Final Project Design Documentation

1. Problem Analysis
   1. Design a Depth First Search (DFS) algorithm, then validate that it performs within the bounds of one of the three parenthesis related to a DFS.
   2. This will require designing a Graph class to be searched by the DFS.
   3. The Graph class will require the ability to be populated with tuple data for vertices of each edge in the graph
   4. To properly store the tuple at a specified vertices, a Multiset architecture will need to be implemented
2. Design Decisions
   1. My primary design decision to separate the Graph object into its root elements, those being its vertices and edges. I handled this by creating a Multiset object comprised of Node, and then initializing the Graph by assigning a Multiset object to each vertices for size + 1.
   2. I chose to initialize all array/multiset objects with size+1 to avoid index out of bounds exceptions, and also to mimic the graph structure by effectively not using the index at 0 ever, and initializing at 1.
   3. For the design
3. Assumptions
   1. My only assumption was that there would be no 0 vertices present in the Graph object.
4. Classes
   1. Node<Edge>
      1. private Edge edge
         1. instance variable used to store node data
      2. private Node<Edge> next
         1. link to next Edge from the current node
      3. public Edge getEdge()
         1. returns Edge stored in Node
      4. public void setEdge(Edge i)
         1. assign a value to edge
      5. public Node<Edge> getNext()
         1. return the next Edge from the current edge
      6. public void setNext(Node<Edge> next)
         1. assign a value to next
   2. Multiset
      1. private int elements
         1. count for the number of elements in the Multiset
      2. private Node<Edge> first
         1. beginning element in the multiset
      3. public Multiset()
         1. constructor, initialize first as null, and elements as 0
      4. public boolean isEmpty()
         1. return whether there are any elements in the Multiset
      5. public int size()
         1. return the size of the Multiset
      6. public void add(Edge i)
         1. add a new edge to the Multiset
      7. public Iterator<Edge> iterator()
         1. instantiate an iterator for the elements of the Multiset
   3. private class ListIterator<Edge> implements Iterator<Edge>
      1. locally instantiated in Multiset
      2. provides customized ListIterator methods to traverse Multiset
   4. Graph
      1. private final int vertices
         1. instance variable for the count of vertices in the Graph
      2. private int edges
         1. instance variable for the count of edges in the Graph
      3. private Multiset<Integer>[] graph
         1. Multiset object used to store other Multiset objects as vertices
      4. public Graph(int i)
         1. constructor, accepts an integer as the number of vertices the graph will contain
         2. create a graph object based on the number of vertices
      5. public int getVertices()
         1. return the number of vertices contained in the Graph
      6. public Iterable<Integer> adj(int v)
         1. provides an iterable object of adjacent edges at a specified vertices
      7. private void validateVertex(int v)
         1. ensures that a vertex to be set or get is within the bounds of the Graph
      8. public void addEdge(int v, int w)
         1. insert a new Node<Edge> at a specified x,y coordinate of the Graph
      9. public int degree(int v)
         1. return the number of degrees for a specified vertices
      10. public int getEdges()
          1. return the number of edges in the graph
   5. DepthFirstSearch
      1. private boolean white[]
         1. an array of booleans to define a vertices which has not yet been traversed
      2. private boolean gray[]
         1. an array of booleans to define a vertices being traversed
      3. private boolean black[]
         1. an array of booleans to define a vertices which has been traversed completely
      4. private int time
         1. an integer used as a counter for the time from discovery to completion for each vertices in the DFS
      5. private int d[]
         1. an array used to store the discovery time for a specific vertices
      6. private int f[]
         1. an array used to store the completion time for a specific vertices
      7. private Graph myGraph
         1. private Graph used to store an instance variable to be processed the dfs method
      8. public DepthFirstSearch(Graph G)
         1. instantiate a DFS object with the specified graph
         2. stores G at myGraph
      9. public void getDF()
         1. returns the value of the vertices, discovery time, and completion time once a DFS has been executed on the provided Graph
      10. public void dfs()
          1. initializes white, gray, black, d, and f.
          2. sets time to 0
          3. executes a private recursive dfsVisit on each vertices
      11. private void dfsVisit(int i)
          1. execute a DFS of all edges on vertices i recrsively.
          2. time is incremented for each visit
          3. if a new dicsovery occurs, add the time for the current vertices to d[]
          4. each vertices which has been visited is set to gray
          5. once finished, the vertices is set to black
          6. finally, time is incremented for the completion, and the time is added to f[] for the current vertices
   6. TestDFS
      1. public void testError()
         1. execute a junit test to validate that index out of bounds exceptions are thrown when necessary on Graph objects
      2. public void functionalTest()
         1. create a Graph with the provided data set, then assert that the depth of each vertices is as expected
      3. public static void main(String[] args)
         1. Test driver for DepthFirstSearch
         2. instantiate a new Graph with the provided vertices and edges
         3. instantiate a new DFS object using the first Graph
         4. execute dfs() method on first DFS object
         5. execute getDF() method on first DFS object
         6. instantiate a new Graph with another set of sample data
         7. nstantiate a new DFS object using the second Graph
         8. execute dfs() method on second DFS object
         9. execute getDF() method on second DFS object
5. Testing
   1. test that a Graph throws an error when an unknown index is inserted upon
   2. test that a Graph functions as expected under ideal circumstances
   3. test that a DFS properly traverses a Graph and outputs the expected data via manual inspection
6. Error Handling
   1. handle IndexOutOfBoundsException errors for Graph object
   2. handle when Graphs are initialized with an index of less than one by throwing an IllegalArgumentException
   3. initialize all indexes at 1 to avoid errors when dealing with vertices
   4. disable the remove method of ListIterator to avoid potential issues of removed edges
   5. throw NoSuchElementException when necessary with ListIterator
   6. handle a state where getDF() is called before the dfs() method is executed on a DFS object
7. Data Analysis
   1. Data

