Estructuras de Datos y Algoritmos

Colas con prioridad, montículos y *heapsort*

Lección 18

Universidad Zaragoza

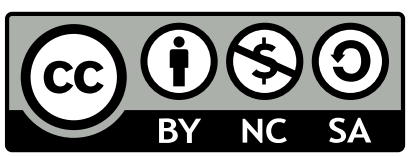
Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, España

Curso 2024/2025

Grado en Ingeniería Informática

Universidad de Zaragoza

Aula 0.04, Edificio Agustín de Betancourt



Adaptadas de diapositivas de Javier Campos

Índice

Universidad Zaragoza

Índice

Universidad Zaragoza

Colas con prioridad

Concepto

- Se pueden añadir elementos en cualquier orden
- Los elementos se extraen por orden según su prioridad
- "*Cola de espera*", pero el valor de cada elemento establece su "prioridad" (por ejemplo, el máximo valor significa máxima prioridad)
  - El primero en salir es el elemento de mayor prioridad
  - Ejemplo: cola en los servicios de urgencia en hospitales
- Elementos de cualquier tipo, con relación de orden total para el campo que exprese la "prioridad"
- Puede ser ***cola con prioridad de mínimos*** o ***cola con prioridad de máximos***

Universidad Zaragoza

Colas con prioridad

TAD cola con prioridad

- Colección de elementos con prioridad, con (al menos) las operaciones:
  - Inserción de *un* elemento
  - Observación de *un* elemento de valor máximo[1] (= máxima prioridad)
  - Eliminación de *un* elemento con valor máximo[1]

[1] Cola con prioridad de máximos

Colas con prioridad

```
espec colasConPrioridadesDeMáximos
parámetro formal
    género elem
    operación _≤_: elem e1, elem e2 -> booleano
    {≤ es una relación de orden total, a≤b significa que b tiene más prioridad que a}
fpf
género cp {Colección de elementos con prioridad}
operaciones
    crear: -> cp
    {Devuelve una cola vacía, sin elementos}
    añadir: cp c, elem e -> cp
    {Devuelve la cola resultante de añadir el elemento e a la cola c}
    esVacía?: cp c -> booleano
    {Devuelve verdad si y sólo si c no tiene elementos}
    parcial máx: cp c -> elem
    {Devuelve un elemento máximo de c.
    Parcial: la operación no está definida si c es vacía}
    eliminarMáx: cp c -> cp
    {Si c es vacía, devuelve c. Si no, devuelve una cola igual a la
    resultante de eliminar de c un elemento máximo.}
fespec
```

Cola de máximos

(si tomamos ≥ tendríamos de mínimos)

Colas con prioridad

Implementaciones

- Lista no ordenada:
  - `añadir`, `crear` y `esVacía?` con coste constante
  - `máx` y `eliminarMáx` con coste lineal
- Lista ordenada (de mayor a menor):
  - `añadir` con coste lineal
  - `máx`, `eliminarMáx`, `esVacía?` y `crear` con coste constante
- Árbol AVL:
  - `añadir`, `máx` y `eliminarMáx`, coste $O(\log N)$
  - `esVacía?` y `crear`, con coste constante
- **Montículo** (*heap*):
  - Implementación eficiente para colas con prioridad (**se representa en memoria estática**, con un vector)

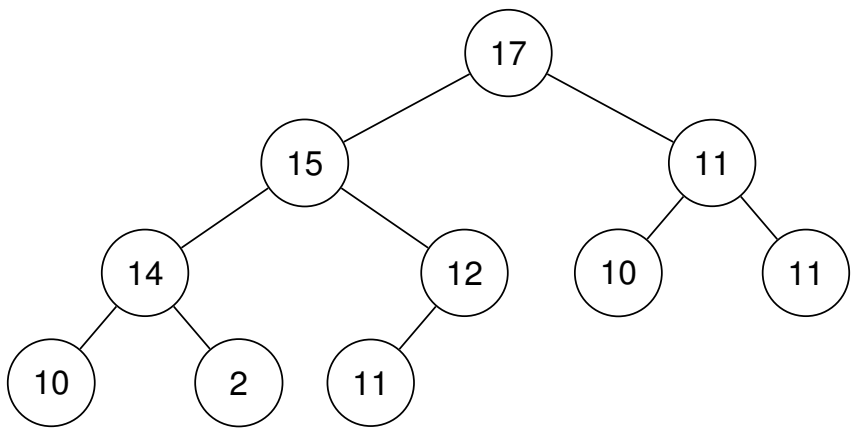
Universidad Zaragoza

Índice

Universidad Zaragoza

Montículos
Definiciones

- Un árbol binario se dice *parcialmente ordenado* si verifica las siguientes propiedades (**propiedad del montículo**):
  - El elemento raíz es mayor o igual[2] que el resto de los elementos del árbol, y
  - Los subárboles izquierdo y derecho son árboles binarios parcialmente ordenados



[2] Árbol parcialmente ordenado "de máximos" (la versión "de mínimos" es análoga).

