Algorithmique Avancée

Antoine Genitrini

Antoine.Genitrini@upmc.fr

Master Informatique 1

http://www-master.ufr-info-p6.jussieu.fr/2017

http://www-master.ufr-info-p6.jussieu.fr/2017/algav

Année 2017-2018

CHAPITRE 2

RECHERCHE ARBORESCENTE

Plan du Chapitre 2

- Arbres binaires de recherche
- Arbres équilibrés
  - AVL
  - Arbres 2-3-4
  - Arbres B
  - Arbres auto-adaptatifs
- Tries
  - Arbres digitaux
  - Arbres lexicographiques
  - Arbres hybrides

Problème de Recherche

Bases de données

- Ensemble d'éléments
  - chaque élément a une clé qui l'identifie
  - ordre total sur les clés ; calcul sur les clés ; accès aux bits des clés
- Opérations
  - *Rechercher* un élément
  - *Ajouter* un élément
  - *Supprimer* un élément
  - *Construction d'un ensemble d'éléments*
  - *Recherches partielles* (intervalle, préfixe commun, joker)

Structures concurrentes

- Arbres de Recherche
- Tries
- Hachage

Structures-Efficacité

Ensemble de n clés.
Chaque clé représentée sur $\leq L$ caractères.

Nombre de comparaisons pour une recherche/ajout

Structure	en Moyenne	au Pire	Mémoire
ABR	$O(\log n)$	$O(n)$	n + 2n ref.
ABR-Equilibré	$O(\log n)$	$O(\log n)$	n + 2n ref.
Arbre Digital [1]	$O(\log n)$	L	n + 2n ref.
Trie [1, 2]	$O(\log n)$	L	n + Ln ref.
Hachage [3]	$O(1)$	$O(n)$	n

1. accès aux caractères
2. comparaisons de caractères
3. calcul fonction de hachage

Plan du cours

Arbres binaires (rappels)

Définition : $\mathcal{B} = \emptyset + < \bullet, \mathcal{B}, \mathcal{B} >$: un *arbre binaire* est

- soit vide ($\emptyset$),
- soit constitué d'un nœud racine, d'un sous-arbre gauche qui est un arbre binaire et d'un sous-arbre droit qui est un arbre binaire.

Parcours : $\mathcal{B} \rightarrow$ Liste (sommets)   Soit $B =< \bullet, G, D >$

1. préfixe : PREF(B) = [visit($\bullet$),PREF(G), PREF(D)]
2. infixe (ou symétrique) : INF(B) = [INF(G), visit($\bullet$), INF(D)]
3. suffixe : SUF(B) = [SUF(G), SUF(D), visit($\bullet$)]

Définition : soit $B =< \bullet, G, D >$ un arbre binaire, le complété de B, noté $\bar{B}$ est l'arbre obtenu en remplaçant tous les sous-arbres vides $\emptyset$ par une feuille, notée $\square$.
Lemme : l'arbre complété $\bar{B}$ a n nœuds (internes) a $(n + 1)$ feuilles.
Preuve par induction.

Plan du cours

Arbres Binaires de Recherche

Définition : un ABR est un arbre binaire étiqueté tel que en chaque nœud, l'étiquette est plus grande que toutes les étiquettes du sous-arbre gauche, et plus petite que toutes les étiquettes du sous-arbre droit.

Propriété : le parcours infixe d'un ABR donne la suite des étiquettes en ordre croissant.
Preuve par induction.

Algorithmes de recherche, ajout et suppression :

- Parcours d'une branche
- Algorithmes simples : modifications minimes

Primitives sur les arbres binaires

```
def ArbreVide():
        """  -> ArbreBin
            Renvoie l'arbre vide."""
```

```
def ArbreBinaire(e, G, D):
        """ elt * ArbreBin * ArbreBin -> ArbreBin
            Renvoie l'arbre binaire dont la racine a pour contenu e,
            et pour fils gauche et droit, respectivement G et D."""
```

```
def EstArbreVide(A):
        """ ArbreBin -> booleen
            Renvoie vrai ssi l'arbre A est vide."""
```

Primitives sur les arbres binaires

```
def Racine(A):
        """ ArbreBin -> elt
            Renvoie le contenu de la racine de A."""
```

```
def SousArbreGauche(A):
        """ ArbreBin -> ArbreBin
            Renvoie une copie du sous-arbre gauche de l'arbre A."""
```

```
def SousArbreDroit(A):
        """ ArbreBin -> ArbreBin
            Renvoie une copie du sous-arbre droit de l'arbre A."""
```

```
def Pere(A):
        """ ArbreBin -> ArbreBin
            Renvoie l'arbre dont A est un des fils de la racine
            (ou l'arbre vide, si A n'est pas un sous arbre)."""
```

Opérations sur les ABR

Les algorithmes pour la Recherche, l'Ajout et la Suppression sont similaires.

