



Diccionarios

Mauricio Avilés

Contenido

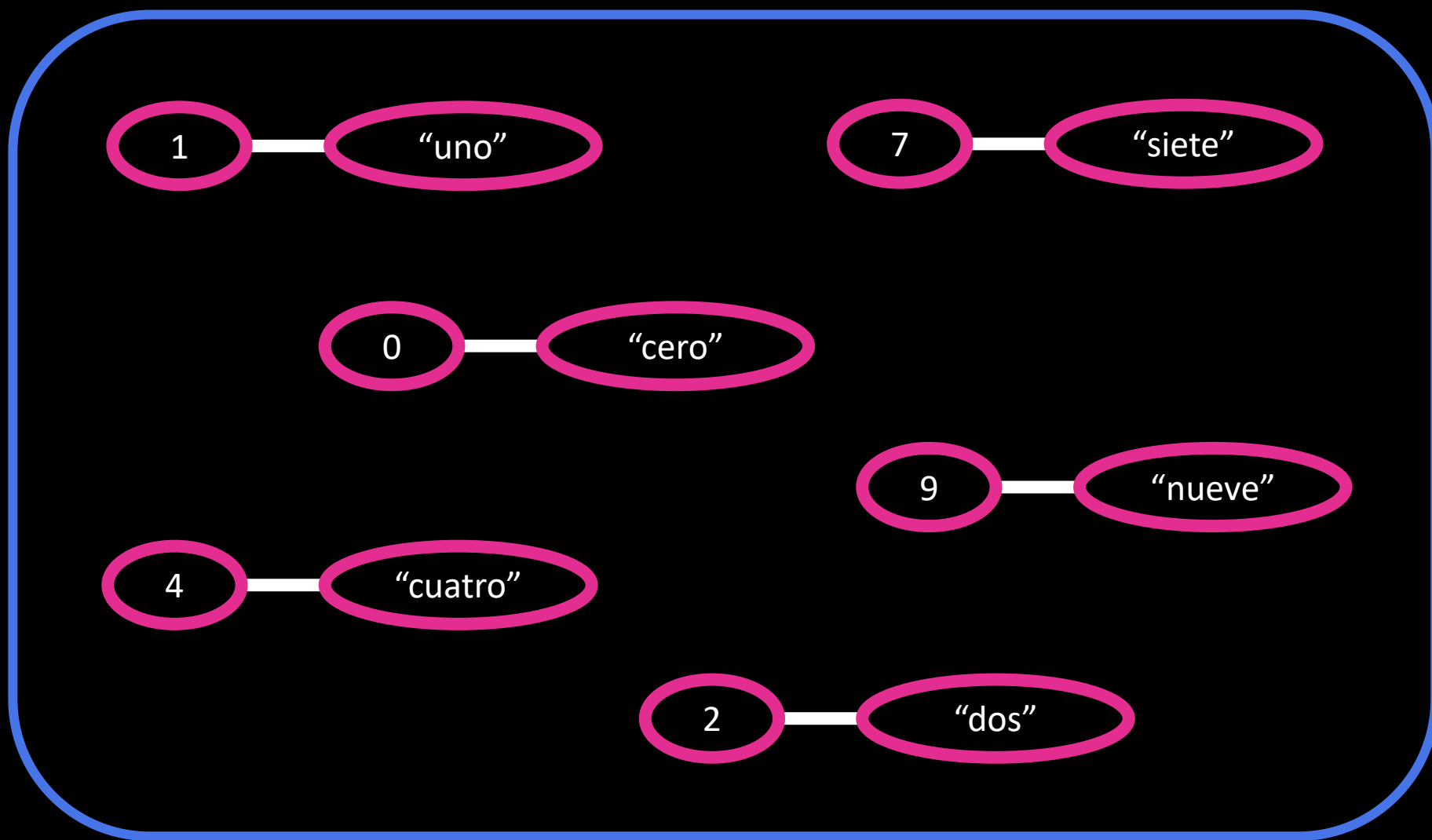
- Definiciones
- Operaciones
- Clase abstracta Dictionary
- Clase KeyValuePair
- Implementación de UnsortedArrayDictionary
- Implementación de SortedArrayDictionary

Lecturas

- Sección 4.4
 - Shaffer, C. A. "Data Structures & Algorithm Analysis in C++" (3rd ed., Dover). Mineola, NY. 2011.
- Sección 9.5
 - Goodrich, M. T., Tamassia, R., & Mount, D. M. "Data structures and algorithms in C++" (2nd ed., Wiley). Hoboken, NJ: Wiley. 2011.