Montículos
Definiciones

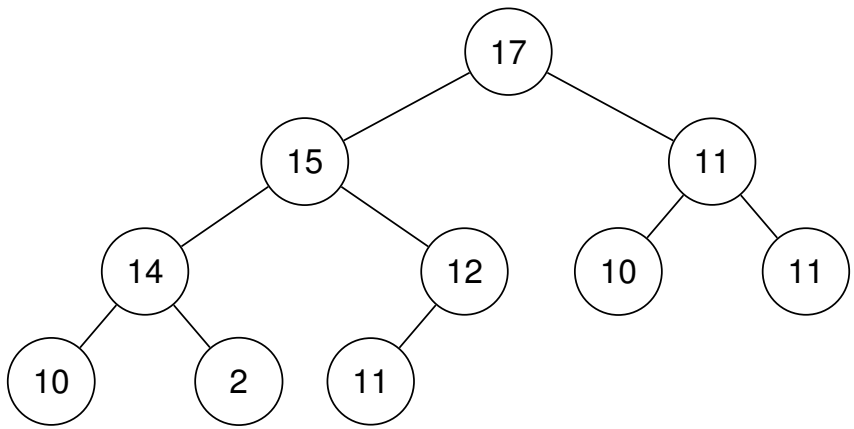
- Un árbol n-ario se dice ***homogéneo*** si todos sus subárboles, excepto las hojas, tienen *n* hijos
- Un árbol es ***completo*** cuando es homogéneo y todas sus hojas tienen la misma profundidad
- Un árbol se dice ***casi-completo*** (también llamado ***semicompleto***) cuando se puede obtener a partir de un árbol completo eliminando hojas consecutivas del último nivel, comenzando por la que está más a la derecha

Montículos

Definiciones

Montículo de máximos: árbol binario casicompleto donde

- El elemento raíz es mayor o igual que todos los elementos en el hijo izquierdo y en el derecho, y
- Ambos hijos son a su vez montículos de máximos



- La definición “de mínimos” es análoga
- Un applet que permite construir montículos tanto de mínimos como de máximos: http://people.ksp.sk/~kuko/bak/

Montículos

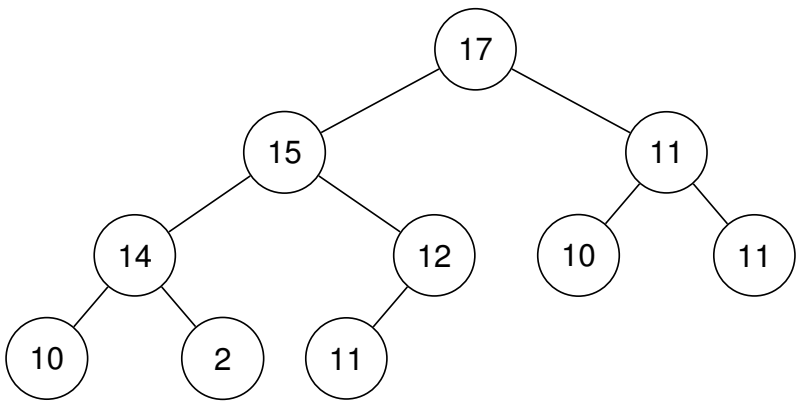
- **Representación de la cola con prioridades como un montículo**:
  - Operaciones `crear`, `máx` y `esVacía?`, serán de coste constante
  - Operaciones `añadir` y `eliminarMáx`: requieren recorrer un camino desde la raíz hasta una hoja (o a la inversa)
    - Como el montículo es casi-completo, su altura es de orden logarítmico en el número de elementos del árbol
    - Por tanto, se podrá garantizar un coste de ese orden, en el caso peor, para `añadir` y `eliminarMáx`

