UNIVERSIDAD DE CONCEPCIÓN
Departamento de Ingeniería Informática y
Ciencias de la Computación

Estructuras de Datos (2022-1)

Proyecto 2
Implementación ADT MAP

Profesor: Alexander Irribarra

Ayudantes: Leonardo Aravena
Diego Gatica
Vicente Lermanda

Alumnos: Jaime Ansorena Carrasco
Vicente Tuki Ibáñez

15 de Julio de 2022

1. Introducción

Los mapas son estructuras de datos que almacenan colecciones de pares (clave, valor), y que permiten acceder a los valores a partir de sus claves asociadas. Además, las claves son únicas dentro de un mapa.

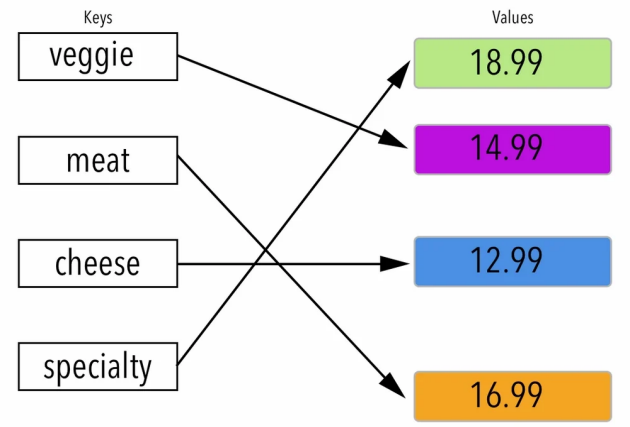

Figura 1: Ejemplo de un mapa

Un problema recurrente es diseñar e implementar una estructura que sea eficiente al momento de buscar pares (clave,valor). En el siguiente proyecto se implementaran tres tipos de mapas:

- Mapas basados en vectores.
- Mapas basados en tablas hash.
- Mapas basados en arboles binarios balanceados (AVL).

Además se realizaran análisis teóricos y experimentales para cada una de las soluciones propuestas.

2. Soluciones Propuestas

2.1. MapSV

La primera implementación corresponde a un mapa implementado con vectores. En esta implementación el vector se mantiene ordenado en base a las claves y la búsqueda se realiza mediante búsqueda binaria.

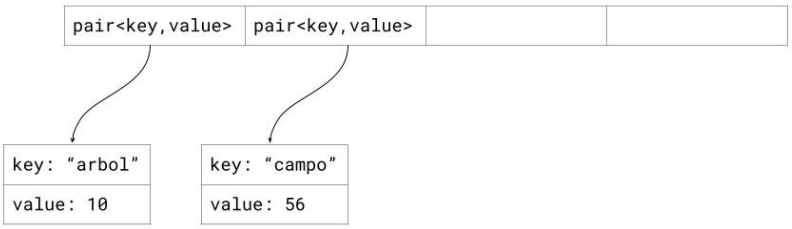

Figura 2: Ejemplo de mapa implementado con vectores

2.2. MapH

La segunda implementación corresponde a un mapa implementado mediante tablas hash. Esta implementación funciona transformando la clave con una función hash en un valor hash, un entero que identifica la posición del vector donde se guardara el par (clave, valor). Para tratar las colisiones se utiliza doble hashing, es decir, una segunda función hash que identifica la posición en caso que la posición original este utilizada por otro par.

Para esta implementación se determino calcular las funciones hash mediante el método de Horner (Lab7-JaimeAnsorena), debido a que experimentalmente calculó mejores tiempos constantes que la acumulación polinomial.

