Estructuras de Datos:
Colas de prioridad

Algoritmos y Estructuras de Datos II

Motivación

La idea es la implementación general y flexible del concepto de atención ordenada según algún orden que se desprende de una prioridad.

Ejemplos: 1.- la cola en la panadería; el órden es el de los numeritos () salvo que una mujer embarazada, una persona con un chico en brazos, una/un anciana/o o una persona con discapacidad, 2.- asignación del procesador a procesos en un sistema operativo, 3.- *scheduling* de tareas en general, etc.

Idea

- Implementar en la forma más eficiente posible las operaciones de **encolar** (agrega un elemento a la cola), **próximo** (obtiene el próximo elemento a ser atendido) y **desencolar** (elimina el próximo elemento a ser atendido de la cola)

- La prioridad se suele expresar como una relación de órden lineal (<T) entre los elementos de tipo T que colocamos en la cola

Representación

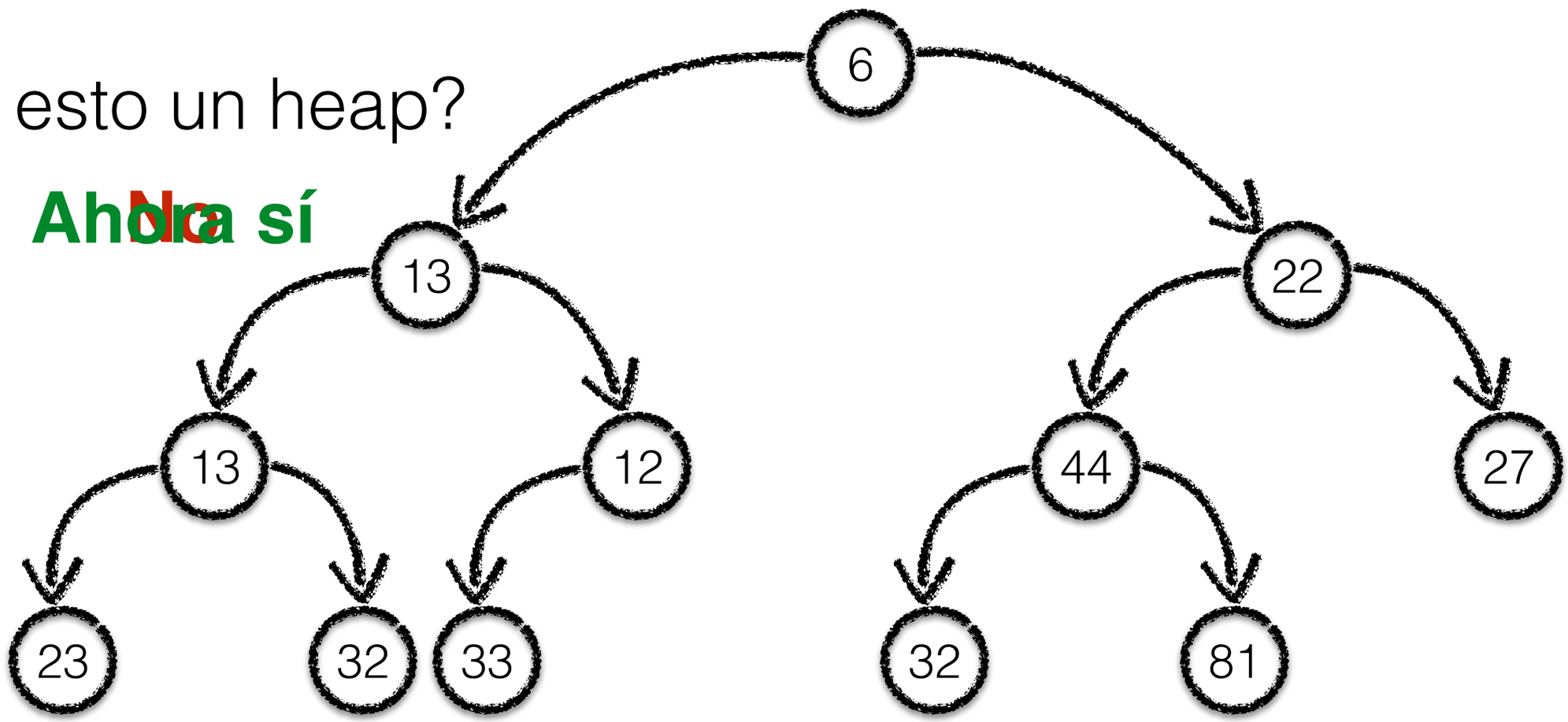
La forma más eficiente que se conoce a efectos de implementar este comportamiento es a través del uso de un *Heap* (Montón o Parva)

Heaps sobre árboles

Se utilizan árboles binarios con invariante de *heap* que dice que la prioridad de un nodo es mayor/menor o igual que las de sus hijos, si los tiene.

