Cours Algorithmique Avancé

Samba Ndojh NDIAYE

Laboratoire d'InfoRmatique en Image et Systèmes d'information
LIRIS UMR 5205 CNRS/Université Claude Bernard Lyon 1
samba-ndojh.ndiaye@univ-lyon1.fr

Évaluation

Notes

- 2 Contrôles : CC1 et CC2
- Note finale : 1/3 CC1 + 2/3 CC2

Supports

Supports

- Cours : https://perso.liris.cnrs.fr/samba-ndojh.ndiaye/fichiers/Cours_algo_avance/cours_algo_avance.pdf
- Slides : https://perso.liris.cnrs.fr/samba-ndojh.ndiaye/fichiers/Cours_algo_avance/slides_algo_avance.pdf

Plan

Motivations

Motivations

- Un Dictionnaire est une collection dynamique d'objets associant une clé à chaque objet : rechercher efficacement un élément à partir de sa clé
- Collections d'objets de natures très variées (entiers, pointeurs, structures...), mais de type identique
- Collections dynamiques : ajouter des éléments et supprimer des éléments
- Structures de données pour représenter des collections dynamiques d'objets
- Efficacité des opérations d'ajout, suppression, recherche...
- Restriction à une collection d'entiers tous différents

Rappels : SDD pour les collections d'entiers

Types abstraits

- Bases : langage python + types pré-définis
- Types de données évolués
- Spécification : ensemble de fonctions permettant de manipuler les objets de ce type
- Pas de connaissance des détails d'implémentation

SDD

- Tableaux : numpy
- Listes chaînées : à définir
- Piles et Files : types abstraits

Tableaux

Tableaux numpy

- Collection de variables de même type, indicées à partir de 0
- La fonction array crée un tableau à partir d'une liste de valeurs
- Les fonctions zeros et empty créent un tableau à partir du nombre et du type des éléments
- Nombre d'éléments fixé à la création : tableau statique
- Première case : nombre d'éléments réels
- Espace mémoire réservé, mais inutilisé

Exemple : $L = \{9, 5, 8, 2\}$

0	1	2	3	4	5	6	7	8	9
4	9	5	8	2	*None*	*None*	*None*	*None*	*None*

Listes chaînées

Listes chaînées

- Une liste chaînée est une suite de maillons
- Chaque maillon est stocke la valeur d'un objet et une référence vers le maillon suivant
- Une référence vers le premier maillon permet de parcourir la liste
- Taille liste variable : espace mémoire réservé proportionnel au nombre d'éléments

Exemple : $L = \{9, 5, 8, 2\}$

Piles et Files

Piles et Files

- Collection de valeurs de même type, de taille variable
- Représentation physique : tableau ou liste chaînée
- Ordre d'accès imposé (inadapté pour un dictionnaire)

Piles

- Ajout au sommet
- Retrait au sommet

Files

- Ajout à la fin
- Retrait en tête

Plan

Adressage direct

Tables à adressage direct

- Tableau classique : une case prévue pour chaque élément potentiel (univers des valeurs possibles)
- Grande efficacité : opérations ajout, recherche, suppression
- La taille de l'univers peut être problématique

Exemple

$U = [\![1, 10]\!]$
$L = \{2, 5, 8, 9\}$
$h : U \longrightarrow [\![0, 9]\!]$
$u \longmapsto u - 1$

0	1	2	3	4	5	6	7	8	9
None	2	*None*	*None*	5	*None*	*None*	8	9	*None*

Plan

Hachage

Tables de hachage

- Tableau : taille proportionnelle au nombre d'éléments à stocker
- Taille à choisir minutieusement
- Fonction de hachage : attribue une case pour chaque valeur possible de l'univers

Exemple

$U = [\![1, 100]\!]$
$L = \{2, 25, 8, 99\}$
$h : U \longrightarrow [\![0, 9]\!]$
$u \longmapsto u\%10$

0	1	2	3	4	5	6	7	8	9
None	*None*	2	*None*	*None*	25	*None*	*None*	8	99

Hachage

Tables de hachage

- Fonction de hachage déterministe : retrouver une valeur
- Taille du tableau plus petite que la taille de l'univers : collisions
- Gestion des collisions : chaînage et adressage ouvert

Plan

Le chaînage

Le chaînage

- Tableau de pointeurs vers des listes chaînées

Exemple

$U = [\![1, 100]\!]$
$L = \{2, 25, 8, 99, 12, 28, 59\}$
$h : U \longrightarrow [\![0, 9]\!]$
$\quad u \longmapsto u\%10$

