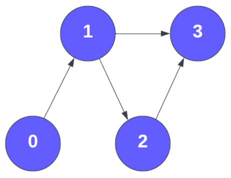
Saleh Alkhalifa
10/10/2022

CS 5800
Module 5

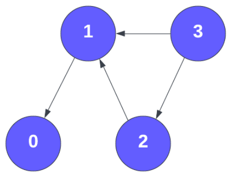
Homework#5

Problem 1 Adjacency Lists (10 pts)

- **The reverse of a directed graph *G = (V , E)* is another directed graph G^R = *(V* , E^R*)* on the same vertex set, but with all edges reversed; that is, E^R= *{(v, u) : (u, v) ∈ E}*.**
- **Given the adjacency list for G, give a linear time algorithm to compute the adjacency list for G^R.**



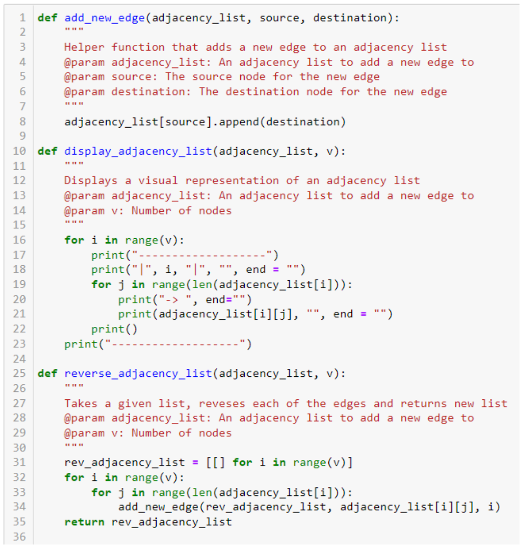
Original Graph



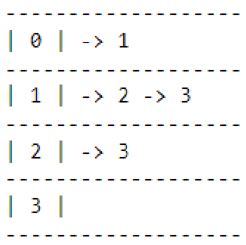
Reverse Graph

Answer:

- We begin creating a new empty graph adjacency list to represent $G^R = (V, E^R)$
- We can then create a loop and iterate over the current list of edges in the directed graph G
- In the loop, for reach element in the list which represents an edge u, we obtain (u, w) we add the opposite of that (w, u) edge to the previously created list
- Finally, we return the list which represents G^R

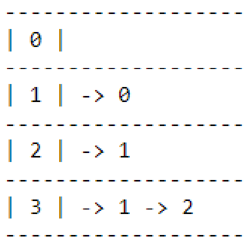


```python
def add_new_edge(adjacency_list, source, destination):
    """
    Helper function that adds a new edge to an adjacency list
    @param adjacency_list: An adjacency list to add a new edge to
    @param source: The source node for the new edge
    @param destination: The destination node for the new edge
    """
    adjacency_list[source].append(destination)

def display_adjacency_list(adjacency_list, v):
    """
    Displays a visual representation of an adjacency list
    @param adjacency_list: An adjacency list to add a new edge to
    @param v: Number of nodes
    """
    for i in range(v):
        print("------------------")
        print("|", i, "|", "", end = "")
        for j in range(len(adjacency_list[i])):
            print("-> ", end="")
            print(adjacency_list[i][j], "", end = "")
        print()
    print("------------------")

def reverse_adjacency_list(adjacency_list, v):
    """
    Takes a given list, reveses each of the edges and returns new list
    @param adjacency_list: An adjacency list to add a new edge to
    @param v: Number of nodes
    """
    rev_adjacency_list = [[] for i in range(v)]
    for i in range(v):
        for j in range(len(adjacency_list[i])):
            add_new_edge(rev_adjacency_list, adjacency_list[i][j], i)
    return rev_adjacency_list

```



```
------------------
| 0 | -> 1
------------------
| 1 | -> 2 -> 3
------------------
| 2 | -> 3
------------------
| 3 |
------------------
```

Original



```
------------------
| 0 |
------------------
| 1 | -> 0
------------------
| 2 | -> 1
------------------
| 3 | -> 1 -> 2
------------------
```

Reversed

Attached Python code can be run via a Jupyter Notebook, showing the operations used here.

In the end, we will have traversed the list/graph once, inserted each of the reversed each of the edges once, and returned the reversed graph once.

