Universidad de Concepción

Proyecto 2
Estructuras de Datos:
Maps

Prof. Diego Seco
Ayudantes: Diego Gatica y Alexander Irribarra

Integrantes: Javiera Cerda Bastías.
Pablo Furet Pereira.

Introducción

Para este proyecto se implementó un tipo abstracto de datos, llamado ADTMap, el cual soporta las siguientes operaciones:

- void insert(pair<string,int>)
- int at(string)
- void erase(string)
- bool empty()
- int size()

Fueron realizadas tres implementaciones distintas de Map, a partir de *ADTMap*, las cuales tienen un funcionamiento y eficiencia completamente distinto entre ellas.

La primera implementación realizada fue *MapSV*, la cual consiste principalmente en elementos ordenados, distribuidos en un vector. La segunda implementación fue *MapH*, basada en *Hashing*, la cual emplea la mejor implementación de las realizadas en el laboratorio 7, es decir, utilizando *double hashing* para lidiar con las colisiones. La última a implementar fue *MapAVL*, utiliza un árbol *AVL*, o en otras palabras, un árbol binario de búsqueda auto-balanceado.

A continuación se presentan cada una de estas implementaciones en detalle: descripción de la solución, pseudocodigos y análisis de peor caso para las operaciones insert, at y erase. También se presentarán las comparaciones de análisis teóricos y experimentales ,y finalmente, una conclusión generada a partir de los datos obtenidos.

Implementación

MapSV

MapSV es un mapa implementado en base a un simple vector, en el cual procuramos insertar y eliminar elementos manteniendo el orden alfabético de las llaves de los elementos, lo que puede llegar a ser muy costoso en términos de tiempo, con el fin de poder encontrar cualquier elemento rapidamente usando busqueda binaria.

Esta solución se caracteriza por ser muy sencilla de implementar, sin embargo los tiempos de insercion y eliminacion escalan rápidamente a medida que aumentamos la cantidad de datos, haciéndola optima solo para conjuntos pequeños.

Las declaraciones e implementaciones de cada una de las funciones utilizadas para esto se encuentran en los ficheros MapSV.h y MapSV.cpp respectivamente.

insert() :

input: *data* es un *pair<string, int>* compuesto de su *key* y *value*

```
function insert(data) {
    if this.isEmpty() {
    //map es el vector de pares en donde guardamos los datos
        map.push(data)
        _size++
    } else {
        pos = busquedaBinaria(data.key)
        if map[pos].key == data.key {
            print "la entrada ya existe"
        } else if pos == _size {
            map.push(data)
            _size++
        } else {
            for i = _size - 1 to pos {
                map[i+1] = map[i]
            }
            map[pos] = data
            _size++
        }
    }
}
```

at():

input: *key* es una *string*. La clave del elemento buscado

```
function at(key) {
    pos = busquedaBinaria(key)
    if map[pos].key == key {
        return map[pos].value
    } else {
        print "no existe"
        return -1 //por defecto
    }
}
```

erase():

input: *key* es una *string*

```
function erase(key) {
    pos = busquedaBinaria(key)
    if map[pos].key == key {
        map.erase(pos)
        _size--
    } else {
        print "no existe"
    }
}
```

MapH

Esta segunda implementación es a base de una tabla hash que lidia con las colisiones utilizando *double hashing*. Estas no suelen ser demasiado complejas de implementar.

La eficiencia de tiempo de esta dependerá de que tan buenas sean las funciones hash escogidas, pero de hacerlo adecuadamente es posible obtener buenos tiempos insertando, buscando y eliminando cualquier elemento. Sin embargo, para maximizar la eficiencia de tiempo de una tabla hash, se requiere utilizar un factor de carga lo más bajo posible, lo que aumenta considerablemente el espacio en memoria desperdiciado.

Es por esto que esta solución es mejor utilizada con conjuntos de datos “medianos”.