Fonctions de hachage

La méthode de la division

- $h(u) = u\%m$, m étant la taille du tableau
- Choix de la taille du tableau
- Éviter que la valeur hachée ne dépende que d'une petite partie de la valeur à stocker
- Bon choix : un nombre premier pas trop proche d'une puissance 2

Exemple

$U = [\![1, 100]\!]$
$L = \{2, 25, 8, 99, 12, 28, 59\}$
$h : U \longrightarrow [\![0, 6]\!]$
$u \longmapsto u\%7$

Fonctions de hachage

La méthode de la multiplication

- $h(u) = \lfloor m(A.u - \lfloor A.u \rfloor) \rfloor$, A est une constante $0 < A < 1$
- Taille du tableau n'est pas problématique
- Bon choix pour A : $(\sqrt{5} - 1)/2$

Exemple

$U = [\![1, 100]\!]$
$L = \{2, 25, 8, 99, 12, 28, 59\}$
$h : U \longrightarrow [\![0, 9]\!]$
$u \longmapsto \lfloor 10(A.u - \lfloor A.u \rfloor) \rfloor$, avec $A = (\sqrt{5} - 1)/2$

Fonctions de hachage

Famille universelle

- Ensemble de fonctions $\mathcal{H} : U \longrightarrow [\![0, m-1]\!]$
- Risque de collisions borné : $\forall u_1, u_2 \in U$, le nombre de fonctions tq $h(u_1) = h(u_2)$ est borné par $|\mathcal{H}|/m$
- Exemple de famille universelle :
  $\mathcal{H} = \{h_{ab}(u) = ((au + b)\%p)\%m), p$ est premier,
  $a \in [\![1, p-1]\!]$, $b \in [\![0, p-1]\!]\}$

La méthode universelle

- Choix aléatoire dans une famille universelle de fonctions de hachage
- Garantie une probabilité très faible de créer le pire des cas

Fonctions de hachage

La méthode universelle

- Choix aléatoire dans une famille universelle de fonctions de hachage
- Garantie une probabilité très faible de créer le pire des cas

La méthode parfaite

- Liste de valeurs statique : pas de changements
- Choix aléatoire dans une famille universelle de fonctions de hachage
- Garantie une opération de recherche en $O(1)$ dans le pire des cas

Plan

Adressage ouvert

Adressage ouvert

- Tous les éléments sont stockés dans le tableau
- Le tableau peut être plein : mais plus de mémoire
- Recherche d'une case voisine en cas de collision
- Fonction de hachage définit une permutation des indices du tableau
- Sonder les cases du tableau et stocker la valeur dans la première case vide rencontrée
- Recherche d'un élément en parcourant la séquence de sondage
- Si des éléments peuvent être supprimés : le chaînage est préférable

Fonctions de hachage

Sondage linéaire

- $h(u,i) = (h'(u) + i)\%m$

Exemple

$U = [\![1, 100]\!]$
$L = \{2, 25, 8, 99, 12, 28, 59\}$
$h : U \times [\![0, 9]\!] \longrightarrow [\![0, 9]\!]$
$\quad (u, i) \quad \longmapsto (u + i)\%10$

Fonctions de hachage

Sondage linéaire

- La première case hachée détermine la séquence
- On ne peut générer que m séquences différentes parmi les $m!$ permutations possibles
- Elle conduit à de longues suites de cases occupées

Fonctions de hachage

Sondage quadratique

- $h(u,i) = (h'(u) + k1.i + k2.i^2)\%m$, $k1$ et $k2$ sont des constantes positives

Exemple

$U = [\![1, 100]\!]$
$L = \{2, 25, 8, 99, 12, 28, 59\}$
$h : U \times [\![0, 9]\!] \longrightarrow [\![0, 9]\!]$
$\quad (u, i) \quad \longmapsto (u + i + i^2)\%10$

Fonctions de hachage

Sondage quadratique

- La première case hachée détermine la séquence : m séquences différentes parmi les $m!$ permutations
- Elle est beaucoup plus performante que le sondage linéaire
- Il faut bien choisir les constantes k_1 et k_2 pour garantir que la séquence produite soit une permutation de tous les indices du tableau
- Si m est une puissance de 2, on peut choisir $k_1 = k_2 = 1/2$ pour obtenir une permutation des indices

Fonctions de hachage

Double hachage

- $h(u,i) = (h_1(u) + i.h_2(u))\%m$, h_1 et h_2 sont des fonctions de hachage classiques

