CI2613: Algoritmos y Estructuras III

Blai Bonet

Universidad Simón Bolívar, Caracas, Venezuela

Enero-Marzo 2015

Mergeable heaps

Heap (binario)

Un **heap** es una ED para almacenar elementos (i.e. repositorio) para los cuales existe un orden lineal total que los ordena

Un heap es un árbol binario **casi completo:** todos los niveles excepto posiblemente el último están completos

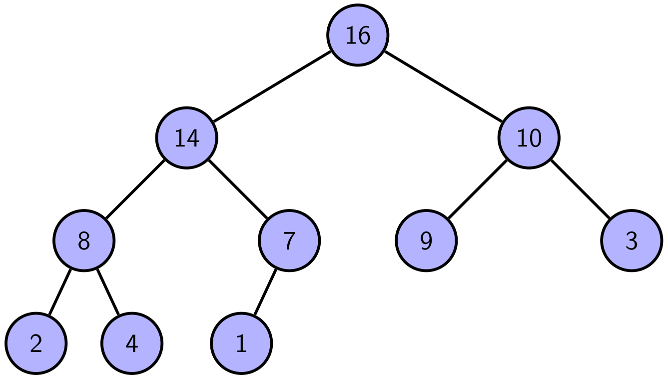
Los elementos se guardan en los nodos del árbol

El árbol satisface la **propiedad de heap:**

Si x es un nodo del árbol y y es un hijo de x, entonces $x \geq y$

En este caso, hablamos de un **max heap**. En un **min heap**, la propiedad de heap indica $x \leq y$

Heap: Ejemplo

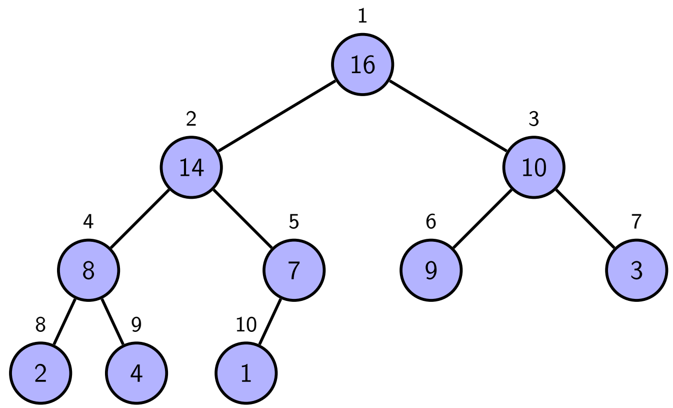


Representación con arreglo

Un heap con n elementos puede guardarse en un arreglo $A[1 \ldots n]$:

- La raíz del árbol es el elemento $A[1]$
- Para $1 \leq i \leq n$, el elemento $A[i]$ tiene hijo izquierdo ssi $2i \leq n$, y tiene hijo derecho ssi $2i + i \leq n$
- El hijo izquierdo de $A[i]$ es $A[2i]$ y el hijo derecho es $A[2i + 1]$
- El padre del elemento $A[i]$, para $i > 1$, es $A[\lfloor i/2 \rfloor]$

Heap: Ejemplo de representación



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Cola de prioridad

Un heap implementa una ED más general llamada **cola de prioridad**

Las **operaciones soportadas** por una cola de prioridad son:

- `make-priority-queue`$()$: crea una cola vacía
- `insert`(x, k): inserta el elemento x en la cola con prioridad k
- `maximum`$()$: retorna el elemento con la clave más grande en la cola
- `extract-max`$()$: remueve el elemento con la clave más grande
- `increase-key`(x, k): aumenta la clave del elemento x al valor k (k debe ser mayor o igual a la clave actual del elemento x)

En una **cola de prioridad mínima**, las operaciones `maximun()` y `extract-max()` se reemplazan por `minimum()` y `extract-min()`