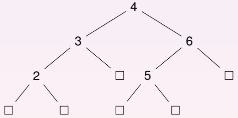
```
def ABR_Ajout(e, A):
        """ elt * ArbreBin -> ArbreBin
            Renvoie l'ABR resultant de l'ajout de e dans A."""

        if EstArbreVide(A):
           return ArbreBinaire(e, ArbreVide(), ArbreVide())
        elif e == Racine(A):
           return A
        elif e < Racine(A):
           return ArbreBinaire(Racine(A), ABR_Ajout(e, SousArbreGauche(A)),
                                          SousArbreDroit(A))
        else:
           return ArbreBinaire(Racine(A), SousArbreGauche(A),
                                          ABR_Ajout(e, SousArbreDroit(A)))
```

Arbres de recherche équilibrés

1. **Arbres binaires de recherche**
   - en moyenne hauteur en $O(\log n)$, mais au pire $O(n)$ (liste)
   - algorithmes Recherche, Ajout et Suppression : parcours d'une branche

2. **Arbres de recherche équilibrés**
   - hauteur toujours en $O(\log n)$
   - algorithmes Recherche, Ajout et Suppression : parcours d'une branche
   - algorithmes sophistiqués : modifications locales sur une branche (rotations, éclatements) pour maintenir hauteur en $O(\log n)$



Ajout de 1 dans l'ABR parfait

Assouplir contraintes sur forme des arbres en autorisant déséquilibre

- soit en hauteur → **Arbres AVL** ;
- soit en largeur → **Arbres B**.

Plan du cours

Arbres AVL

Définition d'un AVL

Un *AVL* (Adelson–Velsky, Landis) est un ABR t.q. en chaque nœud, la hauteur du sous-arbre gauche et celle du sous-arbre droit diffèrent au plus de 1.

Hauteur d'un AVL

Soit h la hauteur d'un AVL avec n nœuds :

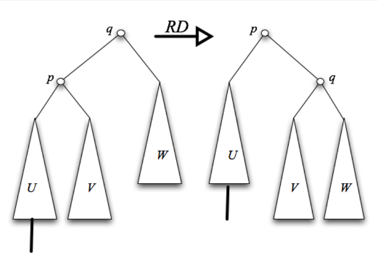
$$\log_2(n+1) \leq h+1 < 1.44 \log_2 n.$$

Au pire les arbres de Fibonacci :
$F_0 =< \bullet, \emptyset, \emptyset >, F_1 =< \bullet, F_0, \emptyset >, F_n =< \bullet, F_{n-1}, F_{n-2} >$

Rotations

Rotations pour rééquilibrer, en gardant les propriétés d'AVL

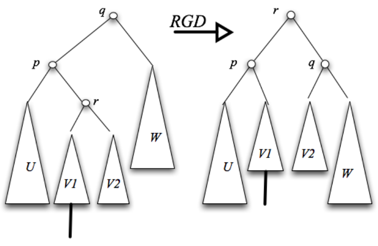
1. Rotation droite $A =< q, < p, U, V >, W >\Longrightarrow RD(A) =< p, U, < q, V, W >>$
propriété d'ABR : parcours infixe :
$\mathrm{INF}(A) = \mathrm{INF}(RD(A)) = \mathrm{INF}(U).p.\mathrm{INF}(V).q.\mathrm{INF}(W)$
hauteur : $h(U) = h(V) = h(W) = H - 2$. Arbre initial A : hauteur H et déséquilibre à gauche en q.
Si insertion aux feuilles de U, arbre résultant $RD(A)$: hauteur H et pas de déséquilibre, ni en p, ni en q.



2. Rotation gauche $A =< p, U, < q, V, W >>\Longrightarrow RG(A) =< q, < p, U, V >, W >$
même propriété que 1)

Rotations

3 Rotation gauche-droite : si $A =< q, < p, U, < r, V_1, V_2 >>, W >$ alors
$RGD(A) =< r, < p, U, V_1 >, < q, V_2, W >>$
propriété d'ABR : parcours infixe :
$\mathrm{INF}(A) = \mathrm{INF}(RDG(A)) = \mathrm{INF}(U).p.\mathrm{INF}(V_1).r.\mathrm{INF}(V_2).q.\mathrm{INF}(W)$
hauteur : $h(U) = h(V) = h(W) = H - 2$. Arbre initial A : hauteur H et déséquilibre à gauche en q.
Si insertion aux feuilles de $V =< r, V_1, V_2 >$, arbre résultant $RDG(A)$: hauteur H et pas de déséquilibre, ni en r, ni en p si insertion en V_1 (ou q si insertion en V_2).