¿Es esto un heap?

Ahora sí



(Ideas para los) Algoritmos

- **Próximo**: obtiene el próximo elemento a ser atendido

- **Encolar**: agrega un elemento a la cola

- **Desencolar**: elimina el próximo elemento a ser atendido de la cola

- **Heapify**: dado un árbol cualquiera lo transforma en heap

(Ideas para los) Algoritmos

- **Próximo**: obtiene el próximo elemento a ser atendido

O(1)

```
T proximo (ab[T]: a){
    return raiz(a);
}
```

(Ideas para los) Algoritmos

- **Encolar**: agrega un elemento a la cola

O(log n)

```
void encolar (ab[T]: a, T: e){
    [colocar el elemento e en la base del árbol a como nodo b]
    upheap (ab[T]: b)
}

void upheap (ab[T]: b){
    if (padre(b) == b) then return; //La raiz del árbol se tiene a sí misma como padre.
    else
        if (raiz(b) > raiz(padre(b))) then {
            aux = raiz(padre(b)); raiz(b) = raiz(padre(b)); raiz(padre(b)) = aux;
            upheap(padre(b));
        }
}
```

(Ideas para los) Algoritmos

- **Desencolar**: elimina el próximo elemento a ser atendido de la cola

O(log n)

```
void desencolar (ab[T]: a){
    [colocar el último elemento del árbol a como raiz de a]
    downheap (ab[T]: a)
}

void downheap (ab[T]: a){
    if (vacio?(izq(a)) && vacio?(der(a))) then return;
    else
        if (!vacio?(izq(a)) && !vacio?(der(a))) then
            if (raiz(izq(a)) > raiz(der(a))) then {
                aux = raiz(a); raiz(a) = raiz(izq(a)); raiz(izq(a)) = aux; downheap (izq(a));
            } else {
                aux = raiz(a); raiz(a) = raiz(der(a)); raiz(der(a)) = aux; downheap (der(a));
            }
        else {
            if (!vacio?(izq(a)) && raiz(a) < raiz(izq(a))) then {
                aux = raiz(a); raiz(a) = raiz(izq(a)); raiz(izq(a)) = aux; downheap (izq(a)); }
            }
            if (!vacio?(der(a)) && raiz(a) < raiz(der(a))) then {
                aux = raiz(a); raiz(a) = raiz(der(a)); raiz(der(a)) = aux; downheap (der(a)); }
            }
        }
}
```

(Ideas para los) Algoritmos

- **Heapify (algoritmo simple)**: dado un árbol cualquiera lo transforma en heap

O(n log n)

```
void heapify (ab[T]: a, ab[T]: b){
    if vacio?(a) then
        return;
    else {
        encolar (b, raiz (a));
        heapify (izq(a), b);
        heapify (der(a), b);
    }
}
```

(Ideas para los) Algoritmos

- **Heapify (algoritmo de Floyd)**: dado un árbol cualquiera lo transforma en heap

```
void heapify (ab[T]: a){
    if vacio?(a) then
        return;
    else {
        if (!vacio?(izq(a)) && !vacio?(der(a))) then {
            heapify (izq(a)); heapify (der(a));
            if (raiz(izq(a)) < raiz(der(a))) then
                if (raiz(a) < raiz(der(a))) then {
                    aux = raiz(a); raiz(a) = raiz(der(a)); raiz(der(a)) = aux; downheap (der (a));
                }
            else
                if (raiz(a) < raiz(izq(a))) then {
                    aux = raiz(a); raiz(a) = raiz(izq(a)); raiz(izq(a)) = aux; downheap (izq (a));
                }
        } else {
            [ Se analizan los casos cuando solo uno de los subárboles no es vacío. ]
        }
}
```

Complejidad del algoritmo de Floyd

- Cada nodo es bajado, en peor caso, log2 (n) menos el nivel en el que se encuentra el nodo

