Informatique MP
TP/devoir
TAS ET FILES DE PRIORITE.
Tri en tas
Algorithme de Dijkstra
Problème du sac à dos

Antoine MOTEAU
antoine.moteau@wanadoo.fr

.../Tas.tex (2003) compilé le lundi 05 mars 2018 à 18h 53m 27s avec LaTeX

.../Tas.tex Compilé le lundi 05 mars 2018 à 18h 53m 27s avec LaTeX.
Compilable avec LaTeX, PDFLaTeX, LuaLaTeX, XeLaTeX.

Creation : 2001-2003.
Ordre du TP : 2-ième tiers d'année (TP 5 ou 6)

Eléments utilisés (OCaml) :

- listes
- arbres
- files FIFO (queues)
-

Documents relatifs au TP :

- Texte du TP : .tex, .pdf
-
- Squelette de programme (OCaml) :
- Programme corrigé (OCaml) :

Antoine MOTEAU – Lycée POTHIER – Option Informatique MP
TP/devoir n^o

TAS ET FILES DE PRIORITE.

Applications : Tri en tas, Algorithme de Dijkstra, Problème du sac à dos.

A un vecteur on associe (**de façon uniquement formelle, "imaginaire", "virtuelle"**) un arbre binaire homogène presque complet (de hauteur minimale), tel que dans le parcours en largeur de gauche à droite de cet arbre, les étiquettes rencontrées soient les composantes successives du vecteur (voir figure ci-dessous).
Le nœud de l'arbre virtuel correspondant à la composante d'indice i ($i \geqslant 0$) du vecteur est dit d'indice i dans l'arbre.

Un *tas* (*heap* en anglais) est un triplet $(V, T, .\alpha)$ où

- V est un vecteur (*vecteur du tas*) de longueur L, **indicé de 0 à $L-1$**,
- T est un entier (*taille du tas*) compris entre 0 et L,
- $.\alpha$ est une relation d'ordre (large) sur l'ensemble des éléments de V,

tel que dans l'arbre (***virtuel***) associé au sous-vecteur de V constitué par les T premières composantes de V, l'étiquette d'un nœud interne soit toujours supérieure (au sens de $.\alpha$) à celles de ses fils.

- Les éléments de V d'indices strictement inférieurs à T sont les éléments *valides* (ou *actifs*) du tas, les autres sont *invalides* (ou *inactifs*).
- Le tas est *total* (totalement construit) si tous les éléments de V sont valides ($T = L$) et il est *partiel* (partiellement construit) sinon ($T < L$).

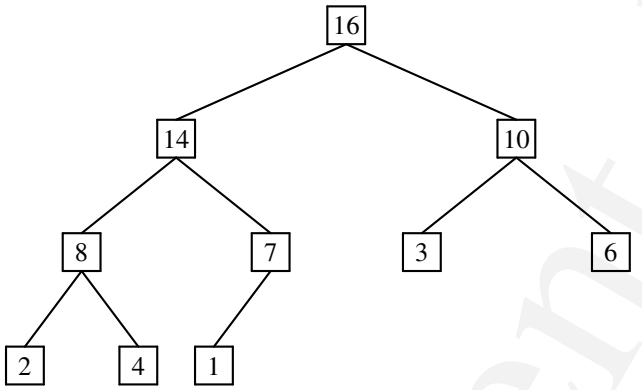

FIGURE 1 – arbre d'un tas total (ordre $\leqslant$), avec T = 10, de vecteur [| 16 ; 14 ; 10 ; 8 ; 7 ; 3 ; 9 ; 2 ; 4 ; 1 |]

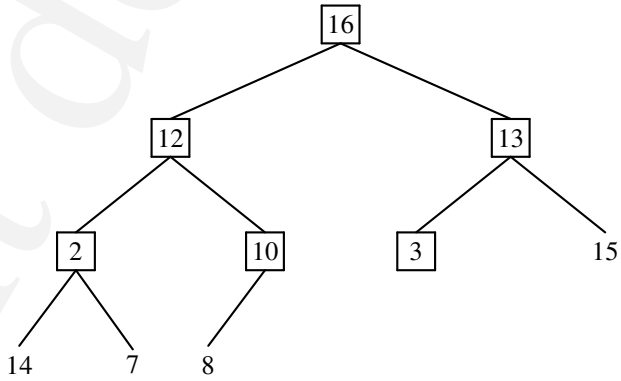

FIGURE 2 – arbre d'un tas partiel (ordre $\leqslant$), avec T = 6, de vecteur [| 16 ; 12 ; 13 ; 2 ; 10 ; 3 ; 15 ; 14 ; 7 ; 8 ; 5 |]

(les éléments actifs sont encadrés)

Remarque. D'après la définition, l'étiquette d'une feuille étant vide, la partie gauche du dernier niveau de l'arbre associé à un vecteur est complètement remplie (arbre quasi-complet, "pesant" à gauche).

On utilise la structure de tas pour
- le tri en tas des vecteurs, asymptotiquement en $O(n \log n)$ (parmi les meilleurs),
- la (une des) représentation des *files de priorité*, qui sont des outils essentiels
  - en gestion de données ou d'informations
  - en gestion de processus et de threads (environnement multi-tâches)
  - dans de nombreux algorithmes (Dijkstra, ...)
- ...

Remarque. Dans la pratique, l'arbre virtuel associé à la partie active d'un tas n'est jamais réellement construit, mais, comme "l'idée de cet arbre" sert de support aux algorithmes de gestion des tas, dans une première partie, on va se familiariser avec cet arbre en le construisant effectivement.

Ressources pour ce TP : dossier `..\MPx\..\TP-Caml\Tas\`, à copier dans votre dossier personnel, contenant

- Tas-S.pdf : énoncé du TP (ce document).
- TasTFD-S.ml : squelette de programme OCaml, à compléter.

1 Arbre binaire virtuel associé à un vecteur (et réciproque)

Pour pouvoir visualiser (par graphes) les résultats des fonctions de gestion des tas et des files de priorité, on construit effectivement ici l'arbre ("virtuel") associé à un vecteur (selon l'ordre des composantes).

Le type `'a arbre` est défini ainsi :

```
type 'a arbre =
    F
  | N of 'a arbre * 'a * 'a arbre
;;
```

Pour la mise au point et la visualisation des résultats, on dispose d'une fonction de représentation graphique (modeste) de (petits) arbres d'étiquettes de type `'a`, pré-écrite :

```
val dessine_arbre : ('a -> string) -> 'a arbre -> unit
```

On rappelle que dans le parcours en largeur de gauche à droite de l'arbre virtuel associé au vecteur V, les étiquettes rencontrées sont les composantes successives du vecteur V.
Le nœud de l'arbre virtuel correspondant à la composante d'indice i ($i \geqslant 0$) du vecteur V est dit d'indice i dans l'arbre.

Question 1. Combien y a t'il de nœuds terminaux (n'ayant que des feuilles comme fils) dans l'arbre associé à un vecteur V de longueur n ?
(formule simple déduite du fait qu'un arbre binaire à k nœuds possède $k+1$ feuilles).

Question 2. Indices du père et des enfants dans l'arbre associé à un vecteur V.

