

# Estructuras de datos: Árboles binarios de búsqueda, Montículos

## Algoritmos

Facultad de Informática  
Universidad de A Coruña



UNIVERSIDADE DA CORUÑA

# Table of Contents

- 1 Árboles binarios de búsqueda
- 2 Montículos

# Table of Contents

1 Árboles binarios de búsqueda

2 Montículos

# Preliminares

- El **camino** de un nodo  $n_1$  a otro  $n_k$  es la secuencia de nodos  $n_1, n_2, \dots, n_k$  tal que  $n_i$  es el padre de  $n_{i+1}$ .
- La **profundidad** de un nodo  $n$  es la longitud del camino entre la raíz y  $n$ .
  - La raíz tiene profundidad cero.
- Para un **árbol binario de búsqueda**, el valor medio de la profundidad es  $O(\log n)$ .
  - Si la inserción en un ABB no es aleatoria, el tiempo computacional aumenta.
    - Para mantener el equilibrio: Árboles AVL, Splay Trees, ...
- La **altura** de  $n$  es el camino más largo de  $n$  a una hoja.
  - La altura de un árbol es la altura de la raíz.

# Operaciones básicas

- **Buscar:** devuelve la posición del nodo con la clave  $x$ .
- **Insertar:** coloca la clave  $x$ . Si ya estuviese, no se hace nada (o se “actualiza” algo).
- **Eliminar:** borra la clave  $x$ .
  - Si  $x$  está en una hoja, se elimina de inmediato.
  - Si el nodo tiene un hijo, se ajusta un apuntador antes de eliminarlo.
  - Si el nodo tiene dos hijos, se sustituye  $x$  por la clave más pequeña,  $w$ , del subárbol derecho.
    - A continuación se elimina en el subárbol derecho el nodo con  $w$  (que no tiene hijo izquierdo)

# Eliminación perezosa

- Si se espera que el número de eliminaciones sea pequeño, la **eliminación perezosa** es una buena estrategia.
  - Al eliminar un elemento, se deja en el árbol **marcándolo** como eliminado.
  - Habiendo claves duplicadas, es posible decrementar el campo con la frecuencia de apariciones.
  - Si una clave eliminada se vuelve a insertar, se evita la sobrecarga de asignar un nodo nuevo.
- Si el número de nodos reales en el árbol es igual al número de nodos “eliminados”, se espera que la profundidad del árbol sólo aumente en uno (**¿por qué?**).
  - La penalización de tiempo es pequeña.

## Implementación de árboles binarios de búsqueda (i)

**tipo**

PArbol = ^Nodo

Nodo = **registro**

Elemento : TipoElemento

Izquierdo, Derecho : PArbol

**fin registro**

ABB = PArbol

**procedimiento** CrearABB (**var** A)

A := nil

**fin procedimiento**

## Implementación de árboles binarios de búsqueda (ii)

```
función Buscar (x, A) : PArbol
  si A = nil entonces devolver nil
  sino si x = A^.Elemento entonces devolver A
  sino si x < A^.Elemento entonces
    devolver Buscar (x, A^.Izquierdo)
  sino devolver Buscar (x, A^.Derecho)
fin función

función BuscarMin (A) : PArbol
  si A = nil entonces devolver nil
  sino si A^.Izquierdo = nil entonces devolver A
  sino devolver BuscarMin (A^.Izquierdo)
fin función
```

## Implementación de árboles binarios de búsqueda (iii)

```
procedimiento Insertar (x, var A)
  si A = nil entonces
    nuevo (A);
    si A = nil entonces error ``sin memoria''
    sino
      A^.Elemento := x;
      A^.Izquierdo := nil;
      A^.Derecho := nil
    sino si x < A^.Elemento entonces
      Insertar (x, A^.Izquierdo)
    sino si x > A^.Elemento entonces
      Insertar (x, A^.Derecho)
  { si x = A^.Elemento : nada }
fin procedimiento
```

## Implementación de árboles binarios de búsqueda (iv)

```
procedimiento Eliminar (x, var A)
  si A = nil entonces error ``no encontrado''
  sino si x < A^.Elemento entonces
    Eliminar (x, A^.Izquierdo)
  sino si x > A^.Elemento entonces
    Eliminar (x, A^.Derecho)
  sino { x = A^.Elemento }
    si A^.Izquierdo = nil entonces
      tmp := A; A := A^.Derecho; liberar (tmp)
    sino si A^.Derecho = nil entonces
      tmp := A; A := A^.Izquierdo; liberar (tmp)
    sino tmp := BuscarMin (A^.Derecho);
      A^.Elemento := tmp^.Elemento;
      Eliminar (A^.Elemento, A^.Derecho)
fin procedimiento
```

