

UNIVERSIDAD NACIONAL DE INGENIERÍA

Estructuras de Datos

Árboles binarios de búsqueda

Recorridos, mapas ordenados y por qué la altura lo decide todo

Gerson Garrido

Semana 10

Hoy

1. Lo que una tabla hash no da
2. Invariante: en cada nodo, *todas* las claves
3. Buscar e insertar: bajar desde la raíz
4. Recorridos, y destruir un árbol sin agotar la pila
5. Borrar: tres casos, con el sucesor
6. Consultas ordenadas: piso, techo y rango
7. El puntero al padre
8. La altura: entre log n y n
9. `std::map` frente a `std::unordered_map`

> **Lectura**
>
> Apuntes de la semana 10. Código en `code/week-10`.

SECCIÓN 01

Lo que una tabla hash no da

01



La misma pregunta a los dos diccionarios

100 000 claves: 0, 3, 6, 9… ¿Cuál es la menor clave $\geq$ 100 000?

terminal

```
$ build/map_queries
std::map, techo de 100000: 100002, comparaciones: 16
std::unordered_map, techo de 100000: 100002, claves visitadas: 100000
std::map, piso de 100000: 99999, comparaciones: 16
std::map, claves en [100000, 100030): 10, comparaciones: 32
std::map, primeras: 0 3 6 9 12
std::unordered_map, primeras: 232704 299571 35535 21798 182190
```

La misma pregunta a los dos diccionarios

100 000 claves: 0, 3, 6, 9… ¿Cuál es la menor clave $\geq$ 100 000?

terminal

```
$ build/map_queries
std::map, techo de 100000: 100002, comparaciones: 16
std::unordered_map, techo de 100000: 100002, claves visitadas: 100000
std::map, piso de 100000: 99999, comparaciones: 16
std::map, claves en [100000, 100030): 10, comparaciones: 32
std::map, primeras: 0 3 6 9 12
std::unordered_map, primeras: 232704 299571 35535 21798 182190
```

- **16 comparaciones frente a mirar todas las claves.**

La misma pregunta a los dos diccionarios

100 000 claves: 0, 3, 6, 9… ¿Cuál es la menor clave $\geq$ 100 000?

terminal

```
$ build/map_queries
std::map, techo de 100000: 100002, comparaciones: 16
std::unordered_map, techo de 100000: 100002, claves visitadas: 100000
std::map, piso de 100000: 99999, comparaciones: 16
std::map, claves en [100000, 100030): 10, comparaciones: 32
std::map, primeras: 0 3 6 9 12
std::unordered_map, primeras: 232704 299571 35535 21798 182190
```

- **16 comparaciones frente a mirar todas las claves.**
- La tabla no es mala, pero no guarda ningún orden.

La misma pregunta a los dos diccionarios

100 000 claves: 0, 3, 6, 9… ¿Cuál es la menor clave $\geq$ 100 000?

terminal

```
$ build/map_queries
std::map, techo de 100000: 100002, comparaciones: 16
std::unordered_map, techo de 100000: 100002, claves visitadas: 100000
std::map, piso de 100000: 99999, comparaciones: 16
std::map, claves en [100000, 100030): 10, comparaciones: 32
std::map, primeras: 0 3 6 9 12
std::unordered_map, primeras: 232704 299571 35535 21798 182190
```

- **16 comparaciones frente a mirar todas las claves.**
- La tabla no es mala, pero no guarda ningún orden.
- Hoy: una estructura que conserva el orden, y que se puede modificar sin mover nada.

Las preguntas de hoy

Consulta	Devuelve
mínimo, máximo	la clave menor, la mayor
piso de x *(floor)*	la mayor clave $\leq$ x
techo de x *(ceiling)*	la menor clave $\geq$ x
rango [a, b)	las k con a $\leq$ k $<$ b
recorrido	todas, en orden

Las preguntas de hoy

Consulta	Devuelve
mínimo, máximo	la clave menor, la mayor
piso de x *(floor)*	la mayor clave $\leq$ x
techo de x *(ceiling)*	la menor clave $\geq$ x
rango [a, b)	las k con a $\leq$ k $<$ b
recorrido	todas, en orden

¿Y un arreglo ordenado?

Con `std::lower_bound` consulta en O(log n). Pero insertar o borrar mueve Θ(n).

Las preguntas de hoy

Consulta	Devuelve
mínimo, máximo	la clave menor, la mayor
piso de x *(floor)*	la mayor clave $\leq$ x
techo de x *(ceiling)*	la menor clave $\geq$ x
rango [a, b)	las k con a $\leq$ k $<$ b
recorrido	todas, en orden

¿Y un arreglo ordenado?

Con `std::lower_bound` consulta en O(log n). Pero insertar o borrar mueve Θ(n).

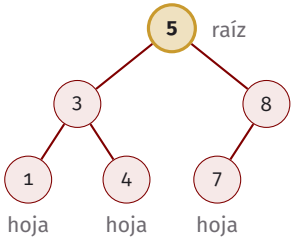
Queremos consultar como un arreglo ordenado y modificar como una lista.

SECCIÓN 02

El árbol binario de búsqueda

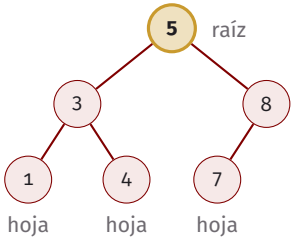
02

Vocabulario



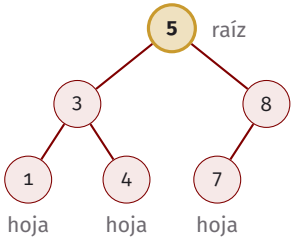
- **Árbol binario:** cada nodo tiene como máximo dos hijos, y todos menos la raíz tienen un padre.
- **Subárbol** de x: x y todo lo que se alcanza bajando desde x.

Vocabulario



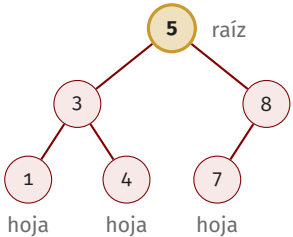
- **Árbol binario:** cada nodo tiene como máximo dos hijos, y todos menos la raíz tienen un padre.
- **Subárbol** de x: x y todo lo que se alcanza bajando desde x.
- **Profundidad** *(depth)*: aristas desde la raíz. El 7 tiene 2.

Vocabulario



- **Árbol binario:** cada nodo tiene como máximo dos hijos, y todos menos la raíz tienen un padre.
- **Subárbol** de x: x y todo lo que se alcanza bajando desde x.
- **Profundidad** *(depth)*: aristas desde la raíz. El 7 tiene 2.
- **Altura** *(height)*: la mayor profundidad. Aquí, 2.

Vocabulario



- **Árbol binario:** cada nodo tiene como máximo dos hijos, y todos menos la raíz tienen un padre.
- **Subárbol** de x: x y todo lo que se alcanza bajando desde x.
- **Profundidad** *(depth)*: aristas desde la raíz. El 7 tiene 2.
- **Altura** *(height)*: la mayor profundidad. Aquí, 2.

Convenio del curso

Se cuentan aristas. Un solo nodo: altura 0. Vacío: altura -1. Otros libros cuentan nodos.

La invariante

Árbol binario de búsqueda ***(binary search tree)***

Para cada nodo x, **todas** las claves del subárbol izquierdo de x son menores que la de x, y **todas** las del derecho son mayores.

La invariante

Árbol binario de búsqueda ***(binary search tree)***

Para cada nodo x, **todas** las claves del subárbol izquierdo de x son menores que la de x, y **todas** las del derecho son mayores.

- La raíz parte las claves en dos, como el medio en una búsqueda binaria.

La invariante

Árbol binario de búsqueda ***(binary search tree)***

Para cada nodo x, **todas** las claves del subárbol izquierdo de x son menores que la de x, y **todas** las del derecho son mayores.

- La raíz parte las claves en dos, como el medio en una búsqueda binaria.
- Una comparación descarta un subárbol entero, sin mirarlo.

La invariante

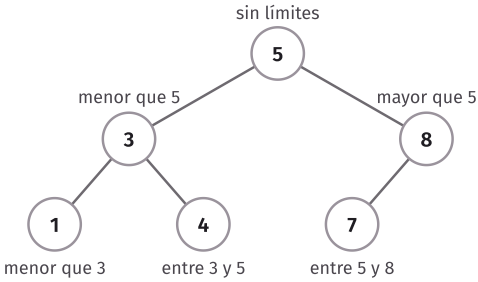
Árbol binario de búsqueda ***(binary search tree)***

Para cada nodo x, **todas** las claves del subárbol izquierdo de x son menores que la de x, y **todas** las del derecho son mayores.

- La raíz parte las claves en dos, como el medio en una búsqueda binaria.
- Una comparación descarta un subárbol entero, sin mirarlo.
- Todas: no basta con mirar a los hijos.

Cada nodo hereda sus límites

CUMPLE LA INVARIANTE

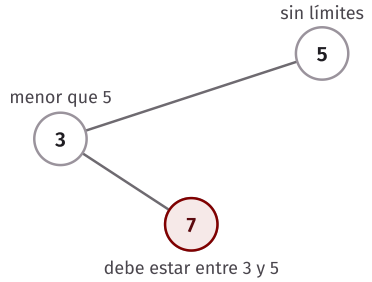


Cada nodo hereda los límites de sus antepasados

`en orden: 1 3 4 5 7 8`

ordenadas

PARECE CORRECTO, Y ESTÁ ROTO



7 > 3: bien con su padre. Pero está a la izquierda del 5, y 7 > 5

`en orden: 3 7 5`

no ordenadas, y contains(7) dice que no

Revisar solo a los hijos no alcanza

terminal

```
$ g++ -std=c++20 errors/local_check.cpp -o build/local_check && build/local_check
revision local: todo bien
contains(7): no
```

Error común

Un `check_invariants` que compara cada nodo solo con sus hijos. El programa que debe avisarles les da la razón, y la clave 7, que está en el árbol, no se encuentra.

Revisar solo a los hijos no alcanza

terminal

```
$ g++ -std=c++20 errors/local_check.cpp -o build/local_check && build/local_check
revision local: todo bien
contains(7): no
```

Error común

Un `check_invariants` que compara cada nodo solo con sus hijos. El programa que debe avisarles les da la razón, y la clave 7, que está en el árbol, no se encuentra.

Hay que bajar con **límites**: cada nodo debe quedar entre el mayor antepasado por el que giramos a la derecha y el menor por el que giramos a la izquierda.

Un conjunto, y su comprobación

bst_set.h

```
1  struct Node {
2      K key;
3      Node* left = nullptr;
4      Node* right = nullptr;
5  };
```

bst_set.h

```
1  static std::size_t count_checked(const Node*
       node, const K* low, const K* high) {
2      if (node == nullptr) {
3          return 0;
4      }
5      assert(low == nullptr || *low < node->key)
       ;
6      assert(high == nullptr || node->key < *
       high);
7      return 1 + count_checked(node->left, low,
       &node->key)
8               + count_checked(node->right, &
       node->key, high);
9  }
```

Un conjunto, y su comprobación

bst_set.h

```
1  struct Node {
2      K key;
3      Node* left = nullptr;
4      Node* right = nullptr;
5  };
```

- `K` solo necesita `<`. Dos claves son la misma si ni `a < b` ni `b < a`.
- Sin claves repetidas.

bst_set.h

```
1  static std::size_t count_checked(const Node*
       node, const K* low, const K* high) {
2      if (node == nullptr) {
3          return 0;
4      }
5      assert(low == nullptr || *low < node->key)
       ;
6      assert(high == nullptr || node->key < *
       high);
7      return 1 + count_checked(node->left, low,
       &node->key)
8               + count_checked(node->right, &
       node->key, high);
9  }
```

Un conjunto, y su comprobación

bst_set.h

```
1  struct Node {
2      K key;
3      Node* left = nullptr;
4      Node* right = nullptr;
5  };
```

- `K` solo necesita `<`. Dos claves son la misma si ni `a < b` ni `b < a`.
- Sin claves repetidas.
- Con `size_`, cada nodo se cuenta una vez.

bst_set.h

```
1  static std::size_t count_checked(const Node*
       node, const K* low, const K* high) {
2      if (node == nullptr) {
3          return 0;
4      }
5      assert(low == nullptr || *low < node->key)
       ;
6      assert(high == nullptr || node->key < *
       high);
7      return 1 + count_checked(node->left, low,
       &node->key)
8               + count_checked(node->right, &
       node->key, high);
9  }
```

SECCIÓN 03

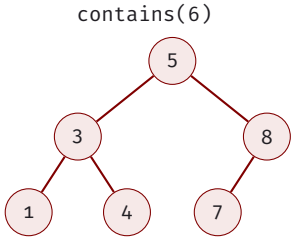
Buscar e insertar

03

Buscar: bajar desde la raíz

bst_set.h

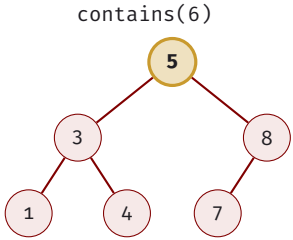
```cpp
bool contains(const K& key) const {
    const Node* node = root_;
    while (node != nullptr) {
        if (key < node->key) {
            node = node->left;
        } else if (node->key < key) {
            node = node->right;
        } else {
            return true;
        }
    }
    return false;
}
```



Buscar: bajar desde la raíz

bst_set.h

```cpp
bool contains(const K& key) const {
    const Node* node = root_;
    while (node != nullptr) {
        if (key < node->key) {
            node = node->left;
        } else if (node->key < key) {
            node = node->right;
        } else {
            return true;
        }
    }
    return false;
}
```

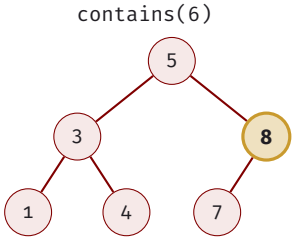


5 < 6: a la derecha.

