Partie 1. Structures de données

1. Types abstraits de données et rappels

Bruno Grenet

Université Grenoble Alpes – IM²AG
L3 Mathématiques et Informatique
UE Algorithmique

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

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 parfois (souvent !) implicite

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

Ingrédients

- le nom du TAD et la description des objets représentés
- des opérations
  - 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

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

Notions d'implantations

Implantation dans un langage

- Définition d'un type de données du langage
- Programmation des opérations
- Indépendant du type de programmation — impératif, objet, fonctionnel, …

« Implantation » algorithmique

- Expliciter la représentation et les algorithmes pour les opérations
- Basée sur un (ou d')autre(s) TAD — *plus « basique »*
- Analyse de la complexité
  - Taille de la représentation
  - Complexité des opérations

Remarques

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

Un exemple : les entiers

Dans des langages

- Entiers bornés (par ex. 64 bits), signés ou non signés — C, C++, Java, OCaml, …
- Entiers non bornés, en général signés — Python

Un TAD « entier » possible

- Description : les entiers relatifs ($\mathbb{Z}$)
- Opérations :
  - Addition, soustraction, multiplication, division euclidienne
  - Comparaisons entre entiers

Quelles implantations ?

- Programmation : `int` en Python, bibliothèques `BigInteger` en Java, `GMP` en C/C++, …
- Algorithmique :
  - Tableau de booléens, et algorithmes de calcul
  - Tableau d'entiers bornés, et algorithmes de calcul
  - Entiers *récursifs* sous forme d'arbre binaire
  - …

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 ensembles disjoints
- Utilisation (implicite ou explicite) de TAD dans les algorithmes

Table des matières

Les tableaux comme un TAD

Définition

- Ensemble de données indexées par un entier
- Opérations :
  - $T \leftarrow \text{NouveauTableau}(n)$: nouveau tableau de taille n — statique
  - $T_{[i]}$: accès au $i^{\text{ème}}$ élément en lecture & écriture
  - $\text{Taille}(T)$: longueur du tableau — Notation : $\#T$

Complexités

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

Remarques

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

Listes (chaînées) comme un TAD

Définition

- Ensemble de données organisé de manière séquentielle
- Opérations :
  - $L \leftarrow$ NouvelleListe() : nouvelle liste vide
  - EstVide(L)
  - Tête(L) : premier élément ; Queue(L) : liste privée de la tête
  - Ajout(L, e) : ajoute l'élément e en tête de L

dynamique

Complexités

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

Remarques

- Souvent : type des entrées à préciser
- Modélise deux structures classiques :
  - Impératif : structure à base de maillons et pointeurs
  - Fonctionnel : listes *à la* OCaml `x::l`

Autres opérations sur les listes

Exemples d'opérations standard

- Parcours de la liste — *itérateur*
- Longueur de la liste
- Recherche d'un élément — par sa valeur, min / max, …
- Insertion d'un élément en fin de liste / à un endroit donné

Complexités

- Dans le TAD précédent : toutes en $O(n)$ — $n =$ taille de la liste
- On peut aussi définir des TAD enrichis, par ex.:
  - Stockage de la longueur de la liste $\rightarrow O(1)$
  - Pointeur vers la fin de la liste $\rightarrow O(1)$ pour l'insertion en fin

- Il n'y a pas **un seul** TAD possible, tout est question de choix de modélisation
- Les TAD enrichis sont implantés (algorithmiquement) à partir du TAD de base

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

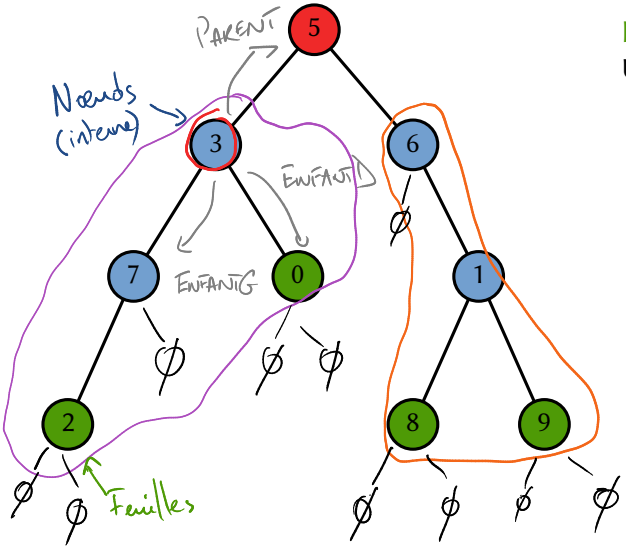
À réviser si vous n'êtes pas à l'aise !

