1 Files de priorité

Types courants de files d'attente :

1. file LIFO ou pile : dernier entré, premier sorti
2. file FIFO ou queue : premier entré, premier sorti
3. file de priorité : celui qui a la plus grande priorité est le premier sorti

Analogue à la file d'attente aux urgence d'un hopital : on commence par s'occuper du plus gravement atteint.

Le type de donnée *File de priorité* doit disposer des méthodes :

- test pour savoir si la liste est vide
- insérer un élément dans la file
- trouver l'élément de plus grande priorité
- retirer l'élément de plus grande priorité

Pour simplifier, nous supposerons par la suite que les objets stockés sont des entiers et que le plus grand a la plus grande priorité.

2 Files de priorité et tris

Toute file de priorité permet de trier des éléments : on insère au fur et à mesure les éléments à trier dans la file de priorité, puis on retire les éléments un par un du plus grand au plus petit.

3 Implémentations élémentaires

3.1 Dans une liste non triée

- test pour savoir si la liste est vide : évident, $O(1)$
- insérer un élément dans la file : en tête de la liste, $O(1)$
- trouver l'élément de plus grande priorité : on parcourt la liste, $O(n)$
- retirer l'élément de plus grande priorité : on parcourt la liste, $O(n)$

Tri associé : tri par sélection.

```
let new_priorite () =
    let file = ref [] in
    let rec max_liste = function
       | [] -> failwith "liste vide"
       | [x] -> x
       | t::q -> max t (max_liste q)
    and extrait_max = function
       | [] -> failwith "liste vide"
       | [x] -> x,[]
       | t::q -> let (x,reste) = extrait_max q in
                 if t >= x then (t,q) else (x,t::reste)
    in
    let est_vide () = !file = []
    and insere x = file := x::!file
    and plus_grand () = max_liste !file
    and depile () = let (x,reste) = extrait_max !file in file:=reste; x
    in (est_vide,insere,plus_grand,depile);;
```

3.2 Dans une liste triée

- test pour savoir si la liste est vide : évident, $O(1)$
- insérer un élément dans la file : parcourir la liste, $O(n)$
- trouver l'élément de plus grande priorité : en tête de la liste, $O(1)$
- retirer l'élément de plus grande priorité : on supprime la tête de la liste, $O(1)$

Tri associé : tri par insertion.

```
let new_priorite () =
    let file = ref [] in
    let rec insere x = function
       | [] -> [x]
       | t::q -> if x >= t then x::t::q else t::(insere x q)
    in
    let est_vide () = !file = []
    and insere x = file := insere x !file
    and plus_grand () = hd !file
    and depile () = match !file with
          | [] -> failwith "liste vide"
          | t::q -> file := q; t
    in (est_vide,insere,plus_grand,depile);;
```

3.3 Arbre maximier

C'est un arbre binaire homogène dans lequel l'étiquette d'un noeud interne est toujours supérieure ou égale aux étiquettes de chacun de ses fils.

Test d'un arbre maximier :

```
type 'a arbre_bin  = Vide | Noeud of ('a arbre_bin * 'a * 'a arbre_bin);;

let rec est_maximier = function
        | Vide -> true
        | Noeud (Vide,_,Vide) -> true
        | Noeud (Noeud(_,y,_) as g, x , Vide ) -> (est_maximier g) && x >= y
        | Noeud (Vide, x , Noeud(_,z,_) as d)  -> (est_maximier d) && x >= z
        | Noeud (Noeud(_,y,_) as g, x ,Noeud(_,z,_) as d) ->
                          (est_maximier g) && (est_maximier d)
                          && (x >= y) && (x >= z);;
```

Fonctions évidentes :

```
let est_vide = function
         | Vide -> true
         | _ -> false
and maximum = function
         | Vide -> failwith "arbre vide"
         | Noeud (_,x,_) -> x;;
```

3.4 Insertion dans un arbre maximier

Par induction structurelle.

Insérer un élément x dans un arbre maximier A.

- Si l'arbre est vide, retourner l'arbre à un seul noeud étiqueté par x.
- Si l'arbre possède une racine r étiquetée par $\varepsilon(r)$
  - si $\varepsilon(r) \geq x$, insérer l'élément x dans l'une des branches g issue de r ; cette branche deviendra un arbre maximier g' dont la racine sera soit étiquetée par $x \leq \varepsilon(r)$, soit étiquetée par l'une des étiquettes d'un noeud de g, donc inférieure ou égale à $\varepsilon(r)$; comme l'autre branche n'aura pas été modifiée, l'arbre obtenu sera bien un arbre maximier
  - si $\varepsilon(r) < x$, remplacer l'étiquette de la racine par x puis insérer l'élément $\varepsilon(r)$ dans l'une des branches d issue de r ; cette branche deviendra un arbre maximier d' dont la racine sera soit étiquetée par $\varepsilon(r) < x$, soit par l'étiquette d'un noeud de d donc inférieure à $\varepsilon(r) < x$ (en fait, il est clair que ce dernier cas ne peut pas se produire) ; quant à l'autre branche g issue de r, elle n'a pas été modifiée, c'est donc encore une arbre maximier et l'étiquette de sa racine est inférieure ou égale à $\varepsilon(r)$ et a fortiori à x.

Dans tous les cas, l'arbre obtenu sera un arbre maximier.

Ceci peut se traduire en Caml par la fonction :

```
#let rec insere x a = match a with
        |  Vide -> Noeud(Vide,x,Vide)
        |  Noeud(gauche,n,droite) when x<n -> Noeud((insere x gauche),n,droite)
        |  Noeud(gauche,n,droite) -> Noeud(gauche,x,(insere n droite));;
insere : 'a -> 'a arbre_bin -> 'a arbre_bin = <fun>
```

Pourquoi gauche-droite ? ?

3.5 Depilement récursif dans un arbre maximier

La suppression de la racine d'un arbre maximier peut se faire de manière récursive : on choisit la branche ayant la racine de plus grande priorité, on extrait cette racine et on en fait la nouvelle racine de l'arbre.

On aboutit à l'algorithme suivant :

```
let rec depile_racine =
    (* retourne la racine d'un maximier et l'arbre privé de sa racine *)
     let rec aux = function (* supprime la racine d'un maximier *)
        |  Vide -> failwith "arbre vide"
        |  Noeud (Vide,_,Vide)-> Vide
        |  Noeud (Vide,_,d) -> d
        |  Noeud (g,_,Vide) -> g
        |  Noeud (Noeud(_,y,_) as g, x ,Noeud(_,z,_) as d) ->
                if y >= z then Noeud (aux g,y,d) else Noeud (g,z,aux d)
     in function
        |  Vide -> failwith "arbre vide"
        |  Noeud (g,x,d) as a -> x,aux a;;
```

Inconvénient : on ne maîtrise absolument pas la géométrie de l'arbre.

3.6 Dépilement dans un arbre maximier par percolation

Il est facile de supprimer sont les noeuds terminaux d'un arbre maximier.

La suppression de la racine peut alors se faire en trois étapes.

- échanger la racine avec un noeud terminal de l'arbre ;
- supprimer ce noeud terminal ;
- rétablir ensuite la structure de file de priorité : la percolation.

La percolation va consister à faire redescendre l'étiquette de la racine tout au long de l'arbre en l'échangeant récursivement avec la plus grande des étiquettes de ses fils, jusqu'à ce que son étiquette soit plus grande que toutes les étiquettes de ses fils.

