

Tema 4. Cues amb prioritat

Estructures de Dades i Algorismes

FIB

Antoni Lozano

Q2 2017–2018

Versió de 10 d'abril de 2018

1 Preliminars matemàtics

2 Cues amb prioritat

- Introducció
- *Heaps*
- Operacions bàsiques
- Implementació recursiva
- Implementació iterativa

3 *Heapsort*

- Algorisme bàsic
- Implementació sobre vector únic
- Construcció d'un *heap* en temps lineal

4 Altres aplicacions

- El problema de selecció

1 Preliminars matemàtics

2 Cues amb prioritat

- Introducció
- *Heaps*
- Operacions bàsiques
- Implementació recursiva
- Implementació iterativa

3 *Heapsort*

- Algorisme bàsic
- Implementació sobre vector únic
- Construcció d'un *heap* en temps lineal

4 Altres aplicacions

- El problema de selecció

Arbres binaris perfectes

Definició

El **nivell** d'un node en un arbre és la distància de l'arrel al node.

Definició

Un **arbre binari** és **perfecte** si tots els nodes interns tenen dos fills i totes les fulles són al mateix nivell.

Exemples



Definició

L' **alçària** d'un arbre és el nivell màxim dels nodes.

Arbres binaris perfectes

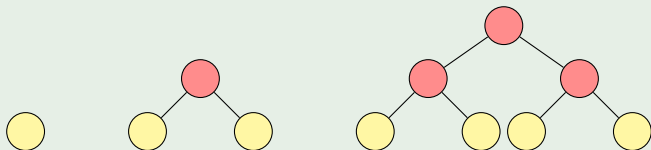
Definició

El **nivell** d'un node en un arbre és la distància de l'arrel al node.

Definició

Un **arbre binari** és **perfecte** si tots els nodes interns tenen dos fills i totes les fulles són al mateix nivell.

Exemples



Definició

L' **alçària** d'un arbre és el nivell màxim dels nodes.

Arbres binaris perfectes

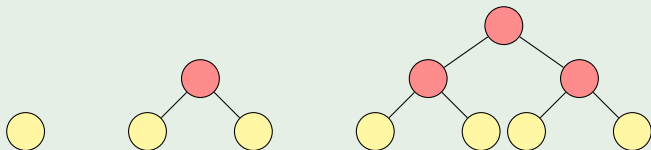
Definició

El **nivell** d'un node en un arbre és la distància de l'arrel al node.

Definició

Un **arbre binari** és **perfecte** si tots els nodes interns tenen dos fills i totes les fulles són al mateix nivell.

Exemples



Definició

L' **alçària** d'un arbre és el nivell màxim dels nodes.

Proposició

Un arbre binari perfecte d'alçària h té $2^{h+1} - 1$ nodes.

Demostració

Fem inducció en l'alçària. Sigui T un arbre binari perfecte d'alçària h .

- Base d'inducció: $h = 0$.

L'arbre té un sol node, però $1 = 2^{0+1} - 1$.

- Pas d'inducció: $h > 0$.

Els subarbres esquerre i dret tenen alçària $h - 1$ i, per hipòtesi d'inducció, cadascun té $2^h - 1$ nodes. El nombre de nodes de T és la suma d'aquests nodes més un (l'arrel):

$$\text{nodes de } T = 2(2^h - 1) + 1 = 2^{h+1} - 2 + 1 = 2^{h+1} - 1.$$

Arbres binaris perfectes

Proposició

Un arbre binari perfecte d'alçària h té $2^{h+1} - 1$ nodes.

Demostració

Fem inducció en l'alçària. Sigui T un arbre binari perfecte d'alçària h .

- **Base d'inducció:** $h = 0$.

L'arbre té un sol node, però $1 = 2^{0+1} - 1$.

- **Pas d'inducció:** $h > 0$.

Els subarbres esquerre i dret tenen alçària $h - 1$ i, per hipòtesi d'inducció, cadascun té $2^h - 1$ nodes. El nombre de nodes de T és la suma d'aquests nodes més un (l'arrel):

$$\text{nodes de } T = 2(2^h - 1) + 1 = 2^{h+1} - 2 + 1 = 2^{h+1} - 1.$$

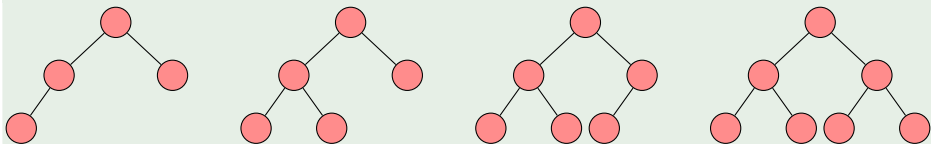
Arbres binaris complets

Definició

Un **arbre binari** d'alçada h és **complet** si

- 1 hi ha tots els nodes possibles amb nivells $0 \dots h - 1$
- 2 tots els nodes de nivell h són el màxim a l'esquerra

Exemple: arbres binaris complets d'alçada 2



Proposició

Un arbre binari complet d'alçària h té entre 2^h i $2^{h+1} - 1$ nodes.

Demostració

Sigui T un arbre binari complet d'alçària h :

- El **mínim** nombre de nodes de T es produeix quan té un sol node a nivell h . Com que fins a nivell $h - 1$, T té $2^h - 1$ nodes, sumant l'únic node a nivell h , s'obtenen 2^h nodes.
- El **màxim** nombre de nodes de T correspon a un arbre perfecte d'alçària h , que té $2^{h+1} - 1$ nodes.

Proposició

Un arbre binari complet d'alçària h té entre 2^h i $2^{h+1} - 1$ nodes.

Demostració

Sigui T un arbre binari complet d'alçària h :

- El **mínim** nombre de nodes de T es produeix quan té un sol node a nivell h . Com que fins a nivell $h - 1$, T té $2^h - 1$ nodes, sumant l'únic node a nivell h , s'obtenen 2^h nodes.
- El **màxim** nombre de nodes de T correspon a un arbre perfecte d'alçària h , que té $2^{h+1} - 1$ nodes.

Corol·lari

L'alçària d'un arbre binari complet de n nodes és $\lfloor \log n \rfloor \in \Theta(\log n)$.

Demostració

Per la proposició anterior, un arbre binari complet d'alçària h i n nodes compleix:

$$2^h \leq n \leq 2^{h+1} - 1.$$

Si prenem logaritmes en base 2, tenim

$$h \leq \log_2 n < h + 1.$$

I prenent la part baixa del logaritme,

$$h = \lfloor \log_2 n \rfloor.$$

Per tant, $h \in \Theta(\log n)$.

Corol·lari

L'alçària d'un arbre binari complet de n nodes és $\lfloor \log n \rfloor \in \Theta(\log n)$.