Montículos

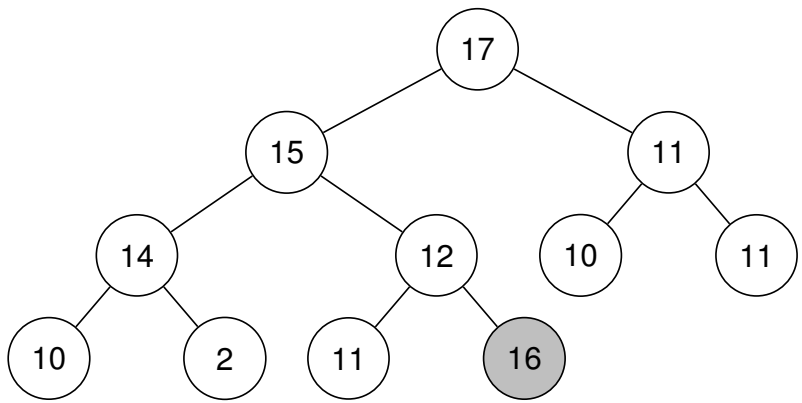
Operación `añadir`

Pasos para añadir un nuevo elemento en un montículo:

1. Se coloca el nuevo elemento lo más a la derecha posible en el nivel más bajo



Inserción de un nuevo elemento (de prioridad 16)



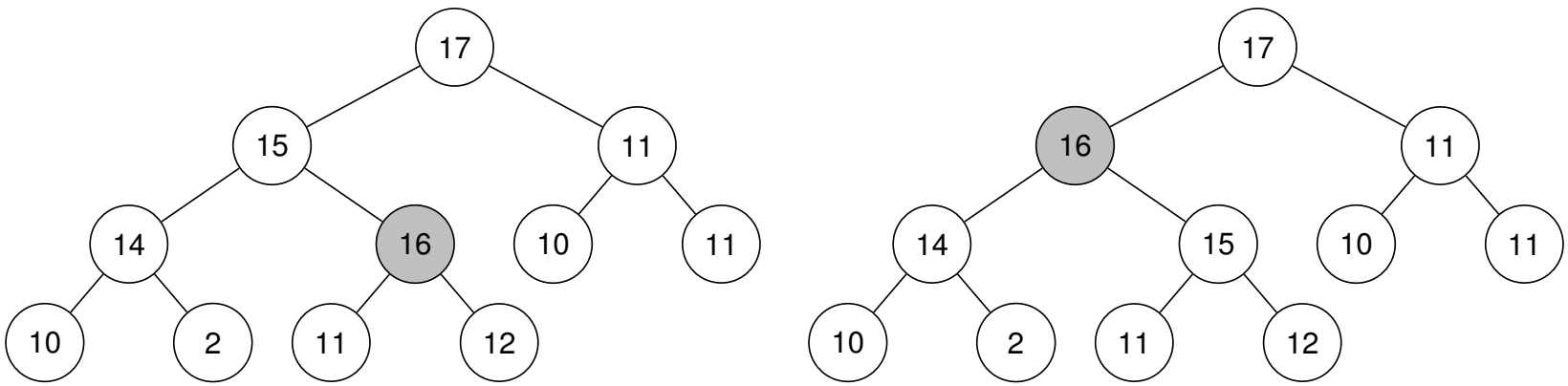
Paso 1: se coloca como siguiente hoja

Montículos

Operación `añadir`

Pasos para añadir un nuevo elemento en un montículo:

2. Si el nuevo elemento es mayor que su padre, se intercambian; y se repite este proceso (comparar con el padre e intercambiar si es mayor) hasta llegar a la raíz o alcanzar una posición en que sea menor o igual que su padre



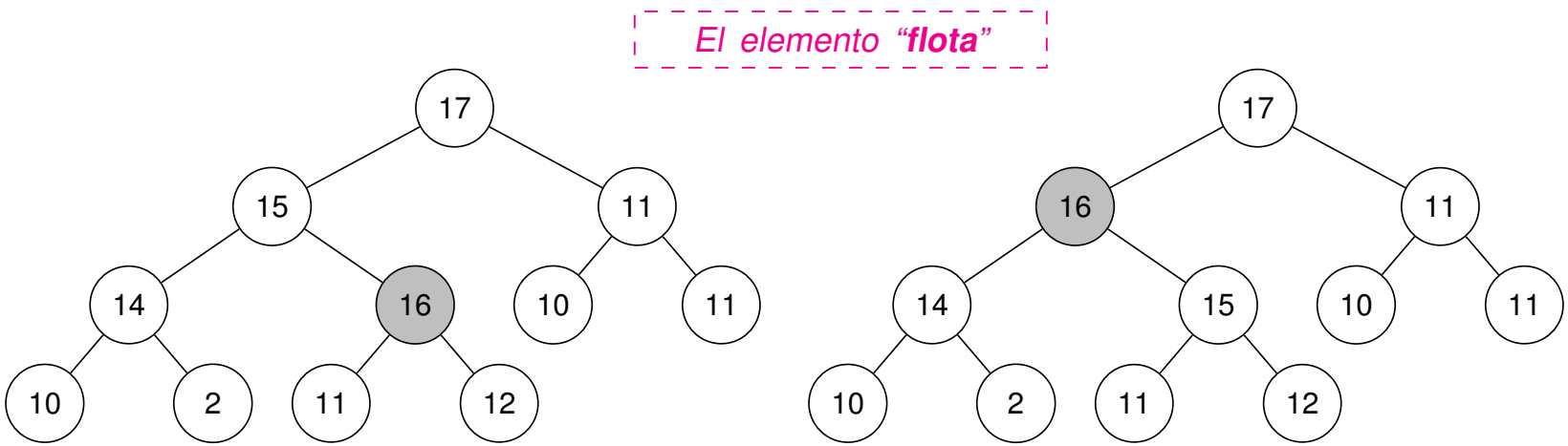
Paso 2: el elemento va subiendo hasta ser menor o igual (en prioridad) que su padre