4 Rotation droite-gauche : si $A =< q, W, < p, < r, V_1, V_2 >, U >>$ alors
$RDG(A) =< r, < q, W, V1 >, < p, V_2, U >>$ *(même propriété que 1))*

Opérations sur les AVL

```
def Hauteur(A):
        """ ArbreBin -> entier
            Renvoie la hauteur de l'arbre pris en argument."""
```

Les 4 fonctions de rotation : `RD`, `RG`, `RDG`, `RGD`.

```
def Equilibrage(A):
        """ ArbreBin -> AVL
            Hypotheses : A est un arbre de recherche,
                         les sous-arbres de A sont des AVL,
                         leurs hauteurs different au plus de 2.
            Renvoie l'arbre AVL obtenu en reequilibrant l'arbre initial."""
```

Les fonctions `Recherche`, `Ajout`, `Suppression`...

Ajout dans un AVL

```
def AVL_Ajout(x, A):
        """ elt * ArbreBin -> AVL
            Renvoie l'AVL resultant de l'ajout de x a A."""

        if EstArbreVide(A):
            return ArbreBinaire(x, ArbreVide(), ArbreVide())
        if x == Racine(A):
            return A
        if x < Racine(A):
            return Equilibrage(ArbreBinaire(Racine(A),
                                  AVL_Ajout(x, SousArbreGauche(A)),
                                  SousArbreDroit(A)))
        else:
            return Equilibrage(ArbreBinaire(Racine(A),
                                  SousArbreGauche(A),
                                  AVL_Ajout(x, SousArbreDroit(A))))
```

Arbre de recherche général

Arbre de recherche général

Dans un *arbre de recherche général*

- chaque nœud contient un k-uplet $(e_1 < \ldots < e_k)$ d'éléments distincts et ordonnés,
- et chaque nœud a $k + 1$ sous-arbres $A_1, \ldots, A_{k+1}$ tels que
  - tous les éléments de A_1 sont $< e_1$,
  - tous les éléments de A_i sont $> e_{i-1}$ et $< e_i$, pour $i = 2, \ldots, k$
  - tous les éléments de A_{k+1} sont $> e_k$

Arbres 2-3-4

Définition d'un arbre 2-3-4

Un *arbre 2-3-4* est un arbre de recherche

- dont les nœuds contiennent des k-uplets de soit 1, soit 2, soit 3 éléments,
- et dont toutes les feuilles sont situées au même niveau.

Hauteur d'un arbre 2-3-4

Soit h la hauteur d'un arbre 2-3-4 avec n éléments :

$$h = \Theta(\log n).$$

- arbre qui ne contient que des 2-nœuds : $h + 1 = \log_2(n + 1)$,
- vs. arbre qui ne contient que des 4-nœuds : $h + 1 = \log_4(n + 1)$

Primitives des Arbres 2-3-4

Notations :

2-nœud : $<(a), T_1, T_2>$ 3-nœud : $<(a, b), T_1, T_2, T_3>$, avec $a < b$
4-nœud : $<(a, b, c), T_1, T_2, T_3, T_4>$, avec $a < b < c$

```
def EstVide(A):
        """ A2-3-4 -> booleen """
```

```
def Degre(A):
        """ A2-3-4 -> entier entre 2 et 4 """
```

```
def Contenu(A):
        """ A2-3-4 -> liste  (de longueur 1 a 3) """
```

```
def EstDans(x, L):
        """ entier * liste -> booleen """
```

```
def Elem_i(A):
        """ A2-3-4 -> entier
            Renvoie le i-eme element du noeud racine(A) (sinon +infini)."""
```

```
def SsA_i(A):
        """ A2-3-4 -> A2-3-4
        Renvoie le i-eme sous-arbre de A (sinon l'arbre vide)."""
```

Algorithme de Recherche

```
def 234Recherche(x, A):
        """ entier * A2-3-4 -> booleen
            Renvoie Vrai ssi x est dans A."""

        if EstVide(A):
            return False
        if EstDans(x, Contenu(A)):
            return True
        if x < Elem_1(A):
            return 234Recherche(x, SsA_1(A))
        if x < Elem_2(A):
            return 234Recherche(x, SsA_2(A))
        if x < Elem_3(A):
            return 234Recherche(x, SsA_3(A))
        return 234Recherche(x, SsA_4(A))
```

Complexité en nombre de comparaisons : $O(\log n)$.

Ajout d'un élément

- Ajout aux feuilles (guidé par la recherche)
- un i-nœud se transforme en $(i+1)$-nœud, par insertion dans la liste
- sauf lorsque la feuille contient déjà 3 éléments ! ! !

