CS-250

Data Structures and Algorithms

Lab 15

PRIM’S AlGORITHM

Name: Mahum Samar

Class: BSCS-9B

CMS ID: 290647

# Code:

|  |
| --- |
| #include <iostream> #include <cmath>  using namespace std; #define arraySize 10 class PrimsArray; class GraphAdjacencyMatrix;     *//template<typename t>* class PrimsArray { public:  int u;  int key;  int predecessor; };  template<typename t> class Heap { public:  int sizeOfHeap = 0;   int LeftChild(int index) {  *//method to return left child* return (2 \* index);  }   int RightChild(int index) {  *//method to return right child* return ((2 \* index) + 1);  }   void Swap(t \*x, t \*y) {  *//method to swap the two values of the array by pointing to the address of the array index* t temp = \*x;  \*x = \*y;  \*y = temp;  }   void MinHeapify(t array[], int index, int sizeOfArray) { *//method to place one value at its logical position* int left = LeftChild(index);  int right = RightChild(index);  int largestIndex;  *//conditions for checking index having largest value* if (left <= sizeOfArray && array[left].key < array[index].key) {  largestIndex = left;  } else {  largestIndex = index;  }  if (right <= sizeOfArray && array[right].key < array[largestIndex].key) {  largestIndex = right;  }  if (largestIndex != index) {  Swap(&array[index], &array[largestIndex]);  *// cout << "swapping index"; //recursive call* MinHeapify(array, largestIndex, sizeOfArray);  }   }   void InsertMinHeapify(t array[], int index, int sizeOfArray) {  *//method to max heapify heap during insertion* if (index >= 1) {  int parentIndex = floor(index / 2); *//finding parent* if (parentIndex > 1 && array[parentIndex].key > array[index].key) {  *//if parent has less value than swap* Swap(&array[parentIndex], &array[index]);  *//recursive call* InsertMinHeapify(array, parentIndex, sizeOfArray);  }  }  }   void BuildMinHeap(t array[], int sizeOfArray) {   *//sizeOfArray/2 because comparison start from the left most subtree  //to avoid the condition to be checked for the leaf nodes  //the elements in range from array[n/2 + 1,...,n] are leaf nodes* for (int index = sizeOfArray / 2; index > 0; index--) {  *//max heapify the current element* MinHeapify(array, index, sizeOfArray);  }  }   void InsertValueInHeap(t array[], int sizeOfArray, t value) {  *//method to insert value in heap* sizeOfHeap++;  if (sizeOfHeap < sizeOfArray) {  array[sizeOfHeap] = value; *// heapify value* InsertMinHeapify(array, sizeOfHeap, sizeOfArray);   } else {  cout << "Array max length reached. More values can't be added.\n";  return;  }   }   t FindMin(t array[]) {  *//method to return maximum element in heap* if (sizeOfHeap != 0) {  return array[1];  }  }   bool IsEmpty(t array[]) {  *//method to check either heap is empty or not* return sizeOfHeap == 0;  }   int Size(t array[]) {  *//method to return size of heap* return sizeOfHeap;  }   void BuildHeap(t array[], int sizeOfArray) {  *//method to build heap from the given array* BuildMinHeap(array, sizeOfArray);  }   void HeapSort(t array[]) {  *//method to sort heap* if (sizeOfHeap == 0) {  return;  } else {  *//recursive call case* Swap(&array[1], &array[sizeOfHeap]);  sizeOfHeap--;  *//max heapify 1st element* MinHeapify(array, 1, sizeOfHeap);  HeapSort(array);  }   }   void DeleteMin(t array[], int sizeOfArray) {  *//method to delete maximum value from heap* array[1] = array[sizeOfHeap];  sizeOfHeap--;  *//heapify 1t element* MinHeapify(array, 1, sizeOfArray);  }   void PrintHeap(t array[]) {  *//method to print heap* for (int i = 1; i <= sizeOfHeap; ++i) {  cout << "For vertex: " << i<<"\t";  cout << array[i].u << "\t";  cout << array[i].key << "\t";  cout << array[i].predecessor << "\n";  }  cout << endl;  }   };   class GraphAdjacencyMatrix {  *// class for implementing adjacency matrix form of graph  //in this class 1 dimensional array is used.  //and it is accessed as a 2 dimensional array for storing the graph edges* public:  int vertex; *//total number of vertices // int matrixSize; //maximum size of array* void InitializeGraph(int AdjacencyMatrix[][arraySize]) {  *//method to initialize 1 dimensional array* for (int i = 0; i < vertex; i++) {  for (int j = 0; j < vertex; ++j) {  AdjacencyMatrix[i][j] = 0;  }  }  }   void InsertEdgeUndirected(int AdjacencyMatrix[][arraySize], int u, int v, int weight) {  *//method used to insert the undirected edges in graph* AdjacencyMatrix[u][v] = weight;  AdjacencyMatrix[v][u] = weight;   }    void PrintGraph(int AdjacencyMatrix[][arraySize]) {  *//method to print graph on screen* for (int i = 1; i < vertex; i++) {   for (int j = 1; j < vertex; ++j) {  if (j % vertex - 1 == 0) { *// adds new line after the vertex length of array values are printed* cout << "\n";  }   cout << AdjacencyMatrix[i][j] << " ";  }  }  cout << "\n";  }   bool IsConnected(int AdjacencyMatrix[][arraySize], int u, int v) {  *//method ot check whether u has edge towards v* if (AdjacencyMatrix[u][v] == 0) {  return false;  } else {  return true;  }  }   void PrimsAlgorithm(int AdjacencyMatrix[][arraySize], int startIndex,Heap<PrimsArray> \*heap2,PrimsArray heapArray2[10]) {  cout<<"Print element of passed array: "<<AdjacencyMatrix[1][2];   PrimsArray heapArray[vertex];  *//initializing the heap array* for (int i = 1; i < vertex ; ++i) {  heapArray[i].u = i;  heapArray[i].key = 10000;  heapArray[i].predecessor = -1;  }  heapArray[startIndex].key = 0;  heapArray[startIndex].predecessor = 0;   Heap<PrimsArray> \*heap = new Heap<PrimsArray>();  heap->sizeOfHeap=vertex-1;  heap->PrintHeap(heapArray);  *//build min heap* heap->BuildHeap(heapArray, vertex);  heap->PrintHeap(heapArray);  *//build another heap array to store vertices which are visited* while (!(heap->IsEmpty(heapArray))) {  PrimsArray u = heap->FindMin(heapArray);  heap->DeleteMin(heapArray, vertex); *// heap->MinHeapify(heapArray, 1, vertex);* heap2->InsertValueInHeap(heapArray2,vertex,u);  for (int i = 1; i < vertex ; ++i) {  if (AdjacencyMatrix[u.u][i] != 0) {  if (i <= heap->sizeOfHeap && AdjacencyMatrix[u.u][i] < heapArray[i].key) { *// if (AdjacencyMatrix[u.u][i] < heapArray[i].key) {* heapArray[i].predecessor = u.u;  heapArray[i].key = AdjacencyMatrix[u.u][i];  heap->MinHeapify(heapArray, i, vertex );  }   }  }   }  }   void PrintPrimsArray(PrimsArray array[]){  for (int i = 1; i < vertex; ++i) {  cout<<"For vertex : "<<i<<" -> ";  cout << "u : " << array[i].u<< " ";  cout << "key : " << array[i].key<< " ";  cout <<"predecessor : " << array[i].predecessor<< "\n";  }  }  };     int main() { *// Adjacency Matrix* GraphAdjacencyMatrix \*graphAdjacencyMatrix = new GraphAdjacencyMatrix(); *// cout<< "Enter the number of vertices: \t"; // cin>>graphAdjacencyMatrix->vertex;* graphAdjacencyMatrix->vertex = arraySize; *// initializing graph // graphAdjacencyMatrix->matrixSize = graphAdjacencyMatrix->vertex \* graphAdjacencyMatrix->vertex;* int AdjacencyMatrix[arraySize][arraySize];  graphAdjacencyMatrix->InitializeGraph(AdjacencyMatrix);    graphAdjacencyMatrix->InsertEdgeUndirected(AdjacencyMatrix, 1, 2, 4);  graphAdjacencyMatrix->InsertEdgeUndirected(AdjacencyMatrix, 1, 4, 1);  graphAdjacencyMatrix->InsertEdgeUndirected(AdjacencyMatrix, 1, 5, 8);  graphAdjacencyMatrix->InsertEdgeUndirected(AdjacencyMatrix, 2, 3, 2);  graphAdjacencyMatrix->InsertEdgeUndirected(AdjacencyMatrix, 2, 4, 2);  graphAdjacencyMatrix->InsertEdgeUndirected(AdjacencyMatrix, 2, 5, 6);  graphAdjacencyMatrix->InsertEdgeUndirected(AdjacencyMatrix, 2, 6, 1);  graphAdjacencyMatrix->InsertEdgeUndirected(AdjacencyMatrix, 3, 5, 2);  graphAdjacencyMatrix->InsertEdgeUndirected(AdjacencyMatrix, 3, 6, 5);  graphAdjacencyMatrix->InsertEdgeUndirected(AdjacencyMatrix, 4, 5, 11);  graphAdjacencyMatrix->InsertEdgeUndirected(AdjacencyMatrix, 4, 7, 9);  graphAdjacencyMatrix->InsertEdgeUndirected(AdjacencyMatrix, 4, 8, 8);  graphAdjacencyMatrix->InsertEdgeUndirected(AdjacencyMatrix, 5, 7, 4);  graphAdjacencyMatrix->InsertEdgeUndirected(AdjacencyMatrix, 5, 8, 1);  graphAdjacencyMatrix->InsertEdgeUndirected(AdjacencyMatrix, 5, 9, 5);  graphAdjacencyMatrix->InsertEdgeUndirected(AdjacencyMatrix, 5, 6, 3);  graphAdjacencyMatrix->InsertEdgeUndirected(AdjacencyMatrix, 6, 8, 7);  graphAdjacencyMatrix->InsertEdgeUndirected(AdjacencyMatrix, 6, 9, 8);  graphAdjacencyMatrix->InsertEdgeUndirected(AdjacencyMatrix, 7, 8, 6);  graphAdjacencyMatrix->InsertEdgeUndirected(AdjacencyMatrix, 8, 9, 3);   graphAdjacencyMatrix->PrintGraph(AdjacencyMatrix);  Heap<PrimsArray> \*heap = new Heap<PrimsArray>();  PrimsArray heapArray2[10];   graphAdjacencyMatrix->PrimsAlgorithm(AdjacencyMatrix,1,heap,heapArray2);   graphAdjacencyMatrix->PrintPrimsArray(heapArray2);    return 0; } |

# Output:



