Table of Contents

TP 3 - Programmation pour la préparation à l'agrégation maths option info

- En OCaml.

In [1]:
```
let print = Printf.printf;;
Sys.command "ocaml -version";;
```

Out[1]:
```
val print : ('a, out_channel, unit) format → 'a = <fun>

The OCaml toplevel, version 4.04.2

- : int = 0
```

Arbres binaires de recherche

Exercice 1 : ABR

Variante non polymorphe et sans utilisation d'enregistrement pour nommer les champs :

In [2]:
```
type abr0 =
  | Leaf0
  | Node0 of (int * string * abr0 * abr0)
;;
```

Out[2]:
```
type abr0 = Leaf0 | Node0 of (int * string * abr0 * abr0)
```

Mais on préfère la variant polymorphe, qui permettra une syntaxe plus concise :

In [3]:
```
type 'a abr =
   | Leaf
   | Node of 'a anode

and 'b anode = {
   key   : int;
   value : 'b;
   left  : 'b abr; (* pour toute clé [k] dans [left],  [k] < [key] *)
   right : 'b abr  (* pour toute clé [k] dans [right], [key] < [k] *)
}
;;
```

Out[3]:
```
type 'a abr = Leaf | Node of 'a anode
and 'b anode = { key : int; value : 'b; left : 'b abr; right : 'b abr; }
```

Compter les clés est facile :

In [4]:
```
let rec nb_keys (a : 'a abr) : int =
   match a with
   | Leaf -> 0
   | Node n -> 1 + nb_keys n.left + nb_keys n.right
   (* | Node (key, value, left, right) -> 1 + nb_keys left + nb_keys right) *)
;;
```

Out[4]:
```
val nb_keys : 'a abr → int = <fun>
```

Deux exemples :

In [5]:
```
let a1 = Node { key=1; value="un"; left=Leaf; right=Leaf } ;;
(* let a1 = Node (1,"un",Leaf,Leaf) *)
```

Out[5]:
```
val a1 : string abr = Node {key = 1; value = "un"; left = Leaf; right = Leaf}
```

In [6]:
```
let a2 = Node { key=2; value="deux"; left=a1; right=Leaf } ;;
```

Out[6]:
```
val a2 : string abr =
  Node
   {key = 2; value = "deux";
    left = Node {key = 1; value = "un"; left = Leaf; right = Leaf};
    right = Leaf}
```

Exercice 2 : `trouve`

Avec un type de retour `'a option`, qui renvoie `None` si rien n'a été trouvé, ou `Some a` si la valeur `a` est trouvée.

In [7]:
```
let rec trouve (a : 'a abr) (k : int) : 'a option =
  match a with
  | Leaf -> None
  | Node n when k = n.key -> Some n.value
  (* sinon on cherche à gauche ou à droite *)
  | Node n when k < n.key -> trouve n.left k
  | Node n -> trouve n.right k
;;
```

Out[7]:
```
val trouve : 'a abr → int → 'a option = <fun>
```

In [8]:
```
trouve a2 1;;
trouve a2 3;;
```

Out[8]:
```
- : string option = Some "un"
```

Out[8]:
```
- : string option = None
```

Avec une exception :

In [9]:
```
let rec trouve2 (a : 'a abr) (k : int) : 'a =
  match a with
  | Leaf -> failwith "Key not found"
  | Node n when k = n.key -> n.value
  (* sinon on cherche à gauche ou à droite *)
  | Node n when k < n.key -> trouve2 n.left k
  | Node n -> trouve2 n.right k
;;
```

Out[9]:
```
val trouve2 : 'a abr → int → 'a = <fun>
```

In [10]:
```
trouve2 a2 1;;
trouve2 a2 3;;
```

Out[10]:
```
- : string = "un"

Exception: Failure "Key not found".
Raised at file "pervasives.ml", line 32, characters 22-33
Called from file "toplevel/toploop.ml", line 180, characters 17-56
```

Exercice 3 : `insertion`

In [11]:
```
let rec insertion (a : 'a abr) (k : int) (v : 'a) : 'a abr =
  match a with
  | Leaf ->
    Node { key=k; value=v; left=Leaf; right=Leaf }
  | Node n when k=n.key ->
    Node { n with value = v; key = k }
  | Node n when k < n.key ->
    Node { n with left  = insertion n.left  k v }
  | Node n ->
    Node { n with right = insertion n.right k v }
;;
```

Out[11]:
```
val insertion : 'a abr → int → 'a → 'a abr = <fun>
```

Quelques tests :

In [12]:
```
trouve (insertion (insertion Leaf 2 "deux") 1 "un") 1;;
trouve (insertion (insertion Leaf 2 "deux") 1 "un") 2;;
trouve (insertion (insertion Leaf 2 "deux") 1 "un") 3;;
```

Out[12]:
```
- : string option = Some "un"
```

Out[12]:
```
- : string option = Some "deux"
```

Out[12]:
```
- : string option = None
```

Exercice 4 : `suppression`

`minimum a` renvoie le couple `(key, value)` de l'arbre `a` avec `key` minimal dans `a`. Lance une exception si `a` est vide.

