Compiler:

Standard JDK

Compiling Instructions:

navigate to the "**Project1_ADS/src"** folder and use following command to compile the code:

javac project1/code/MST.java

Run Instructions:
navigate to "**Project1_ADS/src"** folder and use following command to run the program:

For Random: java project1.code.MST -r <noOfVertics> <percentageDensity>
Example: java project1.code.MST -r 1000 10

For SimpleScheme: java project1.code.MST -s <filePath>
For F-HeapScheme: java project1.code.MST -f <filePath>

Code location:
~/Project1_ADS/src/project1/code/*.java

Dummy Data files:
large data files as inputs are present in following folder:
~/Project1_ADS/data/<SizeName>.txt

<SizeName>: its the name of the dummy data file
input1000d10.txt: mean that the text file contain 1000 vertice data with 10% density

Files:

MST.java
RandomMatrixGenerator.java
FheapTreeMethods.java
SimpleSchema.java

Program Description :
The program implements Prim's MST algorithm on a given connected graph either as random automated input or from a input edge data file by the user.

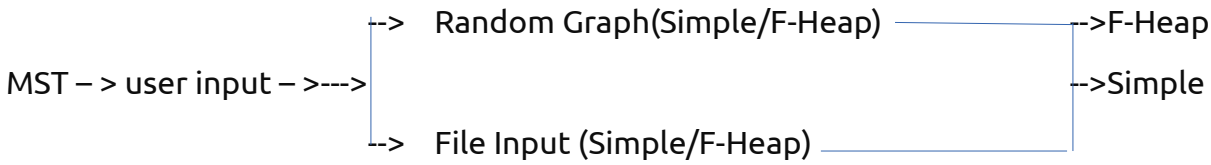
Main file is MST.java

List of CLASS Files

- MST
- RandomMatrixGenerator
- FheapTreeMethods
- SimpleSchema

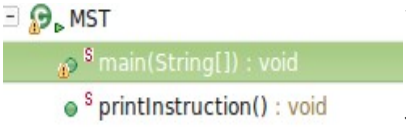
Function Prototypes:

Basic Flow:



Description of class: MST

- **void main()**



1. For Random (will randomly select F-Heap or Simple-Scheme):

   java MST -r <n: noOfVertices> <d: %density>

2.For User Input Mode

   a. Simple-Scheme : java MST -s <file: fullFilePath>

   b. F-Heap Schema : java MST -f <file: fullFilePath>

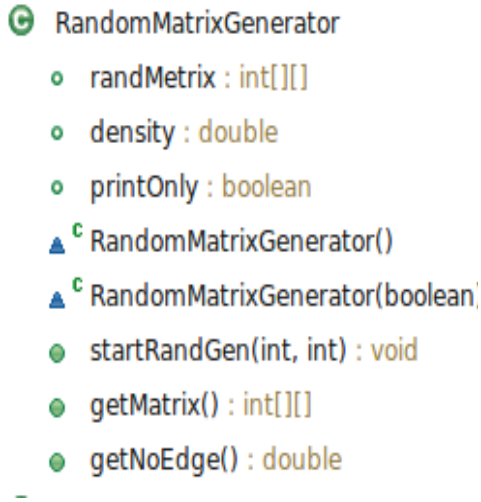
- **printInstruction ():** contain details regarding the usage of the MST class

Description of class RandomMatrixGenerator

1. RandomMatrixGenerator class

This class is used to generate the random graph with the given density. The edge weight is also calculated randomly using the formula:

edgeWeight=random.nextInt(1000)+1



- **startRandGen(int x, int y):** startRandGen method is used to create the random matrix (graph) depending upon the number of vertices (x) and the density of edges in the graph (d). It uses DFS to check the graph is connected or not and depending on status of the random graph if required graph is generated again till the random graph is connected.

```
density=(int) Math.ceil((c/200)*vert*(vert-1));;
```

