Algorithmes de tri interne (4) [tr]
Méthodes par sélections

Karine Zampieri, Stéphane Rivière, Béatrice Amerein-Soltner

Unisciel algoprog Version 21 mai 2018

Table des matières

1 Tri par sélection **3**
1.1 Principe du tri par sélection . . . 3
1.2 Maximum en récursif . . . 3
1.3 Tri par sélection en récursif . . . 4
1.4 Tri par sélection en itératif . . . 4
1.5 Complexité du tri par sélection . . . 4

2 Tri par sélection quadratique **6**
2.1 Principe du tri par sélection quadratique . . . 6
2.2 Complexité du tri par sélection quadratique . . . 6

3 Tri par tas **7**
3.1 Définition d'un tas . . . 7
3.2 Principe du tri par tas . . . 9
3.3 Algorithme entasser . . . 9
3.4 Algorithme du tri par tas . . . 11
3.5 Algorithme construireTas . . . 12
3.6 Algorithme trMaximier . . . 13
3.7 Illustration du tri par tas . . . 14
3.8 Complexité du tri par tas . . . 15

Algorithmes de tri interne (4)

Méthodes par sélections

Les **méthodes par sélections** travaillent par *sélection* de l'élément maximum dans la partie non triée, l'échange avec l'élément frontière et itère le processus jusqu'à ce que la partie non trié soit vide. Il s'agit d'une récurrence sur les maxima successifs.

Méthodes par sélections

```
Action trParSelections ( DR A : Element [ NMAX ] ; n : Entier )
Début
  |  Si ( n > 1 )
  |    |  k <- indexMaximum ( A , n )
  |    |  permuterTab ( A , k , n )
  |    |  trParSelections ( A , n - 1 )
  |  FinSi
Fin
```

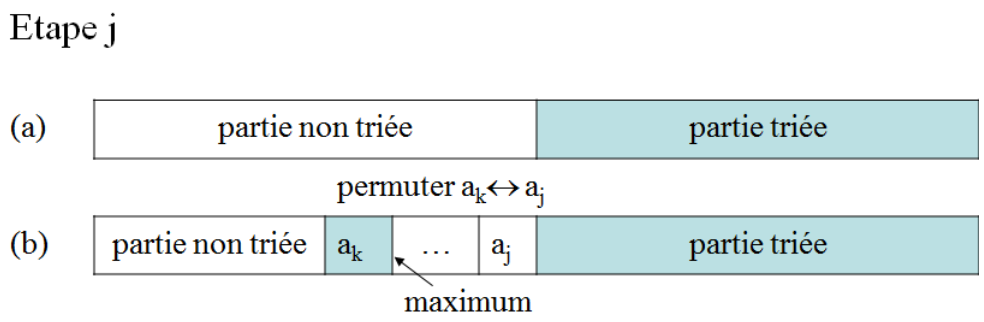
Le « tri par sélection » parcourt la partie non triée en cherchant l'élément maximum. Une version efficace, utilisant la notion d'arbre binaire, est connu sous le nom de « tri par tas ».

1 Tri par sélection

Nom anglais : *selection sort*
Propriétés : stable, sur place
Complexité : dans tous les cas en $\Theta(n^2)$
Visualisation : http://www.sorting-algorithms.com/selection-sort

1.1 Principe du tri par sélection

Le **tri par sélection** sélectionne l'élément le plus grand ainsi que son indice parmi les éléments non encore trié et l'échange avec celui en position `j`.