Buscar: bajar desde la raíz

bst_set.h

```cpp
bool contains(const K& key) const {
    const Node* node = root_;
    while (node != nullptr) {
        if (key < node->key) {
            node = node->left;
        } else if (node->key < key) {
            node = node->right;
        } else {
            return true;
        }
    }
    return false;
}
```

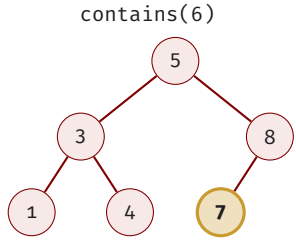


6 < 8: a la izquierda.

Buscar: bajar desde la raíz

bst_set.h

```cpp
bool contains(const K& key) const {
    const Node* node = root_;
    while (node != nullptr) {
        if (key < node->key) {
            node = node->left;
        } else if (node->key < key) {
            node = node->right;
        } else {
            return true;
        }
    }
    return false;
}
```

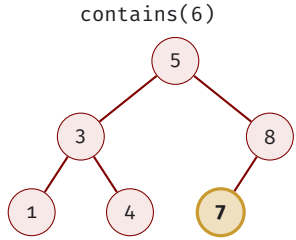


6 < 7: a la izquierda. Nulo: no está.

Buscar: bajar desde la raíz

bst_set.h

```cpp
bool contains(const K& key) const {
    const Node* node = root_;
    while (node != nullptr) {
        if (key < node->key) {
            node = node->left;
        } else if (node->key < key) {
            node = node->right;
        } else {
            return true;
        }
    }
    return false;
}
```



Cada paso baja un nivel. El ciclo no gasta pila.

El costo

Idea clave

Buscar cuesta O(altura), y Θ(altura) en el peor caso, cuando se llega al nodo más profundo.

El costo

Idea clave

Buscar cuesta O(altura), y Θ(altura) en el peor caso, cuando se llega al nodo más profundo.

- No cuesta “O(log n)”. Eso solo es cierto si la altura es logarítmica.

El costo

Idea clave

Buscar cuesta O(altura), y Θ(altura) en el peor caso, cuando se llega al nodo más profundo.

- No cuesta “O(log n)”. Eso solo es cierto si la altura es logarítmica.
- Nada lo garantiza todavía.

El costo

Idea clave

Buscar cuesta O(altura), y Θ(altura) en el peor caso, cuando se llega al nodo más profundo.

- No cuesta “O(log n)”. Eso solo es cierto si la altura es logarítmica.
- Nada lo garantiza todavía.
- Los subárboles descartados pueden tener millones de nodos: no importa.

Insertar: buscar, y colgar una hoja

bst_set.h

```
1  bool insert(const K& key) {
2      Node** link = &root_;
3      while (*link != nullptr) {
4          if (key < (*link)->key) {
5              link = &(*link)->left;
6          } else if ((*link)->key < key) {
7              link = &(*link)->right;
8          } else {
9              return false;
10         }
11     }
12     *link = new Node{key};
13     size_++;
14     return true;
15 }
```

Insertar: buscar, y colgar una hoja

bst_set.h

```
1  bool insert(const K& key) {
2      Node** link = &root_;
3      while (*link != nullptr) {
4          if (key < (*link)->key) {
5              link = &(*link)->left;
6          } else if ((*link)->key < key) {
7              link = &(*link)->right;
8          } else {
9              return false;
10         }
11     }
12     *link = new Node{key};
13     size_++;
14     return true;
15 }
```

- Un árbol solo crece por sus hojas.

Insertar: buscar, y colgar una hoja

bst_set.h

```
1  bool insert(const K& key) {
2      Node** link = &root_;
3      while (*link != nullptr) {
4          if (key < (*link)->key) {
5              link = &(*link)->left;
6          } else if ((*link)->key < key) {
7              link = &(*link)->right;
8          } else {
9              return false;
10         }
11     }
12     *link = new Node{key};
13     size_++;
14     return true;
15 }
```

- Un árbol solo crece por sus hojas.
- El enlace a cambiar es `root_`, o un `left`, o un `right`. Todos son `Node*`, así que `link`, un `Node**`, unifica los tres casos.

Insertar: buscar, y colgar una hoja

bst_set.h

```
1  bool insert(const K& key) {
2      Node** link = &root_;
3      while (*link != nullptr) {
4          if (key < (*link)->key) {
5              link = &(*link)->left;
6          } else if ((*link)->key < key) {
7              link = &(*link)->right;
8          } else {
9              return false;
10         }
11     }
12     *link = new Node{key};
13     size_++;
14     return true;
15 }
```

- Un árbol solo crece por sus hojas.
- El enlace a cambiar es `root_`, o un `left`, o un `right`. Todos son `Node*`, así que `link`, un `Node**`, unifica los tres casos.
- Costo: O(altura), más una reserva.

Insertar: buscar, y colgar una hoja

bst_set.h

```
1  bool insert(const K& key) {
2      Node** link = &root_;
3      while (*link != nullptr) {
4          if (key < (*link)->key) {
5              link = &(*link)->left;
6          } else if ((*link)->key < key) {
7              link = &(*link)->right;
8          } else {
9              return false;
10         }
11     }
12     *link = new Node{key};
13     size_++;
14     return true;
15 }
```

- Un árbol solo crece por sus hojas.
- El enlace a cambiar es `root_`, o un `left`, o un `right`. Todos son `Node*`, así que `link`, un `Node**`, unifica los tres casos.
- Costo: O(altura), más una reserva.

Error común

Comparar con <= para permitir repetidos. Rompe la invariante.

El orden de inserción decide la forma

orden

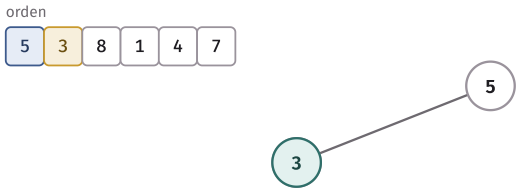
5	3	8	1	4	7



Árbol vacío: el 5 se vuelve la raíz

Las mismas claves en otro orden dan otro árbol. Ahí estará el problema.

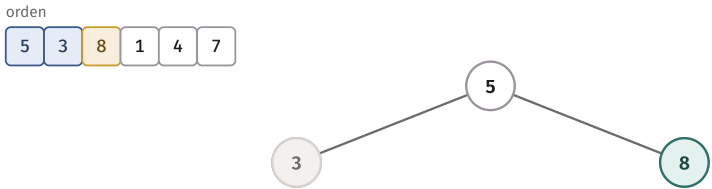
El orden de inserción decide la forma



3 < 5: a la izquierda. Ese enlace es nulo, y ahí se cuelga

Las mismas claves en otro orden dan otro árbol. Ahí estará el problema.

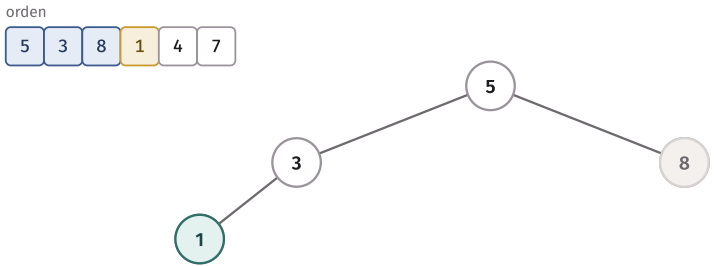
El orden de inserción decide la forma



8 > 5: a la derecha. Otro enlace nulo, otra hoja

Las mismas claves en otro orden dan otro árbol. Ahí estará el problema.

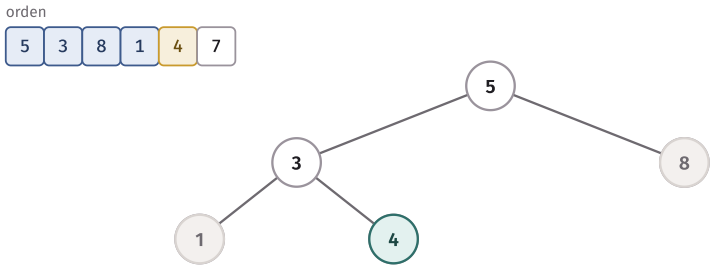
El orden de inserción decide la forma



1 < 5, y todo lo que está a la derecha del 5 ya no se mira. 1 < 3: hoja nueva

Las mismas claves en otro orden dan otro árbol. Ahí estará el problema.

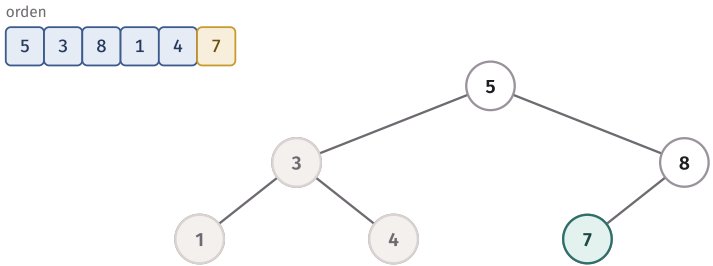
El orden de inserción decide la forma



4 < 5, y luego 4 > 3: queda a la derecha del 3, y a la izquierda del 5

Las mismas claves en otro orden dan otro árbol. Ahí estará el problema.

El orden de inserción decide la forma



7 > 5, y luego 7 < 8: hijo izquierdo del 8. Nada más cambió: el árbol crece por sus hojas

Las mismas claves en otro orden dan otro árbol. Ahí estará el problema.

SECCIÓN 04

Recorridos

04

Tres recorridos, una línea de diferencia

en orden
```
static void walk_in(const Node*
     node, std::vector<K>& keys) {
    if (node == nullptr) {
        return;
    }
    walk_in(node->left, keys);
    keys.push_back(node->key);
    walk_in(node->right, keys);
}
```

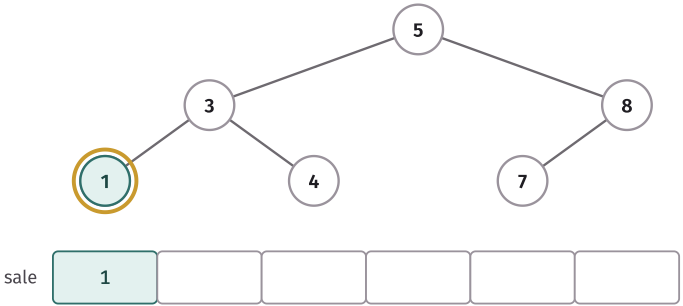
Cada uno cuesta $\Theta(n)$.

en orden	izq., **nodo**, der.
preorden	**nodo**, izq., der.
postorden	izq., der., **nodo**

terminal
```
$ build/tree_demo
forma: 5(3(1,4),8(7,_))
en orden: 1 3 4 5 7 8
preorden: 5 3 1 4 8 7
postorden: 1 4 3 7 8 5
altura: 2
```

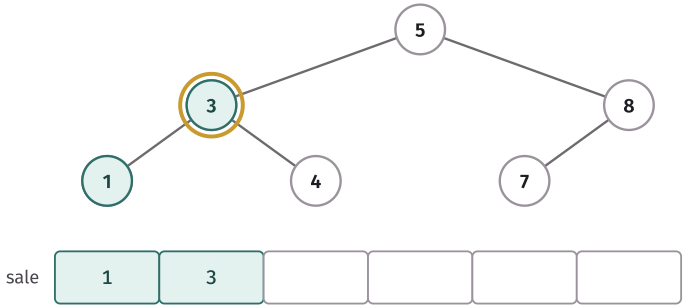
`forma`: clave(izquierdo,derecho), con _ si falta un hijo.