- Si el árbol está balanceado, en el nivel i hay 2^i nodos

$$\sum_{i=0}^{\log n} 2^i \cdot (\log n - i) =$$

$$= \sum_{i=0}^{\log n} 2^{\log n - i} \cdot i =$$

$$= \sum_{i=0}^{\log n} \frac{2^{\log n}}{2^i} \, i =$$

$$= \sum_{i=0}^{\log n} \frac{n}{2^i} \, i =$$

$$= n \sum_{i=0}^{\log n} \frac{i}{2^i} =$$

$$\leq n \, \frac{3}{2} \in O(n)$$

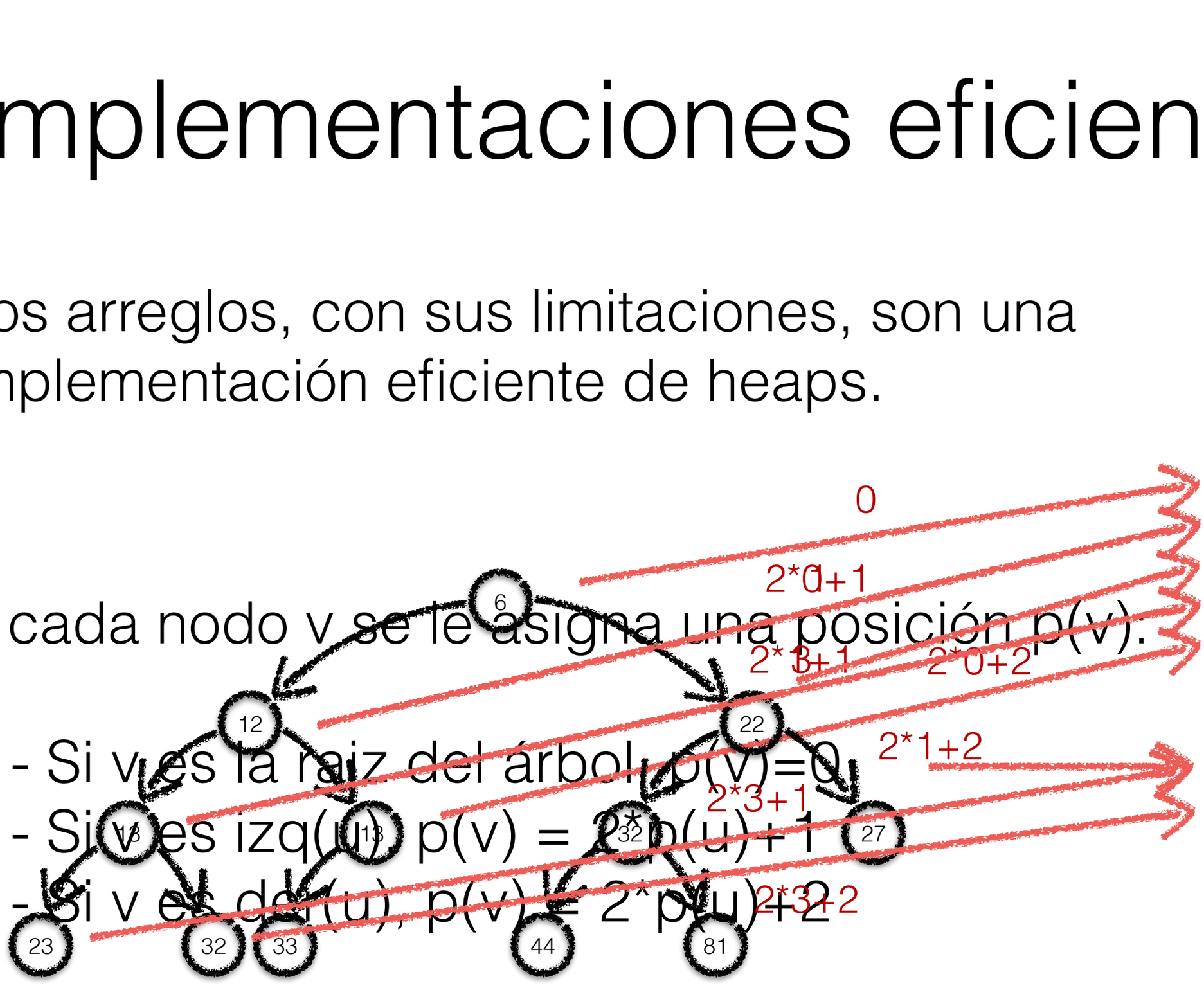
$$\sum_{i=0}^{\infty} \frac{i}{2^i} = \frac{3}{2}$$

Implementaciones eficientes

Los arreglos, con sus limitaciones, son una implementación eficiente de heaps.

A cada nodo v se le asigna una posición p(v).

- Si v es la raiz del árbol, p(v)=0
- Si v es izq(u), p(v) = 2*p(u)+1
- Si v es der(u), p(v) = 2*p(u)+2



0	6
1	12
2	22
3	13
4	13
5	32
6	27
7	23
8	32
9	33
10	X
11	44
12	81
13	X
14	X

Repaso

- Estudiamos implementaciones de las colas de prioridad que permiten aprovechar órdenes de complejidad temporal apropiados para las operaciones del TAD

- Mostramos cómo los arreglos nos pueden brindar una implementación eficiente de árboles a los efectos

¡Es todo por hoy!