Soit $\mathscr{N}$ le nœud interne d'indice $i \geqslant 0$. On note $P(i)$, $G(i)$ et $D(i)$ les indices des composantes du père de $\mathscr{N}$, du fils gauche de $\mathscr{N}$ et du fils droit de $\mathscr{N}$ (lorsqu'ils existent).
Exprimer $P(i)$, $G(i)$, $D(i)$ en fonction de i .

On pourra s'aider d'une représentation de l'arbre déduit de l'arbre associé à V, par remplacement des étiquettes par les indices (sachant les indices commencent à 0).

Question 3. Construction de l'arbre associé à un vecteur V.

A partir de la composante d'indice i de V, on crée le sous-arbre `N(gauche, V.(i), droite)` où

- `gauche` est l'arbre créé à partir de l'indice $G(i)$ (`gauche=F` si $G(i)$ dépasse la longueur de V)
- `droite` est l'arbre créé à partir de l'indice $D(i)$ (`droite=F` si $D(i)$ dépasse la longueur de V)

et on commence avec la composante d'indice 0 de V, pour obtenir l'arbre associé à V.

Ecrire la fonction `arbre_of_vect`, de type `'a array -> 'a arbre = <fun>` , telle que `arbre of_vect v` renvoie l'arbre associé au vecteur `v`.

Question 4. Construction du vecteur associé à un arbre binaire quasi-complet pesant à gauche.

A l'aide d'un parcours en largeur, de gauche à droite, de l'arbre, on construit la liste des étiquettes rencontrées, qui sont, dans un ordre à préciser, les composantes successives du vecteur associé à l'arbre.

On utilisera une **file FIFO** (First In, First Out) pour stocker les fils au fur et à mesure de leur rencontre, afin de pouvoir les traiter ultérieurement dans le bon ordre (structure fournie par le module Queue de OCaml) :

```
open Queue;;                    (* module des files FIFO (queues) *) (Voir annexe)
let Q = Queue.create () in ... (* création d'une file FIFO, vide       *)
Queue.add t Q;
Queue.length Q;                 (* utile pour tester si Q est vide      *)
let u = Queue.take Q in ...
```

Remarque. Si on connaît au préalable le nombre de nœuds, on peut éviter de passer par une liste et créer directement le vecteur, pour le remplir composante après composante, dans l'ordre de parcours.

1. Parcours d'un arbre en largeur, de gauche à droite, avec $\begin{cases} \text{une fonction d'accumulation,} \\ \text{une valeur initiale de l'accumulateur} \end{cases}$
   - Le résultat en cours est initialisé à la valeur initiale.
     Le nœud racine est introduit initialement dans une file FIFO neuve
   - Tant que la file n'est pas vide, on extrait l'élément prêt à sortir de la file pour le traiter :
     - Pour une feuille, on ne fait rien (c'est fini : arbre quasi-complet, "pesant" à gauche).
     - Pour un nœud (sortant de la file),
       - on introduit son fils gauche puis son fils droit dans la file (pour traitement ultérieur)
       - et on traite l'étiquette du nœud : accumulation de la valeur d'étiquette au résultat en cours.
   - En final, on renvoie le résultat en cours, qui est l'accumulation de toutes les étiquettes.

Ecrire la fonction `it_large_arbre`, de type `('a -> 'b -> 'a) -> 'a -> 'b arbre -> 'a`, telle que `it_large_arbre f a t` renvoie `f (... (f (f a b0) b1) ...) bn` si `b0, b1, ..., bn` sont, dans cet ordre, les étiquettes rencontrées dans le parcours en profondeur de gauche à droite de l'arbre `t`.

2. En choisissant comme paramètres la fonction qui accumule l'étiquette en tête d'une liste initialement vide, on en déduit une fonction qui donne la liste inversée des composantes du vecteur :

```
let list_large_of_arbre t = it_large_arbre (fun x y -> y::x) [ ] t;;
(* list_large_of_arbre : 'a arbre -> 'a list = <fun> *)
```

3. D'où une fonction qui donne le vecteur associé à un arbre arbre quasi-complet pesant à gauche :

```
let vect_of_arbre t = vect_of_list (List.rev (list_large_of_arbre t));;
(* vect_of_arbre : 'a arbre -> 'a vect = <fun> *)
```

Question 5. Accès à la valeur de $V.(i)$ depuis l'arbre associé à V.

1. Construction d'un chemin d'accès :

   A partir de la racine de l'arbre associé au vecteur V, pour rechercher le nœud d'indice i (d'étiquette $V.(i)$), on utilise une liste indiquant la nature du déplacement à effectuer :

   - si la tête de liste vaut 0, on descend vers le fils gauche, avec le reste de la liste ;
   - si la tête de liste vaut 1, on descend vers le fils droit, avec le reste de la liste ;
   - lorsque la liste est vide, on est arrivé.

   Ecrire la fonction `chemin`, de type `int -> int list`, qui, à un indice $i \geqslant 0$, associe la liste indiquant le déplacement à effectuer pour accéder au nœud correspondant à la composante i d'un vecteur (où les indices débutent à 0) :

   Indication. Il faut penser "à l'envers" : pour un indice $i > 0$,
   – comment déceler si on a affaire à un fils gauche ou un fils droit ?
   – qui est son père ?
   ou alors interpréter ce chemin comme la liste des bits de l'écriture binaire de $i+1$ (poids forts en tête), privée de sa tête (bit de poids fort).

   Quel est la complexité de cette fonction ?

2. En déduire la fonction `arbre_item`, de type `'a arbre -> int -> 'a`, telle que `arbre_item t i` est la `i`-ième composante du vecteur `v` associé à l'arbre `t`.

   Quel est la complexité de cette fonction ?

Représentation de vecteurs dynamiques (à longueur variable) par un arbre :

D'après ce qui précède, on pourrait utiliser réellement la structure d'arbre associé à un vecteur (en y définissant les étiquettes comme un type mutable) pour représenter un type vecteur à longueur dynamique.

Avantage : la possibilité d'allonger ou de rétrécir les vecteurs, sans coût de recopie.

Inconvénient : un coût d'accès aux composantes en $O(\log n)$ (au pire) au lieu de $O(1)$.

Dans la suite, on travaille seulement avec "l'idée", "l'odeur" de l'arbre, sans le construire.

2 Gestion des tas

On introduit le type `tas`, conformément à la définition de l'introduction :

```
type 'a tas = { vect : 'a array ; mutable taille : int ; ordre : 'a -> 'a -> bool }
;;
```

Le champ `vect` est (naturellement) modifiable en place, le champ `taille` (mutable) est modifiable en place.

Le type `tas` représente en fait un vecteur, accompagné d'informations complémentaires. Les informations taille du tas (`taille`) et ordre du tas (`ordre`, relation d'ordre à utiliser), permettront de travailler avec l'arbre **virtuel** associé au vecteur (`vect`) du tas.

Un tas étant étroitement (bijectivement) lié à l'arbre virtuel associé au vecteur, on utilisera souvent le vocabulaire des arbres pour décrire un tas . . .

On se propose, à partir d'un tas partiel, dont les nœuds terminaux sont naturellement considérés comme étant des tas totaux, de faire évoluer ce tas vers un tas total.

