**DISEÑO DE PRUEBAS UNITARIAS**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba N° 1** | **Objetivo**: Probar el método de inserción en un grafo. | | | |
| **Clase** | **Método** | **Escenario** | **Entradas** | **Resultado** |
| AdjacencyListGraph | + insertVertex(V): void | Insertar un vértice sin ninguna conexión con los otros vértices del grafo. | * Valor que almacena el vértice | El método insertó el nuevo vértice que no conecta con ningún otro vértice. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba N° 2** | **Objetivo**: Probar el método que agrega una arista entre dos vértices. | | | |
| **Clase** | **Método** | **Escenario** | **Entradas** | **Resultado** |
| AdjacencyListGraph | + insertEdge(int, int, E): void | Agregar una arista entre dos vértices que no están conectados. | * Vértice origen. * Vértice destino. * Peso de la arista. | El método conectó a los dos vértices con una arista que va desde el vértice origen hasta el vértice destino con el peso pasado por parámetro. |
| AdjacencyListGraph | + insertEdge(int, int, E): void | Conectar un vértice consigo mismo. | * Vértice origen. * Vértice destino (el mismo que el vértice origen). * Peso de la arista. | El método conectó al vértice con él mismo creando un ciclo. |
| AdjacencyListGraph | + insertEdge(int, int, E): void | Agregar una arista entre dos vértices que ya están conectados por otra arista pero en el mismo sentido. | * Vértice origen. * Vértice destino. * Peso de la arista. | El método agregó una nueva arista que va en el mismo sentido que la que ya existía entre esos dos vértices. |
| AdjacencyListGraph | + insertEdge(int, int, E): void | Agregar una arista entre dos vértices que ya están conectados por otra arista pero en sentido opuesto. | * Vértice origen. * Vértice destino. * Peso de la arista. | El método agregó una nueva arista que va en sentido opuesto a la que ya existía entre esos dos vértices. |
| AdjacencyListGraph | + insertEdge(int, int, E): void | Agregar una arista entre dos vértices que ya están conectados. | * Vértice origen. * Vértice destino. * Peso de la arista. | El método agregó una nueva arista entre esos dos vértices. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba N° 3** | **Objetivo**: Probar el método que elimina un vértice del grafo. | | | |
| **Clase** | **Método** | **Escenario** | **Entradas** | **Resultado** |
| AdjacencyListGraph | + deleteVertex(int): void | Eliminar un vértice que no existe. | Índice del vértice. | El método lanzó una excepción de IndexOutOfBoundsException. |
| AdjacencyListGraph | + deleteVertex(int): void | Eliminar un vértice existente que no tenga conexión con otros vértices. | Índice del vértice. | El método eliminó el vértice sin alterar las conexiones de los otros. |
| AdjacencyListGraph | + deleteVertex(int): void | Eliminar un vértice con una conexión a otro vértice. (grafo dirigido) | Índice del vértice. | El método eliminó el vértice junto con la arista que salía de este. |
| AdjacencyListGraph | + deleteVertex(int): void | Eliminar un vértice con una arista incidente a él. (grafo dirigido) | Índice del vértice. | El método eliminó el vértice junto con la arista que incidía en él. |
| AdjacencyListGraph | + deleteVertex(int): void | Eliminar un vértice con múltiples conexiones a otros vértices. | Índice del vértice. | El método eliminó el vértice junto con las aristas que lo conectaban con sus vértices adyacentes. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba N° 4** | **Objetivo**: Probar el método que elimina una arista del grafo. | | | |
| **Clase** | **Método** | **Escenario** | **Entradas** | **Resultado** |
| AdjacencyListGraph | + deleteEdge(int, int, E): void | Eliminar una arista que no existe. | * Índice del vértice origen. * Índice del vértice destino. * Peso de la arista. | El método lanzó una excepción de IndexOutOfBoundsExcpetion. |
| AdjacencyListGraph | + deleteEdge(int, int, E): void | Eliminar una arista entre dos vértices de un grafo simple. | * Índice del vértice origen. * Índice del vértice destino. * Peso de la arista. | El método eliminó la arista. |
| AdjacencyListGraph | + deleteEdge(int, int, E): void | Eliminar una arista entre dos vértices de un multigrafo. | * Índice del vértice origen. * Índice del vértice destino. * Peso de la arista. | El método eliminó la arista sin interferir en las otras conexiones. |
| AdjacencyListGraph | + deleteEdge(int, int, E): void | Eliminar una arista cuyos vértices origen y destino sean iguales. | * Índice del vértice origen. * Índice del vértice destino. * Peso de la arista. | El método eliminó el ciclo. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba N° 5** | **Objetivo**: Probar el método que elimina todas las aristas entre dos vértices. | | | |