density or no. of edges per the given density (c) percentage

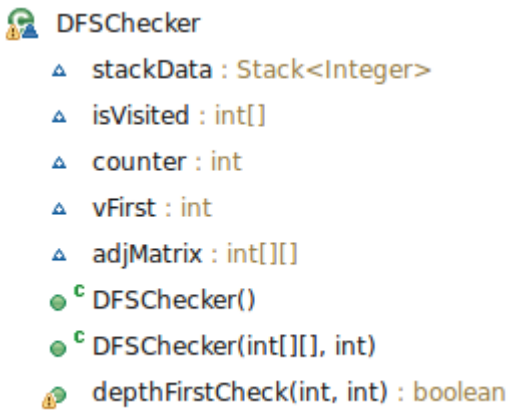
```
int r = generator.nextInt(vert);        int s = generator.nextInt(vert);        int t = generator.nextInt(1000)+1;
```

random nodes and random edge weight are generated.

- **getMatrix():** this method is used to get the random generated matrix which is connected and strored in RandomMatrixGenerator
- **getNoEdge():** this method is used to get the no. of edges added in the matrix (graph) as per the input density

2. DFSChecker class

This class is used to check the random generated graphy with density constrain is connected or not using Depth First Search (DFS) methodology



stackData: used to track the visited node

isVisited: is used to keep track of the nodes need to be visited to make a connected graph

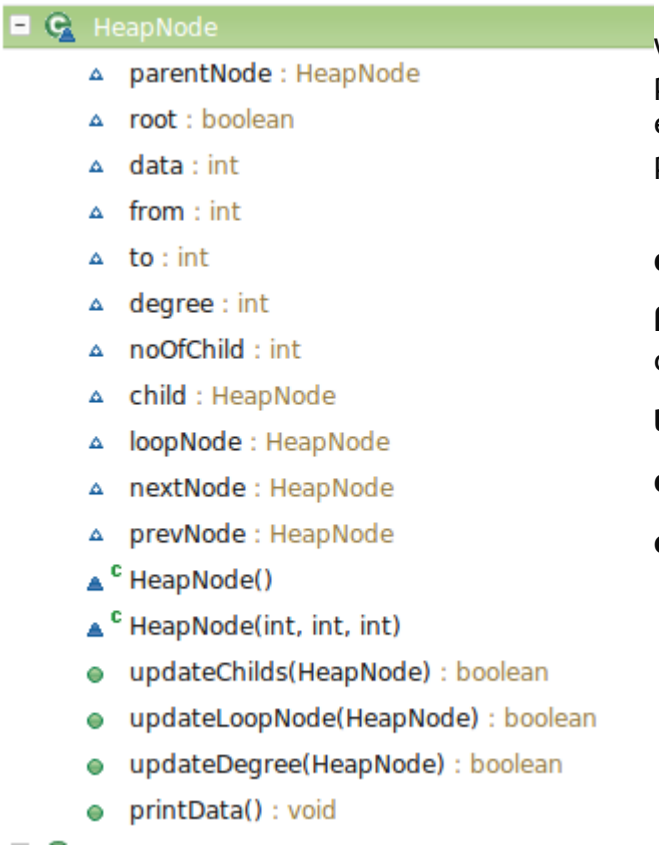
adjMatrix: is the matrix used to comparison to check the weights in the matrix and generate matrix in formate readable by the dfs methods.

- **DFSChecker(int[][] matrix, int vert):** this is the default constructor to intialize the adjcent matrix for the DFS check.
  Parameters:
  - matrix: the input matrix for DFS check
  - vert: is the no. of vertices for the random graph and used to initiate the stack for checking the node connection and visits

- **depthFirstCheck(int start, int):** this method is uesd to check the input matrix is a connected graph or not using Depth Fitst Search methodology. Checking the a Stack and a array to track the no. of vertices visited while tracking the input matrix of the random graph.

