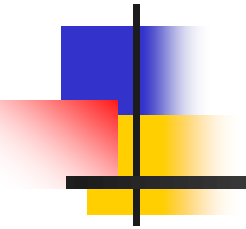


Tema 4
Diccionarios

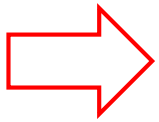
Estructura de datos y algoritmos
Facultad de Informática - Universidad Complutense de Madrid

Transparencias de los Profs.:
Mercedes Gómez Albarrán y José Luis Sierra Rodríguez

Motivación

- Deseamos implementar un programa que analice un texto y que imprima las palabras que aparecen en el texto junto con sus frecuencias de aparición (es decir, el número de veces que cada palabra aparece en el texto). Las palabras deben ordenarse de menor a mayor.

```
El universo (que otros
llaman la Biblioteca)
se componte de un
número indefinido, y
tal vez infinito, de
galerías hexagonales,
con vastos pozos de
ventilación en el
medio, cercados por
barandas bajísimas.
```



```
bajísimas: 1
barandas: 1
biblioteca: 1
cercados: 1
compone: 1
con: 1
de: 3
el: 2
en: 1
galerías: 1
hexagonales: 1
indefinido: 1
infinito: 1
```

```
la: 1
llaman: 1
medio: 1
número: 1
otros: 1
por: 1
pozos: 1
que: 1
se: 1
tal: 1
un: 1
universo: 1
vastos: 1
ventilación: 1
vez: 1
y: 1
```

Introducción

- Los diccionarios son colecciones de pares (*clave*, *valor*), que permiten acceder a los valores a partir de las claves.
- Pueden verse como *extensiones* de los *arrays* convencionales, en las cuáles los valores índice pueden ser claves arbitrarias.
- Como TAD, los diccionarios están parametrizados por dos tipos diferentes:
  - El tipo de las *claves*
  - El tipo de los *valores*
- Examinaremos tres implementaciones alternativas de los diccionarios
  - Implementación basada en **arrays dinámicos**
  - Implementación basada en **árboles de búsqueda**
  - Implementación basada en **tablas dispersas** (***hash tables***)

Operaciones básicas del TAD *Diccionario*

- `diccionarioVacio`: Constructora que genera un diccionario que no tiene ningún par *clave – valor*
- `inserta(`*C*`,` *V*`)`: Mutadora que inserta un par *clave–valor* **(C,V)** en el diccionario. Si el diccionario ya contiene un valor para la clave **C** (es decir, ya contiene un par **(C,V')**), entonces la operación actualiza el valor a **V** (es decir, el viejo valor se pierde, y se cambia por el nuevo valor **V**)
- `borra(`*C*`)`: Mutadora que elimina la entrada para **C** en el diccionario (en caso de que tal entrada exista; si no existe, no tiene efecto alguno)
- `contiene(`*C*`)`: Observadora que devuelve **true** si el diccionario contiene un valor asociado a la clave **C**, o **false** en otro caso
- `valorPara(`*C*`)`: Observadora (parcial) que devuelve el valor asociado a *C* en el diccionario, en caso de que tal valor exista
- `esVacio`: Observadora que devuelve **true** si el diccionario no tiene entradas, y **false** en otro caso

Implementación con estructuras de datos lineales

- Un diccionario puede verse como un conjunto de pares **(C,V)**
- Por tanto, pueden usarse las técnicas vistas en el tema 1 para implementar conjuntos (posiblemente extendidas con las técnicas para el manejo de listas del capítulo 2) para implementar el TAD
- En concreto, el TAD puede implementarse mediante un array dinámico, almacenando, en cada posición, un par **(C,V)**
- Dicho array puede estar desordenado, o bien mantenerse ordenado (en este último caso las complejidades de algunas operaciones mejoran)

Implementación con estructuras de datos lineales: Complejidad de las operaciones

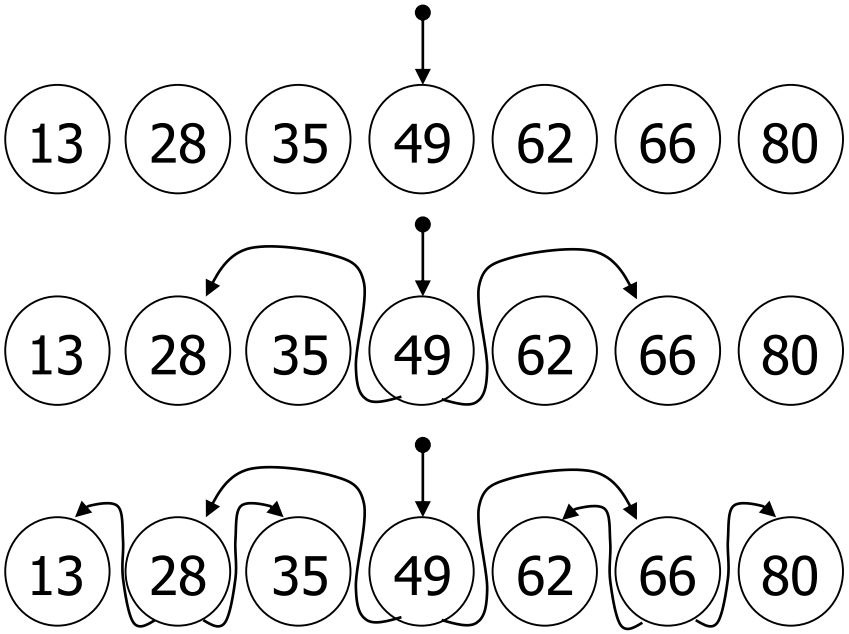
	Array desordenado	Array ordenado
diccionarioVacio	O(1)	O(1)
inserta	O(n)	O(n)
borra	O(n)	O(n)
contiene	O(n)	O(log n)
valorPara	O(n)	O(log n)
esVacio	O(1)	O(1)

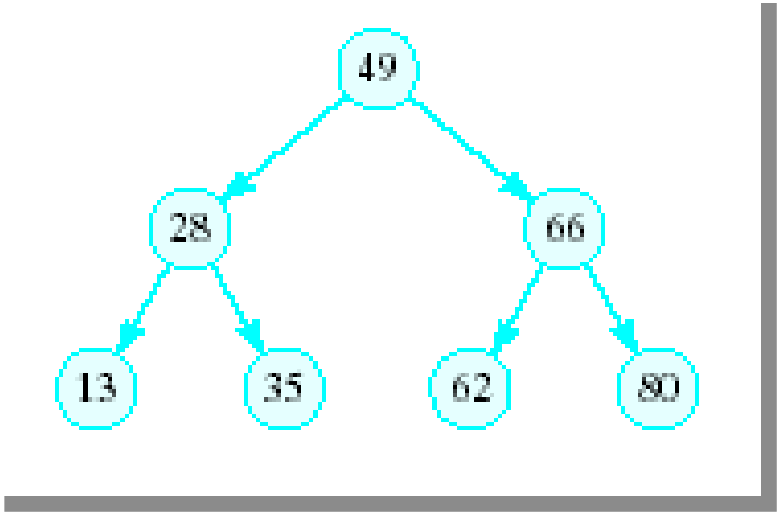
¿Mejorarían los costes si se usara una lista enlazada de pares **(C,V)**?

Implementación con estructuras de datos arbóreas: árboles de búsqueda

- ¿Podemos
  - aprovechar las ventajas del uso de búsqueda binaria (complejidad logarítmica), lo que requiere una estructura (lineal) ordenada …
  - eliminando las desventajas de las estructuras lineales ordenadas "rígidas" a la hora de insertar/borrar

  ?

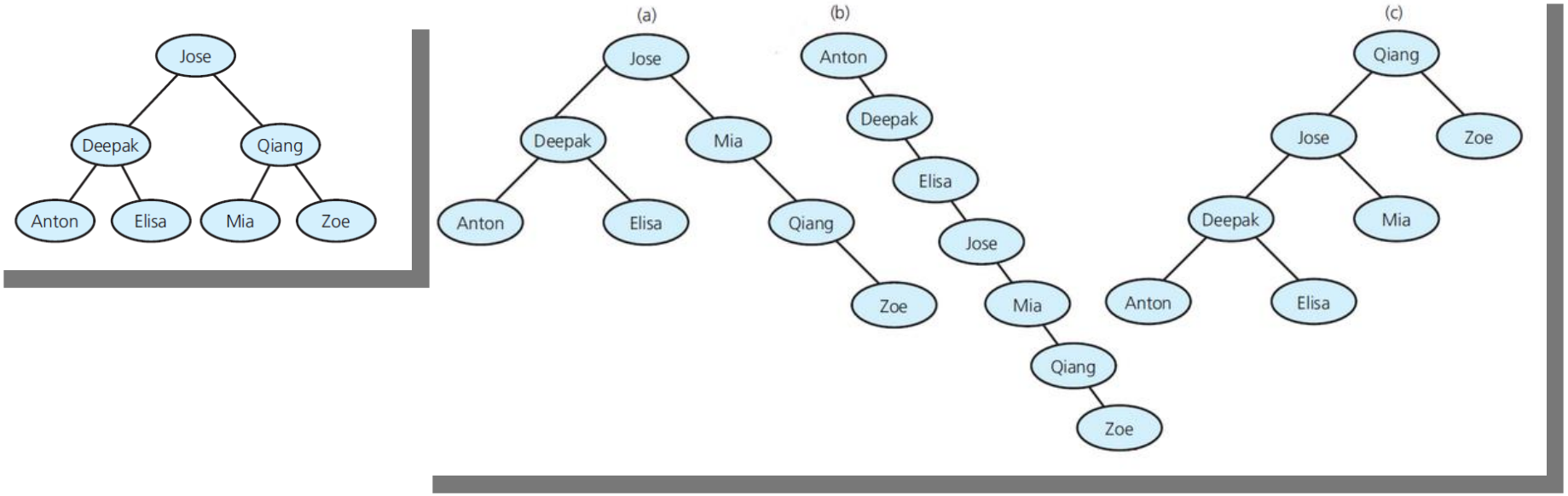




Figuras extraídas de: Nyhoff, ADTs, Data Structures and Problem Solving with C++, 2005 Pearson Education, Inc.

Implementación con estructuras de datos arbóreas: árboles de búsqueda

- A primera vista, esa estructura jerárquica de nodos (árbol binario) no es muy diferente de los que hemos visto en el capítulo anterior; sin embargo, tiene una propiedad que lo hace excepcionalmente diferente y que comparte con estos otros



Todos son árboles de búsqueda…

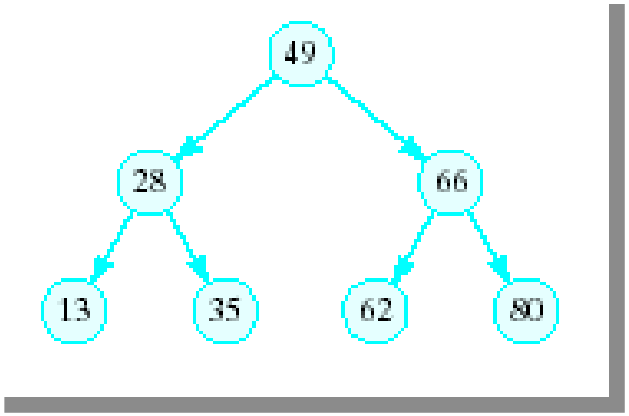
Figuras extraídas de: Carrano, Data Abstraction & Problem Solving with C++, 2013 Pearson Education, Inc.

