

<u>Home</u> <u>Week-5</u> <u>Week-7</u>

PDSA - Week 6

PDSA - Week 6

```
Union-Find data structure
Improved Kruskal's algorithm using Union-find
Priority Queue
Heap
Improve Dijkstra's algorithm using min heap
Heap Sort
Binary Search Tree(BST)
```

Union-Find data structure

- A set S partitioned into components {C1, C2, ..., Ck}
 - Each s ∈ S belongs to exactly one Cj
- Support the following operations
 - \circ MakeUnionFind(S) set up initial singleton components {s}, for each s ∈ S
 - Find(s) return the component containing s
 - Union(s,s') merges components containing s, s 0

Visualization

https://visualgo.net/en/ufds

Naïve Implementation of Union-Find

```
1
    class MakeUnionFind:
 2
        def __init__(self):
 3
            self.components = {}
 4
            self.size = 0
 5
        def make_union_find(self,vertices):
 6
            self.size = vertices
 7
             for vertex in range(vertices):
 8
                 self.components[vertex] = vertex
 9
        def find(self,vertex):
10
             return self.components[vertex]
11
        def union(self,u,v):
12
            c_old = self.components[u]
13
             c_new = self.components[v]
14
            for k in range(self.size):
```

```
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 of Union-Find

```
class MakeUnionFind:
 2
        def __init__(self):
 3
            self.components = {}
 4
            self.members = {}
             self.size = {}
 6
        def make_union_find(self,vertices):
 7
            for vertex in range(vertices):
                 self.components[vertex] = vertex
 8
 9
                 self.members[vertex] = [vertex]
10
                 self.size[vertex] = 1
        def find(self,vertex):
11
             return self.components[vertex]
12
13
        def union(self,u,v):
14
            c_old = self.components[u]
            c_new = self.components[v]
15
            # Always add member in components which have greater size
16
17
            if self.size[c_new] >= self.size[c_old]:
                 for x in self.members[c_old]:
18
19
                     self.components[x] = c_new
                     self.members[c_new].append(x)
21
                     self.size[c_new] += 1
22
             else:
23
                 for x in self.members[c_new]:
                     self.components[x] = c_old
24
25
                     self.members[c_old].append(x)
                     self.size[c_old] += 1
26
```

Complexity

- MakeUnionFind(S) O(n)
- Find(i) O(1)
- Union(i,j) O(logn)

Improved Kruskal's algorithm using Union-find

```
class MakeUnionFind:
def __init__(self):
    self.components = {}
self.members = {}
self.size = {}
def make_union_find(self,vertices):
```

```
for vertex in range(vertices):
 8
                 self.components[vertex] = vertex
9
                 self.members[vertex] = [vertex]
10
                 self.size[vertex] = 1
11
        def find(self,vertex):
12
             return self.components[vertex]
13
        def union(self,u,v):
            c_old = self.components[u]
14
             c_new = self.components[v]
15
16
            # Always add member in components which have greater size
17
            if self.size[c_new] >= self.size[c_old]:
18
                 for x in self.members[c_old]:
19
                     self.components[x] = c_new
20
                     self.members[c_new].append(x)
21
                     self.size[c_new] += 1
            else:
22
                 for x in self.members[c_new]:
23
24
                     self.components[x] = c_old
                     self.members[c\_old].append(x)
25
26
                     self.size[c_old] += 1
27
28
29
    def kruskal(WList):
30
        (edges, TE) = ([], [])
        for u in WList.keys():
31
32
             edges.extend([(d,u,v) \text{ for } (v,d) \text{ in } WList[u]])
33
        edges.sort()
        mf = MakeUnionFind()
34
        mf.make_union_find(len(WList))
35
36
        for (d,u,v) in edges:
            if mf.components[u] != mf.components[v]:
37
38
                 mf.union(u,v)
39
                 TE.append((u,v,d))
40
            # We can stop the process if the size becomes equal to the total
    number of vertices
41
            # Which represent that a spanning tree is completed
            if mf.size[mf.components[u]]>= mf.size[mf.components[u]]:
42
43
                 if mf.size[mf.components[u]] == len(WList):
                     break
44
45
            else:
                 if mf.size[mf.components[v]] == len(WList):
46
47
                     break
48
        return(TE)
49
50
    # Testcase
51
52
    edge = [(0,1,10),(0,2,18),(0,3,6),(0,4,20),(0,5,13),(1,2,10),(1,3,10),
    (1,4,5),(1,5,7),(2,3,2),(2,4,14),(2,5,15),(3,4,17),(3,5,12),(4,5,10)
53
54
    size = 6
55
    WL = \{\}
56
    for i in range(size):
57
        WL[i] = []
58
    for (i,j,d) in edge:
59
        WL[i].append((j,d))
```