In [13]:
```
let rec minimum (a: 'a abr) : int * 'a =
  match a with
  | Leaf -> failwith "empty tree"
  | Node n when n.left = Leaf -> (n.key, n.value)
  | Node n -> minimum n.left
;;
```

Out[13]:
```
val minimum : 'a abr → int * 'a = <fun>
```

In [14]:
```
minimum (insertion (insertion Leaf 2 "deux") 1 "un");;
```

Out[14]:
```
- : int * string = (1, "un")
```

La suppression se fait dans le cas où la clé `k` est trouvée :

In [17]:
```
let rec suppression (a: 'a abr) (k:int) : 'a abr =
  match a with
  | Leaf -> Leaf   (* rien a supprimer *)
  | Node n when k = n.key ->  (* trouvé ! *)
    if n.right = Leaf
    then n.left
    else
      let (k_min, v) = minimum n.right in
      Node { key = k_min; value = v; left = n.left; right = suppression n.right k_min }
  | Node n when k < n.key ->   (* à chercher à gauche *)
    Node { n with left  = suppression n.left  k }
  | Node n -> (* à chercher à droite *)
    Node { n with right = suppression n.right k }
;;
```

Out[17]:
```
val suppression : 'a abr → int → 'a abr = <fun>
```

In [18]:
```
trouve (suppression (insertion (insertion Leaf 2 "deux") 1 "un") 1) 1 ;;

trouve (suppression (insertion (insertion Leaf 2 "deux") 1 "un") 1) 2 ;;
```

Out[18]:
```
- : string option = None
```

Out[18]:
```
- : string option = Some "deux"
```

Exercice 5 : `fusion`

`decoupe a k` sépare l'arbre `a` en deux arbres `(a1, a2)` tels que l'union des clés-valeurs de `a1` et `a2` est égale à l'ensemble des clés-valeurs de `a` (privé de l'association liée à `k` si elle était présente dans `a`).

- Les clés de `a1` sont `<` à `k`.
- Les clés de `a2` sont `>` à `k`.

In [19]:
```
(* [decoupe a k] sépare l'arbre [a] en deux arbres [(a1, a2)]
   tels que l'union des clés-valeurs de [a1] et [a2] est égale à
   l'ensemble des clés-valeurs de [a] (privé de l'association
   liée à [k] si elle était présente dans [a]).
   Les clés de [a1] sont < à [k].
   Les clés de [a2] sont > à [k].
*)

let rec decoupe (a : 'a abr) (k : int) : ('a abr) * ('a abr) =
   match a with
   | Leaf -> (Leaf, Leaf)
   | Node n when k = n.key -> (n.left, n.right)
   | Node n when k < n.key ->
      let (left1, left2) = decoupe n.left k in
      (left1, Node { n with left = left2 })
   | Node n ->
      let (right1, right2) = decoupe n.right k in
      (Node { n with right = right1 }, right2)
;;
```

Out[19]:
```
val decoupe : 'a abr → int → 'a abr * 'a abr = <fun>
```

In [20]:
```
(* si une clé est présente dans les deux arbres, nous gardons celle de [a1] *)
let rec fusion (a1 : 'a abr) (a2 : 'a abr) : 'a abr =
   match a1 with
   | Leaf -> a2
   | Node n ->
      let (left2, right2) = decoupe a2 n.key in
      Node { n with left = fusion n.left left2; right = fusion n.right right2 }
;;
```

Out[20]:
```
val fusion : 'a abr → 'a abr → 'a abr = <fun>
```

In [22]:
```
let a1 = insertion (insertion Leaf 2 "deux") 1 "un" ;;

let a2 = insertion (insertion Leaf 2 "two") 3 "trois" ;;

trouve (fusion a1 a2) 1;;
trouve (fusion a1 a2) 2;;
trouve (fusion a1 a2) 3;;
trouve (fusion a1 a2) 4;;
```

Out[22]:
```
val a1 : string abr =
  Node
   {key = 2; value = "deux";
    left = Node {key = 1; value = "un"; left = Leaf; right = Leaf};
    right = Leaf}
```

Out[22]:
```
val a2 : string abr =
  Node
   {key = 2; value = "two"; left = Leaf;
    right = Node {key = 3; value = "trois"; left = Leaf; right = Leaf}}
```

Out[22]:
```
- : string option = Some "un"
```

Out[22]:
```
- : string option = Some "deux"
```

Out[22]:
```
- : string option = Some "trois"
```

Out[22]:
```
- : string option = None
```

Exercice 6 : avantages et les inconvénients des ABR

Discussions durant la séance...

Tas binaire min (ou max)

Solution concise, à adapter

> Reference: Chris Okasaki, *"Purely Functional Data Structures"*.

In [23]:
```
type 'a heap = E | T of int * 'a * ('a heap) * ('a heap)
```

Out[23]:
```
type 'a heap = E | T of int * 'a * 'a heap * 'a heap
```

In [24]:
```
let rank : 'a heap -> int = function
  | E -> 0
  | T (r, _, _, _) -> r
;;
```

Out[24]:
```
val rank : 'a heap → int = <fun>
```

La première primitive est la création d'un tas avec la clé `x`, et deux sous-tas `a` et `b`. Le rang est minimisé.

In [25]:
```
let make (x : 'a) (a : 'a heap) (b : 'a heap) =
  let ra = rank a
  and rb = rank b in
  if ra >= rb then
    T (rb + 1, x, a, b)
  else
    T (ra + 1, x, b, a)
;;
```

Out[25]:
```
val make : 'a → 'a heap → 'a heap → 'a heap = <fun>
```

On peut vérifier si un tas est vide, ou créer le tas vide.

In [26]:
```
let empty : 'a heap = E
;;
```

Out[26]:
```
val empty : 'a heap = E
```

In [27]:
```
let is_empty : 'a heap -> bool = function
  | E -> true
  | T _ -> false
;;
```

Out[27]:
```
val is_empty : 'a heap → bool = <fun>
```

La fusion est assez naturelle : on procède par récurrence, en joignant deux tas et en continuant la fusion pour les tas plus petits.
On gare la plus petite clé à la racine, pour conserver la propriété *tournoi*.

In [28]:
```
let rec merge (h1 : 'a heap) (h2 : 'a heap) : 'a heap =
  match h1, h2 with
  | E, h | h, E ->
    h
  | T (_, x, a1, b1), T (_, y, a2, b2) ->
    if x <= y then
      make x a1 (merge b1 h2)
    else
      make y a2 (merge h1 b2)
;;
```

Out[28]:
```
val merge : 'a heap → 'a heap → 'a heap = <fun>
```

L'insertion correspond à la fusion d'un tas avec une seule clé et du tas courant :

In [29]:
```
let insert (x : 'a) (h : 'a heap) : 'a heap =
  merge (T (1, x, E, E)) h
;;
```

Out[29]:
```
val insert : 'a → 'a heap → 'a heap = <fun>
```

La lecture de la plus petite clé est triviale :

In [30]:
```
exception Empty;;

let mini : 'a heap -> 'a = function
  | E -> raise Empty
  | T (_, x, _, _) -> x
;;
```

Out[30]:
```
exception Empty
```

Out[30]:
```
val mini : 'a heap → 'a = <fun>
```

Et l'extraction n'est pas compliquée : il suffit de fusionner les deux sous-tas, ce qui va produire un tas tournoi avec les clés restantes.

In [31]:
```
let extract_min : 'a heap -> ('a * 'a heap) = function
  | E -> raise Empty
  | T (_, x, a, b) -> x, merge a b
;;
```

Out[31]:
```
val extract_min : 'a heap → 'a * 'a heap = <fun>
```

Et maintenant pour le tri par tas :

1. On crée un tas vide,
2. Dans lequel on insère les valeurs du tableau à trier, une par une,
3. Puis on déconstruit le tas en extrayant le minimum, un par un, et en les stockant dans un tableau,
4. Le tableau obtenu est trié dans l'ordre croissant.

In [32]:
```
let tripartas (a : 'a array) : 'a array =
  let n = Array.length a in
  let tas = ref empty in
  for i = 0 to n - 1 do
    tas := insert a.(i) !tas
  done;
  let a2 = Array.make n (-1) in
  for i = 0 to n - 1 do
    let m, t = extract_min !tas in
    a2.(i) <- m;
    tas := t;
  done;
  a2
;;
```

Out[32]:
```
val tripartas : int array → int array = <fun>
```

Complexité :

1. L'étape 1. est en $\mathcal{O}(1)$,
2. L'étape 2. est en $\mathcal{O}(\log n)$ pour chacune des n valeurs,
3. L'étape 3. est aussi en $\mathcal{O}(\log n)$ pour chacune des n valeurs,

$\implies$ L'algorithme de tri par tas est en $\mathcal{O}(n \log n)$ en temps et en $\mathcal{O}(n)$ en mémoire externe.

In [33]:
```
tripartas [| 10; 3; 4; 1; 2; 7; 8; 5; 9; 6 |] ;;
```

Out[33]:
```
- : int array = [|1; 2; 3; 4; 5; 6; 7; 8; 9; 10|]
```

Exercice 7 : arbre tournoi

On va utiliser un tableau de taille n pour représenter en place les n éléments du tas min.

La référence pour cette implémentation vient du Cormen, des éléments sont aussi dans Beauquier & Bernstel, et sur Internet sur la page Wikipédia (https://fr.wikipedia.org/wiki/Tas_binaire) des tas binaires.

- Le tableau, `a` contiendra `-1` pour un élément non utilisé.
- On doit retenir le nombre `n` d'éléments dans l'arbre, qui peut être modifié.

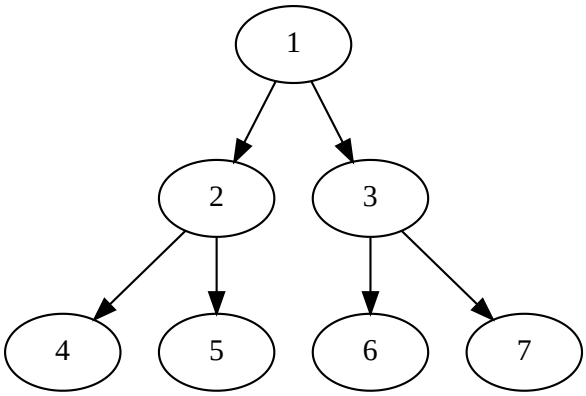
In [1]:
```
type arbre = int array;;

type arbre_tournoi = {
  mutable n : int;
  a : arbre
};;
```

Out[1]:
```
type arbre = int array
```

Out[1]:
```
type arbre_tournoi = { mutable n : int; a : arbre; }
```

Par exemple, l'arbre suivant s'écrit comme suit :



In [2]:
```
let arbre_test = {
    n = 7;
    a = [|1; 2; 3; 4; 5; 6; 7|]
}

let arbre_test2 = {
    n = 6;
    a = [|2; 1; 3; 4; 5; 6; -1; -1|]
}
```

Out[2]:
```
val arbre_test : arbre_tournoi = {n = 7; a = [|1; 2; 3; 4; 5; 6; 7|]}
```

