Algorithmique

Cristina Sirangelo, ENS-Cachan

Préparation à l'option Informatique
de l'agrégation de mathématiques

Plan

1. Analyse et complexité des algorithmes (S.Haddad)
2. Types abstraits et structures de données
3. Algorithmes de tri
4. Techniques classiques de conception d'algorithmes
5. Algorithmes de graphes

Plan

1. Analyse et complexité des algorithmes (S.Haddad)
2. Types abstraits et structures de données
3. Algorithmes de tri
4. Techniques classiques de conception d'algorithmes
5. Algorithmes de graphes

Leçons concernées:

- 901 Structures de données : exemples et applications.
- 921 Algorithmes de recherche et structures de données associées.
- 926 Analyse des algorithmes : complexité. Exemples.

Types abstraits et structures de données

- Les algorithmes opèrent sur des données (données en input, données auxiliaires)
- Aspect central de l'algorithmique: représentation et manipulation des données
- Deux niveaux de représentation:
  - niveau abstrait ou logique (*type abstrait de données*)
  - implémentation (*structure de données*)
- Séparation des deux niveaux : modularité et interface
  - utiliser un objet ou une fonctionnalité par son interface abstraite, indépendamment des détails de son implémentation

Type abstrait

- un ensemble d'objets
- des opérations qui les manipulent

Structure de données

- Représentation concrète des objets décrits par le type abstrait dans la mémoire d'un ordinateur
- Implémentation des opérations sur cette représentation

Représentation des données: en termes de types élémentaires et d'autres types abstraits (**hiérarchie de types**)

Types abstraits et structures de données

- Piles
- Files
- Listes
- Arbres
- Graphes
- Dictionnaires
  - Arbres binaires de recherche, tables de hachage, ...
- Files de priorité
- etc

Pile: type abstrait

Pour un type T,

Type abstrait **Pile(T)**:

Données : les suites d'éléments de type T

Opérations :

PILE-VIDE(P: Pile(T)) : Booléen

SOMMET(P: Pile(T)) : T

EMPILER (P: Pile(T), x: T)

DEPILER (P: Pile(T))

CREER-PILE () : Pile(T)

Pile: type abstrait

Sémantique

PILE-VIDE(P: Pile(T)) : Booléen

Retourne vrai si la suite P est vide ; faux sinon.

SOMMET(P: Pile(T)) : T

Retourne l'élément x de type T tel que P = P'x ; P doit être non vide

EMPILER (P: Pile(T) , x : T)

Modifie la suite P: P ← Px

DEPILER (P: Pile(T))

P←P', P' tel que P =P' x ; P doit être non vide

CREER-PILE ( ): Pile(T)

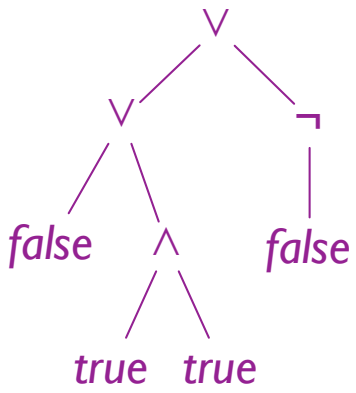
Retourne une suite vide

Pile: type abstrait

On peut utiliser un type abstrait sans connaître son implémentation

Exemple Évaluation d'une expression Booléenne postfixe.

Ex. *false true true* ∧ ∨ *false* ¬ ∨ #




```
P ← CREER-PILE();
s ← next()   // prochain symbole d'input

while ( s ≠ # ) do

    switch (s)

        case Boolean :  EMPILER(P, s)

        case ∧:   s1←SOMMET(P);  DEPILER(P);   s2←SOMMET(P);  DEPILER(P);
                  EMPILER(P,  s1 and s2)

        case ∨:   s1 ← SOMMET(P);  DEPILER(P);   s2 ← SOMMET(P);  DEPILER(P);
                  EMPILER(P,  s1 or s2)

        case ¬:   s1 ← SOMMET(P);  DEPILER(P);  EMPILER(P,  not s1)

    endswitch

    s ← next() ;

endwhile
```

Pile: implémentation par tableau

Permet l'implémentation de piles d'au plus *n* éléments.

