TD3: Algoritmos y Estructuras de Datos

Prof. Agustín Garassino, Gervasio Pérez

Segundo Semestre de 2024

Clase Teórica 11

Árboles

Resumen

En la clase de hoy veremos

- Arbol Binario
- Arbol Binario de Búsqueda
- Balanceo de ABB, AVL
- Heap

Complejidad de `std::set<T>` y `std::map<K,V>`

`std::set<T>` (link a doc)

- Constructor vacío en O(1)
- `s.insert(e)` en $O(\log n)$ comparaciones del tipo `T`
- `s.erase(e)` en $O(\log n)$ comparaciones del tipo `T`
- `s.contains(e)` en $O(\log n)$ comparaciones del tipo `T`

`std::map<K,V>` (link a doc)

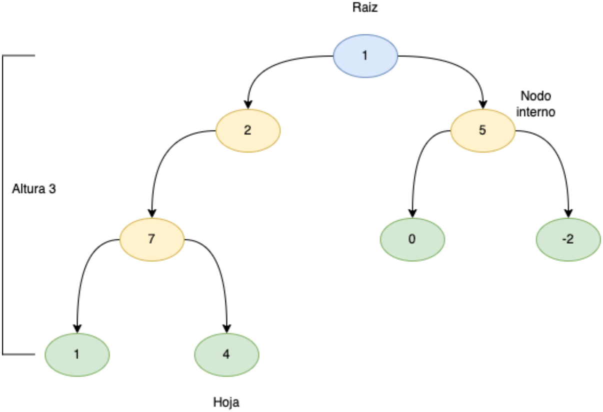
- Constructor vacío en O(1)
- `m.at(k)` en $O(\log n)$ comparaciones del tipo `K`
- `m[k] = v` en $O(\log n + \mathit{copy}(v))$ comparaciones del tipo `K`
- `m.contains(k)` en $O(\log n)$ comparaciones del tipo `K`

¿Cómo se implementan estas estructuras para lograr estas complejidades?

Arbol Binario

Un árbol binario es una estructura de nodos conectados en la que

- cada nodo está conectado con a lo sumo dos nodos hijos,
- hay un único nodo que no tiene padre y lo llamamos raíz,
- no tiene ciclos y cada nodo tiene un único padre.

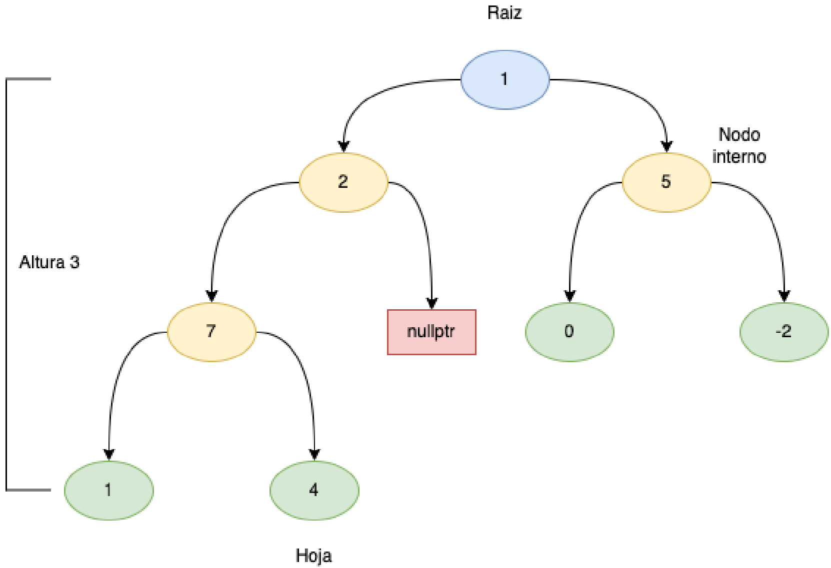


Arbol Binario

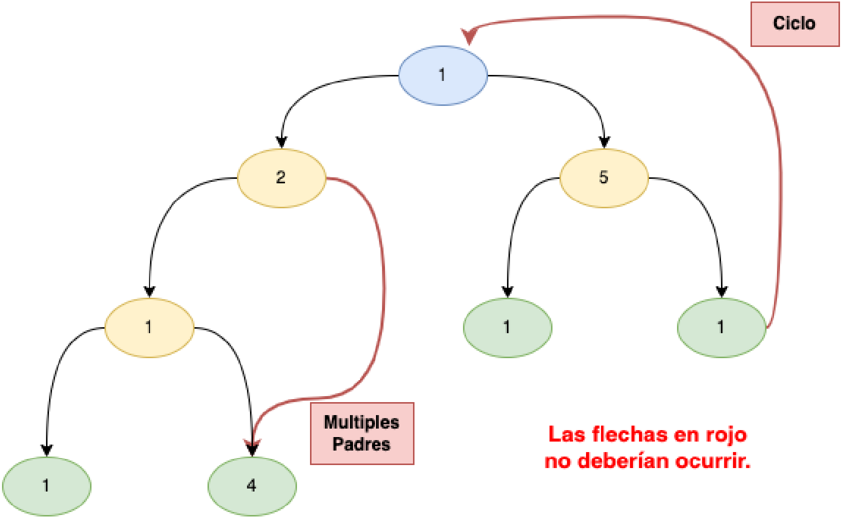
Lo podemos representar con la siguiente estructura:

```
struct nodo {
    int elem;
    nodo* izquierdo = nullptr;
    nodo* derecho = nullptr;

//////////////////////////////////////////////////
// Rep: "Árbol binario bien formado"
//      - no hay ciclos en la cadena de punteros
//      - cada nodo tiene un único "padre"
};

nodo* raiz;
```

