File de priorité 2 / tri par tas

Tas binaire : swim

swim : placement de v dans $H[1..i]$

Entrée: tas $H[1..]$, indice i, valeur v — placement de v dans $H[1..i]$

1 $p \leftarrow \lfloor i/2 \rfloor$

2 **while** $p \neq 0$ && $H[p] > v$ **do** $H[i] \leftarrow H[p]\,; i \leftarrow p\,; p \leftarrow \lfloor i/2 \rfloor$

3 $H[i] \leftarrow v$

```
private static <T extends Comparable<? super T>>
    void  swim(T x, int i, T[] H)
{
    int p = i/2; // arrondi automatique
    while (p>0)
    {
        T vp = H[p]; if(vp.compareTo(x)<=0) break;
        H[i]=vp; i=p; p=i>>>1; // décalage par x = div par 2^x
    }
    H[i]=x;
}
```

Tas binaire : sink

sink : placement de v dans $H[i..n]$

```
private static <T extends Comparable<? super T>>
    void  sink(T v, int i, T[] H, int n)
{
    assert (n<H.length); // capacité H.length, cases occupées 1..n
    int c=2*i; // indice de l'enfant gauche
    while (c<=n)
    {
        T vc = H[c];
        if (c<n)
        {
            T v2 = H[c+1]; if (v2.compareTo(vc)<0){ c++; vc = v2;}
        }
        if (vc.compareTo(x)>=0) break;
        H[i] = vc; i=c; c= i<<1; // décalage gauche = mul par 2
    }
    H[i] = x;
}
```

Tas binaire — efficacité

Nombre d'itérations et temps d'exécution :

procédure	**itérations** (au pire)	get (=H[...])	set (H[...]=)	compare
$\mathsf{swim}(\cdot, i, \cdot)$	$d(i) = \lfloor \lg i \rfloor$	d	$d+1$	d
$\mathsf{sink}(\cdot, i, \cdot, n)$	$h(i) = \lfloor \lg n/i \rfloor$	$2h$	$h+1$	$2h$

- ⋆ deleteMin/sink : $\sim 2 \lg n$ comparaisons au pire
- ⋆ insert/swim : $\sim \lg n$ comparaisons au pire
- ⋆ findMin : $O(1)$

Heapisation

Opération heapify (A) met les éléments de la vecteur $A[1..n]$ dans l'ordre de tas.

Triviale ?

$\mathsf{heapify-top}(A)$ // *tableau arbitraire* $A[1..n]$
H1 **for** $i \leftarrow 1, \ldots, n$ **do** $\mathsf{swim}(A[i], i, A)$ // // *insertion*

$\Rightarrow$ prend $O(n \log n)$

Meilleure solution :

$\mathsf{heapify}(A)$ // *tableau arbitraire* $A[1..n]$
H1 **for** $i \leftarrow \lfloor n/2 \rfloor, \ldots, 1$ **do** $\mathsf{sink}(A[i], i, A, n)$

$\Rightarrow$ prend $O(n)$

Heapify (cont)

Preuve du temps de calcul : si i est à la hauteur $h = \lfloor \lg n/i \rfloor$, alors $\mathsf{sink}(\cdot, i, \cdot)$ fait $O(h)$ temps. Il y a $\leq n/2^h$ nœuds à la hauteur h. Comparaisons au total :

$$\sum_h \frac{n}{2^h} O(h) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n).$$

□

«Évidemment», $O(n)$ est optimal pour construire le tas.

Preuve formelle :

- Trouver le minimum des élements dans une vecteur de taille n prend $n - 1$ comparaisons, donc un temps de $\Omega(n)$ est nécessaire pour trouver le minimum.
- Avec n'importe quelle implantation de heapify, on peut appeler findMin après pour retrouver le minimum en $O(1)$.
- Donc le temps de heapify doit être $\Omega(n)$, sinon on pourrait trouver le minimum en utilisant heapify+findMin en un temps $o(n) + O(1) = o(n)$. □

Tas d-aire

Tas d-aire : on utilise un arbre complet d-aire avec $d \geq 2$.