Exemple : Construire par adjonctions successives un arbre 2-3-4 contenant les éléments

$$(4, 35, 10, 13, 3, 30, 15, 12, 7, 40, 20, 11, 6).$$

Deux méthodes de rééquilibrage

- Éclatements en remontée (au pire en cascade sur toute une branche)
- Éclatements en descente (éclatement systématique de tout 4-nœud)

Comparaison des méthodes

Les deux méthodes ne donnent pas forcément le même arbre.
Elles opèrent toutes les deux en $O(\log n)$ comparaisons (transformations sur une branche)

Avantages de la méthode d'éclatements en descente

- parcours de branche uniquement de haut en bas
- transformation très locale : accès parallèles possibles

Inconvénients de la méthode d'éclatements en descente

- taux d'occupation des nœuds plus faible
- hauteur de l'arbre plus grande

Éclatements en descente

Transformations de rééquilibrage locales : sur le chemin de recherche, on éclate systématiquement les 4-nœuds.

$\boxed{1}$ Le père du nœud à éclater ne peut pas être un 4-nœud.

$\boxed{2}$ Le père du nœud à éclater est un 2-nœud.
Si $P2 =< (x), A1, A2 >$, avec $A1 =< (a, b, c), U1, U2, U3, U4 >$, alors $P2 =< (b, x), < (a), U1, U2 >, < (c), U3, U4 >, A2 >$.

$\boxed{3}$ Le père du nœud à éclater est un 3-nœud.
Si $P3 =< (x, y), A1, A2, A3 >$ et
$A2 =< (a, b, c), U1, U2, U3, U4 >$, alors
$P3 =< (x, b, y), A1, < (a), U1, U2 >, < (c), U3, U4 >, A3 >$.

$\boxed{4}$ Les autres cas du $\boxed{2}$ et du $\boxed{3}$ sont analogues.

Ne modifie pas la profondeur des feuilles (sauf lorsque la racine de l'arbre éclate : la profondeur est alors augmentée de 1).

Algorithme d'ajout

```
def Ajout(x, A):
        """ entier * A2-3-4 -> A2-3-4
            Renvoie l'arbre 2-3-4 dans lequel x a ete ajoute."""

        if Degre(A) == 4:
            return AjoutSimple(x, EcR(A))
        return AjoutSimple(x, A)
```

```
def EcR(A):
        """ A2-3-4 -> A2-3-4
            Renvoie l'arbre 2-3-4 dans lequel la racine de A a ete eclatee."""
```

```
def AjoutSimple(x, A):
        """ entier * A2-3-4 -> A2-3-4
            Renvoie l'arbre 2-3-4 dans lequel x a ete ajoute a la racine."""

        if Degre(A) < 4:
                return AjoutSimple(x, Ui)
        if Degre(Pere(A)) == 2:
                return AjoutSimple(x, P2)
        return AjoutSimple(x, P3)
```

Les notations `Ui`, `P2` et `P3` sont celles du transparent 25.

Remarques

- Ui est le sous-arbre dans lequel doit se poursuivre l'ajout (comme pour une recherche)
- P2 est le transformé2 du père de A (cf. Éclatements)
- P3 est le transformé3 du père de A (cf. Éclatements)

- Complexité en nombre de comparaisons : $O(\log n)$
- Implantation : représentation des arbres 2-3-4 par des arbres binaires bicolores (voir TD).

Arbres-B

- Recherche Externe : éléments stockés sur disque (Mémoire Secondaire : MS ;
  MS paginée $\rightarrow$ allouer et récupérer les pages)
- Temps d'accès MS : 10^5 fois supérieur à Mémoire Principale (MP) : s'organiser pour avoir peu de transferts de pages.
- Arbres-B utilisés dans les Systèmes de Gestion de BD

Définition : Un *B-arbre d'ordre m* est un arbre de recherche

- dont les nœuds contiennent des k-uplets d'éléments, avec $m \leq k \leq 2m$,
- sauf la racine, qui peut contenir entre 1 et $2m$ éléments,
- et dont toutes les feuilles sont situées au même niveau

Hauteur d'un B-arbre d'ordre m contenant n éléments

$$\log_{2m+1}(n+1) \leq h+1 \leq 1+\log_{m+1}((n+1)/2).$$

Algorithmes et implantation

- Choisir m pour qu'un nœud tienne dans une page de MS.
- Prendre m grand : pour $m = 250$, l'arbre peut contenir plus de $125 \cdot 10^6$ éléments en ayant une hauteur 2.
- Algorithmes de recherche, ajout, suppression, analogues à ceux des arbres 2-3-4, avec nœuds d'arité $m + 1$ à $2m + 1$. Modifications sur une branche de l'arbre.
- Hauteur de l'arbre $\leq \log_{m+1}((n+1)/2)$.
  $\log_m(n) = \log_2 n / \log_2 m$: on gagne un facteur $\log_2 m$ par rapport aux arbres équilibrés précédents.
- Seul le nœud racine est en MP $\Longrightarrow$ nombre d'accès à la mémoire secondaire $\leq$ hauteur de l'arbre.
- 1 éclatement $\Rightarrow$ écrire 2 pages en MS (et le nombre d'éclatements est borné par la hauteur).