Therefore the time complexity will be in linear time O(|V| + |E|)

Homework#5

Problem 2 Median - Heap (10 pts)

- **Given a set of numbers, its median, informally, is the "halfway point" of the set.**
- **When the set's size n is odd, the median is unique, occurring index i = (n + 1)/2.**
- **When n is even, there are two medians, occurring at i = n/2 and i = n/2 + 1, which are called the "lower median" and "upper median," respectively.**
- **Regardless of the parity of n, medians occur at i = ⌊(n + 1)/2⌋ (the lower median) and i = ⌈(n + 1)/2⌉ (the upper median).**
- **For simplicity in this question, we use the phrase "the median" to refer to the lower median. So we care about the i = ⌊(n + 1)/2⌋ position.**

Design and describe, in detail, a data structure *Median − Heap* to maintain a collection of numbers S that supports *Build*(S), *Insert*(x), *Extract*(), and *Peek*() operations, defined as follows:

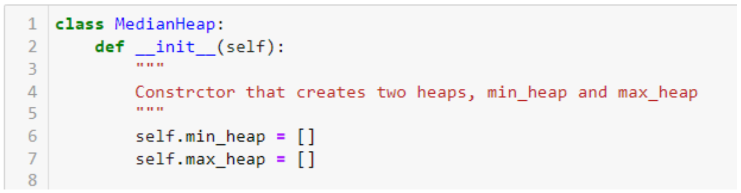
- ***Build*(S): Produces, in linear time, a data structure *Median − Heap* from an unordered input array S. For describing *Build*(S), you can assume access to the procedure *Find_Median*(S), which finds the median of S in linear time.**
- ***Insert*(x): Insert element x into *Median − Heap* in O(log n) time.**
- ***Peek*(): Returns, in O(1) time, the value of the median of *Median − Heap*.**
- ***Extract*(): Remove and return, in O(log n) time, the value of the median element in *Median − Heap*.**

Note: Pseudocode is allowed. Code is not required but can be used to describe the data structure. If you do code, please limit to java python programming languages, and submit .java/.py files (respectively).

Answer:

There are multiple methods to accomplish this given the constraints outlined above, however, I will use a method (binary heap) I used for a previous project at work using two distinct heaps to accomplish the same objective. The main method here is using two separate heaps, min_heap and max_heap, each of which containing half of the numbers of our set of numbers n.

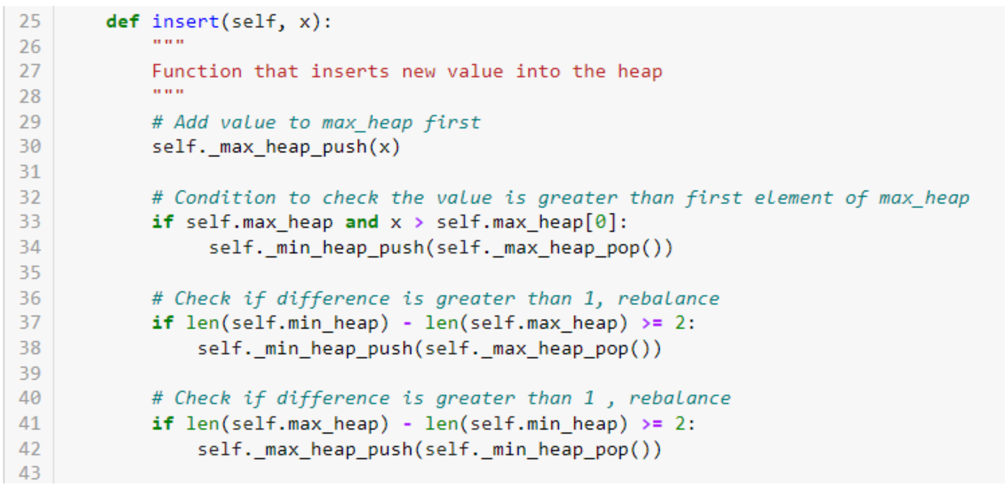
1. **Build(S):**
   - We begin with two heaps, min_heap and max_heap the construction for which relies on the list of elements.
   - We first use our given function $Find\_Median(S)$ to get the median in linear time.
   - Next, we add the elements of our list to one of the heaps depending on whether the value is larger or smaller relative to the $median$.
   - Numbers smaller than the median are stored in the max_heap, whereas the numbers larger than the median are stored in the min_heap. This will ultimately happen in linear time since it depends on the values of the list.



```python
class MedianHeap:
    def __init__(self):
        """
        Constrctor that creates two heaps, min_heap and max_heap
        """
        self.min_heap = []
        self.max_heap = []
```

