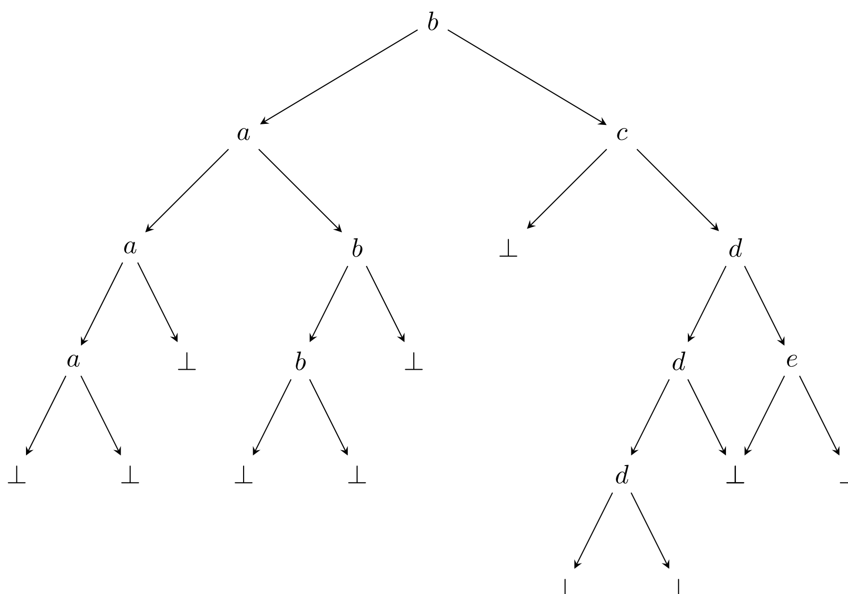
Exercices de révisions MP

Exercice 1 (Arbres binaires de recherche) :

Dans cet exercice, on autorise les doublons dans un arbre binaire de recherche et pour le cas d'égalité on choisira le sous-arbre gauche. On ne cherchera pas à équilibrer les arbres.

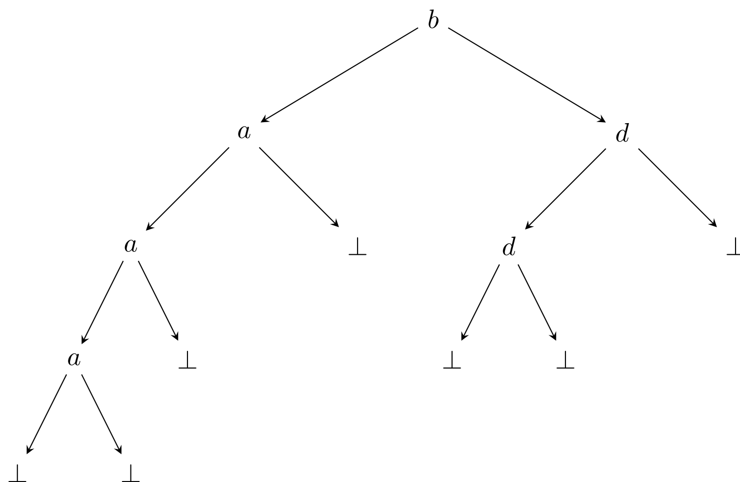
1. Un arbre binaire de recherche est un arbre binaire dont les clés sont à valeurs dans un ensemble totalement ordonné. De plus, pour chaque noeud de cet arbre, son étiquette x est strictement supérieure aux étiquettes dans son fils gauche et strictement inférieure aux étiquettes dans son fils droit (des variantes autorisent une inégalité large d'un des deux côtés).
2. Insérer successivement et une à une dans un arbre binaire de recherche initialement vide toutes les lettres du mot *bacddabdbae* en utilisant l'ordre alphabétique sur les lettres. Quelle est la hauteur de l'arbre ainsi obtenu ?



3. On procède par induction. On note $P(a)$ le parcours infixe de l'ABR a. Le parcours infixe d'un ABR vide est bien sûr trié. De plus, si a est un ABR de racine x et de fils gauche (respectivement droit) g (respectivement d) alors $P(a) = P(g) \cdot x \cdot P(d)$ (où $\cdot$ désigne la concaténation). Par hypothèse d'induction, $P(g)$ et $P(d)$ sont triés dans l'ordre croissant. Par définition d'un ABR, x est plus grand que les éléments apparaissant dans $P(g)$ et plus petit que ceux dans $P(d)$: $P(a)$ est donc trié dans l'ordre croissant.
4. On procède récursivement :
   - Le nombre d'occurrences d'une lettre u dans un arbre vide vaut 0.
   - Si l'ABR n'est pas vide, on compare u à sa racine x. Si $u < x$, on compte le nombre d'occurrences de u à gauche, si $u > x$, on compte le nombre d'occurrences de u à droite et si $u = x$, on compte le nombre d'occurrences de u à gauche et on lui ajoute 1.
   La complexité de cet algorithme est en $O(n)$ au pire cas où n est la taille de l'arbre (atteinte dans le cas du décompte d'une lettre qui étiquette tous les noeuds de l'arbre).
5. On commence par chercher une occurrence de l'élément s à supprimer en descendant soit à gauche soit à droite selon comment s se compare à la racine courante. Une fois s trouvé :
   - S'il n'a pas de fils, on peut le supprimer directement.
   - S'il en a un, on remplace l'arbre enraciné en s par ce fils.