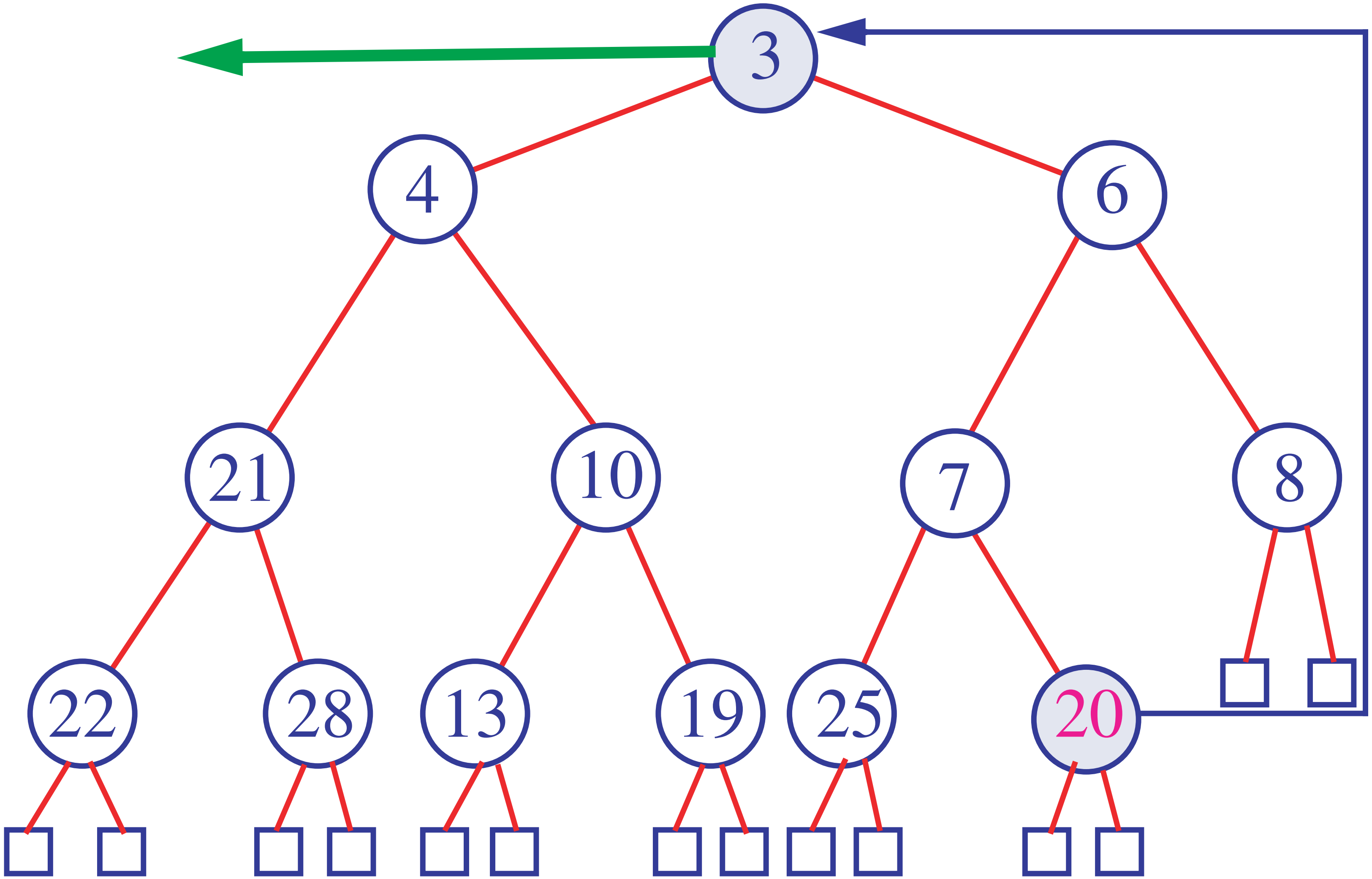


Fig. 1: Retrait de 3, à remplacer par 20

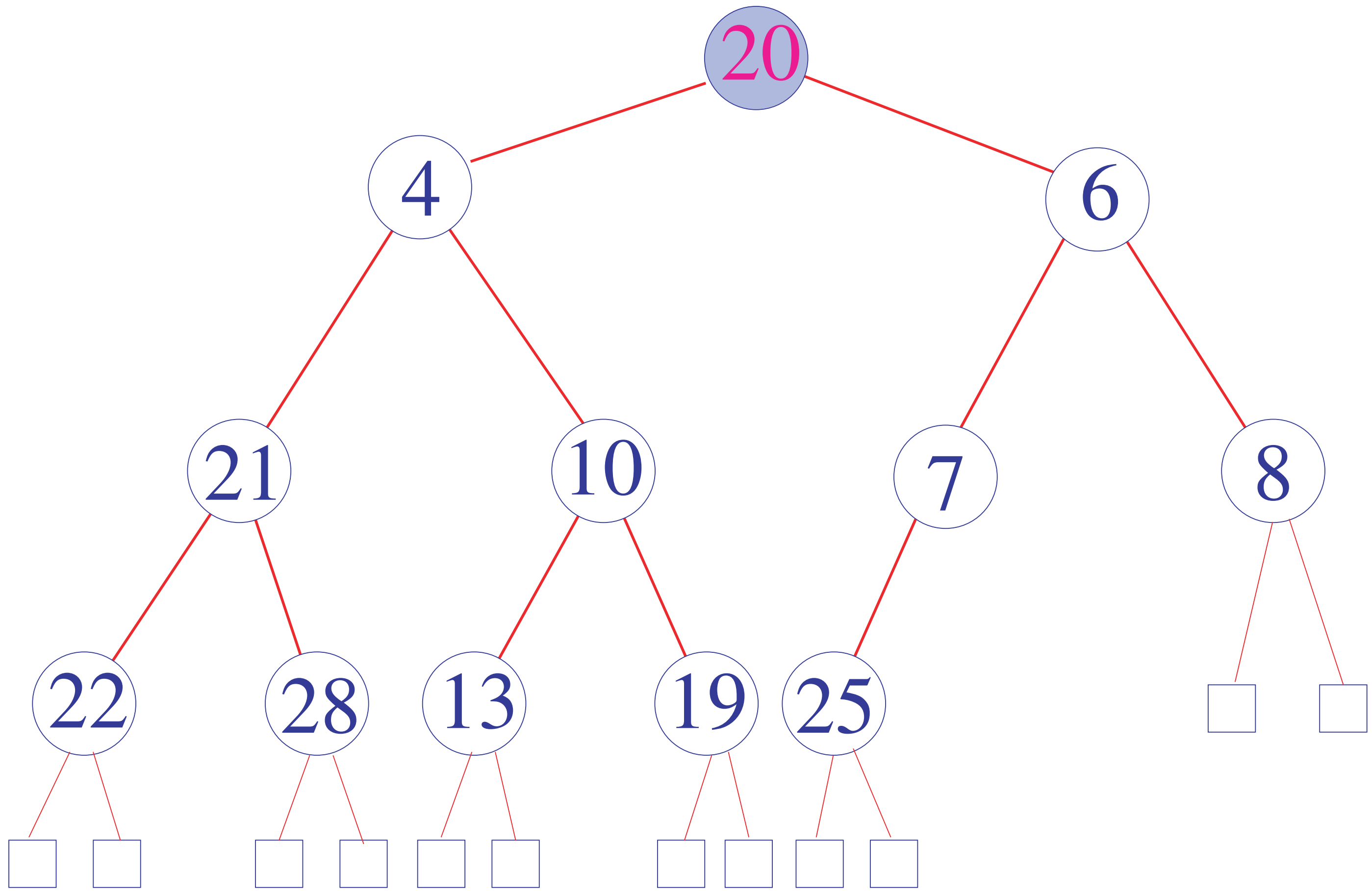


Fig. 2: Arbre à corriger

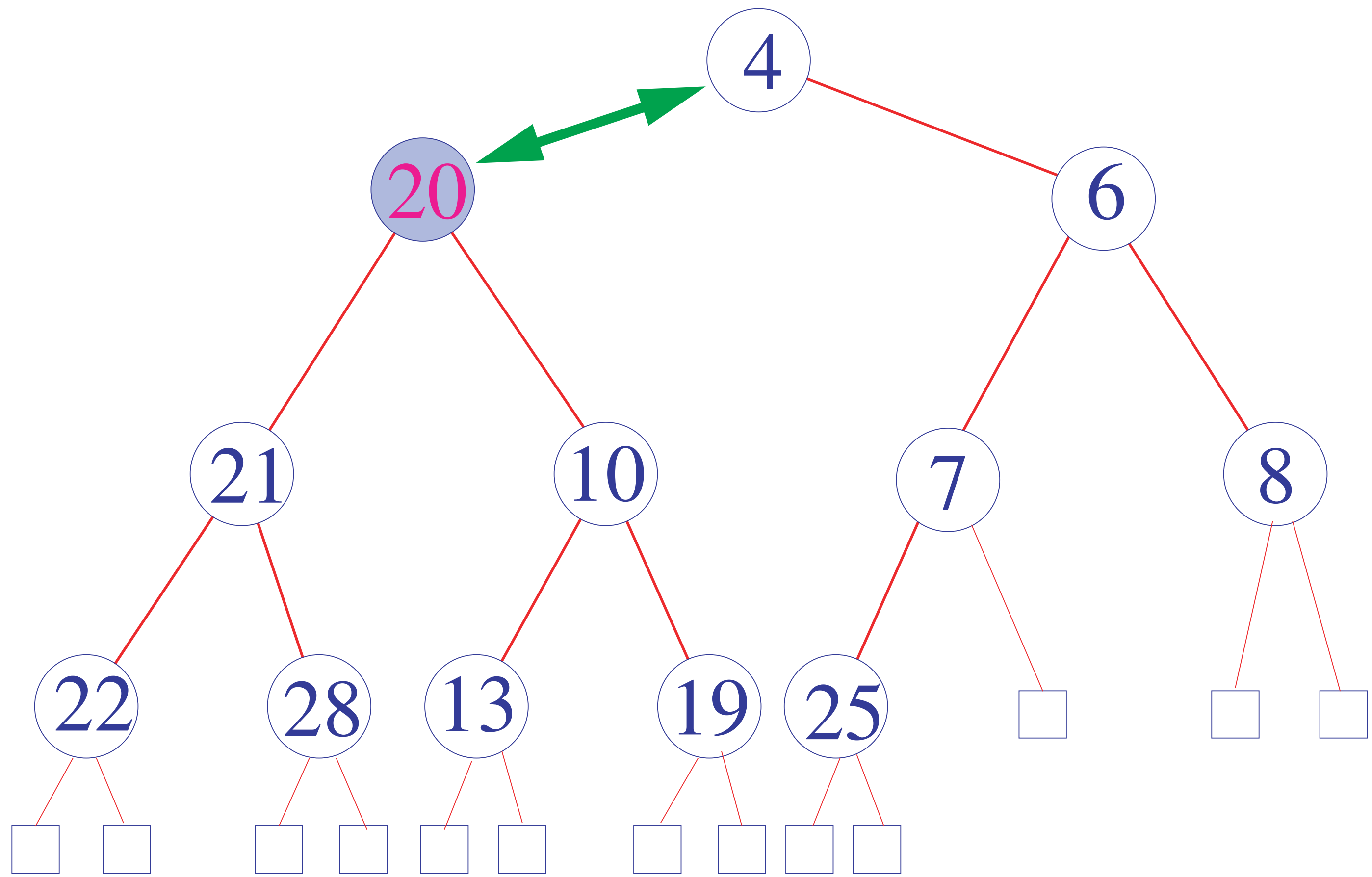


FIG. 3: Echanger 20 avec le plus petit de ses fils

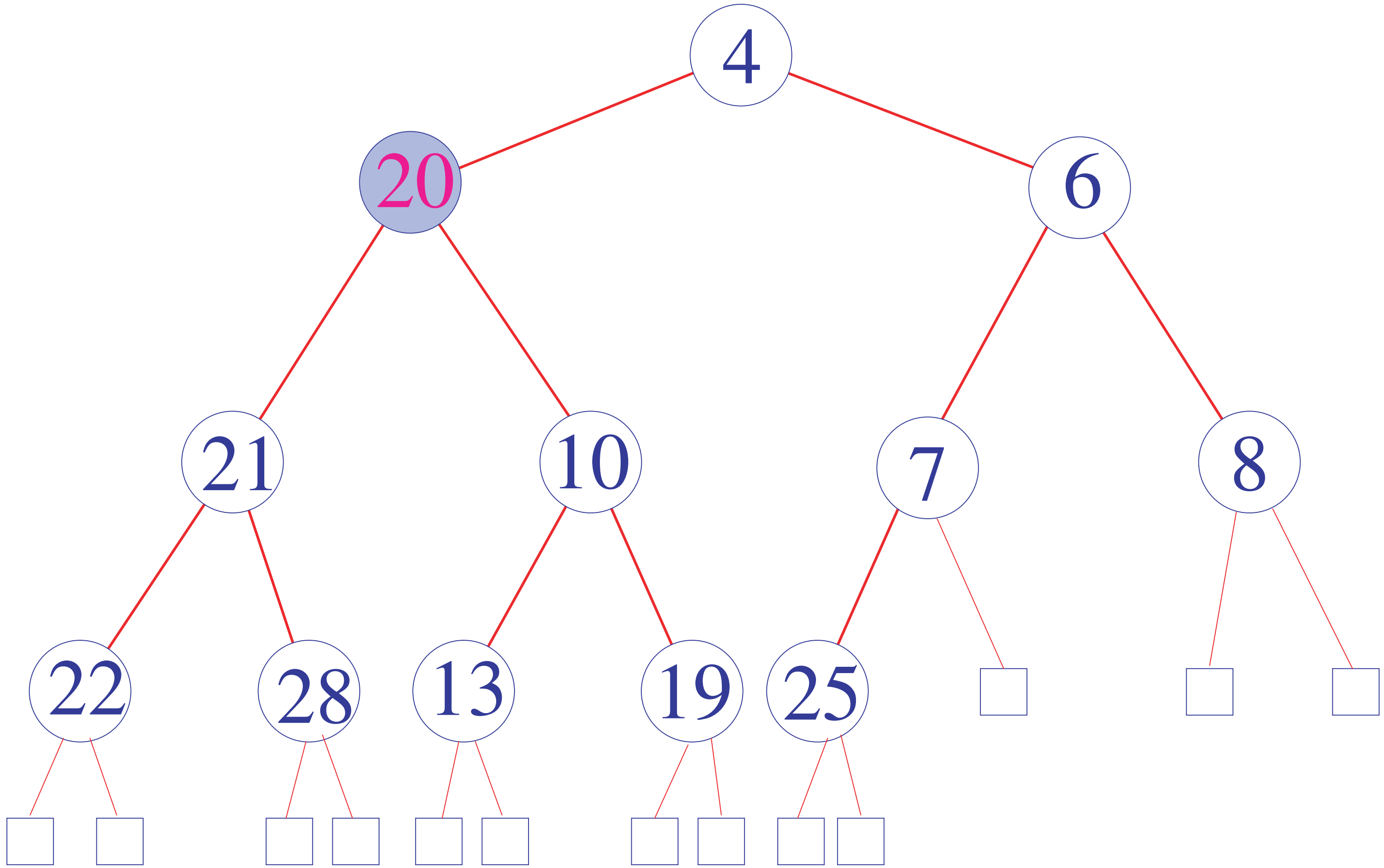


FIG. 4: Arbre à corriger

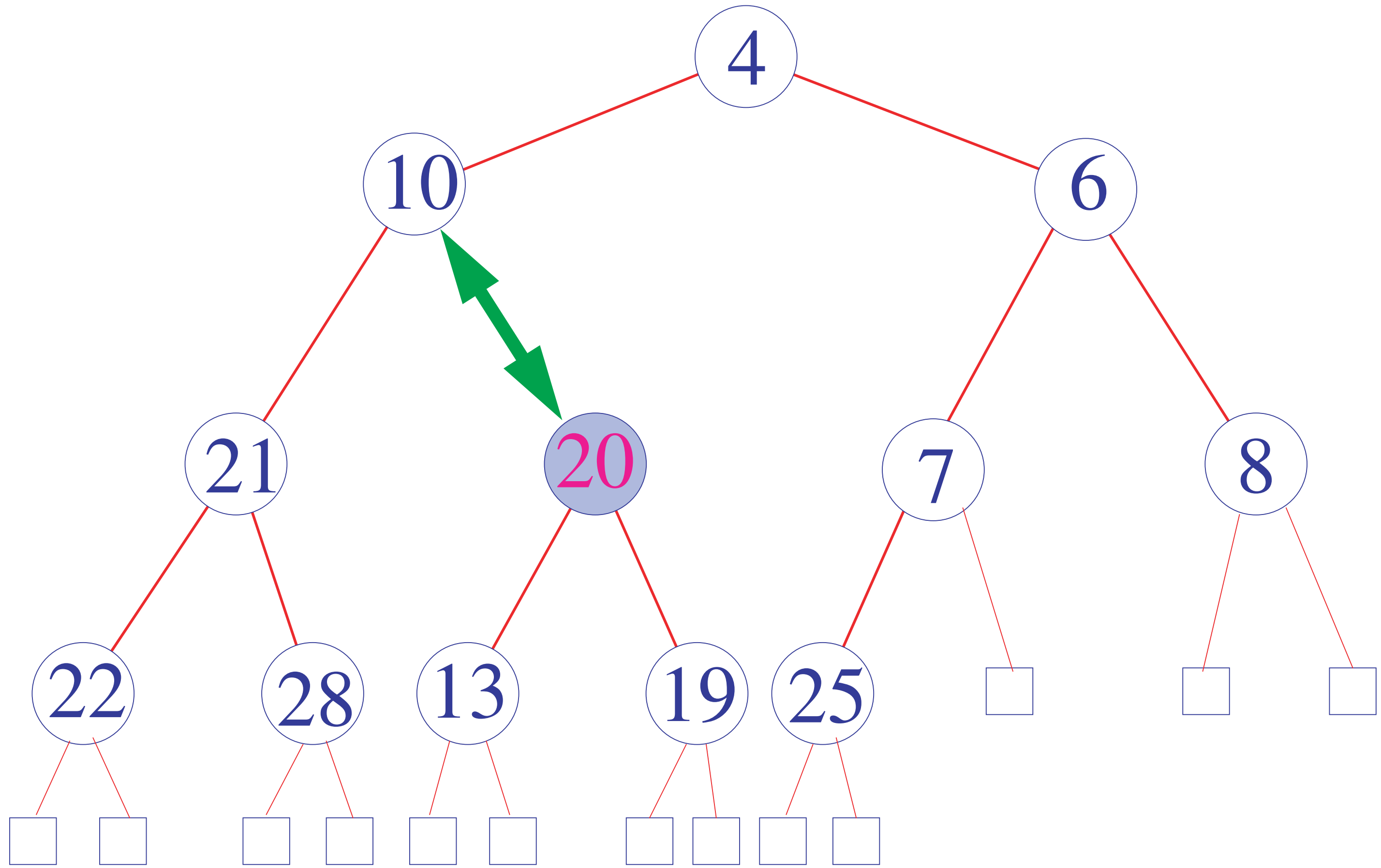


Fig. 5: Echanger 20 avec le plus petit de ses fils

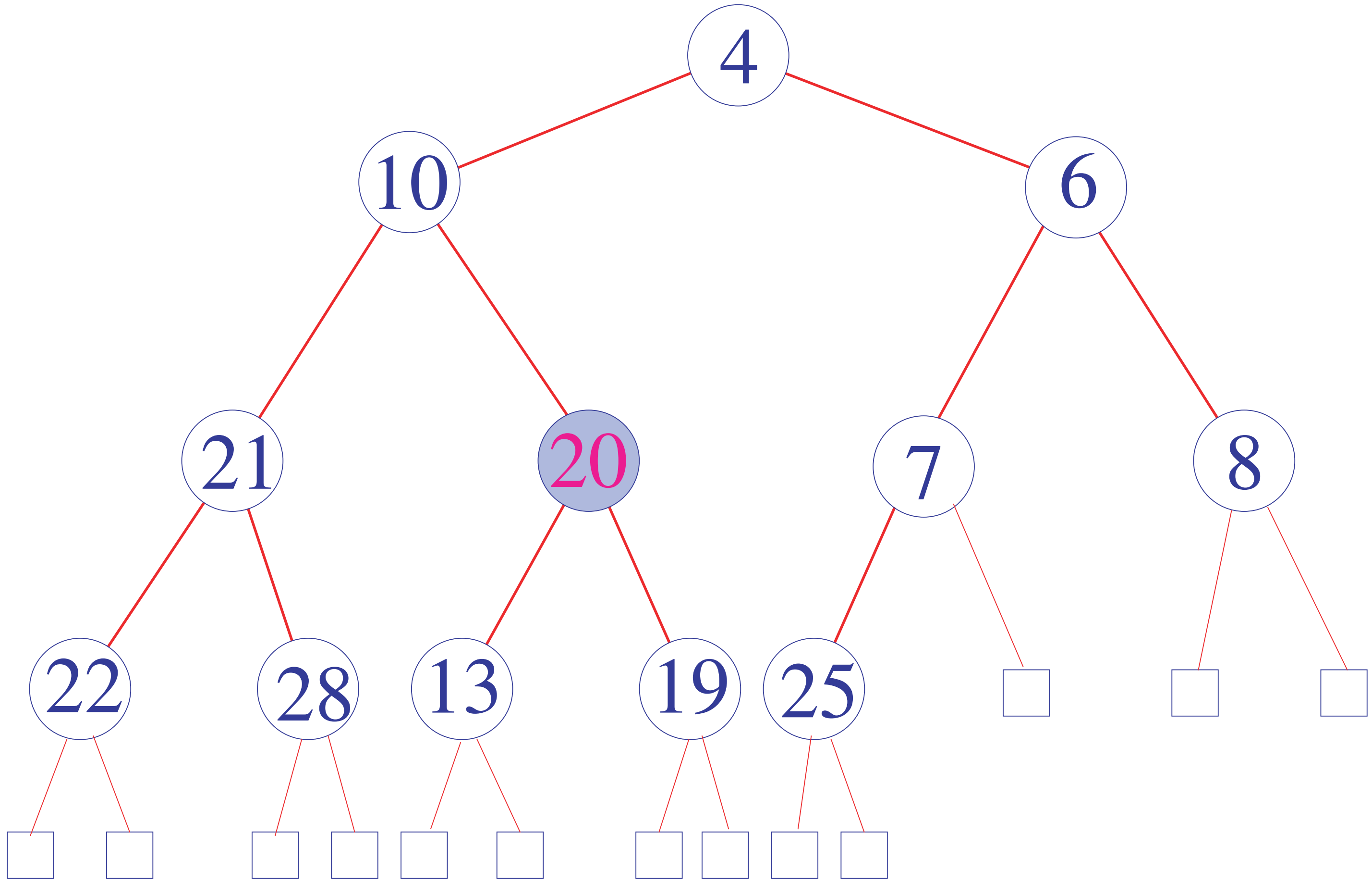


Fig. 6: Arbre à corriger

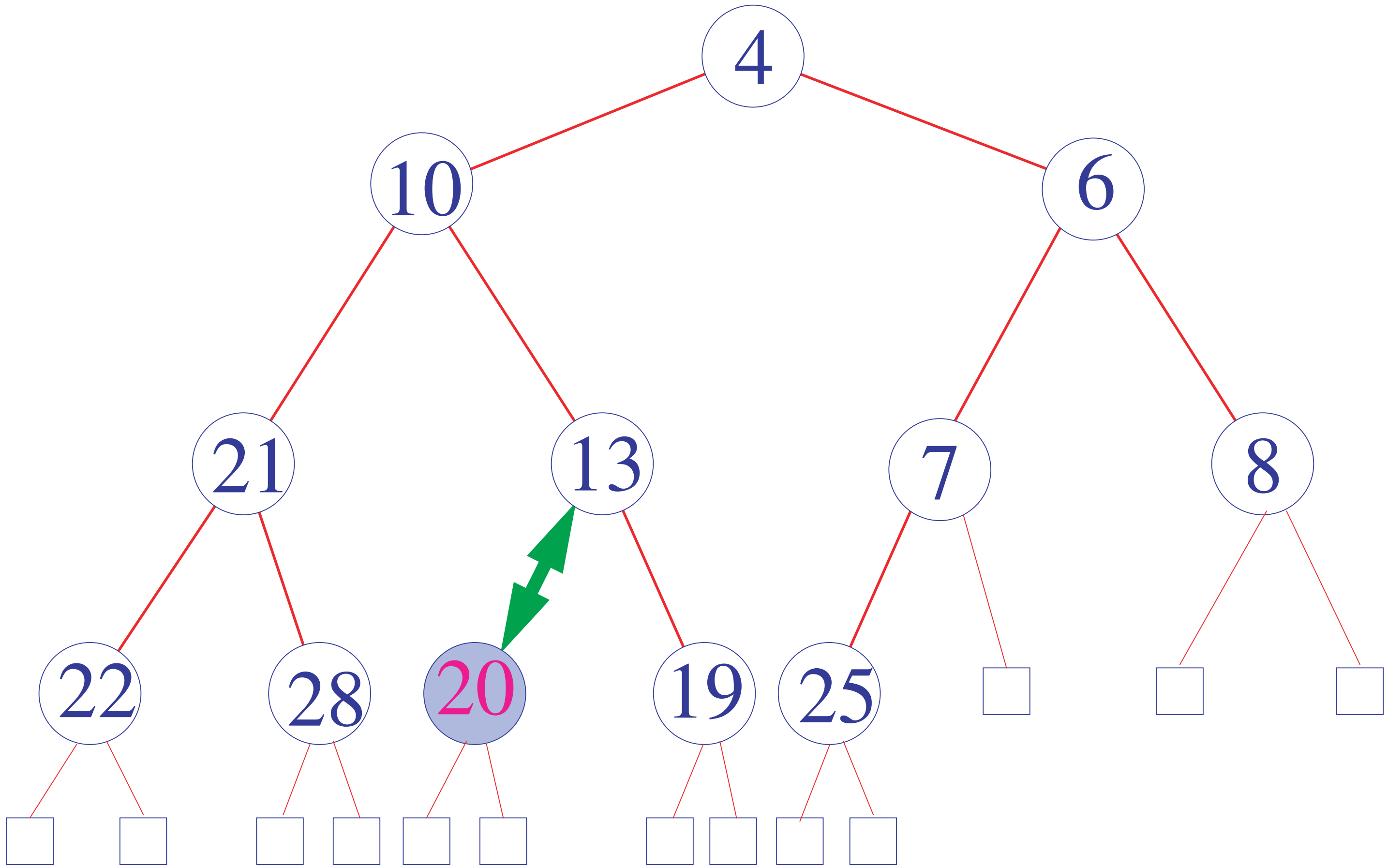


Fig. 7: Echanger 20 avec le plus petit de ses fils

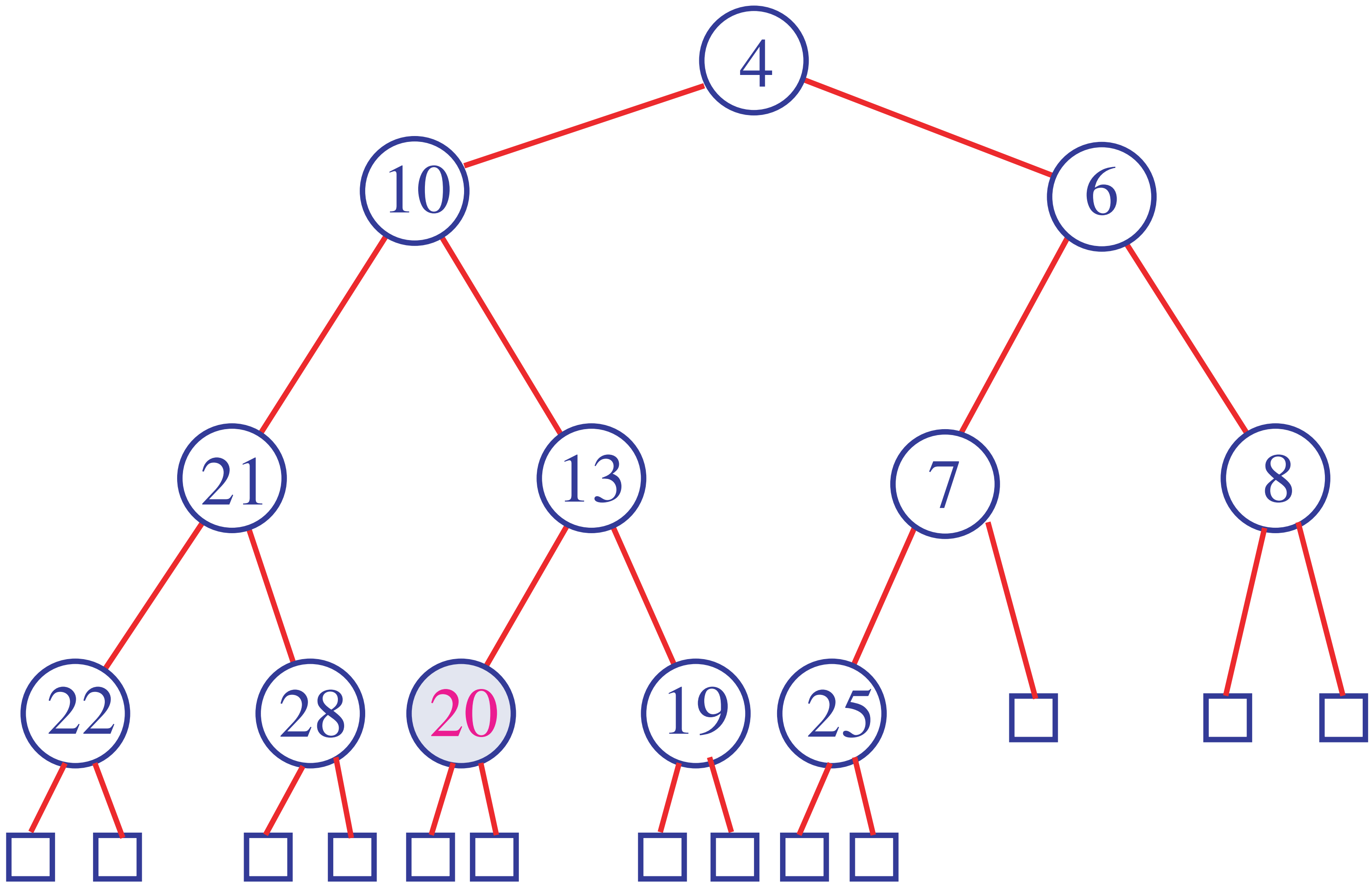


FIG. 8: Arbre corrigé

Percolation de la racine à travers un arbre maximier

