TP n° 4 : Tris évolués

Attention : Les algorithmes présentés ici sont *compliqués* : il est donc important de bien les comprendre **avant** de venir en TP.

1 Tri par tas

Le tri par tas (*heapsort* en anglais) utilise une variante très particulière d'arbre binaire pour gérer les éléments à trier : celle d'*arbre binaire parfait partiellement ordonné*. L'algorithme obtenu permet de trier n éléments avec un nombre de comparaisons proportionnel à $n.log_2 n$ *en moyenne* et *dans le pire cas*.

À titre de comparaison, les tris simples, comme le tri par insertion, par sélection ou à bulle, effectuent un nombre de comparaisons proportionnel à n^2 en moyenne et dans le pire cas (par exemple pour $n = 50\,000$, $n^2 \div n.log_2 n \approx 3200$, pour $n = 10^6$, $n^2 \div n.log_2 n \approx 5.10^4$).

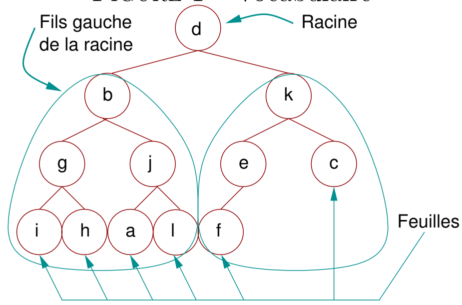
1.1 Vocabulaire d'arbre

NB : ces définitions diffèrent un peu de celles vues en cours.

Un *arbre* est un ensemble d'éléments (*sommets*) tels que :

- la *racine* est l'unique sommet sans père ; tout sommet qui n'est pas la racine possède un unique père
- *fils gauche*, *droit* : les deux sous-arbres directs d'un sommet
- les *feuilles* sont des sommets sans fils ; tout sommet qui n'est pas une feuille possède 1 ou 2 fils.

FIGURE 1 – Vocabulaire



1.2 Arbre binaire parfait et tableau

Un *arbre binaire parfait* est un arbre binaire dont tous les niveaux sont complètement remplis, sauf éventuellement le dernier niveau et dans ce cas les feuilles du dernier niveau sont groupées le plus à gauche possible. Si on numérote les sommets par niveau, on peut représenter un arbre parfait de n sommets dans un tableau de n éléments.

L'arbre binaire de la figure 2 peut donc se représenter par le tableau :

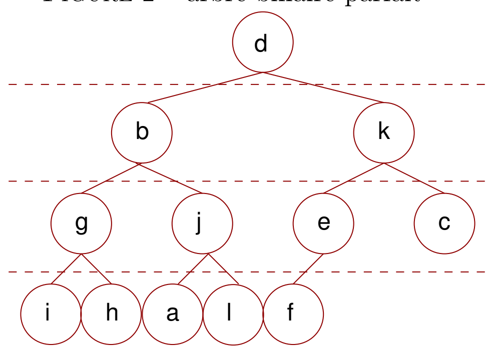
indice	0	1	2	3	4	5	6	7	8	9	10	11
valeur	d	b	k	g	j	e	c	i	h	a	l	f

La *racine* est le sommet d'indice 0.

Soit i l'indice d'un sommet :

- l'indice de son *fils gauche* est $2i + 1$ si $2i + 1 < n$, sinon i n'a pas de fils gauche.
- l'indice de son *fils droit* est $2i + 2$ si $2i + 2 < n$, sinon i n'a pas de fils droit.
- l'indice de son *père* est $(i - 1)/2$ si $i > 0$.

FIGURE 2 – arbre binaire parfait



1.3 Arbre parfait partiellement ordonné et tas

Un arbre *partiellement ordonné* est un arbre tel que la racine de tout sous-arbre a une valeur *supérieure ou égale* à celle de ses fils.

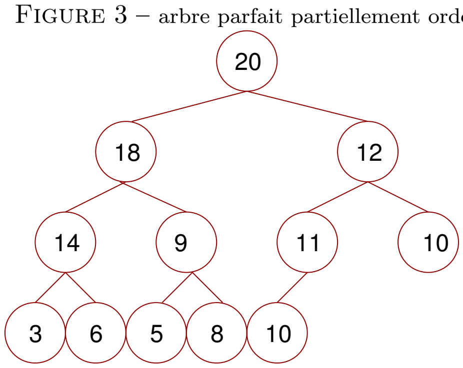
Remarque : l'arbre est *partiellement ordonné*, car il n'y a pas de relation entre les valeurs des « *frères* » ni des « *cousins* ».