| **Clase** | **Método** | **Escenario** | **Entradas** | **Resultado** |
| AdjacencyListGraph | + deleteAllEdge(int, int): boolean | Dos vértices que no están conectados por ninguna arista. | * Índice del vértice origen. * Índice del vértice destino. | El método retornó false. |
| AdjacencyListGraph | + deleteAllEdge(int, int): boolean | Dos vértices conectados por una sola arista y que no tengan conexión con otros vértices. | * Índice del vértice origen. * Índice del vértice destino. | El método eliminó la arista entre esos dos vértices. |
| AdjacencyListGraph | + deleteAllEdge(int, int): boolean | Dos vértices conectados por una sola arista y que tengan conexión con otros vértices. | * Índice del vértice origen. * Índice del vértice destino. | El método eliminó la arista entre esos dos vértices sin eliminar las otras aristas. |
| AdjacencyListGraph | + deleteAllEdge(int, int): boolean | Dos vértices conectados por más de una arista y que no tengan conexión con otros vértices. | * Índice del vértice origen. * Índice del vértice destino. | El método eliminó las aristas incidentes entre los dos vértices. |
| AdjacencyListGraph | + deleteAllEdge(int, int): boolean | Dos vértices conectados por más de una arista y que tengan conexión con otros vértices. | * Índice del vértice origen. * Índice del vértice destino. | El método eliminó las aristas incidentes entre los dos vértices sin eliminar las otras aristas incidentes con otros vértices. |
| AdjacencyListGraph | + deleteAllEdge(int, int): boolean | Un vértice con una sola arista apuntando al mismo sin aristas apuntando a otros vértices. | * Índice del vértice origen. * Índice del vértice destino. | El método eliminó la arista.. |
| AdjacencyListGraph | + deleteAllEdge(int, int): boolean | Un vértice con una sola arista apuntando al mismo y con aristas apuntando a otros vértices. | * Índice del vértice origen. * Índice del vértice destino. | El método eliminó la arista sin eliminar a las otras. |
| AdjacencyListGraph | + deleteAllEdge(int, int): boolean | Un vértice con más de una arista apuntando al mismo y sin otras aristas apuntando a otros vértices. | * Índice del vértice origen. * Índice del vértice destino. | El método eliminó las aristas. |
| AdjacencyListGraph | + deleteAllEdge(int, int): boolean | Un vértice con más de una arista apuntando al mismo y con otras aristas apuntando a otros vértices. | * Índice del vértice origen. * Índice del vértice destino. | El método eliminó las aristas sin eliminar las otras aristas incidentes a otros vértices. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba N° 6** | **Objetivo**: Probar el método BFS desde un vértice de partida. | | | |
| **Clase** | **Método** | **Escenario** | **Entradas** | **Resultado** |
| AdjacencyListGraph | + BFS(int): ArrayList<Vertex<T>> | Un grafo de un solo vértices. | * Índice del vértice de origen. | El método devolvió un ArrayList de tamaño 1. |
| AdjacencyListGraph | + BFS(int): ArrayList<Vertex<T>> | Un grafo no conexo de n vértices. | * Índice del vértice de origen. | El método devolvió un ArrayList de tamaño n menos el número de vértices de los otros subgrafos. |
| AdjacencyListGraph | + BFS(int): ArrayList<Vertex<T>> | Un grafo conexo de n vértices. | * Índice del vértice de origen. | El método devolvió un ArrayList de tamaño n. |
| AdjacencyListGraph | + BFS(int): ArrayList<Vertex<T>> | Un grafo con un ciclo. | * Índice del vértice de origen. | El método devolvió un ArrayList de tamaño n. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba N° 7** | **Objetivo**: Probar el método BFS sin ningún vértice de partida. | | | |
| **Clase** | **Método** | **Escenario** | **Entradas** | **Resultado** |
| AdjacencyListGraph | + BFS(): ArrayList<ArrayList<Vertex<T>>> | Un grafo de un solo vértices. | Ninguna. | El método devolvió un ArrayList de tamaño 1. |
| AdjacencyListGraph | + BFS(): ArrayList<ArrayList<Vertex<T>>> | Un grafo no conexo de n vértices. | Ninguna. | El método devolvió varios ArrayList. |
| AdjacencyListGraph | + BFS(): ArrayList<ArrayList<Vertex<T>>> | Un grafo conexo de n vértices. | Ninguna. | El método devolvió varios ArrayList. |
| AdjacencyListGraph | + BFS(): ArrayList<ArrayList<Vertex<T>>> | Un grafo con un ciclo. | Ninguna. | El método devolvió varios ArrayList. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba N° 8** | **Objetivo**: Probar el método DFS desde un vértice de partida. | | | |
| **Clase** | **Método** | **Escenario** | **Entradas** | **Resultado** |
| AdjacencyListGraph | + DFS(int): ArrayList<Integer> | Recorrer un grafo de un solo vértice. | * Índice del vértice desde donde inicia el recorrido. | El método devolvió un árbol DFS con un solo índice. |