```
let rec percole = function
                 |  Noeud(Noeud (ssgauche,m,ssdroite),n,Vide) when m>n ->
                            Noeud(percole (Noeud(ssgauche,n,ssdroite)),m,Vide)
                 |  Noeud(Vide,n,Noeud (ssgauche,m,ssdroite)) when m>n ->
                            Noeud(Vide,m,percole (Noeud(ssgauche,n,ssdroite)))
                 |  Noeud(Noeud(gg,mg,dg),n,(Noeud(gd,md,dd) as droite))
                        when (mg>n && mg>=md) ->
                            Noeud(percole (Noeud(gg,n,dg)),mg,droite)
                 |  Noeud((Noeud(gg,mg,dg) as gauche),n,Noeud(gd,md,dd))
                        when md>n ->
                            Noeud(gauche,md,percole (Noeud(gd,n,dd)))
                 |  a -> a;;
percole : 'a arbre_bin -> 'a arbre_bin = <fun>
```

Extraction de la racine d'un arbre maximier

```
let supprime_racine =
        let rec supprime_terminal = function
          | Vide -> invalid_arg "arbre vide"
          | Noeud(Vide,n,Vide) -> n,Vide
          | Noeud(Vide,n,droite) -> let (x,d) = supprime_terminal droite
                                          in (x,Noeud(Vide,n,d))
          | Noeud(gauche,n,droite) -> let (x,g) = supprime_terminal gauche
                                          in (x,Noeud(g,n,droite))
        in function
          | Vide -> invalid_arg "arbre vide"