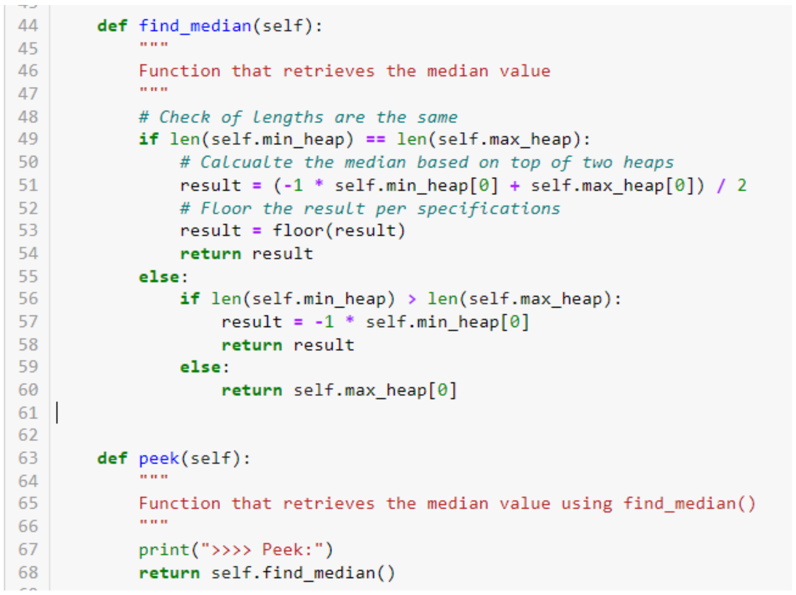
2. **Insert(x):**
   - The process of inserting a new element can be done with relative ease.
   - First, we must compare the new element we wish to insert, to the top of each of the heaps. This will help determine, via a comparison, which heap to add the element to. We can also add the value to the max_heap, and then compare with several conditionals:
     - If the value is greater than the top element of max_heap, the we push the value to max_heap

  - If the difference between the two is greater than 1, then we pop a value off the max heap and push to the min heap.
  - If the opposite is true, then we pop a value off the min_heap and add to max_heap.
- If the size of the two heaps are the same, then the process is complete.
- This allows us to achieve $O(log(n))$.



```python
    def insert(self, x):
        """
        Function that inserts new value into the heap
        """
        # Add value to max_heap first
        self._max_heap_push(x)

        # Condition to check the value is greater than first element of max_heap
        if self.max_heap and x > self.max_heap[0]:
             self._min_heap_push(self._max_heap_pop())

        # Check if difference is greater than 1, rebalance
        if len(self.min_heap) - len(self.max_heap) >= 2:
            self._min_heap_push(self._max_heap_pop())

        # Check if difference is greater than 1 , rebalance
        if len(self.max_heap) - len(self.min_heap) >= 2:
            self._max_heap_push(self._min_heap_pop())
```

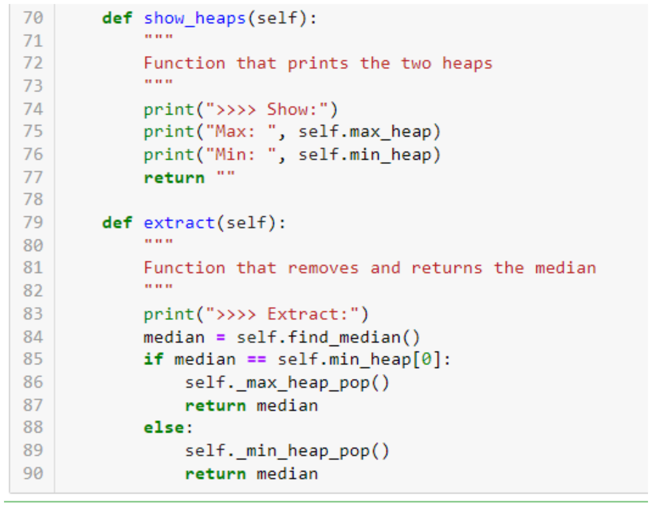
3. **Peek()**:
- Since we have the two separate heaps, we can take a look at the sizes of the two and compare.
- If the sum of the sizes is odd, we can return the top element of the heap with the larger size, effectively representing the $median$ or the half-way point.
- However, if the sum of the two sizes is even, then we can return the smaller of the two top numbers, per the specification listed in this assignment. Since we do not need to go through all the elements, but simply access these points as needed, we can accomplish this step in $O(1)$ time.



```python
    def find_median(self):
        """
        Function that retrieves the median value
        """
        # Check of lengths are the same
        if len(self.min_heap) == len(self.max_heap):
            # Calcualte the median based on top of two heaps
            result = (-1 * self.min_heap[0] + self.max_heap[0]) / 2
            # Floor the result per specifications
            result = floor(result)
            return result
        else:
            if len(self.min_heap) > len(self.max_heap):
                result = -1 * self.min_heap[0]
                return result
            else:
                return self.max_heap[0]


    def peek(self):
        """
        Function that retrieves the median value using find_median()
        """
        print(">>>> Peek:")
        return self.find_median()
```