Question 6. Intermédiaire de construction d'un tas total : la fonction `entasser` .

On se place dans le cas où, pour le nœud $\mathscr{N}$ d'indice $i \geqslant 0$ (de l'arbre virtuel associé au vecteur du tas), le fils gauche et le fils droit de $\mathscr{N}$ sont les racines d'arbres (virtuels) représentant des sous-tas, totalement construits.

L'étiquette de $\mathscr{N}$ peut violer la règle qui veut que cette étiquette soit "supérieure" à celle de ses fils, (situation que l'on rencontrera en cours de construction du tas ou lorsque l'on modifie la composante d'indice i du vecteur de tas).

- Si ce n'est pas le cas, l'arbre enraciné en $\mathscr{N}$ correspond à un sous-tas total.
- Si c'est le cas, on échange l'étiquette de $\mathscr{N}$ avec celle de son fils de plus grande étiquette (échange de deux composantes du vecteur du tas) et on poursuit le processus avec ce fils.
  Le processus se termine lorsque les indices en jeu (qui vont en croissant) dépassent la taille du tas et l'arbre enraciné en $\mathscr{N}$ correspond alors à un sous-tas total.

Ecrire la fonction `entasser`, de type `'a tas -> int -> unit`, telle que `entasser t i`, où `t` est un tas et `i` est l'indice d'un nœud dont les fils sont des sous-tas valides de `t`, modifie le vecteur du tas `t` de façon à ce que le sous-arbre de `t` enraciné en `i` soit un sous tas total.

Cette fonction doit avoir une complexité maximale en $O(\log n)$ où n est la taille du tas.

Question 7. Construction d'un tas total.

A l'origine, on suppose que le tas, de vecteur de longueur n, est (presque) complètement invalide : seules les $\left\lfloor \frac{n+1}{2} \right\rfloor$ dernières composantes du vecteur du tas peuvent être considérées comme des (racines de) sous-tas totaux (elles sont les noeuds terminaux de l'arbre associé au vecteur, racines d'arbres réduits à un seul nœud, qui constituent automatiquement des sous-tas totaux).

- On fixe la taille du tas à n (on ne considérera que les sous tas se terminant à l'indice n du vecteur de tas).
- En "remontant" le vecteur depuis la composante d'indice $\frac{n}{2}$ (dernier indice où l'étiquette du père peut être en confit avec celles de ses fils) jusqu'à la composante d'indice 0, la fonction `entasser` va permettre de construire un tas total.

Ecrire la fonction `construire_tas`, de type `'a tas -> unit`, telle que `construire_tas t`, où `t` est un tas, transforme `t` en un tas total.

Cette fonction doit avoir une complexité maximale en $O(n \log n)$, où n est la longueur du vecteur de tas.

3 Tri en tas (heap sort) d'un vecteur

Le tri en tas, qui est de complexité asymptotique $O(n \log n)$, est du point de vue asymptotique le meilleur tri. Cependant, en complexité moyenne il est deux fois moins rapide que le tri "rapide" (de complexité asymptotique $O(n^2)$ et de complexité moyenne $O(n \log n)$).

Question 8. Utilisation d'un tas pour réaliser le tri (en place) d'un vecteur

On construit un tas total valide sur le vecteur, puis,

- L'élément le plus "grand" étant d'indice 0, on l'échange avec le dernier (d'indice taille-1).
- On réduit alors la taille du tas d'un cran (le dernier élément est à sa place) et comme l'élément d'indice 0 ne respecte plus la structure de tas (mais ses fils sont des racines de tas totaux) on exécute `entasser` à partir de l'indice 0 pour reconstruire le tas (dans la partie limitée à la taille).
- Il ne reste plus qu'à recommencer jusqu'à ce que la taille devienne nulle.

Ecrire la fonction `tri_par_tas`, de type `('a -> 'a -> bool) -> 'a array -> unit`, telle que `tri_par_tas o v`, où `v` est un vecteur et `o` une relation d'ordre sur les éléments de `v`, trie le vecteur `v` selon l'ordre `o` (remarque : `v` est trié en place).

Cette fonction doit avoir une complexité maximale en $O(n \log n)$, où n est la longueur du vecteur.

4 Files de priorité

Les files de priorité sont destinées à la gestion d'un ensemble de tâches (ou de valeurs), selon leur ordre d'importance ou leur temps de déclenchement (niveau de priorité).

- Les objets contenus dans une file de priorité comportent au moins une composante (`clé`) indiquant le niveau de priorité, selon une relation d'ordre sur l'ensemble des clés. On convient ici que, à priorité égale,
  - si la relation d'ordre est stricte, le plus ancien (historiquement) soit placé devant le plus récent
  - si la relation d'ordre est large, le plus récent (historiquement) soit placé devant le plus ancien
- La signature d'une file de priorité comprend les fonctions qui réalisent :
  - l'introduction d'une nouvelle tâche (ou valeur)
  - l'extraction de la tâche (ou valeur) de plus haute importante (prioritaire)
  - la modification de la priorité d'une tâche (ou valeur)

L'utilisation d'une structure de tas, moins contraignante que celle d'arbre binaires de recherche (ABR) et qui présente une hauteur systématiquement minimale (obtenue avec un coût moindre que dans les structures d'ABR équilibrés), est particulièrement adaptée à la représentation des files de priorité.

```
type 'a prior == 'a tas
;;
let prior_new ordre n a = ( {vect = Array.make n a; taille = 0; ordre = ordre} : 'a prior )
;;
(* prior_new : ('a -> 'a -> bool) -> int -> 'a -> 'a prior = <fun> *)
```

Remarque. On utilise ici une structure de tas avec un vecteur **de longueur fixe**, ce qui entraîne que

- la file de priorité ne peut contenir qu'un nombre limité de tâches
- lorsque le nombre de tâches est faible, il peut y avoir beaucoup de place perdue.

On pourrait utiliser une structure de vecteur à longueur variable (par exemple réellement arborescente) pour gérer une file de priorité dont la taille maximale est peu prévisible, avec des accès aux composantes du vecteur en $O(\log n)$ au lieu de $O(1)$. Mais alors autant traiter avec des ABR équilbrés.

Question 9. Extraction de la tâche prioritaire.

Si la taille est strictement positive, la tâche prioritaire se trouve à la racine de l'arbre "virtuel" (composante 0 du vecteur). Il suffit de prendre cette tâche, de mettre dans la composante d'indice 0 la dernière composante valide (d'indice taille-1), de réduire la taille de 1 et de rétablir (avec `entasser`) la structure de tas qui a pu être détruite par la nouvelle valeur de la composante d'indice 0.

Ecrire la fonction `prior_take`, de type `'a prior -> 'a`, telle que `prior_take p` extrait et retourne l'élément prioritaire de la file de priorité `p`, en maintenant la structure de tas de `p`.
Cette fonction doit provoquer l'erreur "Erreur :file vide" si `p` est vide.

Question 10. Introduction d'une nouvelle tâche dans la file de priorité.