- 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
  - NouveauTableau(n) ou « Créer un tableau de taille n » ou …
  - Ajout(e, L) ou « Ajouter un élément e en tête de L » ou …

  → il faut et il suffit que ce soit clair !

Table des matières

Vocabulaire



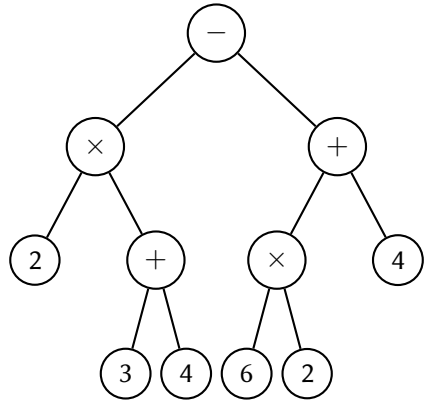
Définition

Un arbre binaire est défini récursivement :

- soit l'arbre vide $\emptyset$
- soit constitué de 3 éléments :
  - la racine — *donnée*
  - le sous-arbre gauche — *arbre binaire*
  - le sous-arbre droit — *arbre binaire*

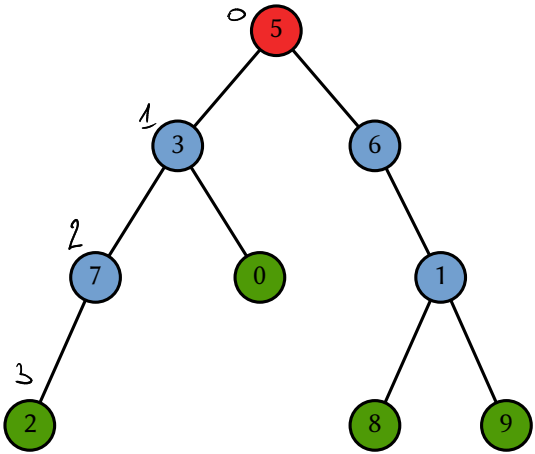
Utilité des arbres binaires

- Arbres binaires de recherche
- Tas
- Analyse syntaxique
- Bases de données
- Partition binaire de l'espace
- Tables de routage
- …



$$2 \times (3 + 4) - (6 \times 2 + 4)$$

Hauteur et niveaux



Définition

- Hauteur $h(x)$ d'un nœud x dans A :
  - 0 si x est la racine de A
  - $1 + h(p)$ sinon, où p est le parent de x
- Hauteur d'un arbre A :
  $h(A) = \max\{h(x) : x \in A\}$

Lemme

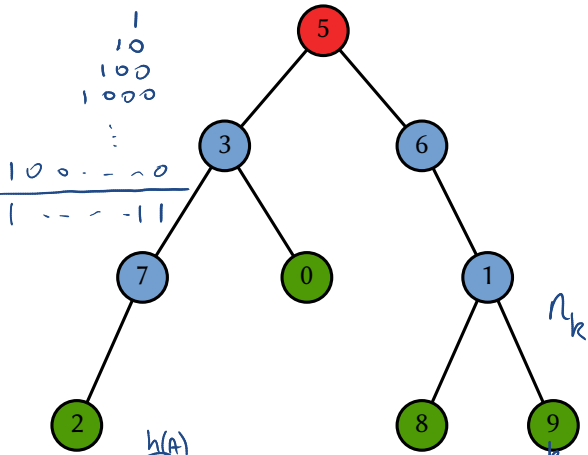
$\#\{x : h(x) = k\} \leq 2^k$

$k=0$: uniquement la racine ✓

$k>0$: On supp. $\#\{x : h(x) = k-1\} \leq 2^{k-1}$

Or un nœud de hauteur $k-1$ a ≤ 2 enfants. Donc $\#\{x : h(x) = k\} \leq 2 \times 2^{k-1}$

