Algorithmique — 1. Structures de données

1. Types abstraits de données – structures linéaires

Bruno Grenet

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

Université Grenoble Alpes – IM²AG
L3 Mathématiques et Informatique

Introduction à la partie 1

Structures de données étudiées

- Rappels : tableau, liste (chaînée), arbre binaire
- Structures *linéaires* : pile, file, file à priorité, tas, tableau dynamique
- Dictionnaire : arbre binaire de recherche, table de hachage

Objectifs

- Description de la structure de données
- Analyse de ses performances
- Comparaisons entre différentes structures

Méthode

1. Identifier les opérations que la structure doit fournir *type abstrait de données*
2. Construire la structure de données en se basant sur autre plus simple
3. Analyser les coûts : espace nécessaire, complexité des opérations
4. Comparer avec d'autres structures similaires

Table des matières

Table des matières

Motivation

Écriture d'algorithmes

- Manipulation de données
- Abstraction vis-à-vis du matériel, langage de programmation, etc.
- *Modèle* des opérations permises et de leur complexité

Exemples

- Tri d'un tableau d'entiers
  - Accès et modifications des éléments
  - Calcul sur les entiers
- Algorithme de multiplication d'entiers
  - Représentation des entiers
  - Calcul sur les bits

- Tout algorithme se base sur un (ou des) type(s) abstrait(s) de données…
- … même si c'est la plupart du temps implicite

Qu'est-ce qu'un type abstrait de données ?

Ingrédients

- le nom du TAD
- des opérations sur le TAD
  - constructeurs — *créer un nouvel objet*
  - modificateurs — *modifier un objet*
  - observateurs — *obtenir de l'information sur un objet*
- Propriété importante : statique ou dynamique — *taille fixée ou variable*

Et c'est tout !

- Pas de représentation des données
- Pas d'implantation des opérations
- TAD = *interface* d'une structure de données

Exemples

Entiers : $+, -, \times, \div, <, >, \ldots$

Liste : estVide(), tête(), queue()

Chaîne : accès au $i^{\text{ème}}$ caractère, longueur, concaténation, ...

À quoi ça sert ?

Pourquoi travailler avec un type *abstrait* de données ?

Indépendance de l'implantation

- Un même algorithme fonctionne avec différentes implantations
- Implantation améliorée → performances améliorées

Se libérer l'esprit

- Fixer un nombre restreint d'opérations possibles… et s'y tenir !
- Se concentrer sur les fonctionnalités, pas les détails d'implantation

Complexité des structures de données

- Fixer des objectifs — *les opérations du* TAD
- Trouver la meilleure représentation possible
- Indépendamment d'un problème particulier à résoudre

Exemple 1 : le TAD « tableau »

Opérations

- $T \leftarrow$ nvTableau(n) : nouveau tableau de taille n (statique)
- écrire(T, i, x) : écriture en case i de T
- $x \leftarrow$ lire(T, i) : lecture de la case i de T
- taille(T) : longueur du tableau

Notations :
$T_{[i]} \leftarrow x$
$x \leftarrow T_{[i]}$
$\#T$

Complexités

- Taille de la représentation : $O(n)$
- nvTableau : $O(n)$
- Accès au $i^{\text{ème}}$ élément et taille(T) : $O(1)$

Remarques

- Souvent : type des entrées à préciser
- Similarité et différences avec une implantation pratique
  - Exemple en C : `malloc`, `T[i]` $\rightarrow$ quasi identique !
  - Complexité : en général cohérent… mais pas toujours $\rightarrow$ questions de cache par ex.

Exemple 2 : un TAD « liste » minimal

Opérations

dynamique

- $L \leftarrow$ nvListe() : nouvelle liste vide
- estVide(L)
- ajouter(L, x) : ajoute l'élément x en tête de L
- $x \leftarrow$ retirer(L) : retire le premier élément de L et le renvoie

Complexités

- Taille proportionnelle au nombre d'éléments
- Opérations en $O(1)$

Remarques

- Souvent : type des entrées à préciser
- Modélise deux structures classiques :
  - liste chaînée : structure à base de maillons et pointeurs
  - liste en prog. fonctionnelle : `x::L`

Autres exemples de TAD

TAD de base

- Exemples :
  - tableau, liste, liste doublement chaînée, …
  - arbre binaire, *n*-aire, … *cf.* cours 2
  - entier naturel, entier relatif, …
- Proches des types de base des langages ou d'implantation *bas niveau*
- Existence et complexités admises *modélisation*