Données :
Une pile:

un tableau S[1..n] d'éléments de type T
un index *sommet*

Opérations :

CREER (): Pile(T) créer un tableau S; sommet ← 0; return (S, sommet)

PILE-VIDE(P: Pile(T)) : Boolean return P.sommet = 0

SOMMET(P: Pile(T)) : T return P.S[P.sommet] // la pile doit être non vide, i.e. sommet ≥ 1

EMPILER (P: Pile(T) , x :T) // la pile doit être non pleine, i.e. sommet < n

P. sommet ← P. sommet + 1; P.S[P.sommet] ← x

DEPILER (P: Pile(T)) // la pile doit être non vide, i.e. sommet ≥ 1

P.sommet ← P.sommet − 1

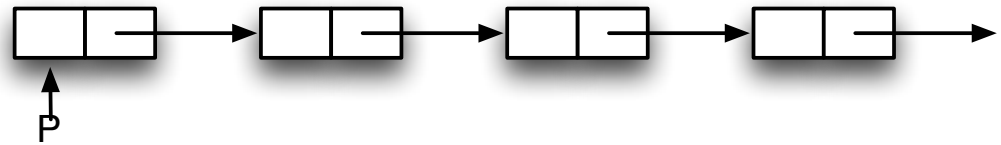
Complexité: chaque opération en temps O(1)

Pile: implémentation par liste chainé

Pour un type T, on utilise le type Block(T) qui décrit les couples (*cont*, *next*) où *cont* est un élément de type T et *next* est un pointeur

Données :

Une pile: un pointeur à un élément de type Block(T) (pointeur au sommet de la pile)



Opérations :

CREER-PILE (): Pile(T) return NIL

PILE-VIDE(P: Pile(T)) : Booléen return P = NIL

SOMMET(P: Pile(T)) :T //la pile doit être non vide, i.e. P ≠ NIL

return P^.cont

EMPILER (P: Pile(T), x :T)

créer un élément B de type Block(T); B.cont ← x; B.next ← P
P ← pointeur à B

DEPILER (P: Pile(T)) //la pile doit être non vide, i.e. P ≠ NIL

N ← P^.next; détruire l'élément pointé par P; P ← N

Complexité : chaque opération en temps O(1)

File

Exercice: Définir le type abstrait File et proposer deux implémentations:

- par tableau

- par liste chainée

Liste: type abstrait

Une liste est une suite d'éléments qui admet des insertions et suppressions arbitraires.

Pour un type T, et un type Place qui peut représenter des identifiants,

Type abstrait **Liste (T)** :

Données: Suites de couples (x, p), où x est un élément de type T et p est un identifiant de type Place; tous les identifiants de place dans la suite sont distincts.

Opérations et leur sémantique:

CREER-LISTE(): Liste(T); crée une liste constituée d'une suite vide

INSERER-EN-TETE(L: Liste(T) , x: T)

Soit p un nouveau identifiant de place (distinct de tous les identifiants déjà en L);
alors L ← (x, p) L

INSERER-APRES(L: Liste(T) , p: Place, x: T)

Soit $L=L_1\ (y, p)\ L_2$ et soit p' un nouvel identifiant de place;
alors $L \leftarrow L_1\ (y, p)\ (x, p')\ L_2$; indéfini si L ne contient aucun élément identifié par p

SUPPRIMER (L: Liste(T), p: Place)

Soit $L=L_1\ (y, p)\ L_2$; alors $L \leftarrow L_1 L_2$;
indéfini si p n'apparait pas dans L

VIDE(L: Liste(T)) : Booléen; retourne *vrai* ssi la suite L est vide.

Listes

Opérations et leur sémantique (suite):

CONTENU(L: Liste(T), p : Place) : T

Donne l'élément x tel que $L=L_1\ (x, p)\ L_2$; indéfini si p n'apparaît pas dans L

PREMIER(L: Liste(T)) : Place;

Donne la place p telle que L=(x,p) L'; indéfini si L est vide

SUIVANT(L: Liste(T), p : Place) : Place;

Donne la place p' telle que $L=L_1\ (x, p)\ (y, p')\ L_2$;

indéfini si (x,p) est le dernier élément de L ou si p n'apparaît pas dans L

EST-DERNIER(L: Liste(T) , p : Place) : Booléen;

Teste si $L = L_1\ (x, p)$

CONCATENER (L: Liste(T), L': Liste(T)) : Liste(T)

Retourne L L' (renommage des identifiants pour que ils soient disjoints)

SCINDER (L: Liste(T), p: Place) : (Liste(T), Liste(T))

Donne un couple de listes (L_1, L_2) tel que $L=L_1L_2$ (modulo les identifiants) et le denier élément de L_1 est identifié par p . Indéfini si p n'apparaît pas dans L.

Listes

Opérations dérivées et leur sémantique :

D'autres opérations peuvent être obtenues à partir des opérations primitives:

DERNIER(L: Liste(T)): Place

Donne la place p telle que L=L'(x, p); indéfini si S est vide

INSERER-EN-QUEUE (L: Liste(T), x :T)

Soit p un nouveau identifiant de place (distinct de tous les identifiants déjà en L); alors L←L (x, p)

TETE(L: Liste(T)) : T;

Donne l'élément x tel que L=(x, p) L' ; indéfini si L est vide

QUEUE(L: Liste(T)) : T;

Donne l'élément x tel que L=L' (x, p) ; indéfini si L est vide

Listes

Implémentation des opérations dérivées:

```
DERNIER( L: Liste(T) ): Place
    p ← PREMIER(L)
    while ( not EST-DERNIER(L, p) )
        p ← SUIVANT(L, p)
    endwhile
    return p
```

