Algorithmique — 1. Structures de données

3. Tableaux dynamiques et arbres binaires de recherche

Bruno Grenet

https://membres-ljk.imag.fr/Bruno.Grenet/Algorithmique.html

Université Grenoble Alpes – IM²AG
L3 Mathématiques et Informatique

Table des matières

Table des matières

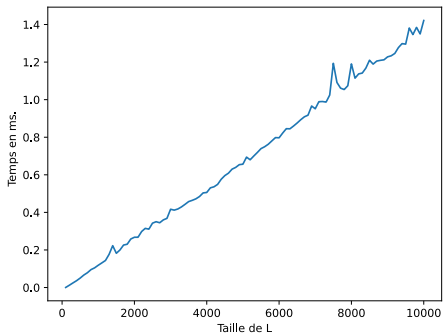
Objectifs

- Combiner l'accès en temps $O(1)$ des tableaux…
- … avec le caractère *dynamique* des listes

- Réalisation des TAD Pile, File, File de priorité

Exemples des `list` Python

```
def test(n):
  L = []
  for i in range(n):
    L.append(i)
  for i in range(1,n):
    L[i],L[n-i-1] = L[n-i-1],L[i]
```



Le TAD tableau dynamique

Opérations

- NVTABLEAUDYNAMIQUE(n), TAILLE($\mathcal{T}$) et $\mathcal{T}_{[i]}$: identiques à un tableau standard
- AJOUTER($\mathcal{T}, x$) : ajoute l'élément x à la fin de $\mathcal{T}$ et met à jour sa taille
- SUPPRIMER($\mathcal{T}$) : supprime le dernier élément de $\mathcal{T}$ et met à jour sa taille

Objectifs

- Complexité *amortie* $O(1)$ pour chaque opération
- Espace mémoire $O(n)$ où $n =$ TAILLE($\mathcal{T}$)

Principes de la réalisation

- Un tableau de taille N pour représenter n éléments $N \geq n$
- Si le tableau devient trop petit : créer un nouveau tableau plus grand
- Ne pas gâcher d'espace : si le tableau est trop grand, en créer un nouveau plus petit

Réalisation du TAD tableau dynamique

Représentation

- Deux données : $\mathcal{T} = (T, n)$ avec
  - T : tableau (*physique*) T de taille N — TAD tableau
  - n : taille (*utile*) de $\mathcal{T}$ — TAD entier
- Éléments stockés en cases $T_{[0]}, \ldots, T_{[n-1]}$
- Règle à respecter : $n \leq N < 4n$

Opérations identiques à un tableau standard : $O(1)$

- NVTABLEAUDYNAMIQUE : crée T de taille n et renvoie $\mathcal{T} = (T, n)$
- Accès en lecture/écriture : $\mathcal{T}_{[i]}$ renvoie $T_{[i]}$
- TAILLE($\mathcal{T}$) renvoie n

Remarques pour NVTABLEAUDYNAMIQUE

- Pour simplifier, on suppose qu'on crée toujours T de taille $N \geq 1$ (même si $n = 0$)
- On peut aussi choisir $N = 2n$ initialement

Ajout et suppression

Ajout d'un élément x à la fin

1. Si $N \leq n$, il faut créer de la place :
   - Créer un nouveau tableau S de taille $2N$
   - Recopier les N cases de T dans S
   - Remplacer T par S
2. Ajouter x en case n de T et incrémenter n

Suppression du dernier élément

- Presque rien à faire : $n \leftarrow n - 1$!
- Pour garder $N \leq 4n$, il faut (parfois) réduire la taille de T
  - Idée 1 : si $n < N/2$ on réduit de moitié $\rightarrow$ mauvaise idée ! *pourquoi ?*
  - Idée 2 : si $n < N/4$ on réduit de moitié $\rightarrow$ bonne idée !