Heap: Pseudocódigo

```
template<T> struct Heap {
    T A_[]                    % indexado A_[1..size_]
    int size_

    make-heap()

    void heapify(int i)       % procedimiento fundamental de heaps

    ...

    T maximum()
    T extract-max()
    void insert(T x, int k)
    void increase-key(int i, int k)
}
```

heapify(i)

`heapify`(i) **restablece la propiedad de heap** en el arreglo

Se asume que los subárboles $A[2i]$ y $A[2i+1]$ son heaps pero que el elemento $A[i]$ puede que no satisfaga la propiedad de heap; i.e. puede ser menor que el elemento $A[2i]$ ó $A[2i+1]$

```
void heapify(int i) {
    left = 2i
    right = 2i+1
    if left <= size_ && A_[left].key_ > A_[i].key_
        largest = left
    else
        largest = i
    if right <= size_ && A_[right].key_ > A_[largest].key_
        largest = right
    if largest != i
        intercambiar A_[i] con A_[largest]
        heapify(largest)
}
```

`heapify`(i) toma tiempo $O(\log n)$ en un heap con n elementos

Heap: Pseudocódigo

```
T maximum() {
    return A_[1]
}

T extract-max() {
    T item = A_[1]
    A_[1] = A[size_]
    size_ = size_ - 1
    heapify(1)
    return item
}

void insert(T x, int k) {
    size_ = size_ + 1
    A_[size_] = x
    A_[size_].key_ = −∞
    increase-key(size_, k)
}

void increase-key(int i, int k) {
    A_[i].key_ = k
    while (i > 1) && A_[⌊i/2⌋] < A_[i]
        intercambiar A_[i] con A_[⌊i/2⌋]
        i = ⌊i/2⌋
}
```


Complejidad de las operaciones

	Heap binario (peor caso)
`make-heap`	$\Theta(1)$
`maximum`	$\Theta(1)$
`insert`	$\Theta(\log n)$
`extract-max`	$\Theta(\log n)$
`increase-key`	$\Theta(\log n)$

Mergeable heap

Es una extensión de heaps que soporta las operaciones:

– `make-priority-queue`$()$: crea una cola vacía

– `insert`(x, k): inserta el elemento x en la cola con prioridad k

– `maximum`$()$: retorna el elemento con la clave más grande en la cola

– `extract-max`$()$: remueve el elemento con la clave más grande

– `increase-key`(x, k): aumenta la clave del elemento x al valor k (k debe ser mayor o igual a la clave actual del elemento x)

– `union`(H_1, H_2): crea y retorna un nuevo heap que contiene todos los elementos de H_1 y H_2. Los heaps H_1 y H_2 son "destruidos"

– `delete`(x): elimina el elemento x del heap

Implementación de mergeable heaps

Tres implementaciones de mergeable heaps:

– Heap binario

– Heap binomial

– Heap de Fibonacci (fuera del alcance de CI2613)

Comparación de heaps

	Heap binario (peor caso)	Heap binomial (peor caso)	Heap Fibonacci **(amortizado)**
`make-heap`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`maximum`	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$
`insert`	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
`extract-max`	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
`increase-key`	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
`union`	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
`delete`	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$

Heap binomial

ED similar al heap que implementa mergeable heap

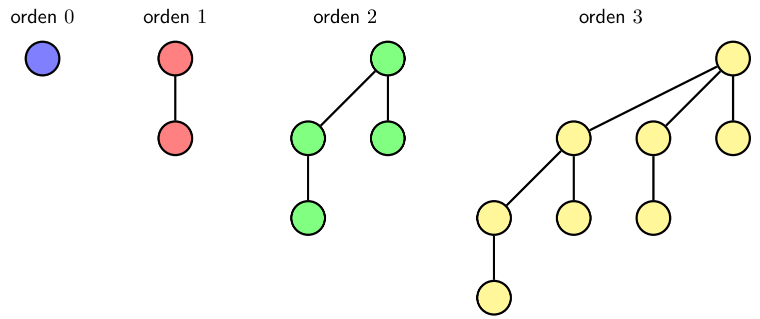
A diferencia de un heap, que es un único árbol binario, un heap binomial es una **colección de árboles binomiales**