El recorrido en orden, paso a paso



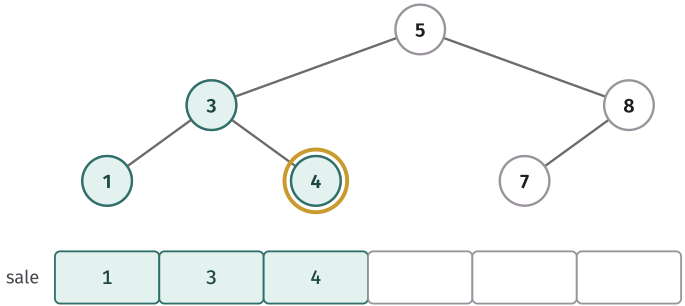
Primero el subárbol izquierdo, y luego otra vez a la izquierda: 5, 3, 1. Sale el 1

El recorrido en orden, paso a paso



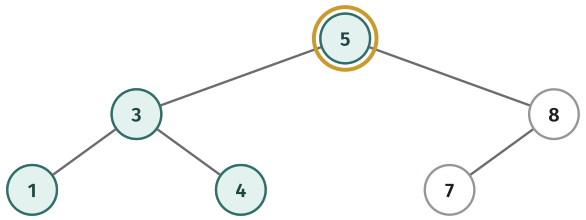
El subárbol izquierdo del 3 ya terminó. Sale el 3

El recorrido en orden, paso a paso



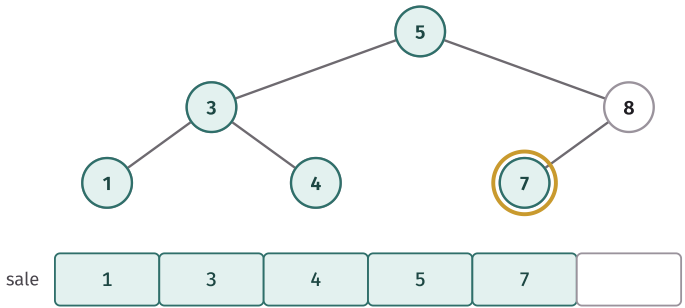
Ahora su subárbol derecho: sale el 4

El recorrido en orden, paso a paso



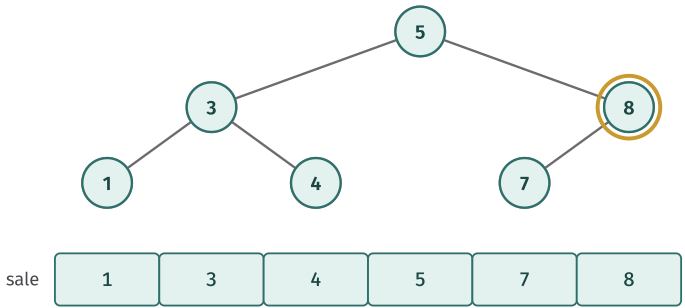
El subárbol izquierdo del 5 terminó entero. Sale el 5, después de todo lo menor

El recorrido en orden, paso a paso



Baja por el subárbol derecho, hasta el extremo izquierdo: sale el 7

El recorrido en orden, paso a paso



Sale el 8. Las claves salieron ordenadas, sin una sola comparación

En orden: las claves ordenadas

¿Por qué? Tomen dos claves a < b, y el nodo x más profundo que tiene a las dos en su subárbol.

En orden: las claves ordenadas

¿Por qué? Tomen dos claves a < b, y el nodo x más profundo que tiene a las dos en su subárbol.

- a es la clave de x: b está a su derecha. **Nodo** antes que **derecho**.

En orden: las claves ordenadas

¿Por qué? Tomen dos claves a < b, y el nodo x más profundo que tiene a las dos en su subárbol.

- a es la clave de x: b está a su derecha. **Nodo** antes que **derecho**.
- b es la clave de x: a está a su izquierda. **Izquierdo** antes que **nodo**.

En orden: las claves ordenadas

¿Por qué? Tomen dos claves a < b, y el nodo x más profundo que tiene a las dos en su subárbol.

- a es la clave de x: b está a su derecha. **Nodo** antes que **derecho**.
- b es la clave de x: a está a su izquierda. **Izquierdo** antes que **nodo**.
- a a la izquierda y b a la derecha. **Izquierdo** antes que **derecho**.

En orden: las claves ordenadas

¿Por qué? Tomen dos claves a < b, y el nodo x más profundo que tiene a las dos en su subárbol.

- a es la clave de x: b está a su derecha. **Nodo** antes que **derecho**.
- b es la clave de x: a está a su izquierda. **Izquierdo** antes que **nodo**.
- a a la izquierda y b a la derecha. **Izquierdo** antes que **derecho**.

No hay más casos: si estuvieran las dos en el mismo subárbol, x no sería el más profundo. Sin inducción: solo la invariante en x.

En orden: las claves ordenadas

¿Por qué? Tomen dos claves a < b, y el nodo x más profundo que tiene a las dos en su subárbol.

- a es la clave de x: b está a su derecha. **Nodo** antes que **derecho**.
- b es la clave de x: a está a su izquierda. **Izquierdo** antes que **nodo**.
- a a la izquierda y b a la derecha. **Izquierdo** antes que **derecho**.

No hay más casos: si estuvieran las dos en el mismo subárbol, x no sería el más profundo. Sin inducción: solo la invariante en x.

> **Idea clave**
>
> El árbol es una forma de guardar una secuencia ordenada. Sucesor, piso, techo y rango encuentran posiciones de esa secuencia sin recorrerla toda.

Preorden y postorden: ¿para qué?

Preorden: cada nodo antes que sus descendientes.

Preorden y postorden: ¿para qué?

Preorden: cada nodo antes que sus descendientes.

- Si se inserta en preorden, cada nodo encuentra a sus antepasados: **sale la misma forma**.

Preorden y postorden: ¿para qué?

Preorden: cada nodo antes que sus descendientes.

- Si se inserta en preorden, cada nodo encuentra a sus antepasados: **sale la misma forma**.
- Sirve para guardar, copiar y probar formas.

Preorden y postorden: ¿para qué?

Preorden: cada nodo antes que sus descendientes.

- Si se inserta en preorden, cada nodo encuentra a sus antepasados: **sale la misma forma**.
- Sirve para guardar, copiar y probar formas.
- En orden no sirve: todas las formas con las mismas claves dan lo mismo.

Preorden y postorden: ¿para qué?

Preorden: cada nodo antes que sus descendientes.

- Si se inserta en preorden, cada nodo encuentra a sus antepasados: **sale la misma forma**.
- Sirve para guardar, copiar y probar formas.
- En orden no sirve: todas las formas con las mismas claves dan lo mismo.

Postorden: cada nodo después de sus descendientes.

- Cuando la respuesta del nodo depende de la de sus hijos: tamaño, altura.

Preorden y postorden: ¿para qué?

Preorden: cada nodo antes que sus descendientes.

- Si se inserta en preorden, cada nodo encuentra a sus antepasados: **sale la misma forma**.
- Sirve para guardar, copiar y probar formas.
- En orden no sirve: todas las formas con las mismas claves dan lo mismo.

Postorden: cada nodo después de sus descendientes.

- Cuando la respuesta del nodo depende de la de sus hijos: tamaño, altura.
- **Liberar la memoria:** primero los hijos.

SECCIÓN 05

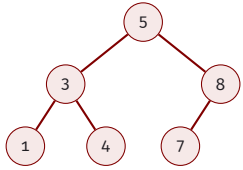
Destruir un árbol

05

Destruir: postorden

recursive_destroy.cpp

```cpp
void destroy(Node* node) {
    if (node == nullptr) {
        return;
    }
    destroy(node->left);
    destroy(node->right);
    delete node;
}
```

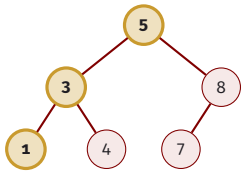


En oro: las llamadas activas en la pila.

Destruir: postorden

recursive_destroy.cpp

```cpp
void destroy(Node* node) {
    if (node == nullptr) {
        return;
    }
    destroy(node->left);
    destroy(node->right);
    delete node;
}
```

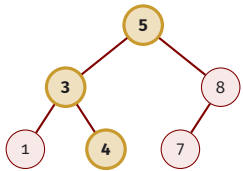


Pila: 5, 3, 1. Se libera 1.

Destruir: postorden

recursive_destroy.cpp

```cpp
void destroy(Node* node) {
    if (node == nullptr) {
        return;
    }
    destroy(node->left);
    destroy(node->right);
    delete node;
}
```

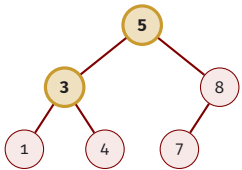


Pila: 5, 3, 4. Se libera 4.

Destruir: postorden

recursive_destroy.cpp

```cpp
void destroy(Node* node) {
    if (node == nullptr) {
        return;
    }
    destroy(node->left);
    destroy(node->right);
    delete node;
}
```

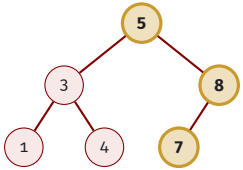


Pila: 5, 3. Se libera 3.

Destruir: postorden

recursive_destroy.cpp

```cpp
void destroy(Node* node) {
    if (node == nullptr) {
        return;
    }
    destroy(node->left);
    destroy(node->right);
    delete node;
}
```

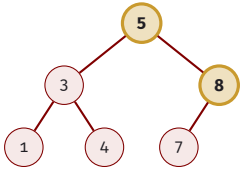


Pila: 5, 8, 7. Se libera 7.

Destruir: postorden

recursive_destroy.cpp

```cpp
void destroy(Node* node) {
    if (node == nullptr) {
        return;
    }
    destroy(node->left);
    destroy(node->right);
    delete node;
}
```

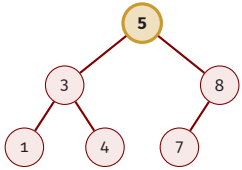


Pila: 5, 8. Se libera 8.

Destruir: postorden

recursive_destroy.cpp

```cpp
void destroy(Node* node) {
    if (node == nullptr) {
        return;
    }
    destroy(node->left);
    destroy(node->right);
    delete node;
}
```



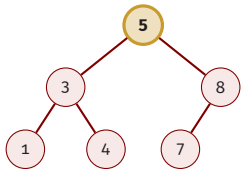
Pila: 5. Se libera 5. Nunca hubo más de altura + 1.

Destruir: postorden

recursive_destroy.cpp

```cpp
void destroy(Node* node) {
    if (node == nullptr) {
        return;
    }
    destroy(node->left);
    destroy(node->right);
    delete node;
}
```

Correcta, corta, y puede tumbar el programa.



Pila: 5. Se libera 5. Nunca hubo más de altura + 1.

Una llamada por nivel

La pila aquí es de 8 MB. Una cadena de un millón de nodos:

terminal

```
$ g++ -std=c++20 -g -fsanitize=address,undefined errors/recursive_destroy.cpp -o build/recursive_destroy && build/recursive_destroy 1000000
AddressSanitizer:DEADLYSIGNAL
=================================================================
==25264==ERROR: AddressSanitizer: stack-overflow on address 0xffffce2a9ff0 (pc 0xaaaae87d1e68 bp 0xffffce2aa010 sp 0xffffce2aa010 T0)
...
SUMMARY: AddressSanitizer: stack-overflow errors/recursive_destroy.cpp:12 in destroy(Node*)
```

Una llamada por nivel

La pila aquí es de 8 MB. Una cadena de un millón de nodos:

terminal

```
$ g++ -std=c++20 -g -fsanitize=address,undefined errors/recursive_destroy.cpp -o build/recursive_destroy && build/recursive_destroy 1000000
AddressSanitizer:DEADLYSIGNAL
=================================================================
==25264==ERROR: AddressSanitizer: stack-overflow on address 0xffffce2a9ff0 (pc 0xaaaae87d1e68 bp 0xffffce2aa010 sp 0xffffce2aa010 T0)
...
SUMMARY: AddressSanitizer: stack-overflow errors/recursive_destroy.cpp:12 in destroy(Node*)
```

Compilación	funcionó	falló
Sin optimizar	200 000	300 000
`-O2`	600 000	1 000 000

Cambian de una máquina a otra.