1.2 Maximum en récursif

Pour écrire de façon récursive la recherche de l'indice du maximum, on utilise la définition suivante :

- Si le tableau ne contient qu'un seul élément, le maximum est cet élément
- Si le tableau contient plusieurs éléments, le maximum est le plus grand entre :
  - le dernier élément de la séquence et
  - le maximum du tableau dont on a ôté le dernier élément

Fonction rechIMaxRec
(Maximum en récursif)

```
Fonction rechIMaxRec ( DR A : Element ( NMAX ] ; n : Entier ) : Entier
Variable imaxSaufDernier : Entier
Début
  |  Si ( n > 1 ) Alors
  |    |  imaxSaufDernier <- rechIMaxRec ( A , n - 1 )
  |    |  Si A [ n ] < A [ imaxSaufDernier ] Alors
  |    |    |  Retourner imaxSaufDernier
  |    |  Sinon
  |    |    |  Retourner n
  |    |  FinSi
  |  Sinon
  |    |  Retourner 1
  |  FinSi
Fin
```

1.3 Tri par sélection en récursif

Pour écrire de façon récursive le tri par sélection, nous partons de la définition suivante. Étant donné un tableau de n éléments :

- Si le tableau ne contient qu'un seul élément, le tableau est déjà trié
- Si le tableau contient plusieurs éléments, le tableau trié qui lui correspond :
  - a pour dernier élément le maximum du tableau non trié
  - pour éléments suivants ceux du tableau trié correspondant au tableau initial (non trié) dont on a ôté le maximum

Procédure trSelectionRec
(Tri par sélection en récursif)

```
Action trSelectionRec ( DR A : Element [ NMAX ] ; n : Entier )
Début
  |  Si ( n > 1 ) Alors
  |    |  k <- rechIMaxRec ( A , n )
  |    |  permuterTab ( A , k , n )
  |    |  trSelectionRec ( A , n - 1 )
  |  FinSi
Fin
```

1.4 Tri par sélection en itératif

Le tri par sélection en itératif transcrit la version récursive.

Procédure trSelection
(Tri par sélection en itératif)

```
Action trSelection ( DR A : Element ( NMAX ] ; n : Entier )
Début
  |  Pour j <- n à 2 Pas - 1 Faire
  |    |  k <- rechIMaxRec ( A , j )
  |    |  permuterTab ( t , k , j )
  |  FinPour
Fin
```

1.5 Complexité du tri par sélection

Nombre de comparaisons
Dans la procédure récursive, la récurrence sur le nombre de comparaisons est :

- $C(1) = 0$: lorsque le tableau n'a qu'un élément on ne fait aucune comparaison
- pour $n > 1$: $C(n) = n - 1 + C(n-1)$

Donc :

$$C(n) = \sum_{k=1}^{n-1} k = \frac{(n-1)n}{2} \in O(n^2)$$

Remarque

Le grand défaut de la méthode est que les comparaisons faites à chaque étape fournissent une information beaucoup plus riche que celle qui est effectivement utilisée pour mettre le `j`-ème élément à sa place.

Nombre d'échanges

Le nombre d'échanges dans le pire cas (= majorant du nombre d'échanges) est celui où le tableau est classé dans l'ordre inverse et donc chaque cellule doit être permutée :

$$E(n) = \sum_{j=1}^{n-1} 1 = (n-1) \in O(n)$$

2 Tri par sélection quadratique

2.1 Principe du tri par sélection quadratique

Le **tri par sélection quadratique** partage `A[1..j]` en `q` sous-segments sensiblement de même longueur, recherche l'élément maximum de chaque sous-segment puis le plus grand élément parmi eux.

2.2 Complexité du tri par sélection quadratique

Nombre de comparaisons

Chaque étape nécessite au pire environ $\frac{n}{q}$ comparaisons pour trouver le maximum d'un sous-segment et q comparaisons pour trouver le maximum parmi eux. Pour un tableau de n éléments :

$$C_{max}(n) = n(n/q + q)$$

Ce nombre est minimum pour $q = \sqrt{n}$, donc :

$$C_{max}(n) \in O(n\sqrt{n})$$

Nombre d'échanges

Le nombre d'échanges est au plus égal à celui des comparaisons :

$$E_{max}(n) \in O(n\sqrt{n})$$

Complexité spatiale

La place occupée en mémoire est plus importante (il faut $\sqrt{n}$ variables supplémentaires) et l'algorithme est (au niveau du code) beaucoup plus long que celui de la sélection ordinaire.

