Cours 6 : Structures de données arborescentes partie 1

Jean-Stéphane Varré

Université Lille 1

jean-stephane.varre@lifl.fr

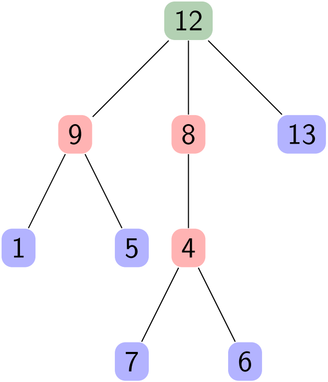
Rappels

- structure de données non linéaire,
- organisation hiérarchique entre les données stockées,
- utilisée pour structurer des données :
  - système de fichiers,
  - base de données
  - sites web
  - fichier xml

Vocabulaire

- noeud : caractérisé par une valeur + un nombre fini de fils, possède un unique père
- feuille : noeud sans fils
- noeud interne : un noeud qui n'est pas une feuille
- arité d'un noeud n : nombre de fils du noeud n
- arité d'un arbre a : nombre maximal de fils d'un noeud de a
- racine d'un arbre a : c'est le seul noeud sans père
- profondeur d'un noeud n : nombre de noeuds sur la branche entre la racine et le noeud n exclu
- hauteur d'un arbre a : c'est le nombre de noeuds sur la branche qui va de la racine de a à la feuille de profondeur maximale

Exemple



Structure de données associée

Cas où l'arité n'est pas bornée

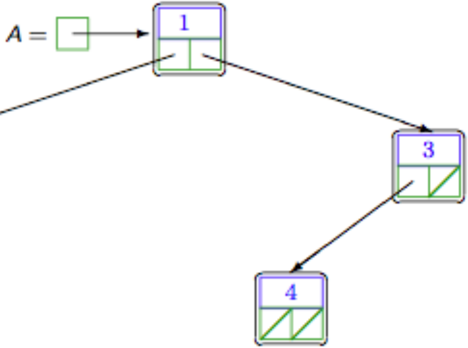
```
type
  ARBRE = ^NOEUD;
  NOEUD = record
    valeur : ELEMENT; // valeur associee au noeud
    fils : LISTE_NOEUDS; // les fils
  end {record};
```

Structure de données associée

Cas où l'arité est bornée (ici un arbre à 2 fils maximum)

```
type
  ARBRE = ^NOEUD;
  NOEUD = record
    valeur : ELEMENT; // valeur associee au noeud
    fils_droit, fils_gauche : ARBRE; // les deux fils
  end {record};
```

Représentation chaînée



Exemple tiré du cours d'API2.

Hauteur, profondeur, arité

Soit A un arbre d'arité a, de taille n et de hauteur h.

- le nombre n_p de noeuds de A à profondeur $0 \leq p \leq h$ vérifie

$$1 \leq n_p \leq a^p$$

- la taille n vérifie l'encadrement

$$h + 1 \leq n \leq \frac{a^{h+1} - 1}{a - 1}$$

- le hauteur h vérifie l'encadrement

$$\log_a(n(a-1)+1) - 1 \leq h \leq n - 1$$

Tiré du cours d'API2

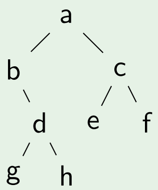
Rappel des primitives de manipulation

```
function creerArbre(e : ELEMENT; g,d : ARBRE) : ARBRE;

// CU : a non vide
function racine(a : ARBRE) : ELEMENT;

// CU : a non vide
function gauche(a : ARBRE) : ARBRE;

// CU : a non vide
function droit(a : ARBRE) : ARBRE;

function estArbreVide(a : ARBRE) : BOOLEAN;

// CU : a non vide
procedure modifierRacine(const a : ARBRE; const e : ELEMENT);

// CU : a non vide
procedure modifierGauche(const a : ARBRE; const g : ARBRE);

// CU : a non vide
procedure modifierDroit(const a : ARBRE; const d : ARBRE);
```

Parcours en profondeur

- On traite récursivement les noeuds de l'arbre.
- Trois sens de parcours :
  - préfixé : traiter la racine puis les fils de gauche à droite,
  - postfixé : traiter les fils de gauche à droite puis la racine,
  - infixé (n'a vraiment de sens que pour les arbres binaires) : traiter le fils de gauche, puis la racine, puis le fils de droite

Exemple



infixé	b, g, d, h, a, e, c, f
postfixé	g, h, d, b, e, f, c, a
préfixé	a, b, d, g, h, c, e, f

Parcours en profondeur

Affichage préfixé

```
procedure afficher (a : ARBRE);
begin
   if not estArbreVide(a) then begin
      // traitement de la racine
      write(racine(a));
      // traitement des fils de gauche a droite
      afficher(gauche(a));
      afficher(droit(a));
   end {if};
end {afficher};
```

Note: pour changer le type de parcours il suffit d'échanger les trois instructions du `if`.