          | Noeud(Vide,n,Vide) -> Vide
          | a -> begin
                   match supprime_terminal a with
                     | (term , Noeud (gauche,r,droite)) ->
                                 (r , percole (Noeud(gauche,term,droite)))
                     | _ -> invalid_arg "erreur inconnue"
                 end;;
```

3.7 La structure de tas

Procédures de suppression de la racine ou d'insertion d'un élément dans un arbre maximier : complexité en $O(h)$ où h est la hauteur de l'arbre.

On peut espérer une complexité moyenne en $O(\log_2 n)$, si n est le nombre de noeuds.

Mais ces opérations répétées ont tendance à déséquilibrer les arbres du fait que l'on a à tout moment la liberté de choix entre la *droite* et la *gauche*, et que la méthode la plus naturelle consiste à faire ce choix de manière systématique.

Or pour des arbres déséquilibrés, la complexité passe en $O(n)$.

Solution : forcer les arbres à être équilibrés. Intervention sur la géométrie de l'arbre.

Exemple : dans la suppression par percolation, supprimer un des noeuds terminaux de profondeur maximale.

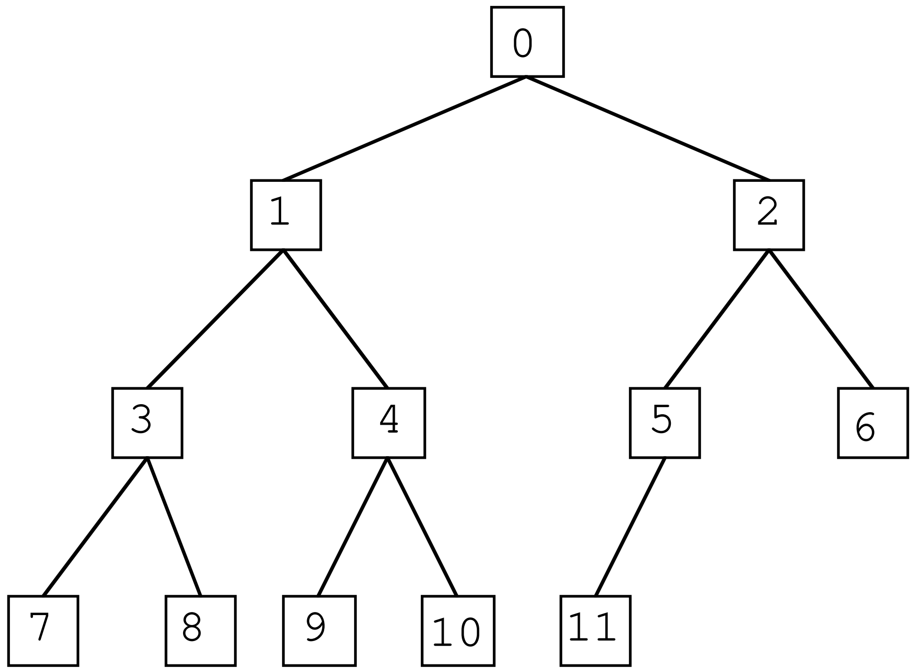
Considérons un arbre binaire homogène de hauteur h. Si $0 \leq d \leq h$, on appellera *niveau* d de l'arbre l'ensemble des noeuds situés à une distance d de la racine.

Le niveau d de l'arbre a au plus 2^d éléments.