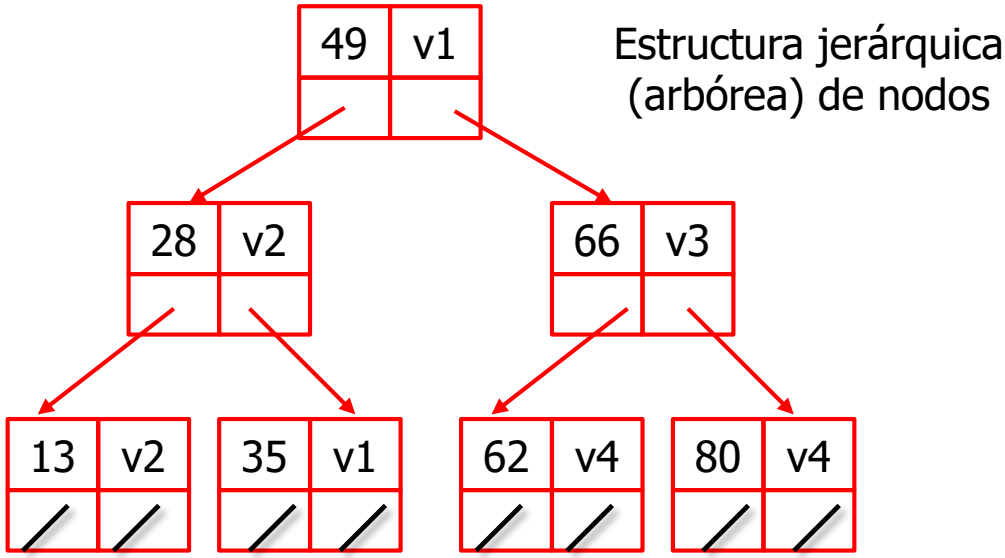
Implementación con estructuras de datos arbóreas: árboles de búsqueda

- Un **árbol de búsqueda** es un árbol binario, cuyo tipo base está ordenado, y que, bien es vacío, bien cumple las siguientes condiciones:
  - El valor de la raíz es **mayor** que todos los valores en el **hijo izquierdo**, y **menor** que todos los valores en el **hijo derecho**
  - Tanto el hijo izquierdo como el hijo derecho son, a su vez, **árboles de búsqueda**
- La **búsqueda**, **inserción** y **borrado** en un árbol de búsqueda que está **balanceado**, puede llevarse a cabo en tiempo logarítmico (lo que mejora la implementación basada en arrays)
  - Un **árbol balanceado** es aquel en el que la diferencia de talla entre hijo izquierdo y derecho es, a lo sumo 1, y en el que dichos hijos son, a su vez, árboles balanceados

Implementación con estructuras de datos arbóreas: árboles de búsqueda

- Cada nodo tendrá:
  - Un par *clave-valor*
  - Un puntero al nodo raíz del subárbol izquierdo
  - Otro puntero al nodo raíz del subárbol derecho
- Los pares *clave-valor* se ordenan por clave (por tanto, el tipo de las claves debe aceptar un orden total)

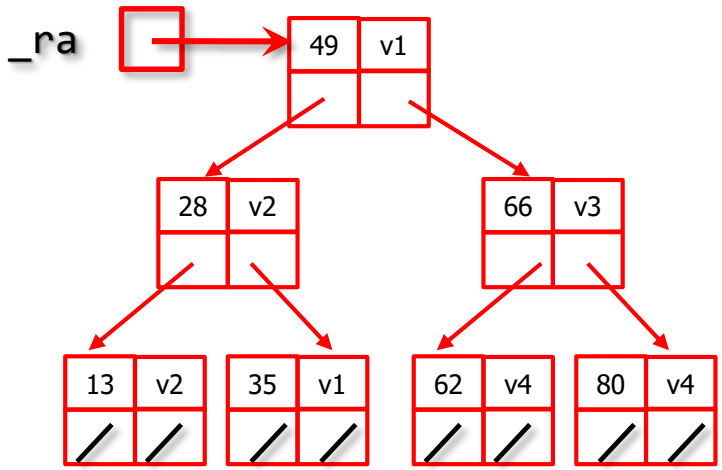




Implementación con estructuras de datos arbóreas: árboles de búsqueda

- **Tipos representantes:** Un puntero a nodo que contiene la raíz del árbol. Estructura jerárquica de nodos

```
template <class tClave, class tValor>
class Diccionario {
    private:
          class Nodo {
          public:
          Nodo() : _iz(NULL), _dr(NULL) {}
          Nodo(const tClave &clave, const tValor &valor)
                    : _clave(clave), _valor(valor), _iz(NULL), _dr(NULL) {}
          Nodo(Nodo *iz, const tClave &clave, const tValor &valor, Nodo *dr)
                    : _clave(clave), _valor(valor), _iz(iz), _dr(dr) {}

          tClave _clave;
          tValor _valor;
          Nodo *_iz;
          Nodo *_dr;
          };
          Nodo *_ra;
     public:
          ...
    };
```



Implementación con estructuras de datos arbóreas: árboles de búsqueda

- Algunas operaciones para el manejo de la estructura de datos arbórea: ¿Cómo se…
  - Genera una copia de una estructura?
  - Libera la memoria ocupada por una estructura?
  - Busca una clave en la estructura?
  - Inserta un par clave-valor?
  - Borra un par clave-valor?

Implementación con estructuras de datos arbóreas: árboles de búsqueda

- Algunas operaciones para el manejo de la estructura de datos arbórea: ¿Cómo se…
  - Genera una copia de una estructura?

```
static Nodo *copia(Nodo *ra) {
  if (ra == NULL)
    return NULL;
  else
    return new Nodo(copia(ra->_iz),
                    ra->_clave,
                    ra->_valor,
                    copia(ra->_dr));
}
```

Implementación con estructuras de datos arbóreas: árboles de búsqueda

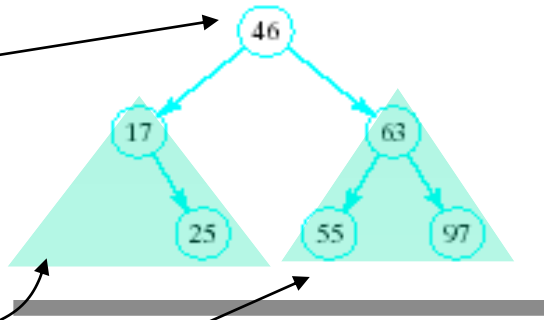
- Algunas operaciones para el manejo de la estructura de datos arbórea: ¿Cómo se…
  - Libera la memoria ocupada por una estructura?

```
static void libera(Nodo *ra) {
  if (ra != NULL) {
        libera(ra->_iz);
        libera(ra->_dr);
        delete ra;
  }
}
```

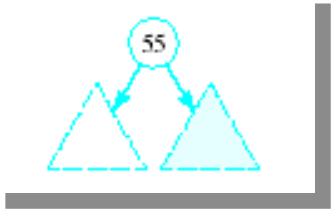
Implementación con estructuras de datos arbóreas: árboles de búsqueda

- Algunas operaciones para el manejo de la estructura de datos arbórea: ¿Cómo se…
  - Busca una clave en la estructura?

```
static Nodo* buscaAux(Nodo* p, const tClave &clave) {
  if (p == NULL)
        return NULL;
  else if (p->_clave == clave)
        return p;
  else if (clave < p->_clave)
        return buscaAux(p->_iz, clave);
  else
        return buscaAux(p->_dr, clave);
}
```



- Este método devuelve:
  - Un puntero al nodo asociado con la clave, en caso de que el árbol contenga una entrada con dicha clave.
  - `NULL` en otro caso
- El algoritmo sigue una estrategia similar a la de la búsqueda binaria



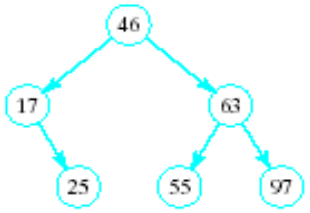
Figuras extraídas de: Nyhoff, ADTs, Data Structures and Problem Solving with C++, 2005 Pearson Education, Inc.

Implementación con estructuras de datos arbóreas: árboles de búsqueda

- Algunas operaciones para el manejo de la estructura de datos arbórea: ¿Cómo se…
  - Inserta un par clave-valor?

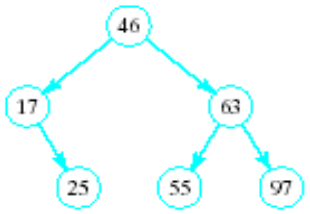
```
static Nodo* insertaAux(const tClave &clave, const tValor
  &valor, Nodo* p) {
        ...
```

  - Se devuelve un puntero a la raíz de la estructura resultante (la recibida, salvo que fuese vacía inicialmente)
    - Si la estructura es no vacía, no se reestructura, se acaba insertando una hoja si es necesario un nuevo nodo
  - El algoritmo se asemeja, de nuevo, al de la búsqueda binaria:
    - Si la estructura era vacía (p = NULL), se devuelve un puntero a una estructura con un único nodo, que aloja el par clave-valor
    - Si la clave de la raíz coincide con *clave*, se actualiza el valor, y se devuelve el puntero al nodo cuyo contenido se ha modificado (la raíz de la estructura)
    - Si la clave a insertar es menor que la raíz, se realiza la inserción en el hijo izquierdo y se actualiza el puntero a dicho hijo. Se devuelve puntero a la raíz de la estructura
    - Si es mayor, se hace lo propio con el hijo derecho



Implementación con estructuras de datos arbóreas: árboles de búsqueda

```
static Nodo* insertaAux(const tClave &clave, const tValor
  &valor, Nodo* p) {
        if (p == NULL)
                return new Nodo(clave, valor);
        else if (p->_clave == clave) {
                p->_valor = valor;
                return p;
        }
        else if (clave < p->_clave) {
                p->_iz = insertaAux(clave, valor, p->_iz);
                return p;
        }
        else { // (clave > p->_clave)
                p->_dr = insertaAux(clave, valor, p->_dr);
                return p;
        }
}
```

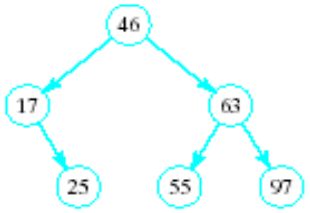


Implementación con estructuras de datos arbóreas: árboles de búsqueda

- Algunas operaciones para el manejo de la estructura de datos arbórea: ¿Cómo se…
  - Borra un par clave-valor?