Comment dérécursiver le parcours ?

Parcours en profondeur

Affichage préfixé avec une pile

```
procedure afficher (a : ARBRE);
var
  p : PILE_D_ARBRE;
  s : ARBRE;
begin
  empiler(p,a);
  while not estPileVide (p) do begin
    s := depiler(p);
    if not estArbreVide (s) do begin
      write(racine(s));
      empiler(droit(s));
      empiler(gauche(s));
    end {if};
  end {while};
end;
```

Il ne faut pas se tromper et bien empiler le sous-arbre droit avant le sous-arbre gauche car la pile est une structure LIFO.

Parcours en largeur

On traite les noeuds, des moins profonds aux plus profonds, par strates.

Exemple

```
      a
     / \
    b   c
     \  / \
     d  e  f
    / \
   g   h
```

a,b,c,d,e,f,g,h

Comment réaliser un tel parcours ?

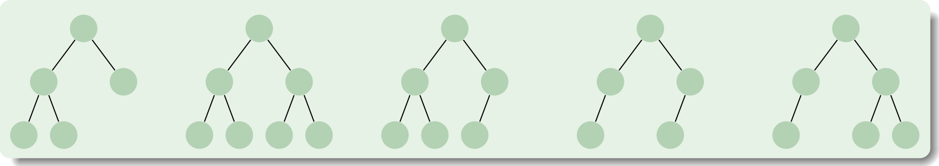
Parcours en largeur

Utilisation d'une file

```
procedure afficher (a : ARBRE);
var
  f : FILE_D_ARBRE;
  s : ARBRE;
begin
  enfiler(f,a);
  while not estFileVide (f) do begin
    s := defiler(f);
    if not estArbreVide (s) then begin
      write(racine(s));
      enfiler(gauche(s));
      enfiler(droit(s));
    end {if};
  end {while};
end {afficher};
```

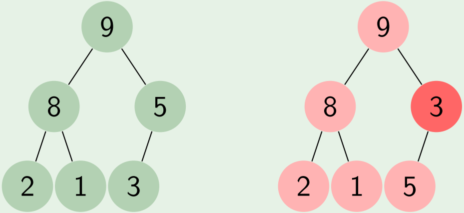
Rappel sur les arbres binaires

- un arbre binaire est un arbre dont chaque noeud possède au plus deux fils
- un arbre binaire complet est un arbre binaire dont tous les noeuds internes possèdent exactement deux fils
- un arbre binaire parfait est un arbre binaire complet pour lequel toutes les feuilles sont à la même profondeur
- un arbre binaire quasi parfait est un arbre binaire tel que :
  - dont tous les noeuds internes sauf eventuellement un possèdent deux fils,
  - toutes les feuilles sont à profondeur h ou $h - 1$,
  - et toutes les feuilles de profondeur h sont groupées à gauche.



Tas

- un tas max est un arbre binaire quasi-parfait dont la valeur associée à chaque noeud est plus grande que celles de ses fils



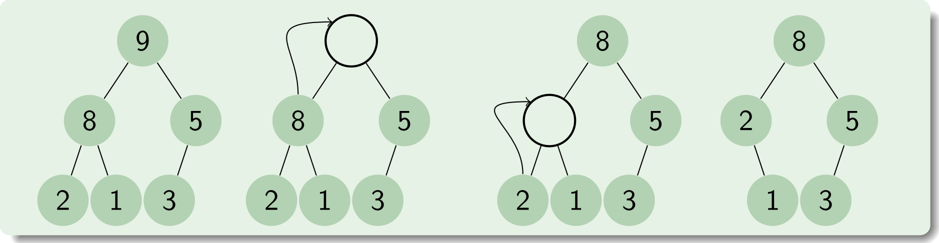
propriétés :

- la valeur la plus grande est située à la racine du tas
- pas d'ordre entre les valeurs des fils d'un noeud (ce n'est pas un arbre binaire ordonné!)
- mais la seconde valeur maximale est nécessairement la valeur d'un des deux fils de la racine
- la hauteur d'un tas de taille n est $h = \lfloor \log_2 n \rfloor$

Trier avec un tas

Idée : extraire la valeur maximale, puis la seconde, etc ...
Principe :