L'implémentation directe de DERNIER peut être plus efficace selon l'implémentation de la liste.

```
INSERER-EN-QUEUE (L: Liste(T) , x:T)
    If VIDE( L ) then INSERER-EN-TETE (L, x)
    else INSERER-APRES( L, DERNIER(L), x )
    endif
```

```
TETE( L: Liste(T) ) : T     return CONTENU( L, PREMIER(L) )
```

```
QUEUE( L: Liste(T) ) : T    return CONTENU( L, DERNIER(L) )
```

Listes

Exemple d'utilisation du type Liste (T)

Écrire une fonction qui teste si un élément x de type T est présent dans une liste

TROUVER(L: Liste(T), x :T) : Booleen

```
if VIDE(L) return false
else
        p←Premier(L)
        while ( CONTENU(L, p)≠x )
                if EST-DERNIER(L, p) then  return false
                else p ← SUIVANT(L, p)
                endif
        endwhile
        return true
endif
```

Listes: implémentation

Implémentation par Tableau (permet d'implémenter des listes de longueur maximale n)

Données:

Une liste:

Un tableau S[1..n] d'éléments de type T (les indices sont les identifiants de place);
une variable s qui maintient à jour la longueur de la liste

Opérations: *exercice*

Complexité des opérations: toutes en temps O(1) sauf:

- les opérations d'insertion et suppression: O(n), où n est la longueur de la liste (déplacer ou re-compacter le contenu du tableau)
- CONCATENER: O(n)
- SCINDER: O(n)

NB: On utilise une implémentation directe de DERNIER en temps O(1)

Listes: implémentation

Implémentation par liste chainée

Pour un type T on utilise le type Block(T) qui représente l'ensembles des couples (*cont*, *next*) où *cont* est de type T et *next* est un pointeur

Données:

Une liste:

Un pointeur à un élément de type Block(T) (la tête de la liste)
(les identifiants de place sont des pointeur à Block(T))

Opérations: *exercice*

Complexité des opérations: Toutes en temps O(1), sauf

SUPPRIMER: O(n) où n est la longueur de la liste. (modifier le pointeur *next* du précédent de l'élément supprimé)

CONCATENER: O(n) où n est la longueur de la première liste (trouver le dernier)

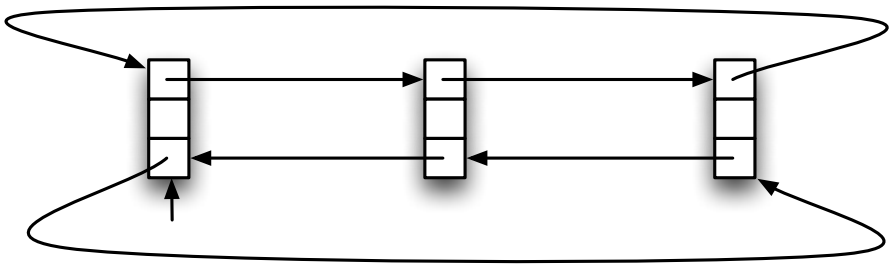
DERNIER, INSERER-EN-QUEUE, QUEUE: O(n) (toutes ces opérations demandent de trouver le dernier de la liste, implémentation dérivée des opérations primitives)

Amélioration possible: maintenir un pointeur au dernier élément de la liste

Listes: implémentation

Implémentation par liste doublement chainée circulaire

Pour un type T on utilise le type DBlock(T) qui représente l'ensemble des triplets (*cont*, *next*, *prev*) où *cont* est un élément de type T; *next* et *prev* sont des pointeurs



Données:

Une liste:

Un pointeur à un élément de type DBlock(T) (la tête de la liste)
(les identifiants de place sont des pointeur à DBlock(T))

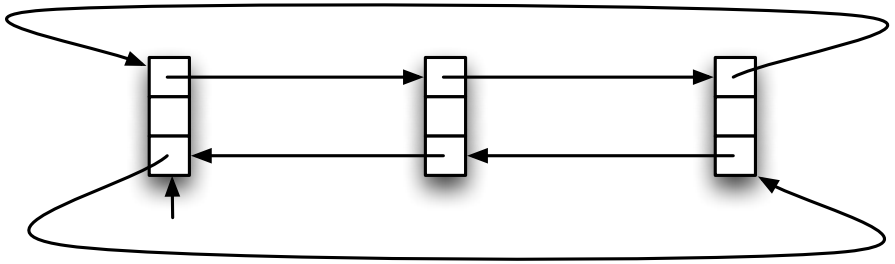
Opérations:

Exemples d'implémentation:

Listes: implémentation

Implémentation par liste doublement chainée circulaire (suite)

```
SUPPRIMER (L: Liste(T),  p: Pointeur à DBlock(T))
        ne ← p^.next;   pr ← p^.prev;
        if (ne=p) then L←nil
        else
                if (L= p) then L←ne endif
                ne^.prev ← pr;  pr^.next ← ne
        endif
        détruire p^
DERNIER( L: Liste(T) ): Pointeur à DBlock(T)
        if (L=nil) then return nil
        return L^.prev
```