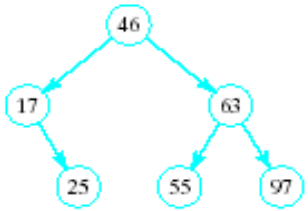
```
static Nodo* borraAux(Nodo* p, const tClave &clave) {
  …
```

    - Se devuelve un puntero a raíz de la estructura resultante
      - Si la estructura es no vacía, puede ser necesario reestructurarla para mantener el orden



- El algoritmo se asemeja, de nuevo, al de la búsqueda binaria:
  - Si la estructura era vacía (p = NULL), se devuelve NULL
  - Si la clave de la raíz coincide con *clave*, hay **que eliminarla y reestructurar adecuadamente**, y devolver un puntero a la raíz de la nueva estructura
  - Si la clave a borrar es menor que la raíz, se realiza el borrado en el hijo izquierdo y se actualiza el puntero a dicho hijo. Se devuelve puntero a la raíz de la estructura dada
  - Si es mayor, se hace lo propio con el hijo derecho

Implementación con estructuras de datos arbóreas: árboles de búsqueda




```
static Nodo *borraAux(Nodo *p, const tClave &clave) {
  if (p == NULL)
     return NULL;
  else
    if (clave == p->_clave)
        return borraRaiz(p);
    else
        if (clave < p->_clave) {
           p->_iz = borraAux(p->_iz, clave);
           return p;
        }
        else { // clave > p->_clave
           p->_dr = borraAux(p->_dr, clave);
           return p;
        }
}
```

Implementación con estructuras de datos arbóreas: árboles de búsqueda

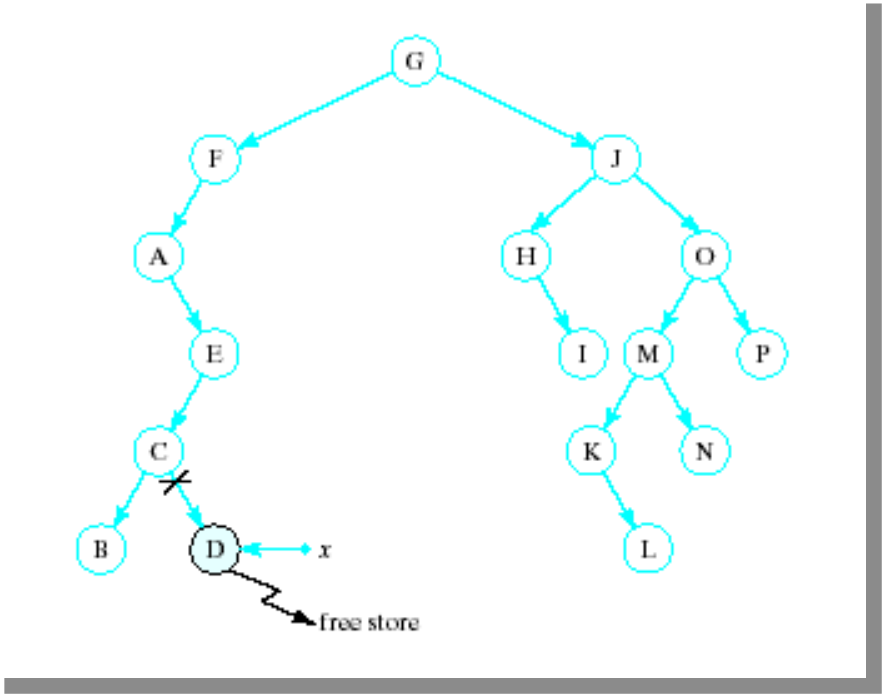
- Cuando se encuentra la clave hay **que eliminarla y reestructurar adecuadamente** y devolver un puntero a la raíz de la nueva estructura

```
static Nodo* borraRaiz(Nodo* p) {
```

  - Los casos a contemplar son los siguientes:
    a) El nodo que contiene la clave (apuntado por p) es una hoja
    b) El nodo que contiene la clave (apuntado por p) tiene un único hijo
    c) El nodo que contiene la clave (apuntado por p) tiene dos hijos

Implementación con estructuras de datos arbóreas: árboles de búsqueda

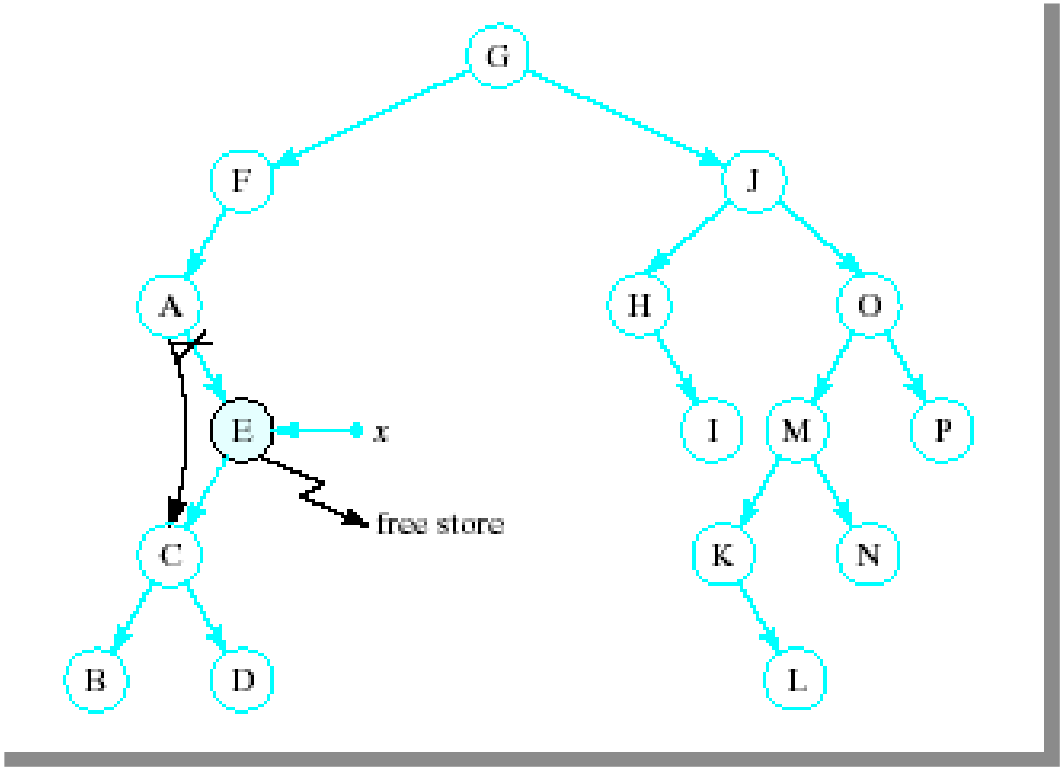
- Cuando se encuentra la clave hay **que eliminarla y reestructurar adecuadamente** y devolver un puntero a la raíz de la nueva estructura
  a) La clave se encuentra en una hoja



Figuras extraídas de: Nyhoff, ADTs, Data Structures and Problem Solving with C++, 2005 Pearson Education, Inc.

Implementación con estructuras de datos arbóreas: árboles de búsqueda

- Cuando se encuentra la clave hay **que eliminarla y reestructurar adecuadamente** y devolver un puntero a la raíz de la nueva estructura
  b) La clave se encuentra en un nodo que tiene un único hijo



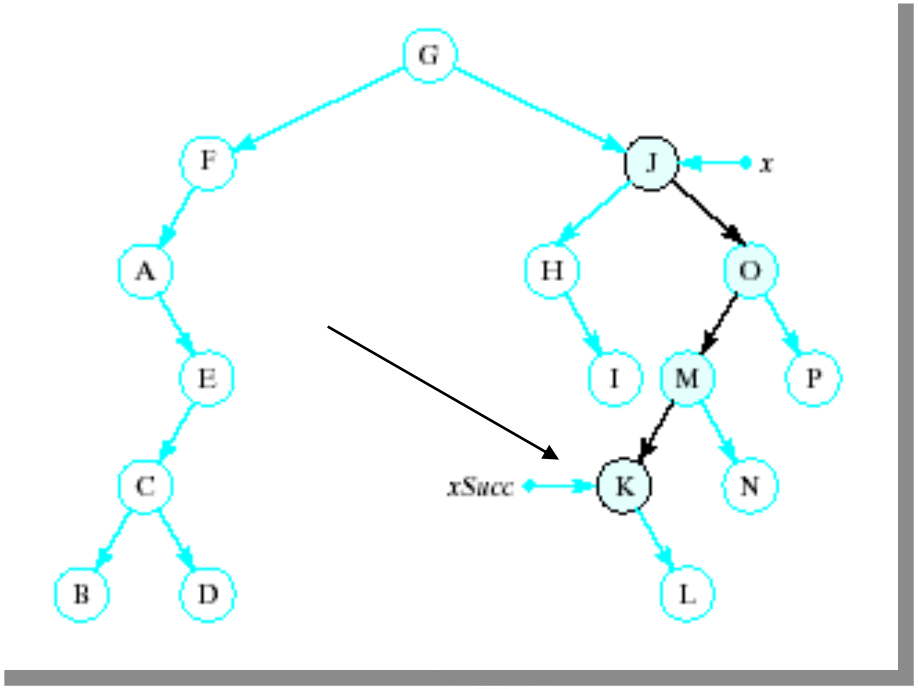
Figuras extraídas de: Nyhoff, ADTs, Data Structures and Problem Solving with C++, 2005 Pearson Education, Inc.

Implementación con estructuras de datos arbóreas: árboles de búsqueda

- Cuando se encuentra la clave hay **que eliminarla y reestructurar adecuadamente** y devolver un puntero a la raíz de la nueva estructura
  - c) La clave se encuentra en un nodo que tiene dos hijos
    1. Reemplazarla con su sucesor (la clave más pequeña de su hijo derecho) → ir al hijo derecho y descender por la izquierda lo máximo posible para localizar al sucesor y coser adecuadamente los enlaces (*)
    2. Eliminar el nodo que contiene la clave a borrar

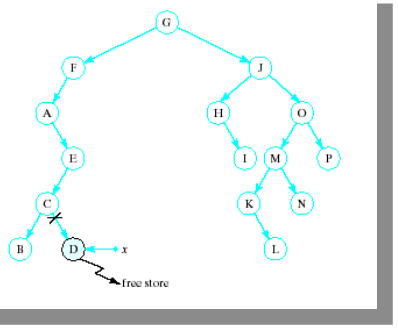
(*) Otra estrategia posible es reemplazarla con su predecesor

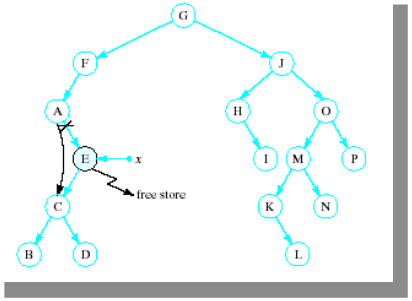
Implementación con estructuras de datos arbóreas: árboles de búsqueda