```
60 print(kruskal(WL))
```

```
1 [(2, 3, 2), (1, 4, 5), (0, 3, 6), (1, 5, 7), (0, 1, 10)]
```

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)

Priority Queue

Need to maintain a collection of items with priorities to optimize the following operations

- delete max()
 - Identify and remove item with highest priority
 - Need not be unique
- insert()
 - Add a new item to the list

Heap

Binary tree

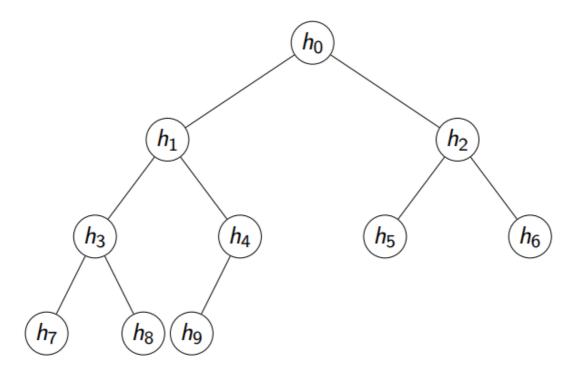
A binary tree is a tree data structure in which each node can contain at most 2 children, which are referred to as the left child and the right child.

Heap

Heap is a binary tree, filled level by level, left to right. There are two types of the heap:

- Max heap For each node V in heap except for leaf nodes, the value of V should be greater or equal to its child's node value.
- Min heap For each node V in heap except for leaf nodes, the value of V should be less or equal to its child's node value.

We can represent heap using array(list in python)



```
H = [h0, h1, h2, h3, h4, h5, h6, h7, h8, h9]

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
```

Visualization

https://visualgo.net/en/heap

Implementation of Maxheap

```
class maxheap:
 1
 2
        def __init__(self):
 3
            self.A = []
 4
        def max_heapify(self,k):
            1 = 2 * k + 1
 5
 6
             r = 2 * k + 2
 7
            largest = k
 8
            if 1 < len(self.A) and self.A[1] > self.A[largest]:
 9
                 largest = 1
10
            if r < len(self.A) and self.A[r] > self.A[largest]:
11
                largest = r
12
            if largest != k:
13
                 self.A[k], self.A[largest] = self.A[largest], self.A[k]
14
                 self.max_heapify(largest)
15
16
        def build_max_heap(self,L):
17
            self.A = []
18
            for i in L:
19
                 self.A.append(i)
            n = int((len(self.A)//2)-1)
20
```

```
for k in range(n, -1, -1):
21
22
                 self.max_heapify(k)
23
24
25
        def delete_max(self):
26
            item = None
            if self.A != []:
27
28
                 self.A[0], self.A[-1] = self.A[-1], self.A[0]
29
                 item = self.A.pop()
30
                 self.max_heapify(0)
             return item
31
32
33
34
        def insert_in_maxheap(self,d):
35
            self.A.append(d)
36
            index = len(self.A)-1
            while index > 0:
37
38
                 parent = (index-1)//2
39
                 if self.A[index] > self.A[parent]:
40
                     self.A[index],self.A[parent] = self.A[parent],self.A[index]
41
                     index = parent
42
                 else:
43
                     break
44
45
    heap = maxheap()
46
    heap.build_max_heap([1,2,3,4,5,6])
47
    print(heap.A)
    heap.insert_in_maxheap(7)
48
49
    print(heap.A)
50
    heap.insert_in_maxheap(8)
    print(heap.A)
51
52
    print(heap.delete_max())
53
    print(heap.delete_max())
54
    print(heap.A)
```

```
1 [6, 5, 3, 4, 2, 1]
2 [7, 5, 6, 4, 2, 1, 3]
3 [8, 7, 6, 5, 2, 1, 3, 4]
4 8
5 7
6 [6, 5, 3, 4, 2, 1]
```

Implementation of Minheap