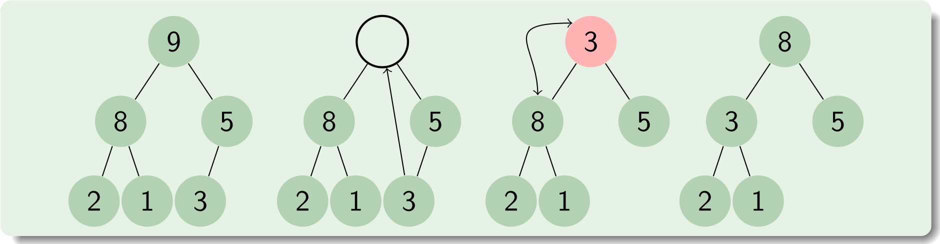
- extraure la racine,
- remonter la seconde valeur maximale à la racine
- recommencer récursivement

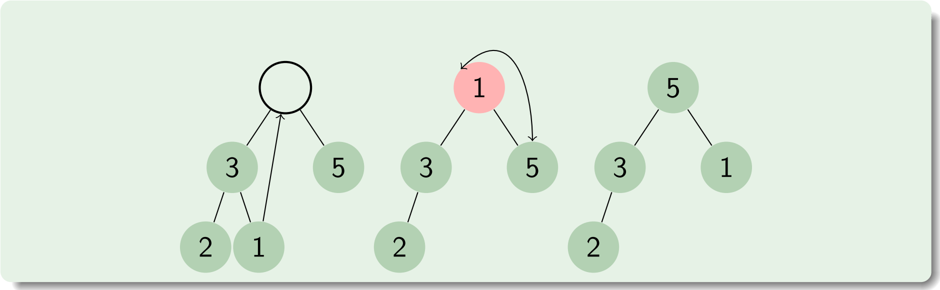


Problème : l'arbre obtenu n'est plus un tas

Suppression de l'élément maximum

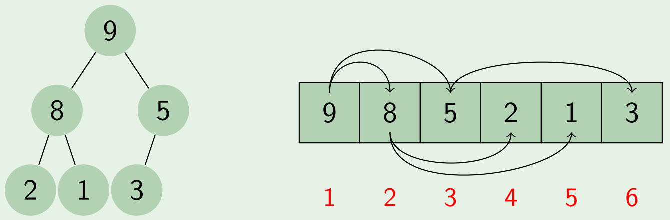
On suppose avoir accès à la dernière feuille du tas (celle la plus à droite)





Implantation d'un tas

- si le tas est utilisé pour un tri, alors le nombre d'éléments est borné
- on peut alors utiliser une structure statique plutôt qu'un structure dynamique



les fils d'un noeud représenté à l'indice i se trouvent en positions $2 \times i$ et $2 \times i + 1$

Implantation de la suppression

```
const
   MAX = 10;

type
   TAS = record
      taille : 0..MAX;
      letas  : array[1..MAX] of CARDINAL;
   end {record};

   // CU: le tas contient au moins un element
   procedure supprimer (var t : TAS; var m : CARDINAL);
   begin
      m := t.letas[1];
      t.letas[1] := t.letas[t.taille];
      t.taille := t.taille - 1;
      if t.taille >= 2 then reorganiser(t);
   end {supprimer};
```

Implantation de la réorganisation

```
procedure reorganiser (var t : TAS);
var
   g,d,max : CARDINAL;
   pere : CARDINAL
   fini : BOOLEAN = false;
begin
   pere := 1;
   repeat
      g := gauche(pere); // gauche(i) = 2*i
      d := droit(pere); // droit(i) = 2*i+1
      { recherche du maximum entre le fils gauche et droit }
      max := pere;
      if (g <= t.taille) and (t.letas[max] < t.letas[g]) then
         max := g;
      if (d <= t.taille) and (t.letas[max] < t.letas[d]) then
         max := d;
      { on s'arrete lorsqu'on ne peut plus descendre dans l'arbre}
      fini := pere = max;
      if not fini then
         echanger(t,pere,max);
      pere := max;
   until fini;
end {reorganiser};
```

Complexité de la réorganisation et de la suppression

Réorganisation :

- a chaque tour de boucle, dans le pire des cas, on descend dans l'arbre, et on a un échange d'éléments
- un tas a une hauteur $\lfloor \log n \rfloor$, la boucle est donc réalisé $\log n$ fois au maximum
- la complexité est donc en $\mathcal{O}(\log n)$

Suppression :