Montículos

Operación `añadir`

Añadir un nuevo elemento en un montículo:

- Si es posible:
  - Acceder a la que será la posición de la nueva hoja con coste $O(1)$; y
  - Acceder desde cualquier elemento a su padre con coste $O(1)$
- Entonces el tiempo de la inserción es proporcional a la distancia que el nuevo elemento debe ascender en el árbol, que está acotada por $\log(N)$, siendo N el número de elementos del árbol

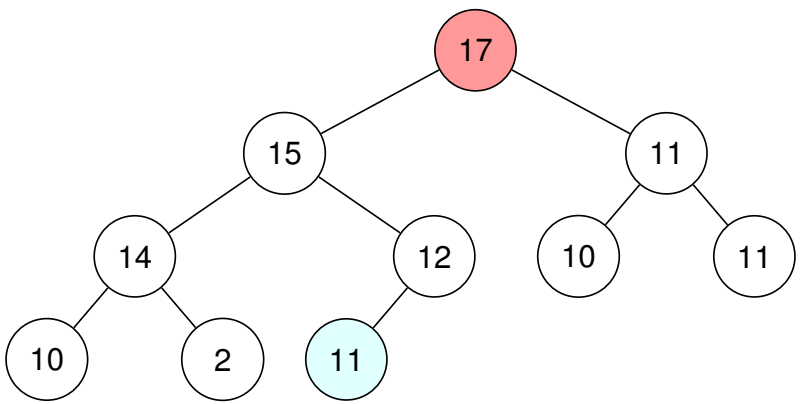


Paso 2: el elemento va subiendo hasta ser menor o igual (en prioridad) que su padre

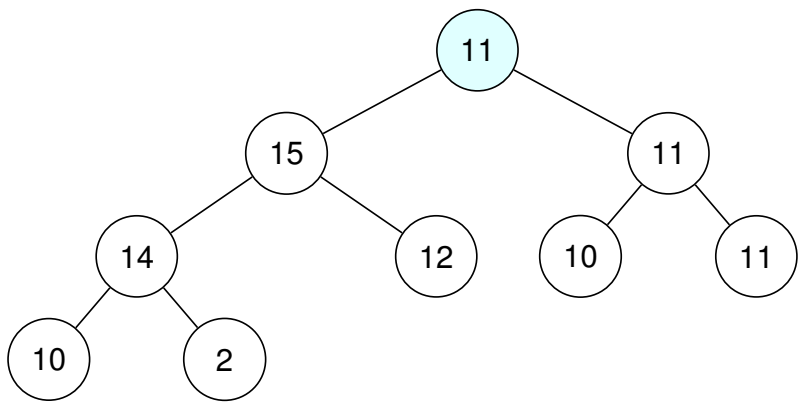
Montículos

Operación `eliminarMáx`

- El máximo elemento del montículo está en la raíz
- Pasos a seguir:
  1. Tomar la hoja de más a la derecha, del nivel más bajo del árbol, y colocarla provisionalmente en la raíz sustituyendo al previo máximo



