
```
→ Calling heap list
→ appending each element in the list to heap.

for non leaf nodes, check for the violation of max heap rules and create proper max heap. (Bottom up approach)

```
    def delete_max(self):
        item = None
        if self.A != []:
            self.A[0],self.A[-1] = self.A[-1],self.A[0]
            item = self.A.pop()
            self.max_heapify(0)
        return item
```
→ Remove and return max.
→ if not empty

swap the max value (root) with last element.
Remove & return last element.
Heapify from root node.

```
    def insert_in_maxheap(self,d):
        self.A.append(d)
        index = len(self.A)-1
        while index > 0:
            parent = (index-1)//2
            if self.A[index] > self.A[parent]:
                self.A[index],self.A[parent] = self.A[parent],self.A[index]
                index = parent
            else:
                break
```
→ Insert new element in heap.
→ get the index of last element.
→ Parent of new element.

add to last of list

if child is greater than parent, do a swap until heap properties satisfy.

Min heap Implementation:

→ Defining min heap class

```
class minheap:
    def __init__(self):
        self.A = []
    def min_heapify(self,k):
```
→ Heap as list

```
    def __init__(self):
        self.A = []
    def min_heapify(self,k):
        l = 2 * k + 1
        r = 2 * k + 2
        smallest = k
        if l < len(self.A) and self.A[l] < self.A[smallest]:
            smallest = l
        if r < len(self.A) and self.A[r] < self.A[smallest]:
            smallest = r
        if smallest != k:
            self.A[k], self.A[smallest] = self.A[smallest], self.A[k]
            self.min_heapify(smallest)
```

Heap as list

left child index

Right child index

Parent as smallest

If child exist and smaller than parent, assign child as smallest.

swap the child and parent

If Parent not smallest

Call heapify on the remaining sub tree.

```
def build_min_heap(self,L):
        self.A = []
        for i in L:
            self.A.append(i)
        n = int((len(self.A)//2)-1)
        for k in range(n, -1, -1):
            self.min_heapify(k)
```

For all in list, append to heap.

from non leaf nodes.

Bottom-up apply min heap logic.

```
def delete_min(self):
        item = None
        if self.A != []:
            self.A[0],self.A[-1] = self.A[-1],self.A[0]
            item = self.A.pop()
            self.min_heapify(0)
        return item
```

Delete and return min value

For non-empty heap, swap root and last element, then return last and heapify the remaining heap.

Return min value

Complexity

Heaps are a tree implementation of priority queues

- insert() is $O(logN)$
- delete max() is $O(logN)$
- heapify() builds a heap in $O(N)$

```
def insert_in_minheap(self,d):
        self.A.append(d)
        index = len(self.A)-1
        while index > 0:
            parent = (index-1)//2
            if self.A[index] < self.A[parent]:
                self.A[index],self.A[parent] = self.A[parent],self.A[index]
                index = parent
            else:
                break
```

Add new element to last.

index of last

Parent of last element

If last is smaller than parent, swap else stop.

Improved Dijkstra's Algorithm using min-heap (by adjacency list) →

⇒ HtoV(i): Heap index i has vertex v of dist d.

⇒ VtoH(v): Vertex v at heap index i.

```
def min_heapify(i,size):
    lchild = 2*i + 1
    rchild = 2*i + 2
    small = i
    if lchild < size-1 and HtoV[lchild][1] < HtoV[small][1]:
        small = lchild
    if rchild < size-1 and HtoV[rchild][1] < HtoV[small][1]:
        small = rchild
    if small != i:
        VtoH[HtoV[small][0]] = i
        VtoH[HtoV[i][0]] = small
        (HtoV[small],HtoV[i]) = (HtoV[i], HtoV[small])
        min_heapify(small,size)
```

Implement of min-heap

Maintain V to H mapping.

update smallest of parent and child. Recursively for entire tree

```
def create_minheap(size):
    for x in range((size//2)-1,-1,-1):
        min_heapify(x,size)
```

Bottom-up approach to heapify into min heap, for the non-leaf nodes.

