ALGORITHM AND DATA STRUCTURES ASSIGNMENT

Student Name: Chrysanthi Mouzakitis

Student Number: 122450272

GRAPH / VERTEX / EDGE CLASS IMPLEMENTATION FROM CANVAS

```python
class Vertex:
    """ A Vertex in a graph. """

    def __init__(self, element):
        """ Create a vertex, with data element. """
        self._element = element

    def __str__(self):
        """ Return a string representation of the vertex. """
        return str(self._element)

    def element(self):
        """ Return the data for the vertex. """
        return self._element

    def __lt__(self, v):
        """ Return true if this object is less than v.

        Args:
            v -- a vertex object
        """
        return self._element < v.element()
```

```python
class Edge:


    def __init__(self, v, w, element):
        """ Create an edge between vertices v and w, with label element.

        Args:
```

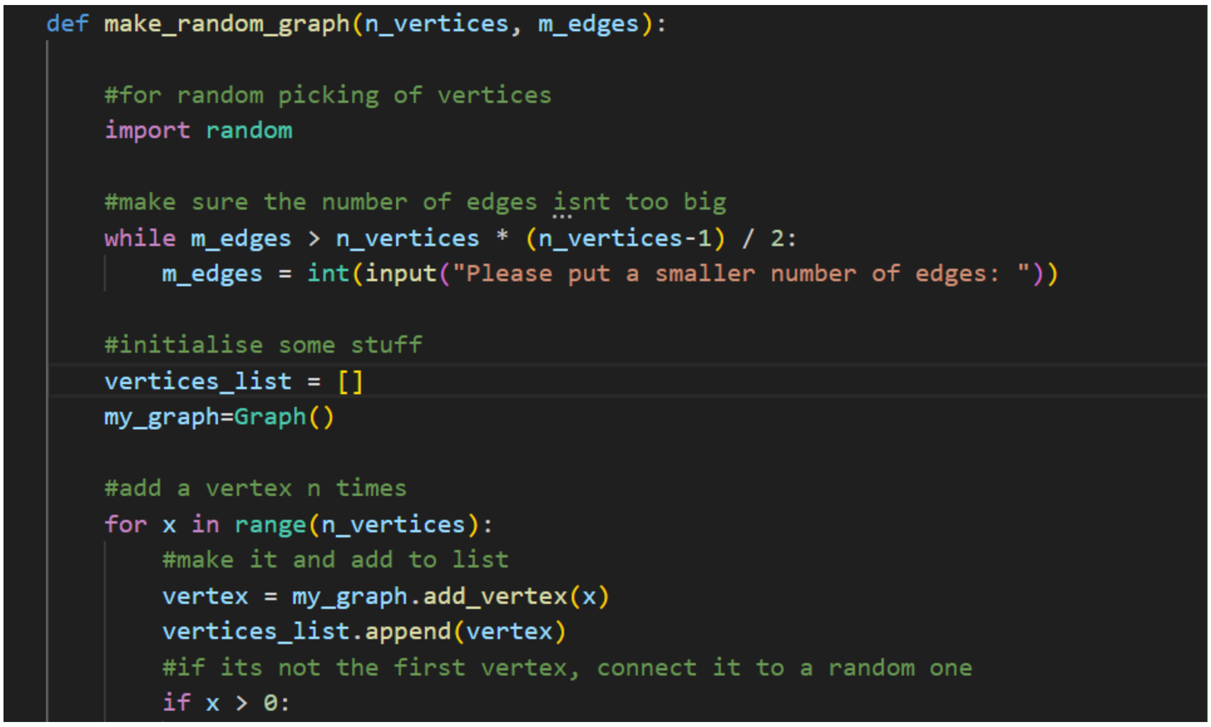
```
        v -- a Vertex object
        w -- a Vertex object
        element -- the label, can be an arbitrarily complex structure.
    """
    self._vertices = (v,w)
    self._element = element

def __str__(self):
    """ Return a string representation of this edge. """
    return ('(' + str(self._vertices[0]) + '--'
               + str(self._vertices[1]) + ' : '
               + str(self._element) + ')')

def vertices(self):
    """ Return an ordered pair of the vertices of this edge."""
    return self._vertices

def opposite(self, v):
    """ Return the opposite vertex to v in this edge, or None if this edge not incident on v.
        """
    if self._vertices[0] == v:
        return self._vertices[1]
    elif self._vertices[1] == v:
        return self._vertices[0]
    else:
        return None

def element(self):
    """ Return the data element for this edge. """
    return self._element

def start(self):
    """ Return the first vertex in the ordered pair. """
    return self._vertices[0]

def end(self):
    """ Return the second vertex in the ordered. pair. """
    return self._vertices[1]
```

```
class Graph:

    def __init__(self):
        self._structure = dict()

    def __str__(self):
        """ Return a string representation of the graph. """
        hstr = ('|V| = ' + str(self.num_vertices())
                + '; |E| = ' + str(self.num_edges()))
        vstr = '\nVertices: '
        for v in self._structure:
            vstr += str(v) + '-'
        edges = self.edges()
        estr = '\nEdges: '
        for e in edges:
            estr += str(e) + ' '
        return hstr + vstr + estr

    #-----------------------------------------------------------#
    #ADT methods to query the graph

    def num_vertices(self):
        """ Return the number of vertices in the graph. """
        return len(self._structure)

    def num_edges(self):
        """ Return the number of edges in the graph. """
        num = 0
        for v in self._structure:
            num += len(self._structure[v])    #the dict of edges for v
        return num //2     #divide by 2, since each edege appears in the
                           #vertex list for both of its vertices

    def vertices(self):
        """ Return a list of all vertices in the graph. """
        return [key for key in self._structure]

    def get_vertex_by_label(self, element):
        for v in self._structure:
            if v.element() == element:
                return v
        return None

    def edges(self):
        """ Return a list of all edges in the graph. """
        edgelist = []
        for v in self._structure:
            for w in self._structure[v]:
                #to avoid duplicates, only return if v is the first vertex
                if self._structure[v][w].start() == v:
                    edgelist.append(self._structure[v][w])
        return edgelist

    def get_edges(self, v):
        """ Return a list of all edges incident on v.
"""
        if v in self._structure:
            edgelist = []
            for w in self._structure[v]:
                edgelist.append(self._structure[v][w])
            return edgelist
        return None

    def get_edge(self, v, w):
        """ Return the edge between v and w, or None, if there is no edge.
        """
        if (self._structure != None
                    and v in self._structure
                    and w in self._structure[v]):
            return self._structure[v][w]
        return None

    def degree(self, v):
```

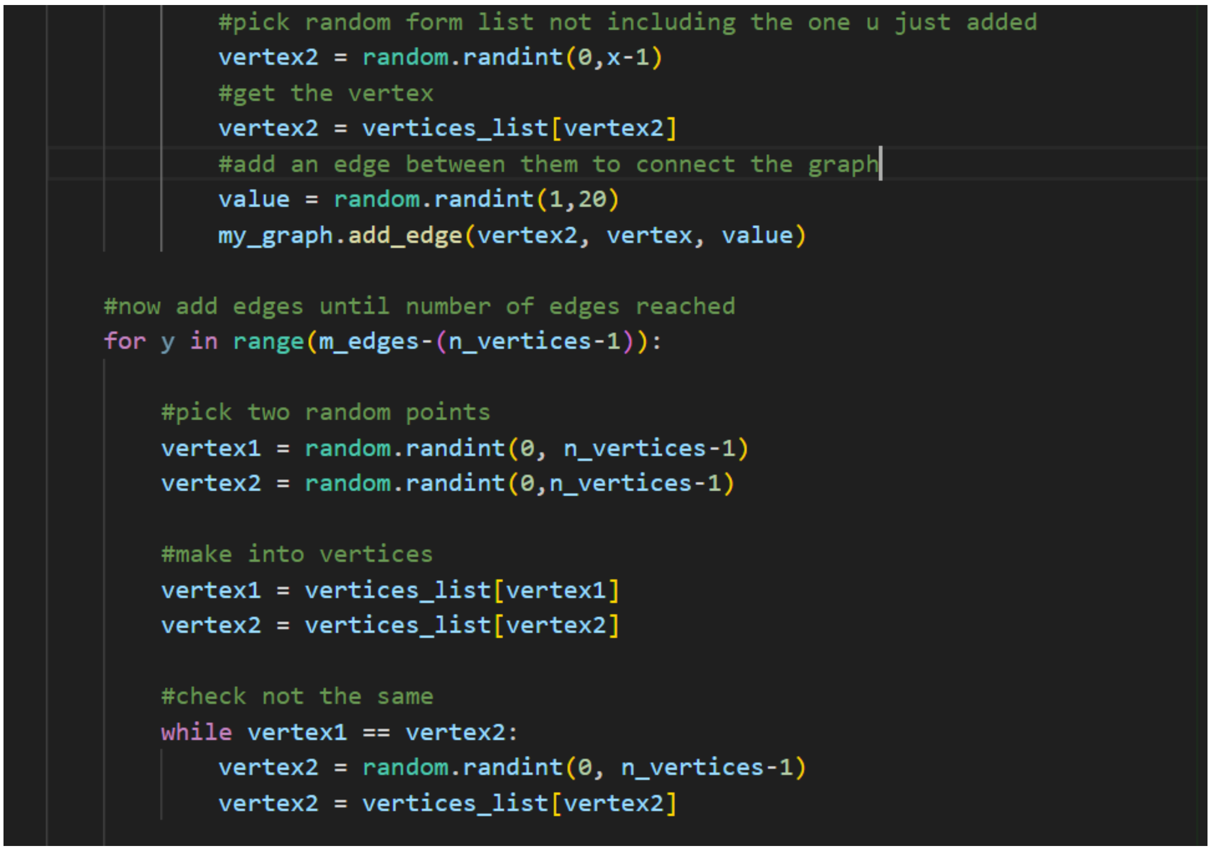
```
    """ Return the degree of vertex v.
    """
    return len(self._structure[v])

#--------------------------------------------------#
#ADT methods to modify the graph

def add_vertex(self, element):
    v = Vertex(element)
    self._structure[v] = dict()  # create an empty dict, ready for edges
    return v

def add_vertex_if_new(self, element):
    for v in self._structure:
        if v.element() == element:
            #print('Already there')
            return v
    return self.add_vertex(element)

def add_edge(self, v, w, element):
    if not v in self._structure or not w in self._structure:
        return None
    e = Edge(v, w, element)
    self._structure[v][w] = e
    self._structure[w][v] = e
    return e

def add_edge_pairs(self, elist):
    """ Add all vertex pairs in elist as edges with empty elements. """
    for (v,w) in elist:
        self.add_edge(v,w,None)

#--------------------------------------------------#
#Additional methods to explore the graph

def highestdegreevertex(self):
    """ Return the vertex with highest degree. """
    hd = -1
```

```
hdv = None
for v in self._structure:
    if self.degree(v) > hd:
        hd = self.degree(v)
        hdv = v
return hdv
```