Borrado del elemento máximo

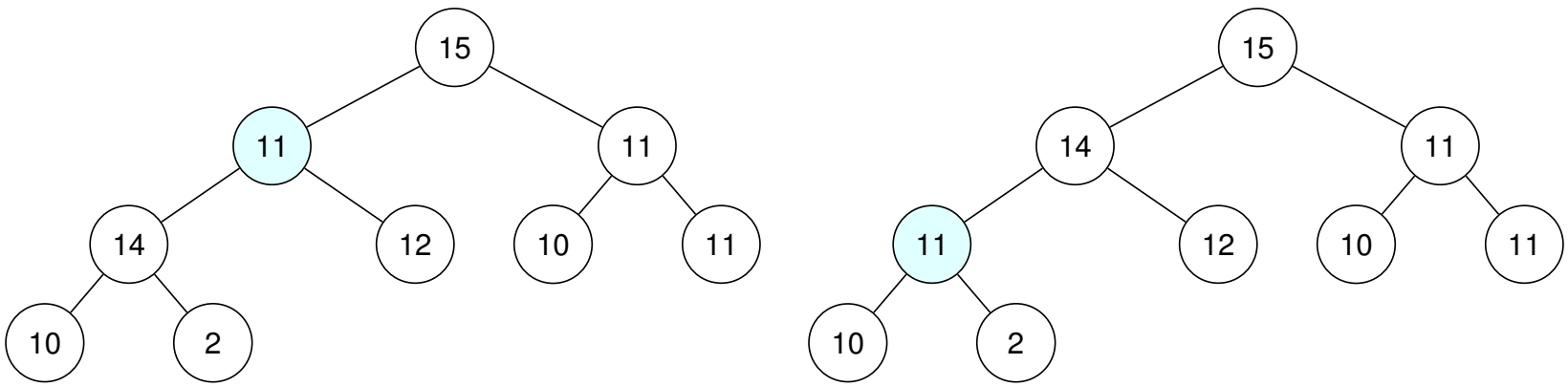


Paso 1: la hoja última pasa a ser raíz

Montículos

Operación `eliminarMáx`

- Pasos a seguir:
  2. El nuevo elemento en la raíz se compara con sus hijos y, si es menor que ellos (uno o ambos) se intercambia con el mayor de ellos; se repite el proceso de comparar con los hijos e intercambiar, hasta que esté en una hoja o sus hijos sean menores o iguales que el elemento



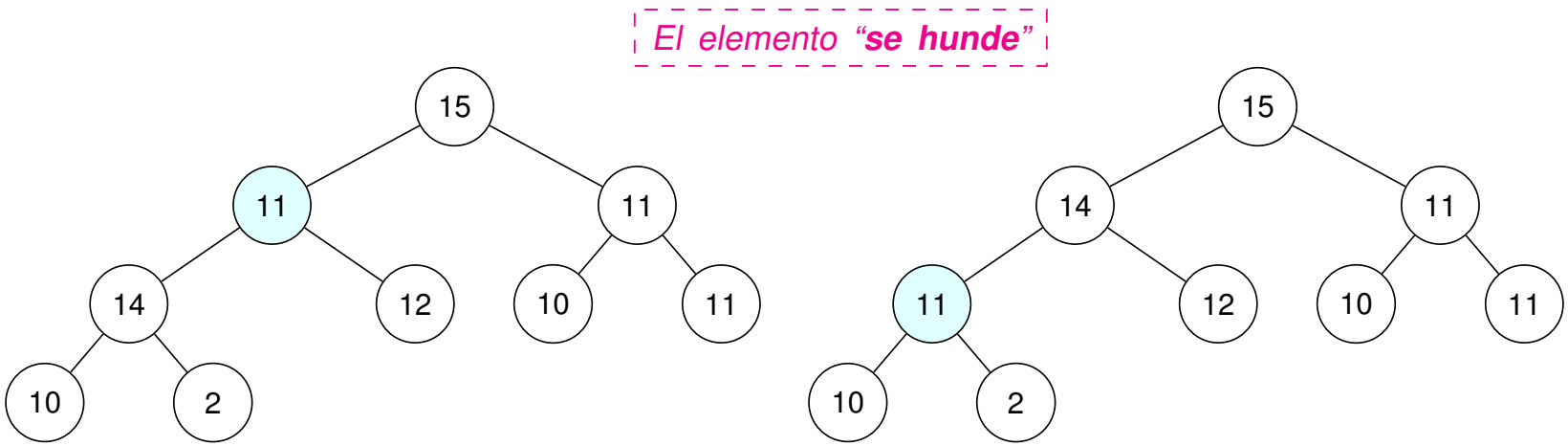
Paso 2: la raíz va bajando hasta alcanzar la posición adecuada

Montículos

Operación `eliminarMáx`

Eliminar el máximo elemento en un montículo:

- El número máximo de operaciones a realizar para suprimir el máximo elemento de un árbol de N nodos es del orden de $\log(N)$ (que es el número de niveles del árbol)
  - Si es posible acceder a la última hoja con coste $\mathcal{O}(1)$, y acceder desde cualquier elemento a sus hijos con coste $\mathcal{O}(1)$