Out[2]:
```
val arbre_test2 : arbre_tournoi = {n = 6; a = [|2; 1; 3; 4; 5; 6; -1; -1|]}
```

In [3]:
```
let capacite (an : arbre_tournoi) : int =
    Array.length an.a
;;

let nb_element (an : arbre_tournoi) : int =
    let n = an.n
    and m = Array.length an.a in
    assert (n <= m);
    n
;;
```

Out[3]:
```
val capacite : arbre_tournoi → int = <fun>
```

Out[3]:
```
val nb_element : arbre_tournoi → int = <fun>
```

In [4]:
```
capacite arbre_test;;
nb_element arbre_test;;

capacite arbre_test2;;
nb_element arbre_test2;;
```

```
- : int = 7
```

```
- : int = 7
```

```
- : int = 8
```

```
- : int = 6
```

In [5]:
```
let a_racine (an : arbre_tournoi) : bool =
    an.n > 0
;;
```

```
val a_racine : arbre_tournoi → bool = <fun>
```

In [6]:
```
let est_racine = (=) 0 ;;
```

```
val est_racine : int → bool = <fun>
```

In [7]:
```
let racine (an : arbre_tournoi) : int * int =
    if 0 >= an.n then failwith "Pas de racine";
    (0, an.a.(0))
;;
```

```
val racine : arbre_tournoi → int * int = <fun>
```

In [8]:
```
racine arbre_test;;
```

```
- : int * int = (0, 1)
```

In [9]:
```
let a_noeud (an : arbre_tournoi) (i : int) : bool =
    an.n > i
;;
```

```
val a_noeud : arbre_tournoi → int → bool = <fun>
```

In [10]:
```
let noeud (an : arbre_tournoi) (i : int) : int * int =
    if i >= an.n then failwith (Printf.sprintf "Pas de noeud i = %i and an.n = %i" i an.n);
    (i, an.a.(i))
;;
```

```
val noeud : arbre_tournoi → int → int * int = <fun>
```

In [11]:
```
let valeur (an : arbre_tournoi) (i : int) : int =
    snd (noeud an i)
;;
```

```
val valeur : arbre_tournoi → int → int = <fun>
```

In [12]:
```
noeud arbre_test 0;;
```

```
- : int * int = (0, 1)
```

In [13]:
```
let a_gauche (an : arbre_tournoi) (i : int) : bool =
    an.n > 2*i + 1
;;
```

```
val a_gauche : arbre_tournoi → int → bool = <fun>
```

In [14]:
```
let gauche (an : arbre_tournoi) (i : int) : int * int =
    if 2*i + 1 >= an.n then failwith (Printf.sprintf "Pas de fils gauche i = %i, 2*i+1 = %i and an.n = %i" i (2*i+1) an.n);
    (2*i + 1, an.a.(2*i + 1))
;;
```

```
val gauche : arbre_tournoi → int → int * int = <fun>
```

In [15]:
```
gauche arbre_test 0;;
```

Out[15]:
```
- : int * int = (1, 2)
```

In [16]:
```
let a_droite (an : arbre_tournoi) (i : int) : bool =
  an.n > 2*i + 2
;;
```

Out[16]:
```
val a_droite : arbre_tournoi → int → bool = <fun>
```

In [17]:
```
let droite (an : arbre_tournoi) (i : int) : int * int =
  if 2*i + 2 >= an.n then failwith (Printf.sprintf "Pas de fils droit i = %i, 2i+2=%i and an.n = %i" i (2*i+2) an.n);
  (2*i + 2, an.a.(2*i + 2))
;;
```

Out[17]:
```
val droite : arbre_tournoi → int → int * int = <fun>
```

Une et deux descentes à droite, par exemple :

In [18]:
```
droite arbre_test 0;;
```

Out[18]:
```
- : int * int = (2, 3)
```

In [19]:
```
let i, _ = droite arbre_test 0 in
droite arbre_test i;;
```

Out[19]:
```
- : int * int = (6, 7)
```

On parcourt les sous-arbres pour trouver le minimum :

In [20]:
```
let rec min_sous_arbre (an : arbre_tournoi) (i : int) : int =
  match (a_gauche an i, a_droite an i) with
  | (false, false) -> max_int
  | (true, false) ->
    let g, vg = gauche an i in
    min vg (min_sous_arbre an g)
  | (false, true) ->
    let d, vd = droite an i in
    min vd (min_sous_arbre an d)
  | (true, true) ->
    let g, vg = gauche an i in
    let d, vd = droite an i in
    min (min vg vd) (min (min_sous_arbre an g) (min_sous_arbre an d))
;;
```

Out[20]:
```
val min_sous_arbre : arbre_tournoi → int → int = <fun>
```

In [21]:
```
arbre_test;;
min_sous_arbre arbre_test 0;;
```

Out[21]:
```
- : arbre_tournoi = {n = 7; a = [|1; 2; 3; 4; 5; 6; 7|]}
```

Out[21]:
```
- : int = 2
```

In [22]:
```
let est_tournoi (an : arbre_tournoi) : bool =
    let rec depuis (i : int) : bool =
        (* cet arbre *)
        let _, vr = noeud an i in
        let min_v = min_sous_arbre an i in
        let res = ref (vr < min_v) in
        (* sous-arbres *)
        if !res && a_gauche an i then
            res := !res && depuis (fst (gauche an i));
        if !res && a_droite an i then
            res := !res && depuis (fst (droite an i));
        !res
    in
    depuis 0
;;
```