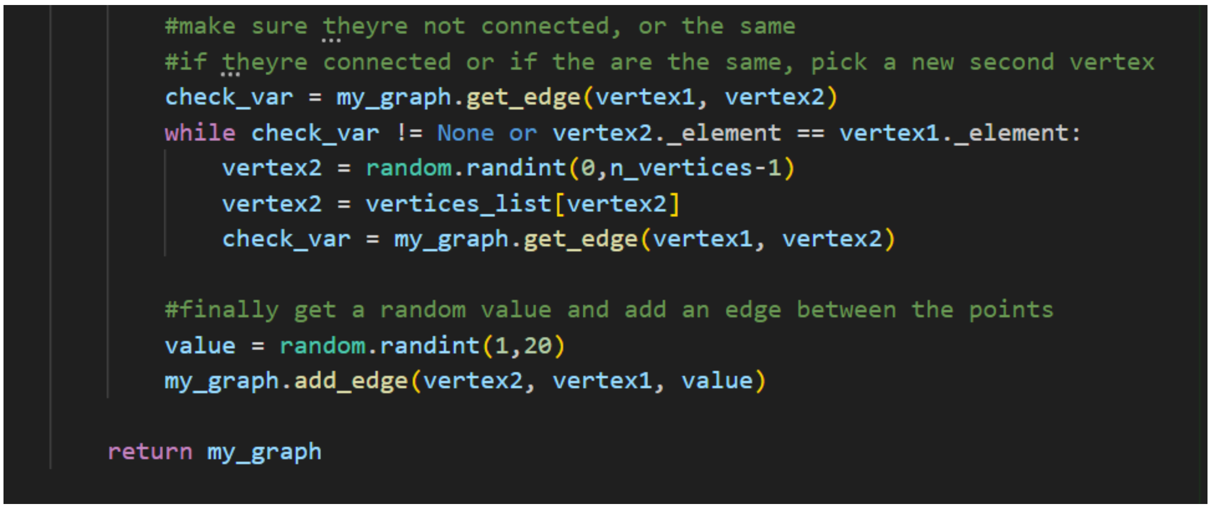
CODE USED TO GENERATE A RANDOM GRAPH



```python
def make_random_graph(n_vertices, m_edges):

    #for random picking of vertices
    import random

    #make sure the number of edges isnt too big
    while m_edges > n_vertices * (n_vertices-1) / 2:
        m_edges = int(input("Please put a smaller number of edges: "))

    #initialise some stuff
    vertices_list = []
    my_graph=Graph()

    #add a vertex n times
    for x in range(n_vertices):
        #make it and add to list
        vertex = my_graph.add_vertex(x)
        vertices_list.append(vertex)
        #if its not the first vertex, connect it to a random one
        if x > 0:
```

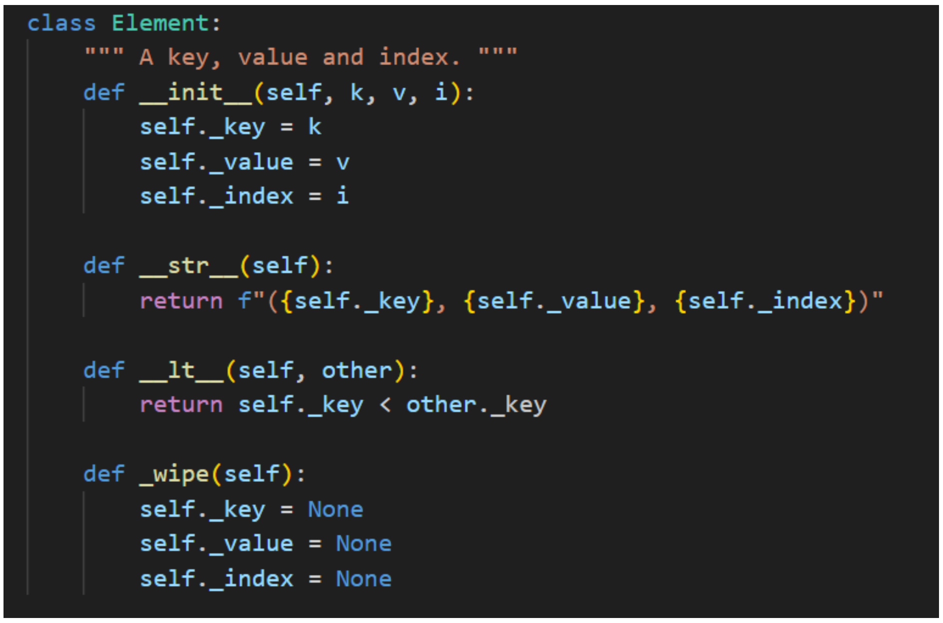


```python
            #pick random form list not including the one u just added
            vertex2 = random.randint(0,x-1)
            #get the vertex
            vertex2 = vertices_list[vertex2]
            #add an edge between them to connect the graph
            value = random.randint(1,20)
            my_graph.add_edge(vertex2, vertex, value)

    #now add edges until number of edges reached
    for y in range(m_edges-(n_vertices-1)):

        #pick two random points
        vertex1 = random.randint(0, n_vertices-1)
        vertex2 = random.randint(0,n_vertices-1)

        #make into vertices
        vertex1 = vertices_list[vertex1]
        vertex2 = vertices_list[vertex2]

        #check not the same
        while vertex1 == vertex2:
            vertex2 = random.randint(0, n_vertices-1)
            vertex2 = vertices_list[vertex2]
```



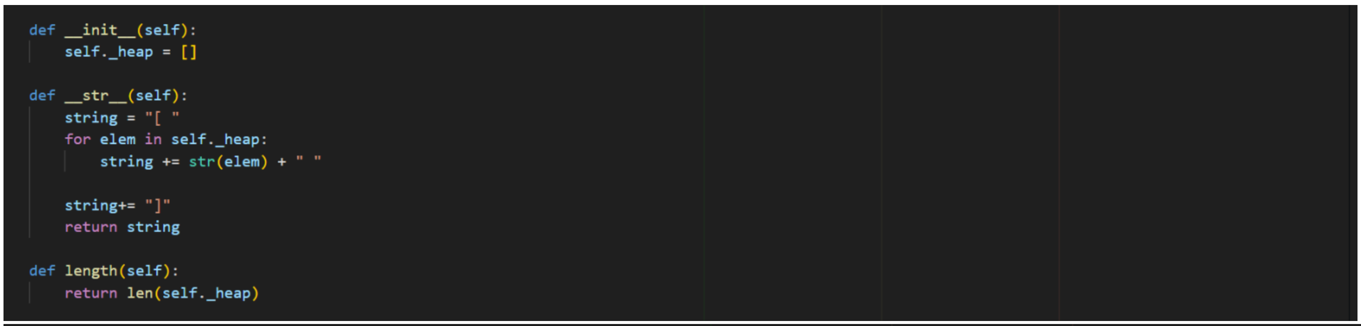
```python
        #make sure theyre not connected, or the same
        #if theyre connected or if the are the same, pick a new second vertex
        check_var = my_graph.get_edge(vertex1, vertex2)
        while check_var != None or vertex2._element == vertex1._element:
            vertex2 = random.randint(0,n_vertices-1)
            vertex2 = vertices_list[vertex2]
            check_var = my_graph.get_edge(vertex1, vertex2)

        #finally get a random value and add an edge between the points
        value = random.randint(1,20)
        my_graph.add_edge(vertex2, vertex1, value)

    return my_graph
```

ELEMENT CLASS TAKEN FROM SLIDES FOR THE ADAPTABLE PRIORITY QUEUE IMPLEMENTATIONS

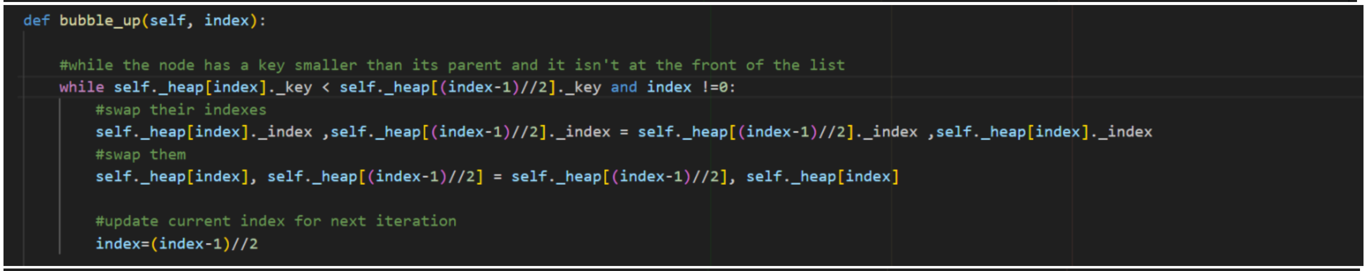


```python
class Element:
    """ A key, value and index. """
    def __init__(self, k, v, i):
        self._key = k
        self._value = v
        self._index = i

    def __str__(self):
        return f"({self._key}, {self._value}, {self._index})"

    def __lt__(self, other):
        return self._key < other._key

    def _wipe(self):
        self._key = None
        self._value = None
        self._index = None
```

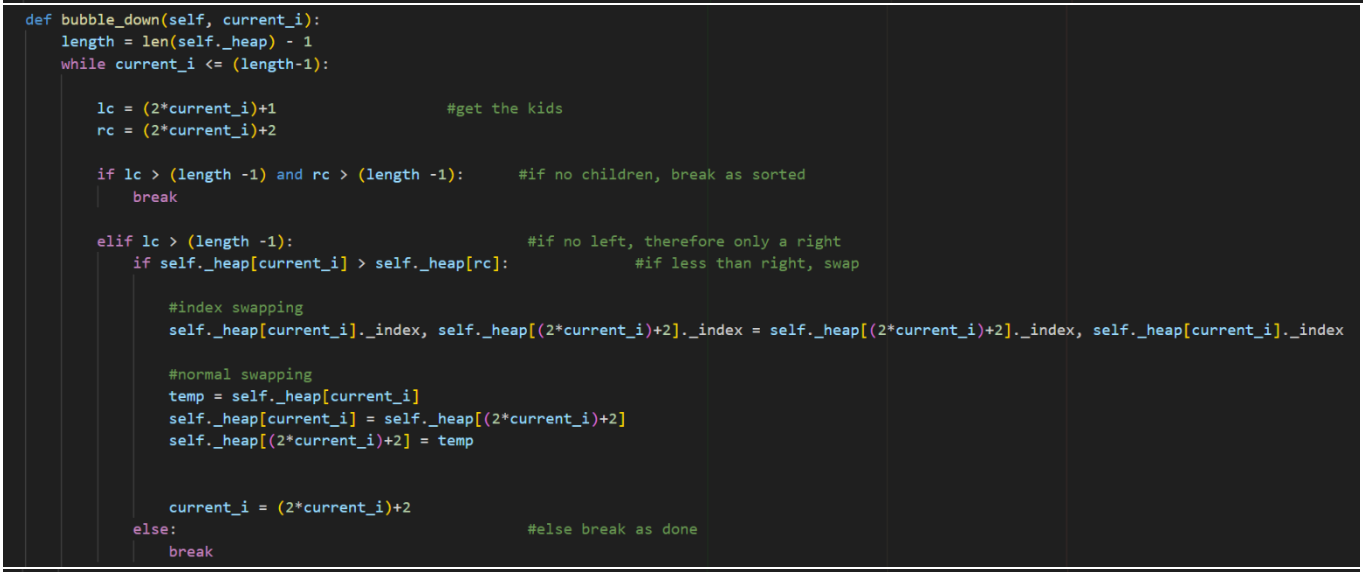
ADAPTABLE PRIORITY QUEUE CLASS WITH AN ARRAY BASED HEAP



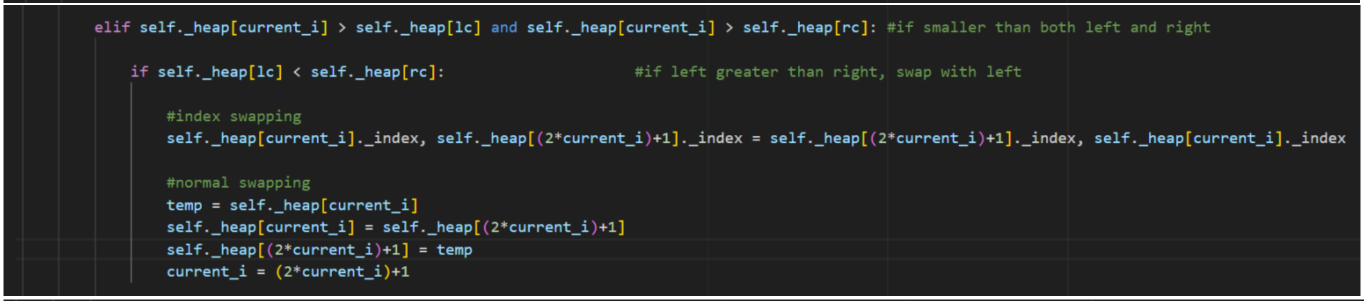
```
def __init__(self):
    self._heap = []

def __str__(self):
    string = "[ "
    for elem in self._heap:
        string += str(elem) + " "

    string+= "]"
    return string

def length(self):
    return len(self._heap)
```