## Recorridos de un árbol (i)

- **En orden:** Se procesa el subárbol izquierdo, el nodo actual y, por último, el subárbol derecho.  $O(n)$

**procedimiento** Visualizar (A)

**si** A <> nil **entonces**

Visualizar (A^.Izquierdo);

Escribir (A^.Elemento);

Visualizar (A^.Derecho)

**fin procedimiento**

- **Post-orden:** Ambos subárboles primero.  $O(n)$

**función** Altura (A) : número

**si** A = nil **entonces devolver** -1

**sino devolver** 1 + max (Altura (A^.Izquierdo),  
Altura (A^.Derecho))

**fin función**

## Recorridos de un árbol (ii)

- **Pre-orden:** El nodo se procesa antes. Ej: una función que marcarse cada nodo con su profundidad.  $O(n)$
- **Orden de nivel:** Todos los nodos con profundidad  $p$  se procesan antes que cualquier nodo con profundidad  $p + 1$ .
  - Se usa una cola en vez de la pila implícita en la recursión.  $O(n)$

# Table of Contents

- 1 Árboles binarios de búsqueda
- 2 Montículos**

# Colas de prioridad

- Permiten únicamente el acceso al mínimo (o máximo) elemento.
- Operaciones básicas: insertar, eliminarMin (eliminarMax).
- Implementaciones simples:
  - Listas enlazadas efectuando inserciones al frente,  $O(1)$ , y recorriendo la lista,  $O(n)$ , para eliminar el mínimo (máximo).
  - Listas ordenadas: inserciones costosas,  $O(n)$ , eliminaciones eficientes,  $O(1)$ .
  - Árboles binarios de búsqueda: tiempo de ejecución medio  $O(\log n)$  para ambas operaciones.
    - A pesar de que las eliminaciones no son aleatorias.
    - Se eliminan repetidamente nodos de un subárbol. No obstante, el otro subárbol es aleatorio y tendría a lo sumo el doble de elementos de los que debería. Y esto sólo incrementa en uno la profundidad esperada.
  - **Montículos**: ambas operaciones se realizan en  $O(\log n)$  para el peor caso. No requieren apuntadores.

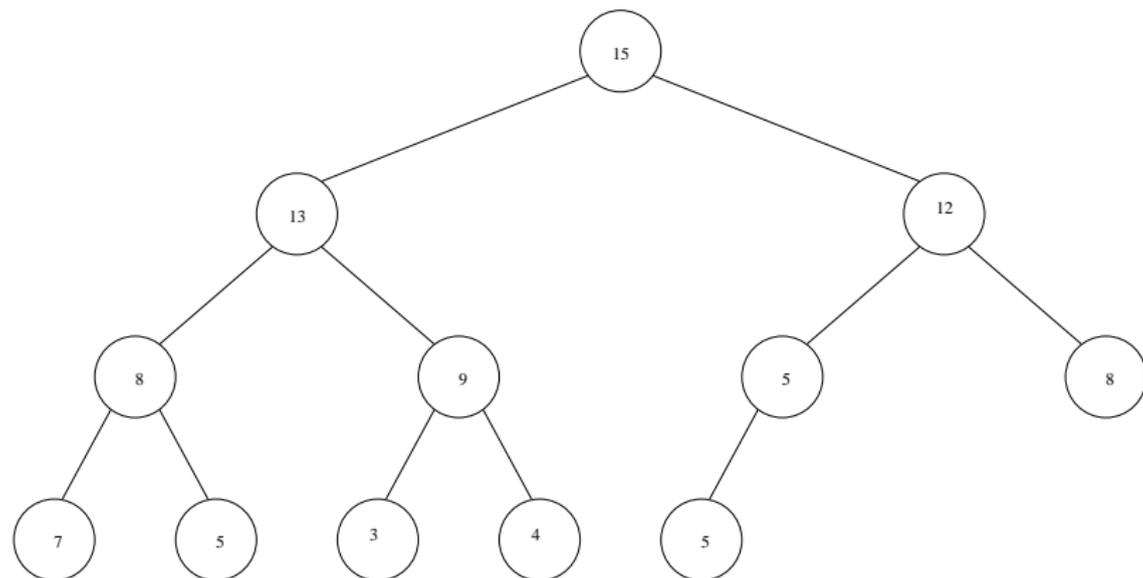
# Propiedades estructurales de los montículos