```
def create_minheap(size):
    for x in range((size//2)-1,-1,-1):
        min_heapify(x,size)
def minheap_update(i,size):
    if i!= 0:
        while i > 0:
            parent = (i-1)//2
            if HtoV[parent][1] >  HtoV[i][1]:
                VtoH[HtoV[parent][0]] = i
                VtoH[HtoV[i][0]] = parent
                (HtoV[parent],HtoV[i]) = (HtoV[i], HtoV[parent])
            else:
                break
            i = parent
def delete_min(hsize):
    VtoH[HtoV[0][0]] = hsize-1
    VtoH[HtoV[hsize-1][0]] = 0
    HtoV[hsize-1],HtoV[0] = HtoV[0],HtoV[hsize-1]
    node,dist = HtoV[hsize-1]
    hsize = hsize - 1
    min_heapify(0,hsize)
    return node,dist,hsize
HtoV, VtoH = {},{}
#global HtoV map heap index to (vertex, distance from source)
#global VtoH map vertex to heap index
```

Bottom-up approach to heapify into min heap, for the non-leaf nodes.

For non root nodes → Updating the distances

Check if the dist of child is smaller than the parent. If so, do a swap. update VtoH & HtoV. Maintain min-heap properties.

For root node → Remove the min element.

Return min element, it's distance value and heap size.

```
def dijkstralist(WList,s):
    infinity = float('inf')
    visited = {}
    heapsize = len(WList)
    for v in WList.keys():
        VtoH[v]=v
        HtoV[v]=[v,infinity]
        visited[v] = False
    HtoV[s]= [s,0]
    create_minheap(heapsize)