Diccionarios

- Colección de registros de datos organizados para **acceso** y **almacenamiento rápido**
- Está conformado por **pares** que constan de una **llave** y un **valor** asociado.
- Un diccionario tiene **operaciones** para almacenar, encontrar y eliminar registros
- El tipo de las **llaves** debe ser **comparable**
- El tipo de los **valores** puede ser **cualquiera**
- Una llave permite **identificar** de manera única un par dentro de la colección
- Cada llave aparece sólo una vez en la colección
- Los elementos pueden estar **ordenados** de alguna manera para optimizar la búsqueda dentro de la estructura



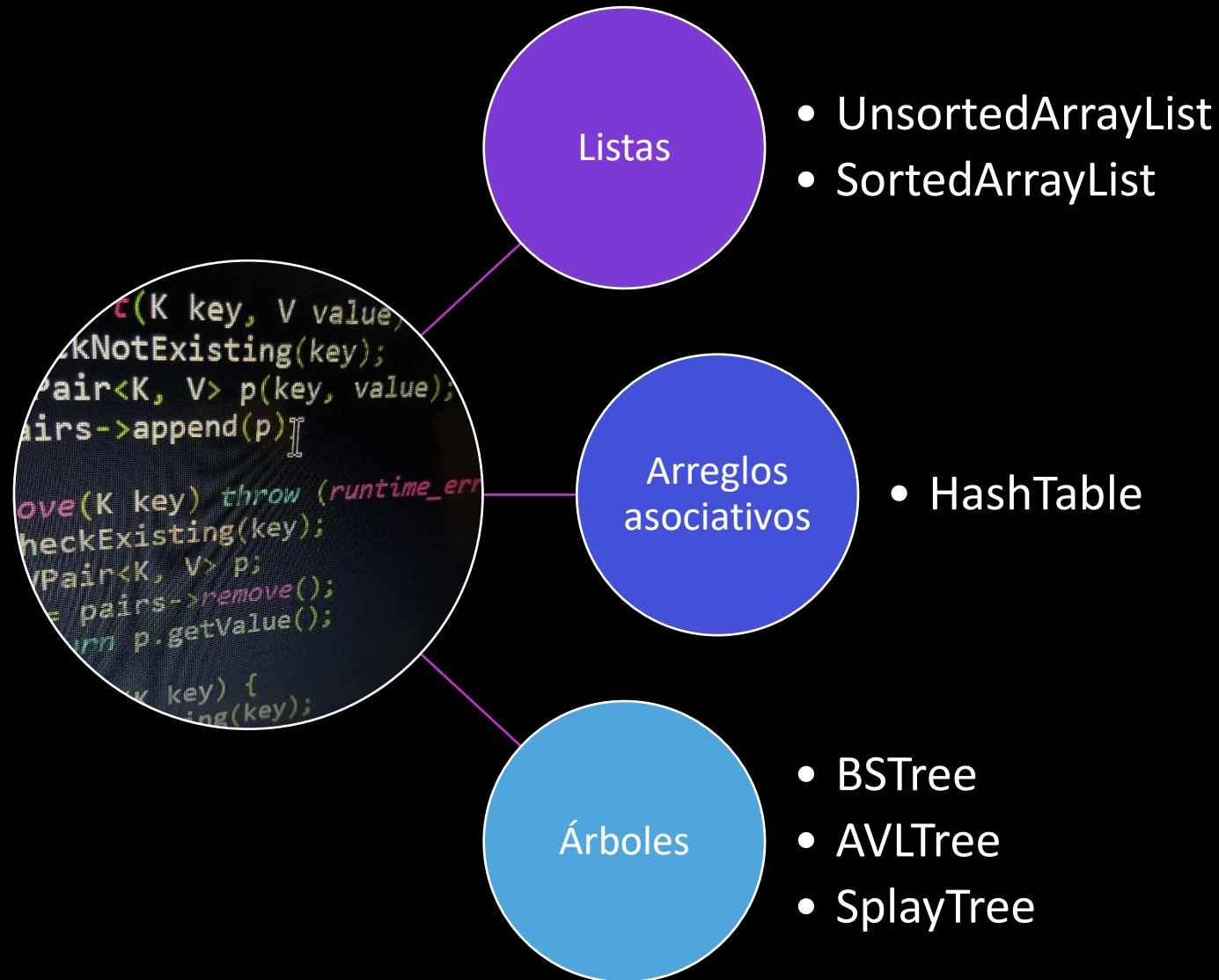
Un diccionario almacena pares (llave, valor). Los pares no tienen ningún orden específico. Cada llave tiene que ser única, no puede repetirse dentro de la estructura. Los valores sí pueden repetirse si se asocian a diferentes llaves.

