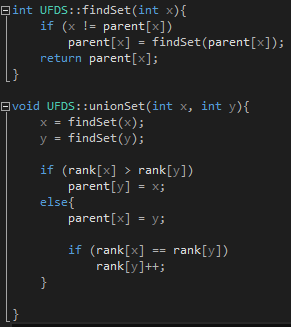
***HASH TABLES***

* Arreglos pequeños
* Cada elemento posee una clave (no existen dos con la misma)
* Representan con conjunto dinámico
* **Métodos de Compresión**
  + M. Aritmética Modular
    - **Lineal** = (key % m) -> (key + n) % m
    - **Cuadrático** = (key % m) -> (key + ) % m
    - **Double Hashing** = (key % m) -> ((key%m+1)-(key % (key%m+1))) % m
  + M. Multiplicación
* **Gestión de Colisiones**
  + **Hashing Abierto**. Listar para cada posición.
  + **Hashing Cerrado**. Buscar un espacio libre hasta encontrarlo.
* **Construcción del código**
  + Se necesita una clase **HashEntry** con dos atributos **key, value** con sus get y constructor con parámetros.
  + Se necesita una clase **HashMap** que tenga como atributo un **HashEntry\*\*** y métodos **get(int key)** y **put(key, value)**
  + Se crean métodos **putLineal, putCuadratico, putDouble** con sus respectivos métodos recursivos que te devuelven un hash válido.

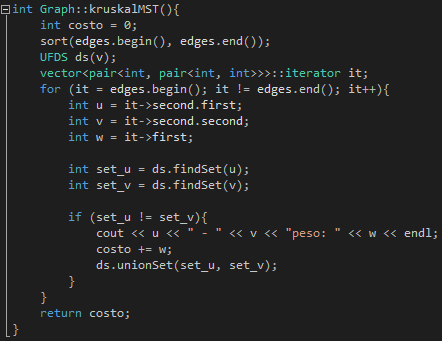
***Union Find Disjoint Sets***

* Permite identificar o rastrear elementos particionados en diferentes grupos o «sets».
* Útil para detectar ciclos dentro de un grafo.
* Internamente utiliza un arreglo de padres.
* Maneja 4 funciones:
  + Inicialización (**initSet**)
  + Unión (**unionSet**)
  + Busqueda (**findSet**) -> Decir quién es su padre
  + MismoSet (**isSameSet**) -> compara si tienen el mismo padre
* **unionSet** (a, b)
  + Si tienen igual rango b se convierte en el padre de a
  + Si tienen distinto rango el de mayor será el padre
  + Si b es un elemento que se encuentra ubicado en los rangos altos y se compara con a, b se ubica en un rango menor con el mismo rango que se ubicará a, pasando los dos a ser hijos de un solo padre.
* **Construcción del código**
  + Se necesitan tres atributos: **parent\***, **rank\*** y **n**
  + Los ranks se inicializan en 0 y cada elemento será su propio padre (parent[i] = i)

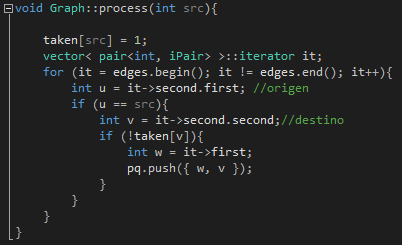


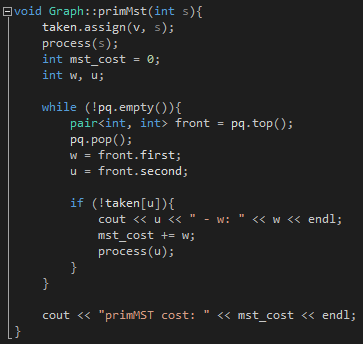
***MST***

* Requisitos: **Grafo no dirigido y conexo**, solo **pesos positivos**, las conexiones entre nodos **no deben formar ciclos**.
* Un árbol de un grafo: Es conexo y sin ciclos. Tiene **n** nodos y **n-1** aristas
* **KRUSKAL O(nlogn)**
  + Al comenzar se selecciona la arista con menor peso.
  + Después se selecciona la siguiente arista con menor peso y se repite lo mismo sin formar ciclos hasta visitar todos los nodos.
  + **Construcción del código**
    - Se necesita una clase **UFDS**
    - Se necesita una clase **Graph** con los atributos: **v** (N° vértices), **e** (N° aristas), **vector<pair<int,pair<int,int>>> edges** (vector de aristas).
    - Se necesita un constructor donde se ingrese el valor de **v** y **e**. Además, se necesita un método addEdge(u, v, w){ edges.push\_back({w, {u, v}}).
    - Se necesita el método **kruskalMST()** que retorna el valor del costo mínimo.



* **PRIM O()**
  + Al comenzar se selecciona la arista con menor peso.
  + Después se selecciona la arista con menor peso incidente con la(s) ya seleccionada(s) sin formar ciclos. Se repite el proceso hasta encontrar el costo mínimo con todos los nodos visitados.
  + **Construcción del código**
    - Se necesita lo mismo que para Kruskal
    - Se crea el método **primMST(int src)** y **process(int src)**
      * En **process:** src se marca como visitado
      * se recorre las aristas
      * para cada vértice origen que sea igual al src
      * se comprueba si su destino no ha sido visitado
      * si es verdad se pushea a la cola de prioridad el peso y el destino





***Bellman Ford***

* **Contrucción del código**
  + Se necesita un struct Edge con u, v, w y se crea un arreglo
  + Se necesita un arreglo d[] con que representará las distancias del source a los demás nodos
  + En el método BellmanFord(int src)
    - Para cada d[] de un nodo se le asigna distancia infinita a excepción del source que es cero.
    - Luego se compara en cada vértice todas las aristas si el peso actual origen + el peso de la arista es menor al peso actual del destino. Si lo es el peso del destino pasa a ser el peso del origen + el peso de la arista.
    - Al final se compara si existe ciclos negativos.