Analyse de complexité

Analyse amortie : méthode par agrégat

Le nombre d'éclatements par clé, dans la construction par adjonctions successives d'un B-arbre d'ordre m est compris entre $1/(2m)$ et $1/m$.

Preuve : au départ l'arbre est vide donc chaque nœud de l'arbre est le résultat d'un éclatement. Si l'arbre contient n clés au final, le nombre total ν de nœuds vérifie $2m\nu = n$ si tous les nœuds sont pleins, et $m\nu + 1 = n$ si les nœuds sont remplis au minimum, donc le nombre total d'éclatements est E tq. $n/2m < E < n/m - 1/m < n/m$.

Autres résultats d'analyse

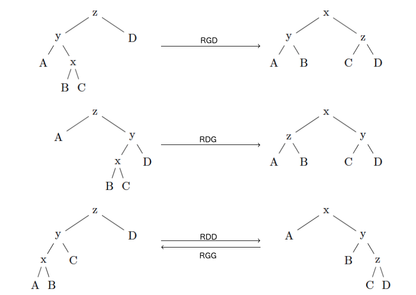
- Analyse de frange : 1 éclatement pour $1.38m$ adjonctions
- Un B-arbre d'ordre m contenant n éléments aléatoires comporte $1.44n/m$ nœuds.

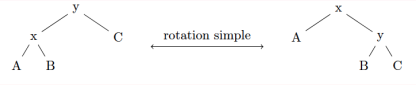
Arbres auto-adaptatifs (Splay-trees)

Définition

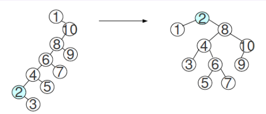
Un *arbre auto-adaptatif* est un arbre binaire de recherche tel que à chaque recherche, ajout, suppression, le *dernier* élément visité est *remonté à la racine* par une suite de rotations.

Suite de doubles rotations, sauf éventuellement la dernière (si x à prof impaire)

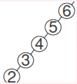




Ex : rechercher 2 dans l'arbre de gauche :



Ex : Insérer successivement 2,3,4,5,6



insérer 1 dans l'arbre obtenu

rôle des rotations doubles (vs rotations simples uniquement)

Coût d'une opération de remontée : analyse amortie

Fonction de potentiel d'un arbre T

Poids de $x \in T$: $w(x)$ = nb nœuds du ss-ab de racine x (x inclus).
Potentiel de T : $\Phi(T) = \sum_{x \in T} r(x)$, avec $r(x) = \log_2 w(x)$, *rang* de x.

Exemple : potentiel d'un arbre de 15 nœuds : filiforme $= \log(15!) = 40.25$
et complètement équilibré $= \log 15 + 2\log 7 + 4\log 3 = 12,21$
Pour un arbre T de taille n : $\Theta(n) < \Phi(T) < \Theta(n \log n)$

Coût amorti des rotations : pour chaque type de rotation on considère

- T' résultat de T par la rotation ;
- $w(x)$ poids de x dans T et $w'(x)$ poids de x dans T' ;
- $r(x)$ rang de x dans T et $r'(x)$ rang de x dans T'.

On montre que :

- Coût amorti d'un "RD" sur x : $\hat{c}_{RD} \leq r'(x) - r(x) + 1$
- Coût amorti d'un "RGD" sur x : $\hat{c}_{RGD} \leq 2(r'(x) - r(x))$
- Coût amorti d'un "RDD" sur x : $\hat{c}_{RDD} \leq 3(r'(x) - r(x))$

Coût amorti des rotations

- $\hat{c}_{RD} = 1 + \Phi(T') - \Phi(T) = 1 + r'(x) + r'(y) - (r(x) + r(y)) =$
  $1 + (r'(x) - r(x)) + (r'(y) - r(y))$
  $\leq 1 + r'(x) - r(x)$ (car $r'(y) \leq r(y)$)
  $\leq 1 + 3(r'(x) - r(x))$ (car $r'(x) \geq r(x)$)

- $\hat{c}_{RDD} = 2 + \Phi(T') - \Phi(T) = 2 + r'(x) + r'(y) + r'(z) - (r(x) + r(y) + r(z))$
  $= 2 + r'(y) + r'(z) - (r(x) + r(y))$ (car $r'(x) = r(z)$)
  $\leq 2 + r'(x) + r'(z) - 2r(x)$ (car $r'(y) \leq r'(x)$ et $r(y) \geq r(x)$)
  $\leq 2 + r(x) + r'(z) + r'(x) - 3r(x)$
  Convexité du logarithme : $\forall a, b, c \in R^+, a + b \leq c \implies \log a + \log b \leq 2 \log c - 2$
  donc ici : $w(x) + w'(z) \leq w'(x) \implies r(x) + r'(z) \leq 2r'(x) - 2$
  d'où finalement $\hat{c}_{RDD} \leq 3(r'(x) - r(x))$.