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Predefined Data Set | | | | | |  | Random Data Set | | | | | |
| x | y | Parent | Vertices | Discovery | Finish |  | x | y | Parent | Vertices | Discovery | Finish |
| 1 | 2 | NA | 1 | 1 | 14 |  | 1 | 4 | NA | 1 | 1 | 14 |
| 1 | 6 | 4 | 2 | 8 | 11 |  | 2 | 5 | 5 | 2 | 9 | 12 |
| 2 | 3 | 2 | 3 | 9 | 10 |  | 3 | 4 | 4 | 3 | 4 | 7 |
| 2 | 4 | 5 | 4 | 3 | 12 |  | 4 | 5 | 5 | 4 | 3 | 8 |
| 2 | 5 | 1 | 5 | 2 | 13 |  | 5 | 1 | 1 | 5 | 2 | 13 |
| 3 | 5 | 4 | 6 | 4 | 7 |  | 6 | 3 | 3 | 6 | 5 | 6 |
| 4 | 5 | 6 | 7 | 5 | 6 |  | 7 | 2 | 2 | 7 | 10 | 11 |
| 5 | 1 |  |  |  |  |  |  |  |  |  |  |  |
| 6 | 4 |  |  |  |  |  |  |  |  |  |  |  |
| 6 | 7 |  |  |  |  |  |  |  |  |  |  |  |

* 1. All data found above can be located in its raw format in output.txt
  2. Key
     1. P1 = The intervals [d[u], f[u]] and [d[v], f[v]] are completely disjoint;
     2. P2 = The [d[u], f[u]]  interval is completely contained within interval [d[v], f[v]], and u is a descendant of v in the dfs tree;
     3. P3 = The [d[v], f[v]] interval is completely contained within interval [d[u], f[u]], and v is a descendant of u in the dfs tree;
  3. Parenthesis Validation
     1. U represents the first vertices
     2. V represents the second vertices
     3. P represents the parenthesis value from the key above