Demostració

Per la proposició anterior, un arbre binari complet d'alçària h i n nodes compleix:

$$2^h \leq n \leq 2^{h+1} - 1.$$

Si prenem logaritmes en base 2, tenim

$$h \leq \log_2 n < h + 1.$$

I prenent la part baixa del logaritme,

$$h = \lfloor \log_2 n \rfloor.$$

Per tant, $h \in \Theta(\log n)$.

1 Preliminars matemàtics

2 Cues amb prioritat

- Introducció
- *Heaps*
- Operacions bàsiques
- Implementació recursiva
- Implementació iterativa

3 *Heapsort*

- Algorisme bàsic
- Implementació sobre vector únic
- Construcció d'un *heap* en temps lineal

4 Altres aplicacions

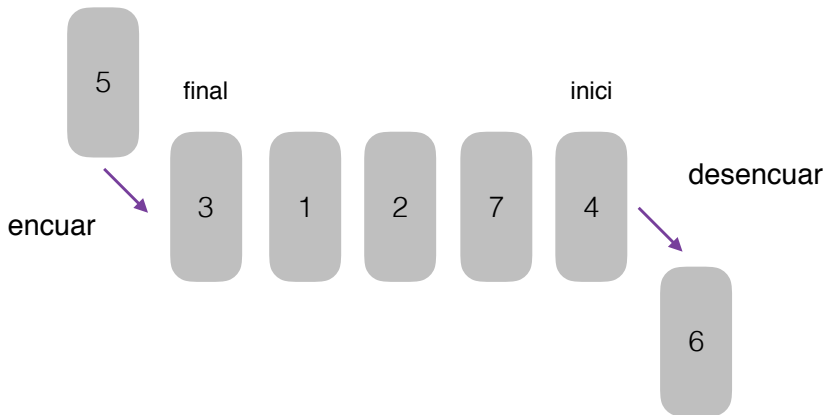
- El problema de selecció

Moltes aplicacions requereixen processar les entrades seguint un ordre parcial donat per certes prioritats.

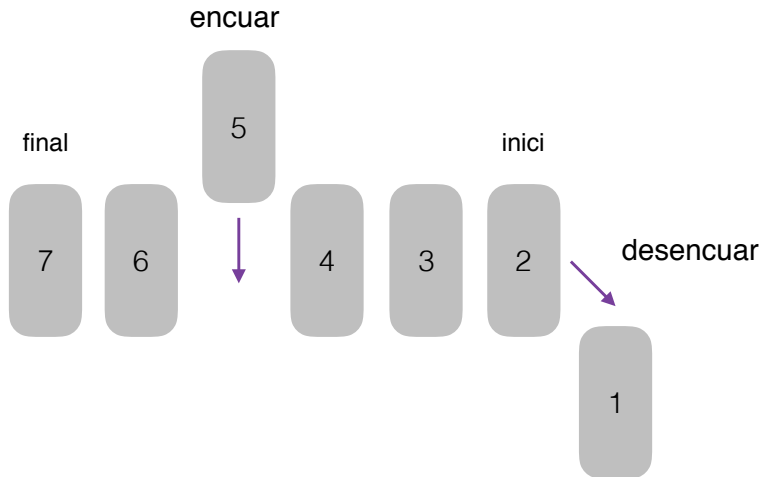
- **Programació de tasques** en sistemes d'ordinador: convé executar abans les més curtes
- **Sistemes de simulació** on convé simular els esdeveniments en ordre cronològic
- **Algorismes d'ordenació** que insereixen primer totes les entrades i després extreuen sempre la mínima de les que queden

L'estructura de dades adequada és la **cua amb prioritat**, una eina bàsica en el disseny d'algorismes.

Cua



Cua amb prioritat



Definició

Una **cua amb prioritat** és una estructura de dades que disposa de dues operacions bàsiques:

- **afegir**: inserir un element (clau més informació) i
- **treure_min** (**treure_max**): esborrar i retornar l'element amb la clau més petita (més gran)

Cues amb prioritat de l'STL

Descripció

- La implementació fa servir *heaps*
- Per defecte, *max-heaps* (clau més alta disponible amb cost $\Theta(1)$)
- Mètodes: `push`, `pop`, `top`, `empty`, `size`

Exemple: *max-heap*

```
#include <queue>
int main() {
    priority_queue<int> Q;
    Q.push(5);
    Q.push(3);
    cout << Q.top();
    Q.pop();
}
```

S'obté 5 al canal de sortida.

Cues amb prioritat de l'STL

Descripció

- La implementació fa servir *heaps*
- Per defecte, *max-heaps* (clau més alta disponible amb cost $\Theta(1)$)
- Mètodes: `push`, `pop`, `top`, `empty`, `size`

Exemple: *min-heap*

```
#include <queue>
int main() {
    priority_queue<int, vector<int>, greater<int> > Q;
    Q.push(5);
    Q.push(3);
    cout << Q.top();
    Q.pop();
}
```

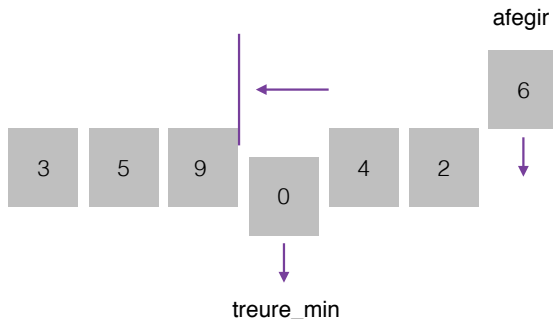
S'obté 3 al canal de sortida.

Implementacions senzilles

implementacions	afegir	treure_min
vector desordenat	$\Theta(1)$	$\Theta(n)$
vector ordenat	$\Theta(n)$	$\Theta(n)$
vector ordenat (decreixent)	$\Theta(n)$	$\Theta(1)$
vector circular ordenat	$\Theta(n)$	$\Theta(1)$
<i>heaps</i>	$\Theta(\log n)$	$\Theta(\log n)$

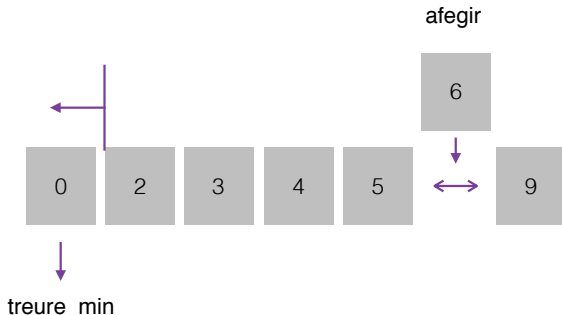
Implementacions senzilles

	implementacions	afegir	treure_min
▷	vector desordenat	$\Theta(1)$	$\Theta(n)$
	vector ordenat	$\Theta(n)$	$\Theta(n)$
	vector ordenat (decreixent)	$\Theta(n)$	$\Theta(1)$
	vector circular ordenat	$\Theta(n)$	$\Theta(1)$
	<i>heaps</i>	$\Theta(\log n)$	$\Theta(\log n)$