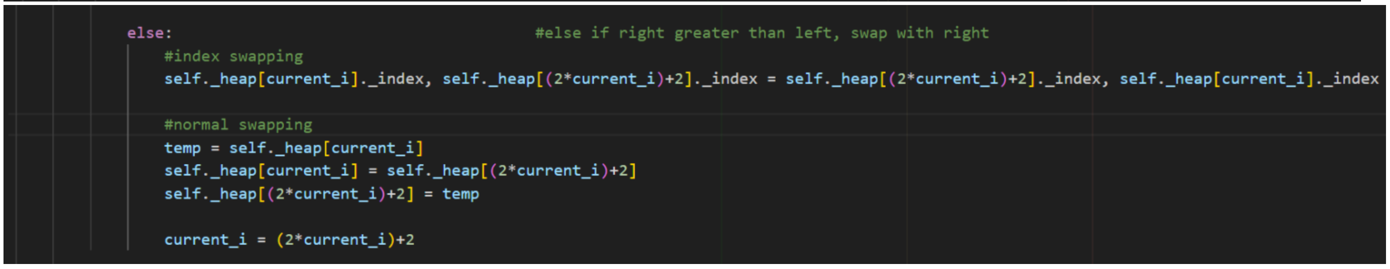
```
def bubble_up(self, index):

    #while the node has a key smaller than its parent and it isn't at the front of the list
    while self._heap[index]._key < self._heap[(index-1)//2]._key and index !=0:
        #swap their indexes
        self._heap[index]._index ,self._heap[(index-1)//2]._index = self._heap[(index-1)//2]._index ,self._heap[index]._index
        #swap them
        self._heap[index], self._heap[(index-1)//2] = self._heap[(index-1)//2], self._heap[index]

        #update current index for next iteration
        index=(index-1)//2
```



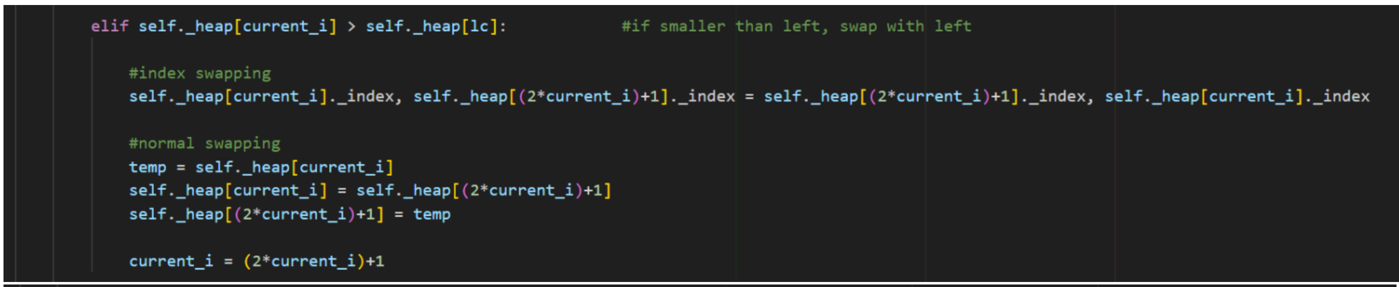
```
def bubble_down(self, current_i):
    length = len(self._heap) - 1
    while current_i <= (length-1):

        lc = (2*current_i)+1                  #get the kids
        rc = (2*current_i)+2

        if lc > (length -1) and rc > (length -1):      #if no children, break as sorted
            break

        elif lc > (length -1):                         #if no left, therefore only a right
            if self._heap[current_i] > self._heap[rc]:             #if less than right, swap

                #index swapping
                self._heap[current_i]._index, self._heap[(2*current_i)+2]._index = self._heap[(2*current_i)+2]._index, self._heap[current_i]._index

                #normal swapping
                temp = self._heap[current_i]
                self._heap[current_i] = self._heap[(2*current_i)+2]
                self._heap[(2*current_i)+2] = temp


                current_i = (2*current_i)+2
            else:                                      #else break as done
                break
```



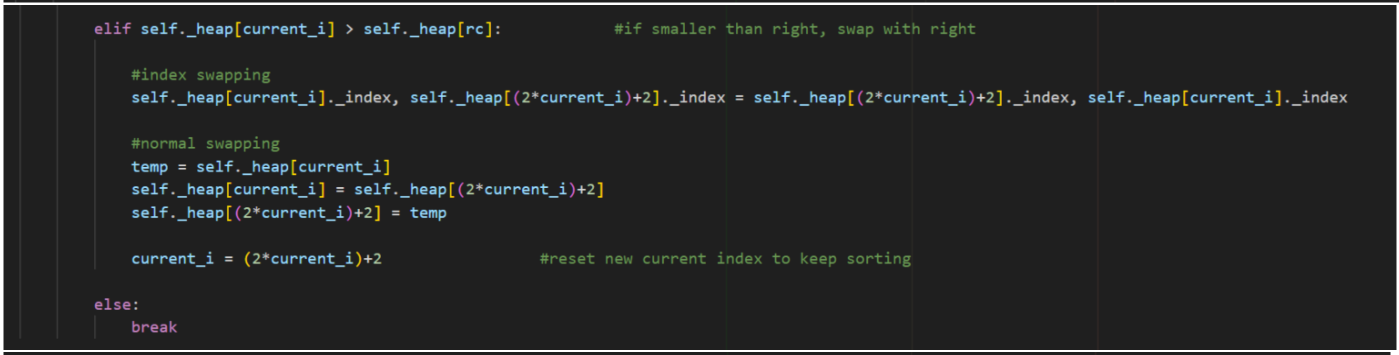
```
        elif self._heap[current_i] > self._heap[lc] and self._heap[current_i] > self._heap[rc]: #if smaller than both left and right

            if self._heap[lc] < self._heap[rc]:                    #if left greater than right, swap with left

                #index swapping
                self._heap[current_i]._index, self._heap[(2*current_i)+1]._index = self._heap[(2*current_i)+1]._index, self._heap[current_i]._index

                #normal swapping
                temp = self._heap[current_i]
                self._heap[current_i] = self._heap[(2*current_i)+1]
                self._heap[(2*current_i)+1] = temp
                current_i = (2*current_i)+1
```



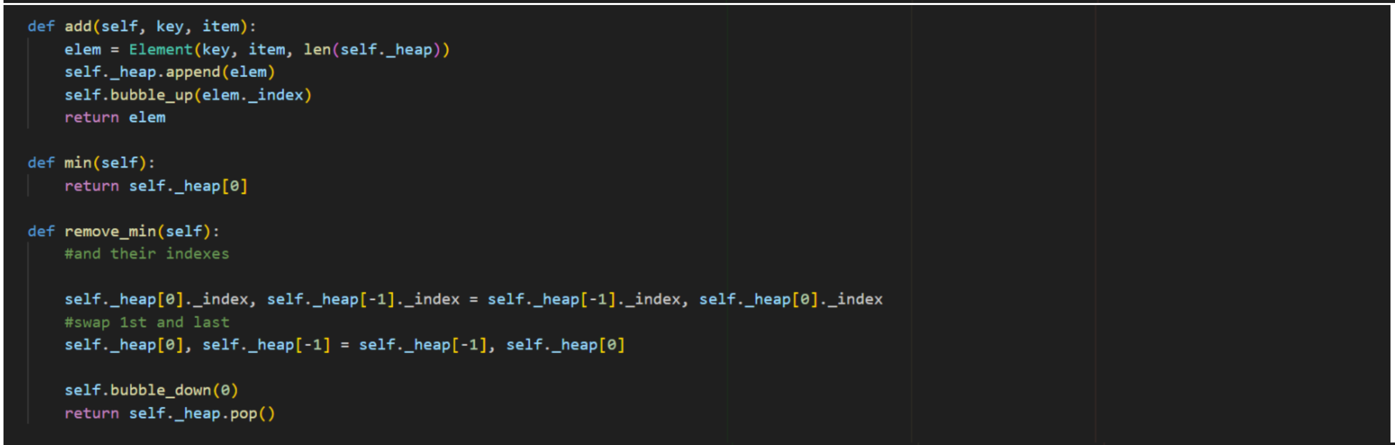
```
            else:                                      #else if right greater than left, swap with right
                #index swapping
                self._heap[current_i]._index, self._heap[(2*current_i)+2]._index = self._heap[(2*current_i)+2]._index, self._heap[current_i]._index

                #normal swapping
                temp = self._heap[current_i]
                self._heap[current_i] = self._heap[(2*current_i)+2]
                self._heap[(2*current_i)+2] = temp

                current_i = (2*current_i)+2
```



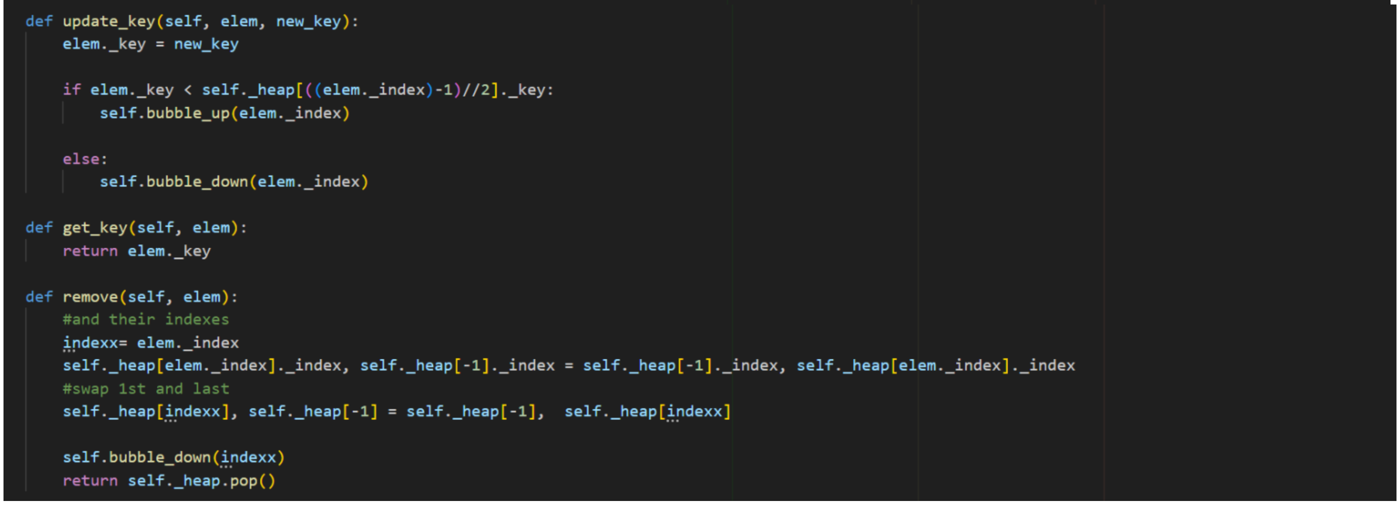
```
            elif self._heap[current_i] > self._heap[lc]:            #if smaller than left, swap with left

                #index swapping
                self._heap[current_i]._index, self._heap[(2*current_i)+1]._index = self._heap[(2*current_i)+1]._index, self._heap[current_i]._index

                #normal swapping
                temp = self._heap[current_i]
                self._heap[current_i] = self._heap[(2*current_i)+1]
                self._heap[(2*current_i)+1] = temp

                current_i = (2*current_i)+1
```



```
            elif self._heap[current_i] > self._heap[rc]:            #if smaller than right, swap with right

                #index swapping
                self._heap[current_i]._index, self._heap[(2*current_i)+2]._index = self._heap[(2*current_i)+2]._index, self._heap[current_i]._index

                #normal swapping
                temp = self._heap[current_i]
                self._heap[current_i] = self._heap[(2*current_i)+2]
                self._heap[(2*current_i)+2] = temp

                current_i = (2*current_i)+2                 #reset new current index to keep sorting

            else:
                break
```