Una llamada por nivel

La pila aquí es de 8 MB. Una cadena de un millón de nodos:

terminal

```
$ g++ -std=c++20 -g -fsanitize=address,undefined errors/recursive_destroy.cpp -o build/recursive_destroy && build/recursive_destroy 1000000
AddressSanitizer:DEADLYSIGNAL
=================================================================
==25264==ERROR: AddressSanitizer: stack-overflow on address 0xffffce2a9ff0 (pc 0xaaaae87d1e68 bp 0xffffce2aa010 sp 0xffffce2aa010 T0)
...
SUMMARY: AddressSanitizer: stack-overflow errors/recursive_destroy.cpp:12 in destroy(Node*)
```

Compilación	funcionó	falló
Sin optimizar	200 000	300 000
-O2	600 000	1 000 000

Cambian de una máquina a otra.

Error común

Probar solo con árboles pequeños o al azar. Falla con datos reales que vienen ordenados.

La solución: no usar la pila de llamadas

bst_set.h

```
~BstSet() {
    std::vector<Node*> pending;
    if (root_ != nullptr) {
        pending.push_back(root_);
    }
    while (!pending.empty()) {
        Node* node = pending.back();
        pending.pop_back();
        if (node->left != nullptr) {
            pending.push_back(node->left);
        }
        if (node->right != nullptr) {
            pending.push_back(node->right);
        }
        delete node;
    }
}
```

La solución: no usar la pila de llamadas

bst_set.h

```
1  ~BstSet() {
2      std::vector<Node*> pending;
3      if (root_ != nullptr) {
4          pending.push_back(root_);
5      }
6      while (!pending.empty()) {
7          Node* node = pending.back();
8          pending.pop_back();
9          if (node->left != nullptr) {
10             pending.push_back(node->left);
11         }
12         if (node->right != nullptr) {
13             pending.push_back(node->right);
14         }
15         delete node;
16     }
17 }
```

- Ya no es postorden: borra al padre antes que a los hijos.

La solución: no usar la pila de llamadas

bst_set.h

```
1  ~BstSet() {
2      std::vector<Node*> pending;
3      if (root_ != nullptr) {
4          pending.push_back(root_);
5      }
6      while (!pending.empty()) {
7          Node* node = pending.back();
8          pending.pop_back();
9          if (node->left != nullptr) {
10             pending.push_back(node->left);
11         }
12         if (node->right != nullptr) {
13             pending.push_back(node->right);
14         }
15         delete node;
16     }
17 }
```

- Ya no es postorden: borra al padre antes que a los hijos.
- Es seguro: los hijos ya se guardaron en `pending`.

La solución: no usar la pila de llamadas

bst_set.h

```
~BstSet() {
    std::vector<Node*> pending;
    if (root_ != nullptr) {
        pending.push_back(root_);
    }
    while (!pending.empty()) {
        Node* node = pending.back();
        pending.pop_back();
        if (node->left != nullptr) {
            pending.push_back(node->left);
        }
        if (node->right != nullptr) {
            pending.push_back(node->right);
        }
        delete node;
    }
}
```

- Ya no es postorden: borra al padre antes que a los hijos.
- Es seguro: los hijos ya se guardaron en `pending`.
- Cada nodo entra y sale una vez.

La solución: no usar la pila de llamadas

bst_set.h

```
1  ~BstSet() {
2      std::vector<Node*> pending;
3      if (root_ != nullptr) {
4          pending.push_back(root_);
5      }
6      while (!pending.empty()) {
7          Node* node = pending.back();
8          pending.pop_back();
9          if (node->left != nullptr) {
10             pending.push_back(node->left);
11         }
12         if (node->right != nullptr) {
13             pending.push_back(node->right);
14         }
15         delete node;
16     }
17 }
```

- Ya no es postorden: borra al padre antes que a los hijos.
- Es seguro: los hijos ya se guardaron en `pending`.
- Cada nodo entra y sale una vez.
- Molestia: reserva memoria en un destructor. El puntero al padre la eliminará.

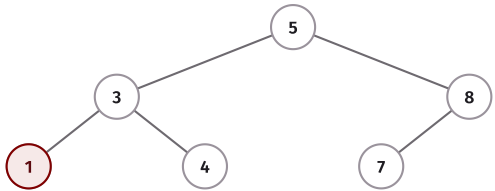
SECCIÓN 06

Borrar

06

Los dos casos fáciles

Sin hijos: borrar 1



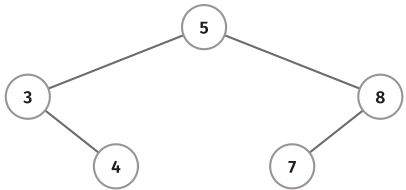
Se baja hasta el 1, como en una búsqueda: 5, 3, 1. Es una hoja

Sin hijos: se quita, y su enlace queda nulo. **Con un hijo:** el hijo sube.

¿Por qué sigue valiendo la invariante? Ese subárbol ya estaba del lado correcto de todos los antepasados del nodo borrado.

Los dos casos fáciles

Sin hijos: borrar 1



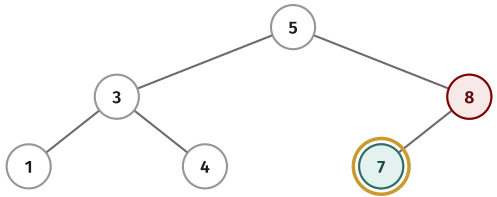
Se quita, y el enlace izquierdo del 3 queda nulo

Sin hijos: se quita, y su enlace queda nulo. **Con un hijo:** el hijo sube.

¿Por qué sigue valiendo la invariante? Ese subárbol ya estaba del lado correcto de todos los antepasados del nodo borrado.

Los dos casos fáciles

Con un hijo: borrar 8



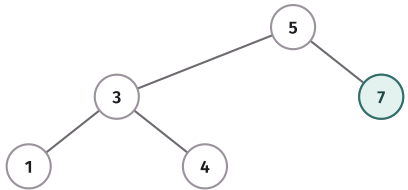
De nuevo con el árbol completo. El 8 tiene un solo hijo, el 7

Sin hijos: se quita, y su enlace queda nulo. **Con un hijo:** el hijo sube.

¿Por qué sigue valiendo la invariante? Ese subárbol ya estaba del lado correcto de todos los antepasados del nodo borrado.

Los dos casos fáciles

Con un hijo: borrar 8



El 7 sube a su lugar. Todo ese subárbol ya estaba a la derecha del 5, así que nada se rompe

Sin hijos: se quita, y su enlace queda nulo. **Con un hijo:** el hijo sube.

¿Por qué sigue valiendo la invariante? Ese subárbol ya estaba del lado correcto de todos los antepasados del nodo borrado.

Dos hijos: el sucesor

Sucesor

La siguiente clave en el recorrido en orden. Si hay subárbol derecho, es su mínimo.

No hay lugar para los dos hijos. Dejamos el nodo donde está y le cambiamos el contenido.

Dos hijos: el sucesor

Sucesor

La siguiente clave en el recorrido en orden. Si hay subárbol derecho, es su mínimo.

No hay lugar para los dos hijos. Dejamos el nodo donde está y le cambiamos el contenido.

1. **El sucesor sirve de reemplazo.** Es mayor que todo el izquierdo, y es menor que todo lo que queda en el derecho.

Dos hijos: el sucesor

Sucesor

La siguiente clave en el recorrido en orden. Si hay subárbol derecho, es su mínimo.

No hay lugar para los dos hijos. Dejamos el nodo donde está y le cambiamos el contenido.

1. **El sucesor sirve de reemplazo.** Es mayor que todo el izquierdo, y es menor que todo lo que queda en el derecho.
2. **El sucesor no tiene hijo izquierdo.** Sería una clave menor en el mismo subárbol. Quitarlo es un caso fácil.

Dos hijos: el sucesor

Sucesor

La siguiente clave en el recorrido en orden. Si hay subárbol derecho, es su mínimo.

No hay lugar para los dos hijos. Dejamos el nodo donde está y le cambiamos el contenido.

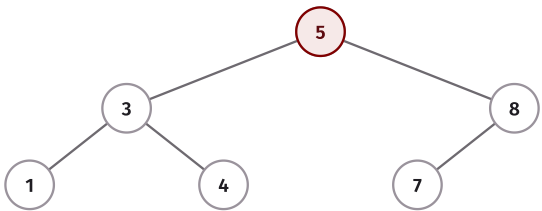
1. **El sucesor sirve de reemplazo.** Es mayor que todo el izquierdo, y es menor que todo lo que queda en el derecho.
2. **El sucesor no tiene hijo izquierdo.** Sería una clave menor en el mismo subárbol. Quitarlo es un caso fácil.

Idea clave

Copiar la clave del sucesor al nodo, y quitar **el nodo del sucesor**.

Borrar el 5

Con dos hijos: borrar 5

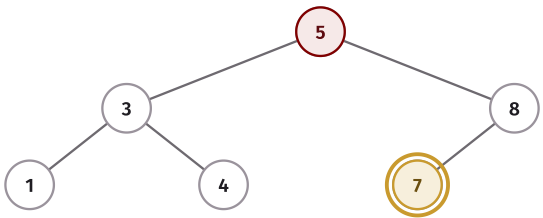


El 5 tiene dos hijos, y un solo enlace no alcanza para los dos

Costo: una bajada hasta el nodo, otra hasta el sucesor. O(altura).

Borrar el 5

Con dos hijos: borrar 5

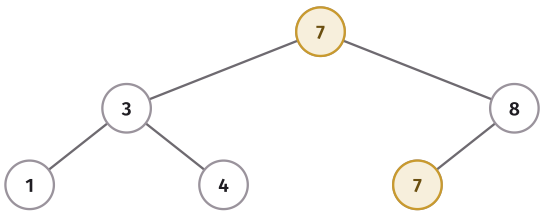


El sucesor es el mínimo de la derecha: una vez a la derecha y a la izquierda hasta el final

Costo: una bajada hasta el nodo, otra hasta el sucesor. O(altura).

Borrar el 5

Con dos hijos: borrar 5



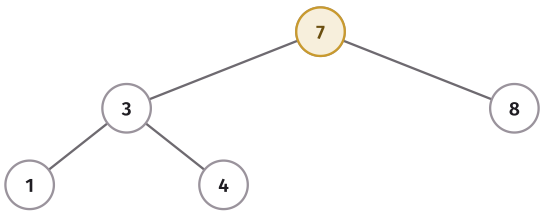
Se copia el 7 en el nodo. Es mayor que toda la izquierda y menor que lo que queda a la derecha

Costo: una bajada hasta el nodo, otra hasta el sucesor. O(altura).

Borrar el 5

Con dos hijos: borrar 5

`sin el 5: 7(3(1,4),8)`



Se quita el nodo de abajo, que no tiene hijo izquierdo: es un caso fácil. Queda un solo 7

Costo: una bajada hasta el nodo, otra hasta el sucesor. O(altura).

En código

bst_set.h

```
1   Node* node = *link;
2   if (node->left != nullptr && node->right != nullptr) {
3       // Dos hijos: el sucesor es el mínimo del subárbol derecho.
4       Node** next_link = &node->right;
5       while ((*next_link)->left != nullptr) {
6           next_link = &(*next_link)->left;
7       }
8       node->key = (*next_link)->key;
9       // Ahora el que se quita es el nodo del sucesor, que no
          tiene hijo izquierdo.
10      link = next_link;
11      node = *link;
12  }
13  *link = node->left != nullptr ? node->left : node->right;
14  delete node;
15  size_--;
16  return true;
```

En código

bst_set.h

```
1   Node* node = *link;
2   if (node->left != nullptr && node->right != nullptr) {
3       // Dos hijos: el sucesor es el mínimo del subárbol derecho.
4       Node** next_link = &node->right;
5       while ((*next_link)->left != nullptr) {
6           next_link = &(*next_link)->left;
7       }
8       node->key = (*next_link)->key;
9       // Ahora el que se quita es el nodo del sucesor, que no
          tiene hijo izquierdo.
10      link = next_link;
11      node = *link;
12  }
13  *link = node->left != nullptr ? node->left : node->right;
14  delete node;
15  size_--;
16  return true;
```

- `link` apunta al enlace del nodo que se va a quitar.

En código

bst_set.h

```
1   Node* node = *link;
2   if (node->left != nullptr && node->right != nullptr) {
3       // Dos hijos: el sucesor es el mínimo del subárbol derecho.
4       Node** next_link = &node->right;
5       while ((*next_link)->left != nullptr) {
6           next_link = &(*next_link)->left;
7       }
8       node->key = (*next_link)->key;
9       // Ahora el que se quita es el nodo del sucesor, que no
          tiene hijo izquierdo.
10      link = next_link;
11      node = *link;
12  }
13  *link = node->left != nullptr ? node->left : node->right;
14  delete node;
15  size_--;
16  return true;
```

- `link` apunta al enlace del nodo que se va a quitar.
- La misma línea sirve para los dos casos fáciles: sin hijos, queda nulo.