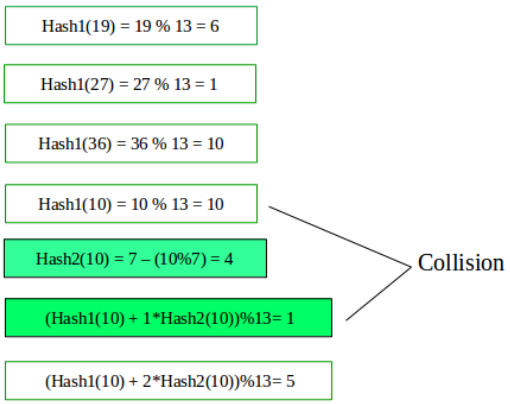

Figura 3: Ejemplo de mapa implementado con tablas hash

2.3. MapAVL

La ultima implementación corresponde a un mapa implementado con árboles binarios balanceados (AVL). En los árboles AVL, las alturas de los dos subárboles hijos de cualquier nodo difieren como máximo en uno. En caso contrario, se debe realizar un balance para restaurar esta propiedad. Además, las inserciones y eliminaciones pueden requerir el balance del árbol mediante una o varias rotaciones del mismo.

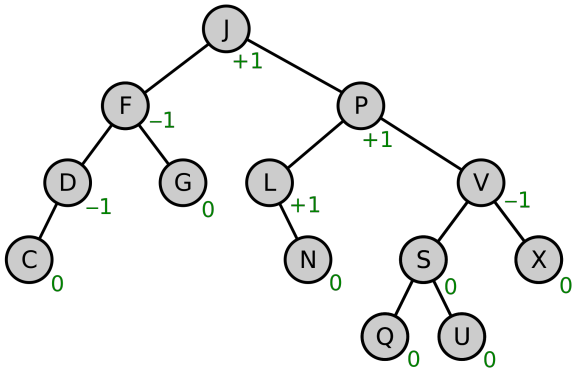

Figura 4: Ejemplo de mapa implementado con AVL junto con las diferencias de altura de los nodos

Los detalles de cada implementación se verán en la sección siguiente.

3. Detalles de la implementación

3.1. MapSV

3.1.1. MapSV::insert

El primer paso para insertar el par (clave,valor) consiste en realizar una búsqueda binaria para determinar si existe el elemento. En caso que no existe agregamos el elemento al vector en la posición del iterador que devuelve la búsqueda binaria.

```
void MapSV::insert(const string &s, int n){
    // busqueda binaria
    auto it = lower_bound(v.begin(), v.end(), make_pair(s,0));

    if (it != v.end() && it->first == s)
        std::cout << "elemento duplicado" << endl;
    else{
        //insertar al vector, desplazando los demas elementos
        v.insert(it,{s,n});
    }
}
```

3.1.2. MapSV::erase

Para borrar elementos del vector, se realiza primero una búsqueda binaria en base a la clave del argumento. Si se encuentra el par se realiza un erase del vector. Como este se encuentra ordenado, el algoritmo desplaza en una posición los elementos a la derecha del par eliminado.

```
void MapSV::erase(const string &s){
    auto it = lower_bound(v.begin(), v.end(), make_pair(s,0));
    if (it != v.end() && it->first == s) v.erase(it);
}
```

3.1.3. MapSV::at

La función at realiza una búsqueda binaria sobre el vector. En caso de encontrar el par, devuelve el valor de este.

```
int MapSV::at(const string &s){
    auto it = lower_bound(v.begin(), v.end(), make_pair(s,0));
    if (it != v.end() && it->first == s) return it->second;
    else return -1;
}
```

3.1.4. MapSV::size

La funcion size retorna el tamaño del vector v.

```
int MapSV::size(){
    return v.size();
}
```

3.1.5. MapSV::empty

La funcion empty retorna true en caso que el vector v este vacio y false en caso contrario.

```
bool MapSV::empty(){
    if(v.empty()) return true;
    else return false;
}
```

3.2. MapH

Para la implementación con tablas hash, se utilizaron 3 funciones auxiliares. Las dos primeras corresponden a las funciones hash. La primera permite obtener el valor hash mediante la regla de Horner, es decir, en tiempo lineal. La segunda funcion hash entrega una posición relativa respecto a la primera función en base a un número primo. Este valor es siempre mayor que 0.