Árbol binomial

Un árbol binomial se define **recursivamente:**

– un árbol binomial de orden 0 es un nodo aislado

– un árbol binomial de orden k tiene una raíz cuyos k hijos son árboles binomiales de orden $k-1, k-2, \ldots, 2, 1, 0$ respectivamente



Árbol binomial: Propiedades

Las siguientes propiedades son fáciles de probar (ejercicio):

1. Un árbol binomial de orden k tiene altura k
2. Un árbol binomial de orden k tiene $\binom{k}{d}$ nodos a profundidad d
3. Un árbol binomial de orden k tiene $\sum_{d=0}^{k} \binom{k}{d} = 2^k$ nodos

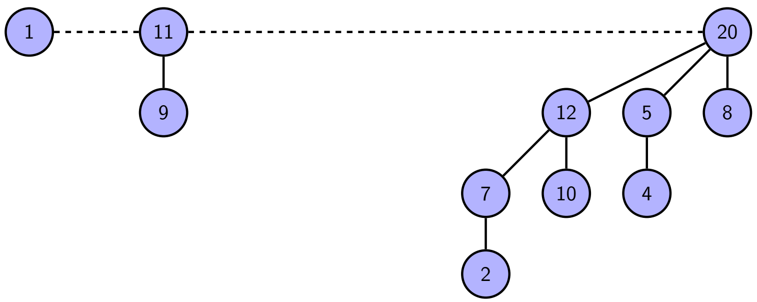
Heap binomial

– Cada árbol binomial en un heap binomial satisface la **propiedad de heap**

– Un heap binomial con n elementos tiene **a lo sumo** un árbol binomial de orden k para $k = 0, 1, \ldots, \lfloor \log n \rfloor$

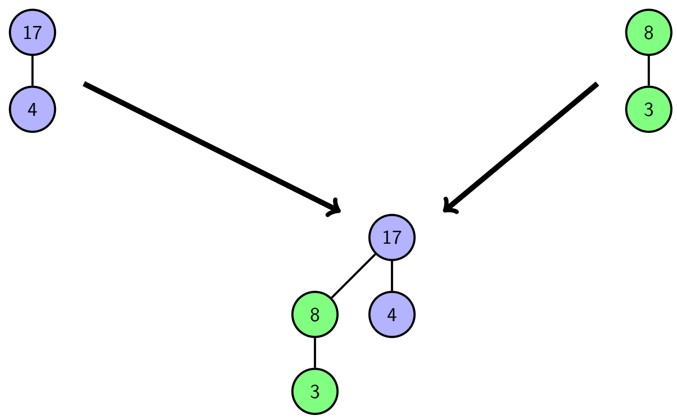
– Como un árbol binomial de orden k tiene 2^k nodos, un heap binomial de orden n contiene un árbol binomial de orden k si y sólo si el k-ésimo bit en la **expansión binaria de n es igual a 1**