Si en plus il est parfait, on parle d'arbre parfait partiellement ordonné (voir figure 3).

Un *tas* est un arbre parfait partiellement ordonné représenté par un tableau ; dans ce cas, le premier élément du tableau est un maximum :

0	1	2	3	4	5	6	7	8	9	10	11
20	18	12	14	9	11	10	3	6	5	8	10

FIGURE 3 – arbre parfait partiellement ordonné



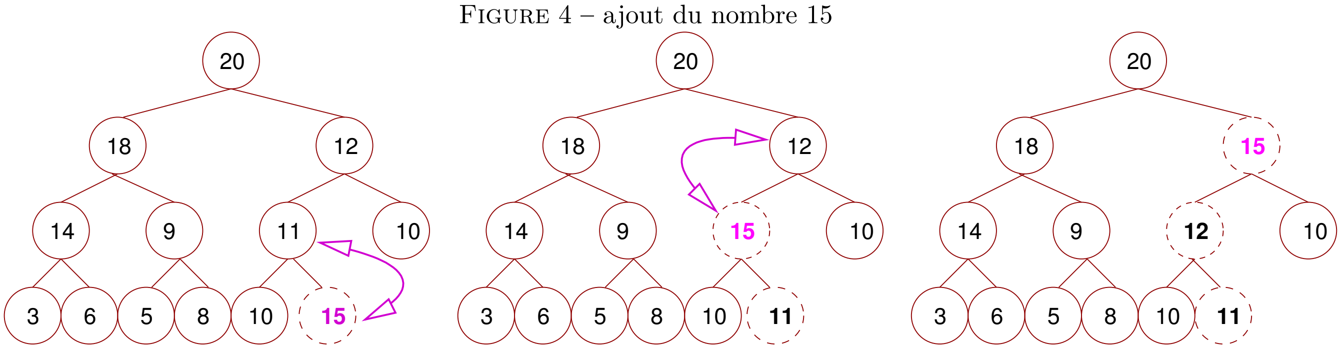
1.3.1 ajout d'élément dans un tas

L'ajout d'un élément dans un tas consiste à placer cet élément dans une feuille au dernier niveau de l'arbre, le plus à gauche possible ; on peut donc obtenir un sous-arbre qui n'est plus ordonné : il faut donc faire remonter l'élément ajouté en l'échangeant avec son père jusqu'à ce que l'arbre soit de nouveau ordonné.

Par exemple, l'ajout du nombre **15** commence par l'ajout d'un fils droit au sous-arbre de racine **11** qui n'est donc plus ordonné ; **15** et **11** sont alors échangés, puis **15** et **12** et on obtient l'arbre de la figure 4 ; le tableau qui représente l'arbre évoluera comme suit :

placement initial de l'entier 15	20	18	12	14	9	11	10	3	6	5	8	10	***15***
échange avec son père 11	20	18	12	14	9	***15***	10	3	6	5	8	10	***11***
échange avec son père 12	20	18	***15***	14	9	***12***	10	3	6	5	8	10	11

FIGURE 4 – ajout du nombre 15

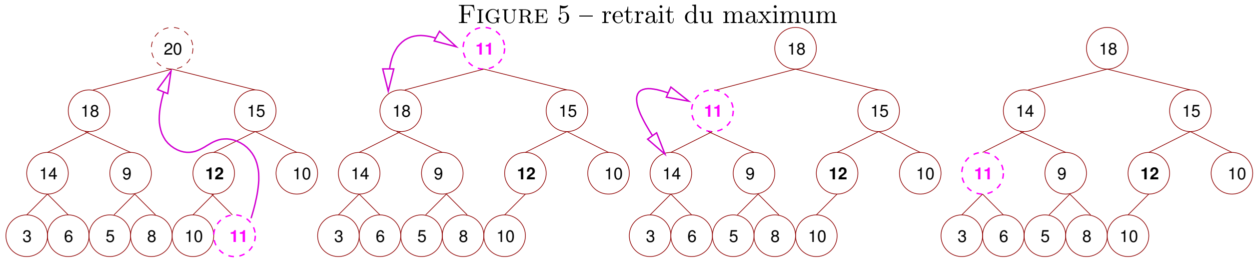


1.3.2 retrait d'un élément d'un tas

L'opération consiste toujours à retirer un maximum, c'est-à-dire l'élément situé à la racine : on remplace l'élément à la racine par la feuille située le plus à droite du dernier niveau et on supprime cette feuille ; on peut donc obtenir un arbre qui n'est plus ordonné : il faut alors itérativement échanger l'élément « remonté » à la racine avec *le plus grand de ses fils*, jusqu'à ce que l'arbre soit à nouveau ordonné (voir figure 5) ; le tableau qui représente l'arbre évoluera comme suit :

situation initiale	***20***	18	15	14	9	12	10	3	6	5	8	10	***11***
remontée de la feuille 11 à la racine	***11***	18	15	14	9	12	10	3	6	5	8	10	
échange avec son mailleur fils 18	***18***	***11***	15	14	9	12	10	3	6	5	8	10	
échange avec son meilleur fils 14	18	***14***	15	***11***	9	12	10	3	6	5	8	10	