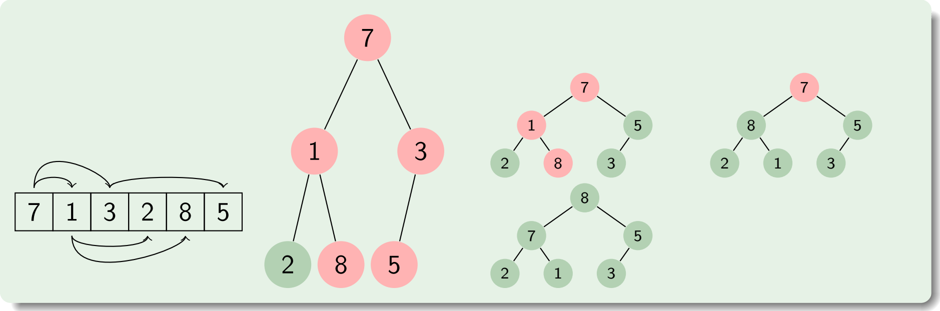
on a un échange d'éléments

$+$ une réorganisation

- la complexité est donc en $\mathcal{O}(\log n)$

Création d'un tas

- on sait comment extraire les valeurs successives d'un tas en partant de la plus grande
- comment construire le tas à partir d'un ensemble de valeurs quelconque ?



- la dernière moitié du tableau représente les feuilles
- idée : réorganiser les sous-arbres en commençant par les plus profonds

Modification de reorganiser

Paramètre indiquant le noeud racine du sous-arbre à réorganiser.

```
procedure reorganiser (var t : TAS; pere : CARDINAL);
var
   g,d,max : CARDINAL;
   fini : BOOLEAN = false;
begin
   repeat
      g := gauche(pere); // gauche(i) = 2*i
      d := droit(pere); // droit(i) = 2*i+1
      { recherche du maximum entre le fils gauche et droit }
      max := pere;
      if (g <= t.taille) and (t.letas[max] < t.letas[g]) then
         max := g;
      if (d <= t.taille) and (t.letas[max] < t.letas[d]) then
         max := d;
      { on s'arrete lorsqu'on ne peut plus descendre dans l'arbre}
      fini := pere = max;
      if not fini then
         echanger(t,pere,max);
      pere := max;
   until fini;
end {reorganiser};
```

Implantation de la création

```
function creer (a : array of CARDINAL) : TAS;
var
   t : TAS;
   i : CARDINAL;
begin
   t.taille := length(a);
   { copie des elements de a dans t }
   for i := 1 to t.taille do
      t.letas[i] := a[low(a)+i-1];
   { reorganisation de t }
   for i := t.taille div 2 downto 1 do
      reorganiser(t,i);
   creer := t;
end {creer};
```

Complexité de la création

En première approche :

une boucle sur la moitié du tableau : $\frac{n}{2}$

$\times$ chaque réorganisation est en $\mathcal{O}(\log n)$

- d'où une complexité en $\mathcal{O}(n \log n)$

Complexité de la création

Plus finement :

- à la construction, la réorganisation se fait sur des sous-arbres de hauteur différentes : de 0 à $\lfloor \log n \rfloor$
- nombre de sous-arbres de hauteur i : 2^{h-i}
- si i est la hauteur d'un sous-arbre, le coût de la réorganisation est $\mathcal{O}(i)$
- on en déduit que la complexité est

$$\sum_{i=0}^{h} 2^{h-i} \cdot \mathcal{O}(i) = \mathcal{O}\left(2^h \cdot \sum_{i=0}^{h} \frac{i}{2^i}\right) = \mathcal{O}\left(n \cdot \sum_{i=0}^{\infty} \frac{i}{2^i}\right) = \mathcal{O}(n \cdot 2) = \mathcal{O}(n)$$

avec $\sum_{i=0}^{\infty} \frac{i}{2^i} = \frac{\frac{1}{2}}{1-\frac{1}{2}} = 2$

Implantation du tri

```
procedure trier (var a : array of CARDINAL);
var
   t : TAS;
   v : CARDINAL;
begin
   t := creer(a);
   while t.taille >= 1 do begin
      { extraction de la valeur maximale de t dans v et reorganisaton }
      supprimer(t,v);
      { rangement de la valeur maximale dans a }
      a[t.taille] := v;
   end;
end {trier};
```

Complexité du tri

création en $\mathcal{O}(n)$

$+$ boucle sur la taille du tableau : n

$\times$ suppression de l'élément maximum en $\mathcal{O}(\log n)$

tri en $\mathcal{O}(n \log n)$