Si le tas n'est pas plein, a priori, on place cette nouvelle tâche en dernière position (au premier indice invalide, c'est à dire à l'indice repéré par la taille du tas) et on ajoute un à la taille du tas.
Puis, dans l'arbre imaginaire associé au tas, tant que le père de la tâche insérée a une priorité inférieure à cette tâche, on échange le père avec le fils, ce qui donne un sous-tas total, enraciné à la tâche à insérer.
Ainsi, en remontant dans l'arbre, on arrivera à un nœud où la tâche à insérer sera à sa place et l'arbre entier sera un arbre de tas total.

Ecrire la fonction `prior_add`, de type `'a prior -> 'a -> unit`, telle que `prior_add p a` insère la tâche `a` dans la file de priorité `p`, selon l'ordre établi pour les éléments de `p`.
Cette fonction doit provoquer l'erreur "Erreur :file pleine" si `p` est plein.

Question 11. Modification de la priorité (ou remplacement) d'une tâche.

Il s'agit de remplacer la tâche d'indice i par une autre tâche, de priorité différente.
Dans l'arbre imaginaire associé au tas, on remplace la tâche d'indice i par la nouvelle tâche.

- si la priorité a diminué, il faut rétablir la structure de tas pour l'arbre enraciné en i (ce qui peut faire "descendre" la tâche dans l'arbre)
- si la priorité a augmenté, alors il faut faire "remonter" cette tâche vers la racine de l'arbre jusqu'à ce qu'elle se trouve à une place convenable (par échanges père-fils, comme pour l'insertion).

Ecrire la fonction `prior_changeto`, de type `'a prior -> int -> 'a -> unit`, telle que
`prior_changeto p i a` remplace la tâche d'indice `i` par la tâche `a`, en rétablissant la structure de tas de `p`.

5 Algorithme de Dijkstra (version utilisant les files de priorité)

Recherche du plus court (meilleur) chemin entre deux sommets d'un graphe orienté valué positivement.

Soit un graphe orienté valué positivement,

à n sommets, numérotés de 0 à $n-1$,

et m arcs valués, représentés par une *matrice d'adjacence* A telle que, pour deux sommets distincts u et v,

si $A.(u).(v) \leqslant 0 \quad (=-1)$, il n'y a pas d'arc allant du sommet u au sommet v

si $A.(u).(v) > 0$, $A.(u).(v)$ est la longueur de l'arc allant du sommet u au sommet v.

Remarque. Implicitement, $A.(u).(u) = 0$. Quand il y a un arc allant de u à $v \neq u$, v est dit *adjacent* à u.

Etant donné un sommet initial s, on cherche à établir les plus courts chemins allant de s aux autres sommets.
Toutes les longueurs étant strictement positives, l'algorithme de Dijkstra permet de résoudre ce problème.
Ici, on réalise l'algorithme de Dijkstra à l'aide **d'une file de priorité** conçue comme un tas.
Quelques ressources complémentaires relatives aux matrices :

```
value print_int_matrix : int array array -> unit = <fun>
let copy_matrix m = Array.map Array.copy m;;              (* utiliser map est fondamental *)
(* copy_matrix : 'a array array -> 'a array array = <fun> *)
```

Remarque. Ces ressources ne sont pas utiles à la réalisation de l'algorithme de Dijkstra.

Question 12. Ecriture de l'algorithme de Dijkstra (plus courts chemins d'origne s).

1. Structures utilisées dans le cas d'un graphe à n sommets et à valuations (distances) entières :
   - vecteur `d` tel que `d.(u)` représente la distance sur un plus court chemin de s à u,
   - vecteur `p` tel que `p.(u)` soit le père de u sur un plus court chemin menant de s à u,
   - file de priorité `fip`, qui est un tas,
     - pouvant contenir n couples $(d.(u),u)$
     - pour la relation d'ordre `?<` définie par (du,u) `?<` (dv,v) ssi $(dv,v) < (du,u)$, soit $\begin{cases} dv < du \\ dv = du \quad \text{et} \quad v < u \end{cases}$

     (si $dv < du$, alors (dv,v) est "prioritaire" sur (du,u), puisque (du,u) `?<` (dv,v)).
2. Initialisations, pour les chemins d'origine s :
   - les n éléments de `d` sont initialisés à `max_int` (sauf $d.(s) = 0$),
   - les n éléments de `p` sont initialisés à -1. $\left(\begin{array}{l}\text{c'était un p'ov gas, y s'app'lait Armand,} \\ \text{l'avait pas d'papa, l'avait pas d'maman ...}\end{array}\right)$
   - `fip` est initialisée de telle sorte que,
     - pour tout indice $u = 0 \cdots n-1$, l'élément (du vecteur de tas) d'indice u soit $(d.(u),u)$,
     - la taille (du tas) soit n,

     et, si $s \neq 0$, on échange la composante d'indice 0 avec celle d'indice s, ce qui met $(d.(s),s)$ en tête de `fip`.
     `fip` représente alors un tas valide (total) pour la relation d'ordre `?<`
3. Algorithme (plus courts chemins d'origne s) : Tant que la file `fip` n'est pas vide,
   - On extrait la tête de `fip`, sommet u qui, parmi les sommets encore dans la file, est un sommet le plus proche de s.
   - Pour tous les sommets qui sont dans `fip` et qui sont adjacents à u, on examine si leur distance à s peut s'améliorer en passant par u.

     Si oui, on met à jour $\begin{cases} \text{leur distance à } s \text{ dans } \texttt{d}, \\ \text{leur distance à } s \text{ dans } \texttt{fip} \text{ (modification de priorité)}, \\ \text{et on leur affecte } u \text{ comme père (dans } \texttt{p}). \end{cases}$

   Lorsque la file `fip` est vide, `d` et `p` contiennent le résultat cherché.

Ecrire la fonction `dijkstra`, de type `int array array -> int -> int array * int array` telle que `dijkstra m s`, renvoie le couple `(d,p)` des vecteurs `d` (distances) et `p` (pères) caractérisant les plus courts chemins d'origine `s` du graphe orienté de matrice d'adjacence `m`.

Question 13. Description d'un plus court chemin de s à u .

On peut utiliser le résultat (`p`) de l'algorithme de Dijkstra sur le sommet initial s pour décrire, à l'aide d'une liste, un plus court chemin allant du sommet initial s à un sommet u :

Ecrire la fonction `court_chemin`, de type `int array -> int -> int list` telle que `court_chemin p u`, où `p` est le vecteur "père" obtenu par l'algorithme de Dijkstra relativement au sommet initial `s`, renvoie la liste des sommets reliant `s` à `u` par un plus court chemin, sous la forme `[s; ... ;u]`.

6 Autres applications (de variantes) de l'algorithme de Dijkstra

Calcul des itinéraires routiers, le poids des arcs pouvant être la distance (pour le trajet le plus court), le temps estimé (pour le trajet le plus rapide), la consommation de carburant et coût des péages (pour le trajet le plus économique).

Le protocole "open shortest path first" utilise l'algorithme de Dijkstra pour obtenir un routage internet très efficace des informations en cherchant le parcours le plus efficace.

Les routeurs "IS-IS" (de bases de données partagées) utilisent également l'algorithme de Dijkstra.

...