Listes: implémentation

Implémentation par liste doublement chainée circulaire (suite)

CONCATENER (Liste(T) L, Liste (T) L1): Liste(T)

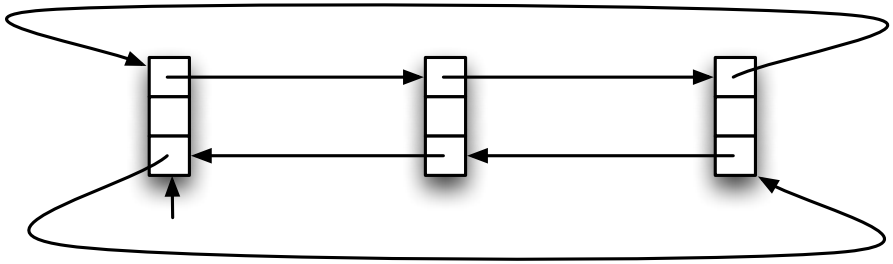
if (L1 = nil) then return L endif

if (L = nil) then return L1 endif

q← L^.prev; fs ← L1; ls ← fs^.prev

q^.next ← fs; fs^.prev ← q

ls^.next ← L; L^.prev ← ls;



Listes: implémentation

Implémentation par liste doublement chainée circulaire (suite)

SCINDER (L: Liste(T), p: Pointeur à DBlock(T)): (Liste(T), Liste(T))

L1←L; L2 ← nil;

ls ← L^.prev

if (p ≠ ls) then

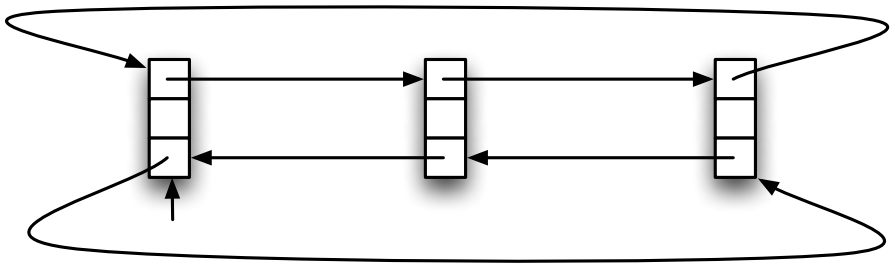
L2 ← p^.next;

L2^.prev ← ls; ls.next ← L2

L^.prev ← p; p^.next ← L;

endif

return (L1, L2)



Complexité des opérations: Toutes en temps O(1)

D'autres opérations typiques en temps O(1) avec l'implémentation par liste doublement chainée circulaire:
SUIVANT-CYCLIQUE(Liste(T) L, p: Place) : Place;
PRECEDENT(Liste(T) L, , p : Place) : Place;
INSERER-AVANT(Liste(T) L, p: Place, x:T)

Arbres binaires

Sur un type T, un arbre binaire peut être défini de façon récursive comme suit:
un arbre binaire est soit vide, soit composé d'un élément de type T (dit *racine*), un *sous-arbre gauche* et un *sous-arbre droit*

Le type abstrait **Arbre(T)**

Données: les arbres binaires d'éléments de type T

Opérations et leur sémantique:

CREER-ARBRE() : Arbre(T); crée un arbre vide

CREER-ARBRE(x: T) : Arbre(T); crée un arbre composé d' une racine x et deux sous-arbres vides

FIXER-RACINE(Arbre(T) A, x:T);

modifie A: la racine de A devient x; indéfini si A est vide

FIXER-GAUCHE(Arbre(T) A, Arbre(T) Ag);

le sous-arbre gauche de A devient Ag; indéfini si A est vide

FIXER-DROIT (Arbre(T) A, Arbre(T) Ad)

le sous-arbre droit de A devient Ad; indéfini si A est vide

Arbres binaires

Opérations et leur sémantique (suite):

EST-VIDE(Arbre(T) A): Booléen; teste si A est vide

RACINE (Arbre(T) A): T; retourne la racine de A

SAG (Arbre(T) A) : Arbre (T); retourne le sous-arbre gauche de A

SAD (Arbre(T) A): Arbre(T); retourne le sous-arbre droit de A

Implémentation: Typiquement dynamique (pointeur à bloc composé d'un élément de type T un pointeur au sous-arbre gauche et un pointeur au sous-arbre droit)

On utilise le type
BlockA(T) : les triplets (cont: T; g: Pointeur; d: Pointeur)

Données:

Un arbre: un pointeur à BlockA(T)

Opérations

CREER() : Arbre(T) return NIL

Arbres binaires

Opérations (suite) :

CREER(x:T) :Arbre(T)

créer un BlockA(T) B; B.cont ← x; B.g← NIL; B.d← NIL;

return pointeur à B

FIXER-RACINE(Arbre(T) A, x:T) ^A.cont ←x;

FIXER-GAUCHE(Arbre(T) A, Arbre(T) Ag) ^A.g ←Ag;

SAG(Arbre(T) A):Arbre(T) return ^A.g

etc.