En código

bst_set.h

```
1   Node* node = *link;
2   if (node->left != nullptr && node->right != nullptr) {
3       // Dos hijos: el sucesor es el mínimo del subárbol derecho.
4       Node** next_link = &node->right;
5       while ((*next_link)->left != nullptr) {
6           next_link = &(*next_link)->left;
7       }
8       node->key = (*next_link)->key;
9       // Ahora el que se quita es el nodo del sucesor, que no
          tiene hijo izquierdo.
10      link = next_link;
11      node = *link;
12  }
13  *link = node->left != nullptr ? node->left : node->right;
14  delete node;
15  size_--;
16  return true;
```

- `link` apunta al enlace del nodo que se va a quitar.
- La misma línea sirve para los dos casos fáciles: sin hijos, queda nulo.
- Si el sucesor es el hijo derecho directo, el ciclo no da ninguna vuelta, y todo funciona igual.

El error común de la semana

Error común

En el borrado con dos hijos, copiar la clave del sucesor y olvidar borrar el nodo del sucesor. La búsqueda encuentra la primera copia, y el programa sigue un buen rato.

terminal

```
$ g++ -std=c++20 -g -fsanitize=address,undefined errors/forgot_successor.cpp -o build/forgot_successor && build/forgot_successor
1 3 4 7 7 8
```

`check_invariants` lo detiene: `Assertion 'low == nullptr || *low < node->key' failed.`

El error común de la semana

Error común

En el borrado con dos hijos, copiar la clave del sucesor y olvidar borrar el nodo del sucesor. La búsqueda encuentra la primera copia, y el programa sigue un buen rato.

terminal

```
$ g++ -std=c++20 -g -fsanitize=address,undefined errors/forgot_successor.cpp -o build/forgot_successor && build/forgot_successor
1 3 4 7 7 8
```

`check_invariants` lo detiene: `Assertion 'low == nullptr || *low < node->key' failed`.

- El segundo 7 tiene como límite inferior al primero.

El error común de la semana

Error común

En el borrado con dos hijos, copiar la clave del sucesor y olvidar borrar el nodo del sucesor. La búsqueda encuentra la primera copia, y el programa sigue un buen rato.

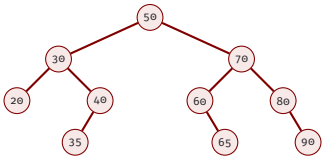
terminal

```
$ g++ -std=c++20 -g -fsanitize=address,undefined errors/forgot_successor.cpp -o build/forgot_successor && build/forgot_successor
1 3 4 7 7 8
```

`check_invariants` lo detiene: `Assertion 'low == nullptr || *low < node->key' failed`.

- El segundo 7 tiene como límite inferior al primero.
- Por eso `check_invariants` va después de **cada** operación en las pruebas al azar.

Los casos sobre un árbol más grande

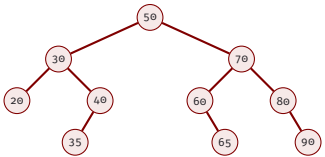


terminal

```
$ build/erase_demo
borrar 20: 50(30(20,40(35,_)),70(60(_,65),80(_,90)))
queda:     50(30(_,40(35,_)),70(60(_,65),80(_,90)))
borrar 80: 50(30(20,40(35,_)),70(60(_,65),80(_,90)))
queda:     50(30(20,40(35,_)),70(60(_,65),90))
borrar 50: 50(30(20,40(35,_)),70(60(_,65),80(_,90)))
queda:     60(30(20,40(35,_)),70(65,80(_,90)))
borrar 70: 50(30(20,40(35,_)),70(60(_,65),80(_,90)))
queda:     50(30(20,40(35,_)),80(60(_,65),90))
```

Con el predecesor también es correcto, y sale otro árbol.
Aquí siempre es el sucesor: las pruebas comparan formas.

Los casos sobre un árbol más grande



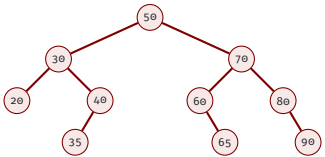
- Borrar 50: su sucesor, 60, sube a la raíz, y 65 toma su sitio.

terminal

```
$ build/erase_demo
borrar 20: 50(30(20,40(35,_)),70(60(_,65),80(_,90)))
queda:     50(30(_,40(35,_)),70(60(_,65),80(_,90)))
borrar 80: 50(30(20,40(35,_)),70(60(_,65),80(_,90)))
queda:     50(30(20,40(35,_)),70(60(_,65),90))
borrar 50: 50(30(20,40(35,_)),70(60(_,65),80(_,90)))
queda:     60(30(20,40(35,_)),70(65,80(_,90)))
borrar 70: 50(30(20,40(35,_)),70(60(_,65),80(_,90)))
queda:     50(30(20,40(35,_)),80(60(_,65),90))
```

Con el predecesor también es correcto, y sale otro árbol.
Aquí siempre es el sucesor: las pruebas comparan formas.

Los casos sobre un árbol más grande



- Borrar 50: su sucesor, 60, sube a la raíz, y 65 toma su sitio.
- Borrar 70: su sucesor es **su hijo derecho**. Con punteros al padre, es donde más se falla.

terminal

```
$ build/erase_demo
borrar 20: 50(30(20,40(35,_)),70(60(_,65),80(_,90)))
queda:    50(30(_,40(35,_)),70(60(_,65),80(_,90)))
borrar 80: 50(30(20,40(35,_)),70(60(_,65),80(_,90)))
queda:    50(30(20,40(35,_)),70(60(_,65),90))
borrar 50: 50(30(20,40(35,_)),70(60(_,65),80(_,90)))
queda:    60(30(20,40(35,_)),70(65,80(_,90)))
borrar 70: 50(30(20,40(35,_)),70(60(_,65),80(_,90)))
queda:    50(30(20,40(35,_)),80(60(_,65),90))
```

Con el predecesor también es correcto, y sale otro árbol.
Aquí siempre es el sucesor: las pruebas comparan formas.

Lo que cuesta copiar la clave

Un puntero a la clave 7, y luego borrar el 5. El nodo del 7 desaparece, aunque el 7 sigue en el conjunto.

terminal

```
$ g++ -std=c++20 -g -fsanitize=address,undefined errors/pointer_after_erase.cpp -o build/pointer_after_erase && build/pointer_after_erase
sigue el 7: 1
=================================================================
==25275==ERROR: AddressSanitizer: heap-use-after-free on address 0x503000000130 at pc 0xaaaaea40c824 bp 0xffffda269b20 sp 0xffffda269b18
READ of size 4 at 0x503000000130 thread T0
    #0 0xaaaaea40c820 in main errors/pointer_after_erase.cpp:15
...
    #1 0xaaaaea40ed4c in BstSet<int>::erase(int const&) errors/../bst_set.h:114
```

Lo que cuesta copiar la clave

Un puntero a la clave 7, y luego borrar el 5. El nodo del 7 desaparece, aunque el 7 sigue en el conjunto.

terminal

```
$ g++ -std=c++20 -g -fsanitize=address,undefined errors/pointer_after_erase.cpp -o build/pointer_after_erase && build/pointer_after_erase
sigue el 7: 1
=================================================================
==25275==ERROR: AddressSanitizer: heap-use-after-free on address 0x503000000130 at pc 0xaaaaea40c824 bp 0xffffda269b20 sp 0xffffda269b18
READ of size 4 at 0x503000000130 thread T0
    #0 0xaaaaea40c820 in main errors/pointer_after_erase.cpp:15
...
    #1 0xaaaaea40ed4c in BstSet<int>::erase(int const&) errors/../bst_set.h:114
```

- Nuestro contrato: los punteros valen hasta el próximo borrado.

Lo que cuesta copiar la clave

Un puntero a la clave 7, y luego borrar el 5. El nodo del 7 desaparece, aunque el 7 sigue en el conjunto.

terminal

```
$ g++ -std=c++20 -g -fsanitize=address,undefined errors/pointer_after_erase.cpp -o build/pointer_after_erase && build/pointer_after_erase
sigue el 7: 1
=================================================================
==25275==ERROR: AddressSanitizer: heap-use-after-free on address 0x503000000130 at pc 0xaaaaea40c824 bp 0xffffda269b20 sp 0xffffda269b18
READ of size 4 at 0x503000000130 thread T0
    #0 0xaaaaea40c820 in main errors/pointer_after_erase.cpp:15
...
    #1 0xaaaaea40ed4c in BstSet<int>::erase(int const&) errors/../bst_set.h:114
```

- Nuestro contrato: los punteros valen hasta el próximo borrado.
- `std::map` promete más. Hace falta **reenlazar** el nodo del sucesor, sin copiar: ejercicio 11.

SECCIÓN 07

Consultas ordenadas

07



Piso: bajar recordando un candidato

pseudocódigo

```
algoritmo piso(arbol, x)
    nodo ← raiz de arbol
    candidato ← NULO
    mientras nodo ≠ NULO hacer
        si x < clave de nodo entonces
            nodo ← hijo izquierdo de nodo
        sino si clave de nodo < x entonces
            candidato ← nodo
            nodo ← hijo derecho de nodo
        sino
            devolver nodo
        fin si
    fin mientras
    devolver candidato
fin algoritmo
```

Mínimo: siempre a la izquierda.
Máximo: siempre a la derecha.

Piso: bajar recordando un candidato

pseudocódigo

```
algoritmo piso(arbol, x)
    nodo ← raiz de arbol
    candidato ← NULO
    mientras nodo ≠ NULO hacer
        si x < clave de nodo entonces
            nodo ← hijo izquierdo de nodo
        sino si clave de nodo < x entonces
            candidato ← nodo
            nodo ← hijo derecho de nodo
        sino
            devolver nodo
        fin si
    fin mientras
    devolver candidato
fin algoritmo
```

Mínimo: siempre a la izquierda.
Máximo: siempre a la derecha.

`piso(6)` en nuestro árbol:

nodo	decisión	candidato
5	5 < 6, derecha	5
8	6 < 8, izquierda	5
7	6 < 7, izquierda	5
nulo	termina	**5**

Piso: bajar recordando un candidato

pseudocódigo

```
algoritmo piso(arbol, x)
    nodo ← raiz de arbol
    candidato ← NULO
    mientras nodo ≠ NULO hacer
        si x < clave de nodo entonces
            nodo ← hijo izquierdo de nodo
        sino si clave de nodo < x entonces
            candidato ← nodo
            nodo ← hijo derecho de nodo
        sino
            devolver nodo
        fin si
    fin mientras
    devolver candidato
fin algoritmo
```

Mínimo: siempre a la izquierda.
Máximo: siempre a la derecha.

`piso(6)` en nuestro árbol:

nodo	decisión	candidato
5	5 < 6, derecha	5
8	6 < 8, izquierda	5
7	6 < 7, izquierda	5
nulo	termina	**5**

Clave mayor que x: ni ella ni su derecha son el piso. Clave menor: puede ser, y lo mejor solo puede estar a su derecha.

Piso: bajar recordando un candidato

pseudocódigo

```
algoritmo piso(arbol, x)
    nodo ← raiz de arbol
    candidato ← NULO
    mientras nodo ≠ NULO hacer
        si x < clave de nodo entonces
            nodo ← hijo izquierdo de nodo
        sino si clave de nodo < x entonces
            candidato ← nodo
            nodo ← hijo derecho de nodo
        sino
            devolver nodo
        fin si
    fin mientras
    devolver candidato
fin algoritmo
```

Mínimo: siempre a la izquierda.
Máximo: siempre a la derecha.

`piso(6)` en nuestro árbol:

nodo	decisión	candidato
5	5 < 6, derecha	5
8	6 < 8, izquierda	5
7	6 < 7, izquierda	5
nulo	termina	**5**

Clave mayor que x: ni ella ni su derecha son el piso. Clave menor: puede ser, y lo mejor solo puede estar a su derecha.

Techo: simétrico. Lo escriben ustedes.

Rango: un recorrido en orden con poda

pseudocódigo

```
algoritmo rango(nodo, a, b, salida)
    si nodo = NULO entonces
        devolver
    fin si
    si a < clave de nodo entonces
        rango(hijo izquierdo de nodo, a, b, salida)
    fin si
    si a ≤ clave de nodo y clave de nodo < b entonces
        agregar clave de nodo a salida
    fin si
    si clave de nodo < b entonces
        rango(hijo derecho de nodo, a, b, salida)
    fin si
fin algoritmo
```

Rango: un recorrido en orden con poda

pseudocódigo

```
algoritmo rango(nodo, a, b, salida)
    si nodo = NULO entonces
        devolver
    fin si
    si a < clave de nodo entonces
        rango(hijo izquierdo de nodo, a, b, salida)
    fin si
    si a ≤ clave de nodo y clave de nodo < b entonces
        agregar clave de nodo a salida
    fin si
    si clave de nodo < b entonces
        rango(hijo derecho de nodo, a, b, salida)
    fin si
fin algoritmo
```

- Clave $\leq$ a: se poda el izquierdo.