Out[22]:
```
val est_tournoi : arbre_tournoi → bool = <fun>
```

In [23]:
```
est_tournoi arbre_test;;
```

Out[23]:
```
- : bool = true
```

In [24]:
```
arbre_test2;;
est_tournoi arbre_test2;;
```

Out[24]:
```
- : arbre_tournoi = {n = 6; a = [|2; 1; 3; 4; 5; 6; -1; -1|]}
```

Out[24]:
```
- : bool = false
```

Exercice 8 : `parent`, `fils_gauche` et `fils_droit`

In [25]:
```
let parent (an : arbre_tournoi) (i : int) =
    noeud an ((i - 1) / 2)
;;

let fils_gauche = gauche;;
let fils_droite = droite;;
```

Out[25]:
```
val parent : arbre_tournoi → int → int * int = <fun>
```

Out[25]:
```
val fils_gauche : arbre_tournoi → int → int * int = <fun>
```

Out[25]:
```
val fils_droite : arbre_tournoi → int → int * int = <fun>
```

In [26]:
```
arbre_test;;

noeud arbre_test 1;;
parent arbre_test 1;;
noeud arbre_test 2;;
parent arbre_test 2;;
```

Out[26]:
```
- : arbre_tournoi = {n = 7; a = [|1; 2; 3; 4; 5; 6; 7|]}
```

Out[26]:
```
- : int * int = (1, 2)
```

Out[26]:
```
- : int * int = (0, 1)
```

Out[26]:
```
- : int * int = (2, 3)
```

Out[26]:
```
- : int * int = (0, 1)
```

In [27]:
```
noeud arbre_test 4;;
parent arbre_test 4;;
gauche arbre_test 1;;
droite arbre_test 1;; (* 4 *)

noeud arbre_test 5;;
parent arbre_test 5;;
gauche arbre_test 2;; (* 5 *)
droite arbre_test 2;;
```

```
Out[27]: - : int * int = (4, 5)
Out[27]: - : int * int = (1, 2)
Out[27]: - : int * int = (3, 4)
Out[27]: - : int * int = (4, 5)
Out[27]: - : int * int = (5, 6)
Out[27]: - : int * int = (2, 3)
Out[27]: - : int * int = (5, 6)
Out[27]: - : int * int = (6, 7)
```

Exercice 9 : `echange`

In [28]:
```
let echange (a : 'a array) (i : int) (j : int) : unit =
    let vi, vj = a.(i), a.(j) in
    a.(i) <- vj;
    a.(j) <- vi;
;;
```

```
Out[28]: val echange : 'a array → int → int → unit = <fun>
```

Exercice 10 : `insertion`

Si besoin, en insérant un élément dans un tableau déjà plein, on doit doubler sa capacité. Ce n'est pas compliqué : d'abord on double le tableau, puis on fait l'insertion normale.

In [29]:
```
let double_capacite (an : arbre_tournoi) : arbre_tournoi =
    let c = capacite an in
    let a2 = Array.make (2 * c) (-1) in
    for i = 0 to an.n - 1 do
        a2.(i) <- an.a.(i)
    done;
    { n = an.n; a = a2 }
;;
```

```
Out[29]: val double_capacite : arbre_tournoi → arbre_tournoi = <fun>
```

L'opération élémentaire s'appelle une "percolation haute" : pour rétablir si nécessaire la propriété d'ordre du tas binaire : tant que `x` n'est pas la racine de l'arbre et que `x` est strictement inférieur (tas min) à son père on échange les positions entre `x` et son père.

In [31]:
```
let percolation_haute (an : arbre_tournoi) (i : int) : unit =
    let i = ref i in
    let p = ref (fst (parent an !i)) in
    (* Printf.printf "\nStart:\ni = %i, p = %i%!" !i !p; flush_all(); *)
    while ((valeur an !p) > (valeur an !i)) do
        echange an.a !i !p;
        i := !p;
        p := fst (parent an !i);
        (* Printf.printf "\ni = %i, p = %i%!" !i !p; flush_all(); *)
    done;
;;
```

Out[31]:
```
val percolation_haute : arbre_tournoi → int → unit = <fun>
```

Maintenant, l'insertion a proprement dite :

In [32]:
```
let rec insertion (an : arbre_tournoi) (x : int) : arbre_tournoi =
    let n, c = an.n, capacite an in
    if n == c then begin
        let an2 = double_capacite an in
        insertion an2 x
    end else begin
        let an2 = { n = n + 1; a = Array.copy an.a } in
        an2.a.(n) <- x;
        percolation_haute an2 n;
        an2
    end
;;
```

Out[32]:
```
val insertion : arbre_tournoi → int → arbre_tournoi = <fun>
```

In [33]:
```
let arbre_test = {
    n = 7;
    a = [|1; 2; 3; 4; 5; 6; 7; -1|]
};;

let a2 = insertion arbre_test (-40);;
arbre_test;;

(* on le voit doubler ! *)
insertion a2 (-20);;
```

Out[33]:
```
val arbre_test : arbre_tournoi = {n = 7; a = [|1; 2; 3; 4; 5; 6; 7; -1|]}
```

Out[33]:
```
val a2 : arbre_tournoi = {n = 8; a = [|-40; 1; 3; 2; 5; 6; 7; 4|]}
```