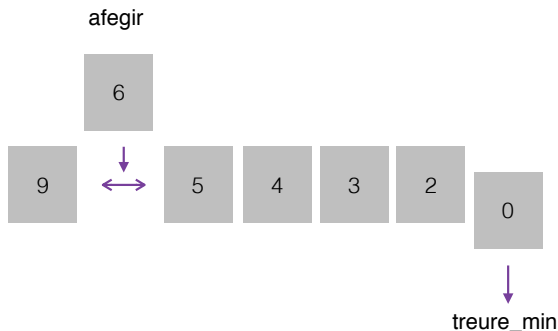
Implementacions senzilles

	implementacions	afegir	treure_min
	vector desordenat	$\Theta(1)$	$\Theta(n)$
▷	vector ordenat	$\Theta(n)$	$\Theta(n)$
	vector ordenat (decreixent)	$\Theta(n)$	$\Theta(1)$
	vector circular ordenat	$\Theta(n)$	$\Theta(1)$
	<i>heaps</i>	$\Theta(\log n)$	$\Theta(\log n)$



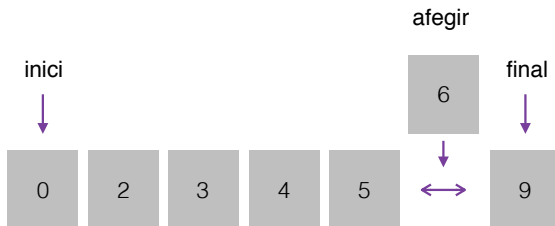
Implementacions senzilles

	implementacions	afegir	treure_min
	vector desordenat	$\Theta(1)$	$\Theta(n)$
	vector ordenat	$\Theta(n)$	$\Theta(n)$
▷	vector ordenat (decreixent)	$\Theta(n)$	$\Theta(1)$
	vector circular ordenat	$\Theta(n)$	$\Theta(1)$
	<i>heaps</i>	$\Theta(\log n)$	$\Theta(\log n)$



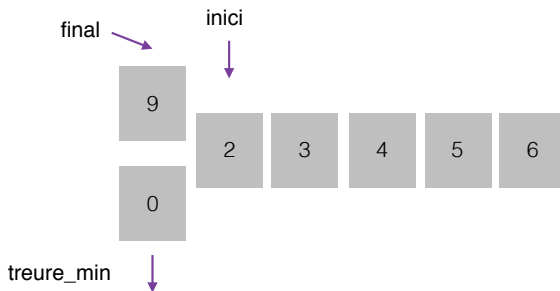
Implementacions senzilles

	implementacions	afegir	treure_min
	vector desordenat	$\Theta(1)$	$\Theta(n)$
	vector ordenat	$\Theta(n)$	$\Theta(n)$
	vector ordenat (decreixent)	$\Theta(n)$	$\Theta(1)$
▷	vector circular ordenat	$\Theta(n)$	$\Theta(1)$
	<i>heaps</i>	$\Theta(\log n)$	$\Theta(\log n)$



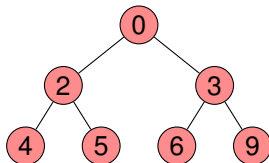
Implementacions senzilles

	implementacions	afegir	treure_min
	vector desordenat	$\Theta(1)$	$\Theta(n)$
	vector ordenat	$\Theta(n)$	$\Theta(n)$
	vector ordenat (decreixent)	$\Theta(n)$	$\Theta(1)$
▷	vector circular ordenat	$\Theta(n)$	$\Theta(1)$
	<i>heaps</i>	$\Theta(\log n)$	$\Theta(\log n)$



Implementacions senzilles

implementacions	afegir	treure_min
vector desordenat	$\Theta(1)$	$\Theta(n)$
vector ordenat	$\Theta(n)$	$\Theta(n)$
vector ordenat (decreixent)	$\Theta(n)$	$\Theta(1)$
vector circular ordenat	$\Theta(n)$	$\Theta(1)$
▷ <i>heaps</i>	$\Theta(\log n)$	$\Theta(\log n)$

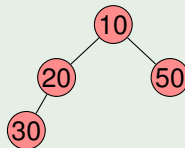
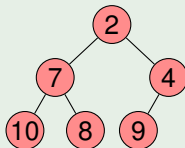


Definició

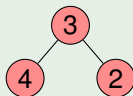
Un *min-heap* és un arbre binari complet on la clau d'un node és sempre més petita que les claus dels seus fills.

Exemples

Són *min-heaps*:



No són *min-heaps*:



Qüestió

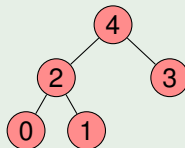
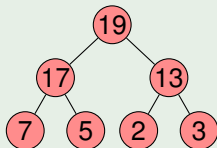
On seria la clau més gran en un *min-heap*? Per què?

Definició

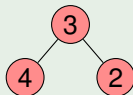
Un *max-heap* és un arbre binari complet on la clau d'un node és sempre més gran que les claus dels seus fills.

Exemples

Són *max-heaps*:



No són *max-heaps*:



Qüestió

Contenen necessàriament les fulles d'un *max-heap* les claus més petites de tot l'arbre? Per què o per què no?

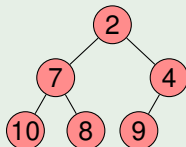
Terminologia

- Quan parlem de *heaps* sense especificar res més, aquí ens referirem als *min-heaps*
- En català, dels *heaps* se'n diu *munts* o *monticles*

Els *heaps* es representen de manera compacta mitjançant vectors.

Representació d'un *heap* mitjançant vectors

El *heap*



es representa amb el vector



No calen apuntadors perquè per a un node en posició i :

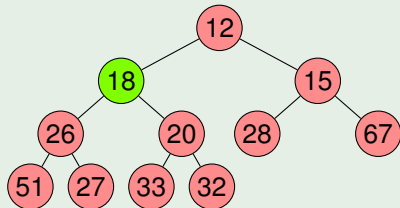
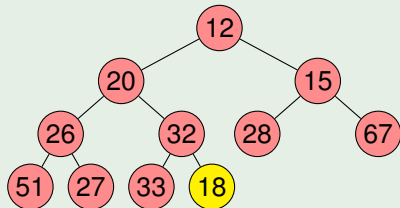
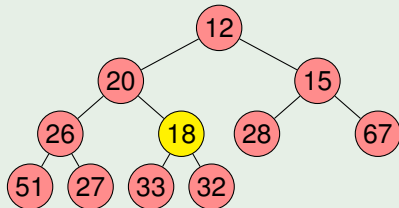
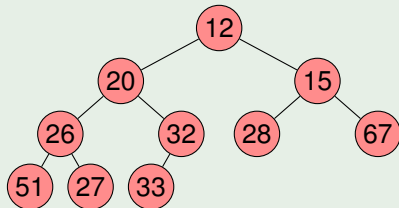
- el **pare** és a la posició $\lfloor i/2 \rfloor$
- el **fill esquerre** és a la posició $2i$
- el **fill dret** és a la posició $2i + 1$

Operacions bàsiques

Operació **afegir**

El més senzill és afegir l'element en la **següent posició lliure** del vector i **fer-lo ascendir** fins la posició en què es torna a complir la propietat del *heap*.