Las declaraciones e implementaciones de las funciones utilizadas aquí estan en MapH.h y MapH.cpp respectivamente

insert():

input: *data* es un *pair<string, int>* compuesto de su *key* y *value*

```
function insert(data) {
    pos = hash1(data.key) % capacidad //posicion
    desp = hash2(data.key) % k //desplazamiento
    //"capacidad" y "k" son co-primos
    for i = 0 to capacidad {
        if map[pos].isAvailable {
            map[pos] = data
        } else if map[pos].key == data.key {
            print "ya existe este elemento"
            break
        }
        pos = (pos + despl) % capacidad
    }
}
```

at():

input: *key* es una *string*

```
function at(key) {
    pos = hash1(key)
    desp = hash2(key)
    for i = 0 to capacidad{
        if map[pos].key == key {
            return map[pos].value;
        } else if map[pos].neverUsed {
            print “no existe”
            return -1;
        }
        pos = (pos + desp) % capacidad;
    }
return -1
}
```

erase():

input: *key* es una *string*

```
function erase(key) {
    pos = hash1(key)
    desp = hash2(key)
    for i = 0 to capacidad {
        if(map[pos].key == key){
            map[pos].value = 0
            map[pos].available = true
            _size--
            break
        } else if map[pos].neverUsed {
            break
        }
        pos = (pos+desp)%capacidad
    }
}
```

MapAVL

Finalmente, la última implementación de ADTMap es utilizando un AVL tree, o en otras palabras, un árbol de búsqueda auto-balanceado. Este permite realizar de manera muy eficiente operaciones de inserción, búsqueda y eliminación de cualquier elemento, sin sacrificar demasiado espacio en memoria. Esto es, sin embargo, a coste de ser (a nuestro parecer) la estructura más difícil de implementar, debido al delicado manejo de punteros requerido para lograr el auto-balanceo. El equilibrio logrado entre la velocidad de sus operaciones y el espacio ocupado la hace útil para trabajar con grandes conjuntos de datos.

Las declaraciones e implementaciones de las funciones utilizadas aqui estan en MapAVL.h y MapAVL.cpp respectivamente

insert():

input: *data* es un *pair<string, int>* compuesto de su *key* y *value*

```
function insert(data) {
    if this.isEmpty {
        raiz<- new node(data)
    } else {
        nodoAux <- raiz
        while true {
            if nodoAux.key == data.key {
                print "ya existe una entrada con esta clave"
                break
            } else if nodoAux.key < data.key {
                if nodoAux.right == NULL { //podemos agregar a la derecha
                    nodoAux.right <- new node(data)
                    nodoAux.right.parent <- nodoAux
                    _size++
                    //aumenta las alturas de todos los nodos que se vean
afectados
                    //por esta adición, y luego balanceamos de ser necesario
                    actualizarAlturas(nodoAux.right)
                    balancear(nodoAux.right)
                    break
                } else { //seguimos bajando
                    nodoAux <- nodoAux.right
                }
            } else if nodoAux.key > data.key {
                /*
                procedimiento análogo para el nodoAux.left
                */
            }
        }
    }
}
```

at():

input: *key* es una *string*

```
function at(key) {
    if this.isEmpty() {
        print "elemento buscado no existe"
        return -1
    } else {
        nodoAux <- raiz
        while true {
            if nodoAux.key == key {
                return nodoAux.value
            }   else if nodoAux.key < data.key {
                if nodoAux.right == NULL {
                    print "elemento buscado no existe"
                    return -1
                } else { //seguimos bajando
                    nodoAux <- nodoAux.right
                }
            } else if nodoAux.key > data.key {
                /*
                procedimiento análogo para nodoAux.left
                */
            }
        }
    }
}
```

erase():

input: *key* es una *string*

```
function erase(key) {
    if this.isEmpty() {
        //el elemento que se desea borrar no existe
    } else {
        nodoAux <- raiz
        while true { //buscamos el elemento que se desea borrar
            if nodoAux.key < key {
                if nodoAux.right == NULL {
                    //el elemento que se desea borrar no existe
                    return 0
                } else {
                    nodoAux = nodoAux.right
                }
            } else if nodoAux.key > key {
                /* procedimiento análogo para nodoAux.left */
            } else if nodoAux.key == key {
                break
            }
        }
        if nodoAux.left != NULL and nodoAux.right != NULL {
            sucesor = nodoAux.right
            while sucesor.left != NULL {
                sucesor = sucesor.left
            }
            nodoAux.key <- sucesor.key
            nodoAux.value <- sucesor.value
            nodoAux <- sucesor
        }
        if nodoAux.left == NULL and nodoAux.right == NULL {
            actualizarAlturas(nodoHermanoDe(nodoAux))
            balancear(nodoHermanoDe(nodoAux))
        } else {
            if nodoAux.left == NULL {
                if (nodoAux.padre.left == nodoAux) nodoAux.padre.left <-
nodoAux.right;
                else nodoAux.padre.right <- nodoAux.right;
                nodoAux.right.padre <- nodoAux.padre
                actualizarAlturas(nodoAux.right)
                balancear(nodoAux.right)
            } else if nodoAux.right == NULL {
                /* análogo para nodoAux.left */
            }
        }
        delete nodoAux
        _size--
    }
}
```