Implementación con estructuras de datos arbóreas: árboles de búsqueda

```
static Nodo *borraRaiz(Nodo *p) {
   Nodo *aux;
   if (p->_iz == NULL) {  // Si no hay hijo izquierdo, la raíz pasa a ser la del hijo derecho
                          // Esto también sirve para el caso de que sea un nodo hoja
         aux = p->_dr;
         delete p;
         return aux;
   }
   else
      if (p->_dr == NULL) {// Si no hay hijo derecho, la raíz pasa a ser la del hijo izquierdo
         aux = p->_iz;
         delete p;
         return aux;
      }
      else   // La nueva raíz es el elemento más pequeño del hijo derecho (sucesor de la actual raíz)
         return mueveMinYBorra(p);
}
```





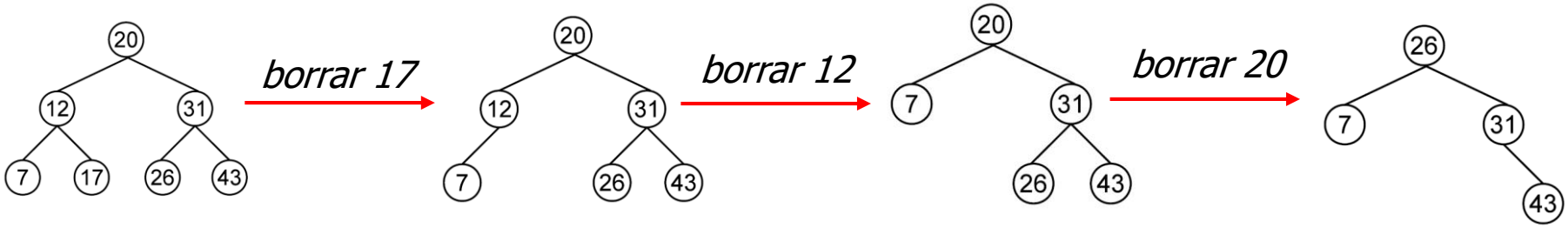
Implementación con estructuras de datos arbóreas: árboles de búsqueda

```
static Nodo *mueveMinYBorra(Nodo *p) {

    // Localizar el sucesor
    Nodo *padre = NULL;
    Nodo *aux = p->_dr;
    while (aux->_iz != NULL) {
            padre = aux;
            aux = aux->_iz;
    }
  // sigue…
```

Implementación con estructuras de datos arbóreas: árboles de búsqueda

```
  // Dos casos dependiendo de si el sucesor es la raíz
  // del hijo derecho del nodo a borrar (=> padre == NULL) o no (=> padre != NULL)
  if (padre != NULL) {
     padre->_iz = aux->_dr; // el hijo derecho del sucesor pasa a
ser el nuevo izquierdo de su padre
     // el sucesor ocupa el puesto del que hay que borrar...
     aux->_iz = p->_iz;
     aux->_dr = p->_dr;
  }
  else // el sucesor es la raíz del subarbol derecho de la clave a borrar
     aux->_iz = p->_iz;

  delete p; // eliminamos el nodo con la clave a borrar
  return aux; // devolvemos la nueva raíz
}
```

Ejemplos de borrados



Implementación con árboles de búsqueda: operaciones básicas

- Operaciones `diccionarioVacio` y `esVacio`

```
Diccionario() : _ra(NULL) {}

bool esVacio() const {
    return _ra == NULL;
}
```

Implementación con árboles de búsqueda: operaciones básicas

- Operaciones `contiene` y `valorPara`, directas usando el método `buscaAux`

```
class EClaveErronea {}; // clase excepción para la operación valorPara

bool contiene(const tClave &clave) const {
    return (buscaAux(_ra, clave) != NULL) ? true : false;
}

const tValor& valorPara(const tClave &clave) const {
    Nodo *p = buscaAux(_ra, clave);
    if (p == NULL)
            throw EClaveErronea();
    return p->_valor;
}
```

Implementación con árboles de búsqueda: operaciones básicas

- Operación `inserta`, directa usando `insertaAux`

```
void inserta(const tClave &clave, const tValor &valor) {
    _ra = insertaAux(clave, valor, _ra);
}
```

- Operación `borra`, directa usando `borraAux`

```
void borra(const tClave &clave) {
    _ra = borraAux(_ra, clave);
}
```

- Las operaciones de destrucción, construcción por copia y operador de asignación son análogas a las vistas en el tema anterior

Implementación con árboles de búsqueda: complejidad de las operaciones

Asumiendo árboles balanceados:

diccionarioVacio	O(1)
inserta	O(log n)
borra	O(log n)
contiene	O(log n)
valorPara	O(log n)
esVacio	O(1)

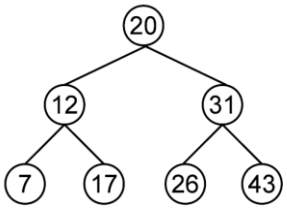
- En el caso de **árboles degenerados**, se pierde el comportamiento logarítmico (las complejidades O(log n) degeneran en O(n))
- Si se supone que la aparición de claves es aleatoria (con distribución uniforme), las complejidades promedio son logarítmicas
- También es posible complicar la implementación, incluyendo lógica de *reequilibrado*, que permite mantener los árboles de búsqueda balanceados tras las inserciones y los borrados (p.e.: **árboles AVL**). Esas implementaciones (que no se verán en este curso) garantizan comportamiento logarítmico.

Implementaciones con árboles de búsqueda: iteradores

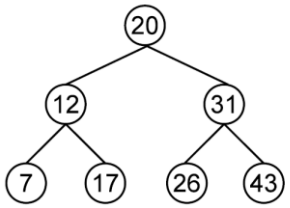
- El recorrido en inorden de un árbol de búsqueda permite recorrer los elementos ordenados en orden ascendente por clave
- Es posible implementar **iteradores** que reproduzcan ese recorrido (sin necesidad de generarlo explícitamente)
- Lo mismo que en el caso de las listas, será posible implementar:
  - Iteradores **constantes**, que únicamente permitan consultar las claves y valores de los elementos
  - Iteradores **no constantes**, que, además, permitan modificar los valores (no las claves, ya que, si se pudieran modificar las claves, podría destruirse el invariante de la representación –es decir, el orden en el árbol)
- La iteración se valdrá de una pila, en la que se apilarán los nodos pendientes sobre los que iterar.

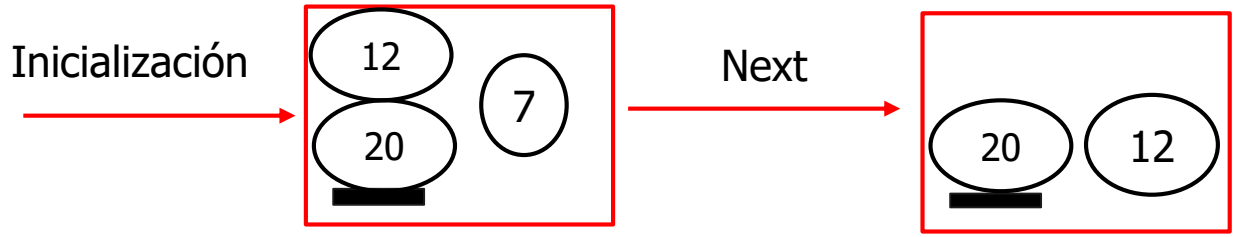
Implementaciones con árboles de búsqueda: iteradores

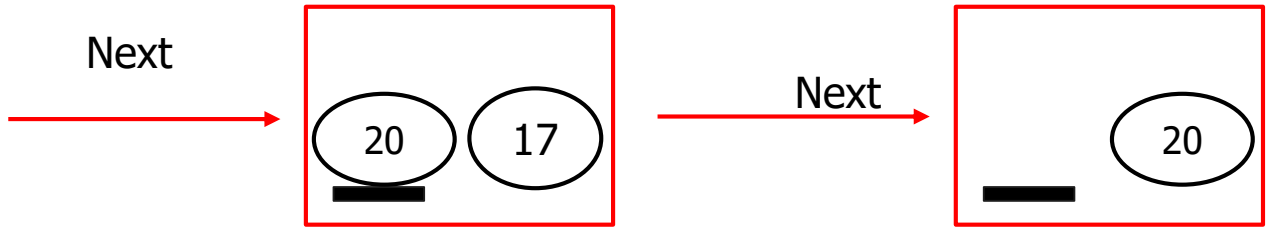
- Para localizar el primer nodo sobre el que iterar, se desciende por la izquierda hasta encontrar un nodo sin hijo izquierdo.
  - Durante el descenso, los nodos que se atraviesan se apilan en la pila de pendientes de ser visitados
- Para localizar el nodo siguiente:
  - Si el actual tiene hijo derecho, se procede con dicho hijo como con la raíz durante la inicialización (es decir, se desciende por la izquierda, apilando los nodos que se atraviesan, hasta encontrar uno sin hijo izquierdo: dicho nodo será el siguiente en el recorrido en inorden)
  - En otro caso, y en caso de que haya nodos pendientes de visitar, se desapila un nodo de la pila de pendientes.

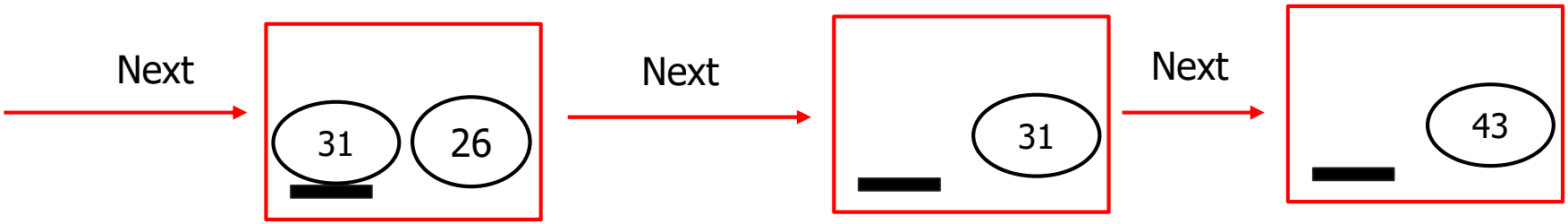


Implementaciones con árboles de búsqueda: iteradores









Implementaciones con árboles de búsqueda: iteradores constantes (ConstIterator)

```
class EAccesoNoValido{}; // Excepción que se levanta si se desea iterar más allá del final
     ...
template <class tClave, class tValor>
class Diccionario {
private:
    // parte privada
public:
    class ConstIterator {
     public:
       void next() {...}
       const tClave & clave() const {...}
       const tValor & valor() const {...}
       bool operator==(const ConstIterator &other) const {...}
       bool operator!=(const ConstIterator &other) const {...}
```

Implementaciones con árboles de búsqueda: iteradores constantes (ConstIterator)

```
    protected:
       friend class Diccionario; // Para que pueda construir objetos del tipo iterador
       ConstIterator() : _act(NULL) {}
       ConstIterator(Nodo *act) {...}
       Nodo *_act; // Puntero al nodo actual del recorrido. NULL si hemos llegado al final.
       Pila<Nodo*> _ascendientes; // Ascendientes del nodo actual aún por visitar
    }; // fin de la clase ConstIterator

    // métodos de Diccionario que devuelven iteradores al principio y al final
     ConstIterator cbegin() const {
           return ConstIterator(_ra);
     }
     ConstIterator cend() const {
           return ConstIterator();
     }
     // resto de elementos públicos de la clase Diccionario
};
```