Exemple: afegir la clau 18

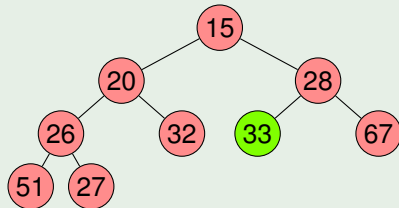
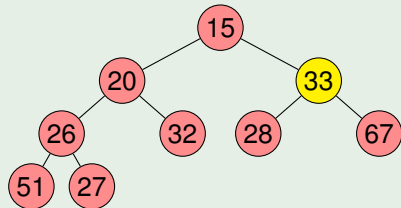
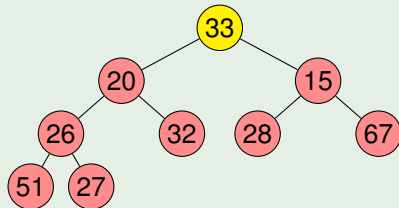
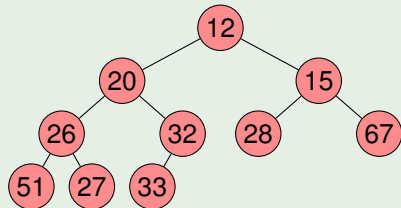


Operacions bàsiques

Operació treure-min

L'element en l'última posició del vector **es trasllada a la primera i es fa descendir** fins que troba la seva posició. Es retorna l'antiga arrel.

Exemple: esborrar el mínim



Costos de les operacions en *heaps*

operacions	cas pitjor	cas mitjà
afegir	$\Theta(\log n)$	$\Theta(1)$
treure_min	$\Theta(\log n)$	$\Theta(\log n)$

Anàlisi del cas pitjor

Donat un *heap* amb n nodes, el cost de `afegir` i `treure_min` és proporcional al nombre d'intercanvis, que està fitat per l'alçària: $\Theta(\log n)$.

Idea de l'anàlisi del cas mitjà (`afegir`)

Donat un *heap* amb n nodes i distribució uniforme de claus, una nova clau inserida en l'última posició té:

- probabilitat $1/2$ de ser més gran que el pare
- probabilitat $1/2$ de ser més gran que l'avi si és més gran que el pare (en total, $1/4$), i així successivament

Llavors, el nombre esperat d'intercanvis és

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots = 1$$

i, per tant, la inserció té un cost $\Theta(1)$ en mitjana.

Operacions bàsiques

Anàlisi del cas pitjor

Donat un *heap* amb n nodes, el cost de `afegir` i `treure_min` és proporcional al nombre d'intercanvis, que està fitat per l'alçària: $\Theta(\log n)$.

Idea de l'anàlisi del cas mitjà (`afegir`)

Donat un *heap* amb n nodes i distribució uniforme de claus, una nova clau inserida en l'última posició té:

- probabilitat $1/2$ de ser més gran que el pare
- probabilitat $1/2$ de ser més gran que l'avi si és més gran que el pare (en total, $1/4$), i així successivament

Llavors, el nombre esperat d'intercanvis és

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots = 1$$

i, per tant, la inserció té un cost $\Theta(1)$ en mitjana.

Anàlisi del cas pitjor

Donat un *heap* amb n nodes, el cost de `afegir` i `treure_min` és proporcional al nombre d'intercanvis, que està fitat per l'alçària: $\Theta(\log n)$.

Idea de l'anàlisi del cas mitjà (`afegir`)

Donat un *heap* amb n nodes i distribució uniforme de claus, una nova clau inserida en l'última posició té:

- probabilitat $1/2$ de ser més gran que el pare
- probabilitat $1/2$ de ser més gran que l'avi si és més gran que el pare (en total, $1/4$), i així successivament

Llavors, el nombre esperat d'intercanvis és

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots = 1$$

i, per tant, la inserció té un cost $\Theta(1)$ en mitjana.

Definició de la classe CuaPrio

El *heap* es forma en la taula `t`. La posició 0 no s'utilitza.

```
template <typename Elem>
class CuaPrio {

private:
    vector<Elem> t;
```


Implementació recursiva: funcions públiques

Constructura

Crea una cua amb prioritat buida. Cost: $\Theta(1)$.

```
CuaPrio () {  
    t.push_back( Elem() );  
}
```

Consultar la talla

Retorna la talla de la cua amb prioritat. Cost: $\Theta(1)$.

```
int talla () {  
    return t.size()-1;  
}
```

Implementació recursiva: funcions públiques

Consultar si és buida

Indica si la cua amb prioritat és buida. Cost: $\Theta(1)$.

```
bool buida () {  
    return t.talla() == 0;  
}
```

Retornar element mínim

Retorna un element amb prioritat mínima. Cost: $\Theta(1)$.

```
Elem minim () {  
    if (buida()) throw ErrorPrec("CuaPrio_buida");  
    return t[1];  
}
```

afegir

Afegeix un nou element. Cost: $\Theta(\log n)$.

```
void afegir (Elem& x) {  
    t.push_back(x);  
    surar(talla());  
}
```

treure_min

Treu i retorna l'element mínim. Cost: $\Theta(1)$.

```
Elem treure_min () {  
    if (buida()) throw ErrorPrec("CuaPrio_buida");  
    Elem x = t[1];  
    t[1] = t.back();  
    t.pop_back();  
    enfonsar(1);  
    return x;  
}
```

surar

Fer ascendir un element fins que ocupi una posició compatible amb la condició d'ordenació del *heap*. Cost: $\Theta(\log n)$.

```
void surar (int i) {  
    if (i != 1 and t[i/2] > t[i]) {  
        swap(t[i],t[i/2]);  
        surar(i/2);  
    }  
}
```

enfonsar

Fer descendir un element fins que ocupi una posició compatible amb la condició d'ordenació del *heap*. Cost: $\Theta(\log n)$.

```
void enfonsar (int i) {  
    int n = talla();  
    int c = 2*i;  
    if (c <= n) {  
        if (c+1 <= n and t[c+1] < t[c]) c++;  
        if (t[i] > t[c]) {  
            swap(t[i],t[c]);  
            enfonsar(c);  
        }  
    }  
}
```

Les operacions que canvien són **afegir** i **treure_min**, on les antigues **surar** i **enfonsar** estan optimitzades.