7 Annexe : Implémentation des piles LIFO et des files FIFO en Caml

7.1 Caml Light

7.1.1 Piles LIFO (stacks) : contenu de stack.ml (Caml-Light)

Il s'agit en fait de l'utilisation quasi-immédiate de la structure de liste usuelle.

```
exception Empty;;
type 'a t = { mutable c : 'a list };;
let create () = { c = [] };;
let clear s = s.c <- [];;
let copy s = { c = s.c };;
let push x s = s.c <- x :: s.c;;
let pop s =  match s.c with
                hd::tl -> s.c <- tl; hd
              | []     -> raise Empty
;;
let top s = match s.c with
                 hd::_ -> hd
               | []    -> raise Empty
;;
let is_empty s = (s.c = []);;
let length s = list__length s.c
let iter f s = list__iter f s.c;;
```

7.1.2 Files FIFO (queues) : contenu de queue.ml (Caml-Light)

Utilisation d'une liste (`head`) chainée par références, avec un pointeur (`tail`) sur la fin de la liste.
(*remarque* : en O-Caml, c'est le premier élément de la liste qui joue le rôle du pointeur sur la fin de liste.)
La compréhension du fonctionnement (surtout de la fonction `add`) n'est pas immédiate ...

```
exception Empty;;
type 'a queue_cell =
    Nil
  | Cons of 'a * 'a queue_cell ref
;;
type 'a t = { mutable head: 'a queue_cell; mutable tail: 'a queue_cell } ;;
let new () = { head = Nil; tail = Nil };;
let clear q = q.head <- Nil; q.tail <- Nil;;

let add x = function
    { head = h; tail = Nil as t } ->           (* if tail = Nil then head = Nil *)
              let c = Cons(x, ref Nil) in
              h <- c; t <- c
  | { tail = Cons(_, ref newtail) as oldtail } ->
              let c = Cons(x, ref Nil) in
              newtail <- c; oldtail <- c
;;
let peek q = match q.head with
    Nil               ->  raise Empty
  | Cons(x, ref rest) ->  x
;;
let take q = match q.head with
    Nil               -> raise Empty
  | Cons(x, ref rest) -> q.head <- rest;
                         begin match rest with
                                 Nil -> q.tail <- Nil
                               | _   -> ()
                         end;
                         x
;;
let rec length_aux = function
    Nil               -> 0
  | Cons(_, ref rest) -> succ (length_aux rest)
;;
let length q = length_aux q.head      (* En O-caml,la longueur est incrite dans la structure *)
;;
let rec iter_aux f = function
    Nil               ->  ()
  | Cons(x, ref rest) ->  f x; iter_aux f rest
;;
let iter f q = iter_aux f q.head
;;
```

7.2 OCaml

7.2.1 Piles LIFO (stacks) : contenu de Stack.ml (OCaml)

Il s'agit en fait de l'utilisation quasi-immédiate de la structure de liste usuelle.

```
type 'a t = { mutable c : 'a list }
exception Empty
let create () = { c = [] }
let clear s = s.c <- []
let copy s = { c = s.c }
let push x s = s.c <- x :: s.c
let pop s =  match s.c with
    hd::tl -> s.c <- tl; hd
  | []     -> raise Empty
let top s = match s.c with
    hd::_ -> hd
  | []    -> raise Empty
let is_empty s = (s.c = [])
let length s = List.length s.c
let iter f s = List.iter f s.c

(* === Fin (module Stack.ml de O-Caml === *)
```

7.2.2 Files FIFO (queues) : contenu de Queue.ml (OCaml)

```
exception Empty

(* O'Caml currently does not allow the components of a sum type to be mutable.
   Yet, for optimal space efficiency, we must have cons cells whose [next] field is mutable.
   This leads us to define a type of  cyclic lists, so as to eliminate the [Nil] case and
   the sum type. *)

type 'a cell = { content: 'a; mutable next: 'a cell }

(* A queue is a reference to either nothing or some cell of a cyclic list.
   By convention, that cell is to be viewed as the last cell in the queue.
   The first cell in the queue is then found in constant time: it is the next
   cell in the cyclic list.
   The queue's length is also recorded, so as to make [length] a constant-time operation.

   The [tail] field should really be of type ['a cell option], but then it would be [None]
   when [length] is 0 and [Some] otherwise, leading to redundant memory allocation and accesses.
   We avoid this overhead by filling [tail] with a dummy value when [length] is 0. Of course, this
   requires bending the type system's arm slightly, because it does not have dependent sums. *)

type 'a t = { mutable length: int; mutable tail: 'a cell }

let create () = { length = 0;
                  tail = Obj.magic None
                }

let clear q =
  q.length <- 0;
  q.tail   <- Obj.magic None

let add x q =
  q.length <- q.length + 1;
  if q.length = 1
  then
       let rec cell = { content = x; next = cell } in
       q.tail <- cell
  else
       let tail = q.tail in
       let head = tail.next in
       let cell = { content = x; next = head } in
       tail.next <- cell;
       q.tail <- cell

let push = add

let peek q =
  if q.length = 0
  then raise Empty
  else q.tail.next.content

let top = peek
```

```
let take q =
  if q.length = 0 then raise Empty;
  q.length <- q.length - 1;
  let tail = q.tail in
  let head = tail.next in
  if head == tail
  then q.tail <- Obj.magic None
  else tail.next <- head.next;
  head.content

let pop = take

let copy q =
  if q.length = 0
  then create()
  else let tail = q.tail in
       let rec tail' = { content = tail.content; next = tail' } in
       let rec copy cell =
            if cell == tail then tail'
            else { content = cell.content; next = copy cell.next }
       in
       tail'.next <- copy tail.next;
      { length = q.length;  tail = tail' }

let is_empty q = q.length = 0

let length q = q.length

let iter f q =
  if q.length > 0 then
    let tail = q.tail in
    let rec iter cell =
        f cell.content;
        if cell != tail then iter cell.next
    in
    iter tail.next

let fold f accu q =
  if q.length = 0 then accu
  else let tail = q.tail in
       let rec fold accu cell =
           let accu = f accu cell.content in
           if cell == tail then accu
           else fold accu cell.next
       in
       fold accu tail.next

let transfer q1 q2 =
  let length1 = q1.length in
  if length1 > 0 then
    let tail1 = q1.tail in
    clear q1;
    if q2.length > 0 then begin
      let tail2 = q2.tail in
      let head1 = tail1.next in
      let head2 = tail2.next in
      tail1.next <- head2;
      tail2.next <- head1
    end;
    q2.length <- q2.length + length1;
    q2.tail <- tail1

(* === Fin (module Queue.ml de O-Caml === *)
```

$< \mathscr{F}\mathscr{I}\mathscr{N} >$ (TP Tas. Enoncé.)