Out[33]:
```
- : arbre_tournoi = {n = 7; a = [|1; 2; 3; 4; 5; 6; 7; -1|]}
```

Out[33]:
```
- : arbre_tournoi =
{n = 9; a = [|-40; -20; 3; 1; 5; 6; 7; 4; 2; -1; -1; -1; -1; -1; -1; -1|]}
```

In [34]:
```
let arbre_test = {
    n = 7;
    a = [|1; 2; 3; 4; 5; 6; 7; -1; -1; -1; -1|]
};;

insertion (insertion (insertion arbre_test (-40)) (-20)) (-10);;
```

Out[34]:
```
val arbre_test : arbre_tournoi =
  {n = 7; a = [|1; 2; 3; 4; 5; 6; 7; -1; -1; -1; -1|]}
```

Out[34]:
```
- : arbre_tournoi = {n = 10; a = [|-40; -20; 3; 1; -10; 6; 7; 4; 2; 5; -1|]}
```

Exercice 11 : `creation`

La sémantique de cette fonction est de créer un tas min à partir d'un tableau de valeur.

In [35]:
```
let creation (a : 'a array) : arbre_tournoi =
    let n = Array.length a in
    let avide = Array.make n (-1) in
    let an = ref {n = 0; a = avide} in
    for i = 0 to n - 1 do
        an := insertion !an a.(i)
    done;
    !an
;;
```

Out[35]:
```
val creation : int array → arbre_tournoi = <fun>
```

In [36]:
```
let arbre_test3 = creation [|20; 1; 3; 5; 7|];;
```

Out[36]:
```
val arbre_test3 : arbre_tournoi = {n = 5; a = [|1; 5; 3; 20; 7|]}
```

Notez que cet arbre est bien tournoi, mais n'est pas trié.

In [37]:
```
est_tournoi arbre_test3;;

let sorted (a : 'a array) : 'a array =
    let a2 = Array.copy a in
    Array.sort Pervasives.compare a2;
    a2
;;

(sorted arbre_test3.a) == (arbre_test3.a);;
```

Out[37]:
```
- : bool = true
```

Out[37]:
```
val sorted : 'a array → 'a array = <fun>
```

Out[37]:
```
- : bool = false
```

Exercice 12 : `diminue_clef`

On peut augmenter ou diminuer la priorité (la clé) d'un nœud mais il faut ensuite satisfaire la contrainte d'ordre. Si l'on augmente la clé on fera donc une percolation-haute à partir de notre nœud et si l'on diminue la clé on fera un percolation-basse.

> Faites le vous-même.

Exercice 13 : `extraire_min`

On fait une percolation basse pour déplacer la racine jusqu'à une feuille, puis on inverse la feuille avec la dernière valeur du tableau (la feuille la plus à droite), et on met une valeur arbitraire (`-1`) dedans et on diminue la taille du tas (`{ n with n = n - 1 }`).

D'abord, on a besoin de récupérer un des deux fils si l'un des deux a une clé plus petite.

In [38]:
```
let indice_min_fils (an : arbre_tournoi) (i : int) : int =
  let g = ref i and d = ref i in
  if a_gauche an i then
    g := fst (fils_gauche an i);
  if a_droite an i then
    d := fst (fils_droite an i);
  if (valeur an !g) < (valeur an !d)
  then !g
  else !d
;;
```

Out[38]:
```
val indice_min_fils : arbre_tournoi → int → int = <fun>
```

La percolation basse n'est pas trop compliquée :

In [40]:
```
let percolation_basse (an : arbre_tournoi) (i : int) : unit =
  let i = ref i in
  let f = ref (indice_min_fils an !i) in
  while ((valeur an !f) < (valeur an !i)) do
    echange an.a !i !f;
    i := !f;
    f := indice_min_fils an !i;
  done;
;;
```

Out[40]:
```
val percolation_basse : arbre_tournoi → int → unit = <fun>
```

Enfin l'extraction du minimum est facile.

In [41]:
```
let extraire_min (an : arbre_tournoi) : (int * arbre_tournoi) =
  let an = { an with a = Array.copy an.a } in
  if a_gauche an 0 then begin
    let m = an.a.(0) in
    an.n <- an.n - 1;
    echange an.a 0 an.n;
    an.a.(an.n) <- (-1);
    percolation_basse an 0;
    m, an
  end
  else
    (snd (racine an)), { n = 0; a = [||] };
;;
```

Out[41]:
```
val extraire_min : arbre_tournoi → int * arbre_tournoi = <fun>
```

Et pour un exemple :

In [42]:
```
let a = creation [|20; 1; 3; 5; 7|];;
let m, a = extraire_min a;;
let m, a = extraire_min a;;
let m, a = extraire_min a;;
let m, a = extraire_min a;;
let m, a = extraire_min a;;
```

```
Out[42]: val a : arbre_tournoi = {n = 5; a = [|1; 5; 3; 20; 7|]}

Out[42]: val m : int = 1
         val a : arbre_tournoi = {n = 4; a = [|3; 5; 7; 20; -1|]}

Out[42]: val m : int = 3
         val a : arbre_tournoi = {n = 3; a = [|5; 20; 7; -1; -1|]}

Out[42]: val m : int = 5
         val a : arbre_tournoi = {n = 2; a = [|7; 20; -1; -1; -1|]}

Out[42]: val m : int = 7
         val a : arbre_tournoi = {n = 1; a = [|20; -1; -1; -1; -1|]}

Out[42]: val m : int = 20
         val a : arbre_tournoi = {n = 0; a = [||]}
```

Exercice 14 : tri par tas

La meilleure façon de vérifier notre implémentation est d'implémenter le tri par tas :

- on construit un tas depuis la liste de valeur,
- on extrait le minimum successivement.