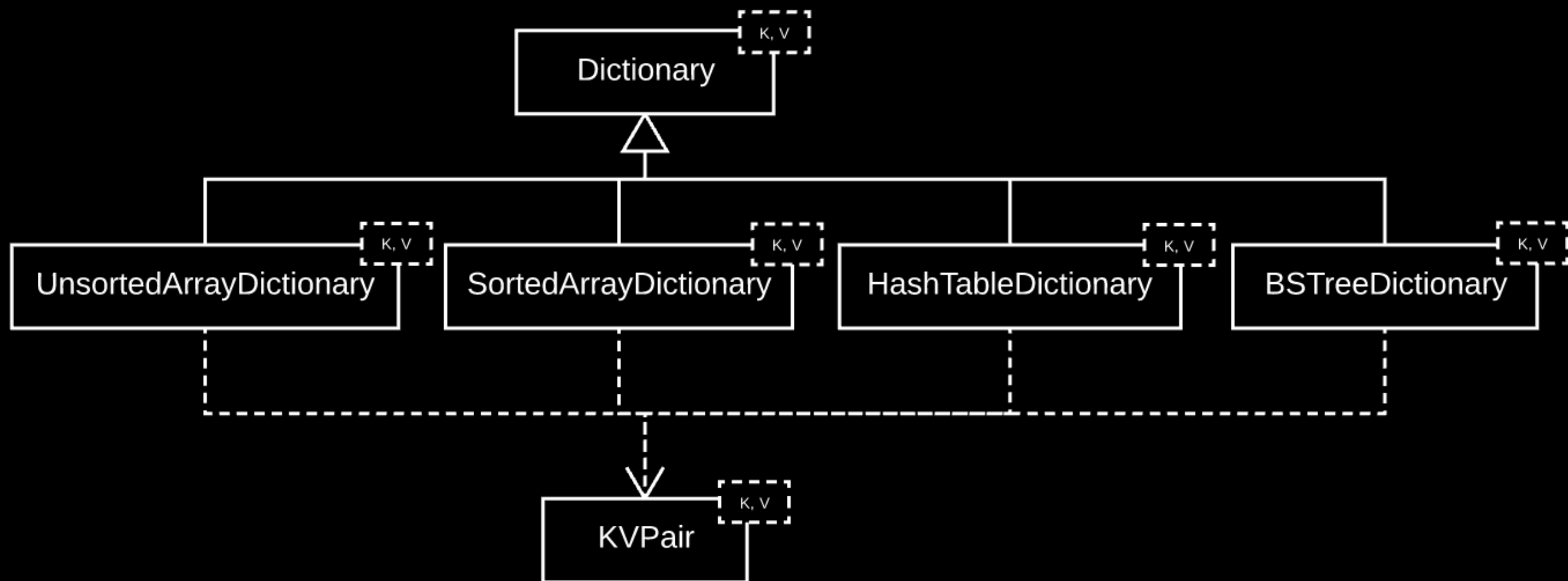
<u>Operación</u>	<u>Descripción</u>
insert	Inserta un nuevo par llave valor en la estructura.
remove	Recibe una llave existente y elimina el par llave valor de la estructura. Retorna el valor del par eliminado.
getValue	Recibe una llave existente y retorna el valor asociado a la misma.
setValue	Recibe una llave existente y un valor y actualiza el valor asociado a la llave.
contains	Recibe una llave y dice si se encuentra dentro de la estructura.
getKeys	Retorna una lista con todas las llaves de la estructura.
getValues	Retorna una lista con todos los valores de la estructura.
getSize	Dice la cantidad de pares almacenados en la estructura

```
template <typename K, typename V>
class Dictionary {
public:
    Dictionary() {}
    virtual ~Dictionary() {}
    virtual void insert(K key, V value) = 0;
    virtual V remove(K key) = 0;
    virtual V getValue(K key) = 0;
    virtual void setValue(K key, V value) = 0;
    virtual bool contains(K key) = 0;
    virtual List<K>* getKeys() = 0;
    virtual List<V>* getValues() = 0;
    virtual int getSize() = 0;
};
```

Esta clase utiliza dos plantillas para determinar el tipo de dato de la llave y el tipo de dato de los valores. El tipo de dato utilizado en la llave debe ser comparable.