3 Tri par tas

Nom anglais : *heap sort*

Propriétés : tri interne, non stable, sur place

Visualisation : http://www.sorting-algorithms.com/heap-sort

Pour obtenir une amélioration sensible (sélection du maximum), il faut utiliser une structure de donnée plus élaborée permettant de conserver autant que possible l'information apportée par les tests successifs. L'idée est en fait de copier sur celle qui prévaut lors de l'organisation de championnats sportifs : l'arbre d'un tournoi.

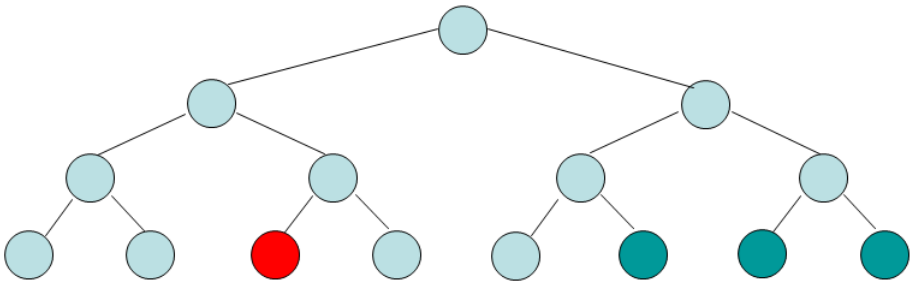
De nombreux algorithmes de tri ont été développés s'inspirant de ce principe et utilisant les arbres. Des perfectionnements successifs, dus en particulier à Williams et Floyd, ont conduit à celui connu sous le nom de « tri par tas » (dit aussi *tri-arbre*).

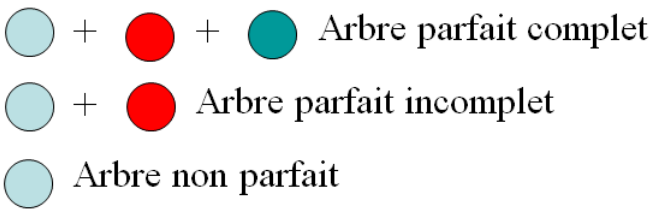
Il convient d'introduire quelques définitions préliminaires relatives aux arbres binaires.

3.1 Définition d'un tas

Arbre parfait

Arbre binaire dont tous les **niveaux** sont **complets** sauf le dernier qui est rempli de la gauche vers la droite.

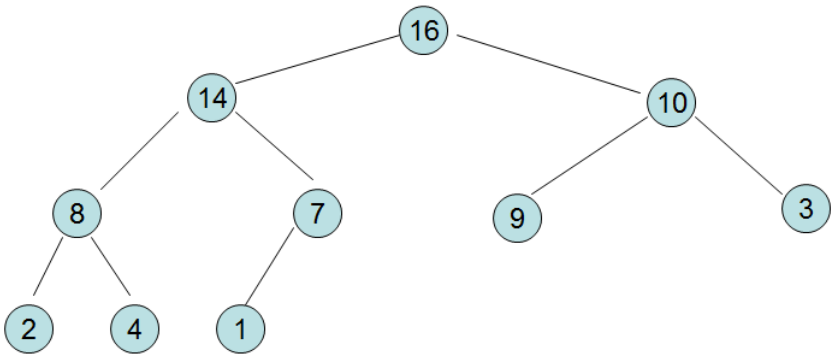




Tas

Arbre binaire parfait tel que l'information associée à chaque noeud à une clé supérieure ou égale à celle des deux fils du noeud, pour autant que ceux-ci existent (on appelle aussi ceci un *maximier* (arbre qui donne des maximums, comme un *pommier* donne des pommes), ou un *maxtas* pour le distinguer du *mintas*).

Exemple
La figure suivante représente un maxtas.