Hauteur et niveaux



$n_k =$

$n = \sum_{k=0}^{h(A)} n_k$ or $1 \leq n_k \leq 2^k$

Donc $\sum_{k=0}^{h(A)} 1 \leq n \leq \sum_{k=0}^{h(A)} 2^k$

Définition

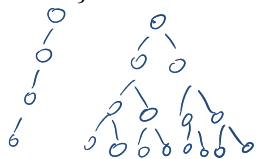
- **Hauteur** $h(x)$ d'un nœud x dans A :
  - 0 si x est la racine de A
  - $1 + h(p)$ sinon, où p est le parent de x
- **Hauteur** d'un arbre A :
  $h(A) = \max\{h(x) : x \in A\}$

Lemme

$\#\{x : h(x) = k\} \leq 2^k$

Lemme

$1 + h(A) \leq n \leq 2^{1+h(A)}$

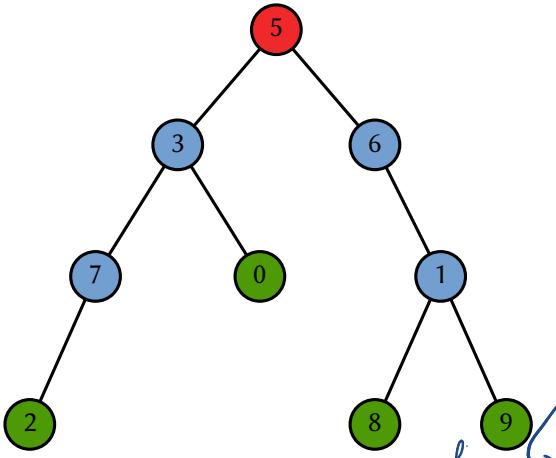


où $n = \#A$

Donc $n \geqslant h(A) + 1$

et $n \leq \sum_{k=0}^{h(A)} 2^k = 2^{h(A)+1} - 1$

Hauteur et niveaux



Définition

- Hauteur $h(x)$ d'un nœud x dans A :
  - 0 si x est la racine de A
  - $1 + h(p)$ sinon, où p est le parent de x
- Hauteur d'un arbre A :
  $h(A) = \max\{h(x) : x \in A\}$

Lemme

$\#\{x : h(x) = k\} \leq 2^k$

Lemme

$1 + h(A) \leq n < 2^{1+h(A)}$ où $n = \#A$

Corollaire

$\lfloor \log(n) \rfloor \leq h(A) < n$

$-\log n < 1 + h(A)$

Un TAD « arbre binaire »

Définition

- Ensemble de données organisé de manière récursive
- Opérations :
  - ArbreVide() : nouvel arbre vide — dynamique
  - CréerArbre(r, G, D) : nouvel arbre de racine r, et d'enfants G et D
  - estVide(A) : test
  - Racine(A), EnfantG(A), EnfantD(A) : racine et enfants — *uniquement si* $\neg$estVide(A)

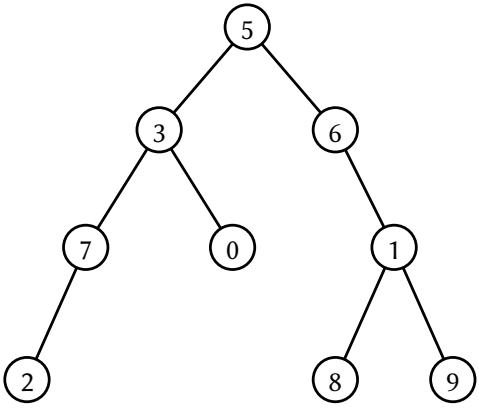
Complexités

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

Remarques

- Souvent : type des données à préciser
- TAD proche de l'implantation pratique — *nœuds* et *pointeurs*
- Opération parfois ajoutée : Parent(A) en $O(1)$