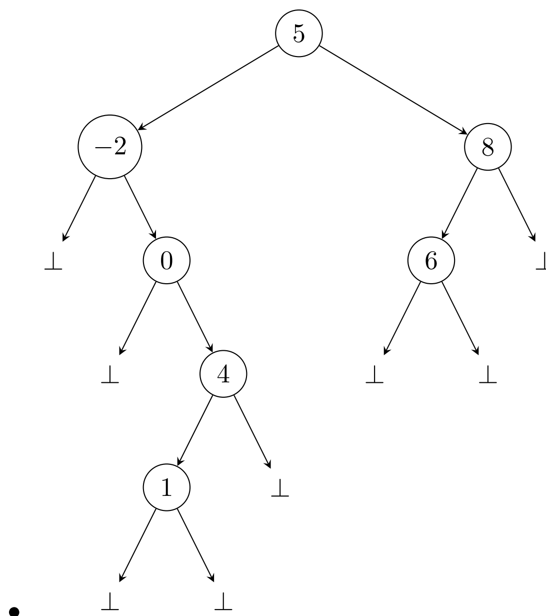
- S'il en a deux, on récupère le maximum m du fils gauche, on écrase s avec m puis on supprime m. Cet appel récursif aboutit à l'un des deux cas précédents car par définition le maximum d'un ABR se trouve en descendant systématiquement à droite dont m n'aura pas de fils droit.

Dans l'exemple précédent, on obtient en sortie l'arbre suivant :



Exercice 2 (Arbres AVL) :

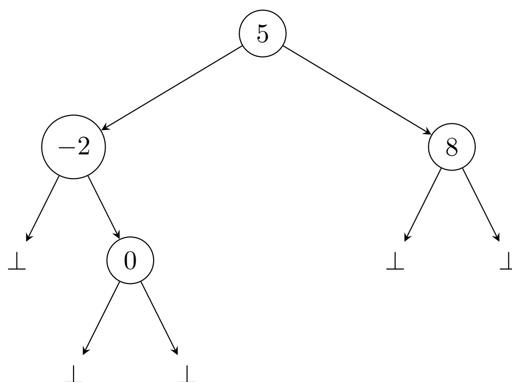
Question de cours :



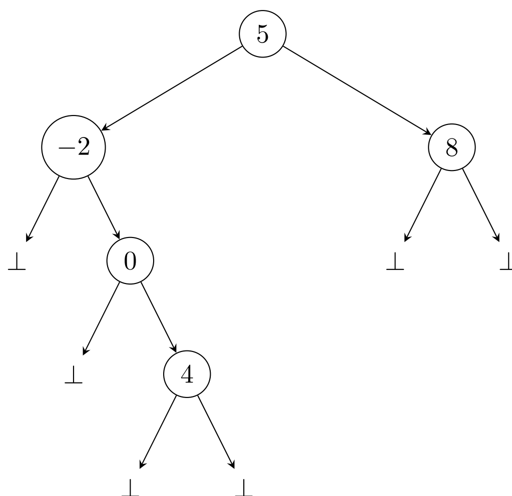
•

- Dans le pire des cas, un arbre binaire de recherche est en peigne, et on cherche à ajouter un élément au plus bas de l'arbre. L'ajout d'un élément demande alors de parcourir tout l'arbre binaire de recherche, et la complexité est alors en $O(n)$. Il en est de même pour la recherche d'un élément.

1. Cet arbre binaire de recherche de hauteur 2 est un AVL



Cet arbre binaire de recherche de hauteur 3 n'est pas un AVL. On a des problèmes d'équilibre avec les deux sous arbres du noeud 5, mais aussi avec les deux sous-arbres du noeud -2.