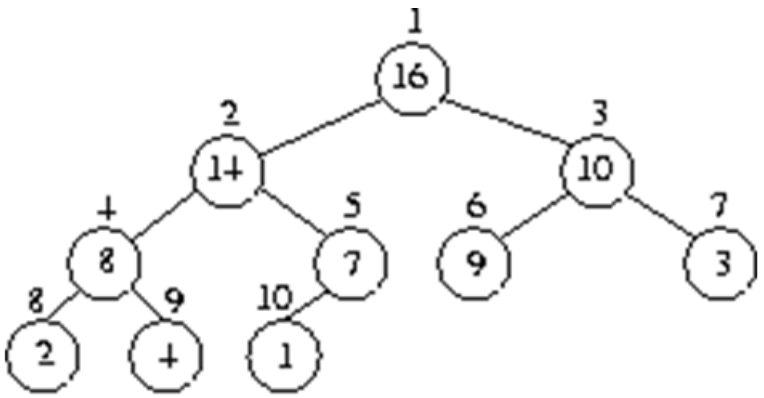
Propriété
Le racine d'un maxtas est un maximum (le plus grand élément de l'ensemble).

Preuve
La valeur de la racine est par construction supérieure à celle de ses fils et par transitivité de la relation d'ordre à celles de ses descendants : c'est donc un maximum. ■

Représentation tabulaire
Les tas sont généralement représentés et manipulés sous la forme d'un tableau :

- Un tableau `A` qui représente un tas est un objet à deux champs :
  1. `capacite(A)` qui est le nombre d'éléments qui peuvent être stockés dans le tableau `A`.
  2. `taille(A)` qui est le nombre d'éléments stockés dans le tableau `A`.
- La racine est stockée dans la première case du tableau `A[1]` (propriété du tas).
- Les éléments de l'arbre sont rangés dans l'ordre, niveau par niveau, et de gauche à droite. Les fonctions d'accès aux éléments du tableau sont alors :
  - `pere(k): retourner (divent(k,2))`
  - `filsgauche(k): retourner(2*k)`
  - `filsdroit(k): retourner(2*k+1)`
- Propriété des maxtas : `A[pere(k)] >= A[k]`

Exemple
La figure représente un maxtas vu comme un arbre binaire et comme un tableau. Le nombre à l'intérieur d'un noeud de l'arbre est la valeur contenue dans ce noeud. Le nombre au-dessus est l'indice correspondant dans le tableau.



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

3.2 Principe du tri par tas

Rappelons l'algorithme des méthodes par sélections :

Algorithme trTas

```
Action trParSelections ( DR A : Element [ NMAX ] ; n : Entier )
Début
  |  Si ( n > 1 )
  |     |  k <- indexMaximum ( A , n )
  |     |  permuterTab ( A , k , n )
  |     |  trParSelections ( A , n - 1 )
  |  FinSi
Fin
```

Le tri par tas utilise un maximier : la recherche `k <- indexMaximum(A,n)` est donc immédiate : c'est 1. Celle-ci est donc suivi de `permuterTab(A,n,k)`.

L'intérêt du maximier vient alors de la propriété suivante : les deux sous-arbres sont des maximiers mais la clé associée à la racine est quelconque. Pour **réorganiser** l'arbre de façon à en faire un maximier, il suffit de parcourir un chemin de la *racine*, de longueur maximale h (la hauteur de l'arbre) et non pas l'arbre tout entier, en faisant « descendre » les éléments `y` de clés « légères » vers le bas de l'arbre.