Complexité des Opérations (implémentation dynamique): toutes en temps O(1)

Dictionnaires et algorithmes de recherche

- Dictionnaire: un type abstrait de données qui permet de représenter et manipuler des ensembles typiquement ordonnés.

- Principales opérations: **recherche** d'un élément, **insertion** et **suppression**

- Éléments de l'ensemble accessibles par un champ *clef* (identifiant)

- D'autres informations associées à chaque élément: schématisées par un seul autre champ *valeur*, qui peut être d'un type complexe.

Dictionnaires et algorithmes de recherche

Type abstrait
Pour un type Tval pour les valeurs et un type Tclef totalement ordonné pour les clefs

Dictionnaire(Tclef, Tval)

Données: Ensembles de couples (*clef*, *valeur*) où *clef* est de type Tclef et *valeur* est de type Tval

Opérations et leur sémantique:

CREER( ): Dictionnaire(Tclef, Tval); crée et retourne un ensemble vide de couples

EST-VIDE(Dictionnaire(Tclef, Tval) D): Booléen; teste si D est l'ensemble vide

CHERCHER (Dictionnaire(Tclef, Tval) D, Tclef c): Tval;
Retourne v tel que $(c,v) \in D$; retourne UNSET si la clef c n'apparaît pas dans D

INSERER (Dictionnaire(Tclef, Tval) D, Tclef c, Tval v); $D \leftarrow D \cup \{ (c, v) \}$

SUPPRIMER (Dictionnaire(Tclef, Tval) D, Tclef c); $D \leftarrow D \setminus \{ (c, v) \}$ où v est tel que $(c, v) \in D$

TAILLE(Dictionnaire (Tclef, Tval) D): Entier; Retourne $|D|$

FUSIONNER (Dictionnaire D1, Dictionnaire D2): Dictionnaire; Retourne $D1 \cup D2$

SCINDER (Dictionnaire D, Tclef c): (Dictionnaire, Dictionnaire) ;
Retourne (D1, D2) où $D1 = \{ (c',v) \in D \mid c' \leq c \}$ et $D2 = D \setminus D1$

Dictionnaires

D'autres opérations typiques (qui utilisent l'ordre total sur les clefs)

MINIMUM, MAXIMUM, PREDECESSEUR, SUCCESSEUR

Implémentations possibles

- par Tableau (non-trié et trié)
- par Liste (non-trié et trié)
- par Arbre Binaire de Recherche
- par Table de Hachage

Dictionnaires: implémentation par tableau non trié

Données

Un dictionnaire sur (Tclef, Tval):

Un tableau *T*[1..N] de couples (clef, val)

Un entier *taille;*

Opérations

CREER( ): Dictionnaire(Tclef, Tval)

créer un tableau T[1..N]; return < T, 0 >

CHERCHER (Dictionnaire(Tclef, Tval) D, Tclef c): Tval

Parcourir le tableau D.T de 1 à D.taille jusqu'à ce que D.T[i].clef = c

if (clef c non trouvée) return UNSET

return D.T[i].val

INSERER (Dictionnaire(Tclef, Tval) D, Tclef c, Tval v) *Indéfini si D. taille = N*

D.taille ← D.taille +1; D.T[D.taille]← (c,v)

Dictionnaires: implémentation par tableau non trié

Opérations (suite)

SUPPRIMER (Dictionnaire(Tclef, Tval) D, Tclef c)

Parcourir le tableau D.T de 1 à D.taille jusqu'à ce que D.T[i].clef = c

if(clef c trouvée) D.T[i] ← D.T[D.taille]; D.taille ← D.taille - 1

Complexité (nombre de comparaisons/écritures)

	cas meilleur	cas pire	cas moyen
Recherche	1	n	hit: n/2 miss: n
Insertion	1	1	1
Suppression	1	n	hit: n/2 miss: n

Analyse du cas moyen pour la recherche/suppression (hit) :

Variable aléatoire Cn : coût de la recherche/suppression sur un dictionnaires de n éléments
Espace des épreuves : toutes les couples (D, c) ù D est un dictionnaire de taille n et c une clef dans D (équiprobables)

coût = position de la clef à rechercher
Probabilité que la clef à rechercher soit en position i = 1/n ⇒ E[Cn] =n/2

Dictionnaires: implémentation par liste non triée

Données

Dictionnaire(Tclef, Tval): Une liste de type Liste(Tclef, Tval)
(implémentation par liste doublement chainée)