Arbol Binario



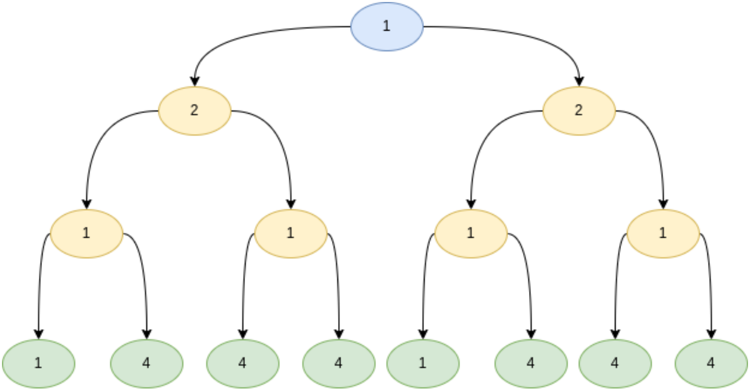
Arbol Binario - problemas



Las flechas en rojo no deberían ocurrir.

Arbol Binario Completo

Un árbol binario está completo si todos los niveles del árbol tienen la máxima cantidad de nodos posible.



Para un árbol binario completo, si la altura es h tenemos que:

- tiene 2^h hojas,
- tiene $2^{h+1} - 1$ nodos

Arbol Binario - ejemplo

```
nodo* crear_hoja(int elem) {
  return new nodo({elem, nullptr, nullptr});
}

nodo* armar_arbol() {
  return new nodo({
    1,
    new nodo({
      1,
      crear_hoja(3),
      crear_hoja(5),
    }),
    new nodo({
      2,
      crear_hoja(8),
      crear_hoja(13),
    }),
  });
}
```

The file tree.pdf hasn't bee
Run ‘dot -Tpdf -o tree.pd
Or invoke LATEX with the -sh

Ejercicio: arbol de divisores

A partir de un número entero $N > 0$ de entrada, devolver un arbol binario tal que

- N esté en la raiz
- Si un nodo padre K **no es primo**, agregarle dos hijos $p > 1$ y $q > 1$ tales que $K = pq$. Si es primo, no agregarle hijos.
- Sugerencia: definir recursivamente `nodo* descomponer(int N)`

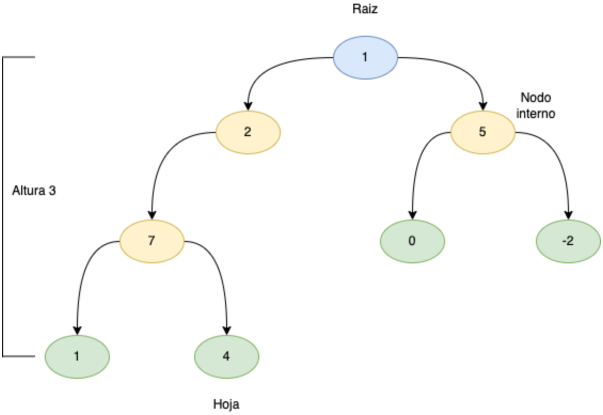
The file `descomponer.pdf` hasn't been created from `descomponer.dot` yet.
Run '`dot -Tpdf -o descomponer.pdf descomponer.dot`' to create it.
Or invoke LaTeX with the `-shell-escape` option to have this done automatic

Un arbol de divisores para $N = 2310$

Arbol Binario: recorrida **inorder**

Inorder recorre todos los nodos del árbol de la siguiente manera:

1. Primero recorrer el subarbol izquierdo,
2. Luego visitar la raíz,
3. Luego recorrer el subarbol derecho.



Inorder: `1, 7, 4, 2, 1, 0, 5, -2`

Arbol Binario – inorder

```
void inorder_imprimir(nodo* arbol) {
    if (arbol == nullptr)
      return;

    inorder_imprimir(arbol->izquierdo);
    cout << arbol->elem << endl;
    inorder_imprimir(arbol->derecho);
}
```

Arbol Binario – preorder

Preorder recorre todos los nodos del árbol de la siguiente manera:

1. Primero visitar la raíz,
2. Luego recorrer el subarbol izquierdo,
3. Luego recorrer el subarbol derecho.

```
void preorder_imprimir(nodo* arbol) {
    if (arbol == nullptr)
      return;

    cout << arbol->elem << endl;
    preorder_imprimir(arbol->izquierdo);
    preorder_imprimir(arbol->derecho);
}
```