3.3 Algorithme entasser

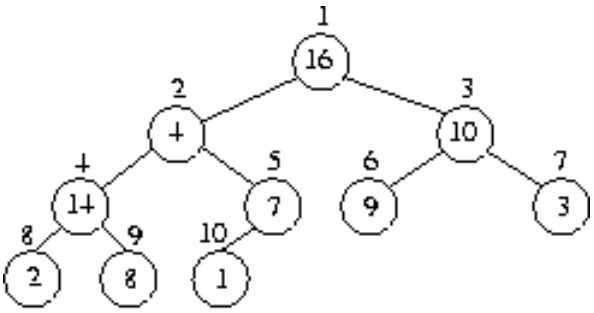
L'algorithme `entasser` prend en entrée un tableau `A` et deux indices `k` et `n`. Les sous-arbres de racines `filsgauche(k)` et `filsdroit(k)` sont des tas. L'algorithme fait « descendre » la valeur de `A[k]` de sorte que le sous-arbre de racine `k` soit un tas.

Procédure entasser

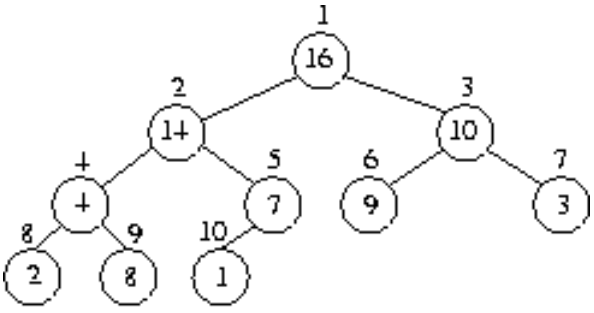
```
Action entasser ( DR A : Element [ NMAX ] ; k : Entier ; n : Entier )
Début
  |  fg <- filsgauche ( k )
  |  fd <- filsdroit ( k )
  |  imax <- k
  |  Si ( fg <= n ) Et A [ imax ] < A [ fg ] Alors
  |    |  imax <- fg
  |  FinSi
  |  Si ( fd <= n ) Et A [ imax ] < A [ fd ] Alors
  |    |  imax <- fd
  |  FinSi
  |  Si ( imax <> k ) Alors
  |    |  permuterTab ( A , k , imax )
  |    |  entasser ( A , imax , n )
  |  FinSi
Fin
```

Exemple

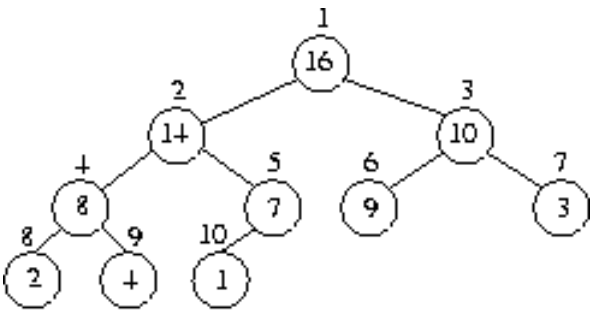
L'action de `entasser(A,2,n)` (avec `n=10`) est illustré par les figures suivantes. La configuration initiale viole la propriété du tas :



Pour `k=2` cette propriété est restaurée par interversion de la clé avec celle du fils gauche.



Le résultat n'est toujours pas un tas et l'appel récursif `entasser(A,4,n)` intervertit la clé du noeud `k=4` avec celle de son fils droit. On obtient finalement le maxtas suivant :



Correction

Le résultat de l'algorithme `entasser` est bien un tas car :

- La structure de l'arbre n'est pas modifiée.
- Un échange de valeurs entre un père et un fils n'a lieu que si la valeur du fils est supérieure à celle du père. Or la valeur du père était supérieure à celles stockées dans ses deux arbres fils exceptée la valeur ajoutée à l'arbre. La nouvelle clé de la racine est donc bien plus grande que l'intégralité de celles stockées dans l'arbre dont elle devient la racine.