Exemple

$U = [\![1, 100]\!]$
$L = \{2, 25, 8, 99, 12, 28, 59\}$
$h : U \times [\![0, 6]\!] \longrightarrow [\![0, 6]\!]$
$\quad (u, i) \quad \longmapsto (u\%7 + i(1 + u\%6))\%7$

Fonctions de hachage

Sondage quadratique

- Si $h_2(u)$ et m sont premières entre elles : m^2 séquences différentes
- Efficacité proche de l'optimum
- Une fonction efficace : choisir m un nombre premier et h_2 inférieure à m

Exemple

- $h_1(u) = u\%m$ et $h_2(u) = 1 + (u\%(m-1))$

Adressage ouvert

Complexité

- Opérations d'ajout et de recherche : la complexité dépend du facteur de remplissage $\alpha(\leq 1)$
- Elle est au plus de $1/(1-\alpha)$ cases sondées
- Si $\alpha = 50\%$, il faut donc sonder au plus 2 cases
- Si $\alpha = 75\%$, 4 cases
- Si $\alpha = 90\%$, 10 cases
- Si $\alpha = 99\%$, 100 cases
- Si α est bornée par une constante, la complexité est en temps constant $O(1)$.

Plan

Arbres binaires de recherche

Arbre binaire

- Un arbre binaire de recherche est une arborescence binaire : chaque nœud a au plus deux nœuds fils
- La racine se trouve au sommet de l'arborescence
- Une feuille est un nœud qui n'a pas de fils

Propriété

- Si x est un nœud de l'arbre de recherche et y un nœud du sous-arbre enraciné au fils gauche de x, alors la valeur de y est inférieure ou égale à celle de x
- Inversement, si y est un nœud du sous-arbre enraciné au fils droit de x, alors la valeur de y est supérieure ou égale à celle de x

Arbres binaires de recherche

Exemple

- Représenter deux arbres binaires de recherche pour l'ensemble $L = \{1, 2, 4, 6, 6, 7, 9\}$

ABR : modélisation

Modélisation

- Chaque nœud de l'arbre de recherche stocke un objet avec une clé unique (sa valeur)
- Un arbre peut être représenté avec une liste chaînée de structures de données
- Chaque maillon contient la valeur du nœud, un pointeur vers son fils gauche, un vers son fils droit et un dernier vers son père

Opérations

- Ajout, recherche et suppression d'une valeur
- Recherche valeur minimale/maximale

Opérations : ajout

Ajout

- On suppose que rac pointe vers le nœud racine de l'arbre
- On suppose l'existence d'un nœud x tel que $x.val$ est initialisée à la valeur à ajouter à l'arbre, $x.pere$, $x.droit$ et $x.gauche$ sont initialisées à $None$
- Le nœud x va être ajouté en tant que nouvelle feuille de l'arbre (seule $x.pere$ sera modifiée) en veillant à respecter la propriété d'un ABR
- Le nœud père de x, s'il existe, sera également mis à jour car il aura un nouveau fils
- Sinon, x sera à la racine de l'arbre

Opérations : suppression

Suppression

- le nœud est une feuille : on met à jour son nœud père, s'il existe, car il va perdre un fils
- le nœud a un seul fils : son fils va prendre sa place dans l'arborescence
- le nœud a deux fils : on va recopier la valeur de son successeur ou son prédécesseur dans le nœud à supprimer, puis retirer le nœud successeur ou prédécesseur de l'arbre sachant qu'il n'a qu'un seul fils

Opérations : suppression

Transfert

- remplace un sous-arbre enraciné en un nœud x par le sous-arbre enraciné au nœud y
- Si x est à la racine de l'arbre : il suffit de mettre y à la racine
- Sinon, x a un père z :
  - si x est le fils droit de z, alors y devient le fils droit de z
  - si x est le fils gauche de z, alors y devient le fils gauche de z

Opérations : suppression

def Transfert(rac, x, y)
    if $x.pere == None$:
        $rac = y$
    elif x == x.pere.gauche :
        $x.pere.gauche = y$
    else :
        $x.pere.droit = y$
    if $y\ != None$:
        $y.pere = x.pere$
    return rac

Opérations : suppression

Suppression

- Si x est une feuille : transférer le nœud $None$ à la place de x
- Si x a un seul fils : transférer le fils à la place de x
- Si x a deux fils : son successeur y est dans le sous-arbre droit de x et y n'a pas de fils gauche
  - Si y est un fils de x (y est fils droit de x) : transférer le sous-arbre en y à la place de x et raccorder comme fils gauche de y l'ancien fils gauche de x
  - Si y n'est pas le fils droit de x : soient d le fils droit de x et g le fils gauche de x
    - On remplace y par son propre fils droit z
    - On fait de d le fils droit de y
    - On transfert y à la place de x
    - On ajoute g comme fils gauche de y