Implementaciones





Clase KeyValuePair

- La estructura diccionario almacena **pares llave-valor**
- Es necesario crear una **clase** que permita agrupar estos elementos
- Esta clase se utiliza como **elemento** para almacenar en otras estructuras
- Sus **atributos** guardan un elemento llave y un elemento valor
- Se necesitan **métodos de acceso** a los atributos
- Es útil implementar los **operadores de comparación** (==, !=, <, <=, >, >=) y la **asignación** (=) para que estas operaciones funcionen cuando se utilicen sobre el par en las estructuras que ya se implementaron

```
template <typename K, typename V>
class KVPair {
private:
    K key;
    V value;
public:
    KVPair() {}
    KVPair(K key) {
        this->key = key;
    }
    KVPair(K key, V value) {
        this->key = key;
        this->value = value;
    }
}
```

```
template <typename K, typename V>
class KVPair {
private:
    K key;
    V value;
public:
    KVPair() {}
    KVPair(K key) {
        this->key = key;
    }
    KVPair(K key, V value) {
        this->key = key;
        this->value = value;
    }
}
```

También se utiliza una plantilla para el tipo de la llave y el valor almacenado en el par.

```
template <typename K, typename V>
class KVPair {
private:
    K key;
    V value;
public:
    KVPair() {}
    KVPair(K key) {
        this->key = key;
    }
    KVPair(K key, V value) {
        this->key = key;
        this->value = value;
    }
}
```

Tres constructores diferentes se utilizan en diferentes casos. El constructor vacío se utiliza cuando no se desea inicializar los atributos. El siguiente constructor sólo inicializa el valor de la llave. El último inicializa llave y valor.

```
void operator= (const KVPair &other) {  
    key = other.key;  
    value = other.value;  
}  
bool operator== (const KVPair &other) {  
    return key == other.key;  
}  
bool operator!= (const KVPair &other) {  
    return key != other.key;  
}  
bool operator< (const KVPair &other) {  
    return key < other.key;  
}  
bool operator<= (const KVPair &other) {  
    return key <= other.key;  
}
```


```
void operator= (const KVPair &other) {  
    key = other.key;  
    value = other.value;  
}  
bool operator== (const KVPair &other) {  
    return key == other.key;  
}  
bool operator!= (const KVPair &other) {  
    return key != other.key;  
}  
bool operator< (const KVPair &other) {  
    return key < other.key;  
}  
bool operator<= (const KVPair &other) {  
    return key <= other.key;  
}
```

Método que implementa el operador de asignación. Cuando se asigne a una variable tipo KVPair el valor de otra variable del mismo tipo, se va a ejecutar de esta forma. La llave de la otra variable se asigna a la variable actual y el valor también.

```
void operator= (const KVPair &other) {  
    key = other.key;  
    value = other.value;  
}  
bool operator== (const KVPair &other) {  
    return key == other.key;  
}  
bool operator!= (const KVPair &other) {  
    return key != other.key;  
}  
bool operator< (const KVPair &other) {  
    return key < other.key;  
}  
bool operator<= (const KVPair &other) {  
    return key <= other.key;  
}
```

Los siguientes métodos implementan los operadores de comparación, por esto retornan un valor booleano. Todos se implementan comparando las llaves. Lo que está almacenado en el valor de los pares no es relevante para la comparación.