La tercera función corresponde a la de rehashing. Se crea un nuevo vector de mínimo el doble de tamaño que el original. Para esto se utiliza un conjunto de números primos precalculados. Para cada par de la tabla original se vuelve a calcular su valor hash, mediante las dos funciones y se almacena en una nueva posición en el vector. Finalmente se cambia la variable v al nuevo vector creado.

```
int functionHash1(const string& s, int size){
    int pos = 0;
    int a = 127;
    for (int i = 0; i < s.length(); i++){
        pos = (a*pos + s[i]) % size;
    }
    return pos;
}

int functionHash2(const string& s, int size){
    int pos = functionHash1(s,size);
    pos = 997 - pos%997;
    return pos;
}
```

```
void MapHash::rehashing(vector<pair<string,int>> &v){
    vector<pair<string,int>> aux(primes[countPrimes]); // nuevo vector del doble de tamano
    countPrimes++;
    for (int i = 0; i < v.size(); i++){ // ingresar los valores al nuevo vector
      if(v[i].first != "\0" && v[i].first != "@"){
            int pos = functionHash1(v[i].first,aux.size());
            int pos2 = functionHash2(v[i].first,aux.size());
            while(true){
              if(aux[pos].first == "\0"){
                aux[pos] = make_pair(v[i].first,v[i].second);
                break;}
             pos = (pos+pos2) % aux.size();}
        }
    }
    v = aux;}
```

3.2.1. MapH::insert

Para cada par, calculamos su primera función hash en base a la string del argumento. Si existen colisiones, ocupamos la segunda función hash para calcular una nueva posición. Finalmente si el factor de carga es mayor a 0.5, es decir, si el vector tiene mas de la mitad de su tamaño ocupado, se realiza la función rehashing.

```
void MapHash::insert(const string &s, int n){
    int pos = functionHash1(s,v.size());
    int pos2 = functionHash2(s,v.size());
    while(true){
       if(v[pos].first == s) break; // no admite claves duplicadas
       if(v[pos].first == "\0" || v[pos].first == "@"){
          v[pos] = make_pair(s,n);
          sizeMap++;
          break;
       }else{
          pos = (pos+pos2) % v.size();
            colisions++;
        }
    }
    if(sizeMap > v.size()/2) rehashing(v);
}
```

3.2.2. MapH::erase

Para la función erase, se calculan ambos valores hash y se itera por las posiciones hasta encontrar el elemento, donde se realiza un borrado lógico. En caso de encontrar una posición vacía, el elemento no existe en la tabla hash.

```
void MapHash::erase(const string &s){
    int pos = functionHash1(s,v.size());
    int pos2 = functionHash2(s,v.size());
   while(true){
      if(v[pos].first == s){ // borrado logico
         v[pos].first = "@";
         v[pos].second = 0;
         sizeMap--;
         break;
      }
      if(v[pos].first == "\0") break; // no se encuentra el elemento
      pos = (pos+pos2) % v.size(); // siguiente posicion
   }
}
```

3.2.3. MapH::at

Al igual que los métodos anteriores, calculamos ambas funciones hash, y se realiza una búsqueda por la tabla en base a los valores hash obtenidos. Si se encuentra la clave, se retorna el valor asociado a esa clave. En caso de encontrar una posición vacía, retorna 0.

```
int MapHash::at(const string &s){
    int pos = functionHash1(s,v.size());
    int pos2 = functionHash2(s,v.size());
   while(true){
      if(v[pos].first == s) return v[pos].second;
      if(v[pos].first == "\0") return 0;
      pos = (pos+pos2) % v.size();
   }
}
```

3.2.4. MapH::empty

Si el tamaño de la tabla es 0 retorna true. En caso contrario, retorna false.

```
bool MapHash::empty(){
   if(sizeMap == 0) return true;
   else return false;
}
```

3.2.5. MapH::size

Retorna el tamaño de la tabla hash.

```
int MapHash::size(){ return sizeMap; }
```