In [48]:
```
let tripartas (a : 'a array) : 'a array =
    let n = Array.length a in
    let avide = Array.make n (-1) in
    let an = ref (creation a) in
    for i = 0 to n - 1 do
        let m, an2 = extraire_min !an in
        avide.(i) <- m;
        an := an2;
    done;
    avide
;;
```

```
Out[48]: val tripartas : int array → int array = <fun>
```

In [53]:
```
let array1 = [| 10; 3; 4; 1; 2; 7; 8; 5; 9; 6 |] ;;
let array2 = tripartas array1;;
assert ((sorted array2) = array2);;
```

```
Out[53]: val array1 : int array = [|10; 3; 4; 1; 2; 7; 8; 5; 9; 6|]

Out[53]: val array2 : int array = [|1; 2; 3; 4; 5; 6; 7; 8; 9; 10|]

Out[53]: - : unit = ()
```

Union-Find

Exercice 15 : Union-Find avec tableaux

Version simple avec des tableaux simples.

In [65]:
```
type representant = Aucun | Element of int;; (* [int option] pourrait suffire *)

type unionfind = representant array;;
```

Out[65]:
```
type representant = Aucun | Element of int
```

Out[65]:
```
type unionfind = representant array
```

In [66]:
```
let create_uf (n : int) : unionfind =
  Array.make n Aucun
;;
```

Out[66]:
```
val create_uf : int → unionfind = <fun>
```

In [67]:
```
let makeset (uf : unionfind) (i : int) : unit =
  if uf.(i) = Aucun then
    uf.(i) <- Element i
  else
    failwith "Element deja present"
;;
```

Out[67]:
```
val makeset : unionfind → int → unit = <fun>
```

In [68]:
```
let union (uf : unionfind) (i : int) (j : int) : unit =
  let n = Array.length uf in
  if (uf.(i) = Aucun || uf.(j) = Aucun) then
    failwith "Element absent";
  for k = 0 to (n - 1) do
    if uf.(k) = Element j then
      uf.(j) <- Element i
  done
;;
```

Out[68]:
```
val union : unionfind → int → int → unit = <fun>
```

In [69]:
```
(* très facile *)
let find (uf : unionfind) (i : int) : int =
  match uf.(i) with
  | Aucun -> failwith "Element absent"
  | Element i -> i
;;
```

Out[69]:
```
val find : unionfind → int → int = <fun>
```

Tests :

In [70]:
```
let uf_test = create_uf 6;;
for i = 0 to 5 do
   makeset uf_test i
done;;

uf_test;;

find uf_test 5;;
union uf_test 1 2;;

uf_test;;
union uf_test 2 5;;

uf_test;;
find uf_test 0;;

uf_test;;
find uf_test 1;;

uf_test;;
find uf_test 1 = find uf_test 5;;
find uf_test 1 = find uf_test 4;;
```

```
Out[70]: val uf_test : unionfind = [|Aucun; Aucun; Aucun; Aucun; Aucun; Aucun|]

Out[70]: - : unit = ()

Out[70]: - : unionfind =
         [|Element 0; Element 1; Element 2; Element 3; Element 4; Element 5|]

Out[70]: - : int = 5

Out[70]: - : unit = ()

Out[70]: - : unionfind =
         [|Element 0; Element 1; Element 1; Element 3; Element 4; Element 5|]

Out[70]: - : unit = ()

Out[70]: - : unionfind =
         [|Element 0; Element 1; Element 1; Element 3; Element 4; Element 2|]

Out[70]: - : int = 0

Out[70]: - : unionfind =
         [|Element 0; Element 1; Element 1; Element 3; Element 4; Element 2|]

Out[70]: - : int = 1

Out[70]: - : unionfind =
         [|Element 0; Element 1; Element 1; Element 3; Element 4; Element 2|]

Out[70]: - : bool = false

Out[70]: - : bool = false
```

Exercice 16 : Union-Find avec forêts

Version avancée avec des forêts.

In [1]:
```
type position = Aucun | Racine | Fils of int;;

type unionfind = position array;;
```

```
Out[1]: type position = Aucun | Racine | Fils of int

Out[1]: type unionfind = position array
```

In [2]:
```
let create_uf (n : int) : unionfind =
  Array.make n Aucun
;;
```

Out[2]:
```
val create_uf : int → unionfind = <fun>
```

In [3]:
```
let makeset (uf : unionfind) (i : int) : unit =
  if uf.(i) = Aucun then
    uf.(i) <- Racine  (* i devient son propre représentant *)
  else
    failwith "Element deja present"
;;
```

Out[3]:
```
val makeset : unionfind → int → unit = <fun>
```

In [4]:
```
let rec find (uf : unionfind) (i : int) : int =
  match uf.(i) with
  | Aucun -> failwith "Element absent"
  | Fils j ->
    let racine = find uf j in
    uf.(i) <- Fils racine; (* modifie la forêt !  *)
    racine
  | Racine -> i             (* la racine est le représentant *)
;;
```