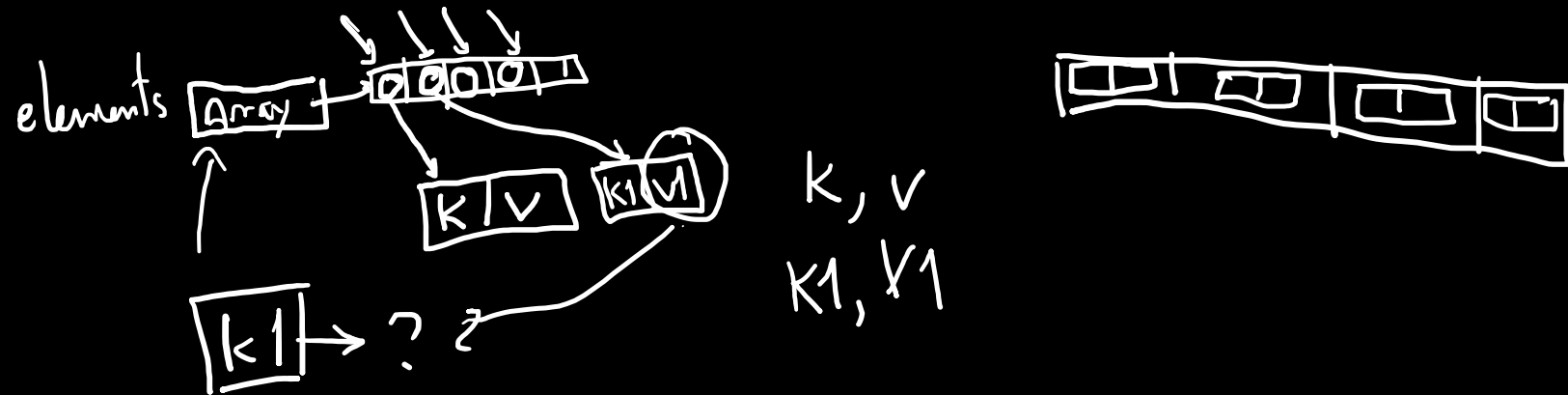

```
bool operator> (const KVPair &other) {  
    return key > other.key;  
}  
bool operator>= (const KVPair &other) {  
    return key >= other.key;  
}  
void setKey(K key) {  
    this->key = key;  
}  
K getKey() {  
    return key;  
}  
void setValue(V value) {  
    this->value = value;  
}  
V getValue() {  
    return value;  
}  
};
```



Métodos getters y setters para los dos atributos de la clase.

UnsortedArrayDictionary

- Implementación de la clase abstracta **Dictionary**
- Utiliza un **ArrayList** con elementos de tipo **KVPair** para almacenar los pares
- El ArrayList es un **atributo** de esta clase
- No hay **ningún tipo de orden** en los elementos que se guardan en el ArrayList



```
#include "Dictionary.h"
#include "KeyValuePair.h"
#include "DLinkedList.h"

template <typename K, typename V>
class UnsortedArrayDictionary : public Dictionary<K, V> {
private:
    ArrayList<KeyValuePair<K, V> > *pairs;

    void checkNotExisting(K key) {
        if (contains(key)) {
            throw runtime_error("Duplicated key.");
        }
    }

    void checkExisting(K key) {
        if (!contains(key)) {
            throw runtime_error("Key not found.");
        }
    }
}
```

```
#include "Dictionary.h"
#include "KeyValuePair.h"
#include "DLinkedList.h"
```

Se incluye la clase abstracta que se va a implementar, la clase que representa los pares llave-valor y la clase lista que se retornará en algunos métodos.

```
template <typename K, typename V>
class UnsortedArrayDictionary : public Dictionary<K, V> {
private:
    ArrayList<KeyValuePair<K, V> > *pairs;

    void checkNotExisting(K key) {
        if (contains(key)) {
            throw runtime_error("Duplicated key.");
        }
    }

    void checkExisting(K key) {
        if (!contains(key)) {
            throw runtime_error("Key not found.");
        }
    }
}
```

```
#include "Dictionary.h"
#include "KeyValuePair.h"
#include "DLinkedList.h"
```

```
template <typename K, typename V>
class UnsortedArrayDictionary : public Dictionary<K, V> {
private:
    ArrayList<KeyValuePair<K, V> > *pairs;

    void checkNotExisting(K key) {
        if (contains(key)) {
            throw runtime_error("Duplicated key.");
        }
    }

    void checkExisting(K key) {
        if (!contains(key)) {
            throw runtime_error("Key not found.");
        }
    }
}
```