Implementaciones con árboles de búsqueda: iteradores constantes (`ConstIterator`)

- Las implementaciones del constructor no vacío (usado por `cbegin`) y del método `next` pueden facilitarse mediante un método privado de la clase `ConstIterator` que localice el primer elemento en el recorrido en inorden de una estructura arbórea, almacenando en la pila de pendientes los nodos en la rama que se atraviesa para ello:

```
Nodo* primeroInOrden(Nodo *p) {
 if (p == NULL) return NULL;
 while (p->_iz != NULL) {
  _ascendientes.apila(p);
  p = p->_iz;
 }
 return p;
}
```

- Con ello, la implementación del constructor no vacío es directa:

```
ConstIterator(Nodo *act) {
   _act = primeroInOrden(act);
}
```

Implementaciones con árboles de búsqueda: iteradores constantes (`ConstIterator`)

- La implementación de `next` refleja la estrategia ya explicada:

```
void next() {
  if (_act == NULL) throw EAccesoNoValido();
    // Si hay hijo derecho, saltamos al primero en inorden del hijo derecho
  if (_act->_dr != NULL)
        _act = primeroInOrden(_act->_dr);
  else {
  // Si no, vamos al primer ascendiente no visitado. Para eso consultamos la pila; si ya está vacía,
  // no quedan ascendientes por visitar
        if (_ascendientes.esVacia())
              _act = NULL;
        else {
              _act = _ascendientes.cima();
              _ascendientes.desapila();
        }
 }
}
```

Implementaciones con árboles de búsqueda: iteradores constantes (`ConstIterator`)

- Implementación del resto de operaciones:

```
const tClave & clave() const {
  if (_act == NULL) throw EAccesoNoValido();
  return _act->_clave;
}

const tValor & valor() const {
  if (_act == NULL) throw EAccesoNoValido();
  return _act->_valor;
}

bool operator==(const ConstIterator &other) const {
  return _act == other._act;
}

bool operator!=(const ConstIterator &other) const {
  return !(this->operator==(other));
}
```

Implementaciones con árboles de búsqueda: iteradores no constantes (`Iterator`)

- Además de `ConstIterator`, el diccionario puede proporcionar iteradores de tipo `Iterator`
- La interfaz de dichos iteradores es análoga a la de `ConstIterator`, con la salvedad de proporcionar un método `setVal`, que permite modificar el valor del par *clave-valor* referido por el iterador

```
void setVal(const tValor& valor) {…}
```

  Y el método `valor`

```
tValor & valor() {…}
```

- La implementación es también análoga a la de `ConstIterator`
- Estos iteradores se obtienen mediante métodos `begin` y `end`

Implementaciones con árboles de búsqueda: más operaciones mediante iteradores

- Si un usuario de nuestro TAD Diccionario quiere protegerse de los accesos indebidos al llamar al método `valorPara` debe asegurarse primero de que la clave existe utilizando `contiene`. Eso implica hacer dos búsquedas sobre el árbol.
- Si, además, ese mismo usuario quiere posteriormente modificar el valor asociado a esa clave, incurrirá en una nueva búsqueda al llamar a `inserta`.
- Para aliviar lo anterior resultaría interesante contar con una operación de búsqueda que, en lugar de devolver un booleano o un valor, devuelva un iterador al punto donde está una clave dada (o "fuera" si no está) (`cbusca` para iteradores constantes, `busca` para iteradores no constantes).

Implementaciones con árboles de búsqueda: más operaciones mediante iteradores

- Operación `cbusca` que devuelve un `ConstIterator` (la operación `busca` que devuelve un `Iterator` será análoga):

```
ConstIterator cbusca(const tClave &c) const {
        Pila<Nodo*> ascendientes;
        Nodo *p = _ra;
        while ((p != NULL) && (p->_clave != c)) {
        if (p->_clave > c) {
                ascendientes.apila(p);
                p = p->_iz;
        }
        else
                p = p->_dr;
        }
        ConstIterator ret;
        ret._act = p;
        if (p != NULL) ret._ascendientes = ascendientes;
        return ret;
}
```

Implementaciones con tablas dispersas: motivación

- Las tablas dispersas (o tablas hash) se basan en almacenar en un array los valores y usar las claves (transformadas) como índices.
- Supongamos que los valores posibles para la clave son, simplemente, números enteros del 0 al 9
  - La representación mediante un array de valores de 10 elementos sería muy eficiente (todas las operaciones pasarían a tener complejidad O(1))
- Si el tipo de la clave fuera `unsigned short` (clave $\in$ [0, 65535]), asumiendo que los valores de estos tipos ocupan 2 bytes, necesitaríamos un array de 2^{16} (65536) elementos. Aunque muchos de ellos estarían sin utilizar, la representación todavía resulta asumible.
- Si el tipo de la clave fuera `short` (clave $\in$ [-32767, 32767]), aparte del array de 2^{16} elementos, necesitaríamos **transformar** la clave K en una posición del array. Por ejemplo:
  - $k \geq 0 \rightarrow k$
  - $k < 0 \rightarrow$ (`SHRT_MAX` - 1)+|k| (`SHRT_MAX`, máximo valor `short`).
- Para otros tipos de datos numerales, el tamaño del array (y, por tanto, el desperdicio en espacio) puede ser mucho mayor.
- En el caso de tipos de datos no numerales, aparte del potencial desperdicio en espacio, debe tenerse en cuenta la **transformación** a índices en el array.
- Esta reflexión conduce otra característica de las tablas dispersas:
  - ¿Por qué no permitir que distintas claves puedan compartir un mismo índice?
  - Para lograr esto, debe pensarse como almacenar y recuperar los valores de estas claves cuando ocurren simultáneamente en la tabla: resolución de **colisiones**.

Implementaciones con tablas dispersas: conceptos

- Una representación basada en tablas dispersas consiste en:
  - Un array de N elementos.
  - Una **función de localización** l: `tClave` → `0 .. N-1`, que asigna un índice en dicho array a cada clave.
- El valor l(*k*) se denomina el **índice** de *k*.
- Una forma de definir l es en términos de una **función de codificación** (función **hash**) h: `tClave` → `0 .. M-1` como:

  `l(k) = h(k)%N`
- Un ejemplo muy sencillo (aunque no demasiado bueno) de función de localización cuando:
  - N = 16
  - `tClave` ≡ `string`

  l(*s*) = ((int)s.back())%16

  (es decir, el código del último carácter del string, módulo 16)

  (en este caso, h(s) = (int)s.back())
- Ejemplos (se asume que el código de los caracteres es ASCII):
  - `l("Fred") = ((int)'d')%16 = 100%16 = 4`
  - `l("Joe") = ((int)'e')%16 = 101%16 = 5`
  - `l("John")= ((int)'n')%16 = 110%16 = 14`

Implementaciones con tablas dispersas: conceptos

- Propiedades deseables para una buena función de localización **l**:
  - **Eficiencia**: El coste de calcular **l** debe ser bajo.
  - **Uniformidad**: **l** debe distribuir las claves uniformemente sobre 0 .. N-1. Idealmente, si *k* se elige al azar, P(l(*k*)=*i*) = 1 / N para cada *i* en 0 .. N-1.
- La uniformidad minimiza la probabilidad de **colisión**.
- Se dice que dos claves k_0 y k_1 distintas entre sí **colisionan** cuando ambas tienen el mismo índice (es decir, cuando l(k_0)=l(k_1)).
- Por ejemplo, con la función de localización anterior, todas las cadenas distintas que terminen con el mismo carácter colisionarán (aparte de poder hacerlo con otras):

```
l("Fred") = l("David") = l(Roland") = l("Violet") = 4
```

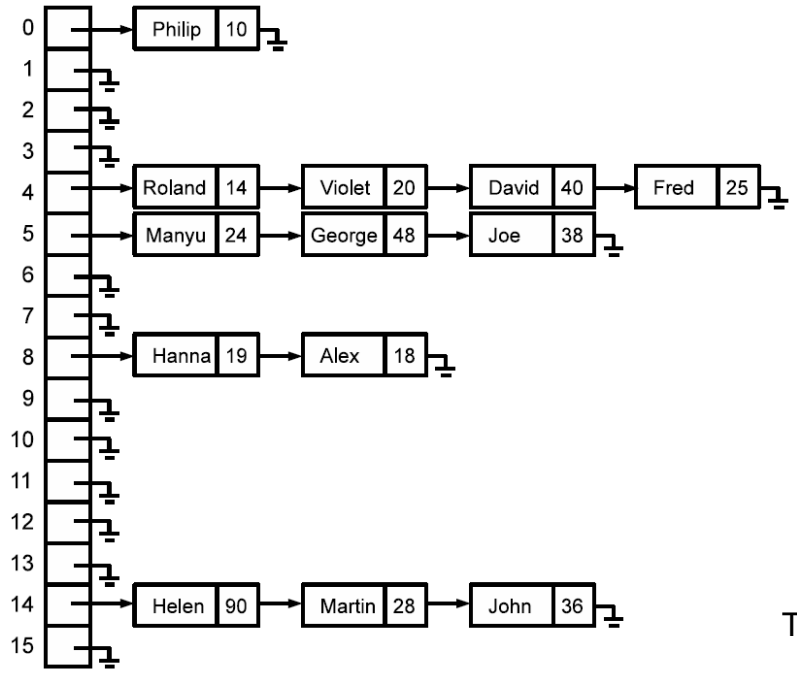
- La probabilidad de que se produzca colisión depende de muchos factores:
  - Lo buena o mala que sea la función de localización
  - El número de índices posible (tamaño de la tabla)
  - El número de claves posible
  - Las probabilidades de ocurrencia de cada clave
- En general, dicha probabilidad es bastante grande (paradoja del cumpleaños: la probabilidad de que en un grupo de 23 o más personas haya al menos dos que cumplen años el mismo día del año es mayor que ½)

Implementaciones con tablas dispersas: conceptos

- **Resolución de colisiones**: política que permite almacenar, en una misma tabla, los valores de las claves que colisionan
- **Tablas abiertas**
  - Los valores de las componentes del array son, a su vez, una estructura que permite almacenar más de un par *clave – valor*, así como recuperar los valores de las claves (p.e., soportado por una lista, un array dinámico ordenado por clave, o un árbol de búsqueda)
  - De esa forma, cada posición *i* en el array almacena los pares asociados a todas las claves que han colisionado con código de localización *i*
  - La recuperación es en dos fases:
    - En la primera se localiza la posición en el array en la que debe residir la clave
    - En la segunda, se busca la clave en sí en la estructura asociada a dicha posición
- **Tablas cerradas**
  - Si se produce una colisión, se busca otra posición en el array en la que almacenar el par clave – valor.
  - Para ello, se aplica una estrategia de recolocación (p.e., comprobar posiciones del array a partir de la posición colisionante en algún orden determinado –p.e. buscar linealmente- hasta encontrar la primera posición libre)
  - Dicha estrategia se tiene también en cuenta durante la recuperación (habrá que comparar claves de acuerdo con la estrategia de recolocación hasta que se encuentre la clave buscada, o bien se agoten las posibilidades)
- En este curso nos ocuparemos de las tablas abiertas