3.3. MapAVL

3.3.1. MapAVL::insert

Para el método insert se agrega la clave y el valor a un par p. Si el árbol esta vacío, se agrega a la raíz del árbol. En caso contrario, se crean los nodos aux1 y aux2 con el fin de saber el camino y encontrar el nodo donde insertar. Cuando se encuentra el sub árbol nulo, crea un nuevo nodo aux0, y se guarda el par clave-cadena. Finalmente se aplica el método balanceo en aux0.

```
void MapAVL::insert(const string &s, int n){
    pair<string,int> p = {s,n};
    if(tam == 0){
        nodo *aux0 = new nodo();
        aux0->tree = p;
        aux0->altura = 1;
        aux0->hijoI = nullptr;
        aux0->hijoD = nullptr;
        aux0->Padre = nullptr;
        this->root = aux0;
        this->tam += 1;
    }else{
        nodo *aux1 = this->root;
        nodo *aux2 = aux1;
        while(aux1 != nullptr ){
            if(aux1 != nullptr && p == aux1->tree) return;
            if(aux1 != nullptr && p > aux1->tree){
                aux2 = aux1;
                aux1 = aux1->hijoD;
            }
            if(aux1 != nullptr && p < aux1->tree){
                aux2 = aux1;
                aux1 = aux1->hijoI;
            }
        }

        nodo *aux0 = new nodo();
        aux0->tree = p;
        aux0->altura = 1;
        aux0->Padre = aux2;
        if(aux0->tree > aux2->tree){
            aux2->hijoD = aux0;
            this->tam+=1;
        }else if(aux0->tree < aux2->tree){
            aux2->hijoI = aux0;
            this->tam+=1;
        }
        balanceo(aux0);
    }
}
```

3.3.2. MapAVL::at

Método recursivo que itera sobre el árbol, encontrando un sub-árbol nulo o el string s. Finalmente se retorna el valor asociado a la clave.

```
int MapAVL::at(const string &s){
    nodo *aux = this->root;
    while(aux!=nullptr){
        if(aux->tree.first == s){
            break;
        }
        else if(s > aux->tree.first){
            aux = aux->hijoD;
        }
        else if(s < aux->tree.first){
            aux = aux->hijoI;
        }
    }
    int t_aux = INT_MIN;
    if(aux!=nullptr)t_aux = aux->tree.second;
    return t_aux;
}
```

3.3.3. MapAVL::size

Método que devuelve el tamaño del AVL.

```
int MapAVL::size(){
    return this->tam;
}
```

3.3.4. MapAVL::empty

Método que retorna true si el árbol esta vació o false en caso de que existan elementos.

```
bool MapAVL::empty(){
    if(this->root == nullptr) return true;
    return false;
}
```

3.3.5. MapAVL::erase

Método que elimina el nodo asociado a la clave. El método recorre iterativamente el árbol hasta encontrar el nodo asociado a la clave s. Si existe solo un elemento en el árbol, des-referencia el puntero y lo elimina. Si existe mas de un elemento asociado, lo elimina a través de removeExternal y luego se verifica mediante balanceo del árbol que el padre v no es null. Si se tiene un nodo interno y el tamaño es mayor a 1, se busca el sucesor del nodo asociado a s y al padre del sucesor de v. Por ultimo se asigna el sucesor a v y se verifica si es externo o no para eliminarlo con el método removeExternal.

```
void MapAVL::erase(const string &s){
    nodo *v = this->root;
    if(this->root == nullptr) return; //Si es nula la raiz el metodo termina
    while(this->tam > 0 && (v->tree.first != s)){
        if(v == nullptr) return;
        if(s > v->tree.first) v = v->hijoD;
        if(s < v->tree.first) v = v->hijoI;
    }

    if(this->tam == 1){ //Si el tam es 1 se busca si hay mas de un elemento asociado y
         borra
        delete v;
        this->root = nullptr;
        tam = tam-1;
    }else if(!isInternal(v) && tam > 1){ //Si el tam es mayor a 1 encuentra al sucesor del
         padre
        nodo *Padre_v = v->Padre;
        removeExternal(v);
        if(Padre_v != nullptr) balanceo(Padre_v);
        tam = tam-1;
    }
    else if(tam > 1){
        nodo *sucesor = get_sucessor(v);
        nodo *Padre_s = sucesor->Padre;
        v->tree = sucesor->tree;
        if(!isInternal(sucesor)) removeExternal(sucesor);
        if(sucesor == Padre_s->hijoI) Padre_s->hijoI = nullptr;
        if(sucesor == Padre_s->hijoD) Padre_s->hijoD = nullptr;
        tam = tam-1;
        balanceo(Padre_s);
    }
}
```