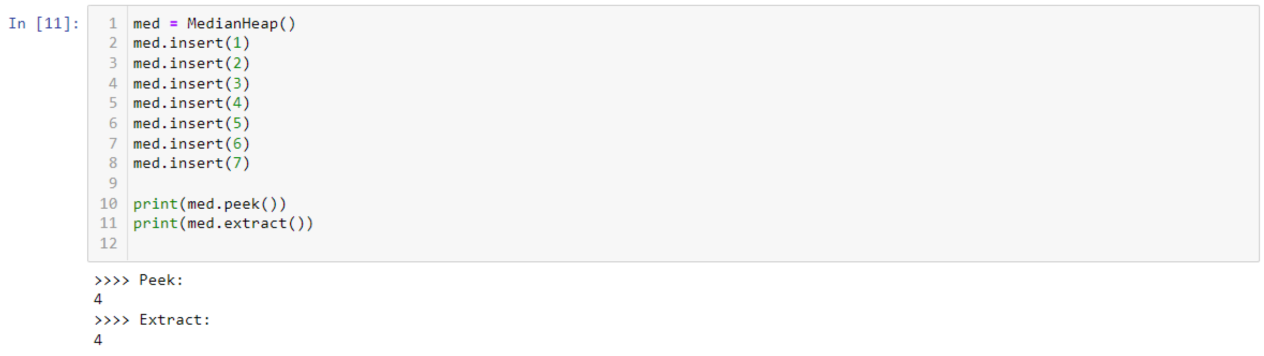
4. **Extract()**:

- In order to remove the Median element, we can use the $Peek()$ function outlined above to identify the element in $O(1)$ time, and then remove it from the list given its location as the top element of the min_heap or max_heap.
- Similarly to the $insert(x)$ function, we will need to account for the difference in size, if the size of one heap is larger than the other by more than one.
- This allows us to achieve $O(log(n))$.

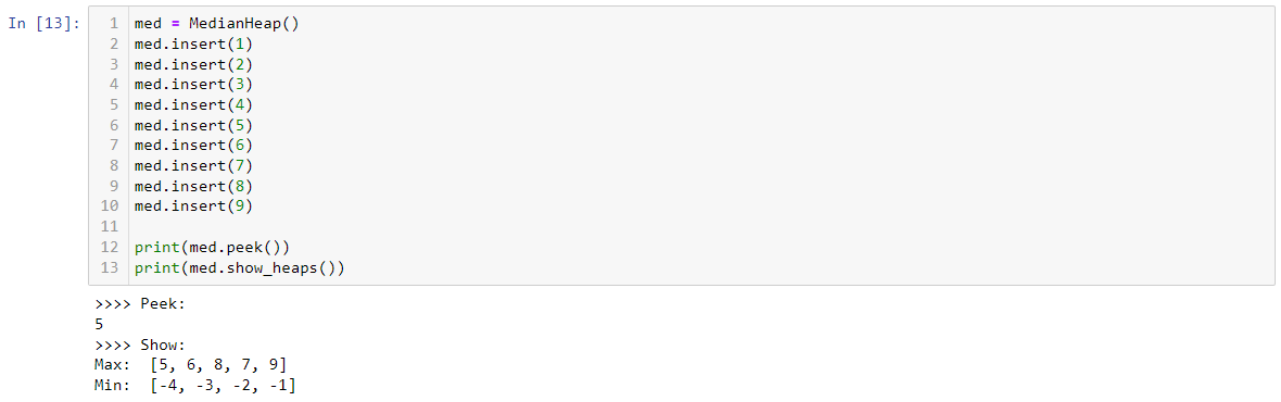


```
    def show_heaps(self):
        """
        Function that prints the two heaps
        """
        print(">>>> Show:")
        print("Max: ", self.max_heap)
        print("Min: ", self.min_heap)
        return ""

    def extract(self):
        """
        Function that removes and returns the median
        """
        print(">>>> Extract:")
        median = self.find_median()
        if median == self.min_heap[0]:
            self._max_heap_pop()
            return median
        else:
            self._min_heap_pop()
            return median
```