- $\hat{c}_{RGD} = 2 + \Phi(T') - \Phi(T)$
  $\leq 2 + r'(y) + r'(z) - 2r(x)$ (car $r(z) = r'(x)$ et $r(y) \geq r(x)$)
  Convexité du logarithme : $w'(y) + w'(z) \leq w'(x) \implies r'(y) + r'(z) \leq 2r'(x) - 2$
  d'où finalement $\hat{c}_{RGD} \leq 2(r'(x) - r(x)) \leq 3(r'(x) - r(x))$.

Coût amorti

- Coût amorti d'une remontée (= suite de rotations)
  Soient $T_1, \ldots, T_n$ les arbres obtenus à chaque rotation d'une remontée, et $r_i(x)$ le rang de x dans l'arbre T_i, le coût amorti de la remontée est

$$\hat{c} \leq 1 + \sum_{i=1}^{n-1} 3(r_{i+1}(x) - r_i(x)) = 1 + 3(r_n(x) - r_1(x)) \leq 1 + \log n = O(\log n)$$

- Pour une recherche positive, le coût amorti est celui de la remontée, et il en est de même pour une insertion ou une suppression (remonter l'élément immédiatement plus petit ou plus grand).

Conclusion

- Une opération particulière peut avoir un coût $O(n)$;
- mais $\forall$ suite de m opérations, le coût est $O(m \log n)$.
  (Les rotations effectuées lors d'une opération coûteuse permettent d'accélérer les opérations futures.)

Plan du cours

Tries

Ensemble de clés $\mathcal{S} \in A^*$. On a *accès aux caractères des clés*.

Trie = Représentation arborescente d'un ensemble de clés en évitant de répéter les préfixes communs.
Exemple : complétion automatique, vérificateur d'orthographe, ...

Opérations sur $\mathcal{S}$: Rechercher, Insérer, Supprimer une clé
Opérations complémentaires :

- Liste des clés de $\mathcal{S}$ en ordre alphanumérique
- Recherches partiellement spécifiées
- Recherche du plus long préfixe dans $\mathcal{S}$, d'un mot donné

Primitives sur les clés des tries

```
def prem(cle):
    """ S -> str
        Renvoie le premier caractere de la cle."""
```

```
def reste(cle):
        """ S -> str
            Renvoie la cle privee de son premier caractere."""
```

```
def car(cle, i):
        """ S * entier -> str
            Renvoie le i-eme caractere de la cle."""
```

```
def lgueur(cle):
        """ S -> entier
        Renvoie le nombre de caracteres de la cle."""
```

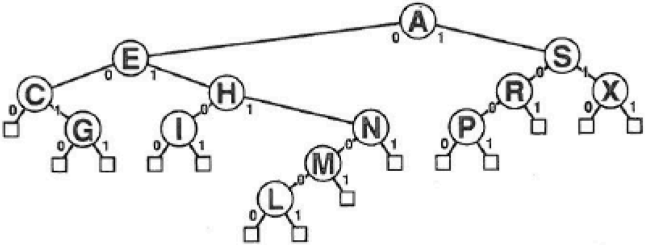
On utilise par ailleurs les primitives analogues à celles vues précédemment sur les arbres pour les tries.

Arbres Digitaux

Définition

Un Arbre Digital (DST) est un arbre binaire dont les nœuds contiennent des clés. Le principe de la recherche est le même que pour les ABR, mais l'aiguillage se fait non pas par comparaison entre clés, mais selon les bits (caractères) de la clé cherchée (1er bit au 1er niveau, 2eme bit au 2eme niveau, . . .).

A	0 0 0 0 1
S	1 0 0 1 1
E	0 0 1 0 1
R	1 0 0 1 0
C	0 0 0 1 1
H	0 1 0 0 0
I	0 1 0 0 1
N	0 1 1 1 0
G	0 0 1 1 1
X	1 1 0 0 0
M	0 1 1 0 1
P	1 0 0 0 0
L	0 1 1 0 0



Ajout dans un Arbre Digital

```
def DST_Ajout(c, A):
        """ S * DST -> DST
            Renvoie l'arbre digital resultant de l'insertion de c."""
        return DST_Ajout_rang(c,1,A)
```

```
def DST_Ajout_rang(c, i, A):
        """ S * entier * DST -> DST
            Renvoie l'arbre digital resultant de l'insertion de c,
            en considerant le bit i."""
        if EstArbreVide(A):
            return ArbreBinaire(c, ArbreVide(), ArbreVide())
        if x == Racine(A):
            return A
        if car(c, i) == 0:     #la cle est binaire
            return ArbreBinaire(Racine(A), DST_Ajout_rang(c, i+1,
                                       SousArbreGauche(A)), SousArbreDroit(A))
        else:
            return ArbreBinaire(Racine(A), SousArbreGauche(A),
                                       DST_Ajout_rang(c, i+1, SousArbreDroit(A)))
```

Propriété

La recherche ou l'ajout d'une clé dans un DST contenant n clés, nécessite en moyenne $\log n$ **comparaisons de clés**, et au pire L (nb max de bits d'une clé) accès aux bits (caractères).