2. Il est facile de constater que :
   - $N_{-1} = 0$; $N_0 = 1$
   - si on se donne $N(g, x, d)$ un arbre AVL non-vide de hauteur h contenant un nombre minimum de noeuds, alors l'un des deux sous-arbres (g ou d) est de hauteur $h-1$ et l'autre est de hauteur $h-2$; les deux étant des AVL, on a alors $N_h = N_{h-1} + N_{h-2} + 1$
3. En substituant dans ce qui précède, on vérifie très facilement que la suite $(F_h)_{h\in\mathbb{N}}$ est la suite de Fibonacci. On peut alors retrouver sa valeur en s'appuyant sur le fait que c'est une suite récurrente linéaire d'ordre 2. On trouve, pour rappel :

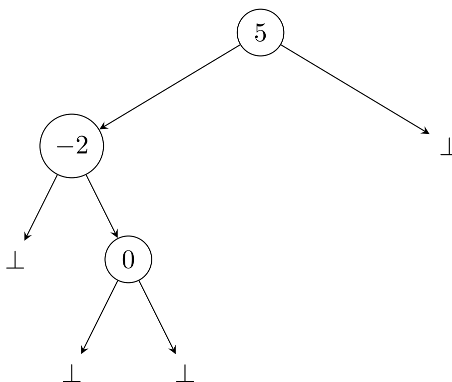
$$\forall h \in \mathbb{N}, F_h = \frac{1}{\sqrt{5}}(\varphi^h - \varphi'^h)$$

où φ est le nombre d'or et φ' son conjugué. En particulier, on a $\varphi' \approx -0,68$, il est donc facile d'obtenir que pour tout h, on a :

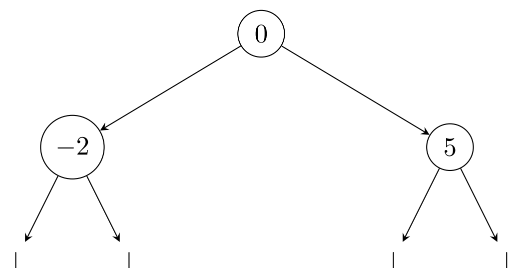
$$\begin{aligned} \frac{1}{\sqrt{5}}(\varphi^h - 1) &\leq F_h = N_{h-1} + 1 \\ \varphi^h &\leq \sqrt{5}(N_{h-1} + 1) + 1 \\ h &\leq \log_\varphi(\sqrt{5}(N_{h-1} + 1) + 1) \end{aligned}$$

On en déduit alors bien $h = O(\log(n))$.

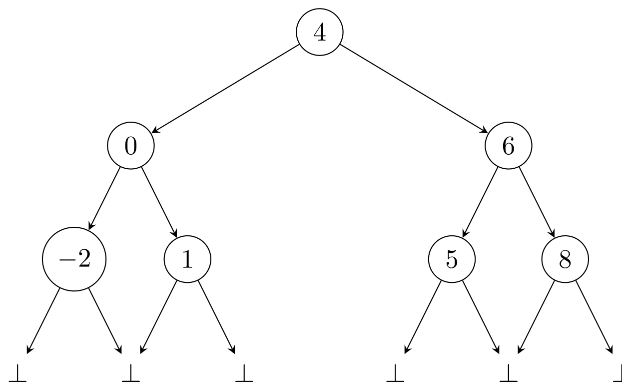
4. Dès l'ajout de 0, on a un arbre déséquilibré. Le sous-arbre gauche de 5 est de hauteur 1, et son sous-arbre droit est de hauteur -1.



On peut rééquilibrer simplement à l'aide d'une rotation gauche sur l'arbre de racine -2, puis une rotation droite sur l'arbre de racine 5. On obtient ce qui suit :



Ajoutons les autres noeuds ; à chaque ajout, il sera nécessaire de corriger au plus un déséquilibre dans l'arbre ; pour corriger chaque déséquilibre, on procède à une ou deux rotations. Voici l'arbre final.



5. Supposons qu'on a un déséquilibre après l'insertion d'un noeud. Soit $N(g, x, d)$ le noeud le plus profond auquel ce déséquilibre se produit. L'ajout de l'élément s'est nécessairement opéré dans le sous-arbre de racine x, et a produit un changement de hauteur : le sous-arbre de racine x a pour hauteur h après l'ajout, et $h - 1$ avant (s'il était déjà de hauteur h, il y avait déjà un déséquilibre, ce qui contredit le fait que T est un AVL).

   Par le déséquilibre en x, on a $|h(g) - h(d)| = 2$ (on sait que cet écart ne peut pas valoir plus ; sinon, il y avait déjà un déséquilibre avant). Il suffit ensuite de distinguer les cas (il est facile de comprendre ce qu'il se passe en dessinant) :

- supposons que $h(g) = h(d)+2$. On a alors $h(g) = h-1$ et $h(d) = h-3$. L'arbre g n'est clairement pas vide, notons alors $g = N(g_1, y, g_2)$. On doit avoir que l'un des deux sous-arbres g_1, g_2 est de hauteur $h-2$ (car $h(g) = h-1$), et l'autre est de hauteur $h-3$ (en effet, s'ils sont tous les deux hauteur $h-2$, il y avait déjà un déséquilibre avant l'insertion).