Complexité

Le temps d'exécution de `entasser` sur un arbre de taille n est en $\Theta(1)$ plus le temps de l'exécution récursive de `entasser` sur un des deux sous-arbres. Or ces deux sous-arbres ont une taille en au plus $\frac{2n}{3}$ (le pire cas survient quand la dernière rangée de l'arbre est exactement remplie à moitié). Le temps d'exécution de `entasser` est donc décrit par la récurrence :

$$T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1)$$

ce qui, d'après le cas 2 du Master-Théorème, nous donne :

$$T(n) = \Theta(\log n)$$

car $a = 1$, $b = \frac{3}{2}$ et $\log_b a = 0$.

3.4 Algorithme du tri par tas

L'algorithme du tri par tas se décompose en deux parties qui utilisent chacune `entasser`. La première `construireTas` « plante l'arbre » en transformant le tableau initial en tas. La seconde `trMaximier` effectue le tri proprement dit en tirant parti de la structure du tas.

Algorithme

```
Action trTas ( DR A : Element [ NMAX ] ; n : Entier )
Début
  |  construireTas ( A , n )
  |  trMaximier ( A , n )
Fin
```

3.5 Algorithme construireTas

Pour écrire l'algorithme `construireTas`, on remarque que `entasser` s'applique à des indices `k` tels que les fils de `k`, s'ils existent, sont déjà des maximiers, et que tel est le cas des `k` compris entre $\lfloor n/2 \rfloor + 1$ et n, puisqu'ils n'ont pas de fils. Il est donc naturel de procéder par récurrence et « à reculons ».

Procédure construireTas

```
Action construireTas ( DR A : Element [ NMAX ] ; n : Entier )
Début
  |  Pour k <- DivEnt ( n , 2 ) à 1 Pas - 1 Faire
  |    |  entasser ( A , k , n )
  |  FinPour
Fin
```

Complexité

Première borne : chaque appel à l'algorithme `entasser` coûte $O(\lg n)$ et il y a $O(n)$ appels de ce type. La complexité de `construireTas` est donc en $O(n \lg n)$. On peut en fait obtenir une borne plus fine.

En effet, un tas à n éléments est de hauteur $\lfloor \lg n \rfloor$ et à une hauteur h, il contient au maximum $\lceil \frac{n}{2^{h+1}} \rceil$ noeuds. De plus, l'algorithme `entasser` requiert un temps d'exécution en $O(h)$ quand il est appelé sur un tas de hauteur h. D'où :