Les algorithmes

AJOUTER$(\mathcal{T}, x)$:

1. $(T, n) \leftarrow \mathcal{T}$; $N \leftarrow$ TAILLE(T)
2. Si $n < N$:
3. $\quad T_{[n]} \leftarrow x$
4. $\quad$ Renvoyer $(T, n+1)$
5. $S \leftarrow$ NVTABLEAU$(2N)$
6. Pour $i = 0$ à $N-1$: $S_{[i]} \leftarrow T_{[i]}$
7. $S_{[n]} \leftarrow x$
8. Renvoyer $(S, n+1)$

SUPPRIMER$(\mathcal{T})$:

1. $(T, n) \leftarrow \mathcal{T}$; $N \leftarrow$ TAILLE(T)
2. Si $n = 1$ ou $n - 1 \geq N/4$:
3. $\quad$ Renvoyer $(T, n-1)$
4. $S \leftarrow$ NVTABLEAU$(N/2)$
5. Pour $i = 0$ à $n-2$: $S_{[i]} \leftarrow T_{[i]}$
6. Renvoyer $(S, n-1)$

Dans le pire cas, AJOUTER et SUPPRIMER effectuent chacun $O(n)$ affectations

Analyse amortie : cas simplifié

Coût de m ajouts dans un tableau dynamique initialement vide ?

Analyse pire cas

- Un ajout dans un tableau de taille k effectue $\leq k + 1$ affectations
- Coût total :

Théorème (analyse *amortie*)

Le nombre total d'affectations pour m ajouts est $\leq 3m$, donc ≤ 3 *par ajout*

Preuve

Analyse amortie : cas général avec m opérations AJOUTER/SUPPRIMER

Analyse pire cas

- Une opération sur un tableau de taille k effectue $\leq k+1$ affectations $\rightarrow$ coût $O(m^2)$

Comment analyser le coût globalement ?

- Après un redimensionnement, $n \simeq N/2$
  - Avant le prochain agrandissement de T, $\geq n$ ajouts
  - Avant le prochain rétrécissement de T, $\geq n/2$ suppressions
- Le coût des redimensionnements est *amorti* par le nombre d'opérations

Une méthode de preuve : la fonction potentiel

- On attribue un potentiel $\Phi_i \in \mathbb{Z}_{\geq 0}$ à $\mathcal{T}$ après i opérations
- À chaque opération : coût *amorti* $a_i = c_i + (\Phi_i - \Phi_{i-1})$ $\quad c_i$: *vrai* coût

$$\text{coût total} = \sum_{i=1}^{m} c_i = \sum_{i=1}^{m} (a_i + \Phi_{i-1} - \Phi_i) = \left(\sum_{i=1}^{m} a_i \right) + (\Phi_0 - \Phi_m)$$

Analyse amortie : définition du potentiel

Fonction potentiel

$$\Phi_i = \begin{cases} 2n_i - N_i & \text{si } n_i \geq N_i/2 \\ N_i/2 - n_i & \text{si } n_i \leq N_i/2 \end{cases}$$

où

- n_i : nombre d'élément dans $\mathcal{T}$ après la $i^{\text{ème}}$ opération
- N_i : taille de T après la $i^{\text{ème}}$ opération

Preuve de l'analyse amortie

Lemme

Le coût amorti $a_i = c_i + \Phi_i - \Phi_{i-1}$ de la $i^{\text{ème}}$ opération est ≤ 3 pour tout i

Preuve de l'analyse amortie

Lemme

Le coût amorti $a_i = c_i + \Phi_i - \Phi_{i-1}$ de la $i^{\text{ème}}$ opération est ≤ 3 pour tout i

Corollaire

Les opérations AJOUTER & SUPPRIMER ont un coût amorti $O(1)$ (≤ 3)

Bilan sur les tableaux dynamiques

Principes

- Tableau de taille variable
  - Mémoire *allouée* supérieure à celle utilisée
  - Taux de remplissage entre $\frac{1}{4}$ et 1
  - Taille doublée ou divisée par deux quand nécessaire
- Accès direct et AJOUTER/SUPPRIMER *en fin de tableau* en temps *amorti* constant

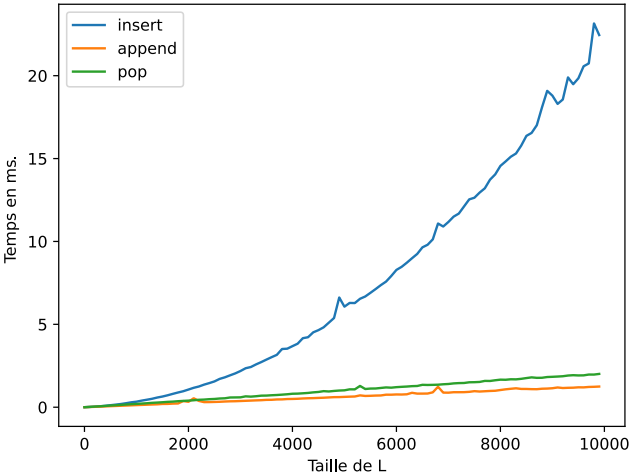
Complexité amortie

- m opérations coûtent $\leq 3m$ affectations $\rightarrow$ coût *amorti* constant
- Mais tout de même : si on connaît à l'avance la taille, coût triplé !