Paso 2: la raíz va bajando hasta alcanzar la posición adecuada

Montículos

Implementación

- **Representación estática para montículos basada en un vector**
  - Al tratarse de un árbol binario casi-completo, esta representación es eficiente

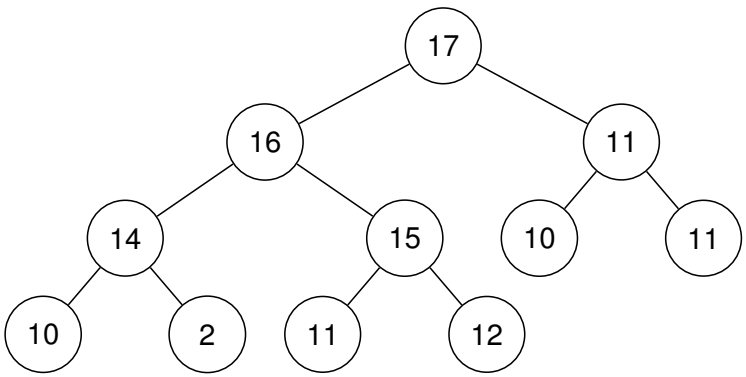
```
constante  maxNum = ...
tipo   montículo  = registro
                        dato: vector[1..maxNum] de elemento;
                        num: 0..maxNum {número de elementos en el montículo}
                    freg
```

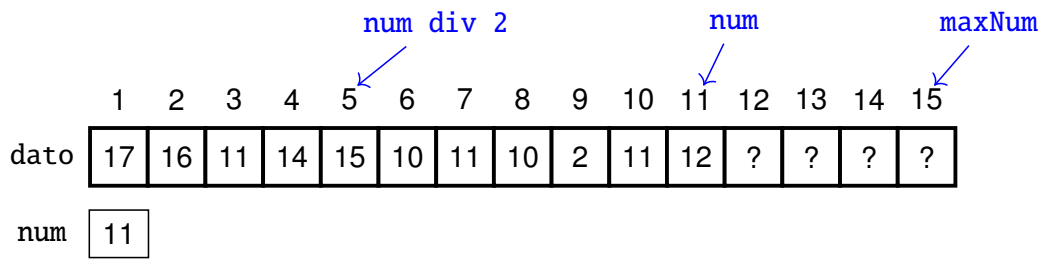
Características de la representación

- Montículo vacío si `num = 0`
- La raíz, si existe (`num > 0`), se almacena en `dato[1]`
- El hijo izquierdo de `dato[i]` estará en `dato[2*i]`, y el hijo derecho en `dato[2*i + 1]`, si existen (`num` $\leq$ `2*i`, o `num` $\leq$ `2*i + 1`)
- El padre del `dato[i]` estará en `dato[i div 2]`, si existe (`i`>1)
- La operación de `añadir` será implementada como parcial (vector lleno...)

Montículos
Implementación

- Los nodos del árbol se almacenan por niveles, de arriba a abajo y de izquierda a derecha en cada nivel





- **Propiedad**: si el montículo tiene `num` elementos, entonces los primeros `num div 2` son nodos internos en el montículo, y el resto son hojas