Description of Member Functions or Methods in class FheapTreeMethods

1. HeapNode class



This class is the basic structre of the Nodes or we can say the vertices. It represent all the properties of the vertices. This nodes acts as elements for F-Heap. The basic structure properties such as :

data: edge weight

from: from vertices (to maintain connectivity details)

to: to vertices (to maintain connectivity details)

degree: represents the no. of child levels

child: pointer for the child node

loopNode: it represent as pointre for next node in the loop

prevNode and nextNode: allow pointer representation to move forward and backward in the F-Heap

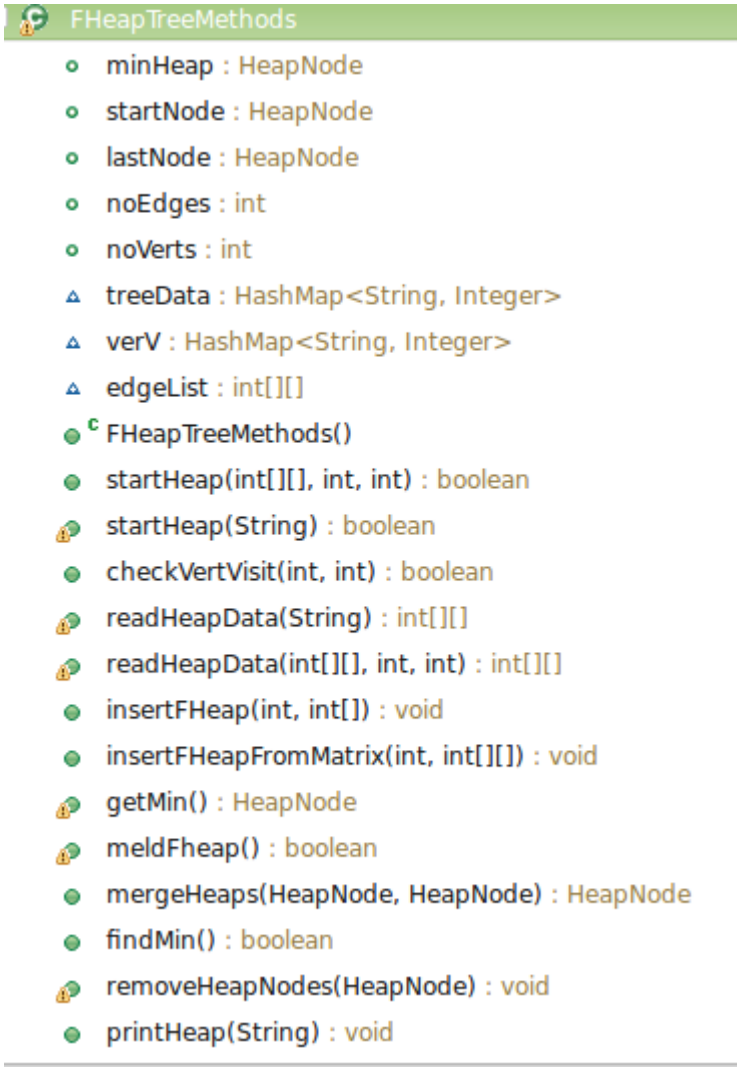
- **HeapNode(int data, int from, int to):** its the default constructor for initialization of the variable of HeapNode
- **updateChilds(HeapNode x):** this method allow to add childs of same degree to the parent HeapNode.
  X: is the childNode we want to add in the HeapNode.

- **updateLooNode(HeapNode x):** this method allow to update the loopNode for F-Heap in particular level of the F-Heap tree (as a child or parent/roots).
- **updateDegree(HeapNode current):** this method is used to update the degree of HeapNode. As degree of parent node need to be updated every time a new child node is added to the parent.

2. FheapTreeMethods class

This the main class generating the F-Heap. Edge weight is used as comparison element for crearion of the f-heap. Various method to simulate the behavour of F-Heap such as MELD, REMOVEMIN, PAIR-WISE COMBINE,DELETE NODE etc.