```
1  class minheap:
2   def __init__(self):
3     self.A = []
4   def min_heapify(self,k):
5     l = 2 * k + 1
        r = 2 * k + 2
        smallest = k
```

```
if 1 < len(self.A) and self.A[1] < self.A[smallest]:</pre>
 8
 9
                 smallest = 1
10
             if r < len(self.A) and self.A[r] < self.A[smallest]:
11
                 smallest = r
12
             if smallest != k:
13
                 self.A[k], self.A[smallest] = self.A[smallest], self.A[k]
14
                 self.min_heapify(smallest)
15
16
        def build_min_heap(self,L):
17
             self.A = []
18
             for i in L:
19
                 self.A.append(i)
20
             n = int((len(self.A)//2)-1)
21
             for k in range(n, -1, -1):
22
                 self.min_heapify(k)
23
24
25
        def delete_min(self):
26
             item = None
27
             if self.A != []:
                 self.A[0], self.A[-1] = self.A[-1], self.A[0]
28
29
                 item = self.A.pop()
30
                 self.min_heapify(0)
31
             return item
32
33
34
        def insert_in_minheap(self,d):
35
             self.A.append(d)
36
             index = len(self.A)-1
37
            while index > 0:
38
                 parent = (index-1)//2
39
                 if self.A[index] < self.A[parent]:</pre>
40
                     self.A[index],self.A[parent] = self.A[parent],self.A[index]
41
                     index = parent
42
                 else:
43
                     break
44
45
    heap = minheap()
46
    heap.build_min_heap([6,5,4,3,2])
47
    print(heap.A)
48
    heap.insert_in_minheap(1)
49
    print(heap.A)
50
    heap.insert_in_minheap(8)
51
    print(heap.A)
52
    print(heap.delete_min())
    print(heap.delete_min())
53
    print(heap.A)
54
```

```
1 [2, 3, 4, 6, 5]

2 [1, 3, 2, 6, 5, 4]

3 [1, 3, 2, 6, 5, 4, 8]

4 1

5 2

6 [3, 5, 4, 6, 8]
```

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)

Improve Dijkstra's algorithm using min heap

Old implementation for adjacency matrix

```
1
    def dijkstra(WMat,s):
 2
        (rows, cols, x) = WMat.shape
 3
        infinity = np.max(WMat)*rows+1
 4
        (visited, distance) = ({},{})
 5
        for v in range(rows):
 6
             (visited[v], distance[v]) = (False, infinity)
 8
        distance[s] = 0
9
        for u in range(rows):
10
            nextd = min([distance[v] for v in range(rows)
11
12
                             if not visited[v]])
13
            nextvlist = [v for v in range(rows)
                             if (not visited[v]) and
14
15
                                 distance[v] == nextd]
            if nextvlist == []:
16
17
                break
            nextv = min(nextvlist)
18
19
20
            visited[nextv] = True
21
            for v in range(cols):
22
                 if WMat[nextv,v,0] == 1 and (not visited[v]):
                     distance[v] = min(distance[v], distance[nextv]
23
24
                                                    +WMat[nextv,v,1])
        return(distance)
25
26
27
28
    dedges = [(0,1,10),(0,2,80),(1,2,6),(1,4,20),(2,3,70),(4,5,50),(4,6,5),
    (5,6,10)
29
    \#edges = dedges + [(j,i,w)] for (i,j,w) in dedges
30
    size = 7
31
    import numpy as np
    W = np.zeros(shape=(size,size,2))
32
    for (i,j,w) in dedges:
```

```
1 {0: 0, 1: 10.0, 2: 16.0, 3: 86.0, 4: 30.0, 5: 80.0, 6: 35.0}
```

Updated Implementation for adjacency matrix using min heap