| AdjacencyListGraph | + DFS(int): ArrayList<Integer> | Recorrer un grafo conexo. | * Índice del vértice desde donde inicia el recorrido. | El método devolvió un árbol DFS con la misma cantidad de índices que de nodos en el grafo. |
| AdjacencyListGraph | + DFS(int): ArrayList<Integer> | Recorrer un grafo no conexo. | * Índice del vértice desde donde inicia el recorrido. | El método devolvió el árbol de expansión mínima correspondiente al índice de inicio. |
| AdjacencyListGraph | + DFS(int): ArrayList<Integer> | Recorrer un grafo con un ciclo. | * Índice del vértice desde donde inicia el recorrido. | El método no tuvo problemas y devolvió el árbol de expansión mínima correspondiente sin haber repetido la visita al vértice con el ciclo. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba N° 9** | **Objetivo**: Probar el método DFS que recorre todo el grafo sin ningún vértice de partida. | | | |
| **Clase** | **Método** | **Escenario** | **Entradas** | **Resultado** |
| AdjacencyListGraph | + DFS(): ArrayList<ArrayList<Integer>> | Recorrer un grafo de un solo vértice. | Ninguna. | El método devolvió un árbol DFS con un solo índice. |
| AdjacencyListGraph | + DFS(): ArrayList<ArrayList<Integer>> | Recorrer un grafo conexo. | Ninguna. | El método devolvió un árbol DFS con la misma cantidad de índices que de nodos en el grafo. |
| AdjacencyListGraph | + DFS(): ArrayList<ArrayList<Integer>> | Recorrer un grafo no conexo. | Ninguna. | El método devolvió una cantidad de árboles de expansión mínima correspondiente al número de subgrafos conexos en el grafo no conexo. |
| AdjacencyListGraph | + DFS(): ArrayList<ArrayList<Integer>> | Recorrer un grafo con un ciclo. | Ninguna. | El método no tuvo problemas y devolvió el árbol o los árboles de expansión mínima correspondientes sin haber repetido la visita al vértice con el ciclo. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba N° 10** | **Objetivo**: Probar el método Prim. | | | |
| **Clase** | **Método** | **Escenario** | **Entradas** | **Resultado** |
| AdjacencyListGraph | + Prim(int): ArrayList<Integer> | Un grafo de un solo vértice. | Índice del vértice desde donde se inicia el recorrido. | El método lanzó una excepción porque no hay aristas en el grafo. |
| AdjacencyListGraph | + Prim(int): ArrayList<Integer> | Recorrer un grafo conexo. | Índice del vértice desde donde se inicia el recorrido. | El método devolvió un árbol de expansión mínima con todos los vértices del grafo. |
| AdjacencyListGraph | + Prim(int): ArrayList<Integer> | Recorrer un multigrafo. | Índice del vértice desde donde se inicia el recorrido. | El método devolvió el árbol de expansión mínima. |
| **Prueba N° 11** | **Objetivo**: Probar el método Kruskal. | | | |
| **Clase** | **Método** | **Escenario** | **Entradas** | **Resultado** |
| AdjacencyListGraph | + Kruskal(int): ArrayList<Integer> | Un grafo de un solo vértice. | Índice del vértice desde donde se inicia el recorrido. | El método lanzó una excepción porque no hay aristas en el grafo. |
| AdjacencyListGraph | + Kruskal(int): ArrayList<Integer> | Recorrer un grafo conexo. | Índice del vértice desde donde se inicia el recorrido. | El método devolvió un árbol de expansión mínima con todos los vértices del grafo. |
| AdjacencyListGraph | + Kruskal(int): ArrayList<Integer> | Recorrer un multigrafo. | Índice del vértice desde donde se inicia el recorrido. | El método devolvió el árbol de expansión mínima. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Prueba N° 12** | **Objetivo**: Probar el método Dijkstra. | | | |
| **Clase** | **Método** | **Escenario** | **Entradas** | **Resultado** |
| AdjacencyListGraph | + Dijkstra(int): Object[] | Un grafo de un solo vértice. | Índice del vértice desde donde se inicia el recorrido. | El método devolvió el arreglo de distancias con respecto al nodo origen y el de predecesores ambos de tamaño 1. |
| AdjacencyListGraph | + Dijkstra(int): Object[] | Recorrer un grafo conexo simple. | Índice del vértice desde donde se inicia el recorrido. | El método devolvió el arreglo de distancias con respecto al nodo origen y el de predecesores ambos de tamaño mayor a uno. |
| AdjacencyListGraph | + Dijkstra(int): Object[] | Recorrer un multigrafo conexo. | Índice del vértice desde donde se inicia el recorrido. | El método devolvió el arreglo de distancias con respecto al nodo origen y el de predecesores ambos de tamaño mayor a uno. |
| AdjacencyListGraph | + Dijkstra(int): Object[] | Hacer el recorrido desde un vértice que no existe. | Índice del vértice desde donde se inicia el recorrido. | El método arrojó una excepción de IndexOutOfBoundsExceptio. |