- Un montículo es un **árbol binario completo**: todos los niveles están llenos con la posible excepción del nivel más bajo, que se llena de izquierda a derecha.
- Un árbol binario completo de altura  $h$  tiene entre  $2^h$  y  $2^{h+1} - 1$  nodos.
  - Su altura es la parte entera de  $\log_2 n$ .
- Esta regularidad facilita su representación mediante un vector.
- Para cualquier elemento en la posición  $i$  del vector, el hijo izquierdo está en la posición  $2i$ , el hijo derecho en  $2i + 1$ , y el padre en  $i \div 2$ .

# Propiedades de orden de los montículos

- El mínimo (o máximo) está en la raíz.
- Y como todo subárbol es también un montículo, todo nodo debe ser menor (mayor) o igual que todos sus descendientes.

## Ejemplo de montículo de máximos



15	13	12	8	9	5	8	7	5	3	4	5
1	2	3	4	5	6	7	8	9	10	11	12

## Implementación de montículos (i)

```
tipo Montículo = registro  
    Tamaño_monticulo : 0..Tamaño_máximo  
    Vector_montículo : vector [1..Tamaño_máximo]  
                        de Tipo_elemento  
  
fin registro  
  
procedimiento Inicializar_Montículo ( M )  
    M.Tamaño_monticulo := 0  
fin procedimiento  
  
función Montículo_Vacío ( M ) : test  
    return M.Tamaño_monticulo = 0  
fin función
```

## Implementación de montículos (ii)

```
procedimiento Flotar ( M, i ) { privado }  
  mientras i > 1 y  
    M.Vector_montículo[i div 2] < M.Vector_montículo[i]  
  hacer intercambiar M.Vector_montículo[i div 2] y  
    M.Vector_montículo[i];  
    i := i div 2  
  fin mientras  
fin procedimiento  
procedimiento Insertar ( x, M )  
  si M.Tamaño_monticulo = Tamaño_máximo entonces  
    error Monticulo lleno  
  sino M.Tamaño_monticulo := M.Tamaño_monticulo + 1;  
    M.Vector_monticulo[M.Tamaño_monticulo] := x;  
    Flotar ( M, M.Tamaño_monticulo )  
fin procedimiento
```

## Implementación de montículos (iii)

```
procedimiento Hundir ( M, i ) { privado }  
  repetir  
    HijoIzq := 2*i;  
    HijoDer := 2*i+1;  
    j := i;  
    si HijoDer <= M.Tamaño_monticulo y  
      M.Vector_montículo[HijoDer] > M.Vector_montículo[i]  
    entonces i := HijoDer;  
    si HijoIzq <= M.Tamaño_monticulo y  
      M.Vector_montículo[HijoIzq] > M.Vector_montículo[i]  
    entonces i := HijoIzq;  
    intercambiar M.Vector_montículo[j] y  
      M.Vector_montículo[i];  
  hasta j=i {Si j=i el nodo alcanzó su posición final}  
fin procedimiento
```

## Implementación de montículos (iv)

```
función EliminarMax ( M ) : Tipo_elemento
  si Monticulo_Vacío ( M ) entonces
    error Monticulo vacío
  sino
    x := M.Vector_montículo[1];
    M.Vector_montículo[1] :=
      M.Vector_montículo[M.Tamaño_monticulo];
    M.Tamaño_monticulo := M.Tamaño_monticulo - 1;
    si M.Tamaño_monticulo > 0 entonces
      Hundir ( M, 1);
    devolver x
fin función
```

# Implementación de montículos (v)

- Creación de montículos en tiempo lineal,  $O(n)$ :

```
procedimiento Crear_Montículo ( V[1..n], M )  
  Copiar V en M.Vector_montículo;  
  M.Tamaño_montículo := n;  
  para i := M.Tamaño_montículo div 2 hasta 1 paso -1  
    Hundir(M, i);  
  fin para  
fin procedimiento
```

- El número de intercambios está acotado por la **suma de las alturas** de los nodos.
- Se demuestra mediante un argumento de marcado del árbol.
  - Para cada nodo con altura  $h$ , marcamos  $h$  aristas:
    - bajamos por la arista izquierda y después sólo por aristas derechas.
    - Así una arista nunca se marca 2 veces.