Els costos asimptòtics són els mateixos que en el cas recursiu: $\Theta(\log n)$.

afegir

```
void afegir (Elem& x) {  
    t.push_back(x);  
    int i = talla();  
    while (i != 1 and t[i/2] > x) {  
        t[i] = t[i/2];  
        i = i/2;  
    }  
    t[i] = x;  
}
```

treure_min

```
Elem treure_min () {  
    if (buida()) throw ErrorPrec("CuaDePrio_buida");  
    int n = talla();  
    Elem e = t[1], x = t[n];  
    t.pop_back(); --n;  
    int i = 1; c = 2*i;  
    while (c <= n) {  
        if (c+1 <= n and t[c+1] < t[c]) ++c;  
        if (x <= t[c]) break;  
        t[i] = t[c];  
        i = c;  
        c = 2*i;  
    }  
    t[i] = x;  
    return e;  
}
```


Tema 4. Cues amb prioritat

1 Preliminars matemàtics

2 Cues amb prioritat

- Introducció
- *Heaps*
- Operacions bàsiques
- Implementació recursiva
- Implementació iterativa

3 *Heapsort*

- Algorisme bàsic
- Implementació sobre vector únic
- Construcció d'un *heap* en temps lineal

4 Altres aplicacions

- El problema de selecció

Les cues amb prioritat es poden fer servir per ordenar en temps $\Theta(n \log n)$.

L'algorisme es diu *heapsort* i va ser presentat el 1964 per J.W.J. Williams. Donat un vector de n elements,

- 1 s'afegeixen els n elements a un *heap*: $\Theta(n \log n)$
- 2 es fan n operacions **treure_min** per construir un vector ordenat: $\Theta(n \log n)$

El temps total és $\Theta(n \log n)$, el mínim asimptòtic per a un algorisme d'ordenació.

Heapsort

Amb vectors separats per al *heap* i l'entrada/sortida.

Temps: $\Theta(n \log n)$.

Espai: $\approx 2n$.

```
template <typename elem>
void heapsort (vector<elem>& v) {
    n = v.size();
    CuaPrio<elem> h;
    for (int i = 0; i < n; ++i)
        h.afegir(v[i]);
    for (int i = 0; i < n; ++i)
        v[i] = h.treure_min();
}
```

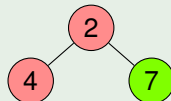
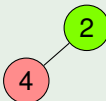
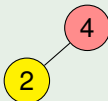
Exemple

Suposem que partim del vector:

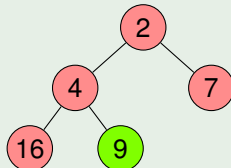
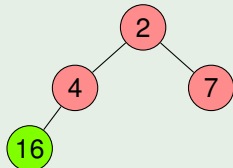
4	2	7	16	9	3	1	5
---	---	---	----	---	---	---	---

i afegim els elements a un *heap*, un per un.

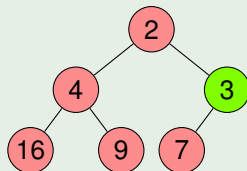
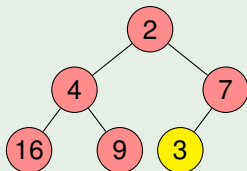
+4, +2, +7:



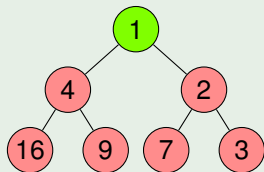
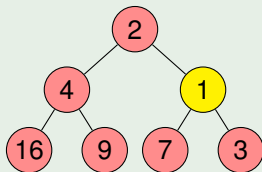
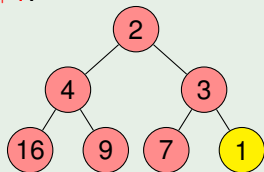
+16, +9:



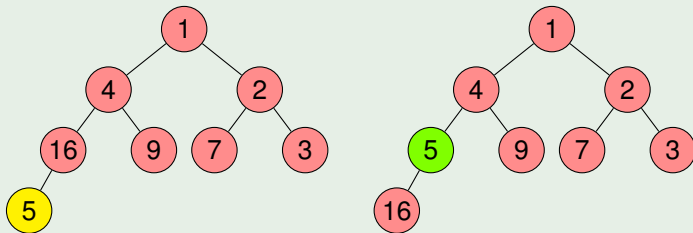
+3:



+1:



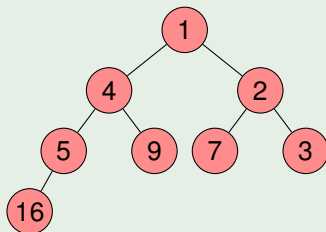
+5:



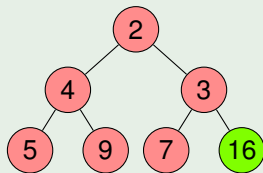
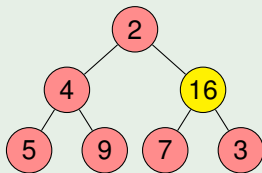
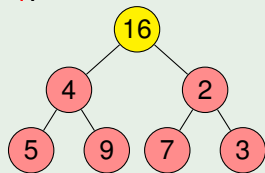
El *heap* resultant s'emmagatzema en el vector:

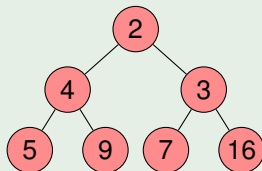
1	4	2	5	9	7	3	16
1	2	3	4	5	6	7	8

Ara traspassem els elements en ordre al vector original.

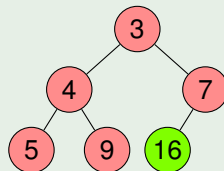
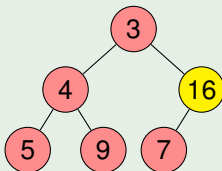
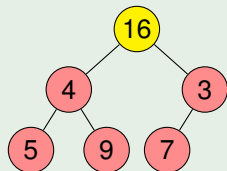


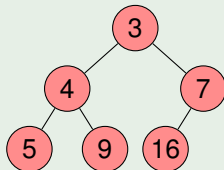
-1:



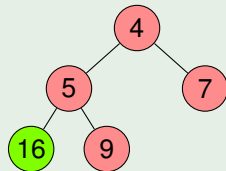
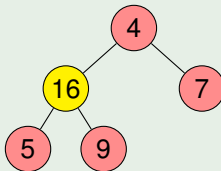
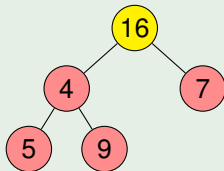


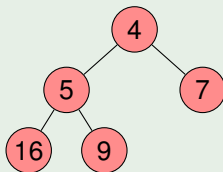
-2:



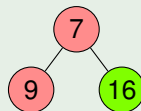
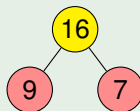
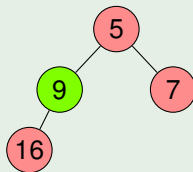
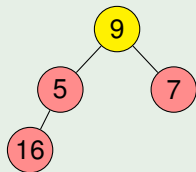


−3:

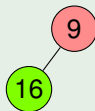
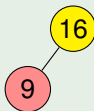




-4, -5:



-7, -9, -16:



Exemple: evolució dels vectors (operació **afegir**)

entrada/sortida

4	2	7	16	9	3	1	5
---	---	---	----	---	---	---	---

1 2 3 4 5 6 7 8

--	--	--	--	--	--	--	--

heap

Exemple: evolució dels vectors (operació **afegir**)

entrada/sortida

	2	7	16	9	3	1	5
--	---	---	----	---	---	---	---

1 2 3 4 5 6 7 8

4							
---	--	--	--	--	--	--	--

heap

Exemple: evolució dels vectors (operació **afegir**)

entrada/sortida

		7	16	9	3	1	5
--	--	---	----	---	---	---	---

1 2 3 4 5 6 7 8

2	4						
---	---	--	--	--	--	--	--

heap

Exemple: evolució dels vectors (operació **afegir**)

entrada/sortida

			16	9	3	1	5
--	--	--	----	---	---	---	---

1 2 3 4 5 6 7 8

2	4	7					
---	---	---	--	--	--	--	--

heap

Exemple: evolució dels vectors (operació **afegir**)

entrada/sortida

				9	3	1	5
--	--	--	--	---	---	---	---

1 2 3 4 5 6 7 8

2	4	7	16				
---	---	---	----	--	--	--	--

heap

Exemple: evolució dels vectors (operació **afegir**)

entrada/sortida

					3	1	5
--	--	--	--	--	---	---	---

1 2 3 4 5 6 7 8

2	4	7	16	9			
---	---	---	----	---	--	--	--

heap

Exemple: evolució dels vectors (operació **afegir**)

entrada/sortida

						1	5
--	--	--	--	--	--	---	---

1 2 3 4 5 6 7 8

2	4	3	16	9	7		
---	---	---	----	---	---	--	--

heap

Exemple: evolució dels vectors (operació **afegir**)

entrada/sortida

							5
--	--	--	--	--	--	--	---

1 2 3 4 5 6 7 8

1	4	2	16	9	7	3	
---	---	---	----	---	---	---	--

heap

Exemple: evolució dels vectors (operació **afegir**)

entrada/sortida

--	--	--	--	--	--	--	--

1 2 3 4 5 6 7 8

1	4	2	5	9	7	3	16
---	---	---	---	---	---	---	----

heap

Exemple: evolució dels vectors (operació **treure_min**)

entrada/**sortida**

--	--	--	--	--	--	--	--

1 2 3 4 5 6 7 8

1	4	2	5	9	7	3	16
---	---	---	---	---	---	---	----

heap

Exemple: evolució dels vectors (operació **treure_min**)

entrada/**sortida**

1							
---	--	--	--	--	--	--	--

1 2 3 4 5 6 7 8

2	4	3	5	9	7	16	
---	---	---	---	---	---	----	--

heap

Exemple: evolució dels vectors (operació **treure_min**)

entrada/**sortida**

1	2						
---	---	--	--	--	--	--	--

1 2 3 4 5 6 7 8

3	4	7	5	9	16		
---	---	---	---	---	----	--	--

heap

Exemple: evolució dels vectors (operació **treure_min**)

entrada/**sortida**

1	2	3					
---	---	---	--	--	--	--	--

1 2 3 4 5 6 7 8

4	5	7	16	9			
---	---	---	----	---	--	--	--

heap

Exemple: evolució dels vectors (operació **treure_min**)

entrada/**sortida**

1	2	3	4				
---	---	---	---	--	--	--	--

1 2 3 4 5 6 7 8

5	9	7	16				
---	---	---	----	--	--	--	--

heap

Exemple: evolució dels vectors (operació **treure_min**)

entrada/**sortida**

1	2	3	4	5			
---	---	---	---	---	--	--	--

1 2 3 4 5 6 7 8

7	9	16					
---	---	----	--	--	--	--	--

heap

Exemple: evolució dels vectors (operació **treure_min**)

entrada/**sortida**

1	2	3	4	5	7		
---	---	---	---	---	---	--	--

1 2 3 4 5 6 7 8

9	16						
---	----	--	--	--	--	--	--

heap

Exemple: evolució dels vectors (operació **treure_min**)

entrada/**sortida**

1	2	3	4	5	7	9	
---	---	---	---	---	---	---	--

1 2 3 4 5 6 7 8

16							
----	--	--	--	--	--	--	--

heap

Exemple: evolució dels vectors (operació **treure_min**)

entrada/**sortida**

1	2	3	4	5	7	9	16
---	---	---	---	---	---	---	----

1 2 3 4 5 6 7 8

--	--	--	--	--	--	--	--

heap

Idea general

Implementar l'algorisme sobre un únic vector fent una divisió en:

- una part esquerra per mantenir el *heap*
- una part dreta per a l'entrada/sortida

Cada cop que es fa una operació de **treure_min**, s'escriu el mínim com a primer element de la part dreta. Els elements queden ordenats de manera **descendent**.

Si es volen en ordre ascendent, es pot fer servir un **max-heap**.

Exemple: *heapsort* amb *max-heap* sobre vector únic

entrada/heap

4	2	7	16	9	3	1	5
---	---	---	----	---	---	---	---

Exemple: *heapsort* amb *max-heap* sobre vector únic

entrada/heap

4	2	7	16	9	3	1	5
---	---	---	----	---	---	---	---

Exemple: *heapsort* amb *max-heap* sobre vector únic

entrada/heap

4	2	7	16	9	3	1	5
---	---	---	----	---	---	---	---

Exemple: *heapsort* amb *max-heap* sobre vector únic

entrada/heap

7	2	4	16	9	3	1	5
---	---	---	----	---	---	---	---

Exemple: *heapsort* amb *max-heap* sobre vector únic

entrada/heap

16	7	4	2	9	3	1	5
----	---	---	---	---	---	---	---

Exemple: *heapsort* amb *max-heap* sobre vector únic

entrada/heap

16	9	4	2	7	3	1	5
----	---	---	---	---	---	---	---

Exemple: *heapsort* amb *max-heap* sobre vector únic

entrada/heap

16	9	3	2	7	4	1	5
----	---	---	---	---	---	---	---

Exemple: *heapsort* amb *max-heap* sobre vector únic

entrada/heap

16	9	3	2	7	4	1	5
----	---	---	---	---	---	---	---

Exemple: *heapsort* amb *max-heap* sobre vector únic

entrada/heap

16	9	3	5	7	4	1	2
----	---	---	---	---	---	---	---

Exemple: *heapsort* amb *max-heap* sobre vector únic

heap/sortida

9	7	3	5	2	4	1	16
---	---	---	---	---	---	---	----

Exemple: *heapsort* amb *max-heap* sobre vector únic

heap/sortida

7	5	3	1	2	4	9	16
---	---	---	---	---	---	---	----

Exemple: *heapsort* amb *max-heap* sobre vector únic

heap/sortida

5	4	3	1	2	7	9	16
---	---	---	---	---	---	---	----

Exemple: *heapsort* amb *max-heap* sobre vector únic

heap/sortida

4	2	3	1	5	7	9	16
---	---	---	---	---	---	---	----

Exemple: *heapsort* amb *max-heap* sobre vector únic

heap/sortida

3	2	1	4	5	7	9	16
---	---	---	---	---	---	---	----

Exemple: *heapsort* amb *max-heap* sobre vector únic

heap/sortida

2	1	3	4	5	7	9	16
---	---	---	---	---	---	---	----

Exemple: *heapsort* amb *max-heap* sobre vector únic

heap/sortida

1	2	3	4	5	7	9	16
---	---	---	---	---	---	---	----

Exemple: *heapsort* amb *max-heap* sobre vector únic

heap/sortida

1	2	3	4	5	7	9	16
---	---	---	---	---	---	---	----

Construcció d'un *heap* en temps lineal

De vegades un *heap* es construeix a partir d'una **col·lecció inicial d'ítems**.

