|  |
| --- |
| Conestoga College |
| Algorithms Assignment 6 |
| ADT Functions & Graphs |
|  |
| **Michael** |
| **RED LETTER DAY** |

|  |
| --- |
|  |

|  |  |
| --- | --- |
| 1. [10] Implement the Graph ADT functions   **void graph\_init(Graph \*graph); // initialize graph**  **int graph\_insert\_vertex(Graph \*graph, Item data);**  **int graph\_insert\_edge(Graph \*pg, Item v1, Item v2);**  **you will probably also want to implement**  int graph\_is\_adjacent(Graph \*pg, Item v1, Item v2)  static void graph\_destroy( Graph \*g )  using either an adjacency list or array. | All values are stored in Structure the nodes are then connected to each other |
| 1. [10] Place N (at least 100) nodes into the graph, and connect each of them (randomly) to between 2 and 6 other nodes (depending on how you do this, more or fewer may result – that’s okay) | No printout is necessary for this portion, I generate an array of random numbers and between 0-100 (2-6x) both sequences are randomized.  I then use this array to connect to different nodes. Please note that some nodes might have more than 6 connections and less than 2.  For the Less than two connections it’s is due to the fact that the random number generated is the same number as the node. – In my code I prevented a connection from a node being made to itself, it felt redundant. |
| 1. [5] Print out the adjacency information for each node | There are instances of some nodes having more than one connection, this is due to the random number generation not negating numbers that appear more than 6x or during the connection generation I don’t check the node for having 6+ connections. Both solutions to the problem seemed wasteful and time-consuming. |