    for u in WList.keys():
        nextd,ds,heapsize = delete_min(heapsize)
        visited[nextd] = True
        for v,d in WList[nextd]:
            if not visited[v]:
                HtoV[VtoH[v]][1] = min(HtoV[VtoH[v]][1],ds+d)
                minheap_update(VtoH[v],heapsize)
```

Dist to all vertices is taken infinity

Total vertices → visited list.

setup VtoH & HtoV dictionary. Mark all nodes un-visited.

Source distance = 0 → create min heap

Remove node with min dist, mark it visited.

For every neighbor of current node, try to update its distance. Fix the position using min-heap conditions.

Updated complexity →

Using min-heaps:-

- Identifying next vertex to visit is $O(logn)$
- Updating distance takes $O(logn)$ per neighbor
- Adjacency list – proportionally to degree

Cumulatively:-

- $O(nlogn)$ to identify vertices to visit across n iterations
- $O(mlogn)$ distance updates overall
- Overall $O((m+n)logn)$ Improved Complexity

Heap SORT → Based on "Deleting min/max value from min/max heap"

→ Insert all the elements in a heap.

→ Start deleting the min/max value and place that in a different list.

→ Start deleting the min/max value and place that in a different list.

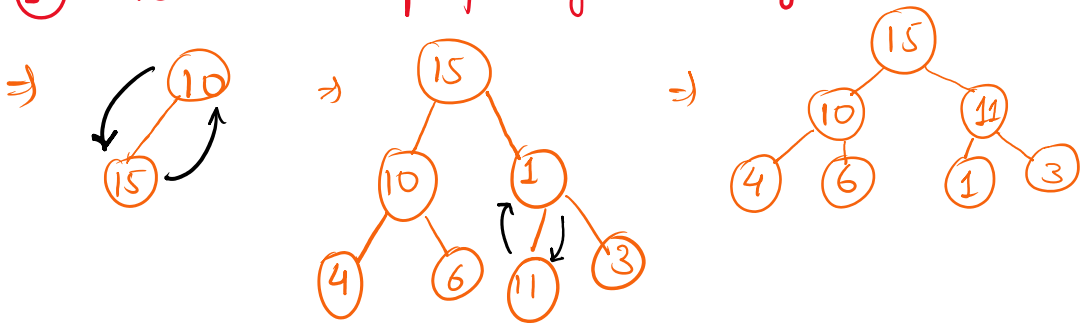
→ Heapify the remaining heap and then again delete and store, until only 1 element is left in the heap.

→ Place that last element into the list. We have a sorted list.

⇒ THIS IS HEAP SORT.

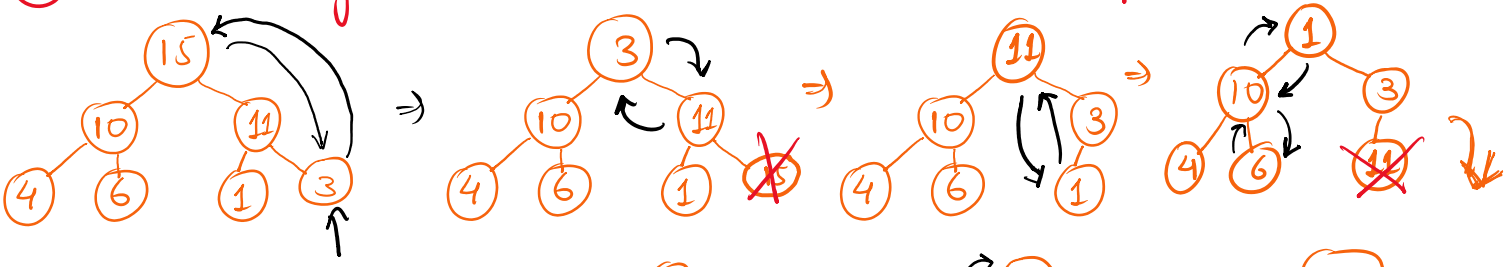
Qns. Keys = [10, 15, 1, 4, 6, 11, 3]. Sort using heap sort.

① create max heap from given keys



The required max heap.

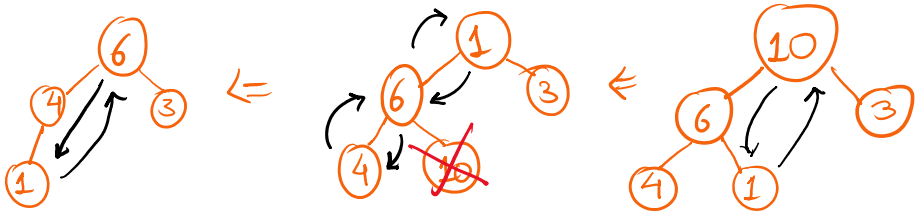
② Start deleting the max and then store it in separate list.



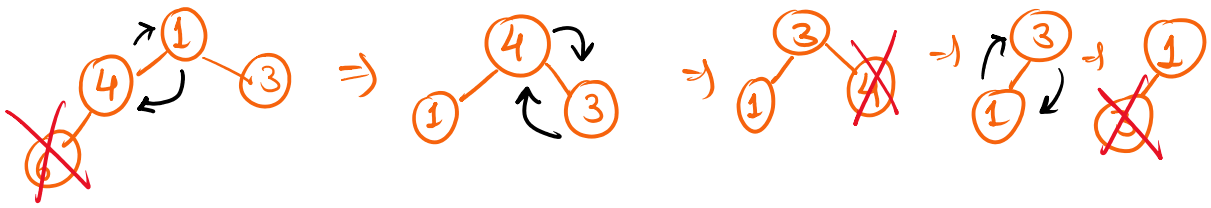
Remove it.

(Descending)

Sorted list = 15, 11, 10, 6, 4, 3, 1



Last left = ~~1~~



→ Max heap returns descending sorted list and min heap returns ascending sorted list.

Heap sort implementation →

```
def max_heapify(A,size,k):
    l = 2 * k + 1
    r = 2 * k + 2
    largest = k
    if l < size and A[l] > A[largest]:
        largest = l
    if r < size and A[r] > A[largest]:
        largest = r
    if largest != k:
        (A[k], A[largest]) = (A[largest], A[k])
        max_heapify(A,size,largest)
```

⇒ Function defining logic of max heap.