  Distinguons alors selon le cas :
  - si $h(g_1) = h-2$ et $h(g_2) = h-3$, alors il suffit de procéder à une rotation droite sur le noeud x. L'arbre de racine x est alors remplacé par $N(g_1, y, N(g_2, x, d))$, et d'après ce qui précède, il est équilibré et de hauteur $h-1$.
  - si $h(g_1) = h-3$ et $h(g_2) = h-2$, alors g_2 se décompose lui-même en deux sous-arbres : $g_2 = (g_2^a, z, g_2^b)$; l'un des deux est de hauteur $h-3$, l'autre éventuellement de hauteur $h-4$. On procède à une rotation gauche en y, puis une rotation droite en x. On obtient alors l'arbre $N(N(g_1, y, g_2^a), z, N(g_2^b, x, d))$; les deux sous-arbres de racines y et x sont équilibrés et de hauteur $h-2$ d'après ce qui précède, et donc l'arbre de racine z est équilibré et de hauteur $h-1$.
- si $h(d) = h(g)+2$, on rééquilibre l'arbre de manière totalement symétrique au cas précédent, en remplaçant les rotations gauche par des rotations droite et réciproquement.

Avec au plus deux rotations, on a ainsi remplacé l'arbre de racine x par un nouvel arbre équilibré, et de hauteur $h-1$. Or, comme cette position était occupée par un arbre de hauteur $h-1$ avant l'ajout et comme t était un AVL, on en déduit aisément qu'il n'y a pas de nouveau déséquilibre à corriger.

6. L'algorithme obtenu est alors en complexité $O(\log(n))$. En effet, il faut accéder à la bonne position où insérer le nouvel élément, ce qui se fait en $O(\log(n))$. Par la suite, chaque rotation s'effectue en temps constant, et on en effectue un nombre fini, d'où le résultat.

7.
```
let rec hauteur t =
        match t with
        | V -> -1
        | N(g,x,d) -> 1 + (max (hauteur g) (hauteur d))

let rotation_droite t =
        match t with
        | N(N(a,x,b),y,c) -> N(a,x,N(b,y,c))
        | _ -> t

let rotation_gauche t =
        match t with
        | N(a,x,N(b,y,c)) -> N(N(a,x,b),y,c)
        | _ -> t

let rec inserer_avl t m =
        match t with
        | V -> N(V,m,V)
        | N(l,p,r) ->
        (* on insère récursivement dans le bon sous-arbre *)
        let t' =
                if m < p then N(inserer_avl l m, p, r)
                    else N(l, p, inserer_avl r m) in

        (* on cherche à corriger un déséquilibre *)
                match t' with
                | N(g,x,d) when hauteur g > (hauteur d) + 1 -> (* fils gauche trop profond *)
                    begin
                      match g with
                            | N(g1,y,g2) when hauteur g1 > hauteur g2 -> rotation_gauche t'
                      | N(g1,y,g2) -> rotation_droite (N(rotation_gauche g, x, d))
                      | _ -> t'
                      end
```

```
        | N(g,x,d) when hauteur d > (hauteur g) + 1 -> (* fils droit trop profond *)
           begin
           match d with
           | N(d1,y,d2) when hauteur d1 < hauteur d2 -> rotation_droite t'
           | N(d1,y,d2) -> rotation_gauche (N(g, x, rotation_droite d))
           | _ -> t'
           end
      | _ -> t' (* aucun déséquilibre *)
```

Exercice 3 Minima locaux dans des arbres :

1. Les nœuds d'étiquettes $4, 0$ et 1 sont les trois minima locaux.
2. Un arbre est soit un arbre vide soit un nœud formé d'une étiquette et de deux sous-arbres. La hauteur est la profondeur maximale d'une feuille, c'est-à-dire la longueur maximale d'un chemin de la racine à une feuille. La hauteur de l'arbre (b) est 3. En particulier, la hauteur d'un arbre vide est -1 par convention.
3. L'arbre possède un nombre fini non vide d'étiquettes et donc une étiquette de valeur minimale, qui est un minimum global et donc local.
4. Si la racine de l'arbre est un minimum local on a trouvé notre minimum local. Sinon, un des deux fils est non vide, avec une étiquette à sa racine plus petite que celle de la racine de l'arbre. Un appel récursif permet d'obtenir un minimum local de ce sous-arbre, qui est également un minimum local de l'arbre (que ce soit la racine du sous-arbre ou un descendant strict). La complexité est linéaire en la hauteur de l'arbre.
5. On propose l'étiquetage à la figure (c) dans lequels les 5 minima locaux sont étiquetés par des entiers strictement négatifs.