$$T(n) = \sum_{j=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

Or

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$$

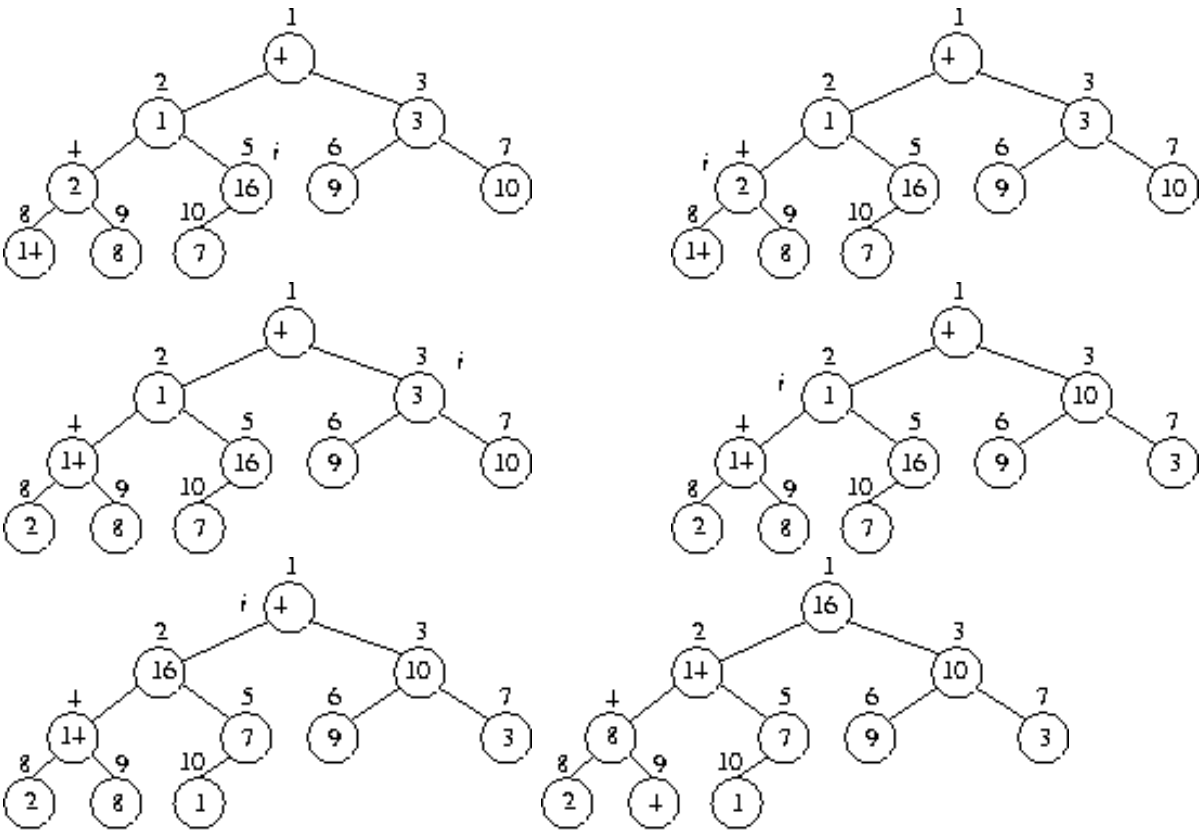
D'où

$$T(n) = O(n)$$

On peut donc construire un tas à partir d'un tableau en temps linéaire.

Illustration de l'algorithme construireTas

Les figures suivantes montrent l'action de `construireTas` sur le tableau [4,1,3,2,16,9,10,14,8,7].



3.6 Algorithme trMaximier

Ayant planté un maximier, il nous reste à cueillir les fruits. En effet `A[1]` a maintenant la valeur du maximum du tableau. Il faut classer le tableau dans l'ordre *croissant*, et non pas décroissant ! Pour cela, il suffit d'exécuter `permuter(A,1,n)` pour que `A[n]` soit maintenant l'élément maximal ; il ne reste plus qu'à trier `A[1..n-1]`. Or ce sous-tableau est lui-même un maximier, à la seule exception possible de l'élément `A[1]| qui est l'ancien \lstinline`A[n]@. On lui appliquera donc l'algorithme `entasser` pour obtenir son maximum en `A[1]`. Le tri du maximier s'écrit donc :

Procédure trMaximier
(Tri par la méthode du tas)

```
Action trMaximier ( DR A : Element [ NMAX ] ; n : Entier )
Début
  |  Pour j <- n à 2 Faire
  |     |  permuterTab ( A , 1 , j )
  |     |  entasser ( A , 1 , j - 1 )
  |  FinPour
Fin
```

Complexité
La complexité de `trmaximier` est :