TP Tas. Eléments pré-écrits, puis éléments à compléter et exemples

```
(* OCaml *)
#load "graphics.cma";;
open Graphics;;
open_graph  " 800x800+10+10";; (* l'espace  semble  nécessaire!  *)
(*==================================*)
(*   Arbres et vecteurs associés    *)
(*==================================*)
type 'a arbre =
    F
  | N of 'a arbre * 'a * 'a arbre
;;

let a0 = N (N (N (F, 14, F), 2, N (F, 8, F)), 1, N (N (F, 7, F), 16, F)),
           4,
           N (N (F, 9, F), 3, N (F, 10, F)))
;;
(* === représentation graphique (petite fonction modeste, pour de petits arbres) === *)
let graph_dx = 2*( (3 * fst (text_size "0") )/2);;
let graph_dy = 2*( 2 * snd (text_size "0"));;

let dessine_arbre sp t =
  let rec ecart = function
    | F          -> 1
    | N(g, p, d) -> (ecart g) + (ecart d)
  and     dessine_noeud x y = function
    | F          -> ()
    | N(g, p, d) ->
          let zg = max 1 (ecart g) and zd = max 1 (ecart d ) in
          lineto x y;
          dessine_noeud (x - zd * graph_dx) (y - graph_dy) g;
          moveto x y;
          dessine_noeud (x + zg * graph_dx) (y - graph_dy) d;

          let s = sp p in
          let (u,v) = text_size s in
          set_color yellow; fill_circle (x-u/2) (y-v/2) 12;
          set_color blue; moveto (x-u) (y-10); draw_string s;    (* pos un pneu empirique *)
          set_color black
  in let  x = (size_x () /2) and y = (size_y () - graph_dy)
     in clear_graph ();
     moveto x y; dessine_noeud x y t
;; (* dessine_arbre : ('a -> string) -> 'a arbre -> unit = <fun> *)
dessine_arbre string_of_int a0;;

(* === Utilitaires matriciel (pour DijKstra , question 11) === *)
let print_int_matrix m =
  print_newline ();
  for i = 0 to Array.length m -1
  do for j = 0 to Array.length m.(0) -1
     do if (m.(i).(j) = max_int) || (m.(i).(j) = -max_int)   (* || or *)
        then Printf.printf "    %c" (char_of_int 64)         (* avec 4 espaces; simule infini par @ *)
        else Printf.printf "%5d" m.(i).(j)
     done;
     print_newline ()
  done
;;      (* print_int_matrix : int array array -> unit  *)

let copy_matrix m = Array.map Array.copy m   (* utiliser map est fondamental *)
;; (* copy_matrix : 'a array array -> 'a array array = <fun> *)

(*============================================================================*)
(* une fonction "bidon" qui évite le plantage de la suite. Ne pas modifier. *)
let A_ECRIRE s = print_string ("A ECRIRE : " ^ s) ; print_newline ()
;;   (* A_ECRIRE : string -> unit = <fun> *)
(*============================================================================*)
```

```
(*=============================================================================*)
(*                                                  INCORRECT A PARTIR D'ICI   *)
(*=============================================================================*)
(* === Question 1. -- ... *)
(* === Question 2. -- ... *)
(* === Question 3. -- Construction de l'arbre associé à un vecteur === *)

let arbre_of_vect v = A_ECRIRE "4 lignes avec fonction auxiliaire"
;;
(* arbre_of_vect : 'a array -> 'a arbre = <fun> *)

let v = [|4;1;3;2;16;9;10;14;8;7|];;
dessine_arbre string_of_int (arbre_of_vect v);;

(* === Question 4. --- Construction du vecteur associé à un arbre === *)
open Queue;;
let it_large_arbre f a t =
  let b = ref a and q = queue__new () in

  Queue.add t q;
  while Queue.length q > 0
  do match (Queue.take q) with
        F        -> ()
      | N(g,p,d) -> A_ECRIRE "3 instructions sur 1 ligne"
  done;
  !b
;;
(* it_large_arbre : ('a -> 'b -> 'a) -> 'a -> 'b arbre -> 'a = <fun> *)

let list_large_of_arbre t = rev (it_large_arbre (fun x y -> y::x) [] t)
;;
(* list__large_of_arbre : 'a arbre -> 'a list = <fun> *)

let vect_of_arbre t = Array.of_list (list_large_of_arbre t)
;;
(* vect_of_arbre : 'a arbre -> 'a array = <fun> *)

vect_of_arbre a0;;

(* === Question 5. --- Accès à l'élément d'indice i dans l'arbre === *)
(* le calcul du chemin peut être récursif ou itératif *)

let chemin i = A_ECRIRE "4 lignes à écrire"  (* 0 = gauche; 1 = Droite *)
;;
(* chemin : int -> int list = <fun> *)

chemin 7;; chemin 13;;chemin 10;; chemin 20;;

let arbre_item t i =
  if (i < 0) then failwith "erreur"
  else let l = chemin i in
       let rec aux = fun          A_ECRIRE "3 lignes"
           ....
       in aux t l
;;
(* arbre_item : 'a arbre -> int -> 'a = <fun> *)

let v = [|4;1;3;2;16;9;10;14;8;7|];;  let a = arbre_of_vect v;; arbre_item a 5;;

(*==================================*)
(*   Tas                            *)
(*==================================*)
type 'a tas = {vect : 'a array; mutable taille : int; ordre : 'a -> 'a -> bool }
;;
let swap_vect v i j = let u = v.(i) in v.(i) <- v.(j); v.(j) <- u
;;
(* swap_vect : 'a array -> int -> int -> unit = <fun> *)
```

```
(* === Question 6. --- Intermédiaire de construction de tas total === *)
let entasser t i =
  let m = ref 0 in
  let rec aux i = A_ECRIRE "5 ou 6 lignes"
  in aux i
;;
(* entasser : 'a tas -> int -> unit = <fun> *)

let t2 = { vect = [|16;4;10;14;7;9;3;2;8;1|]; taille = 10; ordre = prefix < };;
dessine_arbre string_of_int (arbre_of_vect t2.vect);;
entasser t2 1;;
dessine_arbre string_of_int (arbre_of_vect t2.vect);;

(* === Question 7. --- Construction d'un tas total === *)
let construire_tas t = A_ECRIRE "3 lignes"
;;
(* construire_tas : 'a tas -> unit = <fun> *)

let t3 = { vect = [|4;1;3;2;16;9;10;14;8;7|]; taille = 0; ordre = prefix > };;
dessine_arbre string_of_int (arbre_of_vect t3.vect);;
construire_tas t3;;
dessine_arbre string_of_int (arbre_of_vect t3.vect);;
let t4 = { vect = [|4;1;3;2;16;9;10;14;8;7|]; taille = 0; ordre = prefix < };;
dessine_arbre string_of_int (arbre_of_vect t4.vect);;
construire_tas t4;;
dessine_arbre string_of_int (arbre_of_vect t4.vect);;

(*==================================*)
(*   Tri de vecteur en tas          *)
(*==================================*)
(* === Question 8. --- Tri en tas d'un vecteur === *)

let tri_par_tas ordre v = A_ECRIRE "7 lignes"
;;
(* tri_par_tas : ('a -> 'a -> bool) -> 'a array -> unit = <fun> *)

let v1 = [|4;1;3;2;16;9;10;14;8;7|];;  tri_par_tas (fun u v -> u < v) v1;;
v1;;

let v2 = [|4;1;13;5;16;9;10;14;8;7;15;12|];; tri_par_tas (fun u v -> u > v) v2;;
v2;;

(*==================================*)
(*   Files de priorite              *)
(*==================================*)
type 'a prior == 'a tas
;;
let prior_new ordre n a = {vect = make_vect n a; taille = 0; ordre = ordre} : 'a prior
;;
(* prior_new : ('a -> 'a -> bool) -> int -> 'a -> 'a prior = <fun> *)

(* === Question 9. --- Tâche prioritaire === *)

let prior_take (fip : 'a prior) =                    (* extraction de la tête *)
  let lt = fip.taille -1 in
  if lt < 0 then failwith "erreur"
  else A_ECRIRE "5 lignes"
;;
(* prior_take : 'a prior -> 'a = <fun> *)

let t = { vect = [|4;1;3;2;16;9;15;14;8;7|]; taille = 10; ordre = prefix < };;
construire_tas t;;
dessine_arbre string_of_int (arbre_of_vect t.vect);;

prior_take t;;
dessine_arbre string_of_int (arbre_of_vect t.vect);;
```

```
(* === Question 10. --- Insertion d'une nouvelle tâche === *)
let prior_add (fip : 'a prior) c =
  let lt = fip.taille in
  if lt >= Array.length fip.vect then failwith "erreur:file pleine"
  else begin
              A_ECRIRE "8 lignes, avec une boucle"
       end
;;
(* prior_add : 'a prior -> 'a -> unit = <fun> *)

let t = { vect = [|4;1;3;2;16;9;10;14;8;7|]; taille = 0; ordre = (fun u v -> u < v};;
dessine_arbre string_of_int (arbre_of_vect t.vect);;
construire_tas t;;
dessine_arbre string_of_int (arbre_of_vect t.vect);;
prior_take t;;
prior_add t 6;;
dessine_arbre string_of_int (arbre_of_vect t.vect);;

(* === Question 11. --- Modification d'une tâche (remplacement par une autre) === *)
let prior_changeto (fip : 'a prior) i value =
  let lt = fip.taille in
  if fip.ordre fip.vect.(i) value
  then (* "augmente" *)      A_ECRIRE "6 lignes"
  else (* "diminue"  *)      A_ECRIRE "1 ou 2 lignes"
;;
(* prior_changeto : 'a prior -> int -> 'a -> unit = <fun> *)

let t = { vect = [|4;1;3;2;16;9;10;14;8;7|]; taille = 0; ordre = (fun u v > u < v};;
construire_tas t;;
dessine_arbre string_of_int (arbre_of_vect t.vect);;
prior_changeto t 4 17;;
dessine_arbre string_of_int (arbre_of_vect t.vect);;
prior_changeto t 0 7;;
dessine_arbre string_of_int (arbre_of_vect t.vect);;

(*==================================*)
(*   Dijkstra                       *)
(*==================================*)
let ma = Array.make_matrix 5 5 0;;                    (* exemple pour tests *)
    ma.(0).(1) <- 10; ma.(0).(3) <- 5;
    ma.(1).(2) <- 1;  ma.(1).(3) <- 2;
    ma.(2).(4) <- 4;
    ma.(3).(1) <- 3;  ma.(3).(2) <- 9;  ma.(3).(4) <- 2;
    ma.(4).(0) <- 7;  ma.(4).(2) <- 6;;
print_int_matrix ma;;

(* === Question 12. --- Algorithme de Dijkstra === *)
let dijkstra a s =
  let n = Array.length a in
  let d = Array.make n max_int and p = Array.make n (-1) in
  d.(s) <- 0;

  let fip = { vect = Array.make n (0,0); taille = n; ordre = (fun u v -> u > v} in
  for i = 0 to n-1 do fip.vect.(i) <- (d.(i),i) done;
  swap_vect fip.vect 0 s;

  while fip.taille > 0
  do                    A_ECRIRE "11 lignes"
  done;
  (d,p)
;;
(* dijkstra : int array array -> int -> int array * int array = <fun> *)

print_int_matrix ma;;
dijkstra ma 0;;
```

```
(* === Question 13. --- Description d'un plus court chemin de s à u === *)
let court_chemin p u = A_ECRIRE "4 lignes"
;;
(* court_chemin : int array -> int -> int list = <fun> *)

print_int_matrix ma;;
let d, p = dijkstra ma 0;;
court_chemin p 2;;

(*============================================*)
(* === FIN (TP Tas. Squelette de programme) === *)
(*============================================*)
```