Ejercicio: búsqueda de un elemento

Implementar una función `bool buscar(int n, nodo* arbol)` que devuelva `true` si el elemento n está presente en algún nodo del árbol.
Sugerencia: usar Divide & Conquer

Arbol Binario: búsqueda de un elemento

```
bool buscar(int n, nodo* arbol) {
  if (arbol == nullptr){
    return false;
  }
  else if (n == arbol->elem){
    return true;
  }
  else {
    return buscar(n,arbol->izquierdo)
        || buscar(n,arbol->derecho);
  }
}
```

The file buscar.pdf hasn't been
Run ‘dot -Tpdf -o buscar.pdf
Or invoke LaTeX with the -shell

¿Cuántas veces se accede a cada nodo en peor caso?
Una vez a cada nodo.
¿Qué complejidad de peor caso tiene buscar?
O(n) con n cantidad total de nodos

Motivación Árbol Binario de Búsqueda

Problema: Necesito almacenar elementos distintos ordenados, y agregar/quitar/buscar de manera eficiente:

- Agregar en $O(\log n)$
- Quitar en $O(\log n)$
- Buscar en $O(\log n)$

Idea: almacenarlos de manera conveniente en un arbol binario.

- El elemento de la raíz es mayor que los de la izquierda y menor que los de la derecha.
- Mantengo la altura cercana a $\log n$

Arbol Binario de Búsqueda

```
struct nodo {
    int elem;
    nodo* izquierdo = nullptr;
    nodo* derecho = nullptr;

//////////////////////////////////////////////////
// Rep: "ABB"
//      - Rep de arbol binario bien formado
//      - Para todo nodo n del arbol:
//          n.elem > todo elemento de n->izquierdo
//          n.elem < todo elemento de n->derecho
};
nodo* _raiz;
```

Arbol Binario de Búsqueda

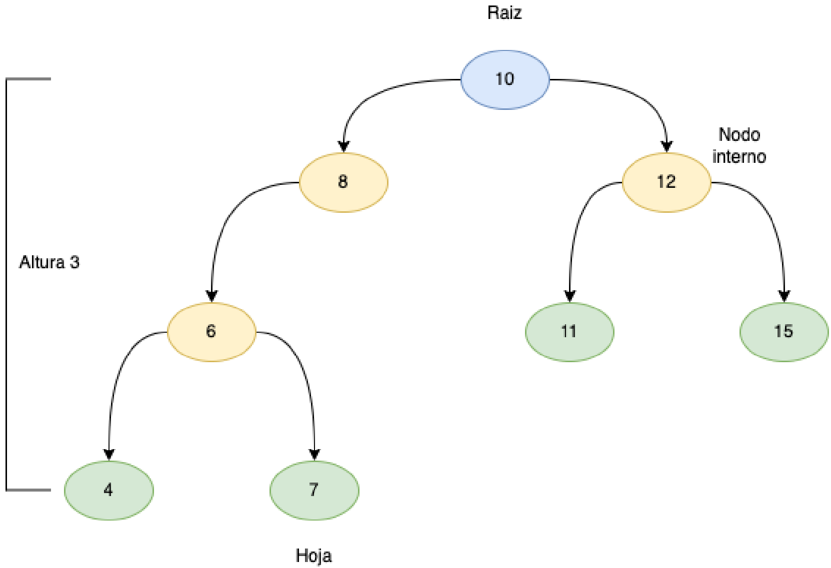


ABB: buscar

```
bool buscar(int n, nodo* abb) {
    if (abb == nullptr){
      return false;
    }
    else if (n == abb->elem){
      return true;
    }
    else if (n < abb->elem){
      return buscar(n, abb->izquierdo);
    }
    else if (n >= abb->elem){
      return buscar(n, abb->derecho);
    }
}
```

¿Qué complejidad de peor caso tiene buscar?
O(h) con h altura del árbol

ABB: insertar

```cpp
// devuelve nodo insertado
nodo* insertar(int n, nodo* abb) {
  if (abb == nullptr) {
    return new nodo(n);
  } else if (n < abb->elem)
      abb->izquierdo = insertar(n, abb->izquierdo);
    else if (n > abb->elem)
      abb->derecho = insertar(n, abb->derecho);

    // devuelve raiz sin modificar
    return abb;
} // Complejidad: O(h)
```

ABB: borrar

Si el nodo a borrar tiene un sólo hijo.

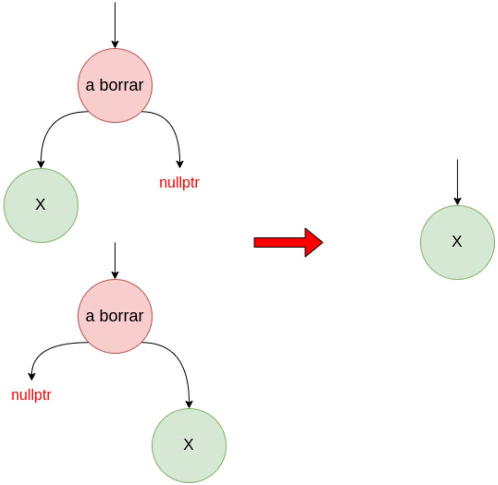


ABB: borrar

Si el nodo a borrar tiene dos hijos y su hijo derecho no tiene hijo izquierdo.