Rango: un recorrido en orden con poda

pseudocódigo

```
algoritmo rango(nodo, a, b, salida)
    si nodo = NULO entonces
        devolver
    fin si
    si a < clave de nodo entonces
        rango(hijo izquierdo de nodo, a, b, salida)
    fin si
    si a ≤ clave de nodo y clave de nodo < b entonces
        agregar clave de nodo a salida
    fin si
    si clave de nodo < b entonces
        rango(hijo derecho de nodo, a, b, salida)
    fin si
fin algoritmo
```

- Clave $\leq$ a: se poda el izquierdo.
- Clave $\geq$ b: se poda el derecho.

Rango: un recorrido en orden con poda

pseudocódigo

```
algoritmo rango(nodo, a, b, salida)
    si nodo = NULO entonces
        devolver
    fin si
    si a < clave de nodo entonces
        rango(hijo izquierdo de nodo, a, b, salida)
    fin si
    si a ≤ clave de nodo y clave de nodo < b entonces
        agregar clave de nodo a salida
    fin si
    si clave de nodo < b entonces
        rango(hijo derecho de nodo, a, b, salida)
    fin si
fin algoritmo
```

- Clave $\leq$ a: se poda el izquierdo.
- Clave $\geq$ b: se poda el derecho.
- Se visitan los nodos del rango y los de dos caminos hacia los bordes.

Rango: un recorrido en orden con poda

pseudocódigo

```
algoritmo rango(nodo, a, b, salida)
    si nodo = NULO entonces
        devolver
    fin si
    si a < clave de nodo entonces
        rango(hijo izquierdo de nodo, a, b, salida)
    fin si
    si a ≤ clave de nodo y clave de nodo < b entonces
        agregar clave de nodo a salida
    fin si
    si clave de nodo < b entonces
        rango(hijo derecho de nodo, a, b, salida)
    fin si
fin algoritmo
```

- Clave $\leq$ a: se poda el izquierdo.
- Clave $\geq$ b: se poda el derecho.
- Se visitan los nodos del rango y los de dos caminos hacia los bordes.

Costo

O(altura + k), con k claves devueltas.

Contando comparaciones

terminal

```
$ build/query_demo
claves: 1023, altura: 9
contains(5000): 1, comparaciones: 17
contains(5001): 0, comparaciones: 17
floor(5001): 5000, comparaciones: 17
range con 1 claves, comparaciones: 38
range con 10 claves, comparaciones: 78
range con 100 claves, comparaciones: 426
range con 1000 claves, comparaciones: 4018
```

Unas 4 comparaciones por clave, más la bajada.

En la biblioteca

`std::map` tiene `lower_bound(x)` (primera $\geq$ x) y `upper_bound(x)` (primera > x).

Contando comparaciones

terminal

```
$ build/query_demo
claves: 1023, altura: 9
contains(5000): 1, comparaciones: 17
contains(5001): 0, comparaciones: 17
floor(5001): 5000, comparaciones: 17
range con 1 claves, comparaciones: 38
range con 10 claves, comparaciones: 78
range con 100 claves, comparaciones: 426
range con 1000 claves, comparaciones: 4018
```

Unas 4 comparaciones por clave, más la bajada.

En la biblioteca

`std::map` tiene `lower_bound(x)` (primera $\geq$ x) y `upper_bound(x)` (primera > x).

- **Techo:** `lower_bound(x)`.

Contando comparaciones

terminal

```
$ build/query_demo
claves: 1023, altura: 9
contains(5000): 1, comparaciones: 17
contains(5001): 0, comparaciones: 17
floor(5001): 5000, comparaciones: 17
range con 1 claves, comparaciones: 38
range con 10 claves, comparaciones: 78
range con 100 claves, comparaciones: 426
range con 1000 claves, comparaciones: 4018
```

Unas 4 comparaciones por clave, más la bajada.

En la biblioteca

`std::map` tiene `lower_bound(x)` (primera $\geq$ x) y `upper_bound(x)` (primera > x).

- **Techo:** `lower_bound(x)`.
- **Piso:** el anterior a `upper_bound(x)`. Si es `begin()`, no hay piso: `std::prev` ahí es comportamiento indefinido.

Contando comparaciones

terminal

```
$ build/query_demo
claves: 1023, altura: 9
contains(5000): 1, comparaciones: 17
contains(5001): 0, comparaciones: 17
floor(5001): 5000, comparaciones: 17
range con 1 claves, comparaciones: 38
range con 10 claves, comparaciones: 78
range con 100 claves, comparaciones: 426
range con 1000 claves, comparaciones: 4018
```

Unas 4 comparaciones por clave, más la bajada.

En la biblioteca

`std::map` tiene `lower_bound(x)` (primera $\geq$ x) y `upper_bound(x)` (primera > x).

- **Techo:** `lower_bound(x)`.
- **Piso:** el anterior a `upper_bound(x)`. Si es `begin()`, no hay piso: `std::prev` ahí es comportamiento indefinido.
- **Rango [a, b):** de `lower_bound(a)` a `lower_bound(b)`.

SECCIÓN 08

El puntero al padre

08



¿Cuál es el siguiente?

Un iterador necesita ir al sucesor de **cualquier** nodo. Si no tiene hijo derecho, hay que **subir**.

lab/bst_map.h

```
1  struct Node {
2      K key;
3      V value;
4      Node* parent = nullptr;
5      Node* left = nullptr;
6      Node* right = nullptr;
7  };
```

Lo que da:

¿Cuál es el siguiente?

Un iterador necesita ir al sucesor de **cualquier** nodo. Si no tiene hijo derecho, hay que **subir**.

lab/bst_map.h

```
1  struct Node {
2      K key;
3      V value;
4      Node* parent = nullptr;
5      Node* left = nullptr;
6      Node* right = nullptr;
7  };
```

Lo que da:

- Un iterador es un puntero a un nodo, y sobrevive a una inserción. Una pila guardada no.

¿Cuál es el siguiente?

Un iterador necesita ir al sucesor de **cualquier** nodo. Si no tiene hijo derecho, hay que **subir**.

lab/bst_map.h

```
1  struct Node {
2      K key;
3      V value;
4      Node* parent = nullptr;
5      Node* left = nullptr;
6      Node* right = nullptr;
7  };
```

Lo que da:

- Un iterador es un puntero a un nodo, y sobrevive a una inserción. Una pila guardada no.
- El destructor es un postorden real, sin recursión ni memoria adicional.

¿Cuál es el siguiente?

Un iterador necesita ir al sucesor de **cualquier** nodo. Si no tiene hijo derecho, hay que **subir**.

lab/bst_map.h

```
1  struct Node {
2      K key;
3      V value;
4      Node* parent = nullptr;
5      Node* left = nullptr;
6      Node* right = nullptr;
7  };
```

Lo que da:

- Un iterador es un puntero a un nodo, y sobrevive a una inserción. Una pila guardada no.
- El destructor es un postorden real, sin recursión ni memoria adicional.
- Las semanas 11 y 12 suben desde el nodo nuevo, reparando.

¿Cuál es el siguiente?

Un iterador necesita ir al sucesor de **cualquier** nodo. Si no tiene hijo derecho, hay que **subir**.

lab/bst_map.h

```
1  struct Node {
2      K key;
3      V value;
4      Node* parent = nullptr;
5      Node* left = nullptr;
6      Node* right = nullptr;
7  };
```

Lo que da:

- Un iterador es un puntero a un nodo, y sobrevive a una inserción. Una pila guardada no.
- El destructor es un postorden real, sin recursión ni memoria adicional.
- Las semanas 11 y 12 suben desde el nodo nuevo, reparando.

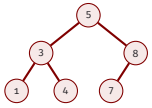
Lo que cuesta

8 bytes por nodo, y una invariante más: si x tiene hijo h, `h->parent == x`. Toda línea que cambie un enlace debe pensar en él.

El sucesor con padre

pseudocódigo

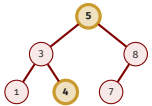
```
algoritmo sucesor(nodo)
    si nodo tiene hijo derecho entonces
        devolver minimo(hijo derecho de nodo)
    fin si
    padre ← padre de nodo
    mientras padre ≠ NULO y nodo es hijo derecho de padre hacer
        nodo ← padre
        padre ← padre de padre
    fin mientras
    devolver padre
fin algoritmo
```



El sucesor con padre

pseudocódigo

```
algoritmo sucesor(nodo)
    si nodo tiene hijo derecho entonces
        devolver minimo(hijo derecho de nodo)
    fin si
    padre ← padre de nodo
    mientras padre ≠ NULO y nodo es hijo derecho de padre hacer
        nodo ← padre
        padre ← padre de padre
    fin mientras
    devolver padre
fin algoritmo
```

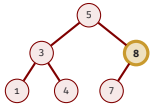


Sucesor de 4: es hijo derecho de 3, y 3 es hijo izquierdo de 5. Es el 5.

El sucesor con padre

pseudocódigo

```
algoritmo sucesor(nodo)
    si nodo tiene hijo derecho entonces
        devolver minimo(hijo derecho de nodo)
    fin si
    padre ← padre de nodo
    mientras padre ≠ NULO y nodo es hijo derecho de padre hacer
        nodo ← padre
        padre ← padre de padre
    fin mientras
    devolver padre
fin algoritmo
```

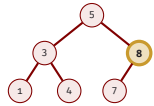


Sucesor de 8: sube hasta salir por la raíz. Nulo: `end()`.

El sucesor con padre

pseudocódigo

```
algoritmo sucesor(nodo)
    si nodo tiene hijo derecho entonces
        devolver minimo(hijo derecho de nodo)
    fin si
    padre ← padre de nodo
    mientras padre ≠ NULO y nodo es hijo derecho de padre hacer
        nodo ← padre
        padre ← padre de padre
    fin mientras
    devolver padre
fin algoritmo
```



Sucesor de 8: sube hasta salir por la raíz. Nulo: `end()`.
Costo. Un `++` puede costar Θ(altura). Un recorrido completo, Θ(n): cada arista se baja y se sube una vez.

SECCIÓN 09

La altura lo decide todo

09

Entre log n y n

- **Máximo: n − 1.** Una cadena: cada nodo con un solo hijo.

Entre log n y n

- **Máximo: n − 1.** Una cadena: cada nodo con un solo hijo.
- **Mínimo: $\lfloor \log_2 n \rfloor$.** Cada nivel tiene a lo sumo el doble que el anterior, así que altura h da como máximo $2^{h+1} - 1$ nodos.

Entre log n y n

- **Máximo: n − 1.** Una cadena: cada nodo con un solo hijo.
- **Mínimo: $\lfloor \log_2 n \rfloor$.** Cada nivel tiene a lo sumo el doble que el anterior, así que altura h da como máximo $2^{h+1} - 1$ nodos.

Con un millón de nodos: entre **19** y **999 999**.

Entre log n y n

- **Máximo: n − 1.** Una cadena: cada nodo con un solo hijo.
- **Mínimo: $\lfloor \log_2 n \rfloor$.** Cada nivel tiene a lo sumo el doble que el anterior, así que altura h da como máximo $2^{h+1} - 1$ nodos.

Con un millón de nodos: entre **19** y **999 999**.

> **Idea clave**
>
> Buscar, insertar y borrar cuestan O(altura). Y la altura la decide el orden de inserción.

Todos los órdenes de 6 claves

terminal

```
$ build/insertion_orders
ordenes: 720
altura 2: 80 ordenes
altura 3: 400 ordenes
altura 4: 208 ordenes
altura 5: 32 ordenes
```

Todos los órdenes de 6 claves

```
terminal

$ build/insertion_orders
ordenes: 720
altura 2: 80 ordenes
altura 3: 400 ordenes
altura 4: 208 ordenes
altura 5: 32 ordenes
```

- La mayor altura con 6 claves es **5**.

Todos los órdenes de 6 claves

terminal

```
$ build/insertion_orders
ordenes: 720
altura 2: 80 ordenes
altura 3: 400 ordenes
altura 4: 208 ordenes
altura 5: 32 ordenes
```

- La mayor altura con 6 claves es **5**.
- El orden creciente da una cadena. También el decreciente, y zigzags como 1, 8, 3, 7, 4, 5.