FIGURE 5 – retrait du maximum



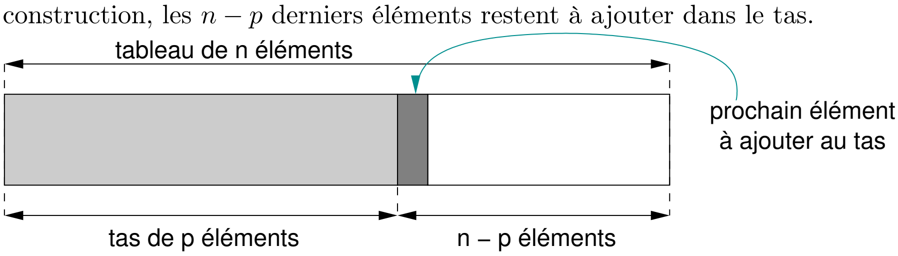
1.4 Trier un tableau avec un tas

1.4.1 algorithme

Pour trier en *ordre croissant* un tableau avec un tas, il faut procéder en deux phases :

1. *construire* un tas avec les n éléments du tableau : ajouter successivement chaque élément du tableau dans le tas, selon le mécanisme décrit au paragraphe 1.3.1 ;

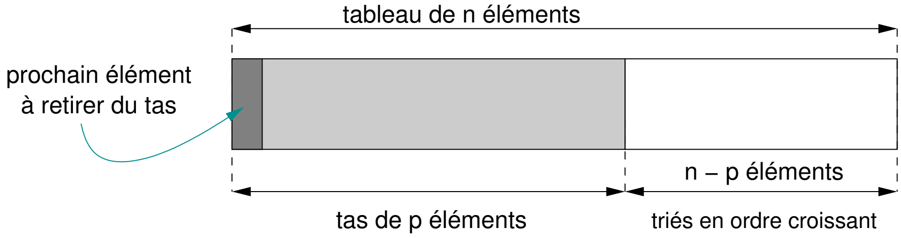
   $\Longrightarrow$ à chaque étape de cette phase, les p premiers éléments du tableau forment un tas en cours de construction, les $n - p$ derniers éléments restent à ajouter dans le tas.

   $\Longrightarrow$ à l'issue de cette phase, le tas est composé des n premiers éléments du tableau.

2. *dé-construire* le tas : prendre le maximum du tas et l'*échanger avec le dernier élément du tas* puis réorganiser le tas, selon le mécanisme décrit au paragraphe 1.3.2 ;

   $\Longrightarrow$ à chaque étape de cette phase, les p premiers éléments forment un tas en cours de déconstruction, les $n - p$ derniers éléments sont ceux déjà triés.

   $\Longrightarrow$ à l'issue de cette phase, le tas est vide ; les n premiers éléments du tableau sont triés en ordre croissant.

1.4.2 Exemple :

Tri en ordre croissant d'un tableau de 13 éléments ; les éléments en gras sont ceux qui sont dans le tas.

Construction du tas

tableau initial de 13 éléments	12	6	20	8	14	10	10	3	18	5	9	11	15
après quelques itérations de construction ; le tas contient les 8 premiers éléments, les 5 derniers restent à ajouter	**20**	**14**	**12**	**6**	**8**	**10**	**10**	**3**	18	5	9	11	15
après ajout de l'élément 18 au tas ; le tas contient les 9 premiers éléments, les 4 derniers restent à ajouter	**20**	**18**	**12**	**14**	**8**	**10**	**10**	**3**	**6**	5	9	11	15
à la fin de la construction du tas ; le tas contient les 13 éléments	**20**	**18**	**15**	**14**	**9**	**12**	**10**	**3**	**6**	**5**	**8**	**10**	**11**