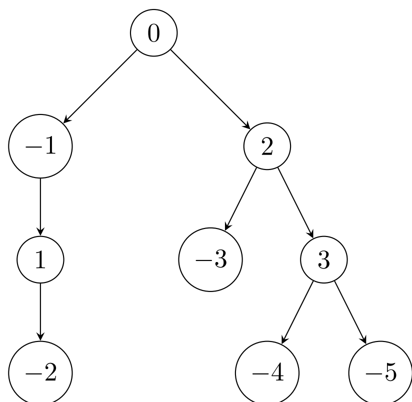
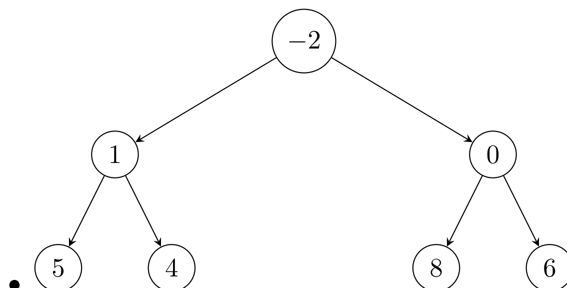


Figure 1: Arbre non-étiqueté (a) à gauche, et arbre étiqueté (b) à droite

6. On propose une approche récursive qui pour un arbre a en entrée calcule $m(a)$ le nombre maximal de nœuds qui peuvent être des minima locaux dans un étiquetage de a, ainsi que, en même temps, la quantité $m_{(a)}$ correspondant à cette même quantité mais en supposant de plus que la racine n'est pas un minimum local. Pour un arbre vide, ces deux valeurs valent 0. Pour un arbre a de filsgauche f_g et de fils droit f_d, on peut obtenir par appels récursifs les quantités $m(f_g)$, $m_{(f_g)}$, $m(f_d)$ et $m_{(f_d)}$. On a alors $m_{(a)=m(f_g)+m(f_d)}$ et $m(a) = \max\{m_{(a),1+m_{(f_g)+m_{(f_d)}\}}}$. La complexité est linéaire en la taille de l'arbre, chaque nœud est visité exactement une fois avec un nombre d'opérations constant.
7. Le résultat est vrai pour $n = 0$ et on peut donc supposer que $n \geq 1$. Considérons un étiquetage et notons X l'ensemble des nœuds qui sont des minima locaux et Y ceux qui ne le sont pas. On remarque que deux nœuds adjacents ne peuvent pas être tous les deux des minima locaux, puisque toutes les étiquettes sont deux à deux distinctes. Ainsi toute arête de l'arbre est extrémité d'au moins un nœud de Y et l'ensemble Y couvre donc toutes les arêtes. Comme chaque nœud de Y est incident à au plus 3 arêtes et qu'il y a exactement $n1$ arêtes dans l'arbre, il faut au moins $\frac{n-1}{3}$ nœuds pour couvrir toutes les arêtes, c'est-à-dire $|Y| \geq \frac{n1}{3}$. On a donc $|X| = n|Y|n\frac{n1}{3} = \frac{2n+1}{3}$, d'où le résultat.

Exercice 4 (Flux de données) :

Questions de cours :



- •
- Les deux opérations s'effectuent en $O(\log(n))$. Dans le premier cas, on insère d'abord un nouvel élément à la première position de feuille disponible (qu'on connaît par avance) ; dans le deuxième cas, on échange la racine avec la dernière feuille, puis on la supprime. Ensuite, dans tous les cas, il faut faire remonter ou faire descendre un noeud ; dans le pire des cas, on parcourt toute la hauteur de l'arbre, ce qui revient à une complexité en $O(h) = O(\log(n))$.

1. Si $k = 1$, il suffit de se donner une variable en dehors de la boucle `while`, et la fonction f va alors seulement mettre à jour cette variable si x est plus grand que sa valeur.

Algorithme 1 : A

```
max ← −∞;
while True do
    Recevoir x
    if x > max then
        max ← x
```

2. Pour $k = 2$, on peut s'en sortir avec 2 variables. Le 2-ème plus grand élément est stocké dans $max2$, et le plus grand dans $max1$.

Algorithme 2 : A

```
max1 ← −∞;
max2 ← −∞;
while True do
    Recevoir x
    if x > max1 then
        max2 ← max1
        max1 ← x
    else
        if x > max2 then
            max2 ← x
```

Pour k quelconque, il faudrait conserver dans un tableau les k plus grandes valeurs connues ; à chaque nouvel élément à insérer dans le tableau, on l'insère à la bonne position et on décale les éléments suivants ; le plus petit est alors éjecté du tableau.