Définition 3.1 *On dit qu'un arbre binaire homogène de hauteur h est complet si pour tout $d \in [0, h-1]$, le niveau d de l'arbre possède 2^d éléments et si les noeuds du niveau h sont les plus à gauche possible.*

Définition 3.2 *On appelle tas (en anglais heap) un arbre binaire homogène qui est un arbre maximier.*

On numérote alors les noeuds de haut en bas et de gauche à droite, depuis 0 jusqu'à $n-1$, suivant le schéma suivant :



Cette numérotation permet de stocker les éléments de l'arbre dans un tableau de longueur n (dont les indices varieront de 0 à $n-1$, attention au décalage).

Les fils de l'élément numéroté i sont l'élément numéroté $2i+1$ pour le fils gauche et $2i+2$ pour le fils droit.

Tableau de grande taille dont seuls les n premiers éléments seront utilisés : type enregistrement pour conserver à la fois la vraie longueur n du tableau et le tableau lui même.

```
#type tas={mutable nombre:int; contenu:int vect};;
Type tas defined.
#let nouveau_tas n={nombre = 0; contenu = make_vect n 0};;
nouveau_tas : int -> tas = <fun>
```

Transformer un tas en un arbre :

```
#let tas_en_arbre h = traite_noeud 0
    where rec traite_noeud i=
               if i >= h.nombre then Vide
               else Noeud(
                          (traite_noeud (2*i+1)),h.contenu.(i),
                          (traite_noeud (2*i+2))
                         );;
tas_en_arbre : tas -> int arbre_bin = <fun>
```

Transformer un arbre (supposé complet) en tas :

```
#let arbre_en_tas a h = traite_noeud 0 a
     where rec traite_noeud i = function
               Vide -> ()
            |  Noeud(gauche,n,droite) ->
                      begin
                         if i>= h.nombre then h.nombre <- i+1;
                         h.contenu.(i) <- n;
                         traite_noeud (2*i+1) gauche;
                         traite_noeud (2*i+2) droite
                      end;;
arbre_en_tas : int arbre_bin -> tas -> unit = <fun>
```

3.8 Suppression de la racine dans un tas

Adapter l'algorithme de suppression de la racine d'un arbre maximier : échanger l'étiquette de la racine avec l'étiquette d'un noeud terminal, puis supprimer ce noeud terminal, et enfin effectuer une percolation.

Supprimer le noeud le plus à droite du niveau maximal, c'est-à-dire dans la structure de tas, le dernier élément numéroté.

```
let supprime_racine h =
          let n = h.nombre and racine = h.contenu.(0) in
          h.contenu.(0) <- h.contenu.(n-1);
          h.nombre <- n-1;
          percole h;
          racine
where percole h=...
```

Percolation : faire redescendre l'étiquette de la racine tout au long du tas en l'échangeant récursivement avec la plus grande des étiquettes de ses fils, jusqu'à ce que son étiquette soit plus grande que toutes les étiquettes de ses fils.

```
#let percole h =
        let n = h.nombre and v = h.contenu in
        let rec traite_noeud i = (* échange l'élément i avec le plus grand
                                    de ses fils si nécessaire et recommence *)
             if 2*i+1 < n then (* le noeud a au moins un fils *)
               let j=2*i+1 in (* le fils gauche *)
               let fils = (* sélectionne le plus grand des "deux" fils *)
                 if (j+1<n && v.(j)<v.(j+1)) then j+1 else j
               in
                 if v.(fils) > v.(i) then
                    begin
                      echange i fils v;
                      traite_noeud fils
                    end
       in traite_noeud 0;;
percole : tas -> unit = <fun>

#let echange i j v = (* échange les éléments de numéros i et j du tableau v*)
             let temp = v.(i) in v.(i) <- v.(j); v.(j) <- temp;;
```

Amélioration : les échanges concernent toujours un même élément, la racine.

Pas nécessaire d'effectuer complètement chaque échange :

- affecter au père l'étiquette de son fils lorsqu'un échange est nécessaire
- lorsque le processus s'arrête, d'affecter au noeud l'étiquette de la racine.

```
let percole h =
        let n = h.longueur and v = h.contenu and racine = h.contenu.(1) in
        let rec traite_noeud i = (* échange l'élément i avec le plus petit
                                   de ses fils si nécessaire et recommence *)
             if 2*i+1 < n then (* le noeud a au moins un fils *)
                begin
                  let j = 2*i+1 in (* le fils gauche *)
                  let fils = (* sélectionne le plus petit des "deux" fils *)
                     if (j+1<n && v.(j)<v.(j+1)) then j+1 else j
                  in
                     if v.(fils) > racine then
                       begin
                         v.(i) <- v.(fils); (* on remonte le fils *)
                         traite_noeud fils  (* on traite la branche *)
                       end
                     else
                       v.(i) <- racine (* on stocke l'élément *)
               end
            else (* le noeud est terminal *)
               v.(i) <- racine (* on stocke l'élément *)
       in traite_noeud 1;;
percole : tas -> unit = <fun>
```

Complexité de cet algorithme est dominée par la hauteur de l'arbre, c'est donc un $O(\log_2 n)$ si n est le nombre de noeuds.

3.9 Insertion dans un tas

La technique d'insertion dans un arbre maximier par induction structurelle est totalement inadaptée à la structure de tas.

Cette méthode consistait

- dans un cas à insérer l'élément dans la branche gauche ou la branche droite de l'arbre
- dans l'autre cas à remplacer l'étiquette de la racine par le nouvel élément puis à insérer l'ancienne racine dans la branche gauche ou la branche droite de l'arbre

Détruit la structure d'arbre complet : un seul chemin d'insertion dans l'arbre qui permette de maintenir cette structure, mais il faudrait pratiquement être devin pour le trouver.

Privilégier la structure d'arbre complet en insérant le nouvel élément à la première place libre. On détruit la structure d'arbre maximier.

Rétablir la structure de tas par une opération inverse de la percolation : remonter l'élément de proche en proche en l'échangeant avec l'étiquette de son père tant que cette étiquette est inférieure.

```
#let insere x h =
            if h.nombre >= vect_length h.contenu then
                  failwith "débordement du tas";
            h.contenu.(h.nombre) <- x;
            h.nombre <- h.nombre +1;
            retablis h
where retablis h =
            let n = ref (h.nombre-1) in (* dernier élément *)
            let p = ref ((!n-1)/2) (* son père *)
            and v = h.contenu in
              while (!n>=1 && v.(!p) < v.(!n)) do
                    echange !n !p  v;
                    n := !p; (* nouveau fils *)
                    p := (!n-1)/2 (* nouveau père *)
              done;;
insere : int -> tas -> unit = <fun>
```

Validité de cet algorithme : la procédure `retablis` appliquée à un tableau qui est un tas, sauf peut-être en ce qui concerne son dernier élément, transforme ce tableau en un tas.

Cette procédure ne modifiant pas la structure de l'arbre conserve la propriété d'être un arbre complet. Il suffit donc de montrer qu'elle transforme l'arbre étiqueté en un arbre maximier.

Terminaison de la boucle : stricte décroissance du contenu de la référence n (qui est divisée par 2 à chaque itération) et le contenu de n est supérieure ou égale à 1 (ce qui garantit que n n'est pas la racine de l'arbre).

Invariant de la boucle :

- au début de l'itération, p est père de n, les deux branches issues de p sont des files de priorité
- l'étiquette de p est supérieure ou égale à celle de son autre fils éventuel n'

La complexité de cette insertion est manifestement majorée par la hauteur de l'arbre, elle est donc un $O(\log_2 n)$ où n est le nombre de noeuds.