Recherche: parcours linéaire

Insertion: insérer en tête

Suppression: parcours linéaire

Complexité: comme dans le cas de tableau non-trié

Avantages: dynamique, pas de borne sur la taille

Inconvénients: gestion des pointeurs

Dictionnaires: implémentation par tableau trié

Données

Dictionnaire(Tclef, Tval):

Un tableau *T*[1..N] de couples (clef, val)

Un entier *taille;*

Opérations

Recherche dichotomique

```
CHERCHER-dicho ( Tableau(Tclef,Tval) T, Tclef c, Entier g, Entier d ) : Tval
        if (g > d) return UNSET
        m = (g+d)/2;
        if ( c = T[m].clef ) return T[m].val
        if ( c < T[m].clef ) return CHERCHER-dicho(T, c, g, m-1)
        else return CHERCHER-dicho(T, c, m+1, d );

CHERCHER ( Dictionnaire(Tclef,Tval) D, Tclef c ) : Tval
        return CHERCHER-dicho ( D.T, c, 1, D.taille )
```

Dictionnaires: implémentation par tableau trié

Opérations (suite)

INSERER (Dictionnaire(Tclef, Tval) D, Tclef c, Tval v) *Indéfini si D. taille = N*

```
i = D.taille
while ( i > 0  &  D.T[ i ]. clef  > c) do
        D.T [ i+1 ] ← D.T[ i ]; i ← i-1
endwhile
D.T [ i+1 ] ← (c,v) ; D.taille ← D.taille +1;
```

SUPPRIMER (Dictionnaire(Tclef, Tval) D, Tclef c)

Parcourir le tableau D.T de D.taille à 1 jusqu'à ce que D.T[i].clef ≤ c

```
if ( clef c trouvée )
        "Shift" de D.T[ i+1 .. taille]   d'une position à gauche
endif
```

Dictionnaires: implémentation par tableau trié

Complexité (nombre de comparaisons/écritures)

	cas meilleur	cas pire	cas moyen
Recherche	1	log n	hit: log n miss: log n
Insertion	1	n	n/2
Suppression	1	n	hit: n/2 miss: n/2

Analyse du cas moyen

Recherche (miss) : comme le cas pire

Recherche (hit): borne sup: cas pire, borne inf: recherche par comparaisons

Insertion / Suppression (miss) :

Coût = position i dans laquelle la clef devrait être insérée (par rapport à la fin)

Probabilité que la clef soit insérée en position i: 1/n

Suppression (hit) :

Coût = position i de la clef à supprimer (par rapport à la fin)

Probabilité que la clef soit en position i: 1/n

Dictionnaires: implémentation par tableau trié

Recherche par interpolation

(clefs numériques)

Dans la recherche binaire remplacer m = (g+d)/2 par

$$m = g + \frac{\text{clef} - T[g].\text{clef}}{T[d].\text{clef} - T[g].\text{clef}} \cdot (d - g)$$

Complexité de la recherche (hit):

O(1) si les clefs sont distribuées de façon régulière entre T[1] et T[n]

O(log log n) dans le cas moyen pour clefs aléatoires (distribution uniforme)

O(n) dans le cas pire

Dictionnaire: implémentation par liste triée

Données

Dictionnaire sur (Tclef, Tval): Une liste de type Liste(Tclef, Tval)

Operations

```
CHERCHER ( Dictionnaire(Tclef, Tval) D, Tclef c ):Tval
    if VIDE(D) return UNSET
    else
        p=Premier(D)
        while ( CONTENU(D, p).clef < c )
            if ( EST-DERNIER(D, p) ) return UNSET
            else p ← SUIVANT(D, p)
            endif
        endwhile
        if ( CONTENU(D, p).clef = c ) return CONTENU(D, p).val
        else return UNSET
    endif
```

Dictionnaire: implémentation par liste triée

Opérations (suite)

INSERER (Dictionnaire(Tclef, Tval) D, Tclef c, Tval v)

Parcours linéaire de la liste jusqu'à ce que:

CONTENU(SUIVANT(D, p)). clef ≥ c ou EST-DERNIER (D, p)

INSERER-APRES (D, p, (c,v))

SUPPRIMER(Dictionnaire(Tclef, Tval) D, Tclef c)

Parcours linéaire de la liste jusqu'à ce que:

CONTENU(D, p). clef ≥ c

if (*clef c trouvée*)

SUPPRIMER (D, p)

Dictionnaires: implementation par liste triée

Complexité (nombre de comparaisons/écritures)

	cas meilleur	cas pire	cas moyen
Recherche	1	n	hit: n/2 miss: n/2
Insertion	1	n	n/2
Suppression	1	n	hit: n/2 miss: n/2

Analyse du cas moyen:

Insertion et suppression: comme dans le cas de tableau trié
Recherche: même coût que la suppression

Comparaison avec la solution par tableau trié

Inconvénients: recherche linéaire

Avantages: dynamique, mises à jour plus efficaces (évite les shifts)