Implementaciones con tablas dispersas: tablas abiertas

- **Tipos representantes:**
  - Se utilizará una lista enlazada simple de nodos, donde cada nodo almacena un par *clave – valor*
  - Se utilizará un array de punteros a nodos para representar la tabla (puntero a puntero a nodo)
  - El array será dinámico: cuando se alcance cierta **ocupación** (número de elementos en la tabla), el array se ampliará y los elementos se recolocarán
  - Además, por ser una representación basada en array dinámico se mantendrán los tamaños del array (inicial y actual) y el número de elementos de la tabla



Implementaciones con tablas dispersas: tablas abiertas

- **Tipos representantes:**

```
template <class tClave, class tValor>
class Diccionario {
private:

  class Nodo {
  public:
        /* Constructores. */
     Nodo(const tClave &clave, const tValor &valor) :
        _clave(clave), _valor(valor), _sig(NULL) {};

     Nodo(const tClave &clave, const tValor &valor, tNodo *sig) :
        _clave(clave), _valor(valor), _sig(sig) {};

        /* Atributos públicos. */
     tClave _clave;
     tValor _valor;
     Nodo *_sig;   // Puntero al siguiente nodo.
   };
```

Clase `Nodo` para formar la estructura enlazada de pares clave-valor en cada posición del array

Implementaciones con tablas dispersas: tablas abiertas

Clase `Tabla` para representar la tabla dispersa utilizada para almacenar el contenido del diccionario

```
class Tabla {
public:
   Nodo **_v;                ///< Array de punteros a Nodo.
   unsigned int _tam;        ///< Tamaño actual del array _v.
   unsigned int _numElems;   ///< Número de elementos en la tabla.
};
```

El atributo `tabla` almacena la tabla en sí.

```
Tabla _tabla;   //< Tabla dispersa

// Resto de elementos privados

public:

    /**
     * Tamaño inicial de la tabla.
     */
    static const int TAM_INICIAL = 8;

// Resto de elementos públicos
```

El atributo `TAM_INICIAL` es el tamaño inicial del array

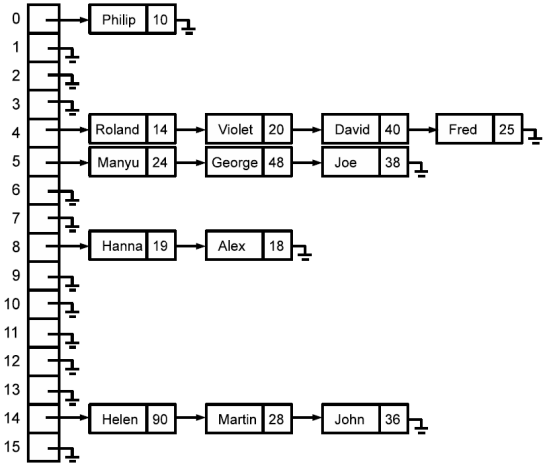
Implementaciones con tablas dispersas: tablas abiertas

- Algunas operaciones para manejar la representación de tablas abiertas. ¿Cómo se
  - Genera una copia de una estructura?
  - Libera la memoria ocupada por una estructura?
  - Busca una clave en la estructura?
  - Inserta un par clave-valor?
  - Amplía la capacidad de la tabla?
  - Borra un par clave-valor?

Implementaciones con tablas dispersas: Tablas abiertas

- Algunas operaciones para manejar la representación de tablas abiertas. ¿Cómo se
  - Genera una copia de una estructura?

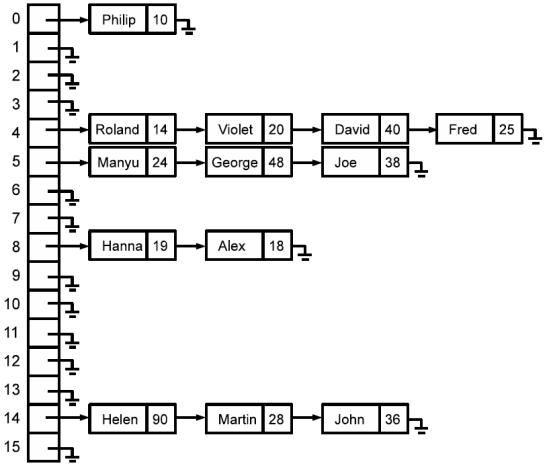
```
static void copia(Tabla& tabla, const Tabla &other) {
  tabla._tam = other._tam;
  tabla._numElems = other._numElems;

   // Reservar memoria para el array de punteros a nodos.
  tabla._v = new Nodo*[tabla._tam];
  for (unsigned int i=0; i<tabla._tam; ++i) {
     tabla._v[i] = NULL;
    // Copiar la lista de nodos de other._v[i] a _v[i].
    // La lista de nodos queda invertida con respecto a la original.
     Nodo *act = other._v[i];
     while (act != NULL) {
         tabla._v[i] = new Nodo(act->_clave, act->_valor,
                                tabla._v[i]);
         act = act->_sig;
     }
   }
}
```



Implementaciones con tablas dispersas: tablas abiertas

- Algunas operaciones para manejar la representación de tablas abiertas. ¿Cómo se
  - Libera la memoria ocupada por una estructura?

```
static void libera(Tabla& tabla) {
  // Liberamos las listas de nodos.
  for (unsigned int i=0; i<tabla._tam; i++)
    liberaNodos(tabla._v[i]);
  }
  // Liberamos el array de punteros a nodos.
  delete[] tabla._v;
  tabla._v = NULL;
 // actualizamos el tamaño y el nº de elementos
  tabla._tam = 0;
  tabla._numElems = 0;
}
```



(* SIGUE *)

Implementaciones con tablas dispersas: tablas abiertas

- Algunas operaciones para manejar la representación de tablas abiertas. ¿Cómo se
  - Libera la memoria ocupada por una estructura?

```
static void liberaNodos(Nodo *prim) {
  while (prim != NULL) {
   Nodo *aux = prim;
   prim = prim->_sig;
   delete aux;
  }
}
```

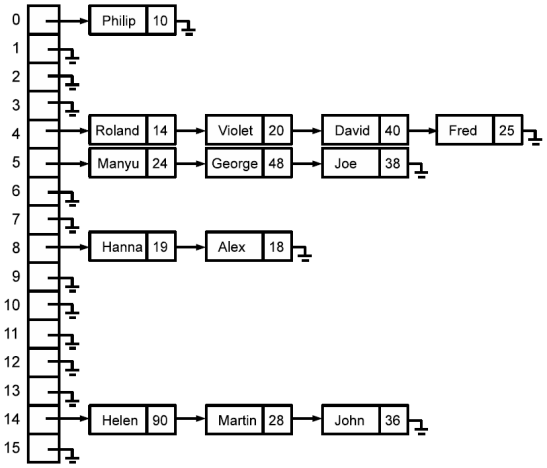
Implementaciones con tablas dispersas: Tablas abiertas

- Algunas operaciones para manejar la representación de tablas abiertas. ¿Cómo se
  - Busca una clave en la estructura?

```
static tValor &buscaAux(const Tabla& tabla,
                        const tClave &clave) {
    // Obtenemos el índice asociado a la clave.
    unsigned int ind = ::h(clave) % tabla._tam;

    // Buscamos un nodo que contenga esa clave.
    Nodo *nodo = buscaNodo(clave, tabla._v[ind]);
    if (nodo == NULL) throw EClaveErronea();
    return nodo->_valor;
}
```

Búsqueda en dos fases:

1. Se localiza la posición en el array en la que debe residir la clave
2. Se busca la clave en sí en la lista enlazada asociada a dicha posición



Función de codificación (función ***hash***). Habrá que sobrecargarla para cada `tClave`.

(* SIGUE *)

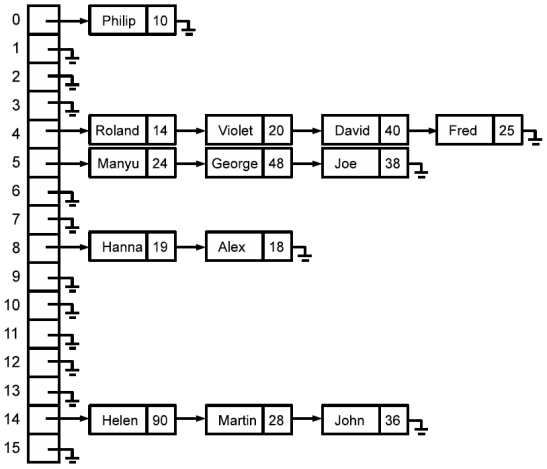
Implementaciones con tablas dispersas: tablas abiertas

- Algunas operaciones para manejar la representación de tablas abiertas. ¿Cómo se
  - Busca una clave en la estructura?

```
static Nodo* buscaNodo(const tClave &clave, Nodo* prim) {
   Nodo *act = prim;
   Nodo *ant;
   buscaNodo(clave, act, ant);
   return act;
}

static void buscaNodo(const tClave &clave, Nodo* &act,
                      Nodo* &ant) {
   ant = NULL;
   bool encontrado = false;
   while ((act != NULL) && !encontrado) {
    // Comprobar si el nodo actual contiene la clave buscada
     if (act->_clave == clave) {encontrado = true;}
     else {ant = act;   act = act->_sig;}
   }
}
```

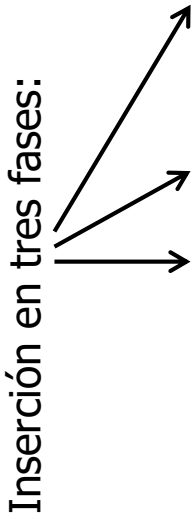
Se reutilizará en la inserción

Se reutilizará en el borrado



Implementaciones con tablas dispersas: tablas abiertas

- Algunas operaciones para manejar la representación de tablas abiertas. ¿Cómo se
  - Inserta un par clave-valor?