Algorithme 3 : $A(k)$

$t \leftarrow$ tableau de $-\infty$ de taille k;
while True **do**
  Recevoir x
  if $x > t[k-1]$ **then**
    $t[k-1] \leftarrow x$
    $j = k - 1$
    while $j > 0 \&\& t[j] > t[j-1]$ **do**
      Echanger $t[j]$ et $t[j-1]$

3. On a un traitement en $O(k)$ dans le pire des cas, car il faut décaler tous les éléments du tableau. La complexité spatiale est en $O(k)$ également, pour stocker ce tableau de taille k.
4. On peut stocker dans un tas-min les éléments reçus, dont la taille serait au plus k. C'est contre-intuitif, mais on a bien besoin d'un tas-MIN qui contient à chaque instant les k plus GRANDS éléments (ou moins, si on n'en a pas encore reçu k).

   Tant que le tas ne contient pas encore k éléments, on rajoute les données reçues. Dès que le tas en contient k, il suffit pour chaque nouvelle donnée de l'enfiler dans le tas, et de tout de suite défiler un élément. On sait alors qu'après avoir enfilé l'élément, le tas contient $k+1$ éléments (les k plus grands connus jusqu'à cette itération, plus la nouvelle donnée) ; alors lorsqu'on défile, on éjecte le plus petit de ces $k+1$ éléments, et on vérifierait facilement avec une disjonction de cas qu'on a bien conservé les k plus grands éléments de toutes les données vues jusqu'ici (y compris la nouvelle donnée).

Algorithme 4 : $A(k)$

$t \leftarrow$ tas-min vide ;
$taille \leftarrow 0$;
while True **do**
  Recevoir x
  Enfiler(t, x)
  if $taille = k$ **then**
    $y \leftarrow$ Défiler(t)
  else
    $taille \leftarrow taille + 1$

5. Le coût du traitement à effectuer à chaque itération est alors en $O(\log(k))$: on effectue systématiquement une opération Enfiler sur un tas de taille au plus k, puis au plus une opération Défiler sur un tas de taille $k+1$.
6. Un tas-min peut être représenté facilement par un tableau, qui sera ici de taille $O(k)$.
7. On va utiliser une approche à base de deux tas-min[1] : un pour stocker les k plus grands éléments rencontrés, et un pour stocker les q suivants. On commence ainsi par stocker les k premiers éléments rencontrés dans le premier tas. Puis, à chaque nouvel élément :
   - on le compare à la racine du premier tas. S'il est plus grand, on l'insère et on en retire la racine. Sinon, on ne fait rien.
   - on a ensuite un nouveau noeud (l'élément ou la racine, selon le cas) à comparer à la racine du deuxième tas ; s'il est plus grand, on l'insère, et on en extrait le plus petit élément, qui est rejeté.

[1] On serait tenté de n'en utiliser qu'un seul, de taille $k+q$. Cette approche aurait la même complexité spatiale que celle que nous proposons ici, mais serait moins adaptée si on suppose que, avant de recevoir une nouvelle donnée, on souhaite effectuer des calculs à partir des données qui nous intéresse, donc ici, les q éléments qui sont les k-ème, $k+1$-ème, ... $k+q-1$-ème plus grands. Il est facile d'y accéder s'ils sont stockés dans un tas à part des éléments encore plus grands ; il est plus difficile d'y accéder s'ils sont tous mélangés dans un même tas.

Le traitement de chaque élément s'effectue alors en $O(\log(k) + \log(q))$. Dans tous les cas, la complexité spatiale est identique : $O(k + q)$. Cela se traduit par le pseudo-code suivant :

Algorithme 5 : $A(k, q)$

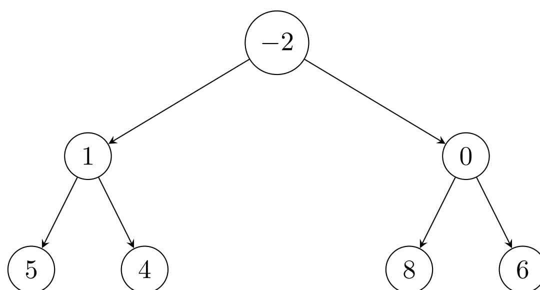
```
t1 ← tas-min vide ;
t2 ← tas-min vide ;
taille1 ← 0;
taille2 ← 0;
while True do
    Recevoir x
    Enfiler(t1, x)
    if taille1 = k then
        y ← Défiler(t1)
        Enfiler(t2, y)
        if taille2 = q then
            z ← Défiler(t2)
        else
            taille2 ← taille2 + 1
    else
        taille1 ← taille1 + 1
```

8. Si toutes les données sont connues à l'avance, la méthode naïve consiste à trier le tableau, on est alors en $O(n\log(n))$. C'est moins efficace que la méthode précédente.[2]
9. Si la taille du flux est connue par avance, on peut essayer d'estimer si k est proche de n ou non ; si c'est le cas, on peut remplacer le tas-min par un tas max, et remplacer k par $n - k$.