Todos los órdenes de 6 claves

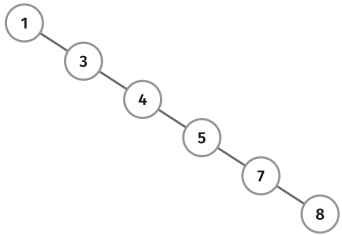
terminal

```
$ build/insertion_orders
ordenes: 720
altura 2: 80 ordenes
altura 3: 400 ordenes
altura 4: 208 ordenes
altura 5: 32 ordenes
```

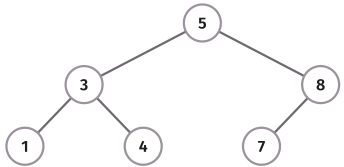
- La mayor altura con 6 claves es **5**.
- El orden creciente da una cadena. También el decreciente, y zigzags como 1, 8, 3, 7, 4, 5.
- Cada clave debe ser la menor o la mayor de las que quedan, y la última no elige: $2^5 = 32$.

Las mismas claves, otra altura

insertar 1, 3, 4, 5, 7, 8: altura 5



insertar 5, 3, 8, 1, 4, 7: altura 2



El recorrido en orden es el mismo: 1 3 4 5 7 8.
Solo cambió el orden de inserción

LOS 720 ÓRDENES DE INSERCIÓN, SEGÚN LA ALTURA QUE DAN



Ordenadas y al azar: los conteos

terminal

```
$ build/height_table
Claves al azar, tres semillas
        n  log2n    h1    h2    h3  h/log2n prof.media  prof/log
     1000   10.0    21    25    19     2.17      11.28      1.13
    16000   14.0    33    31    30     2.24      16.52      1.18
  1024000   20.0    55    46    47     2.47      24.52      1.23
Claves ordenadas
        n   altura  comparaciones         n(n-1)
     1000      999         999000         999000
    16000    15999      255984000      255984000
```

Ordenadas y al azar: los conteos

terminal

```
$ build/height_table
Claves al azar, tres semillas
        n  log2n    h1    h2    h3  h/log2n prof.media  prof/log
     1000   10.0    21    25    19     2.17      11.28      1.13
    16000   14.0    33    31    30     2.24      16.52      1.18
  1024000   20.0    55    46    47     2.47      24.52      1.23
Claves ordenadas
        n   altura  comparaciones         n(n-1)
     1000      999         999000         999000
    16000    15999      255984000      255984000
```

- **Ordenadas:** exactamente $n(n-1)$ comparaciones. Una lista enlazada.

Ordenadas y al azar: los conteos

terminal

```
$ build/height_table
Claves al azar, tres semillas
        n  log2n    h1    h2    h3  h/log2n prof.media  prof/log
     1000   10.0    21    25    19     2.17      11.28      1.13
    16000   14.0    33    31    30     2.24      16.52      1.18
  1024000   20.0    55    46    47     2.47      24.52      1.23
Claves ordenadas
        n   altura  comparaciones         n(n-1)
     1000      999         999000         999000
    16000    15999      255984000      255984000
```

- **Ordenadas:** exactamente $n(n-1)$ comparaciones. Una lista enlazada.
- **Al azar:** se mide, no se demuestra. Altura cerca de 2.2 a 2.5 veces $\log_2 n$, en estas corridas.

Ordenadas y al azar: los conteos

terminal

```
$ build/height_table
Claves al azar, tres semillas
        n  log2n    h1    h2    h3  h/log2n prof.media  prof/log
     1000   10.0    21    25    19     2.17      11.28      1.13
    16000   14.0    33    31    30     2.24      16.52      1.18
  1024000   20.0    55    46    47     2.47      24.52      1.23
Claves ordenadas
        n   altura  comparaciones         n(n-1)
     1000      999         999000         999000
    16000    15999      255984000      255984000
```

- **Ordenadas:** exactamente $n(n-1)$ comparaciones. Una lista enlazada.
- **Al azar:** se mide, no se demuestra. Altura cerca de 2.2 a 2.5 veces $\log_2 n$, en estas corridas.
- "Cercana a log n": de ese orden, no igual. No sabemos dónde se detiene el cociente.

Ordenadas y al azar: los conteos

terminal

```
$ build/height_table
Claves al azar, tres semillas
        n  log2n    h1    h2    h3  h/log2n prof.media  prof/log
     1000   10.0    21    25    19     2.17      11.28      1.13
    16000   14.0    33    31    30     2.24      16.52      1.18
  1024000   20.0    55    46    47     2.47      24.52      1.23
Claves ordenadas
        n   altura  comparaciones         n(n-1)
     1000      999         999000         999000
    16000    15999      255984000      255984000
```

- **Ordenadas:** exactamente $n(n-1)$ comparaciones. Una lista enlazada.
- **Al azar:** se mide, no se demuestra. Altura cerca de 2.2 a 2.5 veces $\log_2 n$, en estas corridas.
- "Cercana a log n": de ese orden, no igual. No sabemos dónde se detiene el cociente.
- Solo mide inserciones. Con borrados mezclados: ejercicio 9.

El reloj dice lo mismo

terminal

```
$ build/insert_times
        n    azar(s)          x    orden(s)          x
     1000    0.00003        0.0     0.00090        0.0
     2000    0.00009        2.6     0.00357        4.0
     4000    0.00023        2.6     0.01438        4.0
     8000    0.00070        3.0     0.06098        4.2
    16000    0.00194        2.8     0.24795        4.1
    32000    0.00513        2.6     1.06629        4.3
```

`-O2`, sin sanitizers, mediana de 5. Dependen de la máquina; se repitió en 3 corridas.

El reloj dice lo mismo

terminal

```
$ build/insert_times
        n    azar(s)         x    orden(s)         x
     1000    0.00003       0.0     0.00090       0.0
     2000    0.00009       2.6     0.00357       4.0
     4000    0.00023       2.6     0.01438       4.0
     8000    0.00070       3.0     0.06098       4.2
    16000    0.00194       2.8     0.24795       4.1
    32000    0.00513       2.6     1.06629       4.3
```

`-O2`, sin sanitizers, mediana de 5. Dependen de la máquina; se repitió en 3 corridas.

- **Ordenadas:** duplicar n multiplica por 4. Es $\Theta(n^2)$.

El reloj dice lo mismo

terminal

```
$ build/insert_times
        n    azar(s)          x    orden(s)          x
     1000    0.00003        0.0     0.00090        0.0
     2000    0.00009        2.6     0.00357        4.0
     4000    0.00023        2.6     0.01438        4.0
     8000    0.00070        3.0     0.06098        4.2
    16000    0.00194        2.8     0.24795        4.1
    32000    0.00513        2.6     1.06629        4.3
```

`-O2`, sin sanitizers, mediana de 5. Dependen de la máquina; se repitió en 3 corridas.

- **Ordenadas:** duplicar n multiplica por 4. Es $\Theta(n^2)$.
- **Al azar:** por 2.2 a 3.2, más del poco más de 2 que predice n log n. Lo probable es la cache: no se midió directamente.

Un caso real: `palabras.txt` viene ordenado

terminal

```
$ build/dictionary_order ../../data/palabras.txt
orden de bytes: si
tal como vienen: 20000 palabras, altura 19999, 1.17084 s
barajadas: 20000 palabras, altura 31, 0.00478758 s
barajadas: 1045757 palabras, altura 49, 1.73514 s
std::map, tal como vienen: 1045757 palabras, 0.174806 s
```

Un caso real: `palabras.txt` viene ordenado

terminal

```
$ build/dictionary_order ../../data/palabras.txt
orden de bytes: si
tal como vienen: 20000 palabras, altura 19999, 1.17084 s
barajadas: 20000 palabras, altura 31, 0.00478758 s
barajadas: 1045757 palabras, altura 49, 1.73514 s
std::map, tal como vienen: 1045757 palabras, 0.174806 s
```

- Del orden de 200 veces más, con el mismo código. Solo cambió el orden.

Un caso real: `palabras.txt` viene ordenado

terminal

```
$ build/dictionary_order ../../data/palabras.txt
orden de bytes: si
tal como vienen: 20000 palabras, altura 19999, 1.17084 s
barajadas: 20000 palabras, altura 31, 0.00478758 s
barajadas: 1045757 palabras, altura 49, 1.73514 s
std::map, tal como vienen: 1045757 palabras, 0.174806 s
```

- Del orden de 200 veces más, con el mismo código. Solo cambió el orden.
- Con el archivo completo: unas 2700 veces más, cerca de una hora, contra menos de 2 segundos. Es una extrapolación.

Un caso real: `palabras.txt` viene ordenado

terminal

```
$ build/dictionary_order ../../data/palabras.txt
orden de bytes: si
tal como vienen: 20000 palabras, altura 19999, 1.17084 s
barajadas: 20000 palabras, altura 31, 0.00478758 s
barajadas: 1045757 palabras, altura 49, 1.73514 s
std::map, tal como vienen: 1045757 palabras, 0.174806 s
```

- Del orden de 200 veces más, con el mismo código. Solo cambió el orden.
- Con el archivo completo: unas 2700 veces más, cerca de una hora, contra menos de 2 segundos. Es una extrapolación.
- **`std::map`** recibió el archivo completo en el mismo orden malo: 0.17 s. No tuvo suerte. Se reparó.

Un caso real: `palabras.txt` viene ordenado

terminal

```
$ build/dictionary_order ../../data/palabras.txt
orden de bytes: si
tal como vienen: 20000 palabras, altura 19999, 1.17084 s
barajadas: 20000 palabras, altura 31, 0.00478758 s
barajadas: 1045757 palabras, altura 49, 1.73514 s
std::map, tal como vienen: 1045757 palabras, 0.174806 s
```

- Del orden de 200 veces más, con el mismo código. Solo cambió el orden.
- Con el archivo completo: unas 2700 veces más, cerca de una hora, contra menos de 2 segundos. Es una extrapolación.
- **`std::map`** recibió el archivo completo en el mismo orden malo: 0.17 s. No tuvo suerte. Se reparó.

Un árbol sin balance es rápido si tiene suerte, y los datos ordenados son muy comunes.

SECCIÓN 10

std::map frente a std::unordered_map

10



Lo que promete el estándar, y lo que hace la biblioteca

El estándar de C++

Claves ordenadas. `find`, `insert`, `erase` de una clave en O(log n). Insertar no invalida iteradores ni referencias; borrar, solo los del borrado. **No dice cómo.**

terminal

```
$ build/map_guarantees
puntero al 7 tras mil inserciones y un borrado: 70
reservas de std::map al insertar 1000 claves: 1000
bytes por nodo de std::map<int, int>: 40
bytes por nodo de std::unordered_map<int, int>: 16
```

Lo que promete el estándar, y lo que hace la biblioteca

El estándar de C++

Claves ordenadas. `find`, `insert`, `erase` de una clave en O(log n). Insertar no invalida iteradores ni referencias; borrar, solo los del borrado. **No dice cómo.**

libstdc++

Árbol rojinegro, en `bits/stl_tree.h`. Nodo: color y tres punteros: padre, izquierdo, derecho.

terminal

```
$ build/map_guarantees
puntero al 7 tras mil inserciones y un borrado: 70
reservas de std::map al insertar 1000 claves: 1000
bytes por nodo de std::map<int, int>: 40
bytes por nodo de std::unordered_map<int, int>: 16
```

Lo que promete el estándar, y lo que hace la biblioteca

El estándar de C++

Claves ordenadas. `find`, `insert`, `erase` de una clave en O(log n). Insertar no invalida iteradores ni referencias; borrar, solo los del borrado. **No dice cómo.**

libstdc++

Árbol rojinegro, en `bits/stl_tree.h`. Nodo: color y tres punteros: padre, izquierdo, derecho.

terminal

```
$ build/map_guarantees
puntero al 7 tras mil inserciones y un borrado: 70
reservas de std::map al insertar 1000 claves: 1000
bytes por nodo de std::map<int, int>: 40
bytes por nodo de std::unordered_map<int, int>: 16
```

- El puntero sobrevive: a diferencia de `BstSet`.

Lo que promete el estándar, y lo que hace la biblioteca

El estándar de C++

Claves ordenadas. `find`, `insert`, `erase` de una clave en O(log n). Insertar no invalida iteradores ni referencias; borrar, solo los del borrado. **No dice cómo.**

libstdc++

Árbol rojinegro, en `bits/stl_tree.h`. Nodo: color y tres punteros: padre, izquierdo, derecho.

terminal

```
$ build/map_guarantees
puntero al 7 tras mil inserciones y un borrado: 70
reservas de std::map al insertar 1000 claves: 1000
bytes por nodo de std::map<int, int>: 40
bytes por nodo de std::unordered_map<int, int>: 16
```

- El puntero sobrevive: a diferencia de `BstSet`.
- Una reserva por clave.