☞ indices $1..n$:
parent de l'indice i est $\lceil (i-1)/d \rceil$, enfants sont à $d(i-1)+2..di+1$

☞ indices $[0..n-1]$:
enfants à $di+1..d(i+1)$, parent à $\lfloor (i-1)/d \rfloor$

deleteMin/sink : $\sim d \log_d n = \frac{d}{\lg d} \lg n$
insert/swim : $\sim \log_d n = \frac{1}{\lg d} \lg n$
findMin : $O(1)$

Permet de balancer le coût de l'insertion et de la suppression si on a une bonne idée de leur fréquence ($d = 4$ est toujours meilleur que $d = 2$)

Files de priorité avancées

Bcp d'autres structures inventées (nécessaires pour un merge efficace) :
binomial heap, skew heap, Fibonacci heap

	liste triée	**liste non-triée**	**binaire**	***d*-aire**	**binomial**	**skew** (amorti)	**Fibonacci** (amorti)
deleteMin	$O(1)$	$O(n)$	$O(\log n)$	$O(d \log n / \log d)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
insert	$O(n)$	$O(1)$	$O(\log n)$	$O(\log n / \log d)$	$O(\log n)$	$O(1)$	$O(1)$
merge	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$	$O(1)$
decreaseKey	$O(n)$	$O(1)$	$O(\log n)$	$O(\log n / \log d)$	$O(\log n)$	$O(\log n)$	$O(1)$

Changer la priorité

changement de la priorité d'un élément — dans un tas binaire on peut le faire à l'aide de swim (si priorité baissée) ou de sink (si priorité élevée)

même technique pour supprimer un élément au milieu du tas : lancer swim ou sink à partir de la case vidée $0 \leq i < n$ et placer $H[n]$

graphe pondéré avec n sommets et m arêtes — algorithmes fondamentaux basés sur file de priorité contenant les sommets :

- ⋆ algorithme de Prim trouve l'arbre couvrant minimal
- ⋆ algorithme de Dijkstra trouve les plus courts chemins d'une source
  (priorité de t = longueur du chemin de s à t)

appellent n fois deleteMin et m fois decreaseKey

$\Rightarrow$ s'exécutent en $O(n \log n + m)$ temps si decreaseKey est $O(1)$ amorti

Note : decreaseKey(e) nécessite la recherche de l'indice de l'élément e dans le tas (dictionnaire élément $\rightarrow$ indice avec tas binaire)

Files de priorité : comparaison

tas d-aire est très simple à implémenter — d'autres sont plus compliqués

Table 1: Programming effort

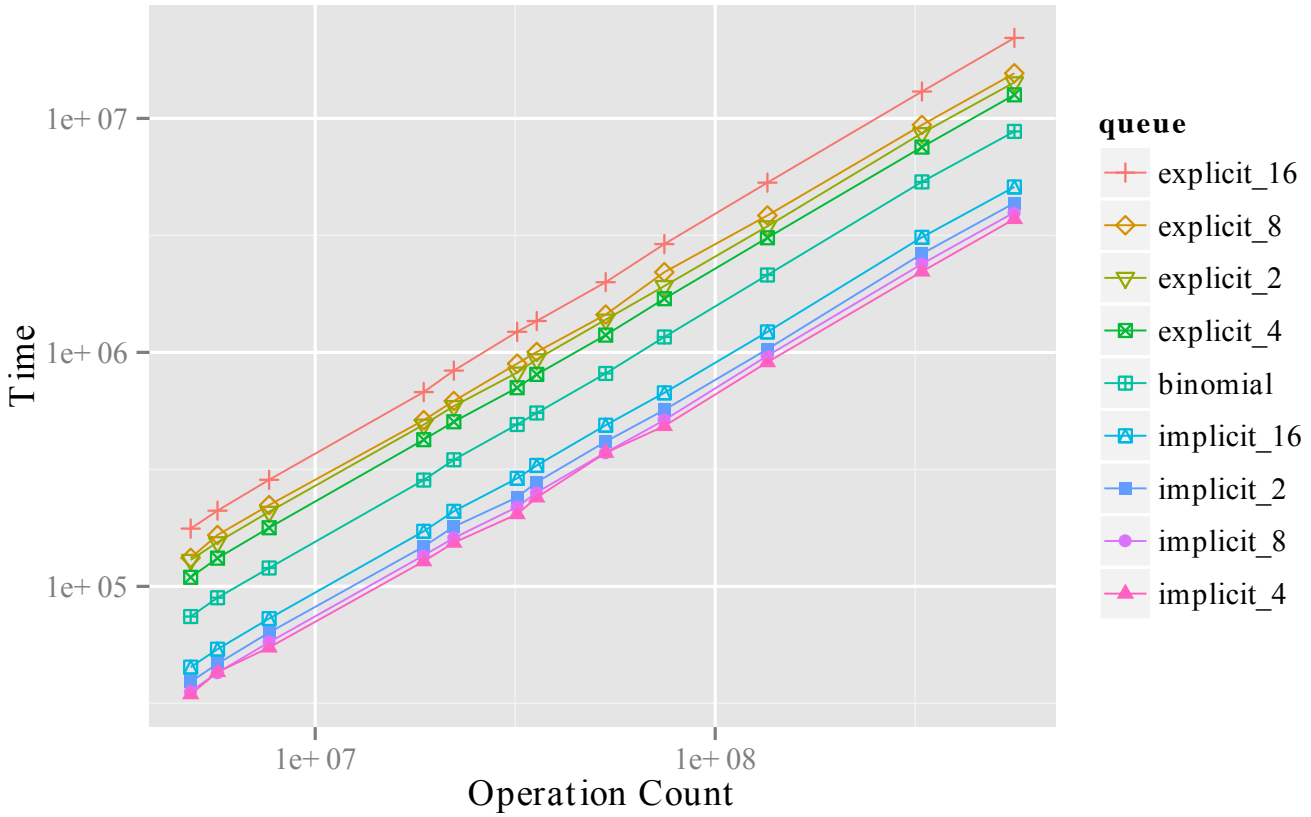
Heap variant	Logical lines of code (lloc)
implicit simple	184
pairing	186
implicit	194
Fibonacci	282
binomial	317
explicit	319
rank-pairing	376
quake	383
violation	481
rank-relaxed weak	638
strict Fibonacci	1009