Los métodos auxiliares implementados en el MapAVL, para el balanceo del árbol binario, se encuentran detallados y comentados en el archivo "MapAVL.cpp"

4. Análisis Teórico

4.1. MapSV

Para la estructura MapSV, implementada con un vector ordenado, la operación insertar y remover tiene una complejidad $\boldsymbol{O(n)}$, ya que en peor caso se debe desplazar n elementos, al insertar o remover del vector. Como la búsqueda se realiza mediante búsqueda binaria, el método at tiene una complejidad $\boldsymbol{O(log\, n)}$.

4.2. MapH

En la estructura implementada con tablas hash, las operaciones presentan una complejidad promedio de $\boldsymbol{O(1)}$, ya que mediante un factor de carga bajo, junto con hashing doble, se puede encontrar las claves en tiempo constante. Sin embargo, en peor caso, las operaciones tienen una complejidad $\boldsymbol{O(n)}$, ya que se debe recorrer toda la tabla hash en búsqueda de las claves.

4.3. MapAVL

Para el caso de la estructura MapAVL, al ser un árbol binario, sus operaciones dependen de la altura del mismo. Al estar balanceado, se establece la garantía que las operaciones insert, remove y at presentan una complejidad $\boldsymbol{O(log\, n)}$.

5. Análisis Experimental

Para el calculo experimental se considero el tiempo promedio que demoran las distintas operaciones.

Como cadenas de texto se generaron aleatoriamente 10000 strings de largo 6, utilizando los 26 caracteres de la tabla ASCII. Esta generación aleatoria se realizo mediante la función printRandomString, y se guardo en un archivo de texto llamado words.txt.

```
string printRandomString(int n, int size){
    char alphabet[26] = {'a','b','c','d','e','f','g','h', 'i', 'j', 'k', 'l', 'm', 'n',
                         'o', 'p', 'q', 'r', 's', 't', 'u','v', 'w', 'x', 'y', 'z' };
    string res = "";
    for (int i = 0; i < n; i++) res = res + alphabet[rand() % size];
    return res;
}
```

El entorno de software corresponde al S.O. Ubuntu 22.04 y las pruebas se realizaron bajo un procesador Intel i5-7400 con 8 GB de memoria RAM.

El archivo words.txt se guarda en un vector, junto con un valor generado aleatoriamente.

Para el método insert se realizan n cantidad de inserciones a cada estructura de datos y se calcula el tiempo promedio de cada operación.

Para el método at se realizan n cantidad de búsquedas, obteniendo la string a buscar desde el vector, en cada estructura y se calcula el tiempo promedio de cada operación.

Para el método erase, se realizan n cantidad de borrados, obteniendo la string a borrar desde el vector, y se calcula el tiempo promedio de cada operación.