Montículos

Implementación

```
módulo colaConPrioridadesDeMáximos
importa defTipoElem
exporta
    constante maxNum {número máximo de elementos de la cola}
    tipo cp {representación del TAD cola con prioridades}
    procedimiento crear(sal c: cp)
    {devuelve una cola vacía, sin elementos}
    procedimiento añadir(e/s c: cp; ent e: elem; sal error: booleano)
    {si número de elementos(c) < maxNum entonces error = falso y añade e a c;
    si no error = verdad}
    función esVacía?(c: cp) devuelve booleano
    {devuelve verdad si y sólo si c es la cola vacía}
    función máx(c: cp) devuelve elem
    {Pre: esVacía?(c) = falso}
    {Post: devuelve un elemento máximo de c}
    procedimiento eliminarMáx(e/s c: cp)
    {Si c es vacía, sigue igual. Si no, elimina de c su elemento máximo actual}
implementación
    constante maxNum = ...
    tipo          cp  = registro
                            dato: vector[1..maxNum] de elemento;
                            num: 0..maxNum {número de elementos en el montículo}
                        freg
...
```

```
...
    procedimiento crear(sal c: cp)
    principio
        c.num := 0
    fin

    procedimiento añadir(e/s c: cp; ent e: elem; sal error: booleano)
    variables
        i: natural;
        debeSubir: booleano;
        aux: elem
    principio
        si c.num = maxNum entonces
            error := verdad
        sino
            error := falso;
            c.num := c.num + 1;
            c.dato[c.num] := e;
            i := c.num; {índice de la posición actual de e}
            { inicio de "flotar" }
            si i > 1 entonces
                debeSubir := c.dato[i] > c.dato[i div 2]
            sino {i = 1}
                debeSubir := falso
            fsi;
            mientrasQue debeSubir hacer
                {subir e en el árbol, intercambiándolo con su padre}
                aux := c.dato[i];
                c.dato[i] := c.dato[i div 2];
                c.dato[i div 2] := aux;
                i := i div 2; {índice de la pos actual de e}
                si i > 1 entonces
                    debeSubir := c.dato[i] > c.dato[i div 2]
                sino {i=1}
                    debeSubir := falso
                fsi
            fmq
            { fin de "flotar" }
        fsi
    fin
...
```

```
...
    procedimiento eliminarMáx(e/s c: cp)
    variables
        i, j: natural;
        aux: elem
    principio
        si c.num > 0 entonces
            c.dato[1] := c.dato[c.num];
            c.num := c.num - 1;
            i := 1;
            {i es el índice de la posición actual del que antes era el último elemento}
            { inicio de "hundir" }
            mientrasQue i ≤ c.num div 2 hacer
                {bajar el anterior último elem del árbol}
                si (2*i = c.num) orElse (c.dato[2*i] > c.dato[2*i + 1]) entonces
                    j := 2*i
                sino
                    j := 2*i + 1
                fsi; {j=hijo de i con mayor prioridad, o único hijo}
                si c.dato[i] < c.dato[j] entonces
                    {intercambia el anterior último elem y su hijo}
                    aux := c.dato[i];
                    c.dato[i] := c.dato[j];
                    c.dato[j] := aux;
                    i := j
                sino
                    i := c.num {para salir del bucle}
                fsi
            fmq
            { fin de "hundir" }
        fsi
    fin

    función máx(c: cp) devuelve elem
    {recordar... Pre: esVacía?(c) = falso}
    principio
        devuelve c.dato[1]
    fin

    función esVacía?(c: cp) devuelve booleano
    principio
        devuelve c.num = 0
    fin
fin {de colaConPrioridadesDeMáximos}
```

Montículos

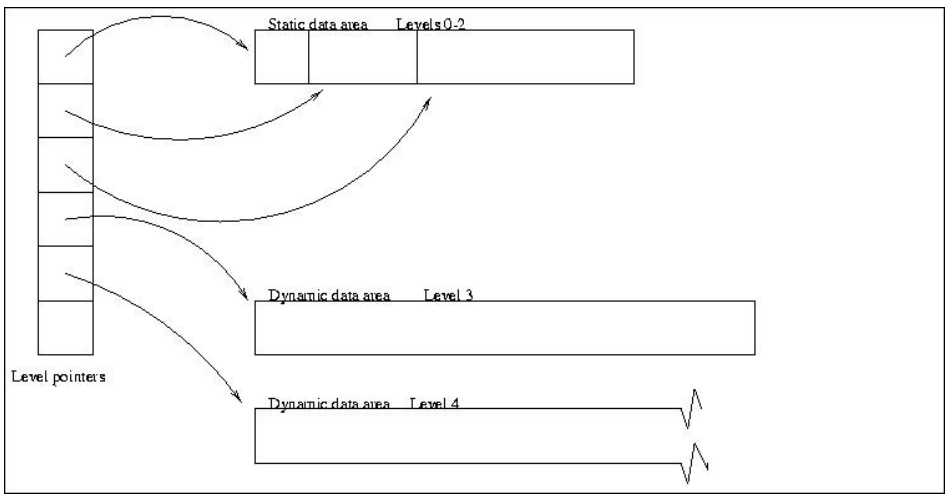
Problemas de la implementación estática

- Para poder usarla es necesario tener al menos una estimación del tamaño máximo que puede alcanzar el montículo (necesario para definir el vector)
- Coste en memoria muy elevado y no ajustado al tamaño real del montículo
- La operación `añadir` no podrá realizarse si el vector está lleno

Montículos

Problemas de la implementación estática

- Para poder usarla es necesario tener al menos una estimación del tamaño máximo que puede alcanzar el montículo (necesario para definir el vector)
- Coste en memoria muy elevado y no ajustado al tamaño real del montículo
- La operación `añadir` no podrá realizarse si el vector está lleno



- **Montículos dinámicos** o flexibles:
  - Consulta más detalles sobre los montículos dinámicos en este enlace
  - **OJO**: La solución propuesta en el dibujo sigue teniendo una capacidad máxima limitada y sus problemas

Índice

Universidad Zaragoza

Heapsort

- *¿Qué coste temporal en el caso peor tienen los algoritmos de ordenación de vectores que conocéis?*