$$T(n) = \sum_{i=2}^{n} O(h_{1j})$$

où h_{1j} est la profondeur du sous-arbre associé à `A[1..j]`. Comme

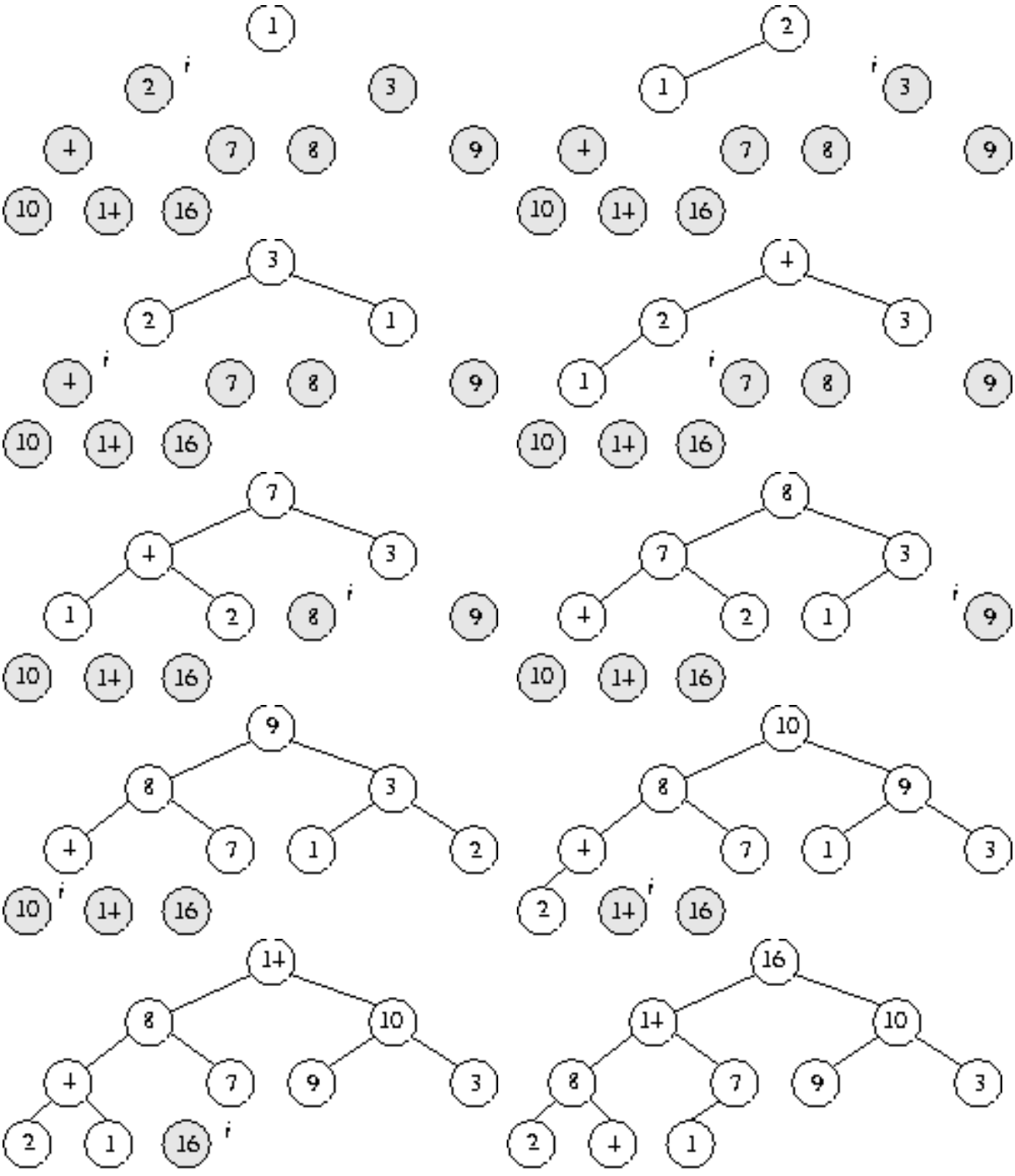
$$h_{1j} = \lfloor \lg n \rfloor + 1$$

on a finalement

$$T(n) = O(n \lg n)$$

3.7 Illustration du tri par tas

Les figures suivantes montrent le tri par tas sur le tableau [4,1,3,2,16,9,10,14,8,7].



3.8 Complexité du tri par tas

La complexité du tri par tas est donc en :

$$O(n + n \lg n) = O(n \lg n)$$

Le point important est que nous n'avons faut aucune hypothèse sur la répartition des clés : **le tri par tas est de complexité moyenne et maximale** $O(n \lg n)$. C'est donc la plus **sûre** de toutes les méthodes de tri comparatifs. Les résultats pratiques font cependant apparaître une constante supérieure à celle du tri rapide dans ses « bons » cas.