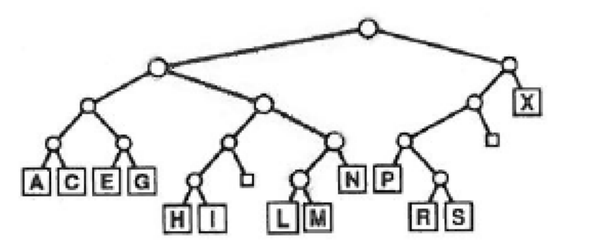
Arbres Lexicographiques (Tries binaires)

Clés de l'ensemble à représenter = suites de bits
(et deux clés ne sont jamais préfixe l'une de l'autre).

Définition

Un trie binaire est un arbre binaire dont les nœuds internes servent d'aiguillage : un nœud interne à profondeur d, dont le chemin depuis la racine est $w \in \{0, 1\}^*$, contient dans son sous arbre gauche toutes les clés commençant par $w0$ et dans son sous arbre droit toutes les clés commençant par $w1$. Les feuilles contiennent les clés.

A	00001
S	10011
E	00101
R	10010
C	00011
H	01000
I	01001
N	01110
G	00111
X	11000
M	01101
P	10000
L	01100



C_1 et C_2 feuilles sœurs à prof k

$\Longrightarrow C_1 = pref0$ et $C_2 = pref1$,

avec $|pref| = k - 1$

Ajout dans un trie binaire (1/2)

```
def TrieBin_Ajout(c, A):
        """ S * TrieBin -> TrieBin
            Renvoie le trie binaire resultant de l'insertion de c."""
        return TrieBin_Ajout_rang(c, 1, A)
```

```
def TrieBin_Ajout_rang(c, i, A):
        """ S * entier * TrieBin -> TrieBin
        Renvoie le trie binaire resultant de l'insertion de c,
        en considerant le bit i."""
        if EstArbreVide(A):
            return ArbreBinaire(c, ArbreVide(), ArbreVide())
        if EstArbreVide(SousArbreGauche(A)) and EstArbreVide(SousArbreDroit(A)):
            if c == Racine(A):
                return A
            else:
                return Split(c, Racine(A), i+1)
        if car(c, i) == 0:
                return ArbreBinaire(Racine(A), TrieBin_Ajout_rang(c, i+1,
                                SousArbreGauche(A)), SousArbreDroit(A))
        else:
                return ArbreBinaire(Racine(A), SousArbreGauche(A),
                            TrieBin_Ajout_rang(c, i+1, SousArbreDroit(A)))
```

Ajout dans un trie binaire (2/2)

```
def split(c, d, i):
        """ S * S * entier -> TrieBin
            Retourne le trie binaire contenant c et d."""

        if car(c, i) == car(d, i) == 0:
            return ArbreBinaire(CleVide(), Split(c, d, i+1), ArbreVide())
        if car(c, i) == car(d, i) == 1:
            return ArbreBinaire(CleVide(), ArbreVide(), Split(c, d, i+1))
        if car(c, i) == 0 and car(d, i) == 1:
            return ArbreBinaire(CleVide(), ArbreBinaire(c, ArbreVide(),
                        ArbreVide()), ArbreBinaire(d, ArbreVide(), ArbreVide()))
        if car(c, i) == 1 and car(d, i) == 0:
            return ArbreBinaire(CleVide(), ArbreBinaire(d, ArbreVide(),
                        ArbreVide()), ArbreBinaire(c, ArbreVide(), ArbreVide()))
```

Propriété

La recherche ou l'ajout d'une clé dans un Trie contenant n clés, nécessite en moyenne $\log n$ **comparaisons de bits**, et au pire L (nbre max de bits d'une clé) comparaisons de bits.

Problème des branches filiformes ($1.44n$ nœuds en moyenne) $\rightarrow$ Patricia tries

n nœuds

R-Trie

Clés = suites de caractères $\in$ alphabet de taille R.

Définition

Un R-trie est un arbre dont tous les nœuds sont d'arité R et servent d'aiguillage. Chaque nœud a aussi une valeur, qui est non vide lorsqu'il représente une clé
(possibilité ensembles qui ne sont pas “sans préfixe”).

Recherche, adjonction et suppression par parcours d'une branche.

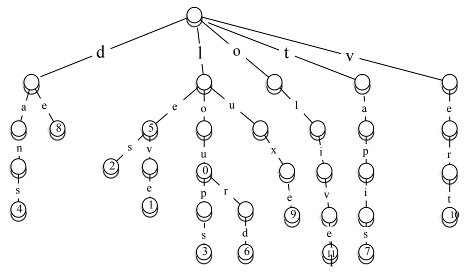
Propriété

La construction ne dépend pas de l'ordre d'insertion.