```
1
    # considering dictionary as a heap for given code
 2
    def min_heapify(i,size):
 3
        1child = 2*i + 1
4
        rchild = 2*i + 2
 5
        small = i
 6
        if lchild < size-1 and HtoV[lchild][1] < HtoV[small][1]:</pre>
 7
             small = 1child
        if rchild < size-1 and HtoV[rchild][1] < HtoV[small][1]:</pre>
8
9
            small = rchild
10
        if small != i:
11
            Vtoh[HtoV[small][0]] = i
12
            Vtoh[HtoV[i][0]] = small
13
             (HtoV[small], HtoV[i]) = (HtoV[i], HtoV[small])
14
             min_heapify(small, size)
15
16
    def create_minheap(size):
17
        for x in range((size//2)-1,-1,-1):
18
            min_heapify(x, size)
19
20
    def minheap_update(i,size):
21
        if i!= 0:
            while i > 0:
22
23
                 parent = (i-1)//2
24
                 if HtoV[parent][1] > HtoV[i][1]:
25
                     Vtoh[HtoV[parent][0]] = i
26
                     VtoH[HtoV[i][0]] = parent
27
                     (HtoV[parent], HtoV[i]) = (HtoV[i], HtoV[parent])
                 else:
28
29
                     break
30
                 i = parent
31
32
    def delete_min(hsize):
33
        Vtoh[HtoV[0][0]] = hsize-1
34
        Vtoh[HtoV[hsize-1][0]] = 0
35
        HtoV[hsize-1],HtoV[0] = HtoV[0],HtoV[hsize-1]
36
        node,dist = HtoV[hsize-1]
37
        hsize = hsize - 1
38
        min_heapify(0,hsize)
39
        return node, dist, hsize
40
41
    #global HtoV map heap index to (vertex, distance from source)
42
    #global VtoH map vertex to heap index
```

```
43
    HtoV, VtoH = \{\},\{\}
44
    def dijkstra(WMat,s):
45
        (rows,cols,x) = WMat.shape
46
        infinity = float('inf')
47
        visited = {}
48
        heapsize = rows
49
        for v in range(rows):
50
            VtoH[v]=v
51
            HtoV[v]=[v,infinity]
52
            visited[v] = False
53
        HtoV[s] = [s,0]
54
        create_minheap(heapsize)
55
56
        for u in range(rows):
57
            nextd,ds,heapsize = delete_min(heapsize)
58
            visited[nextd] = True
            for v in range(cols):
59
60
                 if WMat[nextd, v, 0] == 1 and (not visited[v]):
                     # update distance of adjacent of v if it is less than to
61
    previous one
62
                     HtoV[VtoH[v]][1] = min(HtoV[VtoH[v]][1], ds+WMat[nextd, v, 1])
63
                     minheap_update(VtoH[v],heapsize)
64
65
    dedges = [(0,1,10),(0,2,80),(1,2,6),(1,4,20),(2,3,70),(4,5,50),(4,6,5),
66
    (5,6,10)
    \#edges = dedges + [(j,i,w) for (i,j,w) in dedges]
67
68
    size = 7
    import numpy as np
69
    W = np.zeros(shape=(size,size,2))
70
    for (i,j,w) in dedges:
71
72
        W[i,j,0] = 1
73
        W[i,j,1] = W
74
    s = 0
75
    dijkstra(W,s)
76
    #print(HtoV)
    #print(VtoH)
77
78
    for i in range(size):
        print('Shortest distance from {0} to {1} = {2}'.format(s,i,HtoV[VtoH[i]]
79
    [1]))
```

```
Shortest distance from 0 to 0 = 0

Shortest distance from 0 to 1 = 10.0

Shortest distance from 0 to 2 = 16.0

Shortest distance from 0 to 3 = 86.0

Shortest distance from 0 to 4 = 30.0

Shortest distance from 0 to 5 = 80.0

Shortest distance from 0 to 6 = 35.0
```

Updated Implementation for adjacency list using min heap