Exercice 5 (Multi-fusions) :

Questions de cours :

- Dessins quand j'aurai du temps.



- Les deux opérations s'effectuent en $O(\log(n))$. Dans le premier cas, on insère d'abord un nouvel élément à la première position de feuille disponible (qu'on connaît par avance) ; dans le deuxième cas, on échange la racine avec la dernière feuille, puis on la supprime. Ensuite, dans tous les cas, il faut faire remonter ou faire descendre un noeud ; dans le pire des cas, on parcourt toute la hauteur de l'arbre, ce qui revient à une complexité en $O(h) = O(\log(n))$.

[2] Il existe aussi un algorithme de "sélection rapide" permettant de sélectionner le p-ème plus petit élément d'un tableau non-trié ; il fonctionne sur un principe similaire au tri-rapide, avec une étape de séparation par rapport à un pivot, puis un examen de seulement l'un des deux sous-tableaux au lieu de 2. Sa complexité moyenne est en $O(n)$, mais sa complexité en pire cas est en $O(n^2)$. Il y a peut-être moyen de l'adapter pour chercher spécifiquement les éléments qui nous intéressent sans trier tout le tableau, avec une bonne complexité, mais je n'ai pas cherché comment.

1. C'est la fusion qu'on opère dans le tri-fusion :
   - si l'une des deux listes est vide, la fusion se résume à l'autre liste.
   - si les deux listes contiennent au moins un élément, on prend le premier élément de chaque liste.
     - le plus petit des deux devient la tête de la liste fusionnée.
     - le reste est constitué de la fusion du restant des deux listes (l'une des deux est identique, l'autre est celle dont on a ôté le minimum).
2. On pourrait le montrer par récurrence sur la somme de la longueur des deux listes. C'est immédiat si les deux listes sont vides. Sinon, alors comme les listes sont triées, la plus petite des deux tête de liste est bien plus petite que tous les autres éléments, et l'hypothèse de récurrence donne le reste du résultat. La complexité est linéaire en la somme des tailles des deux listes.
3.
```
let rec merge lst1 lst2 =
match lst1, lst2 with
|[], lst | lst, [] -> lst
|h1::tl1, h2::tl2 when h1 > h2 -> h2::(merge lst1 tl2)
|h1::tl1, h2::tl2 -> h1::(merge tl1 lst2)
```
4. On peut tout à fait l'adapter, mais cela devient lourd : il faut chercher le plus petit élément parmi k têtes de listes, ce qui a un coût linéaire en k ; et on répète cette opération autant de fois qu'il y a d'éléments à traiter, c'est-à-dire la somme des longueurs des listes.
5. On peut utiliser un tas-min. Il y a plusieurs solutions :
   - on peut ajouter dans le tas-min tout le contenu de chaque liste, puis l'extraire en les supprimant un à un. Si on note n la somme totale de la longueur des listes, on a un coût en $O(n\log(n))$.
   - une deuxième idée est d'utiliser un tas de taille k comportant un élément de chaque liste ; on peut les stocker sous forme de couple (x, i), où i est le numéro d'une liste, et x le plus petit élément actuellement restant dans cette liste.

     A chaque étape, on extrait un élément du tas, qui est le prochain plus petit élément de la liste fusionnée. Et on ajoute un nouvel élément de la même liste. La complexité totale est alors en $O(n\log(k))$.

   Voici un pseudo-code mettant en place cette deuxième approche :

   Algorithme 6 : $A(t, k)$
   ```
   tas ← tas-min vide ;
   for i = 0 à k − 1 do
       if t[i] ≠ [] then
           x ← tete(t[i]);
           t[i] ← queue(t[i]);
           Enfiler(t, x);
   return Aux(t, tas);
   ```

   Algorithme 7 : $Aux(t, tas)$
   ```
   if tas vide then
       return liste vide;
   else
       (x, i) ← Defiler(tas);
       if t[i] ≠ [] then
           y ← tete(t[i]);
           t[i] ← queue(t[i]);
           Enfiler(t, y);
       return x :: (Aux(t, tas))
   ```

   Alternativement, si on a peur de déborder de la pile, on peut construire la liste à l'envers, dans l'ordre décroissant, puis renvoyer le miroir de la liste.
6. Voir question précédente.
7. La deuxième méthode de la question précédente peut directement être utilisée pour résoudre ce problème.