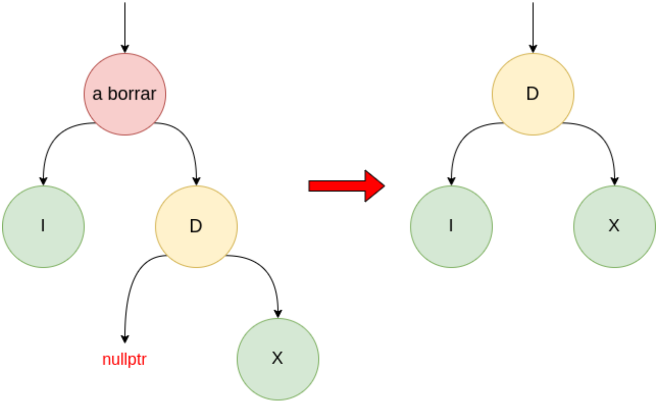


ABB: borrar

Si el nodo a borrar tiene dos hijos y su hijo derecho <u>sí tiene</u> hijo izquierdo.

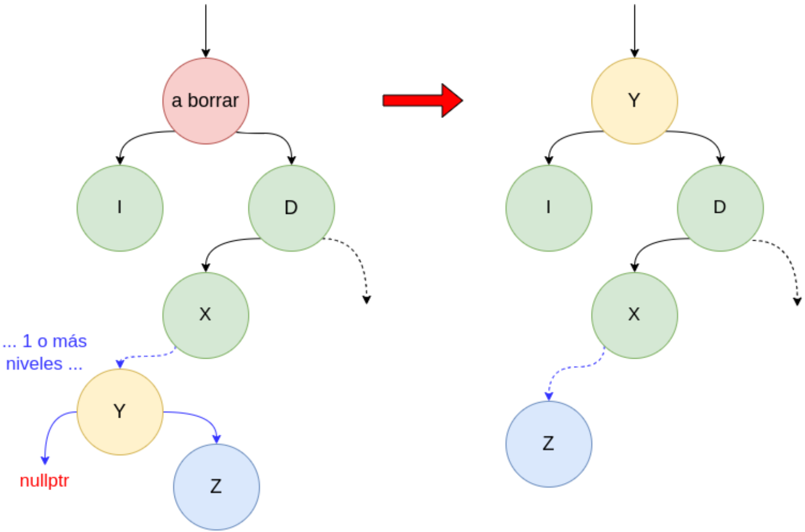


ABB: borrar

Busca recursivamente el nodo, lo borra y reconecta el arbol

```
// PRE: n está en abb
// POST: Elimina n y devuelve la raiz (o la nueva raiz si cambia)
nodo* borrar(int n, nodo* abb) {
  if (n < abb->elem) {
    // recursion izquierda: actualiza izquierdo
    abb->izquierdo = borrar(n, abb->izquierdo);
    return abb;
  } else if (n > abb->elem) {
    // recursion derecha: actualiza derecho
    abb->derecho = borrar(n, abb->derecho);
    return abb;
  } else {
    // calcula reemplazante y elimina nodo actual
    nodo* nueva_raiz = desconectar_raiz(abb);
    delete abb;
    return nueva_raiz;
  }
}
```

ABB: desconectar_raiz

Determina la nueva raiz para reemplazar a la actual y la devuelve

```
// PRE: abb no es nullptr
// POST: desconecta la raiz y devuelve la nueva raiz
nodo * desconectar_raiz(nodo* abb) {
  nodo * nueva_raiz;
  if (abb->izquierdo == nullptr)
      nueva_raiz = abb->derecho;          // caso (a)
  else if (abb->derecho == nullptr)
      nueva_raiz = abb->izquierdo;        // caso (b)
  else if (abb->derecho->izquierdo == nullptr) {
      nueva_raiz = abb->derecho;          // caso (c)
      nueva_raiz->izquierdo = abb->izquierdo;
  } else {
      // caso (d)
      nueva_raiz = quitar_y_devolver_minimo(abb->derecho); // O(h)
      nueva_raiz->izquierdo = abb->izquierdo;
      nueva_raiz->derecho = abb->derecho;
  }
  return nueva_raiz;
}
```

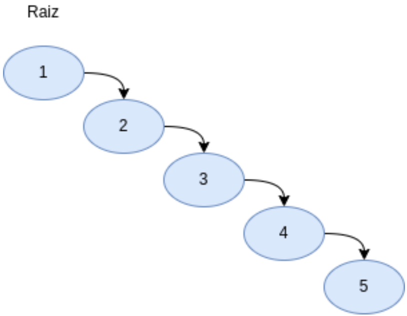
ABB: quitar_y_devolver_minimo

Busca mínimo del abb, lo quita y lo devuelve

```
// Pre: Asume que tiene hijo izquierdo
nodo* quitar_y_devolver_minimo(nodo* abb) {
  while (abb->izquierdo->izquierdo != nullptr) //O(h)
    abb = abb->izquierdo;
  // saco izquierdo, reconecto padre con derecho
  nodo* minimo = abb->izquierdo;
  abb->izquierdo = minimo->derecho;
  return minimo; // devuelvo el quitado
}
```

ABB degenerado

```
nodo* abb = nullptr;
abb = insertar(1,abb);
abb = insertar(2,abb);
abb = insertar(3,abb);
abb = insertar(4,abb);
abb = insertar(5,abb);
```



¡Se convirtió en una lista! buscar: complejidad $O(h) = O(n)$

Si fuera completo... ¡$O(h) = O(\log n)$!