Opérations : suppression

def Suppression(rac, x)
    if $x.gauche == None$:
        $rac =$Transfert($rac, x, x.droit$)
    elif $x.droit == None$:
        $rac =$Transfert($rac, x, x.gauche$)
    else :
        $y =$ Minimum($x.droit$) ou Successeur(x)
        if x ! $=y.pere$:
            $rac =$Transfert($rac, y, y.droit$)
            $y.droit = x.droit$
            $y.droit.pere = y$
        $rac =$Transfert(rac, x, y)
        $y.gauche = x.gauche$
        $y.gauche.pere = y$
    return rac

Opérations : suppression

Suppression

- La complexité de l'algorithme de suppression dépend de la hauteur de l'arbre.

ABR : les parcours

Parcours

Lister ses éléments selon un ordre particulier

Parcours

- Le parcours en profondeur consiste, étant donné un nœud x à parcourir son sous-arbre gauche, puis son sous-arbre droit
- Le parcours *préfixe* consiste à traiter le nœud x avant celui de ces sous-arbres
- Le parcours *postfixe* traite le nœud x après celui de ces sous-arbres
- Le parcours *infixe* traite le nœud x entre celui des sous-arbres.

ABR vs Hachage : complexité

Pire des cas	Ajout	Suppression	Recherche
ABR	$O(n)$	$O(n)$	$O(n)$
ABR équilibré	$O(log(n))$	$O(log(n))$	$O(log(n))$
Hachage chaînage	$O(1)$	$O(1)$	$O(n)$
Hachage adressage ouvert	$O(n)$	$O(1)$	$O(n)$

Cas moyen	Ajout	Suppression	Recherche
ABR	$O(log(n))$	$O(log(n))$	$O(log(n))$
ABR équilibré	$O(log(n))$	$O(log(n))$	$O(log(n))$
Hachage chaînage	$O(1)$	$O(1)$	$O(1)$
Hachage adressage ouvert	$O(1)$	$O(1)$	$O(1)$

ABR vs Hachage : complexité

ABR vs Hachage

- ABR assure un comportement stable avec une complexité pour les opérations nécessaires en $O(log(n))$
- ABR fournit des opérations supplémentaires(minimum/maximum, successeur/prédécesseur) avec une complexité en $O(log(n))$
- L'espace mémoire requis pour un ABR est proportionnel au nombre d'éléments
- Une table de hachage assure un comportement optimal si son taux de remplissage est faible
- Le comportement d'une table de hachage devient chaotique si son taux de remplissage devient important

Plan

Arbres Rouge Noir

Parcours

Un arbre Rouge Noir est un ABR avec les propriétés additionnelles suivantes :

- Chaque nœud est soit rouge ou noir
- La racine est noire
- Si un nœud est rouge alors ses fils éventuels sont noirs
- Pour tout nœud, tous les chemins de ce nœud vers une feuille comportent le même nombre de nœuds noirs.

Arbres Rouge Noir

Complexités

- Un chemin de la racine à une feuille est au plus 2 fois plus long que le plus petit chemin
- Un ARN est presque équilibré : sa hauteur est inférieure ou égale à $2.log(n+1)$
- Complexité des opérations de recherche, de recherche d'un minimum/maximum, de recherche d'un successeur/prédécesseurs est en $O(log(n))$.

Arbres Rouge Noir

Ajout

- Algorithme d'ajout d'un nœud dans un ABR, puis on colorie en rouge le nœud ajouté
- Restauration des propriétés invalidées d'ARN par des changements de couleurs et des rotations
- Complexité en $O(log(n))$ avec au plus 2 rotations

Suppression

- Algorithme de suppression d'un nœud dans un ABR
- Restauration des propriétés invalidées d'ARN par des changements de couleurs et des rotations
- Complexité en $O(log(n))$ avec au plus 3 rotations

Plan

Arbres AVL

Complexités

- AVL pour Adelson-Velskii et Landis
- Pour chaque nœud, la différence de hauteur de ses sous-arbres gauche et droite est d'au plus 1
- Un arbre AVL est équilibré : hauteur en $O(log(n))$
- L'ajout d'un nœud nécessite au plus 2 rotations : complexité en $O(log(n))$
- La suppression d'un nœud peut entraîner au pire $log(n)$ rotations : complexité en $O(log(n))$