TAD plus évolués

- Exemples :
  - pile, file, file à priorité
  - dictionnaire, ensemble, graphe
  - nombre rationnel, polynôme, …
- *Implantation* en générale nécessaire, à partir de types de base bibliothèques
- En algorithmique, *réalisation* de ces TAD à partir des TAD basiques
  - description des algorithmes
  - étude des complexités

Implantation et réalisation d'un TAD

Implantation dans un langage

- Définition d'un type de données du langage
- Programmation des opérations

Réalisation algorithmique

- Expliciter les algorithmes pour les opérations
- En se basant sur un ou plusieurs autres TAD *plus « basique »*
- Analyse de la complexité
  - Taille de la représentation
  - Complexité des opérations

Remarques

- Un TAD a plusieurs implantations et plusieurs réalisations possibles
- TAD *basiques* : existence et complexité des opérations comme hypothèses *modèle*
- Dans ce cours : uniquement des réalisations algorithmiques

Bilan sur les TAD

Rien de nouveau, mais formalisé !

- Tableaux, listes (chaînées), arbres binaires → TADs
- Implicite en l'absence de difficulté
- Besoin d'explicite pour des types plus complexes *ensemble*, *graphe*, …

Pas de TADs « officiels »

- Définition d'un TAD *en fonction des besoins*
- Quelques *classiques* tableau, liste, pile, …
- Variations d'un cours à l'autre, d'un livre à l'autre, etc.
  - en particulier les noms varient

Dans le cours d'algorithmique

- Étude de quelques TAD classiques
  - Types linéaires : file, pile, file à priorité
  - Ensemble et dictionnaire
- Utilisation (implicite ou explicite) de TAD dans les algorithmes

Table des matières

Les structures de données *linéaires*

Définition informelle

- Ensemble de données rangé séquentiellement
- Chaque élément a une position, les autres étant *avant* ou *après*
- Aucune autre hiérarchie entre les éléments

Exemples et contre-exemples

- Liste, tableau $\rightarrow$ structures linéaires
- Arbre binaire $\rightarrow$ structure non linéaire

Plusieurs structures linéaires

- Comment insérer / extraire des éléments
- Comment déterminer les positions

Pile : « dernier arrivé premier servi »

Description informelle

- Analogies : pile d'assiette, tas de cartes, …
- Insertion et extraction en *haut de la pile* uniquement
- Politique LIFO : *Last In First Out*

Utilité

- Pile d'appel de fonctions, opérations *annulables* dans un logiciel, …
- Exemples d'algorithmes :
  - Parcours d'arbres / graphes avec retour en arrière
  - Parenthésage / évaluation d'expressions mathématiques

Le TAD Pile

Opérations

- nvPile() : nouvelle pile vide
- estVide(P)
- empiler(P, x) : ajout de x en haut de la pile P
- dépiler(P) : renvoie l'élément en haut de P et le supprime de P

dynamique

Remarque

- Le TAD Pile est quasiment identique au TAD Liste

Réalisation basée sur une liste

- Représentation d'une pile par une liste

estVide fourni par le TAD liste

nvPile():
1. renvoyer nvListe()

empiler(P, x):
1. ajouter(P, x)

dépiler(P):
1. renvoyer retirer(P)

File : « premier arrivé premier servi »

Description informelle

- Analogie : file d'attente à la caisse d'un magasin
- Insertion à la fin de la file et extraction au début
- Politique FIFO : *First In First Out*

Utilité

- Files d'attente informatique *imprimante, service web, …*
- Exemple d'algorithme : parcours d'arbres / graphes en largeur

Le TAD File

Opérations

- nvFile() : nouvelle file vide
- estVide(F)
- enfiler(F, x) : ajout de x en fin de file F
- défiler(F) : renvoie l'élément au début de F et le supprime de F

dynamique

Réalisation basée sur liste

- Représentation d'une file par une liste
- Trois opérations en complexité $O(1)$:

  nvFile():
  1. renvoyer nvListe()

  défiler(F):
  1. renvoyer retirer(F)

début de file = tête de liste

estVide fourni par le TAD liste

- Problème pour enfiler :
  - insertion en tête dans une liste
  - insertion en fin dans une file

TAD File : réalisation d'ENFILER

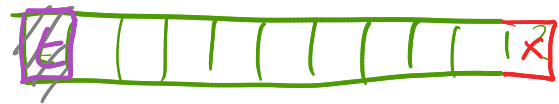
ENFILER(F, x) :
1. Si ESTVIDE(F) : AJOUTER(F, x)
2. Sinon :
3. $t \leftarrow$ RETIRER(F)
4. ENFILER(F, x) *appel récursif*
5. AJOUTER(F, t)

Complexité

- Parcours de toute la file : $O(n)$

(*équation de récurrence :* $T(n) = T(n-1) + O(1)$)

$T(n) = T(n-1) + c = T(n-2) + 2c = \dots = T(0) + cn = O(n)$



Réalisation plus efficace ?

