Union-Find Data Structure : Priority Queue : Heap : 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
  - 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

```python
#Naïve Implementation of Union-Find
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 Component[k] == c_old:
                Component[k] = c_new
```

```python
#Improved Implementation of Union-Find
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]
    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
```

Complexity

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

Complexity

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

Improved Kruskal's using algorithm using Union-find:

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, $O((m+n)logn)$

```python
#Improved Kruskal's algorithm using Union-find
def kruskal(WList):
    (edges,TE) = ([],[])
    for u in WList.keys():
        edges.extend([(d,u,v) for (v,d) in WList[u]])
    edges.sort()
    mf = MakeUnionFind() #Given on Page1
    mf.make_union_find(len(WList))
    for (d,u,v) in edges:
        if mf.components[u] != mf.components[v]:
            mf.union(u,v)
            TE.append((u,v,d))
        '''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[u]]:
            if mf.size[mf.components[u]] == len(WList):
                break
        else:
            if mf.size[mf.components[v]] == len(WList):
                break
    return(TE)
```

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

Implementing Priority Queues

One dimensional :

- 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 $O(n^2)$

Two dimensional :

- $\sqrt{N} \times \sqrt{N}$ array with sorted rows
    - insert( ) is $O(\sqrt{N})$
    - delete_max is $O(\sqrt{N})$
    - Processing N items is $O(N\sqrt{N})$

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:

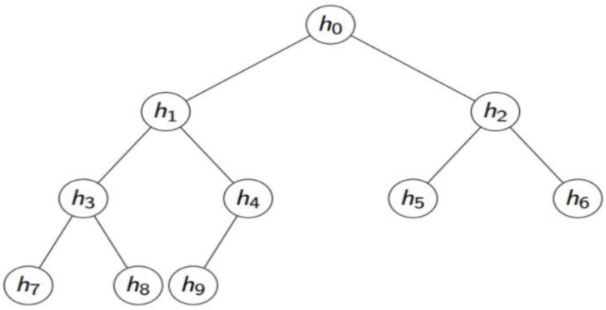
- Max heap - For each node V in heap except for leaf nodes, the value of V should be greater or equal to its child's node value.
- Min heap - For each node V in heap except for leaf nodes, the value of V should be less or equal to its child's node value.
- **We can represent heap using array(list in 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`




```python
#Max Heap
class maxheap:
    def __init__(self):
        self.A = []
    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)

    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)

    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

    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
```

```python
#Min Heap
class minheap:
    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)

    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)

    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

    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
```

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)***

Complexity : Heap Sort

- 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)

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

```
# considering dictionary as a heap for given code
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)

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
```

```
#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)

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

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

```python
#Updated Implementation for adjacency matrix using min heap:
#global HtoV map heap index to (vertex,distance from source)
#global VtoH map vertex to heap index
HtoV, VtoH = {},{}
def dijkstra(WMat,s):
    (rows,cols,x) = WMat.shape
    infinity = float('inf')
    visited = {}
    heapsize = rows
    for v in range(rows):
        VtoH[v]=v
        HtoV[v]=[v,infinity]
        visited[v] = False
    HtoV[s]= [s,0]
    create_minheap(heapsize)

    for u in range(rows):
        nextd,ds,heapsize = delete_min(heapsize)
        visited[nextd] = True
        for v in range(cols):
            if WMat[nextd,v,0] == 1 and (not visited[v]):
                '''update distance of adjacent of v if it is
                 less than to previous one'''
                HtoV[VtoH[v]][1] = min(HtoV[VtoH[v]][1],ds+WMat[nextd,v,1])
                minheap_update(VtoH[v],heapsize)
```

```python
#Updated Implementation for adjacency list using min heap:
HtoV, VtoH = {},{}
#global HtoV map heap index to (vertex,distance from source)
#global VtoH map vertex to heap index
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)
```

Complexity : BST

- 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)***

BY – ADITYA DHAR DWIVEDI