Puntero a la lista de elementos que mantiene los pares. El espacio en blanco que se ubica entre los dos símbolos ">" es necesario para que C++ no los interprete como el operador de entrada >>. Este atributo también podría estar en memoria estática.

```
#include "Dictionary.h"
#include "KeyValuePair.h"
#include "DLinkedList.h"
```

```
template <typename K, typename V>
class UnsortedArrayDictionary : public Dictionary<K, V> {
private:
    ArrayList<KeyValuePair<K, V> > *pairs;

    void checkNotExisting(K key) {
        if (contains(key)) {
            throw runtime_error("Duplicated key.");
        }
    }

    void checkExisting(K key) {
        if (!contains(key)) {
            throw runtime_error("Key not found.");
        }
    }
}
```

Método auxiliar que revisa si una llave ya se encuentra en el diccionario. Esta es una restricción en varios de los métodos a implementar.

```
#include "Dictionary.h"
#include "KeyValuePair.h"
#include "DLinkedList.h"

template <typename K, typename V>
class UnsortedArrayDictionary : public Dictionary<K, V> {
private:
    ArrayList<KeyValuePair<K, V> > *pairs;

    void checkNotExisting(K key) {
        if (contains(key)) {
            throw runtime_error("Duplicated key.");
        }
    }
    void checkExisting(K key) {
        if (!contains(key)) {
            throw runtime_error("Key not found.");
        }
    }
}
```

Método auxiliar que revisa si una llave no se encuentra en el diccionario. También es una restricción de varios métodos de la clase.

El constructor crea en memoria dinámica la lista de elementos donde se van a almacenar los pares.

```
public:
    UnsortedArrayDictionary(int maxSize = DEFAULT_MAX_SIZE) {
        pairs = new ArrayList<KeyValuePair<K, V> >(maxSize);
    };
    ~UnsortedArrayDictionary() {
        delete pairs;
    };
};
```


public:

```
UnsortedArrayDictionary(int maxSize = DEFAULT_MAX_SIZE) {  
    pairs = new ArrayList<KeyValuePair<K, V> >(maxSize);  
};  
~UnsortedArrayDictionary() {  
    delete pairs;  
};
```

El destructor retorna la memoria utilizada por la lista creada en memoria dinámica.

Se verifica que la llave a insertar no exista. Si ya existe el método `checkNotExisting` levanta una excepción.

```
void insert(K key, V value) throw (runtime_error) {  
    checkNotExisting(key);  
    KVPair<K, V> p(key, value);  
    pairs->append(p);  
}  
V remove(K key) throw (runtime_error) {  
    checkExisting(key);  
    KVPair<K, V> p;  
    p = pairs->remove();  
    return p.getValue();  
}
```

```
void insert(K key, V value) throw (runtime_error) {  
    checkNotExisting(key);  
    KVPair<K, V> p(key, value);  
    pairs->append(p);  
}  
V remove(K key) throw (runtime_error) {  
    checkExisting(key);  
    KVPair<K, V> p;  
    p = pairs->remove();  
    return p.getValue();  
}
```

Se crea un nuevo par llave-valor y se agrega a la lista de pares.

```
void insert(K key, V value) throw (runtime_error) {  
    checkNotExisting(key);  
    KeyValuePair<K, V> p(key, value);  
    pairs->append(p);  
}  
V remove(K key) throw (runtime_error) {  
    checkExisting(key);  
    KeyValuePair<K, V> p;  
    p = pairs->remove();  
    return p.getValue();  
}
```

Se verifica que la llave que se va a borrar exista en el diccionario. El método `checkExisting` utiliza al método `contains`, el cual recorre la lista secuencialmente desde el inicio para buscar un elemento. Si el elemento se encuentra, la posición actual de la lista queda apuntando al elemento encontrado. Es por esto que es posible llamar al método `getElement` para obtener el elemento que recién se buscó.

Esta característica se aprovecha en los métodos `getValue` y `setValue`, también.

```
void insert(K key, V value) throw (runtime_error) {  
    checkNotExisting(key);  
    KeyValuePair<K, V> p(key, value);  
    pairs->append(p);  
}  
V remove(K key) throw (runtime_error) {  
    checkExisting(key);  
    KeyValuePair<K, V> p;  
    p = pairs->remove();  
    return p.getValue();  
}
```

Se crea un par vacío donde se almacena el que se elimina de la lista y se retorna como resultado.