Dé-construction du tas

après suppression du *maximum* (20) et réorganisation ; le tas contient les 12 premiers éléments, le dernier est trié	**18**	**14**	**15**	**11**	**9**	**12**	**10**	**3**	**6**	**5**	**8**	**10**	20
après suppression du *maximum* (18) et réorganisation ; le tas contient les 11 premiers éléments, les 2 derniers sont triés	**15**	**14**	**12**	**11**	**9**	**10**	**10**	**3**	**6**	**5**	**8**	18	20
après suppression du *maximum* (15) et réorganisation ; le tas contient les 10 premiers éléments, les 3 derniers sont triés	**14**	**11**	**12**	**8**	**9**	**10**	**10**	**3**	**6**	**5**	15	18	20
après quelques itérations de déconstruction du tas ; le tas contient les 6 premiers éléments, les 7 derniers sont triés	**10**	**9**	**5**	**8**	**6**	**3**	10	11	12	14	15	18	20
à la fin de la déconstruction du tas ; les 13 éléments sont triés	3	5	6	8	9	10	10	11	12	14	15	18	20

Remarque : Pour bien comprendre l'évolution des éléments dans le tableau, il est conseillé de représenter l'arbre correspondant.

1.4.3 Réalisation du tri

Dans la classe TriTas, programmer les fonctions suivantes :

```
/**
  ajouter tnb[p] au tas formé par les p premiers éléments du tableau tnb.
  @param tnb : tableau dont les p premiers éléments forment un tas
  @param p   : indice de l'élément à ajouter au tas
  @pre 1 ≤ p < tnb.length
  @post les p+1 premiers éléments du tableau tnb forment un tas
 */
static void ajouterTas(int [ ] tnb, int p) ;

/**
  supprimer l'élément maximum du tas et réorganiser le reste du tas.
  @param tnb : tableau dont les p premiers éléments forment un tas
  @param p   : nombre d'éléments dans le tas
  @pre 1 < p ≤ tnb.length
  @post : place l'élément maximum en tnb[p−1] ; les p−1 premiers éléments de tnb forment un
      tas
 */
static void supprimerMax(int [ ] tnb, int p);

/**
  trier un tableau d'entiers en ordre croissant avec l'algorithme du tri par tas
  @param tnb : tableau à trier
  @param nb  : nombre d'éléments dans le tableau
  @pre 1 ≤ nb ≤ tnb.length
 */
public static void trier(int [ ] tnb, int nb);
```

1.5 Programme client

Écrire dans un autre fichier un programme qui effectue les opérations suivantes :

- saisir le nom d'un fichier de données
- lire le fichier et initialiser le tableau de nombres
- trier le tableau de nombres
- ***vérifier (à l'aide d'une fonction à écrire) que le tableau est effectivement trié***
- enregistrer le tableau trié dans un fichier
- si vous le souhaitez, vous pouvez afficher la durée du tri en comptant le nombre de millisecondes qui se sont écoulées entre le début et la fin du tri.

Vous pouvez vous servir des fonctions mises à votre disposition dans la classe OutilsTri :

```
/**
  initialiser un tableau avec des nombres lus dans un fichier
  @param nom : nom du fichier de données
  @return tableau initialisé avec le contenu du fichier
 */
public static int [ ] lireTableau(String nomFichier);

/**
  enregistrer un tableau dans un fichier
  @param tnb          : tableau (probablement trié) de nombres (non modifié)
  @param nbelt        : nombre d'éléments du tableau (0 ≤ nbelt ≤ tnb.length)
  @param nomFichier   : nom du fichier où enregistrer les nombres
  @param nomAlgo      : nom de l'algorithme de tri
 */
public static void enregistrerTableau(int [] tnb, int nbelt,
                                      String nomFichier, String nomAlgo);

//  @return nombre de millisecondes correspondant à l'instant présent
public static long getInstantPresent();
```

1.6 Fichiers fournis

Les fonctions de la classe OutilsTri sont disponibles dans le fichier OutilsTri.jar situé dans le répertoire habituel ; ajoutez ce fichier au au *chemin de génération* (*build path*) de votre projet.

Dans le même répertoire, vous trouverez des fichiers de données contenant des nombres entiers ; certains de ces fichiers sont déjà partiellement ou entièrement triés ; vous pouvez copier ces fichiers dans le répertoire de votre projet et vous en servir pour tester vos algorithmes.

Pour vérifier si vos fichiers résultats sont triés, vous pouvez exécuter la commande diff suivante dans une fenêtre terminal :

> diff *fichier_trié fichier_à_vérifier*

S'il n'y a aucun message, c'est que *fichier_trié* et *fichier_à_vérifier* sont identiques ; sinon, diff affiche les lignes qui diffèrent.