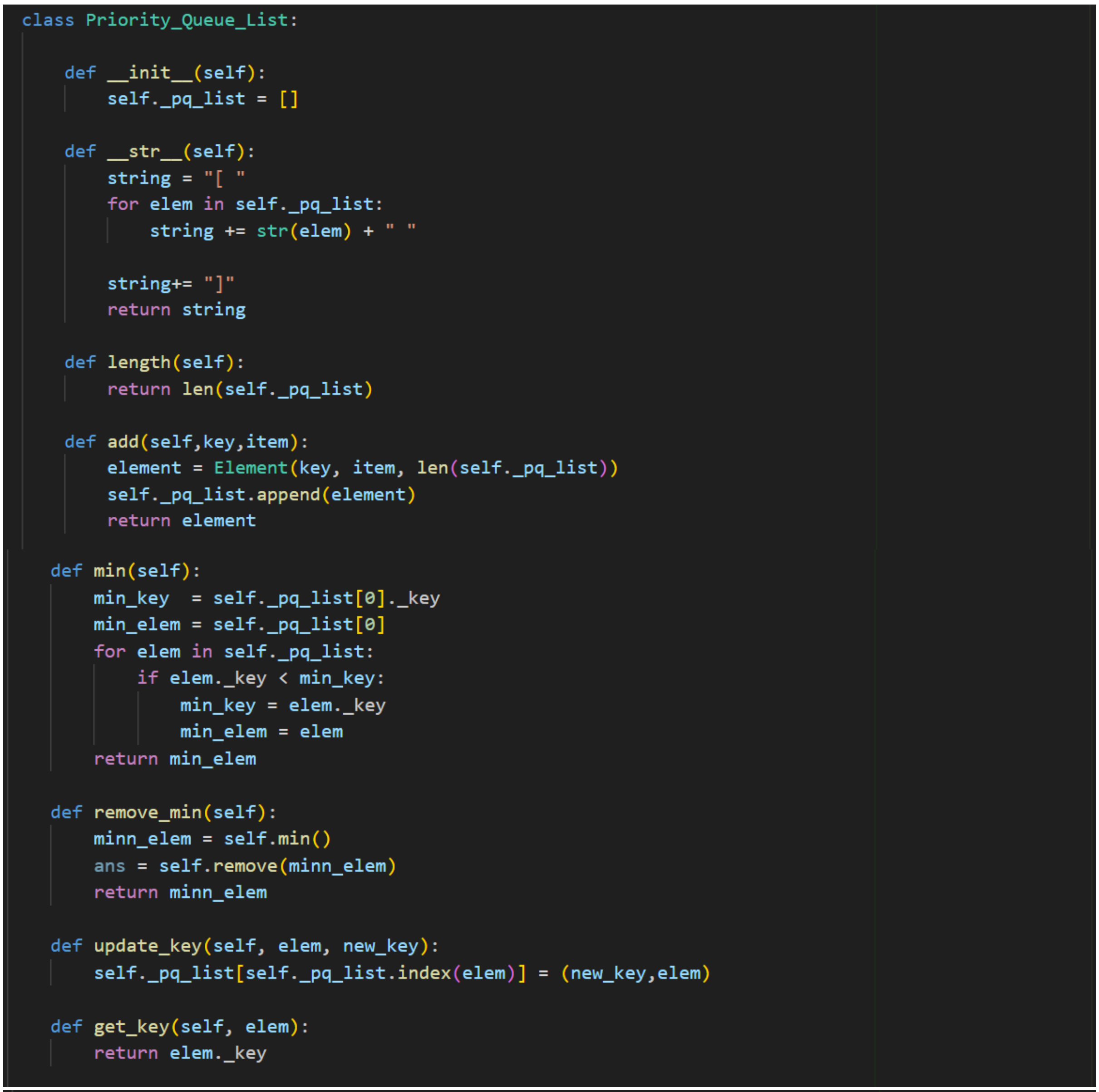


```
    def add(self, key, item):
        elem = Element(key, item, len(self._heap))
        self._heap.append(elem)
        self.bubble_up(elem._index)
        return elem

    def min(self):
        return self._heap[0]

    def remove_min(self):
        #and their indexes

        self._heap[0]._index, self._heap[-1]._index = self._heap[-1]._index, self._heap[0]._index
        #swap 1st and last
        self._heap[0], self._heap[-1] = self._heap[-1], self._heap[0]

        self.bubble_down(0)
        return self._heap.pop()
```

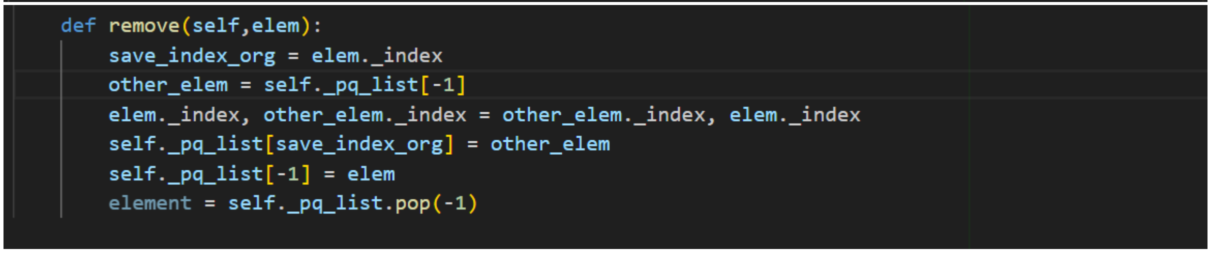


```
    def update_key(self, elem, new_key):
        elem._key = new_key

        if elem._key < self._heap[((elem._index)-1)//2]._key:
            self.bubble_up(elem._index)

        else:
            self.bubble_down(elem._index)

    def get_key(self, elem):
        return elem._key

    def remove(self, elem):
        #and their indexes
        indexx= elem._index
        self._heap[elem._index]._index, self._heap[-1]._index = self._heap[-1]._index, self._heap[elem._index]._index
        #swap 1st and last
        self._heap[indexx], self._heap[-1] = self._heap[-1],  self._heap[indexx]

        self.bubble_down(indexx)
        return self._heap.pop()
```

ADAPTABLE PRIORITY QUEUE CLASS WITH AN UNSORTED ARRAY

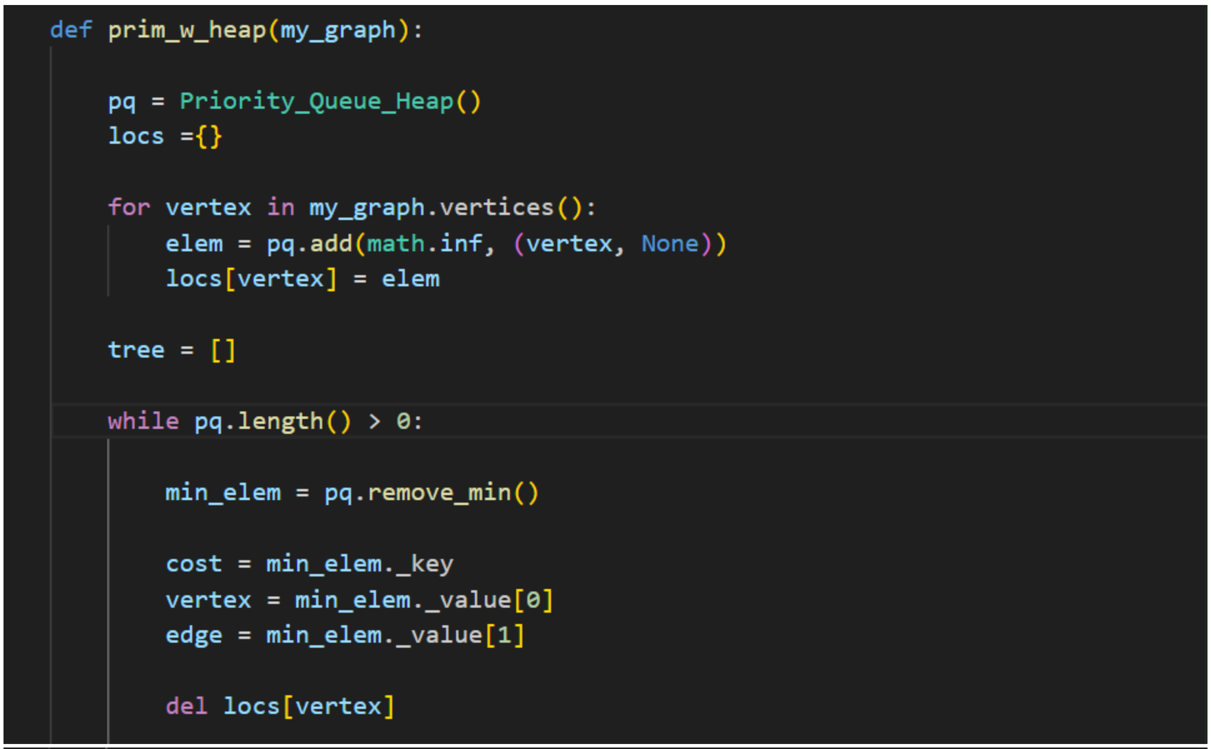


```python
class Priority_Queue_List:

    def __init__(self):
        self._pq_list = []

    def __str__(self):
        string = "[ "
        for elem in self._pq_list:
            string += str(elem) + " "

        string+= "]"
        return string

    def length(self):
        return len(self._pq_list)

    def add(self,key,item):
        element = Element(key, item, len(self._pq_list))
        self._pq_list.append(element)
        return element

    def min(self):
        min_key  = self._pq_list[0]._key
        min_elem = self._pq_list[0]
        for elem in self._pq_list:
            if elem._key < min_key:
                min_key = elem._key
                min_elem = elem
        return min_elem

    def remove_min(self):
        minn_elem = self.min()
        ans = self.remove(minn_elem)
        return minn_elem

    def update_key(self, elem, new_key):
        self._pq_list[self._pq_list.index(elem)] = (new_key,elem)

    def get_key(self, elem):
        return elem._key
```