- En *heapsort* es fa amb n insercions successives:

```
for (int i = 0; i < n; ++i)
    h.afegir(v[i]);
```

- El cost de cada *afegir* és:
 - $\Theta(1)$ en mitjana
 - $\Theta(\log n)$ en el cas pitjor
- Com que no hi ha altres operacions involucrades, és raonable esperar un cost
 - $\Theta(n)$ en mitjana
 - $\Theta(n \log n)$ en el cas pitjorper a les n insercions

Construcció d'un *heap* en temps lineal

De vegades un *heap* es construeix a partir d'una **col·lecció inicial d'ítems**.

- En *heapsort* es fa amb n insercions successives:

```
for (int i = 0; i < n; ++i)
    h.afegir(v[i]);
```

- El cost de cada *afegir* és:
 - $\Theta(1)$ en mitjana
 - $\Theta(\log n)$ en el cas pitjor
- Com que no hi ha altres operacions involucrades, és raonable esperar un cost
 - $\Theta(n)$ en mitjana
 - $\Theta(n \log n)$ en el cas pitjorper a les n insercions

Construcció d'un *heap* en temps lineal

De vegades un *heap* es construeix a partir d'una **col·lecció inicial d'ítems**.

- En *heapsort* es fa amb n insercions successives:

```
for (int i = 0; i < n; ++i)
    h.afegir(v[i]);
```

- El cost de cada *afegir* és:
 - $\Theta(1)$ en mitjana
 - $\Theta(\log n)$ en el cas pitjor
- Com que no hi ha altres operacions involucrades, és raonable esperar un cost
 - $\Theta(n)$ en mitjana
 - $\Theta(n \log n)$ en el cas pitjorper a les n insercions

Construcció d'un *heap* en temps lineal

Funció buildHeap

Construir el *heap* en temps $\Theta(n)$ seguint els passos següents:

- 1 Introduir els elements en el *heap* en qualsevol ordre (i temps lineal)
- 2 Si el *heap* té h nivells, per a $i = h - 1, h - 2, \dots, 1$:
 - **enfonsar** tots els nodes del nivell i

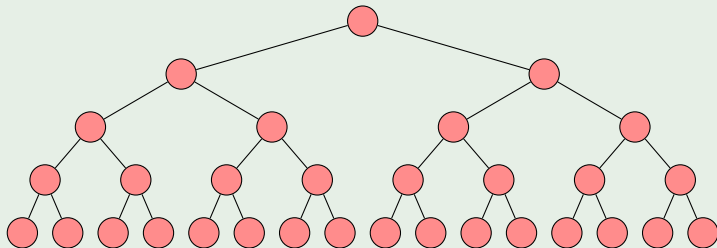
El fet que la majoria de *subheaps* tractats siguin petits fa que el nombre d'intercanvis fets per **enfonsar** sigui lineal.

Construcció d'un *heap* en temps lineal

Exemple

Per a un *heap* de 31 nodes, hi ha

- 8 *heaps* de mida 3
- 4 *heaps* de mida 7
- 2 *heaps* de mida 15
- 1 *heap* de mida 31



Construcció d'un *heap* en temps lineal

Constructor que pren els elements d'un vector com a entrada:

```
explicit PrioQueue (const vector<Elem>& v)
: t(v.size()+1) {
    for (int i = 0; i < v.size(); ++i)
        t[i+1] = v[i];
    buildHeap();
}
```

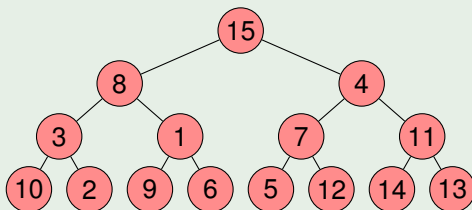
Establir propietat d'ordre del heap a partir d'una ordenació arbitrària d'ítems:

```
void buildHeap () {
    for (int i = size()/2; i > 0; --i)
        enfonsar(i);
}
```

Construcció d'un *heap* en temps lineal

Exemple de buildHeap

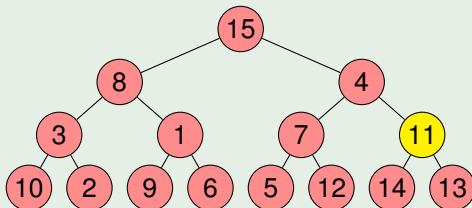
Heap inicial



Construcció d'un *heap* en temps lineal

Exemple de buildHeap

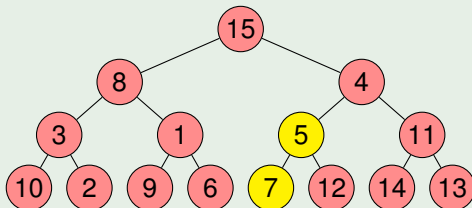
Després de enfonsar (7)



Construcció d'un *heap* en temps lineal

Exemple de buildHeap

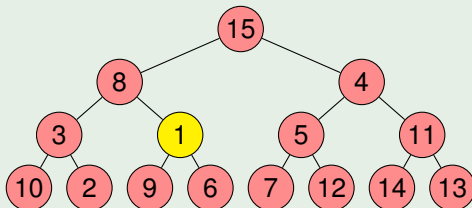
Després de enfonsar (6)



Construcció d'un *heap* en temps lineal

Exemple de buildHeap

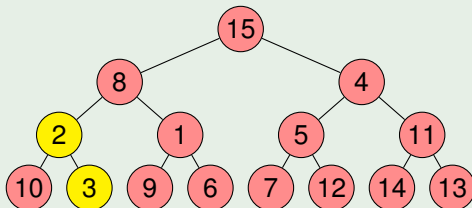
Després de enfonsar (5)



Construcció d'un *heap* en temps lineal

Exemple de buildHeap

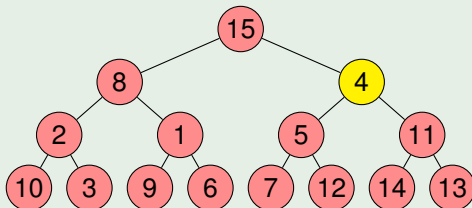
Després de enfonsar (4)