Dictionnaire: implémentation par ABR

Combine la flexibilité de l'implémentation par liste, et l'avantage de la recherche dichotomique de l'implémentation par tableau

Soit Tclef un domaine totalement ordonné

Arbre binaire de recherche sur (Tclef, Tval):

un arbre binaire sur un domaine (Tclef, Tval) tel que pour tout sommet x

clefs dans le sous-arbre gauche de x < x. clef < clefs dans le sous-arbre droit de x

Dictionnaire: implémentation par ABR

Données:

un dictionnaire sur (Tclef, Tval): un élément de type Arbre(Tclef, Tval)

Opérations:

```
CREER ( ) : Dictionnaire(Tclef, Tval)
        return CREER-ARBRE();

CHERCHER ( Dictionnaire(Tclef, Tval) D, Tclef c ): Tval
        if ( EST-VIDE(D) ) then   return UNSET
        elseif  ( c = RACINE (D).clef ) then retourner RACINE (D). val
        elseif ( c < RACINE (D).clef ) then
                return CHERCHER( SAG(D), c )
        else
                return CHERCHER( SAD(D), c )
        endif
```

Dictionnaire: implémentation par ABR

Opérations (suite) :

```
INSERER ( Dictionnaire(Tclef,Tval) D,  Tclef c, Tval v )
        if ( EST-VIDE(D) ) then
                D ← CREER-ARBRE(c, v)
        elseif ( c < RACINE(D).clef ) then
                INSERER( SAG(D), c, v )
        else
                INSERER( SAD(D), c, v )
        endif
```

Dictionnaire: implémentation par ABR

Opérations (suite) :

```
SUPPRIMER ( Dictionnaire(Tclef,Tval) D,  Tclef c )
        if ( EST-VIDE (D) ) return
        if ( c < RACINE(D).clef ) then
                SUPPRIMER(SAG (D), c)
        elseif ( c > RACINE(D).clef ) then
                SUPPRIMER(SAD (D), c)
        else // c = RACINE(D).clef
                // D a un seul sous-arbre non-vide
                if EST-VIDE( SAG( D ) )  then D ← SAD (D)
                elseif EST-VIDE( SAD( D ) )  then D ← SAG (D)
                else
                // D a deux sous-arbres non-vides
                FIXER-RACINE ( D,  SUPPRIMER-MAX(SAG (D)) )
        endif
```

Dictionnaire: implémentation par ABR

SUPPRIMER-MAX (S)

- trouve l'élément de S avec clef maximale (l'élément le plus à droite)
- le supprime et retourne son contenu

```
SUPPRIMER-MAX ( Arbre(Tclef, Tval) A ):  (Tclef, Tval)   // A doit être non-vide
        if EST-VIDE ( SAD(A) ) then
            x ← RACINE (A)
            A ← SAG(A);
        else
            x ← SUPPRIMER-MAX ( SAD(A) )
        endif
        return x
```

Dictionnaire: implémentation par ABR

Complexité

Recherche, Insertion, Suppression

Cas meilleure: O(1)

(recherche d'un élément à la racine;
insertion dans un sous-arbre vide de la racine;
suppression de la racine avec un sous-arbre vide)

Cas pire: O(n)

arbre binaire de recherche = liste (croissante/décroissante) de n éléments :
recherche/insertion/ suppression du maximum/minimum;

Dictionnaire: implémentation par ABR

Complexité Recherche (hit/miss), Insertion, Suppression

Cas Moyen: O(log n)

- Espace des épreuves: Toutes les permutations des clefs { 1,..., n } (équiprobables)
- **ABR construit aléatoirement:** arbre construit par insertions successives des clefs de la permutation
  - Cela ne coïncide pas avec la distribution uniforme sur les ABR sur les clefs {1, ..., n}
- Sur chaque ABR T

  Coût(T, clef) ≤ hauteur de T

  variable aléatoire Cn : coût de la recherche (insertion/suppression) d'une clef aléatoire sur un ABR de n clefs construit aléatoirement

  variable aléatoire Hn: hauteur d'un ABR de n clefs construit aléatoirement

$$Cn \leq Hn \Rightarrow C_{moy}(n) = E[Cn] \leq E[Hn]$$

E[Hn]: hauteur moyenne d'un ABR construit aléatoirement

On montre E[Hn] = O(log n)

⇒ **Coût moyen de la recherche (insertion/ suppression) O(log n)**

Hauteur moyenne d'un ABR construit aléatoirement

On montre E[Hn] = O(log n) :

Soit Gn (Dn, resp) la hauteur du sous-arbre gauche (droit) d'un ABR construit aléatoirement

$$H_n = 1 + \max\{G_n, D_n\} \quad \text{pour } n \geq 1$$
$$H_0 = 0$$

On travaille avec les variables aléatoires 2^{G_n}, 2^{D_n}, et $Y_n = 2^{H_n}$

$Y_n = 2 \cdot \max\{ 2^{G_n}, 2^{D_n} \} \leq 2 (2^{G_n} + 2^{D_n})$ donc $E[Y_n] \leq 2 (E[2^{G_n}] + E[2^{D_n}])$ pour $n \geq 1$