Heap binomial: Ejemplo

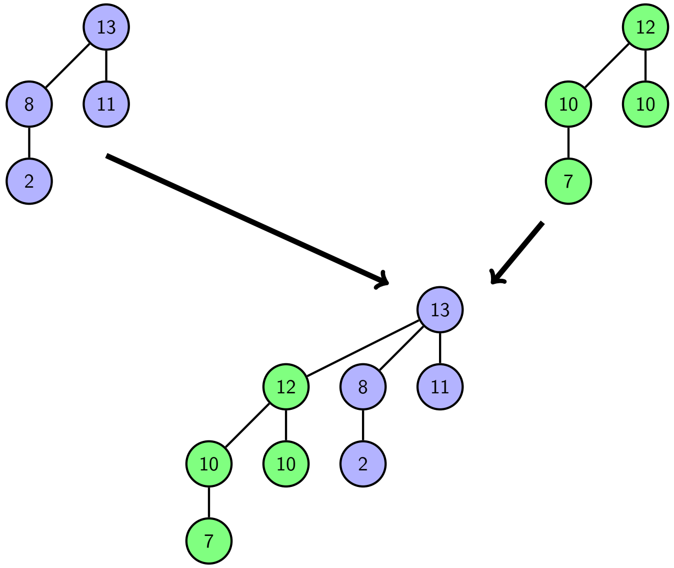


Heap binomial con $n = 11$ elementos: $11 = 01011_b$

Unión de árboles binomiales del mismo orden



Unión de árboles binomiales del mismo orden



Unión de árboles binomiales del mismo orden

La unión de dos árboles binomiales de orden k resulta en un árbol binomial de orden $k + 1$

Operación realizada en **tiempo constante**

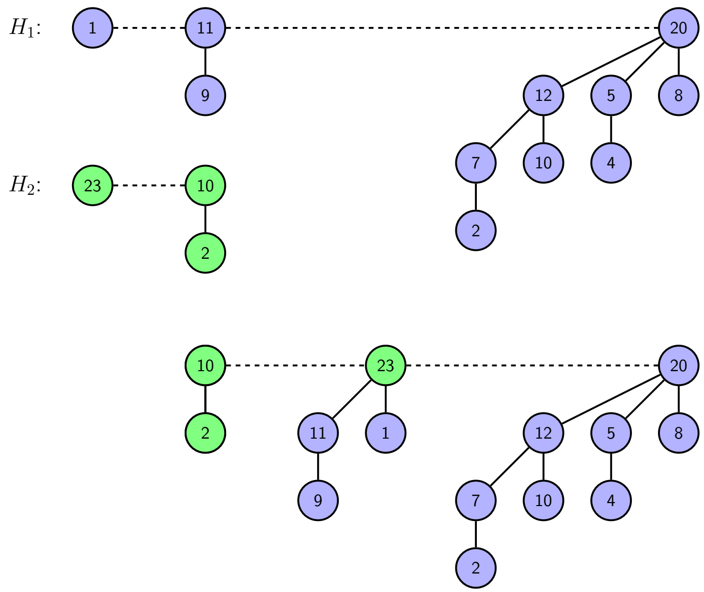
Unión de heap binomiales

– Se recorren los árboles binomiales en ambos heaps simultáneamente en orden de tamaño

– Si se encuentran dos árboles del mismo tamaño se hace la unión de los árboles, y se continua

Como un heap binomial con n elementos tiene a lo sumo $\log n$ árboles, la operación toma tiempo $O(\log n + \log m)$

Unión de heaps binomiales: Ejemplo



Heap binomial: Otras operaciones

- **Insertar:** para insertar x con clave k, creamos un heap binomial con sólo el elemento x y se une al heap. Tiempo $= O(\log n)$
- **Encontrar máximo:** se recorren las raíces de los árboles para encontrar el máximo. Tiempo $= O(\log n)$
- **Extraer máximo:** se busca el árbol con raíz máxima y se extrae del heap. Se crea un heap cuyos árboles son los hijos de la raíz del árbol encontrado. Se unen los heaps. Tiempo $= O(\log n)$
- **Incrementar clave:** parecido a un heap binario. Se incrementa la clave y luego se "flota hacia arriba" hasta la posición correcta. Tiempo $= O(\log n)$
- **Eliminar:** se incrementa la clave de x hasta ∞, y luego se extrae el máximo. Tiempo $= O(\log n)$

Comparación de heaps

	Heap binario (peor caso)	Heap binomial (peor caso)	Heap Fibonacci **(amortizado)**
`make-heap`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`maximum`	$\Theta(1)$	$\Theta(\log n)^*$	$\Theta(1)$
`insert`	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
`extract-max`	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
`increase-key`	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
`union`	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
`delete`	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$

* En un heap binomial, `maximum()` puede realizarse en tiempo $\Theta(1)$ si se mantiene un apuntador al mayor elemento del heap. El apuntador se debe actualizar usando tiempo $O(\log n)$ cada vez que el heap cambia