Los resultados obtenidos se resumen en las siguientes tablas:

n	insert MapSV (s)	insert MapH (s)	insert MapAVL (s)
1000	0.0000036620	0.0000004600	0.0000006220
2000	0.0000068890	0.0000005985	0.0000006870
3000	0.0000103253	0.0000008093	0.0000007140
4000	0.0000133130	0.0000006955	0.0000007510
5000	0.0000162676	0.0000009552	0.0000007890
6000	0.0000193652	0.0000008637	0.0000007800
7000	0.0000228249	0.0000007294	0.0000007979
8000	0.0000259004	0.0000006899	0.0000008200
9000	0.0000288272	0.0000011062	0.0000008692
10000	0.0000318967	0.0000010166	0.0000008761

Cuadro 1: Análisis experimental método insert

n	at MapSV (s)	at MapH (s)	at MapAVL (s)
1000	0.0000004680	0.0000002250	0.0000003050
2000	0.0000005220	0.0000001670	0.0000003400
3000	0.0000005553	0.0000001803	0.0000003577
4000	0.0000005530	0.0000001780	0.0000003875
5000	0.0000005882	0.0000001596	0.0000003946
6000	0.0000005878	0.0000001640	0.0000003982
7000	0.0000006001	0.0000001913	0.0000004089
8000	0.0000005955	0.0000001756	0.0000004399
9000	0.0000006201	0.0000001731	0.0000004249
10000	0.0000006466	0.0000001757	0.0000004330

Cuadro 2: Análisis experimental método at

n	erase MapSV (s)	erase MapH (s)	erase MapAVL (s)
1000	0.0000036910	0.0000001760	0.0000004540
2000	0.0000067470	0.0000001745	0.0000004750
3000	0.0000095910	0.0000001717	0.0000005080
4000	0.0000128925	0.0000001800	0.0000005307
5000	0.0000161054	0.0000001760	0.0000005684
6000	0.0000189102	0.0000001763	0.0000005757
7000	0.0000218394	0.0000001850	0.0000005929
8000	0.0000248503	0.0000001889	0.0000006069
9000	0.0000279361	0.0000001977	0.0000006383
10000	0.0000310823	0.0000001986	0.0000006340

Cuadro 3: Análisis experimental método erase

En base a las tablas de análisis experimental se realizan los siguientes gráficos comparando las distintas implementaciones:

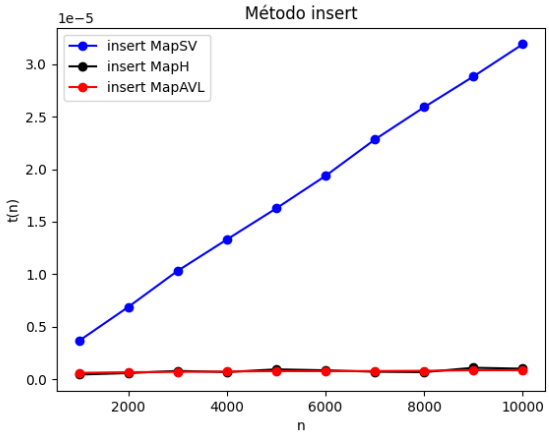

Figura 5: Análisis experimental método insert

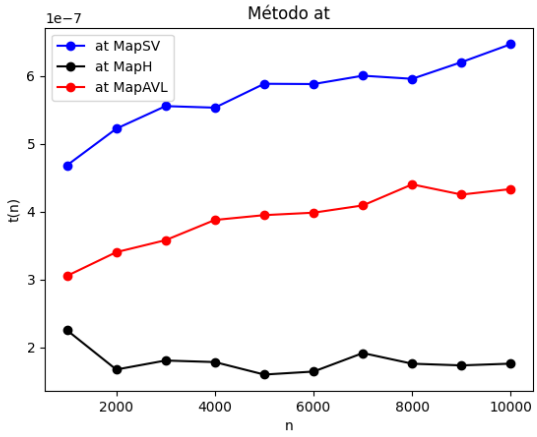

Figura 6: Análisis experimental método at

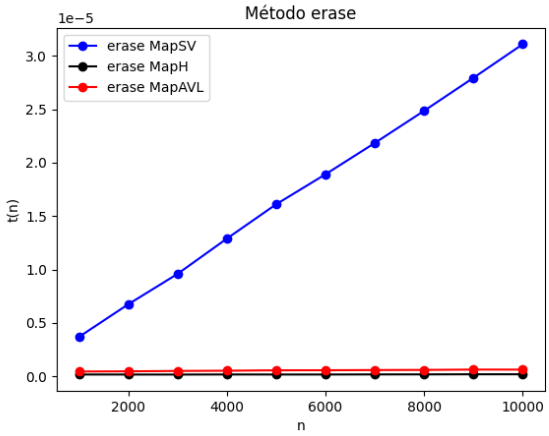

Figura 7: Análisis experimental método erase

A partir de los resultados obtenidos se verifica que para la estructura MapSV, el método insert presenta una complejidad $\boldsymbol{O(n)}$, ya que en peor caso, debe desplazar n elementos hacia la derecha del vector; para la estructura MapH, una complejidad $\boldsymbol{O(1)}$, aumentando solo en los valores de n que presenta rehashing y para MapAVL una complejidad de $\boldsymbol{O(log\ n)}$, al ser un árbol binario balanceado.

Para el metodo at, las estructuras MapSV y MapAVL presentan una complejidad $\boldsymbol{O(log\ n)}$, mientras que MapH presenta una complejidad $\boldsymbol{O(1)}$.

Finalmente para el método erase, las complejidades son las mismas que para el metodo insert, es decir, $\boldsymbol{O(n)}$ para MapSV, $\boldsymbol{O(1)}$ para MapH y $\boldsymbol{O(log\ n)}$ para MapAVL.

Cabe destacar que en el caso de la tabla hash, no es forzado el peor caso, que es $\boldsymbol{O(n)}$, ya que el factor de carga, al ser 0.5, no permite que ocurran tantas colisiones.

6. Conclusiones

A partir de las implementaciones y sus resultados se pueden concluir los siguientes tópicos:

- **Tiempo de búsqueda**:

  Si sabe que sus claves (lo que está usando para buscar sus datos) se encuentran en un rango de enteros muy estrecho, entonces una matriz (o preferiblemente un vector) es ideal si lo mas importante es la búsqueda. Se puede buscar usando una clave en tiempo constante.

- **Orden**:

  El mapa almacena elementos en orden clave. De modo que puede iterar sobre todos los elementos de un mapa, de principio a fin, en orden de clave ordenado. Obviamente, se puede realizar con una matriz/vector, sin embargo, como se explico anteriormente, un mapa le permite tener cualquier tipo de clave arbitraria con cualquier orden arbitrario definido.

- **Inserción**:

  Para insertar en el medio de una matriz/vector, debe desplazar todos los elementos hacia la derecha. Para una matriz dinámica y un vector, debe cambiar el tamaño del vector, lo que hará que copie toda la matriz en la nueva memoria. Un mapa mediante arboles tiene un tiempo de inserción razonable. $\boldsymbol{O(log\, n)}$.

- **Otras alternativas**:

  Un multimap es un map pero permite que las claves no sean únicas. unordered_map es un mapa que no almacena elementos en orden, pero puede proporcionar un mejor rendimiento de búsqueda si se proporciona una buena función hash.

En general la implementación MapSV presenta los peores resultados, al insertar y remover en tiempo lineal en caso promedio.

Para las otras estructuras, la mejor implementación depende de las garantías de peor caso requeridas. La estructura MapH presenta en promedio complejidades constantes para todas sus operaciones, pero complejidades lineales en peor caso.

En caso de necesitar la garantía de peor caso, la estructura MapAVL es la mas adecuada, ya que presenta complejidades logarítmicas para todas las operaciones, en peor y caso promedio.

7. Referencias

1. "Data Structures and Algorithms in Java", Michael Goodrich and Roberto Tamassia, John Wiley & Sons, 2006, 4th Edition, ISBN: 0-471-73884-0.

2. Visualizacion de AVL Tree.
   Recuperado de https://www.cs.usfca.edu/~galles/visualization/AVLtree.html.

3. Diapositivas de la clase Estructura de Datos-Dictionaries. Profesor Alexander Irribarra.