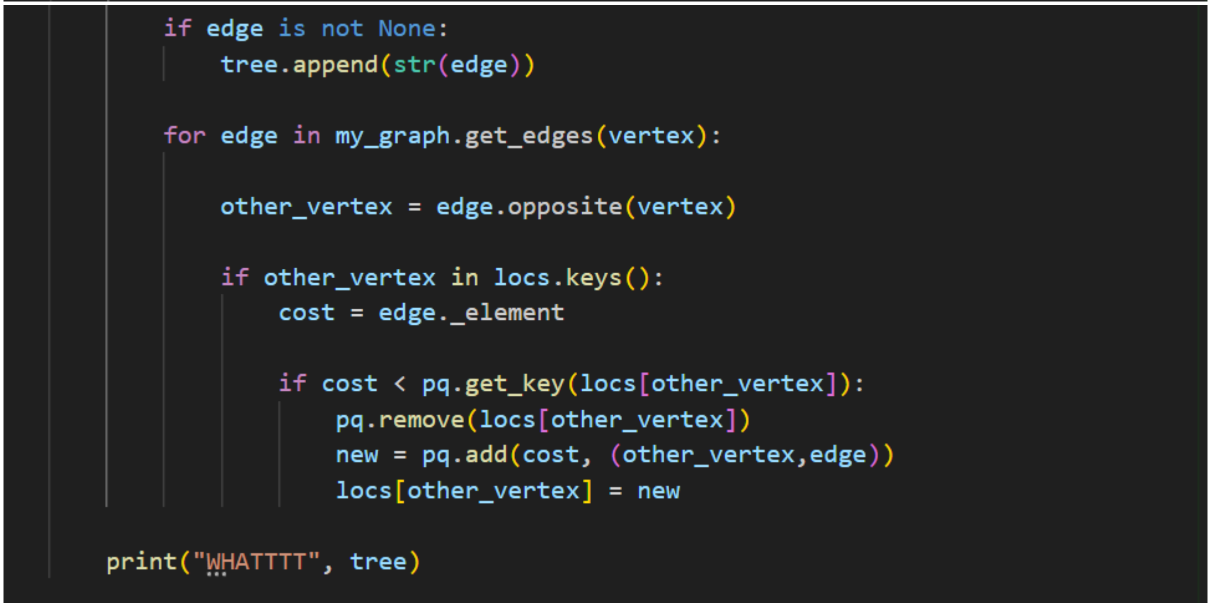


```python
    def remove(self,elem):
        save_index_org = elem._index
        other_elem = self._pq_list[-1]
        elem._index, other_elem._index = other_elem._index, elem._index
        self._pq_list[save_index_org] = other_elem
        self._pq_list[-1] = elem
        element = self._pq_list.pop(-1)
```

PRIMS IMPLEMENTATION WITH APQ WITH HEAP

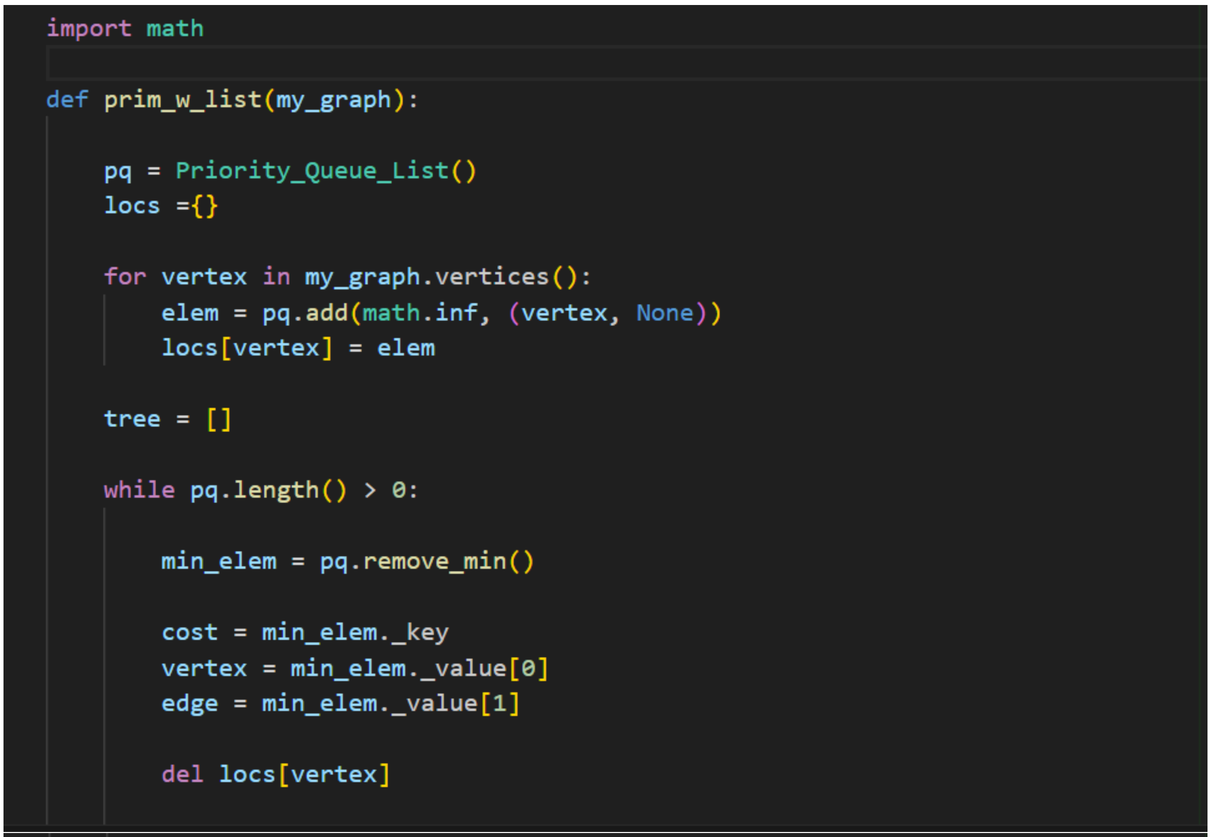


```python
def prim_w_heap(my_graph):

    pq = Priority_Queue_Heap()
    locs ={}

    for vertex in my_graph.vertices():
        elem = pq.add(math.inf, (vertex, None))
        locs[vertex] = elem

    tree = []

    while pq.length() > 0:

        min_elem = pq.remove_min()

        cost = min_elem._key
        vertex = min_elem._value[0]
        edge = min_elem._value[1]

        del locs[vertex]
```

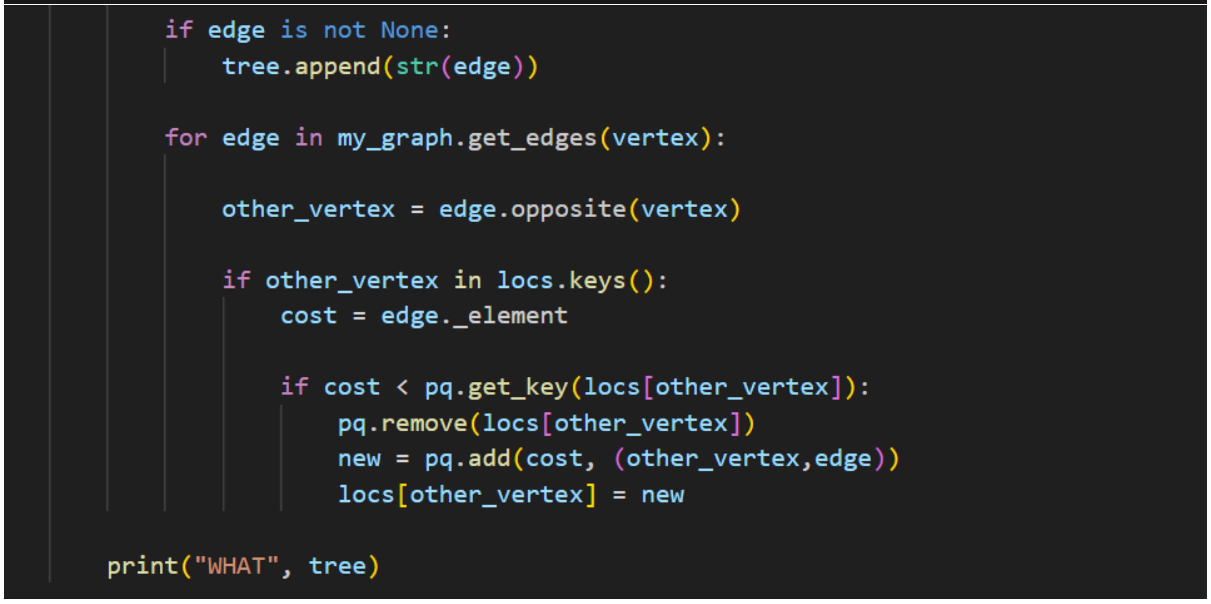


```python
        if edge is not None:
            tree.append(str(edge))

        for edge in my_graph.get_edges(vertex):

            other_vertex = edge.opposite(vertex)

            if other_vertex in locs.keys():
                cost = edge._element

                if cost < pq.get_key(locs[other_vertex]):
                    pq.remove(locs[other_vertex])
                    new = pq.add(cost, (other_vertex,edge))
                    locs[other_vertex] = new

    print("WHATTTT", tree)
```

PRIMS IMPLEMENTATION WITH APQ WITH UNSORTED ARRAY



```
import math

def prim_w_list(my_graph):

    pq = Priority_Queue_List()
    locs ={}

    for vertex in my_graph.vertices():
        elem = pq.add(math.inf, (vertex, None))
        locs[vertex] = elem

    tree = []

    while pq.length() > 0:

        min_elem = pq.remove_min()

        cost = min_elem._key
        vertex = min_elem._value[0]
        edge = min_elem._value[1]

        del locs[vertex]

```

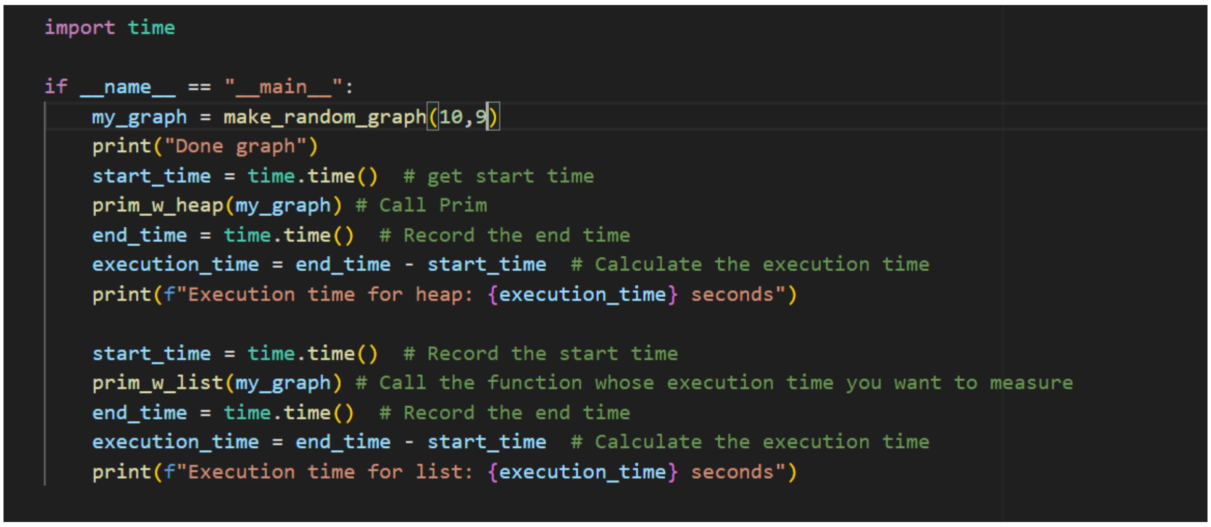


```
        if edge is not None:
            tree.append(str(edge))

        for edge in my_graph.get_edges(vertex):

            other_vertex = edge.opposite(vertex)

            if other_vertex in locs.keys():
                cost = edge._element

                if cost < pq.get_key(locs[other_vertex]):
                    pq.remove(locs[other_vertex])
                    new = pq.add(cost, (other_vertex,edge))
                    locs[other_vertex] = new

    print("WHAT", tree)
```

Please excuse the “what” before printing the tree, it was to differentiate which version of prim was executing.