Universidad Zaragoza

Heapsort

- *¿Qué coste temporal en el caso peor tienen los algoritmos de ordenación de vectores que conocéis?*
- *¿Qué coste temporal en el caso peor tendría este algoritmo?*

```
procedimiento ordena(e/s v: vector[1..maxNum] de elemento; ent n: 1..maxNum)
{Precondición: n>0.
Postcondición: Permuta los n primeros datos de v, dejándolos en orden creciente.}
variables
    t: titi;
    i: entero
principio
    crear(t); {un titi vacío}
    para i := 1 hasta n hacer {copia los datos de v en t}
        añadir(t, v[i])
    fpara;
    para i := n descendiendo hasta 1 hacer
        {extrae los datos ordenados de t, de mayor a menor}
        v[i] := max(t);
        eliminarMax(t)
    fpara
fin
```

¿Qué TAD es el titi?

Heapsort

- La disminución del número de operaciones necesarias para ordenar el vector se debe la duplicación en memoria utilizada (TAD `titi`)

Universidad Zaragoza

Heapsort

- La disminución del número de operaciones necesarias para ordenar el vector se debe la duplicación en memoria utilizada (TAD `titi`)
- Puede modificarse el algoritmo para trabajar sobre el mismo vector a ordenar
  - Procedimientos auxiliares: `intercambiar` y `empujar`

Universidad Zaragoza

Heapsort

```
procedimiento intercambia(e/s v: vector[1..maxNum] de elemento; ent i, j: 1..maxNum)
{Intercambia los valores de v[i] y v[j]}
variable
    aux: elem
principio
    aux := v[i];
    v[i] := v[j];
    v[j] := aux
fin

procedimiento empuja(e/s v: vector[1..maxNum] de elemento; ent pri, ult: 1..maxNum)
{Pre: v[pri],...,v[ult] satisfacen la propiedad de montículo excepto, quizá,
que los hijos de v[pri] pueden ser mayores que él}
variables
    r: entero {r indicará la posición 'actual' de v[pri]}
principio
    r := pri;
    mientrasQue r ≤ ult div 2 hacer
        si ult = 2*r entonces {r tiene un hijo en 2*r=ult}
            si v[r] < v[2*r] entonces
                intercambia(v, r, 2*r)
            fsi;
            r := ult {fuerza la terminación del bucle}
        sino {r tiene dos hijos, en 2*r y 2*r+1}
            si (v[r] < v[2*r]) and (v[2*r] ≥ v[2*r+1]) entonces {intercambia r con su hijo izq}
                intercambia(v, r, 2*r);
                r := 2*r
            sino
                si (v[r] < v[2*r + 1]) and (v[2*r + 1] > v[2*r]) entonces {intercambia r con su hijo dch}
                    intercambia(v, r, 2*r + 1);
                    r := 2*r + 1
                sino {r no viola la propiedad de montículo}
                    r := ult {para forzar la terminación del bucle}
                fsi
            fsi
        fsi
    fmq {Post: permuta los elementos v[pri]..v[últ] para que TODOS, }
fin { v[pri] incluido, verifiquen la propiedad del montículo }
```

Heapsort

```
procedimiento heapsort(e/s v: vector[1..maxNum] de elemento; ent n: 1..maxNum)
{Precondición: n>0.
Postcondición: Permuta los n primeros datos de v, dejándolos en orden creciente.}
variables
    i:entero
principio
    para i := n div 2 descendiendoHasta 1 hacer
        empuja(v, i, n)
    fpara; {ahora v es un montículo de máximos}
    para i := n descendiendoHasta 2 hacer
        {elimina el máximo de la raíz del montículo}
        intercambia(v, 1, i);
        empuja(v, 1, i - 1) {restablece el montículo hasta la posición i - 1}
    fpara
fin
```

Coste en tiempo $O(n \log n)$ en caso peor

Universidad Zaragoza

Trabajo encargado

Leer lección 18 de los apuntes.

Ejercicio 2.1 de la hoja 6

Estructuras de Datos y Algoritmos

Colas con prioridad, montículos y *heapsort*

Lección 18

Universidad Zaragoza

1542

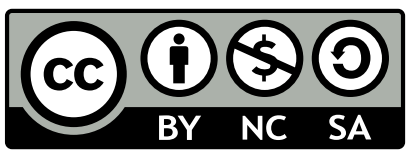
Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, España

Curso 2024/2025

Grado en Ingeniería Informática

Universidad de Zaragoza

Aula 0.04, Edificio Agustín de Betancourt



Adaptadas de diapositivas de Javier Campos