```
def build_max_heap(A):
    n = (len(A)//2)-1
    for i in range(n, -1, -1):
        max_heapify(A,len(A),i)
```

⇒ Creating max heap for a given list of elements.

Defining heapsort →

```
def heapsort(A):
    build_max_heap(A) → creating max heap
    n = len(A)
    for i in range(n-1,-1,-1):
        A[0],A[i] = A[i],A[0]
        max_heapify(A,i,0)
```

Heapify the entire list → } For non leaf nodes, swap with root element.

Complexity of Heap sort ⇒

Complexity

- Start with an unordered list
- Build a heap – $O(n)$
- Call delete max() n times to extract elements in descending order – $O(nlogn)$
- After each delete max(), heap shrinks by 1
- Store maximum value at the end of current heap
- In place $O(nlogn)$ sort

→ Inplace sorting.

→ Not stable.

Not stable though!

BINARY SEARCH TREE: Aims to improve the time complexity of deletion and insertion.

for a tree to be binary search tree, i) All left subtrees of node V, must have value less than V.

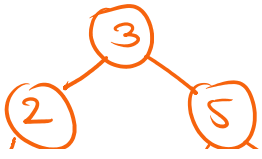
ii) All right subtrees of node V, must have value greater than V.

Assume all nodes are unique.

Search in BST takes ⇒ O(height) time.

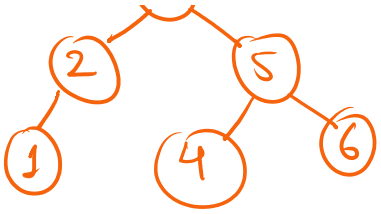
→ For balanced BST, height = log(n)

Making a BST → Nodes = [3, 2, 1, 5, 6, 4]



① 1st element as root node.

② Follow the property of BST for inserting a new node,



for inserting a new node, starting comparisions from the root node only.

For searching in a BST, start comparing with root node value V.

a) Searching value < V → move to left sub tree.

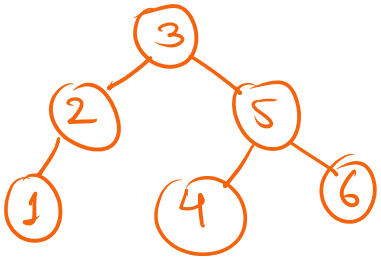
b) Searching value > V → move to right subtree.

→ Move to the next node (one node at a time).

→ Follow the same steps as above.

Deleting a node from a BST → 1. search for the node to be deleted.

2. Delete the node, rearrange the tree, keeping the tree property intact



For such a tree we have 3 type of nodes.

i) with no child node (ex ⑥)

ii) with 1 child node (ex ②)

iii) with 2 child nodes (ex ⑤)

Case i) → Delete the node, return null.

Case ii) → Delete the node value, and replace in it, the value of not null child

Here; Delete 2, and replace 1 in it

Case iii) → Find the "In-order successor" of the node to be deleted.

→ Left most child of right sub tree.

→ In-order successor has no child of it's own

→ In-order successor has no child of it's own

Delete the node value and replace with the in-order successor.

Clear the in-order successor's node.

(Properties of BST will now hold)

Implementation of BST →

```
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
```

→ If has some initial values then create null left and right subtrees else empty node.

```
    # 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())
```

→ check empty node

Check leaf node. (one that doesn't have a child node)

```
    # 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()))
    # 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))
```

→ In order traversal is left, root, right.

→ Displaying as string.

→ Search functionality.

Compare with root element, based on that move to left or right subtree.

```
    # return minimum value for tree rooted on self - Minimum is left most node in the tree
    def minval(self):
        if self.left.isempty():
            return(self.value)
        else:
            return(self.left.minval())
```

Recursively reach to min value (Left-most value)

```
    # return max value for tree rooted on self  - Maximum is right most node in the tree
    def maxval(self):
        if self.right.isempty():
            return(self.value)
        else:
            return(self.right.maxval())
```

Recursively reach to max value (Right-most value)

```
    # insert new element in binary search tree
    def insert(self,v):
```

```
        return(self.right.maxval())
```

```
    # insert new element in binary search 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
```

Recursively compare starting from root, then based on comparision move to the left or right tree.
Add/insert value at it's correct place.

```
    # delete element from binary search 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 functionality
i) If no child, remove node
ii) If left child exist, promote left subtree.
iii) If right child exist, promote right subtree.
iv) If both child, then replace by in-order predesor (rightmost of left subtree)

```
    # 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
```

→ For making current node empty.

→ For promoting left child.

→ For promoting right child.

Time Complexities =>

- 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 $O(logn)$
- Will see how to keep a tree balanced to ensure all operations remain $O(logn)$