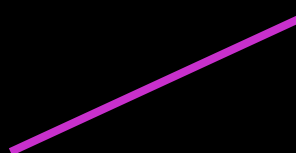
```
V getValue(K key) {  
    checkExisting(key);  
    KVPair<K, V> p = pairs->getElement();  
    return p.getValue();  
}  
void setValue(K key, V value) {  
    checkExisting(key);  
    KVPair<K, V> p(key, value);  
    pairs->remove();  
    pairs->insert(p);  
}  
bool contains(K key) {  
    KVPair<K, V> p(key);  
    return pairs->contains(p);  
}
```

Se verifica que la llave exista, se almacena en una variable temporal y se retorna el valor asociado.

```
V getValue(K key) {  
    checkExisting(key);  
    KVPair<K, V> p = pairs->getElement();  
    return p.getValue();  
}  
void setValue(K key, V value) {  
    checkExisting(key);  
    KVPair<K, V> p(key, value);  
    pairs->remove();  
    pairs->insert(p);  
}  
bool contains(K key) {  
    KVPair<K, V> p(key);  
    return pairs->contains(p);  
}
```

Funciona de manera muy similar al método remove, pero en lugar de retornar el valor, se vuelve a insertar la llave con el nuevo valor. Esto se hace por comodidad ya que es más fácil que actualizar el valor dentro del arreglo de pares.

```
V getValue(K key) {  
    checkExisting(key);  
    KeyValuePair<K, V> p = pairs->getElement();  
    return p.getValue();  
}  
void setValue(K key, V value) {  
    checkExisting(key);  
    KeyValuePair<K, V> p(key, value);  
    pairs->remove();  
    pairs->insert(p);  
}  
bool contains(K key) {  
    KeyValuePair<K, V> p(key);  
    return pairs->contains(p);  
}
```



Crea un par con la llave enviada y utiliza el método de la clase ArrayList para determinar si se encuentra dentro de la lista.


```
List<K>* getKeys() {
    List<K>* keys = new DLinkedList<K>();
    for (pairs->goToStart(); !pairs->atEnd(); pairs->next()) {
        KVPair<K, V> p = pairs->getElement();
        keys->append(p.getKey());
    }
    return keys;
}

List<V>* getValues() {
    List<V>* values = new DLinkedList<V>();
    for (pairs->goToStart(); !pairs->atEnd(); pairs->next()) {
        KVPair<K, V> p = pairs->getElement();
        values->append(p.getValue());
    }
    return values;
}

int getSize() {
    return pairs->getSize();
}

};
```

Los métodos getKeys y getValues son muy similares. En ambos se crea una DLinkedList en memoria dinámica, y se llena con los elementos del arreglo. Finalmente se retorna la dirección de dicha lista. Es responsabilidad del que invoca el método liberar la memoria reservada para la lista.

```
List<K>* getKeys() {
    List<K>* keys = new DLinkedList<K>();
    for (pairs->goToStart(); !pairs->atEnd(); pairs->next()) {
        KVPair<K, V> p = pairs->getElement();
        keys->append(p.getKey());
    }
    return keys;
}

List<V>* getValues() {
    List<V>* values = new DLinkedList<V>();
    for (pairs->goToStart(); !pairs->atEnd(); pairs->next()) {
        KVPair<K, V> p = pairs->getElement();
        values->append(p.getValue());
    }
    return values;
}

int getSize() {
    return pairs->getSize();
}

};
```

Los métodos getKeys y getValues son muy similares. En ambos se crea una DLinkedList en memoria dinámica, y se llena con los elementos del arreglo. Finalmente se retorna la dirección de dicha lista. Es responsabilidad del que invoca el método liberar la memoria reservada para la lista.