(implicit, explicit = tas d-aire sans ou avec decreaseKey*)*

Larkin, Sen & Tarjan 2014, arXiv :1403.0252

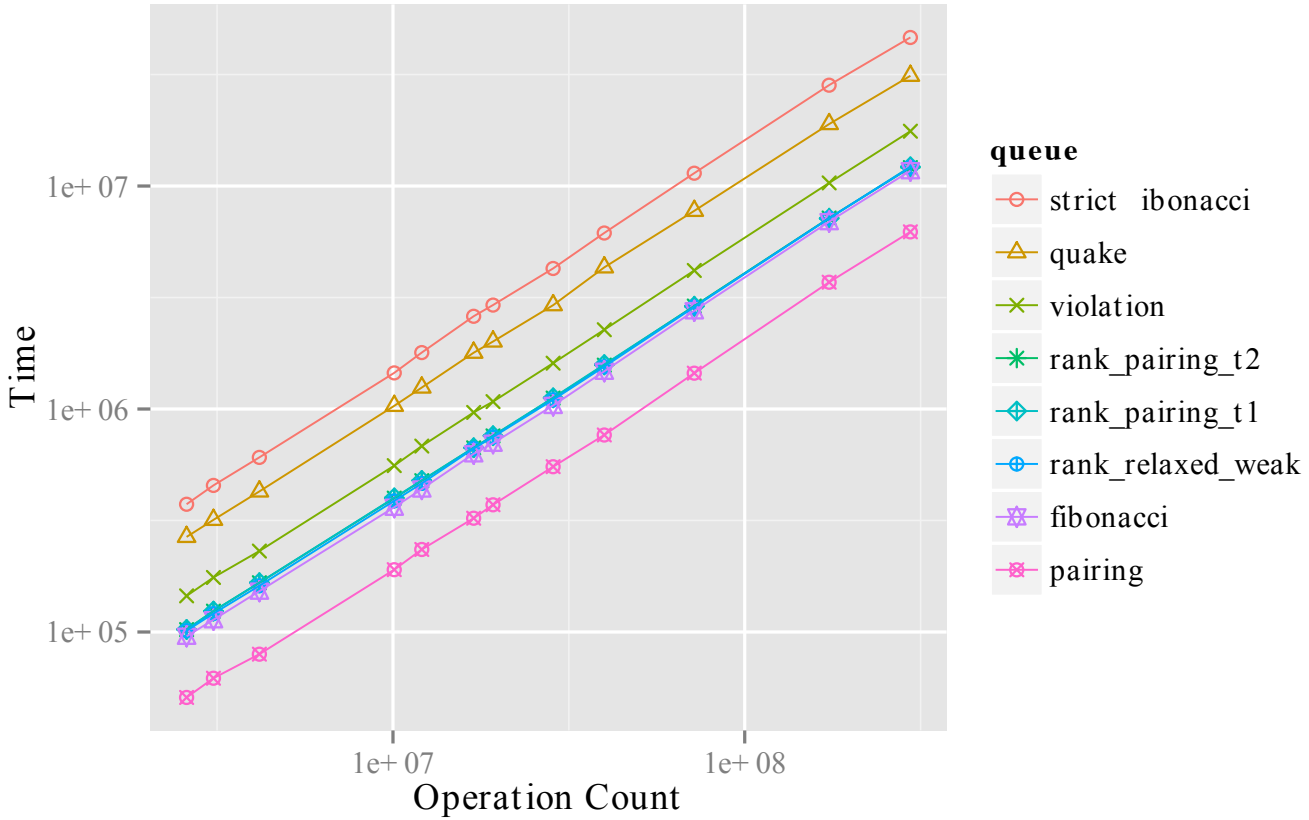
Étude empirique

sous-réseaux de la carte routière des USA



Larkin, Sen & Tarjan 2014

Étude empirique



Larkin, Sen & Tarjan 2014

Étude empirique

Table 3: Dijkstra – full USA road map

Heap Size – max = 4200, average = 2489

Ratio of Operations – Insert : DeleteMin : DecreaseKey = 13.98 : 13.98 : 1.00

queue	time	inst	l1_rd	l1_wr	l2_rd	l2_wr	br	l1_m	l2_m	br_m
implicit_4	**1.00**	**1.00**	**1.00**	1.10	1.35	1.06	**1.00**	**1.00**	1.00	**1.00**
implicit_8	1.07	1.12	1.12	1.03	1.61	1.18	1.01	1.07	1.20	1.01
implicit_2	1.17	1.10	1.01	1.27	1.35	**1.00**	1.33	1.05	**1.00**	1.33
implicit_16	1.37	1.42	1.38	**1.00**	2.20	1.35	1.21	1.24	1.63	1.21
pairing	1.68	1.09	1.12	2.95	1.71	28.57	1.39	1.60	1.75	1.39
binomial	2.37	1.49	1.83	3.49	1.30	34.57	1.49	2.24	1.56	1.49
fibonacci	3.15	2.00	2.09	5.03	1.73	79.53	2.91	2.85	2.67	2.91
rank_pairing_t2	3.26	1.98	2.16	2.85	1.34	35.46	3.19	2.29	1.61	3.19
rank_relaxed_weak	3.27	2.21	2.72	3.62	2.34	10.01	3.08	2.90	1.89	3.08
rank_pairing_t1	3.29	1.98	2.16	2.85	1.33	35.35	3.19	2.29	1.60	3.19
explicit_4	3.39	2.69	2.83	4.11	1.97	104.57	4.22	3.11	3.29	4.22
explicit_2	3.84	3.35	3.39	4.84	**1.00**	74.61	5.01	3.71	2.05	5.01
explicit_8	4.20	3.01	3.32	5.00	4.50	168.91	5.04	3.70	6.28	5.04
violation	4.74	2.85	2.67	3.92	2.60	4.24	4.38	2.95	1.97	4.38
explicit_16	5.94	3.94	4.56	6.81	8.02	276.59	7.13	5.06	10.76	7.13
quake	8.40	5.84	6.82	10.69	3.45	137.91	6.90	7.72	4.97	6.90
strict_fibonacci	12.49	9.47	12.50	22.07	6.96	84.51	11.47	14.83	6.58	11.47