EXPERIMENTS

CODE FOUND ONLINE USED TO TEST THE TIME TAKEN

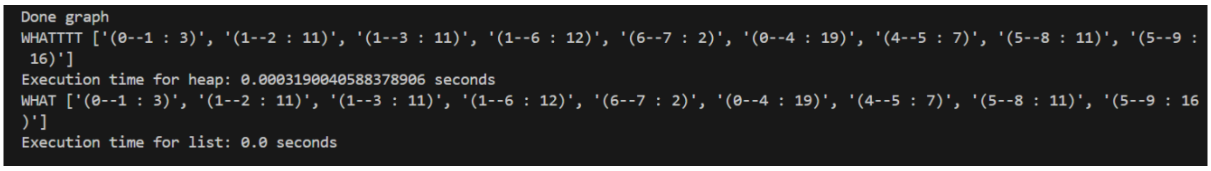


```python
import time

if __name__ == "__main__":
    my_graph = make_random_graph(10,9)
    print("Done graph")
    start_time = time.time()  # get start time
    prim_w_heap(my_graph) # Call Prim
    end_time = time.time()  # Record the end time
    execution_time = end_time - start_time  # Calculate the execution time
    print(f"Execution time for heap: {execution_time} seconds")

    start_time = time.time()  # Record the start time
    prim_w_list(my_graph) # Call the function whose execution time you want to measure
    end_time = time.time()  # Record the end time
    execution_time = end_time - start_time  # Calculate the execution time
    print(f"Execution time for list: {execution_time} seconds")
```

OUTPUTS FOR THE FOLLOWING CASES

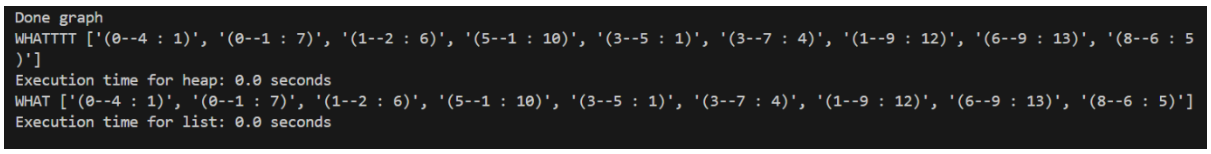
TEN VERTICES

1) **VERTICIES 10, EDGES 9**



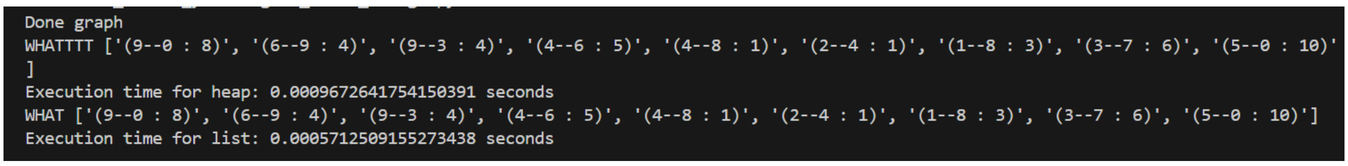
```
Done graph
WHATTTT ['(0--1 : 3)', '(1--2 : 11)', '(1--3 : 11)', '(1--6 : 12)', '(6--7 : 2)', '(0--4 : 19)', '(4--5 : 7)', '(5--8 : 11)', '(5--9 :
 16)']
Execution time for heap: 0.0003190040588378906 seconds
WHAT ['(0--1 : 3)', '(1--2 : 11)', '(1--3 : 11)', '(1--6 : 12)', '(6--7 : 2)', '(0--4 : 19)', '(4--5 : 7)', '(5--8 : 11)', '(5--9 : 16
)']
Execution time for list: 0.0 seconds
```

2) **VERTICES 10, EDGES 19**



```
Done graph
WHATTTT ['(0--4 : 1)', '(0--1 : 7)', '(1--2 : 6)', '(5--1 : 10)', '(3--5 : 1)', '(3--7 : 4)', '(1--9 : 12)', '(6--9 : 13)', '(8--6 : 5
)']
Execution time for heap: 0.0 seconds
WHAT ['(0--4 : 1)', '(0--1 : 7)', '(1--2 : 6)', '(5--1 : 10)', '(3--5 : 1)', '(3--7 : 4)', '(1--9 : 12)', '(6--9 : 13)', '(8--6 : 5)']
Execution time for list: 0.0 seconds
```

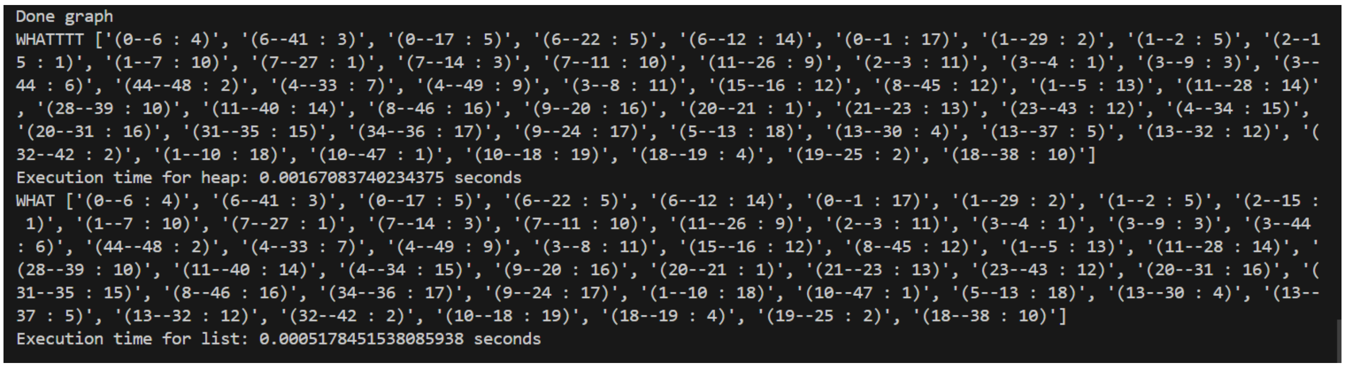
3) **VERTICES 10, EDGES 45**



```
Done graph
WHATTTT ['(9--0 : 8)', '(6--9 : 4)', '(9--3 : 4)', '(4--6 : 5)', '(4--8 : 1)', '(2--4 : 1)', '(1--8 : 3)', '(3--7 : 6)', '(5--0 : 10)'
]
Execution time for heap: 0.0009672641754150391 seconds
WHAT ['(9--0 : 8)', '(6--9 : 4)', '(9--3 : 4)', '(4--6 : 5)', '(4--8 : 1)', '(2--4 : 1)', '(1--8 : 3)', '(3--7 : 6)', '(5--0 : 10)']
Execution time for list: 0.0005712509155273438 seconds
```

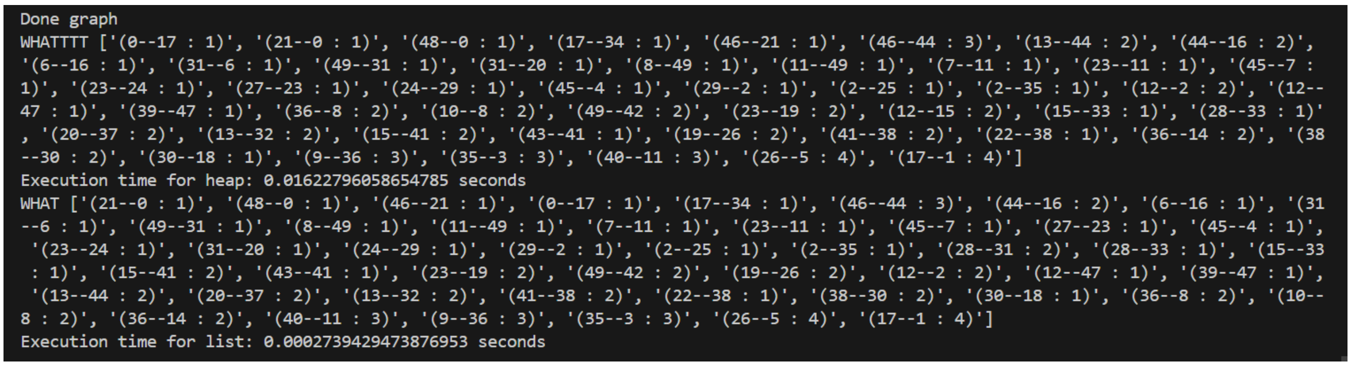
FIFTY VERTICES

1) **VERTICES 50, EDGES 49**



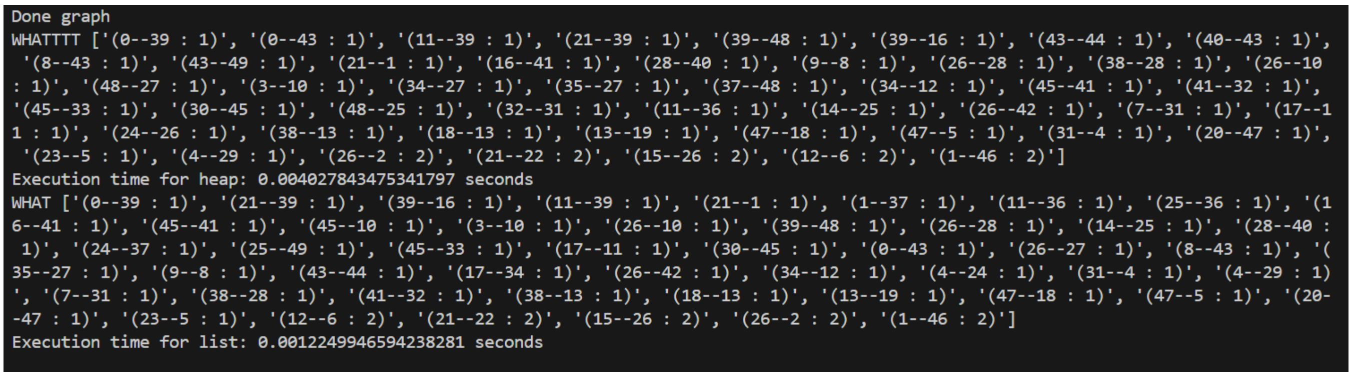
```
Done graph
WHATTTT ['(0--6 : 4)', '(6--41 : 3)', '(0--17 : 5)', '(6--22 : 5)', '(6--12 : 14)', '(0--1 : 17)', '(1--29 : 2)', '(1--2 : 5)', '(2--1
5 : 1)', '(1--7 : 10)', '(7--27 : 1)', '(7--14 : 3)', '(7--11 : 10)', '(11--26 : 9)', '(2--3 : 11)', '(3--4 : 1)', '(3--9 : 3)', '(3--
44 : 6)', '(44--48 : 2)', '(4--33 : 7)', '(4--49 : 9)', '(3--8 : 11)', '(15--16 : 12)', '(8--45 : 12)', '(1--5 : 13)', '(11--28 : 14)'
, '(28--39 : 10)', '(11--40 : 14)', '(8--46 : 16)', '(9--20 : 16)', '(20--21 : 1)', '(21--23 : 13)', '(23--43 : 12)', '(4--34 : 15)',
'(20--31 : 16)', '(31--35 : 15)', '(34--36 : 17)', '(9--24 : 17)', '(5--13 : 18)', '(13--30 : 4)', '(13--37 : 5)', '(13--32 : 12)', '(
32--42 : 2)', '(1--10 : 18)', '(10--47 : 1)', '(10--18 : 19)', '(18--19 : 4)', '(19--25 : 2)', '(18--38 : 10)']
Execution time for heap: 0.00167083740234375 seconds
WHAT ['(0--6 : 4)', '(6--41 : 3)', '(0--17 : 5)', '(6--22 : 5)', '(6--12 : 14)', '(0--1 : 17)', '(1--29 : 2)', '(1--2 : 5)', '(2--15 :
 1)', '(1--7 : 10)', '(7--27 : 1)', '(7--14 : 3)', '(7--11 : 10)', '(11--26 : 9)', '(2--3 : 11)', '(3--4 : 1)', '(3--9 : 3)', '(3--44
: 6)', '(44--48 : 2)', '(4--33 : 7)', '(4--49 : 9)', '(3--8 : 11)', '(15--16 : 12)', '(8--45 : 12)', '(1--5 : 13)', '(11--28 : 14)', '
(28--39 : 10)', '(11--40 : 14)', '(4--34 : 15)', '(9--20 : 16)', '(20--21 : 1)', '(21--23 : 13)', '(23--43 : 12)', '(20--31 : 16)', '(
31--35 : 15)', '(8--46 : 16)', '(34--36 : 17)', '(9--24 : 17)', '(1--10 : 18)', '(10--47 : 1)', '(5--13 : 18)', '(13--30 : 4)', '(13--
37 : 5)', '(13--32 : 12)', '(32--42 : 2)', '(10--18 : 19)', '(18--19 : 4)', '(19--25 : 2)', '(18--38 : 10)']
Execution time for list: 0.0005178451538085938 seconds
```