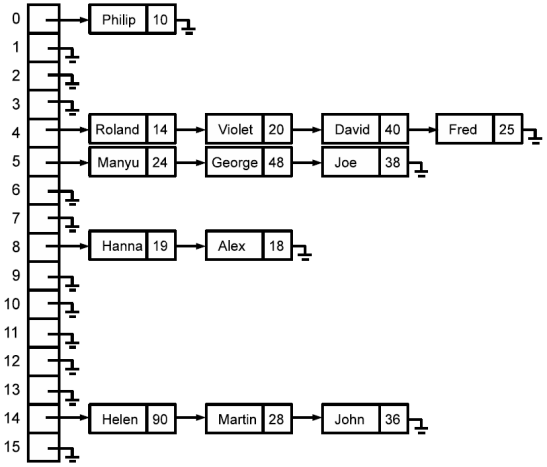



```
// Ocupación máxima permitida antes de ampliar la tabla en tanto por ciento.
static const unsigned int MAX_OCUPACION = 80;

static void insertaAux(Tabla& tabla, const tClave &clave, const tValor &valor) {
    // Si la ocupación es muy alta ampliamos la tabla
    float ocupacion = 100 * ((float)tabla._numElems) / tabla._tam;
    if (ocupacion > MAX_OCUPACION)  amplia(tabla);
    // Obtenemos el índice asociado a la clave.
    unsigned int ind = ::h(clave) % tabla._tam;
    // Localización del punto de inserción, y actualización o inclusión (según proceda)
    Nodo *nodo = buscaNodo(clave, tabla._v[ind]);
    if (nodo != NULL) // Si la clave ya existía, actualizamos su valor
            nodo->_valor = valor;
    else {
    // Si la clave no existía, creamos un nuevo nodo y lo insertamos al principio
            tabla._v[ind] = new Nodo(clave, valor, tabla._v[ind]);
            tabla._numElems++;
    }
}
```

Implementaciones con tablas dispersas: tablas abiertas

- Algunas operaciones para manejar la representación de tablas abiertas. ¿Cómo se
  - Amplía la capacidad de la tabla?

```
static void amplia(Tabla& tabla) {
 // Creamos un puntero al array actual y anotamos su tamaño.
 Nodo **vAnt = tabla._v;
 unsigned int tamAnt = tabla._tam;

 // Duplicamos el array en otra posición de memoria.
 tabla._tam *= 2;
 tabla._v = new Nodo*[tabla._tam];
 for (unsigned int i=0; i<tabla._tam; ++i)
     tabla._v[i] = NULL;
```

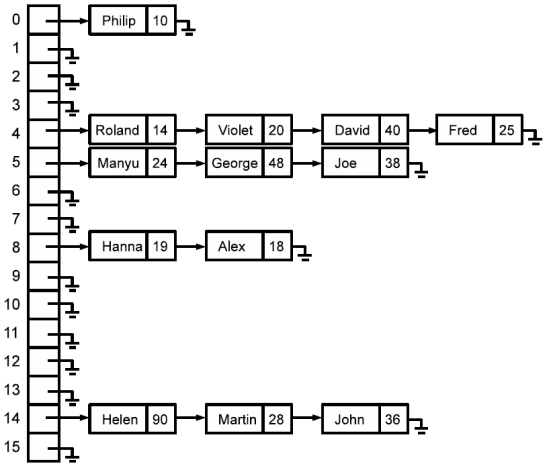



```
(* SIGUE *)
```

Implementaciones con tablas dispersas: tablas abiertas

- Algunas operaciones para manejar la representación de tablas abiertas. ¿Cómo se
  - Amplía la capacidad de la tabla?

```
// Recorremos el array original moviendo cada nodo a la nueva
 // posición que le corresponde en el nuevo array.
 for (unsigned int i=0; i<tamAnt; ++i) {
   // IMPORTANTE: Al modificar el tamaño también se modifica el índice
   // asociado a cada nodo. Es decir, los nodos se mueven a posiciones (potencialmente)
   // distintas en el nuevo array. Por eficiencia movemos los nodos del array antiguo al
   // nuevo, no creamos nuevos nodos.
   // Recorremos la lista de nodos llevando dos punteros para posibilitar
   // el cosido de enlaces sin extraviar nodos
   Nodo *nodo = vAnt[i];
   while (nodo != NULL) {
     Nodo *aux = nodo;
     nodo = nodo->_sig;
      // Calculamos el nuevo índice del nodo, lo desenganchamos del
      // array antiguo y lo enganchamos al nuevo.
     unsigned int ind = ::h(aux->_clave) % tabla._tam;
     aux->_sig = tabla._v[ind];
     tabla._v[ind] = aux;
    }
   }
 // Borramos el array antiguo (ya no contiene ningún nodo).
 delete[] vAnt;
}
```

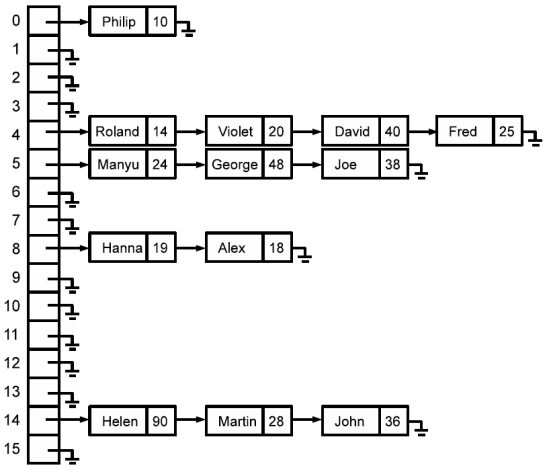


Implementaciones con tablas dispersas: tablas abiertas

- Algunas operaciones para manejar la representación de tablas abiertas. ¿Cómo se
  - Borra un par clave-valor?

Borrado en tres fases:

```
static void borraAux(Tabla& tabla, const tClave& clave) {
  // Obtenemos el índice asociado a la clave.
  unsigned int ind = ::h(clave) % tabla._tam;

  // Buscamos el nodo que contiene esa clave y el nodo anterior.
  Nodo *act = tabla._v[ind];
  Nodo *ant;
  buscaNodo(tabla, clave, act, ant);
  if (act != NULL) {// Si existe el par clave-valor, lo elimina
    // Sacamos el nodo de la secuencia de nodos.
    if (ant != NULL) {// Si no es el primer nodo de la lista
      ant->_sig = act->_sig;
    }
    else {
      tabla._v[ind] = act->_sig;
    }
     // Borramos el nodo extraído.
    delete act;
    tabla._numElems--;
   }
}
```



Implementaciones con tablas dispersas: operaciones básicas

- Operaciones `DiccionarioVacio` y `esVacio`

```
Diccionario()  {
  _tabla._tam = TAM_INICIAL;
  _tabla._numElems = 0;
  _tabla._v = new Nodo*[TAM_INICIAL];
  for (unsigned int i = 0; i<_tabla._tam; ++i) {
     _tabla._v[i] = NULL;
   }
 }

 bool esVacio() const {
   return _tabla._numElems == 0;
 }
```

Implementaciones con tablas dispersas: operaciones básicas

- Operaciones `contiene` y `valorPara`

```
bool contiene(const tClave &clave) const {
  // Obtenemos el índice asociado a la clave.
  unsigned int ind = ::h(clave) % _tabla._tam;
  // Buscamos un nodo que contenga esa clave.
  Nodo *nodo = buscaNodo(clave, _tabla._v[ind]);
  return nodo != NULL;
}

const tValor &valorPara(const tClave &clave) const {
  return buscaAux(_tabla, clave);
}
```

Implementaciones con tablas dispersas: operaciones básicas

- Operación `inserta`

```
void inserta(const tClave &clave, const tValor &valor) {
   insertaAux(_tabla, clave, valor);
}
```

- Operación `borra`

```
void borra(const tClave &clave) {
   borraAux(_tabla, clave);
}
```

Implementaciones con tablas dispersas: resto de operaciones

- Destructor

```
~Diccionario() {
    libera(_tabla);
}
```

- Operador de asignación

```
Diccionario &operator=(const Diccionario &other) {
  if (this != &other) {
    libera(_tabla);
    copia(_tabla, other._tabla);
  }
  return *this;
}
```

- Constructor de copia

```
Diccionario(const Diccionario &other) {
    copia(_tabla, other._tabla);
}
```

Implementaciones con tablas dispersas: Iteradores

- A veces puede ser útil recorrer todos los pares clave-valor almacenados en una tabla
  Por ello extenderemos el TAD con operaciones que permitan recorrer los elementos almacenados usando iteradores
- Hay que tener en cuenta que con esta implementación los pares *clave-valor* en el recorrido **no** tienen porque estar necesariamente ordenados (de hecho no se almacenan en la tabla siguiendo ningún orden)
- Nuestra elección será recorrer la lista de colisiones de la posición 0 del array, luego la de la posición 1, etc.
  - Puede haber listas de colisiones vacías, lo que implicará que puede que tengamos que saltarnos posiciones del array
- Comentamos a continuación los iteradores mutables (`Iterator`)
  - La implementación de `ConstIterator` es análoga

Implementaciones con tablas dispersas: Iteradores mutables

```
class Iterator {
  public:
   void next() { ...}
   const tClave &clave() const {...}
   tValor &valor() const {...}
   void setVal(const tValor& v) {...}
   bool operator==(const Iterator &other) const {...}
   bool operator!=(const Iterator &other) const {...}
  protected:
   // Para que pueda construir objetos del tipo iterador
   friend class Diccionario;
   Iterator(const Tabla& tabla, Nodo* act, unsigned int ind)
           : _tabla(tabla), _act(act), _ind(ind) { }

    const Tabla& _tabla;   ///< Tabla que se está recorriendo
    Nodo* _act;            ///< Puntero al nodo actual del recorrido
    unsigned int _ind;     ///< Índice actual en el vector _v
};
```

Implementaciones con tablas dispersas: Iteradores mutables

- Método next

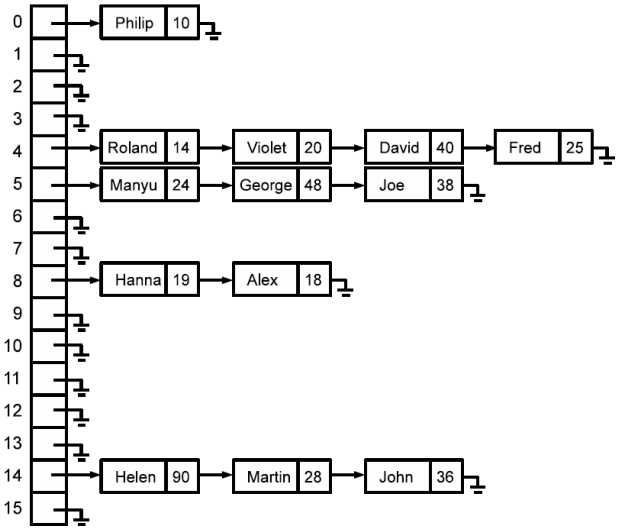
```
void next() {
  if (_act == NULL) throw EAccesoNoValido();
  // Buscamos el siguiente nodo de la lista de nodos.
  _act = _act->_sig;
  // Si hemos llegado al final de la lista de nodos, seguimos buscando por el vector _v.
  // Hay que tener en cuenta que puede haber listas de colisiones vacías
  while ((_act == NULL) && (_ind < _tabla._tam - 1)) {
    ++_ind;
    _act = _tabla._v[_ind];
  }
}
```

- Método clave

```
const tClave &clave() const {
  if (_act == NULL) throw EAccesoNoValido();
  return _act->_clave;
}
```

- Método valor

```
tValor &valor() const {
   if (_act == NULL) throw EAccesoNoValido();
   return _act->_valor;
}
```



Implementaciones con tablas dispersas: Iteradores mutables

- Método `setVal`

```
void setVal(const tValor& v) {
  if (_act == NULL) throw EAccesoNoValido();
  _act->_valor = v;
}
```