time = wallclock, inst = compte d'instructions, l1/l2 : cache L1/L2, br : branching, rd : read, wr : write, m : miss

Larkin, Sen & Tarjan 2014

Tri par sélection

```
     Algo TRI-SELECTION(A[0..n − 1])
S1   for i ← 0, 1, ..., n − 2 do
S2       minidx ← i
S3       for j ← i + 1, ..., n − 1 do
S4           if A[j] < A[minidx] then minidx ← j
         // (maintenant A[minidx] = min{A[i], ..., A[n]})
S5       if i ≠ minidx then échanger A[i] ↔ A[minidx]
```

Complexité :

⋆ comparaison d'éléments $(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2}$ fois ;

⋆ échange d'éléments [ligne S5] $\leq (n-1)$ fois

Temps de calcul : toujours $\Theta(n^2)$ toujours

☞mais pas nécessairement une mauvaise idée si l'échange est beaucoup plus cher que la comparaison

et si on se sert d'une file de priorité ?

```
     Algo TRI-PQ(A[0..n − 1])
P1   for i ← 0, 1, . . . , n − 1 do PQ.insert(A[i])
P2   for i ← 0, 1, . . . , n − 1 do A[i] ← PQ.deleteMin()
```

- ☛ on peut utiliser heapify en P1 avec un tas binaire (d-aire)
- ☛ la boucle de P2 suggère la borne inférieure $\Omega(\log n)$ pour le temps amorti de deleteMin dans n'mporte quel implémentation ($\Omega(n \log n)$ temps minimal pour tri)

Tri par tas

```
     HeapSort(A) // vecteur non-trié A[1..n]
H1   heapify(A)
H2   for i ← |A|, ... 2 do
H3       échanger A[1] ↔ A[i]
H4       sink(A[1], 1, A, i − 1)
```

$A[1..n]$ est dans l'ordre décroissant à la fin

(pour l'ordre croissant, utiliser un max-tas)

Temps $O(n \log n)$ au pire, sans espace additionnelle !

quicksort : $\Theta(n^2)$ au pire (mais très rarement)
mergesort : $O(n \log n)$ toujours mais utilise un espace auxiliaire de taille n