$Y_0 = 1$

On calcule $E[2^{G_n}]$ et $E[2^{D_n}]$ pour $n \geq 1$:

Soit Rn la variable aléatoire qui décrit la clef dans la racine de l'arbre (i.e. la première clef de la permutation)

$Pr[R_n = i] = \frac{1}{n}$ pour tout i

$$E[2^{G_n}] = \sum_{i=1..n} E[2^{G_n} \mid R_n = i] \cdot Pr[R_n = i] = \frac{1}{n} \sum_{i=1..n} E[2^{G_n} \mid R_n = i]$$

similaire pour $E[2^{D_n}]$

Hauteur moyenne d'un ABR construit aléatoirement

Sachant Rn = i, le sous-arbre gauche d'un ABR construit aléatoirement sur {1,..., n}

est un ABR construit aléatoirement sur les clefs {1,.., i-1}:

$Pr[Gn = k \mid Rn = i] = Pr[Hi-1 = k] \Rightarrow$

$E[2^{Gn} \mid Rn = i] = E[2^{Hi-1}] = E[Yi-1]$

De façon similaire:

$Pr[Dn = k \mid Rn = i] = Pr[Hn-i = k] \Rightarrow$

$E[2^{Dn} \mid Rn = i] = E[Yn-i]$

et

$$E[2^{Gn}] = \frac{1}{n} \sum_{i=1..n} E[Yi-1]$$

$$E[2^{Dn}] = \frac{1}{n} \sum_{i=1..n} E[Yn-i]$$

Hauteur moyenne d'un ABR construit aléatoirement

$$E[\, 2^{G_n} \,] = \frac{1}{n} \sum_{i=1..n} E[\, Y_{i-1} \,]$$

$$E[\, 2^{D_n} \,] = \frac{1}{n} \sum_{i=1..n} E[\, Y_{n-i} \,]$$

En utilisant $E[Y_n] \leq 2 \, (E[\, 2^{G_n} \,] + E[\, 2^{D_n} \,])$ pour $n \geq 1$:

$$E[Y_n] \leq \frac{2}{n} \sum_{i=1..n} (\, E[Y_{i-1}] + E[\, Y_{n-i} \,] \,) \text{ pour } n \geq 1 \qquad E[Y_0] = 1$$

Soit $f(n) = E[\, Y_n \,]$

$f(0) = 1$

$$f(n) \leq \frac{2}{n} \sum_{i=1..n} (\, f(i-1) + f(n-i) \,) = \frac{4}{n} \sum_{i=0..n-1} f(i) \qquad \text{pour } n \geq 1$$

Hauteur moyenne d'un ABR construit aléatoirement

Récurrence

$f(n) \leq \frac{4}{n} \sum_{i=0..n-1} f(i)$ pour $n \geq 1$ $\quad f(0)=1$

On peut démontrer par substitution que $f(n) \leq \binom{n+3}{3}$

base : $f(0) = 1 = \binom{0+3}{3}$

induction:

$$f(n) \leq \frac{4}{n} \sum_{i=0..n-1} \binom{i+3}{3} = \frac{4}{n} \binom{n+3}{4} = \frac{4\ (n+3)!}{n\ 4!\ (n-1)!} = \binom{n+3}{3}$$

$\Rightarrow$ $f(n) = E[\ 2^{Hn}\ ]$ est polynomiale en n: $O(n^3)$

Inégalité de Jensen: $2^{E[\ Hn\ ]} \leq E[\ 2^{Hn}\ ]$ (puisque la fonction 2^x est convexe)

$E[\ Hn\ ] \leq \log \binom{n+3}{3}$ $\quad E[\ Hn\ ] = O(\log n)$

Bibliographie

1) A.V. Aho, J.E. Hopcroft, J.D. Ullmann. (*types abstraits*)
Data Structures and Algorithms. Addison-Wesley.

2) Danièle Beauquier, Jean Berstel, Philippe Chrétienne. (*types abstraits, ABR*)
Éléments d'Algorithmique.
Masson, 1992. http://www-igm.univ-mlv.fr/~berstel/Elements/Elements.html

3) Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. (*structures de données*)
Introduction à l'algorithmique. 2e édition, Dunod, 2002.

4) Robert Sedgewick. (*analyse des implementations des dictionnaires*)
Algorithms in Java. Addison-Wesley.

5) Christine Froideveaux, Marie-Claude Gaudel et Michèle Soria.
Types de données et algorithmes. (*types abstraits, algorithmes de recherche séquentielle*)
Ediscience, 1993

6) J.V. Guttag, J.J. Horning.
The Algebraic Specification of Abstract Data Types (*types abstraits, approche algébrique*)
Acta Informatica, 1978