$< \mathscr{F}\mathscr{I}\mathscr{N} >$ (TP Tas. Enoncé + squelette)

TP Tas. Corrigé (sans les exemples)

Arbre virtuel associé à un vecteur.

Question 1. Soit k le nombre de nœuds terminaux dans l'arbre binaire à n nœuds $\mathscr{A}$.

On considère l'arbre $\mathscr{A}'$ déduit de $\mathscr{A}$ par suppression des feuilles issues des nœuds terminaux et remplacement de ces nœuds terminaux par des feuilles.

- Si n est pair, il restera une feuille de l'ancien arbre et l'arbre $\mathscr{A}'$ aura $k+1$ feuilles
- Si n est impair, l'arbre $\mathscr{A}'$ aura k feuilles

D'autre part, comme l'arbre $\mathscr{A}'$ possède $n-k$ nœuds, il a $n-k+1$ feuilles.
Ainsi, si n est pair, on a $k+1 = n-k+1$ et si n est impair on a $k = n-k+1$
On en déduit que $k = \left\lfloor \frac{n+1}{2} \right\rfloor$.

Question 2.

Les lignes de l'arbre sont numérotées à partir de 1, et sur la ligne d'indice k, lorsqu'elle est complètement remplie, il y a 2^{k-1} nœuds, dont les indices, de gauche à droite, varient de $2^{k-1}-1$ à 2^k-2.
Le nœud $\mathscr{N}$ d'indice i étant le j-ème élément sur la k-ème ligne, on a $i = 2^{k-1}-2+j$.
Lorsque le nœud $\mathscr{N}$ a des fils, cela signifie que sa ligne est complètement remplie et les $j-1$ nœuds situés avant $\mathscr{N}$ sur la ligne k ont chacun deux fils, ce qui fait que, sur la ligne $k+1$, avant le fils gauche de $\mathscr{N}$, il y a $2*(j-1)$ nœuds, le premier étant d'indice 2^k-1 et le dernier étant d'indice $2^k-1+2*(j-1)-1$.
Ainsi le fils gauche de $\mathscr{N}$ est d'indice $2^k-1+2*(j-1)-1+1 = 2(2^{k-1}-2+j)+1 = 2i+1$

On a donc $G(i) = 2i+1$, d'où $D(i) = 2i+2$ et on en déduit que $P(i) = \left\lfloor \frac{i-1}{2} \right\rfloor$ (si $i > 0$).

Avec $i = 0$ (racine), la formule donnerait $P(0) = -1$, qui signifie (naturellement) "pas de père".

Question 3. Construction de l'arbre associé à un vecteur

```
let arbre_of_vect v =
  let n = Array.length v-1 in
  let rec creer = function
    | i when i > n -> F
    | i            -> N(creer (i*2+1), v.(i), creer (i*2+2))
  in creer 0
;;
(* arbre_of_vect : 'a array -> 'a arbre = <fun> *)
```

Question 4. Construction du vecteur associé à un arbre

```
open Queue;;
let it_large_arbre f a t =
  let b = ref a and q = Queue.create () in
  Queue.add t q;
  while Queue.length q > 0
  do match (Queue.take q) with
        F        -> ()
      | N(g,p,d) -> Queue.add g q; Queue.add d q ; b := f !b p;
  done;
  !b
;;
(* it_large_arbre : ('a -> 'b -> 'a) -> 'a -> 'b arbre -> 'a = <fun> *)

let list_large_of_arbre t = List.rev (it_large_arbre (fun x y -> y::x) [] t);;
(* list__large_of_arbre : 'a arbre -> 'a list = <fun> *)

let vect_of_arbre t = Array.of_list (list_large_of_arbre t);;
(* vect_of_arbre : 'a arbre -> 'a vect = <fun> *)
```

Question 5. Accès à l'élément d'indice i dans l'arbre.