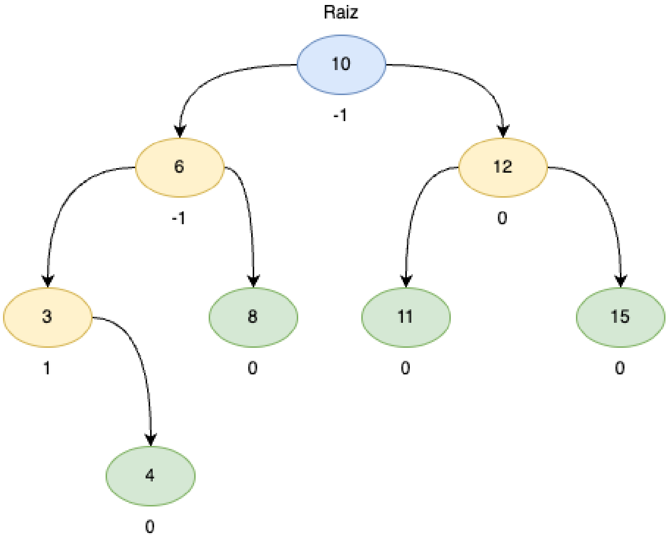
Pero no es razonable requerir que sea completo.

AVL: ABB balanceado

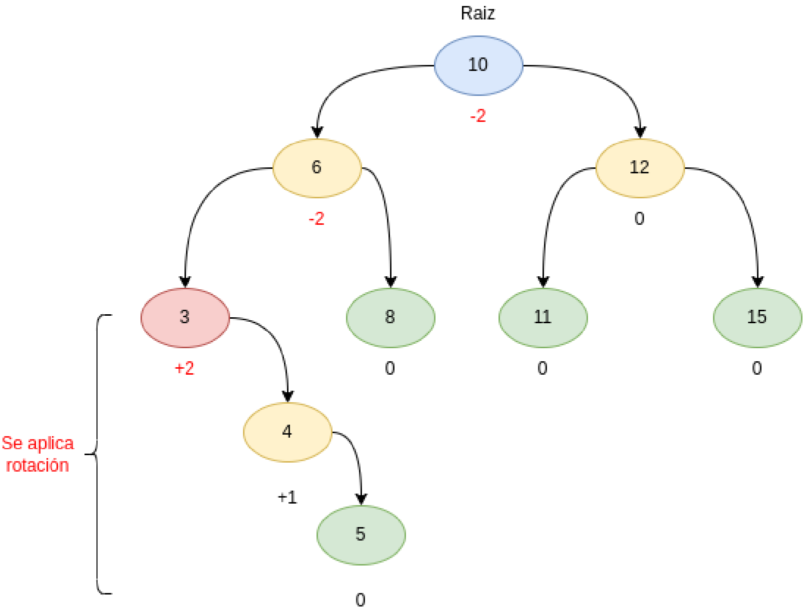
```
struct nodo {
    int elem;
    nodo* izquierdo = nullptr;
    nodo* derecho = nullptr;

///////////////////////////////////////////////////
// Rep: "ABB balanceado"
//      - Rep de ABB
//      - |altura(n.izquierdo) - altura(n.derecho)| <= 1
};
```

Asegura que la altura total es $O(\log n)$.
La diferencia de alturas se preserva rebalancenado el arbol al insertar y borrar

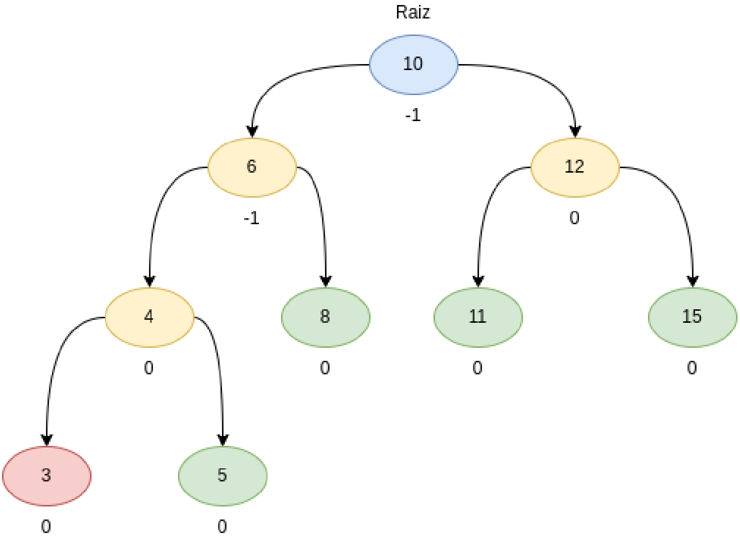
AVL: factor de balanceo



AVL: desbalanceo



AVL: rebalanceado



Arboles balanceados en la C++ std

En C++ `std`, `std::set` y `std::map` están implementados sobre árboles balanceados.
Es decir, sus complejidades de búsqueda/inserción/borrado son $O(h) = O(\log n)$ garantizadas.