Lo que promete el estándar, y lo que hace la biblioteca

El estándar de C++

Claves ordenadas. `find`, `insert`, `erase` de una clave en O(log n). Insertar no invalida iteradores ni referencias; borrar, solo los del borrado. **No dice cómo.**

libstdc++

Árbol rojinegro, en `bits/stl_tree.h`. Nodo: color y tres punteros: padre, izquierdo, derecho.

terminal

```
$ build/map_guarantees
puntero al 7 tras mil inserciones y un borrado: 70
reservas de std::map al insertar 1000 claves: 1000
bytes por nodo de std::map<int, int>: 40
bytes por nodo de std::unordered_map<int, int>: 16
```

- El puntero sobrevive: a diferencia de `BstSet`.
- Una reserva por clave.
- La tabla tiene además su arreglo de buckets: estas dos cifras no dicen cuál ocupa menos.

Medido con palabras reales

terminal

```
$ build/map_vs_unordered ../../data/palabras.txt
  (tiempo por palabra, en ns; mediana de 3)
        n   ins.map  ins.hash      x   bus.map  bus.hash      x   ord.map  ord.hash      x
    10000       209        85    2.5       158        34    4.6        16       144    8.8
   100000       369       133    2.8       500        66    7.6        63       298    4.7
  1045757      1600       479    3.3      1542       220    7.0       140       640    4.6
```

`-O2`, palabras barajadas. `ord`: sacar todo en orden; la tabla tiene que ordenar. Dependen de la máquina.

Medido con palabras reales

terminal

```
$ build/map_vs_unordered ../../data/palabras.txt
  (tiempo por palabra, en ns; mediana de 3)
       n   ins.map  ins.hash      x   bus.map  bus.hash      x   ord.map  ord.hash      x
   10000       209        85    2.5       158        34    4.6        16       144    8.8
  100000       369       133    2.8       500        66    7.6        63       298    4.7
 1045757      1600       479    3.3      1542       220    7.0       140       640    4.6
```

`-O2`, palabras barajadas. `ord`: sacar todo en orden; la tabla tiene que ordenar. Dependen de la máquina.

- En 4 corridas, la tabla ganó entre 2.5 y 3.7 veces al insertar, y entre 4.6 y 9.0 al buscar.

Medido con palabras reales

terminal

```
$ build/map_vs_unordered ../../data/palabras.txt
  (tiempo por palabra, en ns; mediana de 3)
        n   ins.map  ins.hash      x   bus.map  bus.hash      x   ord.map  ord.hash      x
    10000       209        85    2.5       158        34    4.6        16       144    8.8
   100000       369       133    2.8       500        66    7.6        63       298    4.7
  1045757      1600       479    3.3      1542       220    7.0       140       640    4.6
```

`-O2`, palabras barajadas. `ord`: sacar todo en orden; la tabla tiene que ordenar. Dependen de la máquina.

- En 4 corridas, la tabla ganó entre 2.5 y 3.7 veces al insertar, y entre 4.6 y 9.0 al buscar.
- El mapa ganó entre 3.4 y 9.1 veces al sacar todo en orden.

Medido con palabras reales

terminal

```
$ build/map_vs_unordered ../../data/palabras.txt
  (tiempo por palabra, en ns; mediana de 3)
        n   ins.map  ins.hash      x   bus.map  bus.hash      x   ord.map  ord.hash      x
    10000       209        85    2.5       158        34    4.6        16       144    8.8
   100000       369       133    2.8       500        66    7.6        63       298    4.7
  1045757      1600       479    3.3      1542       220    7.0       140       640    4.6
```

`-O2`, palabras barajadas. `ord`: sacar todo en orden; la tabla tiene que ordenar. Dependen de la máquina.

- En 4 corridas, la tabla ganó entre 2.5 y 3.7 veces al insertar, y entre 4.6 y 9.0 al buscar.
- El mapa ganó entre 3.4 y 9.1 veces al sacar todo en orden.
- Unas 20 comparaciones de cadenas y otros tantos punteros, frente a un hash y casi directo.

Qué se gana y qué se paga

	`std::map`	`std::unordered_map`
Buscar, insertar, borrar	O(log n) en el peor caso	O(1) en promedio, O(n) en el peor
Recorrer en orden	Sí, Θ(n)	No: copiar y ordenar
Mínimo, piso, techo, siguiente	O(log n)	Θ(n)
Rango con k claves	O(log n + k)	Θ(n)
Qué pide a la clave	<	hash e igualdad
Iteradores	Estables	Se invalidan al redistribuir

Qué se gana y qué se paga

	`std::map`	`std::unordered_map`
Buscar, insertar, borrar	O(log n) en el peor caso	O(1) en promedio, O(n) en el peor
Recorrer en orden	Sí, Θ(n)	No: copiar y ordenar
Mínimo, piso, techo, siguiente	O(log n)	Θ(n)
Rango con k claves	O(log n + k)	Θ(n)
Qué pide a la clave	<	hash e igualdad
Iteradores	Estables	Se invalidan al redistribuir

Idea clave

Si no necesitan el orden, tabla hash. Si necesitan piso, techo, rangos, recorrer ordenado o un peor caso garantizado, árbol.

Qué se gana y qué se paga

	`std::map`	`std::unordered_map`
Buscar, insertar, borrar	O(log n) en el peor caso	O(1) en promedio, O(n) en el peor
Recorrer en orden	Sí, $\Theta(n)$	No: copiar y ordenar
Mínimo, piso, techo, siguiente	O(log n)	$\Theta(n)$
Rango con k claves	O(log n + k)	$\Theta(n)$
Qué pide a la clave	`<`	hash e igualdad
Iteradores	Estables	Se invalidan al redistribuir

Idea clave

Si no necesitan el orden, tabla hash. Si necesitan piso, techo, rangos, recorrer ordenado o un peor caso garantizado, árbol.

Error común

`mapa[clave]` **inserta** si no estaba. Para consultar: `find` o `contains`. `insert` no sobrescribe: use `insert_or_assign`.

Qué es orden para una cadena

terminal

```
$ build/byte_order ../../data/palabras.txt
ultima: úvulas
[a, b): 157836, de a a aúpo
[á, â): 261, de ábaco a ázoes
[n, o): 8267, de naaguán a núñez
[ñ, ò): 92, de ña a ñuñoa
[pre, prf): 9580, de preadamita a preñó
despues de zuzos: zuña
piso de rodillas: rodilla, siguiente: rodillazo
```

- `std::string` compara **bytes**. UTF-8 pone `á` y `ñ` después de `z`.

Qué es orden para una cadena

terminal

```
$ build/byte_order ../../data/palabras.txt
ultima: úvulas
[a, b): 157836, de a a aúpo
[á, â): 261, de ábaco a ázoes
[n, o): 8267, de naaguán a núñez
[ñ, ò): 92, de ña a ñuñoa
[pre, prf): 9580, de preadamita a preñó
despues de zuzos: zuña
piso de rodillas: rodilla, siguiente: rodillazo
```

- `std::string` compara **bytes**. UTF-8 pone `á` y `ñ` después de `z`.
- `ábaco` no está en [a, b). `ña` no está en [n, o).

Qué es orden para una cadena

terminal

```
$ build/byte_order ../../data/palabras.txt
ultima: úvulas
[a, b): 157836, de a a aúpo
[á, â): 261, de ábaco a ázoes
[n, o): 8267, de naaguán a núñez
[ñ, ò): 92, de ña a ñuñoa
[pre, prf): 9580, de preadamita a preñó
despues de zuzos: zuña
piso de rodillas: rodilla, siguiente: rodillazo
```

- `std::string` compara **bytes**. UTF-8 pone `á` y `ñ` después de `z`.
- `ábaco` no está en [a, b). `ña` no está en [n, o).
- Prefijo: [`pre`, `prf`).

Qué es orden para una cadena

terminal

```
$ build/byte_order ../../data/palabras.txt
ultima: úvulas
[a, b): 157836, de a a aúpo
[á, â): 261, de ábaco a ázoes
[n, o): 8267, de naaguán a núñez
[ñ, ò): 92, de ña a ñuñoa
[pre, prf): 9580, de preadamita a preñó
despues de zuzos: zuña
piso de rodillas: rodilla, siguiente: rodillazo
```

- `std::string` compara **bytes**. UTF-8 pone `á` y `ñ` después de `z`.
- `ábaco` no está en [a, b). `ña` no está en [n, o).
- Prefijo: [`pre`, `prf`).
- `rodillas` no está: es un hueco de los datos, no de la estructura.

Qué es orden para una cadena

terminal

```
$ build/byte_order ../../data/palabras.txt
ultima: úvulas
[a, b): 157836, de a a aúpo
[á, â): 261, de ábaco a ázoes
[n, o): 8267, de naaguán a núñez
[ñ, ò): 92, de ña a ñuñoa
[pre, prf): 9580, de preadamita a preñó
despues de zuzos: zuña
piso de rodillas: rodilla, siguiente: rodillazo
```

- `std::string` compara **bytes**. UTF-8 pone `á` y `ñ` después de `z`.
- `ábaco` no está en [a, b). `ña` no está en [n, o).
- Prefijo: [`pre`, `prf`).
- `rodillas` no está: es un hueco de los datos, no de la estructura.

Un orden alfabético de verdad necesita otra comparación. Ejercicio 10.

SECCIÓN 11

Laboratorio

11

Un mapa ordenado, probado contra `std::map`

Sesión 1: `BstMap<K, V>`.

Sesión 2:

```
terminal
make test-basics
make test-erase
make test-queries
make test-words
make test-lab
make bench
```

Un mapa ordenado, probado contra `std::map`

Sesión 1: `BstMap<K, V>`.

1. `insert`, `find`, recorrido, destructor sin recursión, `check_invariants` con límites.

Sesión 2:

terminal

```
make test-basics
make test-erase
make test-queries
make test-words
make test-lab
make bench
```

Un mapa ordenado, probado contra `std::map`

Sesión 1: `BstMap<K, V>`.

1. `insert`, `find`, recorrido, destructor sin recursión, `check_invariants` con límites.
2. `erase`, con los tres casos. Al azar contra `std::map`.

Sesión 2:

terminal

```
make test-basics
make test-erase
make test-queries
make test-words
make test-lab
make bench
```

Un mapa ordenado, probado contra `std::map`

Sesión 1: `BstMap<K, V>`.

1. `insert`, `find`, recorrido, destructor sin recursión, `check_invariants` con límites.
2. `erase`, con los tres casos. Al azar contra `std::map`.

Sesión 2:

3. `floor`, `ceiling`, `range` e iterador.

terminal

```
make test-basics
make test-erase
make test-queries
make test-words
make test-lab
make bench
```

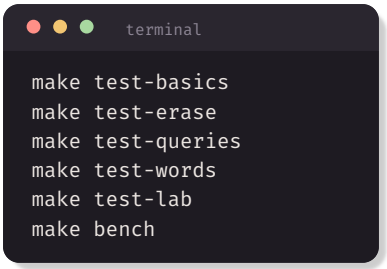
Un mapa ordenado, probado contra `std::map`

Sesión 1: `BstMap<K, V>`.

1. `insert`, `find`, recorrido, destructor sin recursión, `check_invariants` con límites.
2. `erase`, con los tres casos. Al azar contra `std::map`.

Sesión 2:

3. `floor`, `ceiling`, `range` e iterador.
4. `word_freq`: frecuencias, en orden y por rango.



Un mapa ordenado, probado contra `std::map`

Sesión 1: `BstMap<K, V>`.

1. `insert`, `find`, recorrido, destructor sin recursión, `check_invariants` con límites.
2. `erase`, con los tres casos. Al azar contra `std::map`.

Sesión 2:

3. `floor`, `ceiling`, `range` e iterador.
4. `word_freq`: frecuencias, en orden y por rango.
5. Medir: `make bench` y `lab/report.md`.

terminal

```
make test-basics
make test-erase
make test-queries
make test-words
make test-lab
make bench
```

Un mapa ordenado, probado contra `std::map`

Sesión 1: `BstMap<K, V>`.

1. `insert`, `find`, recorrido, destructor sin recursión, `check_invariants` con límites.
2. `erase`, con los tres casos. Al azar contra `std::map`.

Sesión 2:

3. `floor`, `ceiling`, `range` e iterador.
4. `word_freq`: frecuencias, en orden y por rango.
5. Medir: `make bench` y `lab/report.md`.

Vienen hechos: copia, movimiento, `height`, `pre_order_keys`, `split_words`. No es `BstSet` con un campo más: el padre cambia casi todas las líneas que tocan un enlace.

terminal

```
make test-basics
make test-erase
make test-queries
make test-words
make test-lab
make bench
```

Para la próxima clase

- Leer los apuntes de la semana 11: árboles AVL.
- Laboratorio: `BstMap`. Dibujen los punteros `parent` antes de escribir `erase`.
- Guarden su `BstMap`: la semana 11 parte de él.

Pregunta

`std::map` recibió `palabras.txt` en orden y tardó 0.17 s. ¿Qué tendría que hacer un árbol al insertar para no volverse una lista, sin revisar todos sus nodos?