Análisis teórico

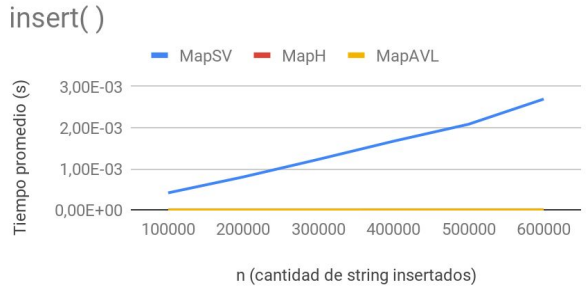
A continuación se presenta el orden de los métodos de las distintas implementaciones:

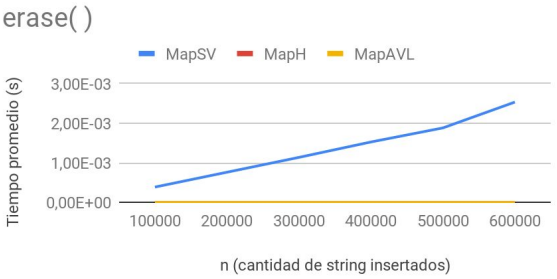
Métodos	MapVS	MapH	MapAVL
Insert	**O(**n+m*log(n)**) amortizado**	**O(**m*n**) amortizado**	**O(**m*(log(n))²**)**
At	**O(**m*log(n)**)**	**O(**m+n**)**	**O(**m*log(n)**)**
Erase	**O(**n+m*log(n)**)**	**O(**m+n**)**	**O(**m*log(n)+(log(n))²**)**

Análisis experimental

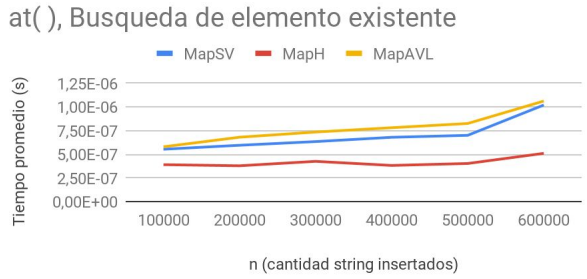
Para obtener los tiempos del análisis experimental se generaron “2n” palabras aleatorias de largo “m”, para mayor equidad y ya que “m” influye en el orden se decidió probar con palabras de largo 10. Al insertar se tuvo en consideración las primeras “n” palabras, al igual que para borrar.

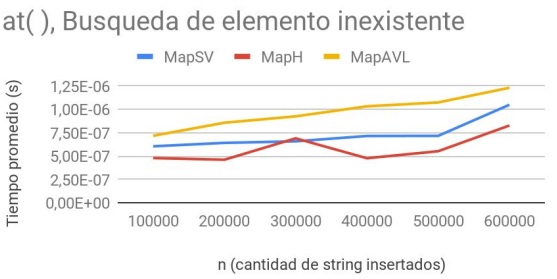
Al tratarse de claves string en vez de números, aumenta la complejidad de nuestros algoritmos, ya que el tiempo que tarda una comparación entre strings depende del largo de las palabras.





Para obtener el análisis de at() dependiendo de si los datos se encontraban o no, se procedió a hacer búsqueda de los primeros “n” elementos para el primer gráfico y búsqueda de las “n” palabras restantes para el segundo.





Como se puede observar, de los gráficos obtenidos, a la hora de insertar y borrar los datos las implementaciones más eficientes son: *MapH* y *MapAVL*, quienes superan con creces el tiempo de *MapSV*. En cuanto a la búsqueda el más lento es *MapAVL* y el más rápido *MapH*, sin embargo este último demora más en el momento de que su arreglo está casi lleno. Todos estos resultados concuerdan muy bien con la teoría.

Conclusión

Si bien los resultados muestran que un mapa basado en tablas hash logra los mejores resultados, es importante recordar que logra esto a costa de un gran desperdicio de espacio. Un mapa basado en un árbol AVL, si bien es ligeramente más lento, hace muy buen uso del espacio pero puede ser difícil de implementar. Por último, aunque es la más lenta de las 3, la implementación basada en vector ordenado es extremadamente sencilla de construir.

Todo esto nos enseña que escoger la solución solo basado en su eficiencia de tiempo no es siempre la mejor opción y que puede ser muy valioso tener en consideración el volumen de datos aproximado con el que se va a trabajar antes de escoger qué estructura utilizar.