El árbol usado en C++ es un **red-black tree**, que es una variante de arbol balanceado que asegura que desde ningún nodo ningún camino a las hojas es más del doble de largo que otro.

La bibliografía de la materia (Cormen) hace hincapié en red-black tree. Explicamos AVL por ser más intuitiva la idea.

Motivación para Heap

Queremos una estructura donde acceder al máximo elemento sea muy eficiente, pero no queremos pagar el costo de mantenerla siempre completamente ordenada.

Nos gustaría almacenar n elementos y poder:

- Acceder al máximo en $O(1)$
- Quitar el máximo (y recalcular máximo) en $O(\log n)$
- Agregar nuevo elemento (y recalcular máximo) en $O(\log n)$
- Buscar otros elementos: no es prioritario, $O(n)$.

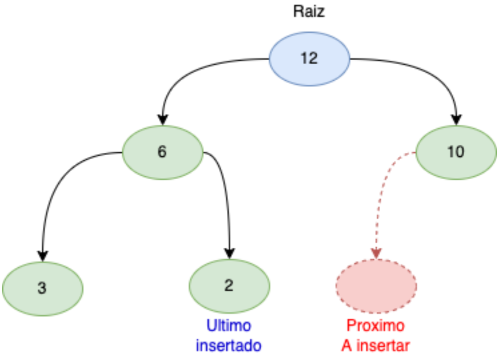
Idea: almacenarlos de manera conveniente en un arbol binario.

- En la raíz almacenamos el máximo
- Mantenemos la altura cercana a $\log n$

Arbol binario: Max Heap

Un árbol binario max-heap cumple el siguiente invariante:

- Cada nodo del árbol tiene un elemento mayor o igual al de todos sus descendientes **directos o indirectos**,
- Es izquierdista: todos los niveles están completos salvo el último, que se rellena de izquierda a derecha.

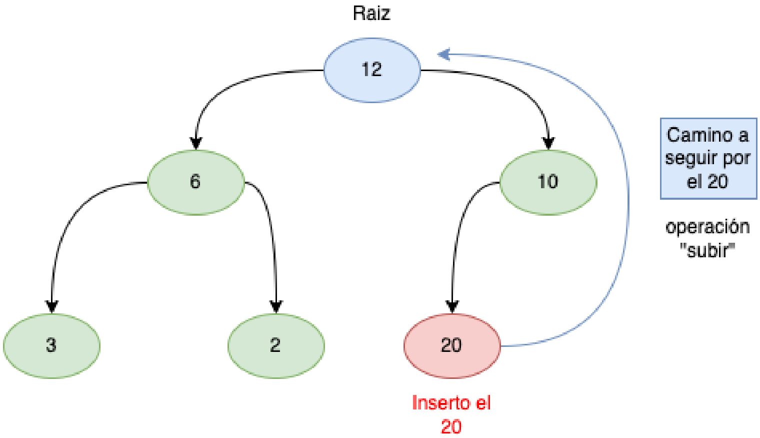


Notar que, como el árbol es izquierdista, su áltura es $\log n$.

Max Heap: Inserción (1)

Para insertar un elemento nuevo:

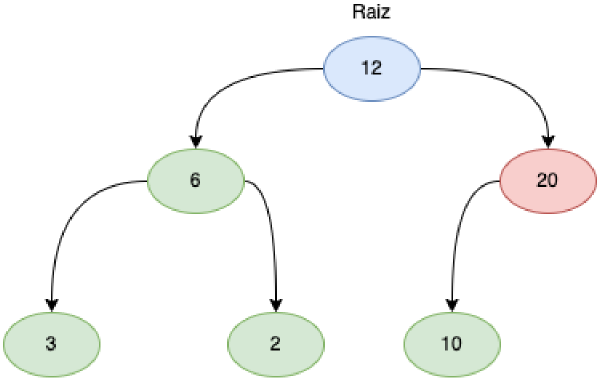
1. lo colocamos en la primera hoja libre,
2. lo vamos subiendo de nivel mientras sea mayor que su padre. (para mantener el invariante)



Max Heap: Inserción (2)

Para insertar un elemento nuevo:

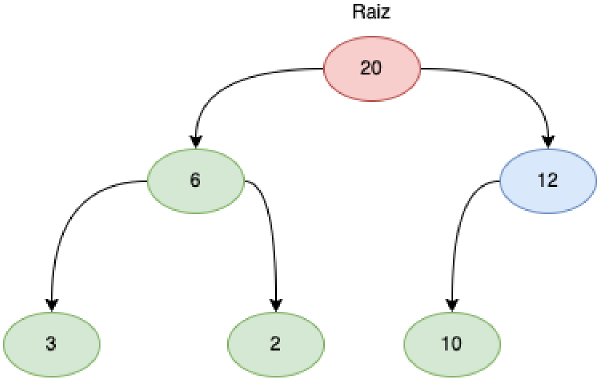
1. lo colocamos en la primera hoja libre,
2. lo vamos subiendo de nivel mientras sea mayor que su padre. (para mantener el invariante)