Construcció d'un *heap* en temps lineal

Exemple de buildHeap

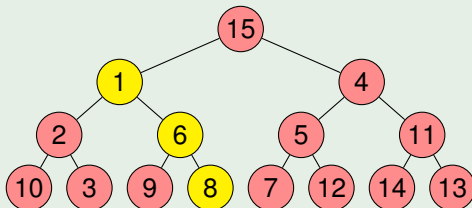
Després de enfonsar (3)



Construcció d'un *heap* en temps lineal

Exemple de buildHeap

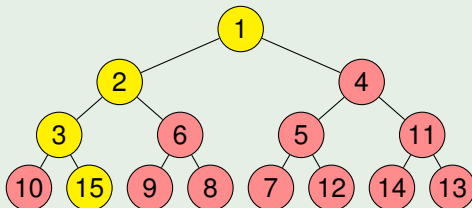
Després de enfonsar (2)



Construcció d'un *heap* en temps lineal

Exemple de buildHeap

Després de enfonsar (1)



Construcció d'un *heap* en temps lineal

El temps de càlcul de `buildHeap` està fitat per la suma de les alçàries de tots els nodes.

Volem demostrar que aquesta suma és $O(n)$.

Construcció d'un *heap* en temps lineal

Teorema

Per a l'arbre binari perfecte d'alçària h i $2^{h+1} - 1$ nodes, la suma de les alçàries dels seus nodes és $2^{h+1} - 1 - (h + 1)$.

Demostració

Com que hi ha 2^i nodes d'alçària $h - i$, la suma d'alçàries és:

$$\begin{aligned} S &= \sum_{i=0}^h 2^i (h - i) \\ &= h + 2(h - 1) + 4(h - 2) + 8(h - 3) + 16(h - 4) + \dots + 2^{h-1}(1) \end{aligned}$$

Multiplicant per 2, tenim

$$2S = 2h + 4(h - 1) + 8(h - 2) + 16(h - 3) + \dots + 2^h(1)$$

Construcció d'un *heap* en temps lineal

Teorema

Per a l'arbre binari perfecte d'alçària h i $2^{h+1} - 1$ nodes, la suma de les alçàries dels seus nodes és $2^{h+1} - 1 - (h + 1)$.

Demostració

Ara, a partir de

$$S = h + 2(h - 1) + 4(h - 2) + 8(h - 3) + 16(h - 4) + \dots + 2^{h-1}(1)$$

$$2S = 2h + 4(h - 1) + 8(h - 2) + 16(h - 3) + \dots + 2^h(1)$$

calculem $2S - S$ i obtenim

$$\begin{aligned} S &= -h + 2 + 4 + 8 + \dots + 2^{h-1} + 2^h \\ &= -h + (2^{h+1} - 1) - 1 \\ &= 2^{h+1} - 1 - (h + 1). \end{aligned}$$

Construcció d'un *heap* en temps lineal

Teorema

Per a l'arbre binari perfecte d'alçària h i $2^{h+1} - 1$ nodes, la suma de les alçàries dels seus nodes és $2^{h+1} - 1 - (h + 1)$.

Donat un arbre binari complet T de n nodes i alçària h , hem vist:

$$2^h \leq n \leq 2^{h+1} - 1.$$

És clar que la suma d'alçàries de T és com a màxim la de l'arbre binari complet d'alçària h , que és:

$$\begin{aligned} 2^{h+1} - 1 - (h + 1) &< 2^{h+1} \\ &\leq 2n \in O(n). \end{aligned}$$

Corol·lari

La suma d'alçàries d'un arbre binari complet de n nodes és $O(n)$.

Construcció d'un *heap* en temps lineal

Teorema

Per a l'arbre binari perfecte d'alçària h i $2^{h+1} - 1$ nodes, la suma de les alçàries dels seus nodes és $2^{h+1} - 1 - (h + 1)$.

Donat un arbre binari complet T de n nodes i alçària h , hem vist:

$$2^h \leq n \leq 2^{h+1} - 1.$$

És clar que la suma d'alçàries de T és com a màxim la de l'arbre binari complet d'alçària h , que és:

$$\begin{aligned} 2^{h+1} - 1 - (h + 1) &< 2^{h+1} \\ &\leq 2n \in O(n). \end{aligned}$$

Corol·lari

La suma d'alçàries d'un arbre binari complet de n nodes és $O(n)$.

Construcció d'un *heap* en temps lineal

Teorema

Per a l'arbre binari perfecte d'alçària h i $2^{h+1} - 1$ nodes, la suma de les alçàries dels seus nodes és $2^{h+1} - 1 - (h + 1)$.

Donat un arbre binari complet T de n nodes i alçària h , hem vist:

$$2^h \leq n \leq 2^{h+1} - 1.$$

És clar que la suma d'alçàries de T és com a màxim la de l'arbre binari complet d'alçària h , que és:

$$\begin{aligned} 2^{h+1} - 1 - (h + 1) &< 2^{h+1} \\ &\leq 2n \in O(n). \end{aligned}$$

Corol·lari

La suma d'alçàries d'un arbre binari complet de n nodes és $O(n)$.

1 Preliminars matemàtics

2 Cues amb prioritat

- Introducció
- *Heaps*
- Operacions bàsiques
- Implementació recursiva
- Implementació iterativa

3 *Heapsort*

- Algorisme bàsic
- Implementació sobre vector únic
- Construcció d'un *heap* en temps lineal

4 Altres aplicacions

- El problema de selecció

El problema de selecció

Problema de selecció

Donada una llista S de naturals i un $k \in \mathbb{N}$, determinar el k -èsim element més petit de S .

Fent servir *heaps*, podem trobar un nou algorisme:

- 1 Construir un *min-heap* a partir de S $\Theta(n)$
- 2 Efectuar k operacions **treure_min** del *min-heap* $\Theta(k \log n)$
- 3 Retornar l'últim element extret $\Theta(1)$

Cost total: $\Theta(n + k \log n)$.

La **mediana** correspon a $k = n/2$. Cost: $\Theta(n \log n)$.

En el cas $k = n / \log n$, el cost és $\Theta(n)$.

El problema de selecció

Problema de selecció

Donada una llista S de naturals i un $k \in \mathbb{N}$, determinar el k -èsim element més petit de S .

Fent servir *heaps*, podem trobar un nou algorisme:

- 1 Construir un *min-heap* a partir de S $\Theta(n)$
- 2 Efectuar k operacions **treure_min** del *min-heap* $\Theta(k \log n)$
- 3 Retornar l'últim element extret $\Theta(1)$

Cost total: $\Theta(n + k \log n)$.

La **mediana** correspon a $k = n/2$. Cost: $\Theta(n \log n)$.

En el cas $k = n / \log n$, el cost és $\Theta(n)$.