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| U | V | Predefined | P | U | V | Random | P |
| 1 | 2 | (1,14)(8,11) | P3 | 1 | 2 | (1,14)(9,12) | P3 |
| 1 | 3 | (1,14)(9,10) | P3 | 1 | 3 | (1,14)(4,7) | P3 |
| 1 | 4 | (1,14)(3,12) | P3 | 1 | 4 | (1,14)(3,8) | P3 |
| 1 | 5 | (1,14)(2,13) | P3 | 1 | 5 | (1,14)(2,13) | P3 |
| 1 | 6 | (1,14)(4,7) | P3 | 1 | 6 | (1,14)(5,6) | P3 |
| 1 | 7 | (1,14)(5,6) | P3 | 1 | 7 | (1,14)(10,11) | P3 |
| 2 | 1 | (8,11)(1,14) | P2 | 2 | 1 | (9,12)(1,14) | P2 |
| 2 | 3 | (8,11)(9,10) | P3 | 2 | 3 | (9,12)(4,7) | P1 |
| 2 | 4 | (8,11)(3,12) | P2 | 2 | 4 | (9,12)(3,8) | P1 |
| 2 | 5 | (8,11)(2,13) | P2 | 2 | 5 | (9,12)(2,13) | P2 |
| 2 | 6 | (8,11)(4,7) | P1 | 2 | 6 | (9,12)(5,6) | P1 |
| 2 | 7 | (8,11)(5,6) | P1 | 2 | 7 | (9,12)(10,11) | P3 |
| 3 | 1 | (9,10)(1,14) | P2 | 3 | 1 | (4,7)(1,14) | P2 |
| 3 | 2 | (9,10)(8,11) | P2 | 3 | 2 | (4,7)(9,12) | P1 |
| 3 | 4 | (9,10)(3,12) | P2 | 3 | 4 | (4,7)(3,8) | P2 |
| 3 | 5 | (9,10)(2,13) | P2 | 3 | 5 | (4,7)(2,13) | P2 |
| 3 | 6 | (9,10)(4,7) | P1 | 3 | 6 | (4,7)(5,6) | P3 |
| 3 | 7 | (9,10)(5,6) | P1 | 3 | 7 | (4,7)(10,11) | P1 |
| 4 | 1 | (3,12)(1,14) | P2 | 4 | 1 | (3,8)(1,14) | P2 |
| 4 | 2 | (3,12)(8,11) | P3 | 4 | 2 | (3,8)(9,12) | P1 |
| 4 | 3 | (3,12)(9,10) | P3 | 4 | 3 | (3,8)(4,7) | P3 |
| 4 | 5 | (3,12)(2,13) | P2 | 4 | 5 | (3,8)(2,13) | P2 |
| 4 | 6 | (3,12)(4,7) | P1 | 4 | 6 | (3,8)(5,6) | P3 |
| 4 | 7 | (3,12)(5,6) | P1 | 4 | 7 | (3,8)(10,11) | P1 |
| 5 | 1 | (2,13)(1,14) | P2 | 5 | 1 | (2,13)(1,14) | P2 |
| 5 | 2 | (2,13)(8,11) | P3 | 5 | 2 | (2,13)(9,12) | P3 |
| 5 | 3 | (2,13)(9,10) | P3 | 5 | 3 | (2,13)(4,7) | P3 |
| 5 | 4 | (2,13)(3,12) | P3 | 5 | 4 | (2,13)(3,8) | P3 |
| 5 | 6 | (2,13)(4,7) | P1 | 5 | 6 | (2,13)(5,6) | P3 |
| 5 | 7 | (2,13)(5,6) | P1 | 5 | 7 | (2,13)(10,11) | P3 |
| 6 | 1 | (4,7)(1,14) | P2 | 6 | 1 | (5,6)(1,14) | P2 |
| 6 | 2 | (4,7)(8,11) | P1 | 6 | 2 | (5,6)(9,12) | P1 |
| 6 | 3 | (4,7)(9,10) | P1 | 6 | 3 | (5,6)(4,7) | P2 |
| 6 | 4 | (4,7)(3,12) | P1 | 6 | 4 | (5,6)(3,8) | P2 |
| 6 | 5 | (4,7)(2,13) | P1 | 6 | 5 | (5,6)(2,13) | P2 |
| 6 | 7 | (4,7)(5,6) | P3 | 6 | 7 | (5,6)(10,11) | P1 |
| 7 | 1 | (5,6)(1,14) | P2 | 7 | 1 | (10,11)(1,14) | P2 |
| 7 | 2 | (5,6)(8,11) | P1 | 7 | 2 | (10,11)(9,12) | P2 |
| 7 | 3 | (5,6)(9,10) | P1 | 7 | 3 | (10,11)(4,7) | P1 |
| 7 | 4 | (5,6)(3,12) | P2 | 7 | 4 | (10,11)(3,8) | P1 |
| 7 | 5 | (5,6)(2,13) | P2 | 7 | 5 | (10,11)(2,13) | P2 |
| 7 | 6 | (5,6)(4,7) | P3 | 7 | 6 | (10,11)(5,6) | P1 |

1. Lessons Learned
   1. I learned a good lesson with this project about how to effectively monitor the state of a recursive function using arrays of boolean values. I had not thought of such an effective method to do so before this exercise.
   2. I also learned architecting a Graph data structure can be somewhat complex
   3. lastly, I learned how to successfully implement a depth first search algorithm