Some Code Tests:



```
In [11]:
med = MedianHeap()
med.insert(1)
med.insert(2)
med.insert(3)
med.insert(4)
med.insert(5)
med.insert(6)
med.insert(7)

print(med.peek())
print(med.extract())

>>>> Peek:
4
>>>> Extract:
4
```

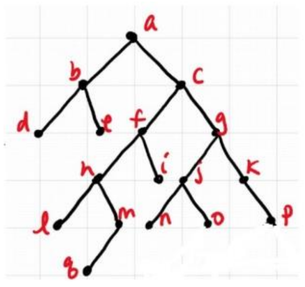


```
In [13]:
med = MedianHeap()
med.insert(1)
med.insert(2)
med.insert(3)
med.insert(4)
med.insert(5)
med.insert(6)
med.insert(7)
med.insert(8)
med.insert(9)

print(med.peek())
print(med.show_heaps())

>>>> Peek:
5
>>>> Show:
Max:  [5, 6, 8, 7, 9]
Min:  [-4, -3, -2, -1]
```

Full code can be found included in the submitted assignment

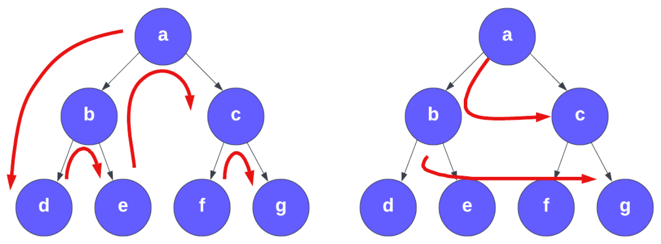
Homework#5

Problem 3 BFS/DFS Tree (20 pts total)

- **3.1 (5pt):**
  - **Consider the following undirected binary tree T with 17 vertices.**
  - 

  - **Starting with the root vertex a, we can use Breadth-First Search (BFS) or Depth-First Search (DFS) to pass through all of the vertices in this tree.**
  - **Whenever we have more than one option, we always pick the vertex that appears earlier in the alphabet. For example, from vertex a, we go to b instead of c.**
  - **Clearly explain the difference between Breadth-First Search and Depth-First Search, and determine the order in which the 17 vertices are reached using each algorithm.**

Answer:



Depth-First Search

Bredth-First Search

When comparing **Depth-First Search (DFS)** and **Breadth-First Search (BFS)** from a high level, we can note that DFS is an edge-based method whereas BFS is more of a vertex-based method. Let us explore this in more detail.

When it comes to **DFS**, we generally consider the stack data structure which operates on a Last In First Out (LIFO) type of methodology. The DFS method of traversing begins at a given node, and proceeds to traverse the left subtree first before traversing the others towards the right. The main idea here is exploring the subtrees as deep as possible before moving on. This edge-based methodology operates by visiting the vertices that were pushed into the stack. The time complexity of DFS is O(V+E) when using an adjacency list, or O(V^2) when using an adjacency matrix.

On the other hand, when it comes to **BFS**, we generally consider a queue data structure that operates on a First In First Out (FIFO) type of methodology. The BFS method of traversing begins at a given node, and proceeds to traverse through a single level of children nodes before proceeding to travers a lower level. The time complexity of BFS is O(V+E) when using an adjacency list, or O(V^2) when using an adjacency matrix.

DFS Order: a, b, d, e, c, f, h, l, m, q, **I**, g, j, n, o, k, p

BFS Order: a, b, c, d, e, f, g, h, **I**, j, k, l, m, n, o, p, q

Homework#5

- **3.2 (5pt):**
  - **Let T be an undirected binary tree with n vertices. Write an efficient algorithm that shows how you can walk through the tree by crossing each edge of T exactly twice: once in each direction.**
  - **Clearly explain how your algorithm works, why each edge is guaranteed to be crossed exactly twice, and determine the running time of your algorithm.**