```
def min_heapify(i,size):
 1
 2
         lchild = 2*i + 1
 3
         rchild = 2*i + 2
 4
        small = i
 5
        if lchild < size-1 and HtoV[lchild][1] < HtoV[small][1]:</pre>
 6
             small = 1child
 7
        if rchild < size-1 and HtoV[rchild][1] < HtoV[small][1]:</pre>
 8
             small = rchild
        if small != i:
 9
10
            Vtoh[HtoV[small][0]] = i
11
             Vtoh[HtoV[i][0]] = small
12
             (HtoV[small], HtoV[i]) = (HtoV[i], HtoV[small])
13
            min_heapify(small, size)
14
15
    def create_minheap(size):
16
        for x in range((size//2)-1,-1,-1):
17
             min_heapify(x, size)
18
19
    def minheap_update(i,size):
        if i!= 0:
20
             while i > 0:
21
22
                 parent = (i-1)//2
23
                 if HtoV[parent][1] > HtoV[i][1]:
24
                     VtoH[HtoV[parent][0]] = i
25
                     VtoH[HtoV[i][0]] = parent
26
                     (HtoV[parent],HtoV[i]) = (HtoV[i], HtoV[parent])
27
                 else:
28
                     break
29
                 i = parent
30
    def delete_min(hsize):
31
32
        Vtoh[HtoV[0][0]] = hsize-1
33
        Vtoh[HtoV[hsize-1][0]] = 0
34
        HtoV[hsize-1],HtoV[0] = HtoV[0],HtoV[hsize-1]
35
        node,dist = HtoV[hsize-1]
36
        hsize = hsize - 1
37
        min_heapify(0,hsize)
38
         return node, dist, hsize
39
40
    HtoV, VtoH = \{\}, \{\}
41
42
    #global HtoV map heap index to (vertex,distance from source)
43
    #global VtoH map vertex to heap index
44
    def dijkstralist(WList,s):
45
        infinity = float('inf')
46
        visited = {}
        heapsize = len(WList)
47
48
        for v in WList.keys():
49
             VtoH[v]=v
50
             HtoV[v]=[v,infinity]
51
             visited[v] = False
52
        HtoV[s] = [s,0]
53
        create_minheap(heapsize)
54
55
         for u in WList.keys():
```

```
nextd,ds,heapsize = delete_min(heapsize)
56
57
            visited[nextd] = True
58
            for v,d in WList[nextd]:
59
                if not visited[v]:
60
                     HtoV[VtoH[v]][1] = min(HtoV[VtoH[v]][1], ds+d)
61
                     minheap_update(VtoH[v],heapsize)
62
63
64
    dedges = [(0,1,10),(0,2,80),(1,2,6),(1,4,20),(2,3,70),(4,5,50),(4,6,5),
    (5,6,10)]
    \#edges = dedges + [(j,i,w) for (i,j,w) in dedges]
65
66
    size = 7
67
    WL = \{\}
68
   for i in range(size):
69
        WL[i] = []
70
   for (i,j,d) in dedges:
        WL[i].append((j,d))
71
72
    s = 0
73
    dijkstralist(WL,s)
74
   #print(HtoV)
75
    #print(VtoH)
76
    for i in range(size):
77
        print('Shortest distance from {0} to {1} = {2}'.format(s,i,HtoV[VtoH[i]]
    [1]))
78
```

```
Shortest distance from 0 to 0 = 0

Shortest distance from 0 to 1 = 10

Shortest distance from 0 to 2 = 16

Shortest distance from 0 to 3 = 86

Shortest distance from 0 to 4 = 30

Shortest distance from 0 to 5 = 80

Shortest distance from 0 to 6 = 35
```

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)

Heap Sort

Implementation

```
def max_heapify(A, size, k):
 1
 2
        1 = 2 * k + 1
 3
        r = 2 * k + 2
        largest = k
 4
        if 1 < size and A[1] > A[largest]:
 5
 6
             largest = 1
 7
        if r < size and A[r] > A[largest]:
 8
            largest = r
 9
        if largest != k:
10
             (A[k], A[largest]) = (A[largest], A[k])
11
            max_heapify(A,size,largest)
12
13
    def build_max_heap(A):
14
        n = (1en(A)//2)-1
15
        for i in range(n, -1, -1):
16
            max_heapify(A,len(A),i)
17
18
    def heapsort(A):
19
        build_max_heap(A)
20
        n = len(A)
        for i in range(n-1,-1,-1):
21
22
            A[0], A[i] = A[i], A[0]
23
            \max_{heapify(A,i,0)}
24
25
26
    A = [8,6,9,3,4,5,61,6666]
27
    heapsort(A)
    print(A)
28
```