2) **VERTICES 50, EDGES 500**



```
Done graph
WHATTTT ['(0--17 : 1)', '(21--0 : 1)', '(48--0 : 1)', '(17--34 : 1)', '(46--21 : 1)', '(46--44 : 3)', '(13--44 : 2)', '(44--16 : 2)',
'(6--16 : 1)', '(31--6 : 1)', '(49--31 : 1)', '(31--20 : 1)', '(8--49 : 1)', '(11--49 : 1)', '(7--11 : 1)', '(23--11 : 1)', '(45--7 :
1)', '(23--24 : 1)', '(27--23 : 1)', '(24--29 : 1)', '(45--4 : 1)', '(29--2 : 1)', '(2--25 : 1)', '(2--35 : 1)', '(12--2 : 2)', '(12--
47 : 1)', '(39--47 : 1)', '(36--8 : 2)', '(10--8 : 2)', '(49--42 : 2)', '(23--19 : 2)', '(12--15 : 2)', '(15--33 : 1)', '(28--33 : 1)'
, '(20--37 : 2)', '(13--32 : 2)', '(15--41 : 2)', '(43--41 : 1)', '(19--26 : 2)', '(41--38 : 2)', '(22--38 : 1)', '(36--14 : 2)', '(38
--30 : 2)', '(30--18 : 1)', '(9--36 : 3)', '(35--3 : 3)', '(40--11 : 3)', '(26--5 : 4)', '(17--1 : 4)']
Execution time for heap: 0.01622796058654785 seconds
WHAT ['(21--0 : 1)', '(48--0 : 1)', '(46--21 : 1)', '(0--17 : 1)', '(17--34 : 1)', '(46--44 : 3)', '(44--16 : 2)', '(6--16 : 1)', '(31
--6 : 1)', '(49--31 : 1)', '(8--49 : 1)', '(11--49 : 1)', '(7--11 : 1)', '(23--11 : 1)', '(45--7 : 1)', '(27--23 : 1)', '(45--4 : 1)',
 '(23--24 : 1)', '(31--20 : 1)', '(24--29 : 1)', '(29--2 : 1)', '(2--25 : 1)', '(2--35 : 1)', '(28--31 : 2)', '(28--33 : 1)', '(15--33
 : 1)', '(15--41 : 2)', '(43--41 : 1)', '(23--19 : 2)', '(49--42 : 2)', '(19--26 : 2)', '(12--2 : 2)', '(12--47 : 1)', '(39--47 : 1)',
 '(13--44 : 2)', '(20--37 : 2)', '(13--32 : 2)', '(41--38 : 2)', '(22--38 : 1)', '(38--30 : 2)', '(30--18 : 1)', '(36--8 : 2)', '(10--
8 : 2)', '(36--14 : 2)', '(40--11 : 3)', '(9--36 : 3)', '(35--3 : 3)', '(26--5 : 4)', '(17--1 : 4)']
Execution time for list: 0.0002739429473876953 seconds
```

3) **VERTICES 50, EDGES 1225**

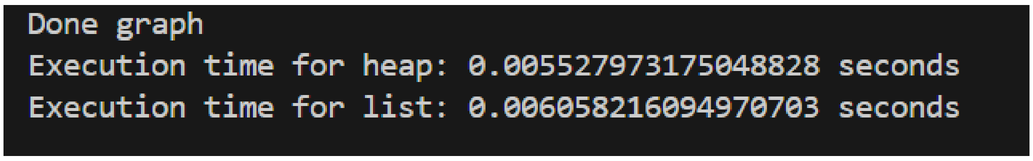


```
Done graph
WHATTTT ['(0--39 : 1)', '(0--43 : 1)', '(11--39 : 1)', '(21--39 : 1)', '(39--48 : 1)', '(39--16 : 1)', '(43--44 : 1)', '(40--43 : 1)',
 '(8--43 : 1)', '(43--49 : 1)', '(21--1 : 1)', '(16--41 : 1)', '(28--40 : 1)', '(9--8 : 1)', '(26--28 : 1)', '(38--28 : 1)', '(26--10
: 1)', '(48--27 : 1)', '(3--10 : 1)', '(34--27 : 1)', '(35--27 : 1)', '(37--48 : 1)', '(34--12 : 1)', '(45--41 : 1)', '(41--32 : 1)',
'(45--33 : 1)', '(30--45 : 1)', '(48--25 : 1)', '(32--31 : 1)', '(11--36 : 1)', '(14--25 : 1)', '(26--42 : 1)', '(7--31 : 1)', '(17--1
1 : 1)', '(24--26 : 1)', '(38--13 : 1)', '(18--13 : 1)', '(13--19 : 1)', '(47--18 : 1)', '(47--5 : 1)', '(31--4 : 1)', '(20--47 : 1)',
 '(23--5 : 1)', '(4--29 : 1)', '(26--2 : 2)', '(21--22 : 2)', '(15--26 : 2)', '(12--6 : 2)', '(1--46 : 2)']
Execution time for heap: 0.004027843475341797 seconds
WHAT ['(0--39 : 1)', '(21--39 : 1)', '(39--16 : 1)', '(11--39 : 1)', '(21--1 : 1)', '(1--37 : 1)', '(11--36 : 1)', '(25--36 : 1)', '(1
6--41 : 1)', '(45--41 : 1)', '(45--10 : 1)', '(3--10 : 1)', '(26--10 : 1)', '(39--48 : 1)', '(26--28 : 1)', '(14--25 : 1)', '(28--40 :
 1)', '(24--37 : 1)', '(25--49 : 1)', '(45--33 : 1)', '(17--11 : 1)', '(30--45 : 1)', '(0--43 : 1)', '(26--27 : 1)', '(8--43 : 1)', '(
35--27 : 1)', '(9--8 : 1)', '(43--44 : 1)', '(17--34 : 1)', '(26--42 : 1)', '(34--12 : 1)', '(4--24 : 1)', '(31--4 : 1)', '(4--29 : 1)
', '(7--31 : 1)', '(38--28 : 1)', '(41--32 : 1)', '(38--13 : 1)', '(18--13 : 1)', '(13--19 : 1)', '(47--18 : 1)', '(47--5 : 1)', '(20-
-47 : 1)', '(23--5 : 1)', '(12--6 : 2)', '(21--22 : 2)', '(15--26 : 2)', '(26--2 : 2)', '(1--46 : 2)']
Execution time for list: 0.0012249946594238281 seconds
```

<u>FIVE HUNDRED VERTICES</u>

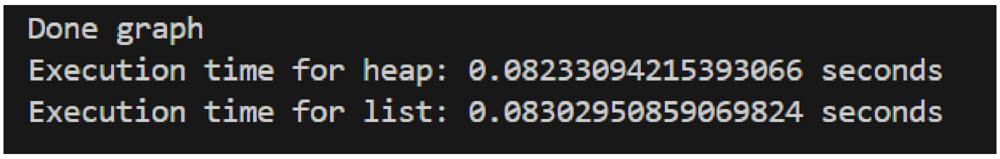
(I stopped printing the tree as it became too large)

1) **500 VERTICES 499 EDGES**



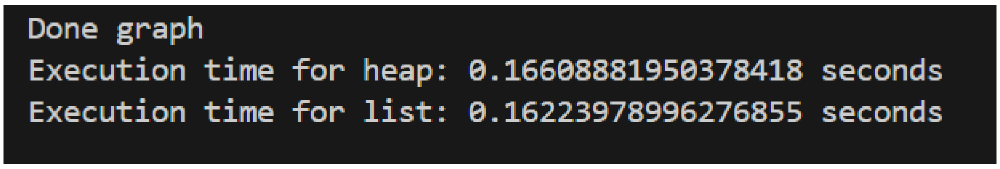
```
Done graph
Execution time for heap: 0.005527973175048828 seconds
Execution time for list: 0.006058216094970703 seconds
```

2) **500 VERTICES 63,000 EDGES**



```
Done graph
Execution time for heap: 0.08233094215393066 seconds
Execution time for list: 0.08302950859069824 seconds
```

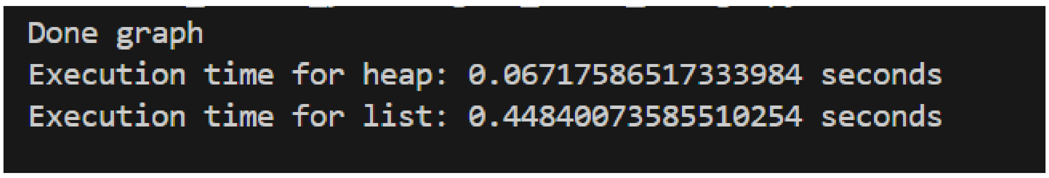
3) **500 VERTICES 124,750 EDGES**



```
Done graph
Execution time for heap: 0.16608881950378418 seconds
Execution time for list: 0.16223978996276855 seconds
```

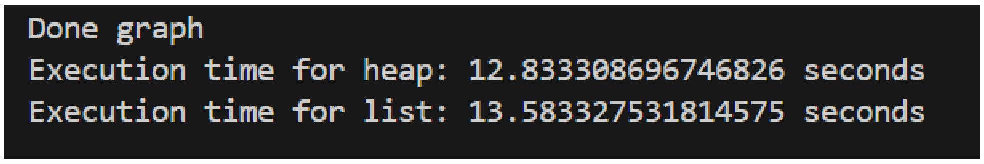
<u>FIVE THOUSAND VERTICES</u>

1) **5000 VERTICES, 4999 EDGES**



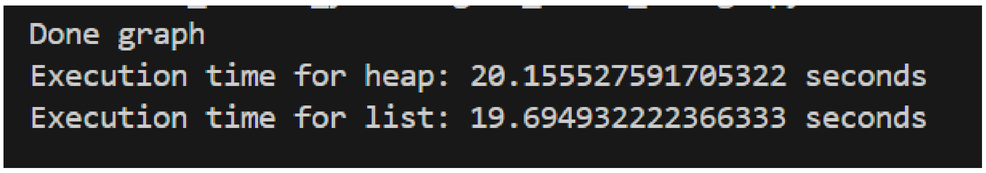
```
Done graph
Execution time for heap: 0.06717586517333984 seconds
Execution time for list: 0.44840073585510254 seconds
```