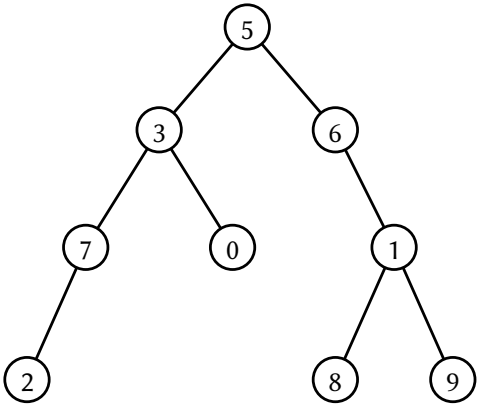
L'outil de base : le parcours en profondeur



ParcoursInfixe(A) :

1. Si non(estVide(A)) :
2. ParcoursInfixe(EnfantG(A))
3. *Traiter* Racine(A)
4. ParcoursInfixe(EnfantD(A))

L'outil de base : le parcours en profondeur



ParcoursInfixe(A) :

1. Si non(estVide(A)) :
2. ParcoursInfixe(EnfantG(A))
3. *Traiter* Racine(A)
4. ParcoursInfixe(EnfantD(A))

Correction

Chaque sommet est traité une fois et une seule

Complexité

$O(n)$ appels à *Traiter* $\qquad n = \#A$

Ordre de traitement

2 7 3 0 5 6 8 1 9

Deux variantes

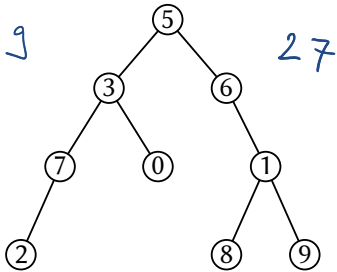
ParcoursPréfixe(A) :

1. Si non(estVide(A)) :
2. *Traiter* Racine(A)
3. ParcoursPréfixe(EnfantG(A))
4. ParcoursPréfixe(EnfantD(A))

ParcoursPostfixe(A) :

1. Si non(estVide(A)) :
2. ParcoursPostfixe(EnfantG(A))
3. ParcoursPostfixe(EnfantD(A))
4. *Traiter* Racine(A)

5 3 7 2 0 6 1 8 9

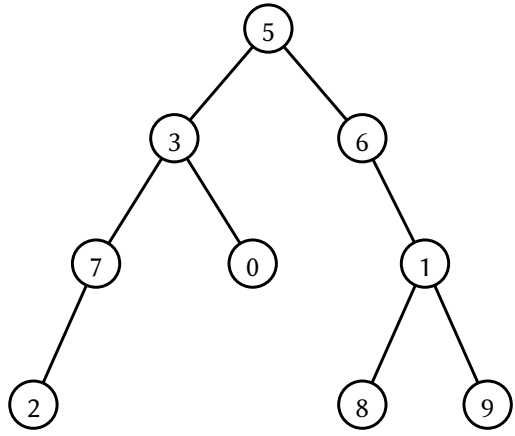


2 7 0 3 8 9 1 6 5

Exemples d'algorithmes

Minimum(A) :

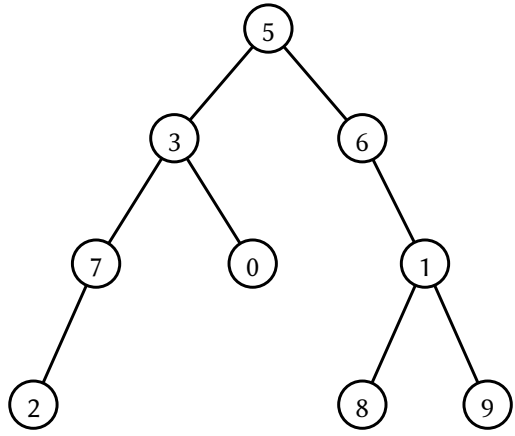
1. $m \leftarrow +\infty$
2. Si Non(estVide(A)) :
3. $\quad m_G \leftarrow$ Minimum(EnfantG(A))
4. $\quad m_D \leftarrow$ Minimum(EnfantD(A))
5. $\quad m \leftarrow \min(m_G, m_D, \text{Racine}(A))$
6. Renvoyer m



Exemples d'algorithmes

NbNœuds(A) :