Il faut rechercher, en remontant depuis le nœud d'indice i les pères successifs.

L'indice du père est $P(i) = \left\lfloor \frac{i-1}{2} \right\rfloor$ (si $i > 0$), donc $\begin{cases} \text{si } i \text{ est impair, le nœud est le fils gauche de son père,} \\ \text{si } i \text{ est pair, le nœud est le fils droit de son père.} \end{cases}$

En fait, le chemin à suivre est la queue de l'écriture binaire de $i+1$, poids fort en tête.

```
let chemin i =
  let u = ref (i+1) and l = ref [] in
  while !u > 1
  do l := (!u mod 2) ::!l; u := !u / 2 done;
  !l
;;
(* chemin : int -> int list = <fun> *)
```

```
let chemin i =
  let rec aux acc = function
    | 0 -> acc
    | n -> aux ((n mod 2) :: acc) (n/2)
  in List.tl (aux [] (i+1))
;;
(* chemin : int -> int list = <fun> *)
```

La complexité de `(chemin n)` est un $O(\log n)$.

```
let arbre_item t i =
  if (i < 0) then failwith "erreur";
  let l = chemin i in
  let rec aux t l = match (t,l) with  (* fun n'est pas apprécié par OCaml ? *)
    | (N(g,p,d)), []     -> p
    | (N(g,p,d)), (h::r) -> if h = 0 then aux g r else aux d r
    | _ ,  _             -> failwith "erreur"
  in aux t l
;;
(* arbre_item : 'a arbre -> int -> 'a = <fun> *)
```

Question 6. Intermédiaire de construction de tas total

```
type 'a tas = {vect: 'a array; mutable taille:int; ordre : 'a -> 'a -> bool }
;;
let swap_vect v i j = let u = v.(i) in v.(i) <- v.(j); v.(j) <- u
;;
(* swap_vect : 'a array -> int -> int -> unit = <fun> *)
```

```
let entasser t i =
  let m = ref 0 in
  let rec aux i =
    let g = 2*i+1 and d = 2*i+2 in
    if g < t.taille && (t.ordre t.vect.(i) t.vect.(g))   then m := g else m := i;
    if d < t.taille && (t.ordre t.vect.(!m) t.vect.(d))  then m := d;
    if !m <> i then begin swap_vect t.vect !m i; aux !m end
  in aux i
;;
(* entasser : 'a tas -> int -> unit = <fun> *)
```

Question 7. Construction d'un tas total

```
let construire_tas t =
  let n = Array.length t.vect in
  t.taille <- n;
  for i = n/2 downto 0 do entasser t i done;
;;
(* construire_tas : 'a tas -> unit = <fun> *)
```

Question 8. Tri en tas d'un vecteur

```
let tri_par_tas ordre v =
  let t = {vect = v; taille = 0; ordre = ordre} in
  construire_tas t;
  for k = Array.length t.vect -1 downto 1
  do swap_vect t.vect 0 k;
     t.taille <- k;
     entasser t 0
  done
;;
(* tri_par_tas : ('a -> 'a -> bool) -> 'a array -> unit = <fun> *)
```

Files de priorité.

```
type 'a prior = 'a tas
;;
let prior_new ordre n a =
   ( { vect = Array.make n a; taille = 0; ordre = ordre }: 'a prior )  (* () obligatoire *)
;;
(* prior_new : ('a -> 'a -> bool) -> int -> 'a -> 'a prior = <fun> *)
```

Question 9. Extraction de la tâche prioritaire

```
let prior_take (fip: 'a prior) =                    (* extraction de la tête *)
  let lt = fip.taille -1 in
  if lt < 0 then failwith "erreur:file vide";
  let m = fip.vect.(0) in
  fip.vect.(0) <- fip.vect.(lt);
  fip.taille <- lt;
  entasser fip 0;
  m
;;
(* prior_take : 'a prior -> 'a = <fun> *)
```

Question 10. Insertion d'une nouvelle tâche

```
let prior_add (fip: 'a prior) c =
  let lt = fip.taille in
  if lt >= Array.length fip.vect then failwith "erreur:file pleine";

  fip.taille <- lt + 1;
  let i = ref lt in
  while (!i > 0) && (fip.ordre fip.vect.((!i-1)/2)  c)
  do let u = (!i-1)/2 in
     fip.vect.(!i) <- fip.vect.(u);
     i := u
  done;
  fip.vect.(!i) <- c
;;
(* prior_add : 'a prior -> 'a -> unit = <fun> *)
```

Question 11. Modification de la priorité d'une tâche (ou remplacement d'une tâche par une autre)

```
let prior_changeto (fip: 'a prior) i valeur =
  if fip.ordre fip.vect.(i) valeur                                  (* "augmente" *)
  then begin let j = ref i in
             while (!j > 0) &&  (fip.ordre fip.vect.( (!j-1)/2 )  valeur)
             do let u = (!j-1)/2 in
                fip.vect.(!j) <- fip.vect.(u);
                j := u
             done;
             fip.vect.(!j) <- valeur
       end
  else if fip.ordre valeur fip.vect.(i)                             (* "diminue"  *)
       then begin fip.vect.(i) <- valeur; entasser fip i end
;;
(* prior_changeto : 'a prior -> int -> 'a -> unit = <fun> *)
```

Algorithme de Dijkstra.

Question 12. Algorithme de Dijkstra

```
let dijkstra a s =
  let n = Array.length a in
  let d = Array.make n max_int and p = Array.make n (-1) in
  d.(s) <- 0;

  let fip = { vect = Array.make n (0,0); taille = n; ordre = (fun u v -> u > v)} in
  for i = 0 to n-1 do fip.vect.(i) <- (d.(i),i) done;
  swap_vect fip.vect 0 s;

  while fip.taille > 0
  do let (du,u) = prior_take fip in
     for i = 0 to fip.taille -1
     do let (dv,v) = fip.vect.(i) in
        let duv = a.(u).(v) in
        if (duv > 0) && (du + duv < dv) then
        begin d.(v) <- du + duv;
              p.(v) <- u;
              prior_changeto fip i (d.(v),v)
        end
    done
  done;
  (d,p)
;;
(* dijkstra : int array array -> int -> int array * int array = <fun> *)
```

Question 13. Description d'un plus court chemin de s à u

```
let court_chemin p u =
  let f = ref u and x = ref [] in
  while !f >= 0 do x := (!f)::(!x); f := p.(!f) done;
  !x
;;
(* court_chemin : int array -> int -> int list = <fun> *)
```

$< \mathscr{FIN} >$ (TP Tas. Corrigé, sans les exemples)