3.10 Tri en tas

Le principe général : on insère les éléments du tableau successivement dans un tas initialement vide puis on extrait itérativement la racine de ce tas (qui en est toujours le plus petit élément) jusqu'à ce que ce tas soit vide.

```
#let tri_en_tas v =
         let n = vect_length v in
         let h = nouveau_tas n in
         for i=0 to n-1 do
             insere v.(i) h
         done;
         for i=0 to n-1 do
             v.(i) <- supprime_racine h
         done;;
tri_en_tas : int vect -> unit = <fun>
```

Comme l'insertion et la suppression de la racine ont une complexité en $O(\log_2 n)$, le tri en tas a une complexité en $O(n \log_2 n)$ dans le pire des cas.

Eviter le recours à un tas auxiliaire : économie de temps et de mémoire.

Propager la structure de tas à partir des sous-arbres de l'arbre complet obtenu à partir du tableau, en remontant la structure de tas depuis les noeuds terminaux jusqu'à la racine de l'arbre.

Les sous-arbres réduits aux noeuds terminaux sont évidemment des tas.

Soit n un noeud non terminal et supposons que la ou les branches issues de ce noeud soient des tas. Si le sous-arbre de racine n n'est pas un tas, il le deviendra en effectuant une percolation de l'étiquette de n à travers ce sous arbre. Après l'exécution complète de cette procédure, le tableau aura été transformé en un tas sans avoir utilisé d'insertions dans un autre tas.

```
#let vect_en_tas v=
        let n = (vect_length v)-1 in
        let rec traite_noeud i a_inserer=
                                (* échange l'élément i avec le plus grand
                                   de ses fils si nécessaire et recommence;
                                   au final insère l'élément a_inserer *)
             if 2*i+1 <= n then (* le noeud a au moins un fils *)
                begin
                  let j = 2*i+1 in (* le fils gauche *)
                  let fils = (* sélectionne le plus grand des "deux" fils *)
                     if (j<n && v.(j)<v.(j+1)) then j+1 else j
                  in
                     if v.(fils) > a_inserer then
                       begin
                         v.(i) <- v.(fils); (* on remonte le fils *)
                         traite_noeud fils a_inserer (* on traite la branche *)
                       end
                     else
                       v.(i) <- a_inserer (* on insère l'élément *)
               end
            else (* le noeud est terminal *)
               v.(i) <- a_inserer (* on insère l'élément *)
```

```
       in
            for i = (n+1)/2 downto 0 do
                   traite_noeud i v.(i)
            done;;
vect_en_tas : 'a vect -> unit = <fun>
```

Invariant de la dernière boucle indexée par i :

- après exécution de l'itération d'indice i, tous les sous-arbres ayant pour racines $v.(i), v.(i+1) \dots, v.(n)$ sont des tas

La terminaison de la boucle est évidente puisqu'il s'agit d'une boucle indexée. Après l'exécution de l'itération d'indice 0, le tableau tout entier a été transformé en tas.

L'extraction successive des racines peut alors se faire directement en place : pour extraire la racine d'un tas (c'est à dire son plus grand élément), nous l'avons échangée avec le dernier élément du tas (ce qui la place donc en dernière position), puis nous l'avons supprimée (ce qui revient à diminuer de 1 la longueur de travail) et enfin nous avons effectué une percolation de l'étiquette du premier élément du tableau. En itérant cette méthode, nous allons donc obtenir un tableau dans lequel le plus grand élément sera en dernière position, puis l'élément juste un peu plus petit sera en avant-dernière position, et ainsi de suite. Autrement dit, notre tableau sera trié en ordre croissant.

Ceci nous conduit à la procédure suivant de tri par ordre croissant d'un tas :

```
#let tri_tas v =
       let rec traite_noeud i a_inserer n =
                       (* échange l'élément i avec le plus petit
                          de ses fils si nécessaire et recommence;
                          au final insère l'élément a_inserer;
                          ne traite que les éléments du tableau d'indice 0 à n *)
             if 2*i+1 <= n then (* le noeud a au moins un fils *)
                begin
                  let j = 2*i+1 in (* le fils gauche *)
                  let fils = (* sélectionne le plus grand des "deux" fils *)
                     if (j<n && v.(j)<v.(j+1)) then j+1 else j
                  in
                     if v.(fils) > a_inserer then
```

```
                        begin
                          v.(i) <- v.(fils); (* on remonte le fils *)
                          traite_noeud fils a_inserer n (* on traite la branche *)
                        end
                      else
                        v.(i) <- a_inserer (* on insère l'élément *)
                end
             else (* le noeud est terminal *)
                v.(i) <- a_inserer (* on insère l'élément *)
        in
          for n = (vect_length v)-1 downto 1 do
           echange 0 n v;
              traite_noeud 0 v.(0) (n-1)
          done;;
tri_tas : 'a vect -> unit = <fun>
```

Invariant de la boucle finale :

après l'itération d'indice n les éléments $v.(0), \ldots, v.(n-1)$ forment un tas et on a

$$v.(0) \geq v.(n) \geq v.(n+1) \geq \cdots \geq v.(N-1)$$

si N désigne la longueur du tableau.

Tri en tas d’un tableau :

```
#let tri_en_tas v= (* trie les éléments du tableau v *)
let rec traite_noeud i a_inserer n=
                      (* échange l'élément i avec le plus petit
                         de ses fils si nécessaire et recommence;
                         au final insère l'élément a_inserer;
                         ne traite que les éléments du tableau
                         d'indice 0 à n *)
             if 2*i+1 <= n then (* le noeud a au moins un fils *)
                begin
                  let j = 2*i+1 in (* le fils gauche *)
                  let fils = (* sélectionne le plus petit des "deux" fils *)
                     if (j<n && v.(j)<v.(j+1)) then j+1 else j
                  in
                     if v.(fils) > a_inserer then
                       begin
                         v.(i) <- v.(fils); (* on remonte le fils *)
                         traite_noeud fils a_inserer n(* on traite la branche *)
                       end
                     else
                       v.(i) <- a_inserer (* on insère l'élément *)
               end
```

```
        else (* le noeud est terminal *)
           v.(i) <- a_inserer (* on insère l'élément *)
    in
     let dernier = (vect_length v) -1 in
        for i=(dernier+1)/2 downto 0 do
               traite_noeud i v.(i) dernier
        done;
        for n=dernier downto 1 do
            echange 0 n v;
               traite_noeud 0 v.(0) (n-1)
        done;;
tri_en_tas : 'a vect -> unit = <fun>

#let v= [|8;1;2;3;4;3;8;5;1;8;2;4;12;14;1;3;5|];;
v : int vect = [|8; 1; 2; 3; 4; 3; 8; 5; 1; 8; 2; 4; 12; 14; 1; 3; 5|]
#tri_en_tas v; v;;
- : int vect = [|1; 1; 1; 2; 2; 3; 3; 3; 4; 4; 5; 5; 8; 8; 8; 12; 14|]
```

Comme la procédure `traite_noeud` a une complexité en $O(\log_2 n)$ et que les deux boucles sont au plus de longueur n, la complexité de la procédure de tri en tas est en $O(n \log_2 n)$ dans le pire des cas.