- TAD liste enrichi, avec accès aux deux extrémités
- Utilisation de deux piles *cf.* TD
- Réalisation basée sur les tableaux *cf.* diapo 22

Remarque : deux choix symétriques

- tête de liste = début de file : DÉFILER en $O(1)$, ENFILER en $O(n)$
- tête de liste = fin de file : ENFILER en $O(1)$, DÉFILER en $O(n)$

File de priorité : « plus prioritaire premier servi »

Description informelle

- Analogie : embarquement dans un avion *Business class*, familles, etc.
- Chaque élément inséré a une priorité, on extrait toujours *le plus prioritaire*

Utilité

- Ordonnancement des processus dans un système d'exploitation
- Exemples d'algorithmes :
  - algorithmes de tri tri par tas
  - compression de données
  - recherche de plus court chemins

Le TAD File de priorité

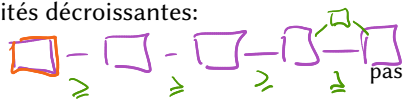
Opérations

- nvFilePriorité() : nouvelle file de priorité vide — dynamique
- estVide(F)
- insérer(F, x, p) : insertion de x avec priorité p dans F
- extraire(F) : renvoie un élément de priorité maximale et le supprime de F

Réalisation basée sur une liste

- Représentation par une liste de couples (x, p)
- nvFilePriorité(): renvoyer nvListe() — estVide fourni par le TAD liste — $O(1)$
- Deux possibilités pour insérer/extraire :
  - liste non ordonnée :
    - insérer en $O(1)$
    - extraire en $O(n)$ — recherche

  - liste ordonnée par priorités décroissantes:
    - extraire en $O(1)$
    - insérer en $O(n)$ — pas de recherche dichotomique !



Réalisations à base de tableau

Principes

Objectif : Simplifier les opérations en utilisant un tableau sous-jacent

Avantage : accès en temps $O(1)$ à n'importe quel élément

Inconvénient : TAD *statique* pour réaliser des TAD *dynamiques*

Solution : Utilisation d'un tableau de taille fixée

- Perte de place $\rightarrow$ tableau plus grand que le strict nécessaire
- Perte de généralité $\rightarrow$ taille maximale fixée à l'avance

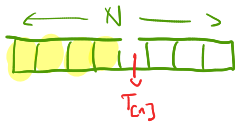
Idée générale

- Un tableau de taille N pour stocker n éléments $\quad n \leq N$
- Information en plus pour savoir où sont les éléments dans le tableau
  - Pile : cases 0 à $n-1$
  - File : cases contigües, pas forcément au début
  - File de priorité : à voir…

Piles basées sur des tableaux

Réalisation de Pile

- Représentation d'une pile par un couple (T, n)
- nvPile() :
  1. $T \leftarrow$ nvTableau(N)
  2. renvoyer $(T, 0)$
- estVide(P) :
  1. renvoyer « $n = 0$? »
- dépiler(P) :
  1. $n \leftarrow n - 1$
  2. renvoyer $T_{[n]}$



- empiler(P, x) :
  1. si $n < N$:
  2. $\quad T_{[n]} \leftarrow x$
  3. $\quad n \leftarrow n + 1$
  4. sinon :
  5. $\quad$ *erreur (la pile est pleine)*

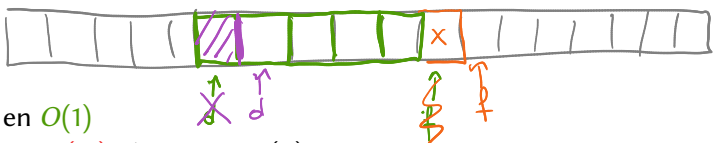
Remarques

- Toutes les opérations en $O(1)$
- Taille de représentation : $O(N)$ plutôt que $O(n)$
- Manque de généralité

Files basées sur des tableaux

Réalisation de File

- Représentation d'une file par un triplet (T, d, f)
- nvFile() :
  1. $T \leftarrow$ nvTableau(N)
  2. renvoyer $(T, 0, 0)$
- estVide(F) :
  1. renvoyer « $f - d = 0$? »
- défiler(F) :
  1. $d \leftarrow d + 1$
  2. renvoyer $T_{[d-1]}$
- enfiler(F, x) :
  1. si $f < N$:
  2. $\quad T_{[f]} \leftarrow x$
  3. $\quad f \leftarrow f + 1$
  4. sinon :
  5. $\quad$ *erreur (la file est pleine)*