- **startHeap**: this method are of two version one which works with the data as FILE and one which works with the random connected graph matrix.
  - **startHeap(int[][] matrix, int noV,int noE):** this one is for random connected graph martrix. It will take input and convert the matrix into class methods readable formate using **readHeapData** method, then selecting a random node and inserting the corrosponding edge values into the f-heap. In the end calling the **getMin** method. This flow is repeated till MST is generated.
  - **startHeap(String filePath):** this is the same method as the previous one but this one work with filePath for user data file.



- **checkVertVisit(int, int):** this method is used to check the pair of nodes are already visited or not. This is to avoid any loop trevarsal during MST creation

- **readHeapData:** similar to startHeap method this is also of two type one to handle matrix data and the other for reading structured data from file.

  - readHeapData(String)
  - readHeapData(int[][],int,int)

- **insertFHeap(int ,int []):** as the name suggest this method is used to insert corresponding edges of the visiting node into the F-Heap. This one is basically for the read from file version of insert.

- **insertFHeapFromMatrix(int, in[][]):** its similar to insertFHeap with only different in extractopm of edge weight from 2D array the data conversion methodology.
- **meldFheap():** this method is used to meld the F-Heap data after every getMin call. Once the minimum HeapNode is removed from F-Heap, all the roots nodes are meld together. The nodes with same degree are combined based on pair wise combine methodology.
  Find nodes with same degree → remove both nodes from root list → Meld → increase degree of the root node after meld → add back the node in F-Heap

- **findMin():** this method returns the minimum node from the F-Heap and call **meldFheap** to add back the child nodes in the F-Heap if exists and ping the minimum Nodes again.

- **removeHeapNodes(HeapNode m):** this method remove the input nodes from the F-Heap and recreate the root list. This method is called by the **meldFheap** to remove nodes with same degree for pair-wise comparision.

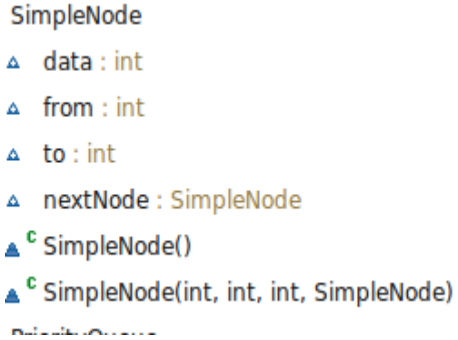
- **printHeap():** this method prints the details of the all the root nodes in the F-Heap. It provide the details of the F-Heap, with details like:
  - Minimum node
  - Start Node
  - Last Node
  - List of root node

- **mergeHeap(HeapNode x, HeapNode y):** this method perform the pair wise combine of the HeapNode passed in the argument. Comparision is based on the weight assicoiated with the Nodes. Degree of node with smaller weight is choosen as root node and degree associated with the node is increased and other node is added as child to the root node.

Description of Member Functions or Methods in class SimpleScheme:

This class represents the simple scheme implementation for generating the MST based on Edge weight. Edge weight is used as comparison element for creartion in simple scheme.

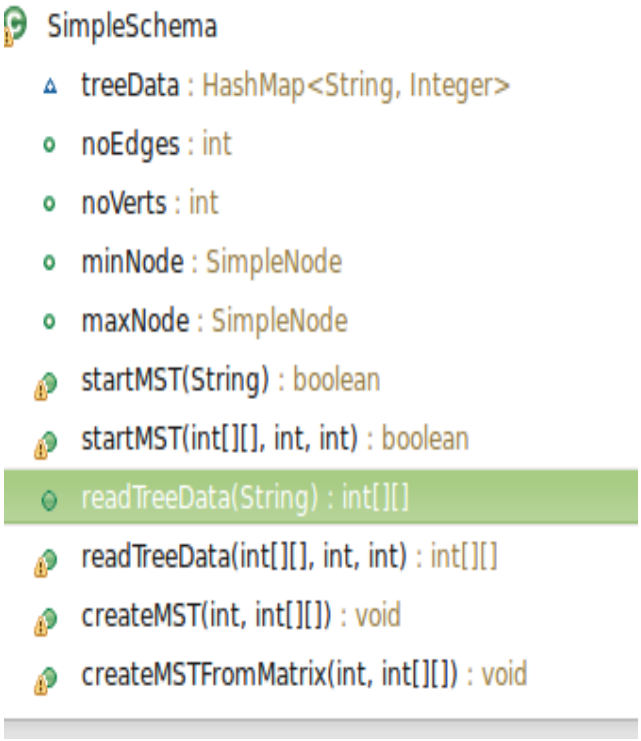
1. SimpleNode class: this class is used to create simple structured object for the simple scheme to store various attributes of the vertices and edges like edge weight, from node counter, to node counter, next node