2 Tri rapide

Le tri rapide (*quicksort* en anglais) permet de trier n éléments avec un nombre de comparaisons proportionnel à $n.log_2n$ en moyenne ; par contre, sa version la plus simple se dégrade lorsque les nombres sont déjà partiellement triés et le nombre de comparaisons peut être proportionnel à n^2 dans le pire cas.

2.1 Principe général

Le principe général de l'algorithme du tri rapide d'un tableau est le suivant :

1. choisir un élément qu'on appelle le *pivot* ; on peut, par exemple, prendre le premier élément du tableau (c'est la solution la plus simple, mais pas forcément la plus efficace)
2. partager le tableau en deux parties : la partie gauche contient les éléments de valeur *inférieure ou égale* au pivot, la partie droite contient ceux de valeur *strictement supérieure* au pivot ; le pivot est ensuite placé entre les deux parties (voir figure 6)
3. trier chaque partie séparément selon le même principe.

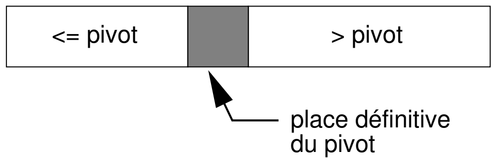


FIGURE 6 – partage du tableau

L'opération de partage se fait directement sur le tableau à trier ; la frontière entre les deux parties est définitive et donne donc l'emplacement définitif de l'élément pivot.

2.2 Principe du partage

On suppose ici qu'on choisit pour pivot le premier élément du sous-tableau à partager.

Le partage d'un sous-tableau (défini par les indices de ses bornes) se fait en un seul parcours, à l'aide de deux indices :

- l'un part de l'extrémité droite et s'arrête quand il désigne un élément de valeur $\leq$ *pivot* ;
- l'autre part *ensuite* de l'extrémité gauche et s'arrête quand il désigne un élément de valeur $>$ *pivot* ;
- on échange les deux éléments ;
- on continue à faire progresser les deux indices et à faire des échanges jusqu'à ce que les indices se croisent.
- Il reste ensuite à mettre le pivot à sa place : il suffit pour cela d'échanger le pivot et l'élément $\leq$ *pivot* le plus à droite.

Attention : la gestion des indices demande un peu de finesse.

Exemple : on trie le tableau en ordre *croissant* et on choisit pour pivot le premier élément de chaque sous-tableau.

indices	0	1	2	3	4	5	6	7	8	9	10	11	12
tableau initial de 13 éléments	15	45	23	50	83	60	2	45	99	3	2	68	30
tableau obtenu après la première opération de partage (le pivot est 15)	[2	2	3]	**15**	[83	60	50	45	99	23	45	68	30]
après partage des éléments d'indices 0 à 2 (le pivot est 2) ; les autres éléments sont inchangés	[2]	**2**	[3]										
après partage des éléments d'indices 4 à 12 (le pivot est 83) ; la partie droite ne contient qu'un élément, elle est donc triée					[68	60	50	45	30	23	45]	**83**	[99]
après partage des éléments d'indices 4 à 10 (pivot = 68)					[45	60	50	45	30	23]	**68**		
etc ...					[45	23	30]	**45**	[50	60]			
tableau trié final	2	2	3	15	23	30	45	45	50	60	68	83	99

2.3 Réalisation du tri

Dans la classe TriRapide, programmer les fonctions suivantes :

```
/** partager les éléments d'un sous-tableau en 2 parties.
  @param T     : tableau à partager
  @param binf , bsup : indices du premier et du dernier élément du sous-tableau à partager
  @pre 0 ≤ binf, bsup < T.length
  @post partage les éléments de T compris entre les indices binf et bsup selon le principe
      décrit au paragraphe 2.2 et met le pivot à sa place.
  @return indice auquel a été placé le pivot
 */
static int partager(int [] T, int binf, int bsup);

/** triRapide : trier récursivement un sous-tableau (algorithme du tri rapide)
  @param T     : tableau à trier
  @param binf,bsup : indices du premier et du dernier élément du sous-tableau à trier
  @pre 0 ≤ binf, bsup < T.length
 */
static void triRapide (int [] T, int binf, int bsup);

/** trier : trier un tableau par ordre croissant avec l'algorithme du tri rapide
  @param T : tableau à trier
  @param nb : nombre d'éléments à trier dans le tableau
  @pre 1 < nb ≤ T.length
 */
public static void trier(int [] T, int nb);
```

NB : la fonction trier est celle qui sera appelée par le programme client ; elle se contente d'effectuer l'appel initial de la fonction récursive triRapide avec les paramètres adéquats.

2.4 Programme client

Compléter le programme client précédemment écrit pour trier les fichiers avec ce deuxième algorithme.