Max Heap: Inserción (3)

Para insertar un elemento nuevo:

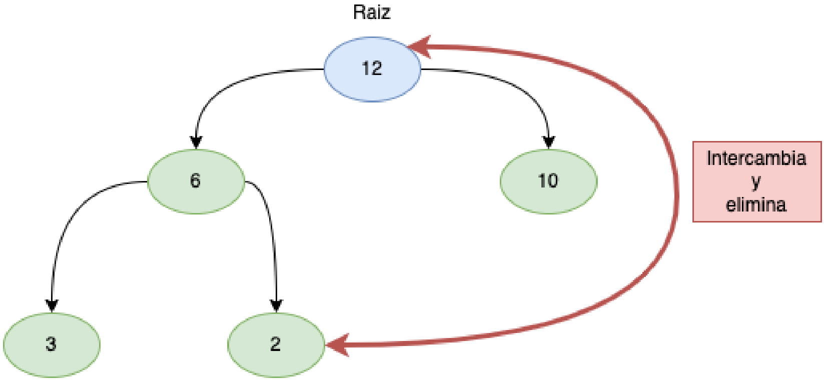
1. lo colocamos en la primera hoja libre,
2. lo vamos subiendo de nivel mientras sea mayor que su padre. (para mantener el invariante)



Max Heap: Eliminación (1)

Para eliminar el máximo:

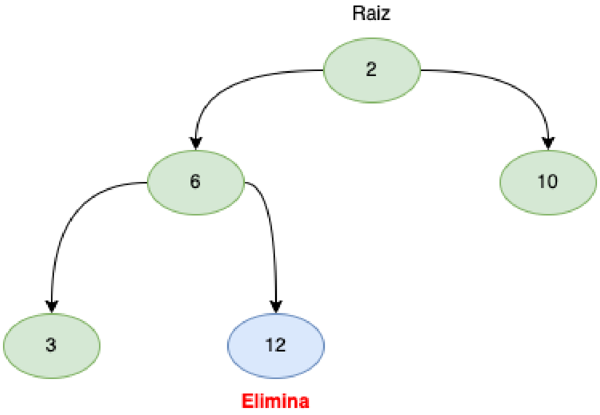
1. intercambiamos la raiz con la última hoja,
2. eliminamos la última hoja,
3. vamos bajando de nivel el elemento que quedo en la raíz mientra sea menor que algún hijo.
   (si es menor que ambos hijos lo bajamos en dirección al hijo más grande)



Max Heap: Eliminación (2)

Para eliminar el máximo:

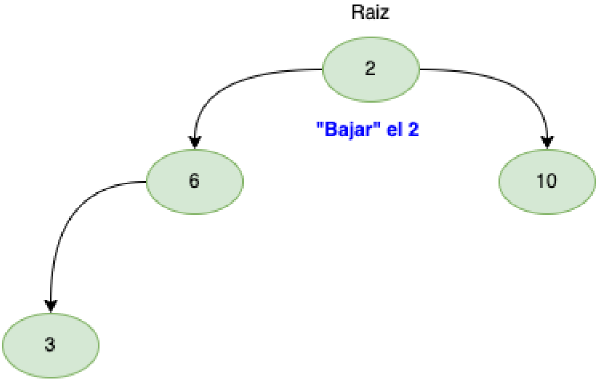
1. intercambiamos la raiz con la última hoja,
2. eliminamos la última hoja,
3. vamos bajando de nivel el elemento que quedo en la raíz mientra sea menor que algún hijo.
   (si es menor que ambos hijos lo bajamos en dirección al hijo más grande)



Max Heap: Eliminación (3)

Para eliminar el máximo:

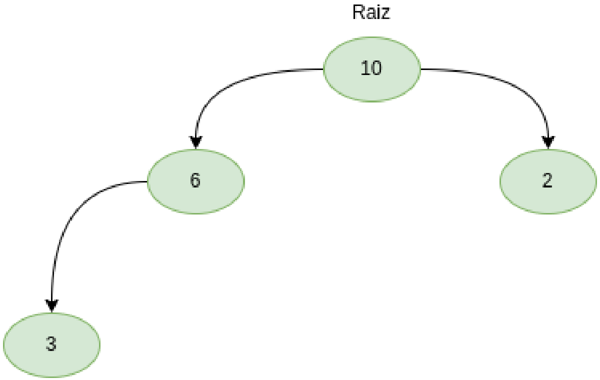
1. intercambiamos la raiz con la última hoja,
2. eliminamos la última hoja,
3. vamos bajando de nivel el elemento que quedo en la raíz mientra sea menor que algún hijo.
   (si es menor que ambos hijos lo bajamos en dirección al hijo más grande)



Max Heap: Eliminación (4)