CODE

|  |
| --- |
| Assignment\_6.h |
| #ifndef ASSIGNMENT\_6\_H\_INCLUDED\_  #define ASSIGNMENT\_6\_H\_INCLUDED\_  #include <stdio.h> /\* printf \*/  #include <time.h> /\* clockt, clock, CLOCKSPERSEC \*/  #include <math.h> /\* sqrt ---- compile with -lm flag \*/  #include <stdlib.h> /\* rand, srand , malloc\*/  #define CONNECT\_MAX 6  #define CONNECT\_MIN 2  #define NODE\_MAX 100  **void** **Assignment\_6**(**void**);  **int** **Rand\_Array**(**int** Random\_Connection\_Array[]);  **int** **Create\_Nodes**(**int** Number\_of\_Nodes);  **unsigned** **int** **RandomNumberGenerate**(**unsigned** **int** RandomValMin, **unsigned** **int** RandomValMax);  #endif |
| Assignment\_6.c |
| #include "Assignment\_6.h"  /\*  1. [10] Implement the Graph ADT functions  void graph\_init(Graph \*graph); // initialize graph  int graph\_insert\_vertex(Graph \*graph, Item data);  int graph\_insert\_edge(Graph \*pg, Item v1, Item v2);  you will probably also want to implement  int graph\_is\_adjacent(Graph \*pg, Item v1, Item v2)  static void graph\_destroy( Graph \*g )  using either an adjacency list or array.  2. [10] Place N (at least 100) nodes into the graph, and connect each of them (randomly) to between 2 and 6 other nodes (depending on how you do this, more or fewer may result – that’s okay)  3. [5] Print out the adjacency information for each node  \*/  **void** **Assignment\_6**(**void**){  **const** **int** Random\_Connection\_Array[**6**];  **int** count = **0**;  **int** node\_count = **0**;  **int** Connected\_Nodes;  **time\_t** t;  /// Create the Graph (this creates the array of all possible verticies as well  **struct** Graph\* graph = createGraph(NODE\_MAX);  // seed rand with time  srand((**unsigned**)time(&t));  /// go through all nodes 0-99  **for** (node\_count = **0**; node\_count < NODE\_MAX; node\_count++){  // Function retuns number of random numbers generated, number is limited between 2 & 6  Connected\_Nodes = Rand\_Array(Random\_Connection\_Array);  // Loop for that number create an edge  **for** (count = **0**; count < (Connected\_Nodes-**1**); count++)  {  /// add an edge for each value in the random connection array//  // printf("\nadding connection: %d -> %d", node\_count, Random\_Connection\_Array[count] );  graph\_insert\_edge(graph, node\_count, Random\_Connection\_Array[count]);  }  }  // print the adjacency list representation of the above graph  printGraph(graph);  **return**;  }  **int** **Rand\_Array**(**int** Random\_Connection\_Array[]){  **int** Random\_Number = **0**;  **int** Random\_Nodes\_Connected = **0**;  **int** Random\_Connection = **0**;  **int** count = **0**;    Random\_Nodes\_Connected = RandomNumberGenerate(CONNECT\_MIN, CONNECT\_MAX);  printf(" 0x%x ", Random\_Nodes\_Connected);  //printf("\n Randle: ");  **for** (count = **0**; count < (Random\_Nodes\_Connected - **1**); count++){  Random\_Connection\_Array[count] = RandomNumberGenerate(**0**, NODE\_MAX);  //printf(" %d ", Random\_Connection\_Array[count]);  }  **return** Random\_Nodes\_Connected;  }  **unsigned** **int** **RandomNumberGenerate**(**unsigned** **int** RandomValMin, **unsigned** **int** RandomValMax)  {  **unsigned** **int** RandomNumber = **0**;  RandomNumber = ((**unsigned** **int**)rand() % RandomValMax) + RandomValMin;  **return** (RandomNumber);  } |
| Graph.h |
| #ifndef GRAPH\_H\_INCLUDED\_  #define GRAPH\_H\_INCLUDED\_  #include <stdio.h> /\* printf \*/  #include <time.h> /\* clockt, clock, CLOCKSPERSEC \*/  #include <math.h> /\* sqrt ---- compile with -lm flag \*/  #include <stdlib.h> /\* rand, srand , malloc\*/  #include "Assignment\_6.h"  **typedef** **struct** Vert\_ListNode Node;  **typedef** **struct** Graph Graph\_Node;  Node \*HeadNode, \*PreviousNode, \*CurrentNode;  Graph\_Node \*Current\_Graph\_Node;  // A structure to represent an adjacency list node  **struct** Vert\_ListNode  {  **int** dest;  **struct** Vert\_ListNode\* previous;  **struct** Vert\_ListNode\* next;  };  // A structure to represent HEAD NODE or adjency list  **struct** Vert\_List  {  **struct** Vert\_ListNode \*head; // pointer to head node of list  };  // A structure to represent a graph. A graph is an array of adjacency lists.  // Size of array will be V (number of vertices in graph)  **struct** Graph  {  **int** V;  **struct** Vert\_List\* array;  };  // A utility function to create a new adjacency list node  **struct** Vert\_ListNode\* **newVert\_ListNode**(**int** dest);  // A utility function that creates a graph of V vertices  **struct** Graph\* **createGraph**(**int** V);  // Adds an edge to an undirected graph  **void** **addEdge**(**struct** Graph\* graph, **int** src, **int** dest);  // A utility function to print the adjacenncy list representation of graph  **void** **printGraph**(**struct** Graph\* graph);  **int** **graph\_insert\_edge**(Graph\_Node \*graph, **int** source, **int** destination);  Node\* **graph\_insert\_vertex**(**int** destination);  #endif |
| Graph.c |
| #include "Graph.h"  /\*  Credit goes to the following websites whom I used for refrence  http://www.cs.yale.edu/homes/aspnes/pinewiki/C%282f%29Graphs.html  https://www.cs.bu.edu/teaching/c/graph/linked/  http://www.geeksforgeeks.org/graph-and-its-representations/  \*/  // A utility function that creates a graph of V vertices  **struct** Graph\* **createGraph**(**int** V)  {  **int** i;  Current\_Graph\_Node = (**struct** Graph\*) malloc(**sizeof**(**struct** Graph));  Current\_Graph\_Node->V = V; // store the total number of vertexs in the main graph  // Create an array of verticies. Size of array will be V  Current\_Graph\_Node->array = (**struct** Vert\_List\*) malloc(V \* **sizeof**(**struct** Vert\_List));  // Initialize each adjacency list as empty by making head as NULL  **for** (i = **0**; i < V; ++i){  Current\_Graph\_Node->array[i].head = NULL;  }  **return** Current\_Graph\_Node;  }  **int** **graph\_insert\_edge**(Graph\_Node \*graph, **int** source, **int** destination)  {  **int** chk\_src = **0**;  **int** chk\_Vert\_List = **0**;  // no sence making connections to itslef  **if** (source == destination)  **return** **1**;  //Search through current source Node for destination if connection exists in current src  // no need to recreate connection  CurrentNode = graph->array[source].head;  **for** (; CurrentNode != NULL; CurrentNode = CurrentNode->next){  chk\_Vert\_List++;  chk\_src = CurrentNode->dest;  // Create no connection if destination == source or if links >= 6  **if** (destination == chk\_src || chk\_Vert\_List >= **6**)  **return** **1**;  }  // Add an edge from src to dest. A new node is added to the adjacency  // list of src. The node is added at the begining  CurrentNode = graph\_insert\_vertex(destination);  CurrentNode->next = graph->array[source].head;  graph->array[source].head = CurrentNode;  **return** **0**;  }  // Print the Entire graph, one vertex at a time  **void** **printGraph**(**struct** Graph\* graph)  {  **int** v;  **for** (v = **0**; v < graph->V; ++v)  {  CurrentNode = graph->array[v].head;  printf("**\n** Vertex: %d**\n** ---------------------------------------**\n** head ", v);  **while** (CurrentNode != NULL)  {  printf("-> %d", CurrentNode->dest);  CurrentNode = CurrentNode->next;  }  printf("**\n**");  }  }  Node\* **graph\_insert\_vertex**(**int** destination){  CurrentNode =(**struct** Vert\_ListNode\*) malloc(**sizeof**(**struct** Vert\_ListNode));  CurrentNode->dest = destination;  CurrentNode->next = NULL;  CurrentNode->previous = NULL; // not used or necessary  **return** CurrentNode;  } |