- **SimpleNode(int d,int f,int t,SimpleNode x):** the constructor of SimpleNode class for initialization of the variables.

2. <u>SimpleScheme class:</u> this class provide the basic functionality of implementing MST on conneted graph using simple scheme (adjacent matrix based implementation). Data related to edges are stored in matrix and depending upon the random node from the vertice list the MST is created.

- **StartMST:** this method are of two version for supporting randomo architecture as well as file based graph structre input for MST generation. It act as controller for the SimpleScheme class
  - startMST(String filePath)
  - startMST(int[][] matrix, int noVert, int noEdge)



- **readTreeData:** this method is also of two version for supporting random matrix as well as file input. The main function is to transform the inputs to SimpleScheme readable format for processing
  - readTreeData(String Filepath)
  - readTreeData(int[][], int, int)

- **createMST(int rand, int[][] m):** this is the main method creates node based on the random node selection from the input matrix generated from the file data
- **createMSTFromMatrix(int ran, int[][] m):** this the random matrix version of the create with same functionality.

<u>Program Flow :</u>

Running Mode – r n d

- ✓ Call **startRandomGen**() for < noVert,density > as parameters
- ✓ The undirected connected graph thus generated is passed to <Fibonacci scheme>.<method_name>
- ✓ The undirected connected graph thus generated is passed to <Simple scheme>.<method_name>
- ✓ Average performance time of 5 runs of a distinct parameter pair <noVert,density> measured for both
  f-heap scheme and *Simple* Scheme

Running Mode – f file-name

- ✓ The path of user input **file-name i**s read from command line.
- ✓ Function <Fibonacci Scheme>.<method_name> is invoked for the user input graph
- ✓ The minimum cost of the Prims's MST is displayed
- ✓ The final MST is displayed in the required output format

Running Mode – s file-name

- ✓ The path of user input **file-name i**s read from command line.
- ✓ Function <Simple Scheme> .<method_name> is invoked for the user input graph
- ✓ The minimum cost of the Prim's MST is displayed
- ✓ The final MST is displayed in the required output format

Structural Description of Source Code :

INPUT : Command Line Argument for choice of MODE where

-r // Implementation of PRIM's MST in Random Graph Generator mode

-f // Implementation of PRIM's MST in User Input mode through ***F-HEAP*** Scheme

-s // Implementation of PRIM's MST in User Input mode through ***SIMPLE*** Scheme

STRUCTURAL IMPLEMENTATION :

When Mode Choice is -r

- The user is prompted to enter the **#Nodes** and **Edge Density**
- The user input values are accepted from console window
- Function <Random graph generator> is invoked for the parameter pair <**#Nodes, Edge Density**
- The undirected connected graph thus generated is passed to <Fibonacci scheme>
- The undirected connected graph thus generated is passed to <Simple scheme>
- The average execution time of 5 runs of a unique parameter pair <**#Nodes, Edge Density** >
  for both ***F-HEAP*** Scheme & ***SIMPLE*** Scheme is calculated

Gaurab Dey
gaurabde@ufl.edu

UFID : 78029140

When Mode Choice is -f

- The user is prompted to enter the filename containing a undirected graph in specified input format
- The user input file name is accepted from console window
- Function <Fibonacci Scheme> is invoked for the user input graph
- The cost of the MST is displayed
- The MST is displayed in the specified output format