Remarques

- Toutes les opérations en $O(1)$
- Taille de représentation : $O(N)$ plutôt que $O(n)$
- Manque de généralité
  - Tableau utilisé de manière *circulaire*

cf. TD

Bilan sur les piles, files, files à priorité

Similarités et différences

- *Syntaxiquement* très proches — ajout / suppression
- *Sémantiquement* différentes
  - Remarque : Pile/File peuvent être vues comme de Files de priorité

Pile : priorité = ordre d'insertion
File : priorité = ordre inverse d'insertion

Réalisations à base de liste

Pile : Quasiment une liste, opérations en $O(1)$

File : Réalisation directe inefficace → ENFILER ou DÉFILER en $O(n)$

Avec un TAD Liste enrichi → complexité $O(1)$

File de priorité : Inefficace → INSÉRER ou EXTRAIRE en $O(n)$

Réalisations à base de tableau

- Inconvénient majeur : taille fixée à l'avance
- Avantage : opérations de Pile et File en $O(1)$
- À résoudre : réalisation des files de priorité

Bilan sur les piles, files, files à priorité

Similarités et différences

- *Syntaxiquement* très proches — ajout / suppression
- *Sémantiquement* différentes
  - Remarque : Pile/File peuvent être vues comme de Files de priorité

Réalisations à base de liste

Pile : Quasiment une liste, opérations en $O(1)$

File : Réalisation directe inefficace → ENFILER ou DÉFILER en $O(n)$

Avec un TAD Liste enrichi → complexité $O(1)$

File de priorité : Inefficace → INSÉRER ou EXTRAIRE en $O(n)$

Réalisations à base de tableau

- Inconvénient majeur : taille fixée à l'avance — tableaux *dynamiques* (cours 3)
- Avantage : opérations de Pile et File en $O(1)$
- À résoudre : réalisation des files de priorité — *tas* (cours 2)

Quel TAD choisir ?

Problème : reconnaître les expressions *bien parenthésées*

Exemples (ou pas ?)

- $[(a-(b+c))\times(-3)]/\{a-b\}$
- `<p><b>Attention</b> à bien <i>fermer</i> les balises<br/></p>`
- $(3\times 2-(3-7\times(6+3\times(1-4))))$

Idée d'algorithme

- Parcourir de gauche à droite
  - en *retenant* les parenthèses ouvertes
  - en *vérifiant* lorsqu'une parenthèse est fermée
- Bonne structure de donnée (TAD) : Pile

L'algorithme

estBienParenthésée(*expr*) :

1. $P \leftarrow$ nvPile()
2. pour tout caractère c de *expr* : — caractère ou *token* (ex. html)
3. si c *est une parenthèse ouvrante* :
4. Empiler(P, c)
5. sinon si c *est une parenthèse fermante* :
6. si estVide(P) : renvoyer faux
7. $x \leftarrow$ dépiler(P)
8. si x *ne correspond pas à* c : renvoyer faux
9. renvoyer estVide(P)

Analyse

- Complexité : $O(\text{longueur}(expr))$
- Correction : admise — définition de « bon parenthésage »

Conclusion

Type abstrait de données

- Décrit un ensemble de *données* et les opérations qu'on effectue dessus
- Ne spécifie pas la représentation des données ni l'implantation des opérations

→ Focus sur les fonctionnalités

Usage en algorithmique et dans ce cours

- Description de TADs *classiques*
  - Utilisés fréquemment et implantés dans beaucoup de (bibliothèques de) langages
  - Réalisation algorithmique de TADs
- Description d'algorithmes basés sur les TADs

→ Fixe les limites des opérations autorisées dans un pseudo-code

C'est nouveau pour vous ?

Non vous en utilisez depuis longtemps sans le savoir…

Oui nouveau cadre formel d'étude des algorithmes

Bilan sur les tableaux et les listes

Deux structures de base en algorithmique

Tableau : représentation *statique* de n éléments ; accès en $O(1)$

Liste : représentation *dynamique* de n éléments ; accès en $O(n)$

Tableaux et listes comme TAD

- Utilisés dans ce cours pour construire d'autres TAD
- Proches des implantations pratiques en programmation

À réviser si vous n'êtes pas à l'aise !

- Exemples :
  - calculer le maximum dans un tableau ou une liste
  - recherche d'un élément dans un tableau trié ou dans une liste triée
- Noms exacts des opérations pas importants
  - NVTABLEAU(n) ou « Créer un tableau de taille n » ou …
  - AJOUT(L, x) ou « Ajouter un élément x en tête de L » ou …

→ **il faut** (et il suffit) **que ce soit clair !**