Pour réaliser les Pile, File et File de priorité

- Remplacement du tableau par un tableau dynamique
- Plus de souci de taille maximale fixée
- Complexités *amorties* aussi bonnes

Performance des `list` Python



- Insertion en début de tableau
- Insertion en fin de tableau
- Suppression en fin de tableau

Table des matières

Dictionnaires

(ou tableau associatif, ou table de symboles)

Définition informelle

- Structure dynamique pour représenter un ensemble de couples (clé, valeur)
- Une clé ne peut être présente qu'une seule fois
- Accès rapide à la valeur associée à une clé

Exemples

- *dictionaries* en Python, .NET, Swift, C#, …
- *maps* en C++, Java, OCaml, Haskell, …
- *tables* en Lua, Maple, …
- *arrays* en PHP

Quelques utilisations

- (*vrai*) dictionnaire, annuaire, …
- types et valeurs des variables pour un compilateur
- base de données type NoSQL

Le TAD Dictionnaire

Opérations

- nvDictionnaire() : crée un dictionnaire vide
- insérer(D, c, v) : ajoute c à D avec valeur v ; si c déjà présente, remplace la valeur
- supprimer(D, c) : supprime c de D si c est dans D ; *erreur* sinon
- rechercher(D, c) : renvoie la valeur associée à c si c est dans D ; *erreur* sinon

Remarque

- On peut ajouter facilement un test d'appartenance : est-ce que $c \in D$?
- Notations allégées :
  - $D_{[c]} \leftarrow v$ pour insérer(D, c, v)
  - $D_{[c]}$ pour rechercher(D, c)

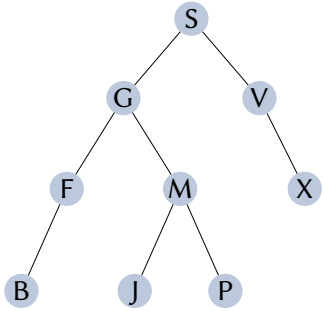
Principales réalisations

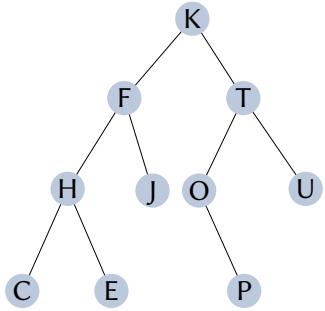
- Arbres binaires de recherche et extensions
- Tables de hachage

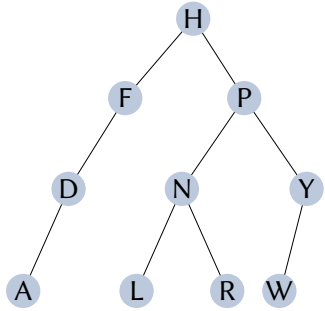
Définition

Un **arbre binaire de recherche** (ABR) est un arbre binaire dont chaque nœud possède une clé, qui est

- supérieure aux clés des nœuds de son sous-arbre gauche
- inférieure aux clés des nœuds de son sous-arbre droit







Réalisation du TAD Dictionnaire avec un ABR

Représentation des données

- Un arbre binaire de recherche $\mathcal{A}$
- Chaque nœud possède aussi une valeur
  - Un nœud est un couple (clé, valeur)
  - Structure d'ABR sur les clés
- nvDictionnaire() $\rightsquigarrow$ nvArbre()

À résoudre

- rechercher une clé donnée
- insérer une nouvelle clé
- supprimer une clé

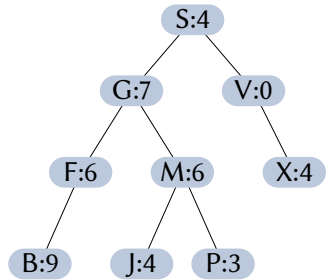
Remarque

- On manipule les nœuds *via* leurs clés $\rightsquigarrow$ « clé de racine($\mathcal{A}$) »