Algorithme 8 : $A(t,k)$

$tas \leftarrow$ tas-min vide ;
for $i = 0$ à $k-1$ **do**
  $x \leftarrow Recevoir(t[i])$;
  if $x \neq EOF$ **then**
    Enfiler(t,x);
return $Aux(t, tas)$;

Algorithme 9 : $Aux(t, tas)$

if tas vide **then**
  return liste vide;
else
  $(x,i) \leftarrow Defiler(tas)$;
  $y \leftarrow Recevoir(t[i])$;
  if $y \neq EOF$ **then**
    Enfiler(t,y);
  return $x :: (Aux(t, tas))$

Alternativement, si on a peur de déborder de la pile, on peut construire la liste à l'envers, dans l'ordre décroissant, puis renvoyer le miroir de la liste.

8.
```
let merge streams_array =
        let h = heap_create () in
        for i = 0 to (Array.length streams_array) - 1 do
        let x = stream_pop streams_array.(i) in
        if x <> eof then
                heap_insert (stream_pop streams_array.(i), i) h
        done;
        while (heap_size h > 0) do
                let min_e, i = heap_extract h in
                Printf.printf "%d; " min_e;
        let y = stream_pop streams_array.(i) in
        if y <> eof then
                heap_insert (stream_pop streams_array.(i)) h
        done
```

Exercice 6 :

1.
$$\dfrac{\dfrac{\dfrac{\dfrac{}{A, (A \to B) \vdash A}\text{ax} \quad \dfrac{}{A, (A \to B) \vdash (A \to B)}\text{ax}}{A, (A \to B) \vdash B}\to_e}{A \vdash (A \to B) \to B}\to_i}{\vdash A \to (A \to B) \to B}\to_i$$

2.
$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{((A \vee B) \to C), A \vdash Z}\text{ax}}{((A \vee B) \to C), A \vdash (A \vee B)}\vee_g^i \quad \dfrac{}{((A \vee B) \to C), A \vdash ((A \vee B) \to C)}\text{ax}}{((A \vee B) \to C), A \vdash C}\to_e}{((A \vee B) \to C) \vdash (A \to C)}\to_i \quad \dfrac{\dots}{((A \vee B) \to C) \vdash (B \to C)}}{((A \vee B) \to C) \vdash (A \to C) \wedge (B \to C)}\wedge_i}{\vdash ((A \vee B) \to C) \to (A \to C) \wedge (B \to C)}\to_i$$

Exercice 7 :

1.
$$\dfrac{\dfrac{\dfrac{\dfrac{}{A, \neg A \vdash A}\text{ax} \quad \dfrac{}{A, \neg A \vdash \neg A}\text{ax}}{A, \neg A \vdash \bot}\neg_e}{\neg A \vdash A \to \bot}\to_i}{\vdash \neg A \to (A \to \bot)}\to_i$$

2.
$$\dfrac{\dfrac{\dfrac{\dfrac{}{A, A \to \bot \vdash A}\,\text{ax} \qquad \dfrac{}{A, A \to \bot \vdash A \to \bot}\,\text{ax}}{A, A \to \bot \vdash \bot}\,\to_e}{A \to \bot \vdash \neg A}\,\neg_i}{\vdash (A \to \bot) \to \neg A}\,\to_i$$

3. Trivial, on applique $\wedge_i$ à ce qui précède.
4. On pourrait dresser une table de vérité, mais on peut aussi raisonner plus simplement : soit v une valuation.
   - si $v(A) = vrai$, alors elle satisfait immédiatement P, qui est une formule de la forme $Q \to A$.
   - si $v(A) = faux$, alors l'implication $A \to B$ s'évalue donc à vrai quelle que soit la valeur de B. Puis $(A \to B) \to A$ s'évalue à faux, et on a le résultat.

5.
$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, A, \neg B \vdash \neg A}\,\text{ax} \qquad \dfrac{}{\Gamma, A, \neg B \vdash A}\,\text{ax}}{\Gamma, A, \neg B \vdash \bot}\,\neg_e}{\Gamma, A \vdash B}\,\bot}{\Gamma \vdash (A \to B)}\,\to_i \qquad \dfrac{}{\Gamma \vdash (A \to B) \to A}\,\text{ax}}{\Gamma \vdash A}\,\to_e \qquad \dfrac{}{\Gamma \vdash \neg A}\,\text{ax}}{\Gamma = \{(A \to B) \to A, \neg A\} \vdash \bot}\,\neg_e}{(A \to B) \to A \vdash A}\,\bot}{\vdash P}\,\to_i$$