Ejemplo de utilización

```
UnsortedArrayDictionary<int, string> dict;
dict.insert(3, "tres");
dict.insert(1, "uno");
dict.insert(2, "dos");
dict.insert(5, "cinco");
dict.insert(4, "cuatro");
printKeys(dict);
printValues(dict);
dict.remove(2);
try {
    dict.remove(0);
}
catch (runtime_error e) {
    cout << "ERROR: " << e.what() << endl;
}
cout << dict.getValue(1) << endl;
cout << dict.getValue(5) << endl;
```

```
try {
    dict.getValue(2);
}
catch (runtime_error e) {
    cout << "ERROR: " << e.what() << endl;
}
dict.setValue(1, "UNO");
dict.setValue(3, "TRES");
try {
    dict.setValue(2, "DOS");
}
catch (runtime_error e) {
    cout << "ERROR: " << e.what() << endl;
}
printKeys(dict);
printValues(dict);
```

```
3
1
2
5
4
tres
uno
dos
cinco
cuatro
ERROR: Key not found.
uno
cinco
ERROR: Key not found.
ERROR: Key not found.
3
1
5
4
TRES
UNO
cinco
cuatro
SortedArrayList: Insertando números aleatorios.
SortedArrayList: Imprimiendo valores.
```

```
302
653
952
6198
6702
9464
9905
9999
9999
10763
14106
18460
18517
19541
19733
19897
23144
24982
28699
30599
31147
31274
SortedArrayList::indexOf(9999): 7
SortedArrayList::contains(9999): 1
SortedArrayList::contains(9998): 0
```

```
Process returned 0 (0x0)   execution time : 0.117 s
Press any key to continue.
```

SortedArrayDictionary

- Implementación de la clase abstracta **Dictionary**
- Utiliza un **SortedArrayList** con elementos de tipo **KVPair** para almacenar los pares
- El SortedArrayList de KVPair es un **atributo** de esta clase
- Esta clase utiliza **comparaciones** entre sus elementos para mantenerlos ordenados, por eso es importante que KVPair implemente los operadores de comparación
- También aprovecha el **orden** de los elementos para hacer búsquedas eficientes, por lo que esta implementación del diccionario resulta más **eficiente** que UnsortedArrayDictionary

```
#include "Dictionary.h"
#include "KeyValuePair.h"
#include "DLinkedList.h"
#include "SortedArrayList.h"

template <typename K, typename V>
class SortedArrayDictionary : public Dictionary<K, V> {
private:
    SortedArrayList<KeyValuePair<K, V> > *pairs;
```

Esta implementación es básicamente igual a la implementación del UnsortedArrayDictionary, la diferencia está en el tipo de lista que se utiliza para almacenar los pares, que es SortedArrayList en lugar de un ArrayList.

```
void search(K key) {  
    KeyValuePair<K, V> p(key);  
    pairs->goToPos(pairs->indexOf(p));  
}  
void checkNotExisting(K key) {  
    if (contains(key)) {  
        throw runtime_error("Duplicated key.");  
    }  
}  
void checkExisting(K key) {  
    if (!contains(key)) {  
        throw runtime_error("Key not found.");  
    }  
}
```

public:

```
SortedArrayDictionary(int maxSize = DEFAULT_MAX_SIZE) {  
    pairs = new SortedArrayList<KeyValuePair<K, V> >(maxSize);  
}  
~SortedArrayDictionary() {  
    delete pairs;  
}
```



```
void insert(K key, V value) {  
    checkNotExisting(key);  
    KeyValuePair<K, V> p(key, value);  
    pairs->insert(p);  
}  
V remove(K key) {  
    checkExisting(key);  
    search(key);  
    KeyValuePair<K, V> p = pairs->remove();  
    return p.getValue();  
}
```

```
V getValue(K key) {  
    checkExisting(key);  
    search(key);  
    KVPair<K, V> p = pairs->getElement();  
    return p.getValue();  
}  
void setValue(K key, V value) {  
    checkExisting(key);  
    search(key);  
    KVPair<K, V> p = pairs->remove();  
    p.setValue(value);  
    pairs->insert(p);  
}  
bool contains(K key) {  
    KVPair<K, V> p(key);  
    return pairs->contains(p);  
}
```

```
List<K>* getKeys() {
    List<K>* keys = new DLinkedList<K>();
    for (pairs->goToStart(); !pairs->atEnd(); pairs->next()) {
        KVPair<K, V> p = pairs->getElement();
        keys->append(p.getKey());
    }
    return keys;
}

List<V>* getValues() {
    List<V>* values = new DLinkedList<V>();
    for (pairs->goToStart(); !pairs->atEnd(); pairs->next()) {
        KVPair<K, V> p = pairs->getElement();
        values->append(p.getValue());
    }
    return values;
}

int getSize() {
    return pairs->getSize();
}

};
```



Diccionarios

Mauricio Avilés