2) **5000 VERTICES 6,250,000 EDGES**



```
Done graph
Execution time for heap: 12.833308696746826 seconds
Execution time for list: 13.583327531814575 seconds
```

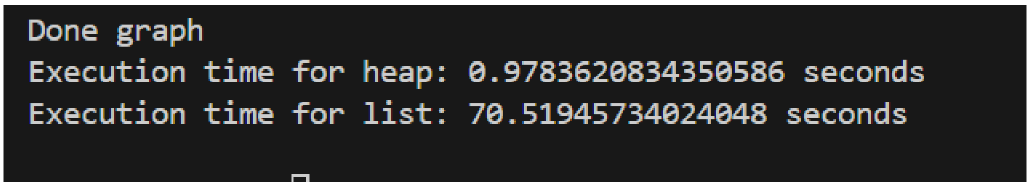
3) **5000 VERTICES 12,497,500 EDGES**



```
Done graph
Execution time for heap: 20.155527591705322 seconds
Execution time for list: 19.694932222366333 seconds
```

FIFTY THOUSAND VERTICES

1) **50,000 VERTICES 49,999 EDGES**



```
Done graph
Execution time for heap: 0.9783620834350586 seconds
Execution time for list: 70.51945734024048 seconds
```

CONCLUSIONS

For my experiments I tested my implementations on a variety of numbers of vertices. For each number of vertices, I made sure to test a very sparse graph, a max density graph and one that is approximately halfway between the other two to get a full picture.

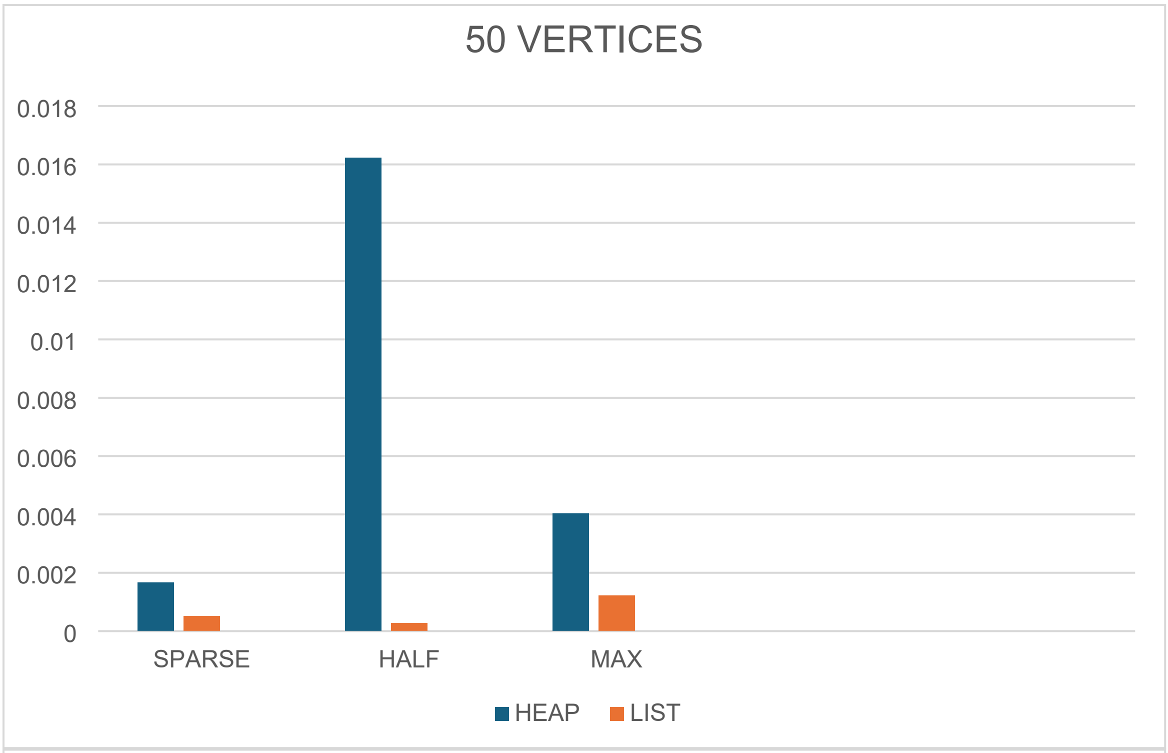
For the tests on very small graphs, the difference between them was very negligible (<0.0005 for the max density graph) and the time taken to execute each one was so small, any differences couldn't be accurately measured (no observable differences in the sparse and half-full graphs).

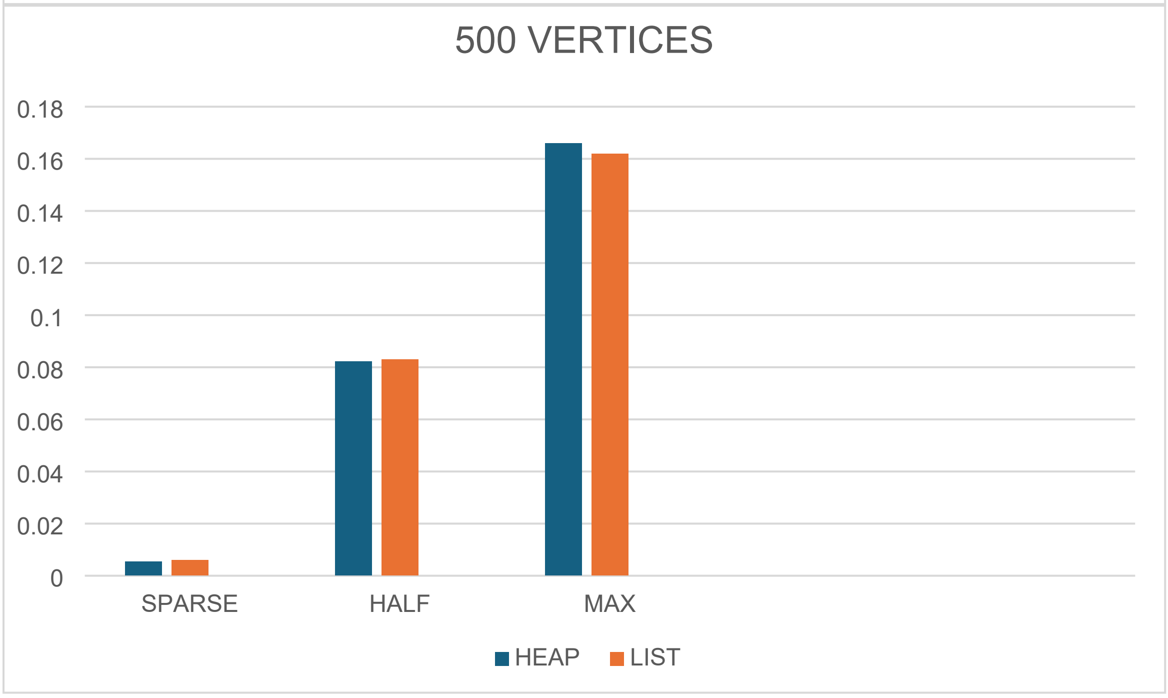
We only start to see the real affects of the different implementations when we increase the number of vertices. For fifty vertices, the list is slightly more efficient each time, the heap implementation is 3.2 times slower than the list one in the sparse and max density graph. However, in the halfway full graph, the heap implementation is 59 times slower than the list, a significant difference. For 500 vertices, both implementations run almost at the same rate. The list is 1.09 times the heap implementation for the sparse graph, 1.01 times the heap implementation for the middle graph and the heap implementation is 1.02 times the list implementation for the max density graph. For 5,000 vertices, there is only a difference for the sparse graph, the list implementation is 6.7 times slower than the heap, but for the max density and half-full graphs, the difference between them is negligible

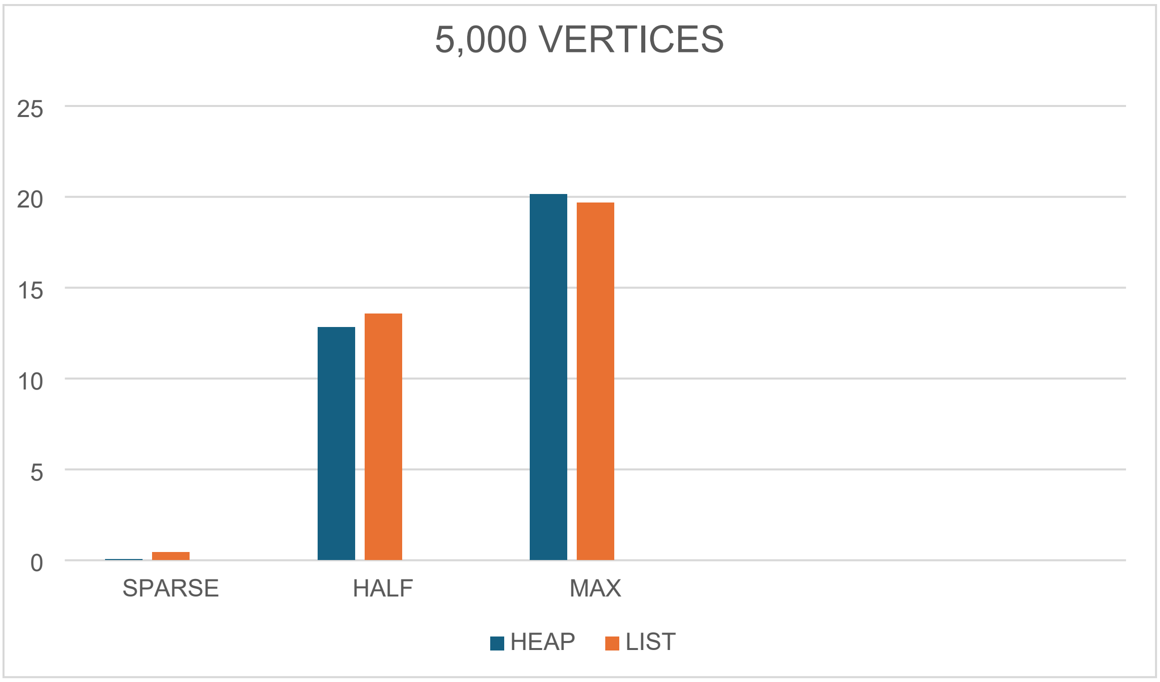
For a visual aid of my results, please see below my conclusion for graphs depicting the different runtimes of each implementation on different numbers of vertices.

Due to this I am led to the conclusion, that for smaller graphs the implementation does not matter, unless of course u have a very small, not very densely filled graph and in that case the list is much better. For a large graph size such as 50,000 vertices, the difference is astounding with the heap being so much faster than the list for a sparse graph. However, as the graph density increases the list's efficiency does too.

In conclusion, you should be aware of which type of graph, sparse/dense you will be running your algorithm on and pick the implementation based on that.


50 VERTICES
0.018
0.016
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
SPARSE
HALF
MAX
HEAP
LIST


500 VERTICES
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
SPARSE
HALF
MAX
HEAP
LIST


5,000 VERTICES
25
20
15
10
5
0
SPARSE
HALF
MAX
HEAP
LIST