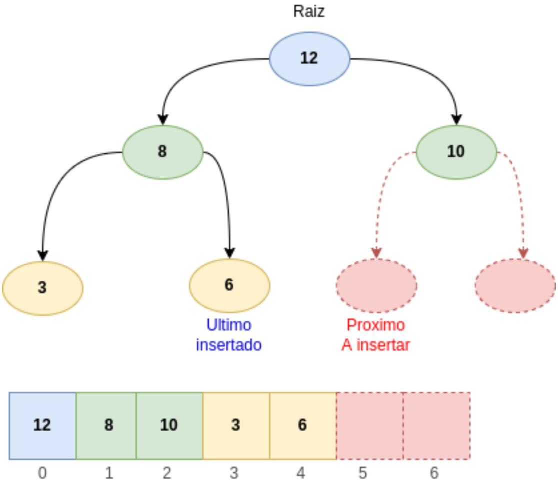
Para eliminar el máximo:

1. intercambiamos la raiz con la última hoja,
2. eliminamos la última hoja,
3. vamos bajando de nivel el elemento que quedo en la raíz mientra sea menor que algún hijo.
   (si es menor que ambos hijos lo bajamos en dirección al hijo más grande)

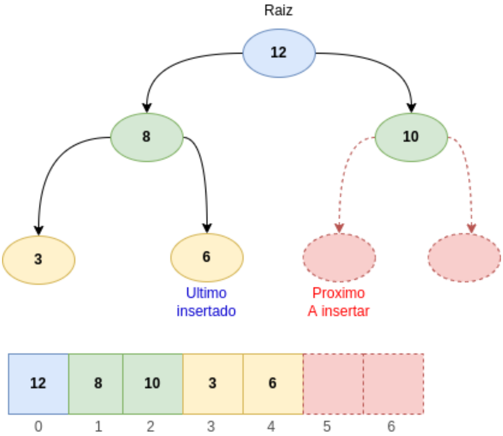


Arbol binario: Max Heap en vector

- Como el árbol binario heap es izquierdista, se puede representar con un vector.
- Se almacena cada nivel de arriba hacia abajo y de izquierda a derecha.



Arbol binario: Max Heap en vector



Notar que:

- La raíz está en `v[0]`.
- Los hijos del nodo `v[i]` están en `v[2i + 1]` y `v[2i + 2]`.
- El padre del nodo `v[i]` está en `v[(i - 1)/2]`.

Max Heap – Vector

```
  vector<int> _heap;
    // Rep: - para todo 1 <= i < _heap.size():
    //        - _heap[i] <= _heap[(i-1)/2]

int padre(int pos) {
  return (pos-1)/2;
}
int hijo_izq(int pos) {
  return 2*pos+1;
}
int hijo_der(int pos) {
  return 2*pos+2;
}
```

Para el caso del heap esta representación de árbol binario es más simple de implementar.

Heap: bajar

```
// hace descender por el heap al elemento en la ubicación pos
void bajar(vector<int>& heap, int pos) {
    int pos_bajar = max_hijo(heap, pos);
    if (pos_bajar != pos) {
      std::swap(heap[pos], heap[pos_bajar]);
      bajar(heap, pos_bajar);
  }
}
int max_hijo(const vector<int>& heap, int pos) {
  int ret = pos;   // default: raiz
  if (hijo_izq(pos) < heap.size() &&
        heap[ret] < heap[hijo_izq(pos)])
          ret = hijo_izq(pos); // elijo izquierdo
  if (hijo_der(pos) < heap.size() &&
        heap[ret] < heap[hijo_der(pos)]) {
          ret = hijo_der(pos); // elijo derecho
  return ret;
}
```

Ejercicio: escribir el algoritmo para `int subir(vector<int>& heap, int pos)`

Max Heap: insertar y eliminar

```cpp
void insertar_heap(int n, vector<int> & heap) {
  heap.push_back(n);                // agrega nuevo al final
  subir(heap,heap.size()-1);        // sube el último(*)
}

void eliminar_maximo(vector<int> & heap) {
  heap[0] = heap[heap.size()-1];  // pone último en la raiz
  heap.pop_back();                  // saca ultimo
  bajar(heap,0);                    // baja la raiz(*)
}
 // (*) para reestablecer el invariante de max-heap
```

Ejercicio: Heapsort

Ejercicio: implementar la función

`void heapsort(vector<int>& v)`

que, usando un max heap, ordene de manera **creciente** el vector `v` con complejidad $O(n \log n)$.

Necesitará implementar primero la operación faltante `subir`.

¡Bajar `T11-ejercicios.zip` del campus y a trabajar!

Heap en la C++ std

- La biblioteca estándar de `C++` provee la clase `std::priority_queue<T>` cuya estructura de representación es un max-heap.
- `https://en.cppreference.com/w/cpp/container/priority_queue`

T12: Resumen de hoy

Árboles Binarios (Cormen, Capítulo 10.4)

- Características
- Invariante de estructura
- Operaciones básicas

Árbol Binario de Búsqueda (Cormen, Capítulo 12)

- Estructura y Operaciones
- Problema de desbalanceo

Max-Heap (Cormen, Capítulo 6.1)

- Estructura y Versión sobre vector
- Algoritmos
- C++: `std::priority_queue`