```
1 [3, 4, 5, 6, 8, 9, 61, 6666]
```

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

Binary Search Tree(BST)

For dynamic stored data

- · Sorting is useful for efficient searching
- What if the data is changing dynamically?
- Items are periodically inserted and deleted Insert/delete in a sorted list takes time O(n)

How can we improve Insert/delete time? - using tree structure?

A **binary search tree** is a binary tree that is either empty or satisfies the following conditions:

For each node V in the Tree

- The value of the left child or left subtree is always less than the value of V.
- The value of the right child or right subtree is always greater than the value of V.

Visualization

https://visualgo.net/en/bst

Implementation

```
class Tree:
1
2
    # Constructor:
 3
        def __init__(self,initval=None):
4
            self.value = initval
 5
            if self.value:
                 self.left = Tree()
 6
                 self.right = Tree()
 7
 8
            else:
9
                 self.left = None
10
                 self.right = None
11
             return
12
        # Only empty node has value None
13
        def isempty(self):
14
             return (self.value == None)
15
        # Leaf nodes have both children empty
16
        def isleaf(self):
             return (self.value != None and self.left.isempty() and
17
    self.right.isempty())
        # Inorder traversal
18
        def inorder(self):
19
20
            if self.isempty():
21
                 return([])
22
            else:
                 return(self.left.inorder()+[self.value]+self.right.inorder())
23
24
            # Display Tree as a string
        def __str__(self):
25
             return(str(self.inorder()))
26
27
        # Check if value v occurs in tree
        def find(self,v):
28
29
            if self.isempty():
30
                 return(False)
31
            if self.value == v:
32
                return(True)
33
            if v < self.value:</pre>
34
                 return(self.left.find(v))
             if v > self.value:
35
36
                 return(self.right.find(v))
        # return minimum value for tree rooted on self - Minimum is left most
37
    node in the tree
        def minval(self):
38
39
            if self.left.isempty():
40
                 return(self.value)
41
            else:
42
                 return(self.left.minval())
```

```
43
        # return max value for tree rooted on self - Maximum is right most
    node in the tree
44
        def maxval(self):
45
            if self.right.isempty():
46
                 return(self.value)
47
             else:
                 return(self.right.maxval())
48
49
        # insert new element in binary search tree
        def insert(self,v):
50
51
            if self.isempty():
                 self.value = v
52
53
                 self.left = Tree()
54
                 self.right = Tree()
55
            if self.value == v:
56
                 return
57
            if v < self.value:</pre>
                self.left.insert(v)
58
59
                 return
60
            if v > self.value:
61
                 self.right.insert(v)
62
                 return
63
        # delete element from binary search tree
64
        def delete(self,v):
65
            if self.isempty():
66
                 return
67
            if v < self.value:
68
                 self.left.delete(v)
69
                 return
70
            if v > self.value:
71
                self.right.delete(v)
72
                 return
            if v == self.value:
73
74
                 if self.isleaf():
75
                     self.makeempty()
                 elif self.left.isempty():
76
77
                     self.copyright()
78
                 elif self.right.isempty():
79
                     self.copyleft()
80
                 else:
                     self.value = self.left.maxval()
81
82
                     self.left.delete(self.left.maxval())
83
                 return
84
        # Convert leaf node to empty node
85
        def makeempty(self):
             self.value = None
86
87
             self.left = None
             self.right = None
88
89
             return
        # Promote left child
90
91
        def copyleft(self):
             self.value = self.left.value
92
93
             self.right = self.left.right
             self.left = self.left.left
94
95
             return
96
        # Promote right child
```

```
97
         def copyright(self):
 98
             self.value = self.right.value
             self.left = self.right.left
99
100
             self.right = self.right.right
101
             return
102
103
104
105
    T = Tree()
106
    bst = [9,8,7,6,5,4,3,2,1]
107
    k = 4
108 for i in bst:
109
         T.insert(i)
110 | print('Element in BST are:= ',T.inorder())
print('Maximum element in BST are:= ',T.maxval())
print('Minimum element in BST are:= ',T.minval())
113
    print(k,'is present or not = ',T.find(k))
114
    T.delete(3)
115
     print('Element in BST after delete 3:= ',T.inorder())
```

```
1    Element in BST are:= [1, 2, 3, 4, 5, 6, 7, 8, 9]
2    Maximum element in BST are:= 9
3    Minimum element in BST are:= 1
4    4 is present or not = True
5    Element in BST after delete 3:= [1, 2, 4, 5, 6, 7, 8, 9]
```

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