When Mode Choice is -s

- The user is prompted to enter the filename containing a undirected graph in specified input format
- The user input file name is accepted from console window
- Function <Simple Scheme> is invoked for the user input graph
- The cost of the MST is displayed
- The MST is displayed in the specified output format

Real Time Performance for RANDOM MODE :

# NODES	EDGE DENSITY	T (Simple Scheme)	T (F-Heap Scheme)
100	10	50	89
300	20	2702	334
400	30	21360	787
500	10	7056	581
1000	10	207638	1720
2000	30	680324	2509
3000	50	1806315	7969

Data for large no. of Nodes (F-Heap scheme/Simple Scheme)

DENSITY(%)	TOTAL TIME FOR EXECUTION(IN MILLISEC)					
	No of vertices(1000)		No of vertices(3000)		No of vertices(5000)	
	Simple Scheme	F-Heap Scheme	Simple Scheme	F-Heap Scheme	Simple Scheme	F-Heap Scheme
10	33087	324	38089	3424	220541	85481
20	45004	466	44875	4167	345746	12963
30	60543	456	78032	4678	562841	16886
40	75456	549	88475	5322	854564	18968
50	102546	585	89574	5557	635415	22803
60	115486	569	100547	7635	954287	29841

70	129562	606	324245	7941	1254251	38780
80	346466	611	382356	9814	1854544	41827
90	372113	624	367223	10032	2344768	45409
100	602345	701	803256	11950	2756491	50212

Time Complexity Expectation :

- ***SIMPLE*** Scheme
  Expected time complexity for the *Simple* Scheme to execute is **$O(n^2)$** time where
  ***n* ---** number of nodes in the connected undirected graph

- ***F-HEAP*** Scheme
  Expected time complexity for the *F-Heap* Scheme to execute is **O(e + n log n)** where
  ***e* ---** number of edges in the connected undirected graph
  ***n* ---** number of nodes in the connected undirected graph
  In a Fibonacci Heap Scheme implementation, the initialization of the heap requires ***n*** insertions. Each ***Insertion*** operation takes **O(1)** time, hence taking total **O(n)** time for heap initialisation.
  As the final minimum cost spanning tree shall have **(n-1)** edges, the ***RemoveMin*** operation is invoked **(n-1)** times or **O(n)**.Now, each ***RemoveMin*** operation takes a total of **O(log n)** time.
  Thus, the total time taken all ***RemoveMin*** operations is **O(n log n).**
  Each time a ***RemoveMin*** is invoked, all the adjacent edges from the removed node are scanned.
  As each node is considered only once, the total no. of edges scanned after all ***RemoveMin*** operations is twice the no. of edges in the graph since every edge is scanned twice for each of its end nodes. This scanning should take **O(e)** in total.
  Thus, asymptotically, the entire program is expected to run in **O(e + n log n) time.**

Performance Analysis & Conclusion:

Relative Performance Data *Simple Scheme* vs *F-Heap Scheme*

No. of Nodes	Performance Simple	Performance F-Heap
100	200	112.35
300	11.1	89.82
400	1.9	50.82
500	7.1	86.25
1000	0.48	58.14
2000	0.42	11.26

Relative performance graph:

Gaurab Dey
gaurabde@ufl.edu

UFID : 78029140

Expectations -

1. If we only keep inserting the elements into both data structures, we should get a better performance in case of Fibonacci heap than simple scheme as simple scheme has amortized cost (n*n) for insert operation where as in Fibonacci heap has amortized cost of O(1) for the insert operation.

2. If we only keep deleting the minimum elements from the both data structures, we should get a better performance in case of simple scheme as though both have amortized complexity of O(1) as only one need to be deleted but in case of fibonacci along with delete we need to merge data back in F-Heap causing complexity to be O(log n), deletemin of min Fibonacci heap takes O(n) in the worst case whereas deletemin of simpleschme takes O(n*n) in the worst case.