Recherche dans un ABR

RECHERCHER($\mathcal{A}, c$):

1. Si ESTVIDE($\mathcal{A}$) : renvoyer une erreur (*non trouvé*)
2. $c' \leftarrow$ clé de RACINE($\mathcal{A}$)
3. Si $c = c'$: renvoyer v
4. Si $c < c'$: renvoyer RECHERCHER(ENFANTG($\mathcal{A}$), c)
5. renvoyer RECHERCHER(ENFANTD($\mathcal{A}$), c)



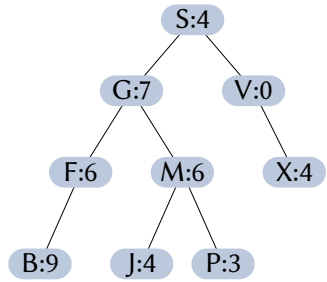
Théorème

RECHERCHER($\mathcal{A}, c$) est correct et a une complexité $O(h_{\mathcal{A}})$ où $h_{\mathcal{A}} =$ hauteur de $\mathcal{A}$.

Insertion d'un élément

insérer($\mathcal{A}, c, v$):

1. Si estVide($\mathcal{A}$) :
2. $\quad \mathcal{A} \leftarrow$ nvArbre($(c, v), \emptyset, \emptyset$)
3. Sinon :
4. $\quad c' \leftarrow$ clé de racine($\mathcal{A}$)
5. $\quad (\mathcal{G}, \mathcal{D}) \leftarrow$ (enfantG($\mathcal{A}$), enfantD($\mathcal{A}$))
6. $\quad$ Si $c' = c$: $\mathcal{A} \leftarrow$ nvArbre($(c, v), \mathcal{G}, \mathcal{D}$)
7. $\quad$ Sinon si $c < c'$: insérer($\mathcal{G}, c, v$)
8. $\quad$ Sinon : insérer($\mathcal{D}, c, v$)



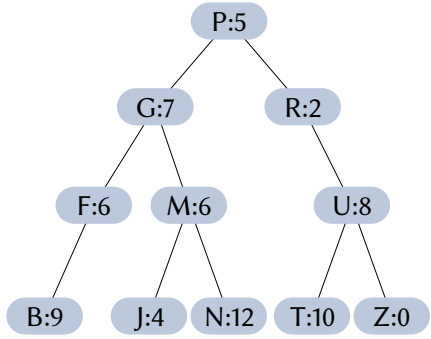
Théorème

insérer a une complexité $O(h_{\mathcal{A}})$ et conserve la structure d'ABR

Suppression d'un élément

Exemple

1. Supprimer J
2. Supprimer F
3. Supprimer P



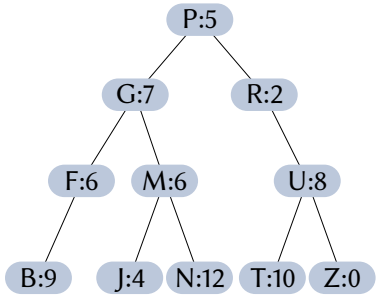
Algorithme pour supprimer le nœud N de clé c

1. Si N n'a pas d'enfant : le supprimer
2. Si N a un seul enfant : le remplacer par son unique enfant
3. Sinon : remplacer N par le minimum de son sous-arbre droit

Trouver et supprimer le minimum

supprimerMin($\mathcal{A}$):

1. Si enfantG($\mathcal{A}$) $= \emptyset$:
2. $\quad (c, v) \leftarrow$ racine($\mathcal{A}$)
3. $\quad \mathcal{A} \leftarrow$ enfantD($\mathcal{A}$)
4. $\quad$ Renvoyer (c, v)
5. Renvoyer supprimerMin(enfantG($\mathcal{A}$))



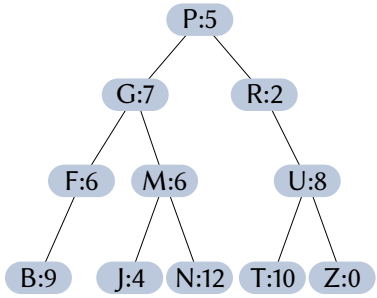
Lemme

supprimerMin a une complexité $O(h_{\mathcal{A}})$ et est correct

Algorithme de suppression

SUPPRIMER($\mathcal{A}, c$):