- Igualdad

```
bool operator==(const Iterator &other) const {
  return _act == other._act;
}
```

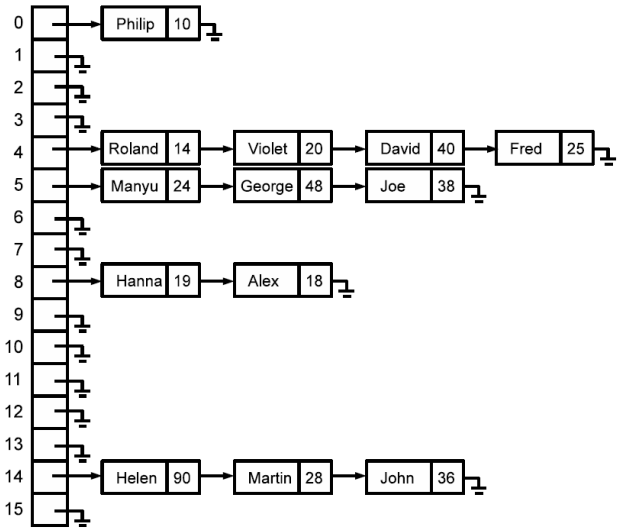
- Desigualdad

```
bool operator!=(const Iterator &other) const {
   return !(this->operator==(other));
}
```

Implementaciones con tablas dispersas: Iteradores mutables

- Al igual que en la implementación basada en árboles de búsqueda, la clase `Diccionario` tendrá métodos para devolver iteradores al comienzo y al final del recorrido

```
Iterator begin() {
  unsigned int ind = 0;
  Nodo *act = _tabla._v[0];
  while (ind < _tabla._tam-1 && act == NULL) {
   ind++;
   act = _tabla._v[ind];
  }
  return Iterator(_tabla, act, ind);
}

Iterator end() const {
   return Iterator(_tabla, NULL, 0);
}
```



Implementaciones con tablas dispersas: más operaciones mediante iteradores

- Operación `busca` para devolver un iterador situado en un par clave-valor (`cbusca` será análoga):

```
Iterator busca(const tClave &clave) {
 // Obtenemos el índice asociado a la clave.
 unsigned int ind = ::h(clave) % _tabla._tam;
 // Buscamos un nodo que contenga esa clave.
 Nodo *nodo = buscaNodo(clave, _tabla._v[ind]);
  // si nodo == NULL, devuelve en realidad end()
 return Iterator(_tabla, nodo, ind);
}
```

Implementaciones con tablas dispersas: Funciones de localización

- Técnicas para valores ordinales (enteros …)
  - Aritmética modular y uso de números primos: *n* % N, con N un número primo
    - N primo para minimizar la probabilidad de que los valores ordinales de las claves tengan divisores comunes con N.
      - Espacio de claves uniforme 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,…
        - Si N=10 → 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1,…
        - Si N=4 → 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3,…
        - Si N=7 → 0, 1, 2, 3, 4, 5, 6, 0, 1, 2, 3, 4,
      - Espacio de claves no uniforme 10,20,30,40,50,60,70,80,90,100,…
        - Si N=10 → 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,…
        - Si N=4 → 2, 0, 2, 0, 2, 0, 2, 0, 2, 0,…
        - Si N=7 → 3, 6, 2, 5, 1, 4, 0, 3, 6, 2,…
      - En general, la sucesión de valores *k*M (k≥1) se transformará, módulo N, en N/mcd(M,N) valores diferentes. Por tanto, si N es primo y N≠M, se maximiza el rango de valores resultantes: cualquier valor en 0 .. N-1
    - Ejemplos N=23
      - 1679 % 23 = 0
      - 4567 % 23 = 13
      - 8471 % 23 = 7
      - 0435 % 23 = 21
      - 5033 % 23 = 19

Implementaciones con tablas dispersas: Funciones de localización

- Técnicas para valores ordinales (enteros …) (cont)
  - Mitad del cuadrado: Se eleva el número al cuadrado y se cogen las cifras centrales
    - Ejemplos:
      - $709^2 = 502681 \rightarrow 26$
      - $456^2 = 207936 \rightarrow 79$
      - $105^2 = 011025 \rightarrow 10$
      - $879^2 = 772641 \rightarrow 26$
      - $619^2 = 383161 \rightarrow 31$
  - Truncamiento: seleccionar ciertos dígitos e ignorar el resto
    - Ejemplo cogiendo los dígitos segundo, cuarto y sexto:
      - $5700931 \rightarrow 703$
      - $3498610 \rightarrow 481$
      - $0056241 \rightarrow 064$
      - $9134720 \rightarrow 142$

Implementaciones con tablas dispersas: Funciones de localización

- Técnicas para valores ordinales (enteros …) (cont)
  - Plegamiento: dividir el número en partes, y realizar operaciones con ellas (normalmente sumas o multiplicaciones)
    - Ejemplo: se divide el número en bloques de dos dígitos, y estos se suman:
      - 570093 → 57 + 00 + 93 = 150
      - 349861 → 34 + 98 + 61 = 193
      - 005624 → 00 + 56 + 24 = 80
      - 913472 → 91 + 34 + 72 = 197
      - 517492 → 51 + 74 + 92 = 217
  - Operaciones de bits. Uso de operadores de bits para mezclar fragmentos del código (p.e., <<, desplazamiento a la izquierda, y ^, x-or)
    - Ejemplos :
      - 570093 → (570 << 3) ^ 093 = 4493
      - 349861 → (349 << 3) ^ 861 = 2485
      - 005624 → (005 << 3) ^ 624 = 600
      - 913472 → (913 << 3) ^ 472 = 7504
      - 517492 → (517 << 3) ^ 492 = 4548

Implementaciones con tablas dispersas: Funciones de localización

- Técnicas para cadenas
  - La ya propuesta ((int)s.back())%N
    - No se comporta demasiado bien. Por ejemplo, si las cadenas son alfanuméricas, el código es ASCII, y la tabla es grande, todos los valores se aglutinarán en el rango 48-122 (que es el rango de códigos de las letras)
  - Una posible forma de evitar el problema es tener en cuenta los códigos de todos los caracteres:

$$(\sum_{i=0}^{s.size()-1}(int)s[i])\%N$$

    - El valor de la suma sigue sin ser demasiado grande (suponiendo caracteres con códigos en el rango bajo). Si la tabla es muy grande, los pares clave-valor tenderán a acumularse en la parte inicial del array.
  - Otra alternativa es considerar los caracteres como dígitos de un número expresado en cierta base *B*:

$$(\sum_{i=0}^{s.size()-1}((int)s[i]) * B^i)\%N$$

Implementaciones con tablas dispersas: Funciones de localización

- En la implementación propuesta, las funciones de localización se suponen definidas siempre como:

  `l(k) = h(k)%N`

  donde N es el tamaño del array interno (dicho tamaño puede ajustarse dinámicamente)

- Si:
  - El rango de valores de **h** es grande comparado con N (p.e., todo `unsigned int`)
  - Y, además, **h** distribuye uniformemente las claves

  los índices resultantes se distribuirán también uniformemente en 0..N-1 (independientemente del valor N)

- Algunos ejemplos (para tipos básicos):

```
inline unsigned int h(unsigned int clave) {
   return clave;}
inline unsigned int h(int clave) {
   return (unsigned int) clave;}
inline unsigned int h(char clave) {
   return clave;}
```

Estas funciones darán buenos resultados si la aparición de las claves sigue una distribución uniforme.

Implementaciones con tablas dispersas: Funciones de localización

- Algunos ejemplos (para tipos básicos) (cont):

```
// Fowler/Noll/Vo (FNV) -- adaptada de http://papa.bretmulvey.com/post/124027987928/hash-functions
inline unsigned int h(std::string clave) {
    const unsigned int p = 16777619; // primo grande
    unsigned int hash = 2166136261;  // valor inicial
    for (unsigned int i=0; i<clave.size(); i++)
      hash = (hash ^ clave[i]) * p; // ^ es xor binario
    // mezcla final
    hash += hash << 13;
    hash ^= hash >> 7;
    hash += hash << 3;
    hash ^= hash >> 17;
    hash += hash << 5;
    return hash;
}
```

Implementaciones con tablas dispersas: Funciones de localización

- Algunos ejemplos (cont). ¿Y si la clave es un objeto de una clase definida por el programador?
  - Función genérica para objetos de clases que implementan un método `hashcode`

```
template<class C>
unsigned int h(const C &clave) {
 return clave.hashcode();
}
```

  - Con esta función, es posible definir clases / plantillas que puedan servir como claves en el diccionario:

```
template<class A, class B>
class Pareja {
  A _prim;
  B _seg;
public:
   ...
 unsigned int hashcode() const {
            return ::h(_prim) * 1021 + ::h(_seg);
 };
   ...
};
```

Implementación del ejemplo motivador

- Para que la lista de términos salga ordenada, será necesario utilizar la implementación basada en árboles de búsqueda

```
void refsCruzadas(const Lista<string> &texto) {
  // Construimos um diccionario a partir del recorrido de la lista de cadenas
  Lista<string>::ConstIterator it(texto.cbegin());
  Diccionario<string, int> refs;
  while (it != texto.cend()) {
     Diccionario<string, int>::Iterator p = refs.busca(it.elem());
     if (p == refs.end())
          refs.inserta(it.elem(), 1);
     else
          p.valor()++; // p.setVal(p.valor()+1);
     it.next();
  }
  // Y ahora escribimos
  Diccionario<string, int>::ConstIterator ita = refs.cbegin();
  while (ita != refs.cend()) {
     cout << ita.clave() << " " << ita.valor() << endl;
     ita.next();
  }
}
```

Bibliografía del tema

- Apuntes ISBN 978-84-697-0852-1
- Fundamentals of Data Structures in C++ / Horowitz, Sahni & Mehta / Computer Science Press / 1995
- Data abstraction & Problem Solving with C++: Walls and Mirrors, 6th edition / Frank Carrano & Timothy Henry / Pearson, 2013
- ADTs, Data Structures, and Problem Solving with C++, 2nd edition / Larry Nyhoff / Pearson – Prentice Hall, 2005
- Estructuras de datos y métodos algorítmicos: Ejercicios resueltos / Martí Oliet, Ortega Mallén y Verdejo López / Pearson – Prentice Hall, 2010

Acerca de Creative Commons

Licencia CC (Creative Commons)

Este tipo de licencias ofrecen algunos derechos a terceras personas bajo ciertas condiciones.

Este documento tiene establecidas las siguientes:


Reconocimiento (*Attribution*):
En cualquier explotación de la obra autorizada por la licencia hará falta reconocer la autoría.


No comercial (*Non commercial*):
La explotación de la obra queda limitada a usos no comerciales.


Compartir igual (*Share alike*):
La explotación autorizada incluye la creación de obras derivadas siempre que mantengan la misma licencia al ser divulgadas.

En http://es.creativecommons.org/ y http://creativecommons.org/ puedes saber más de Creative Commons.