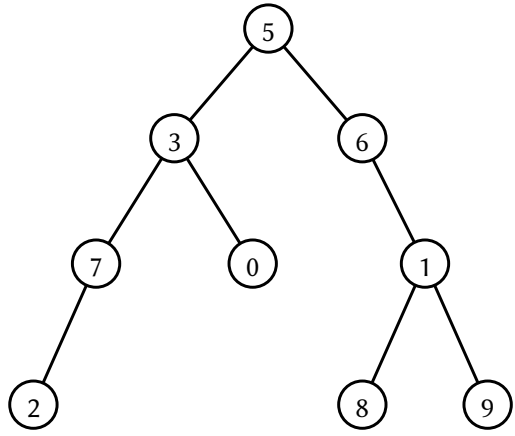
1. $n \leftarrow 0$
2. Si Non(estVide(A)) :
3. $\quad n_G \leftarrow$ NbNœuds(EnfantG(A))
4. $\quad n_D \leftarrow$ NbNœuds(EnfantD(A))
5. $\quad n \leftarrow n_G + n_D + 1$
6. Renvoyer n



Exemples d'algorithmes

HAUTEUR(A) :

1. $h \leftarrow 0$
2. Si NON(ESTVIDE(A)) :
3. $\quad h_G \leftarrow$ HAUTEUR(ENFANTG(A))
4. $\quad h_D \leftarrow$ HAUTEUR(ENFANTD(A))
5. $\quad h \leftarrow 1 + \max(h_G + h_D)$
6. Renvoyer h

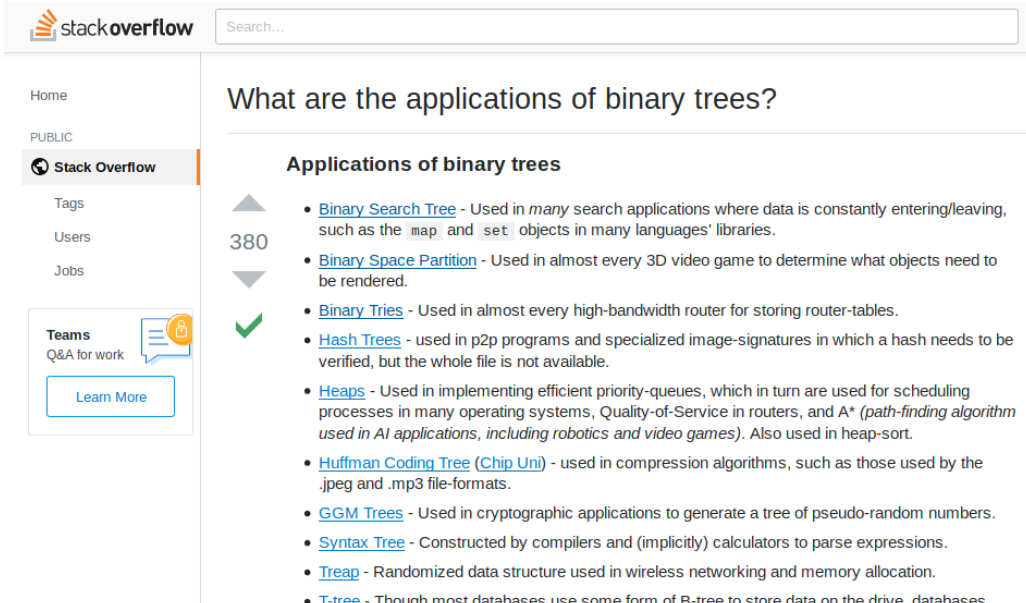


Bilan sur les arbres binaires

Une (autre) structure de base en algorithmique

- Représentation structurée de l'information
  - arbres binaires de recherche
  - tas
  - …

Bilan sur les arbres binaires



Bilan sur les arbres binaires

Une (autre) structure de base en algorithmique

- Représentation structurée de l'information
  - arbres binaires de recherche
  - tas
  - …

Arbres binaires comme TAD

- Utilisé dans ce cours pour construire d'autres TAD
- Proche des implantations pratiques en programmation

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

- Exemples :
  - compter le nombre de feuilles
  - recherche d'un élément particulier

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

→ 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