1. Si ESTVIDE($\mathcal{A}$) : renvoyer une erreur (*non trouvé*)
2. $c' \leftarrow$ clé de RACINE($\mathcal{A}$)
3. $(\mathcal{G}, \mathcal{D}) \leftarrow$ (ENFANTG($\mathcal{A}$), ENFANTD($\mathcal{A}$))
4. Si $c = c'$:
5. Si $\mathcal{G} = \emptyset$: $\mathcal{A} \leftarrow \mathcal{D}$
6. Sinon si $\mathcal{D} = \emptyset$: $\mathcal{A} \leftarrow \mathcal{G}$
7. Sinon :
8. $(c_m, v_m) \leftarrow$ SUPPRIMERMIN($\mathcal{D}$)
9. $\mathcal{A} \leftarrow$ NVARBRE($(c_m, v_m), \mathcal{G}, \mathcal{D}$)
10. Sinon si $c < c'$: SUPPRIMER($\mathcal{G}, c$)
11. Sinon : SUPPRIMER($\mathcal{D}, c$)



Théorème

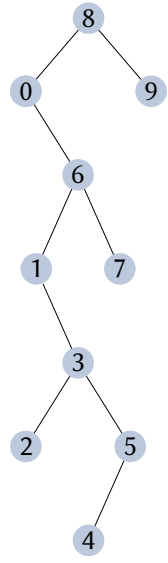
SUPPRIMER a une complexité $O(h_{\mathcal{A}})$, et conserve la structure d'ABR de $\mathcal{A}$

Quelle efficacité alors ?

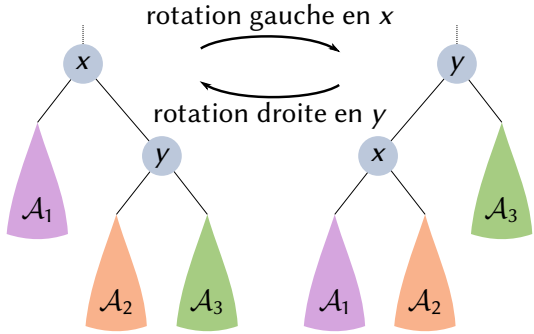
Rappel des complexités

- RECHERCHER, INSÉRER et SUPPRIMER : $O(h_{\mathcal{A}})$
- Dans un arbre binaire, $\lfloor \log \#\mathcal{A} \rfloor \leq h_{\mathcal{A}} \leq \#\mathcal{A}$

Un ABR est une structure de donnée efficace s'il est équilibré, c'est-à-dire si $h_{\mathcal{A}} = O(\log(\#\mathcal{A}))$.



Outil de base : les rotations



Théorème

Les rotations s'effectuent en temps $O(1)$ et conservent la structure d'ABR

Utilisation des rotations

- Augmentation de la hauteur d'un côté, diminution de l'autre $\rightsquigarrow$ meilleur équilibre
- Nombreuses techniques d'équilibrage : arbre rouge-noir, AVL, B, déployé, tarbre, …
  - hors programme de ce cours

Bilan sur les ABR

- **Une** réalisation possible du TAD Dictionnaire
- INSÉRER/SUPPRIMER, RECHERCHER, ... : $O(h_{\mathcal{A}})$
  - Efficace uniquement si $h_{\mathcal{A}} = O(\log(\#\mathcal{A}))$
  - Vrai si insertion en ordre aléatoire
  - Techniques d'équilibrage basées sur les rotations

TAD Dictionnaire *ordonné*

- Mêmes opérations que Dictionnaire, plus
  - MIN(D), MAX(D) : clés minimale et maximale dans D
  - SUIVANT(D, c) : plus petite clé $> c$ dans D
  - PRÉCÉDENT(D, c) : plus grande clé $< c$ dans D
- Les ABR réalisent le TAD Dictionnaire ordonné — opérations en $O(h_{\mathcal{A}})$

Autres utilisations des ABR

- Tri $\rightarrow$ parcours infixe
- Files de priorité
- TAD Ensemble ou Ensemble ordonné — clés uniquement