Remarque : 2-Trie $\neq$ Trie binaire !

Exemple de R-Trie

Exemple : lou, leve, les, loups, dans, le, lourd, tapis, de, luxe, vert, olive
Alphabet de 26 lettres -> nœuds d'arité 26
(les liens vides ne sont pas représentés)



Primitives sur les R-Tries

R-trie = 1 champ valeur (ex numéro d'insertion) + R liens vers des R-tries.

```
def TrieVide():
        """  -> R-trie
            Renvoie le trie a 1 noeud vide avec R liens vides."""
```

```
def EstVide(A):
        """ R-trie -> booleen
            Renvoie vrai ssi A est vide."""
```

```
def Val(A):
        """ R-trie -> elt
            Renvoie la valeur de la racine du trie."""
```

```
def SousArbre(A, i):
        """ R-trie * entier -> R-trie
            Renvoie une copie du i-eme sous-arbre de A."""
```

```
def FilsSauf(A, i):
        """ R-trie * entier -> liste[R-trie]
            Renvoie la liste des sous-arbres du trie privee du i-eme sous-arbre.
```

```
def R_Trie(i, L, A):
        """ entier * liste[R-trie] * R-trie -> R-trie
            Renvoie le trie construit a partir de L,
            en inserant A a la i-eme position."""
```

Ajout dans un R-Trie

```
def R_Trie_Ajout(c, A, v):
        """ cle * R-Trie * valeur -> R-trie
            Renvoie le trie resultant de l'insertion de c dans A."""

        if EstVide(A):
            A = TrieVide()
        if lgueur(c) == 0:
            StockVal(A) = v     # enregistre v dans la racine de A
            return A
        p = prem(c)
        return R_Trie(p, tousFilsSauf(p),
                      R_Trie_Ajout(reste(c), SousArbre(A, p), v))
```

Propriété

La recherche ou l'ajout d'une clé dans un R-Trie contenant n clés, nécessite en moyenne $\log n$ **comparaisons de caractères**, et au pire L (nb max de caractères d'une clé) comparaisons de caractères.

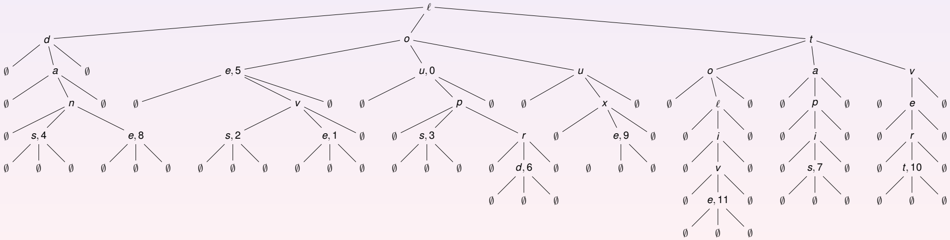
Problème : Place mémoire n nœuds avec R pointeurs chacun (dont de nombreux vides).

Tries Hybrides

Pour éviter perte de place des liens vide dans les R-Trie

Définition

Un trie hybride est un arbre ternaire dont chaque nœud contient un caractère et une valeur (non vide lorsque le nœud représente une clé). Chaque nœud a 3 pointeurs : un lien Inf (resp. Eq, resp. Sup) vers le sous arbre dont le premier caractère est inférieur (resp. égal, resp. supérieur) à son caractère.



Ajout dans un Trie Hybride

```
def TH_Ajout(c, A, v):
        """ cle * TrieH * valeur -> trieH
                Renvoie le trie hybride resultant de l'insertion de c dans A."""

        if EstVide(A):
                if lgeur(c) == 1:
                    return TrieH(prem(c), TH_Vide(), TH_Vide(), TH_Vide(), v)
                else:
                    return TrieH(prem(c), TH_Vide(),
                           TH_Ajout(reste(c), Eq(A), v), TH_Vide(), ValVide())
        else:
            p = prem(c)
            if p < Rac(A):
                return TrieH(Rac(A), TH_Ajout(c, Inf(A), v),
                           Eq(A), Sup(A), Val(A))
            if p > Rac(A):
                return TrieH(Rac(A), Inf(A), Eq(A),
                           TH_Ajout(c, Sup(A), v), Val(A))
            return TrieH(Rac(A), Inf(A), TH_ajout(reste(c), Eq(A), v),
                           Sup(A), Val(A))
```

Propriété

La recherche ou l'ajout d'une clé dans un Trie Hybride contenant n clés, nécessite en moyenne L+ log n **comparaisons de caractères**, et au pire $2L$ (2 fois le nb max de caractères d'une clé) comparaisons de caractères.
La place mémoire occupée est de $2n + 3n$ références.