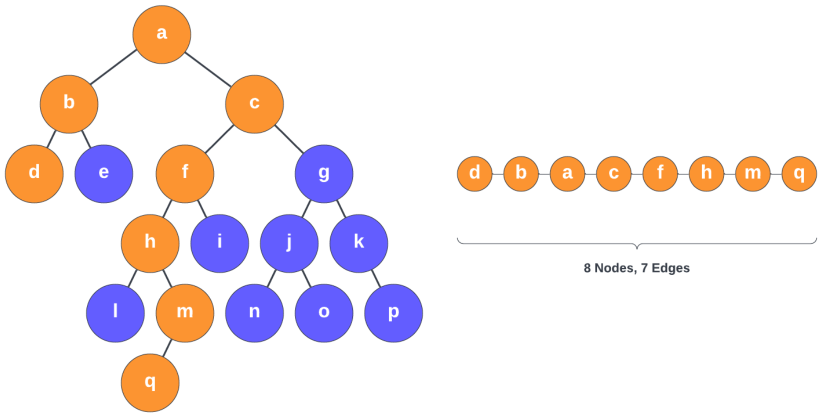
  Answer:

    - To accomplish this, we can make use of the DFS algorithm which searches a tree starting with the leftmost branch and traversing far down the child nodes before backtracking and moving rightward. (Full explanation given in earlier question)
      - First, we can start at a given vertex and conduct the DFS traversal of our undirected tree.
      - As each set of child nodes are visited, one will be chosen its edge (u, v) traversed, again towards another unvisited edge until a leaf node is reached. Note that every edge would need to be marked the first time it is traversed.
      - When the lead node is reached, the algorithm will backtrack by a step to the previous vertex or node that was visited and select the other child. As it back-tracks, the edge v, u is marked.
      - This process will continue over and over until each node is visited once, and as each edge was visited twice, we can add the edges $(v, u$ and $u, v)$ in our path– ensuring that they are now visited twice, once in each direction. This will given a time complexity of O(|V|+|E|)

- **3.3 (5pt):**
  - **Let T be the undirected binary tree from problem 3.1. For each pair of vertices, we can compute the distance between these vertices. In the binary tree above, for example, we have dist(d,i) = 5, and dist(l,o) = 6. We define the diameter of T to be the maximum value of dist(x,y), chosen over all pairs of vertices x and y in T. Clearly explain why the diameter of the above tree is 7.**

  Answer:

  We can define the diameter of a tree $T = (V, E)$ as $\max \delta(u, v)$ where u and v are elements of V. The diameter is essentially the largest of all shortest path distances in a given tree. If we traverse from our root node a, all the way down the furthest node q, we can see that we pass through 5 edges. We must note however that our root node a does have another branch extending through b and d, which provide an additional 2 edges. If we add 5+2, we arrive at 7 which is the diameter of the tree.



Homework#5

- **3.4 (5pt):**
    - **Let T be an undirected binary tree with n vertices. Create an algorithm to compute the diameter of T. Clearly explain how your algorithm works, why it guarantees the correct output, and determine the running time of your algorithm.**

Answer:

- We previously defined the BFS as a traversal algorithm that operates by visiting all the child nodes on a given level before moving deeper the following level. As the algorithm traverses a tree, the nodes or vertices are marked as visited to ensure that the same node is not visited twice. (Full explanation given in earlier question)
- We can define the diameter of a tree $T = (V, E)$ as $\max \delta(u, v)$
- We can use these concepts to build an algorithm that utilizes these concepts to compute the diameter of a tree:
    - First we establish a global variable for the maximum length of the tree and set the value to zero.
    - Next, we can traverse the vertices in the graph
    - We can now use BFS to determine for every vertex to get a path representing the minimum number of edges. These edges represent the difference between the start and current vertices.
    - We can then compare the current and global length to determine whether the global length should be updated.
    - Finally, we can go ahead and return that global length representing the diameter tree.

- The output will be correct given the methodology in which BFS is being used. Recall that we use BFS on a node in the graph, and we can note which node u was discovered last. We can run BFS again and note which node v was discovered last. Let us assume we have two nodes a and b and there is a path between these two nodes, with a third node t representing the first node along that give path. If we observe the paths $path1$ from s to u and $path2$ from a to b are unique and do not share any edges along their paths, then:

$$d(t,u) \geq d(s,u)$$
$$d(t,u) \geq d(s,a)$$
$$d(t,u) \geq d(t,a)$$
$$d(b,u) \geq d(b,a)$$

Given that u, as previously noted was the last node identified using the BFS algorithm we can see that the distances of t, a and t, u are equal, and therefore $d(u, v)$ is in fact the correct diameter. The running time for this algorithm is O(|V|+|E|)

References:

[1] Introduction to Algorithms, Cormen, Third Edition. (CLRS)

[2] Min-Max Heaps, Wikipedia (https://en.wikipedia.org/wiki/Min-max_heap)

[3] LucidChart Drawing Tool (https://lucid.app/lucidchart)