Out[4]:
```
val find : unionfind → int → int = <fun>
```

In [5]:
```
let union (uf : unionfind) (i : int) (j : int) : unit =
  if (uf.(i) = Aucun || uf.(j) = Aucun) then
    failwith "Element absent"
  else
    (* choix arbitraire de préférer la racine de i,
       on devrait préférer celle de l'arbre le plus petit pour "écraser" la forêt.
       Cf. Papadimitriou ou Cormen (ou Wikipédia).
    *)
    let racinei = find uf i in
    uf.(racinei) <- Fils j
;;
```

Out[5]:
```
val union : unionfind → int → int → unit = <fun>
```

Tests :

In [6]:
```
let uf_test = create_uf 6;;
for i = 0 to 5 do
   makeset uf_test i
done;;

uf_test;;

find uf_test 5;;
union uf_test 1 2;;

uf_test;;
union uf_test 2 5;;

uf_test;;
find uf_test 0;;

uf_test;;
find uf_test 1;;

uf_test;;
find uf_test 1 = find uf_test 5;;
find uf_test 1 = find uf_test 4;;
```

```
Out[6]: val uf_test : unionfind = [|Aucun; Aucun; Aucun; Aucun; Aucun; Aucun|]
Out[6]: - : unit = ()
Out[6]: - : unionfind = [|Racine; Racine; Racine; Racine; Racine; Racine|]
Out[6]: - : int = 5
Out[6]: - : unit = ()
Out[6]: - : unionfind = [|Racine; Fils 2; Racine; Racine; Racine; Racine|]
Out[6]: - : unit = ()
Out[6]: - : unionfind = [|Racine; Fils 2; Fils 5; Racine; Racine; Racine|]
Out[6]: - : int = 0
Out[6]: - : unionfind = [|Racine; Fils 2; Fils 5; Racine; Racine; Racine|]
Out[6]: - : int = 5
Out[6]: - : unionfind = [|Racine; Fils 5; Fils 5; Racine; Racine; Racine|]
Out[6]: - : bool = true
Out[6]: - : bool = false
```

Exercice 17 : Bonus & discussions

En classe.

Je recommande aussi la lecture de ce document (en anglais) (http://jeffe.cs.illinois.edu/teaching/algorithms/notes/17-unionfind.pdf), si tout ça vous intéresse et si vous envisagez d'en faire un développement. Ce document contient notamment une analyse bien propre de la complexité amortie de l'opération Find pour l'algorithme optimisé, qui donne une complexité en $\mathcal{O}(\alpha(n))$ (pour n valeurs dans la structure Union-Find, et si α est la fonction inverse d'Ackermann, cf. Theorem 4 page 9).

Bonus : algorithme de Kruskal

Representations de graphe pondérés

In [7]:
```
type poids = int;;

type arete = Absent | Present of poids;;
type graphe_matrix = arete array array;;

type graphe_list = ((int * poids) list) array;;
```

Out[7]:
```
type poids = int
```

Out[7]:
```
type arete = Absent | Present of poids
```

Out[7]:
```
type graphe_matrix = arete array array
```

Out[7]:
```
type graphe_list = (int * poids) list array
```

In [8]:
```
let taille_graphe = Array.length;; (* nb sommets *)
```

Out[8]:
```
val taille_graphe : 'a array → int = <fun>
```

In [9]:
```
let liste_aretes (gl : graphe_list) =
   let resultat = ref [] in
   let n = taille_graphe gl in
   let rec traitement_liste (i : int) = function (* c'est un List.iter... *)
      | [] -> ()
      | (j, p) :: q -> begin
         resultat := (i, j, p) :: !resultat;
         traitement_liste i q
      end
   in
   for i = 0 to (n - 1) do              (* c'est un Array.iter *)
      traitement_liste i gl.(i)
   done;
   !resultat
;;
```

Out[9]:
```
val liste_aretes : graphe_list → (int * int * poids) list = <fun>
```

In [10]:
```
let graphe_test : graphe_list =
 [| [(1, 11); (2, 2); (3, 1)];
    [(2, 7)];
    [];
    [(4, 5)];
    [(1, 1)]
 |]
;;

liste_aretes graphe_test;;
```

Out[10]:
```
val graphe_test : graphe_list =
  [|[(1, 11); (2, 2); (3, 1)]; [(2, 7)]; []; [(4, 5)]; [(1, 1)]|]
```

Out[10]:
```
- : (int * int * poids) list =
[(4, 1, 1); (3, 4, 5); (1, 2, 7); (0, 3, 1); (0, 2, 2); (0, 1, 11)]
```

Algorithme de Kruskal

In [11]:
```
let kruskal (gl : graphe_list) =
  let aretes = liste_aretes gl in
  let aretes = List.sort (
    fun (_, _, x) -> fun (_, _, y) -> x - y
  ) aretes in
  let n = taille_graphe gl in
  let uf = create_uf n in
  let result = ref [] in  (* difficle de faire sans référence ici... *)
  let traitement_arete (i, j, p) =
  if not (find uf i = find uf j) then begin
    result := (i, j, p) :: !result;
    union uf i j
  end in
  for i = 0 to (n - 1) do
    makeset uf i
  done;
  List.iter traitement_arete aretes;
  !result
;;
```

Out[11]:
```
val kruskal : graphe_list → (int * int * poids) list = <fun>
```

Cet algorithme donne bien un arbre couvrant, il faudrait vérifier sa minimalité.

In [12]:
```
kruskal graphe_test;;
```

Out[12]:
```
- : (int * int * poids) list = [(3, 4, 5); (0, 2, 2); (0, 3, 1); (4, 1, 1)]
```

Conclusion

Fin. À la séance prochaine.