M1 ALMA
Université de Nantes
2011-2012

Rapport de Projet : Programmation générative

Marguerite Alain, Pluchon Gwenael
Rincé Romain, Sebari Nabila

Université de Nantes
2 rue de la Houssinière, BP92208, F-44322 Nantes cedex 03, FRANCE

Table des matières

1 Introduction

Le projet réalisé s'inscrit dans le cadre d'une démarche d'approfondissement des notions abordées en cours de programmation générative. Ce projet propose, dans sa première partie, l'étude et l'implémentation de deux cas concrets :

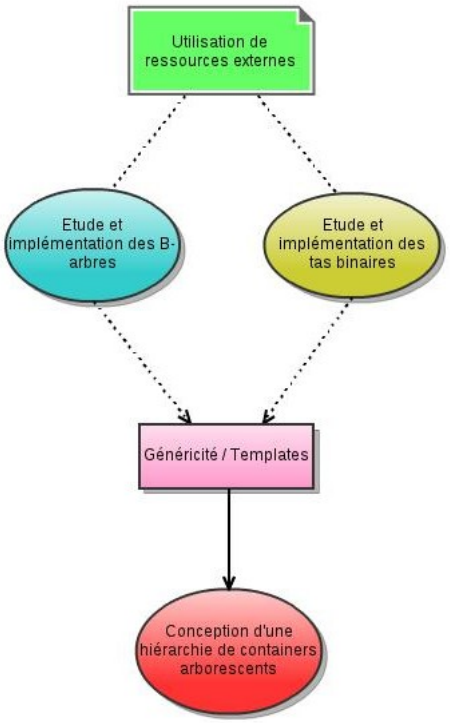

FIGURE 1.1 – Organisation du projet

- Les tas binaires
- Les B-arbres

À partir de cette première étape, nous avons, dans une deuxième partie, porté notre analyse sur l'étude d'une hiérarchie plus complète pouvant englober les deux cas concrets

cités précédemment ainsi que d'autres implémentations d'arborescences comme les tas binaires, tas binomiaux, B-arbre, arbre rouge-noir etc...

Le but de la démarche de ce projet est d'aboutir sur des programmes robustes et fiables en mettant en oeuvre les principes vus en cours et en TD à savoir la généricité, l'utilisation des templates ainsi qu'une gestion rigoureuse de la mémoire.

Le but de la démarche de ce projet est d'aboutir sur des programmes robustes et fiables en mettant en oeuvre les principes vus en cours et en TD à savoir la généricité, l'utilisation des templates ainsi qu'une gestion rigoureuse de la mémoire.

2 Cas concret : Tas binaire

2.1 Choix de la structure concrète

Nous avons recherché la structure concrète la plus performante dans l'ensemble des opérations. Ne pouvant savoir quelle serait la nature de l'utilisation de cette classe, nous avons opté pour une structure offrant les meilleures performances pour l'implémentation du tas. Plusieurs solutions se présentaient pour une implémentation de tas binaire :

Tableau des pères Non approprié (dédié aux arborescence non-ordonnées).

Une liste d'adjacence Un noeud étant un élément (étiquette, père, liste des fils) de la liste. Ses performances sont dégradées lors d'opération comme la suppression ($O(n)$ pour un tas binaire) ou l'accès à un noeud à partir de son étiquette ($O(n)$). Dans le cas d'une utilisation où les opérations principales aurait été l'ajout d'un fils par exemple, cette solution aurait pu être retenue.

Liste de fratrie Un noeud étant un élément (étiquette, père, liste des frères, premier fils) de la liste.

Tableau modulaire definition manquante Structure présentant les meilleurs complexités pour l'ensemble des opérations. Cette solution répond le mieux aux contraintes que nous avons exposés précedement. C'est donc elle que nous avons retenue.

2.2 Développement du tas binaire

Une fois la sélection de la structure faite. Nous avons au cours des semaines adopté le plan de développement suivant :

1. Définition des prototypes des méthodes
2. Implémentation des méthodes
3. Rajout de l'itérateur

4. Rajout de l'allocateur et du comparator
5. Debug

Cette démarche nous a permis d'avancer rapidément au cours des première semaine. Les cours d'algorithmiques du 1er semestre encore en tête. Nous avons rapidement déployées les méthodes principales du tas binaire (Extraire, Tasser, Insérer...), sans trop nous soucier de la syntaxe C++ (nouvelle pour notre bînome). Nous avons amèrement regretté notre insoucience de ne jamais tester nos méthodes avant la 5ème ou 6ème semaine. Nous avons donc passé un temps important à débuguer un nombre de d'erreurs de syntaxe conséquent.

La création d'un itérateur a été un réel frein dans le développement du projet. Notre premier objectif était de construire un itérateur permettant de parcourir les valeurs dans un ordre trié dépendant de la classe `Compare`. Cependant un tel itérateur nécessitait de maintenir une structure de taille égale à celle du tas, ce qui s'avérait coûteux en mémoire. Nous avons donc décidé de nous orienter vers un itérateur `DFS` plus classique. Malgré tout nous avons eu du mal a maitriser la syntaxe C++ lié aux itérateurs et ils nous a fallu malgré tout un certain temps.

3 Cas concret : B-arbres

3.1 Définitions

Un B-arbre est une structure de données en arbre équilibré. Le principe général de cette structure est la minimisation de la taille des arbres en permettant aux noeuds la possession de plusieurs clés. Un B-arbre est donc caractérisé par les propriétés suivantes :

- Un ordre m
- Chaque noeud contient k clés triées avec :
  - pour le noeud racine : $m \leq k \leq 2m$
  - pour un noeud non racine : $1 \leq k \leq 2m$
- Un noeud est soit :
  - Terminal et donc est une feuille.
  - Non terminal et possède alors $k + 1$ fils. Les clés du $i^{\text{ème}}$ fils ont des valeurs comprises entre les valeurs des $(i-1)^{\text{ème}}$ et $i^{\text{ème}}$ clés du père.
- Chaque chemin de la racine à une feuille est de même longueur h (hauteur).

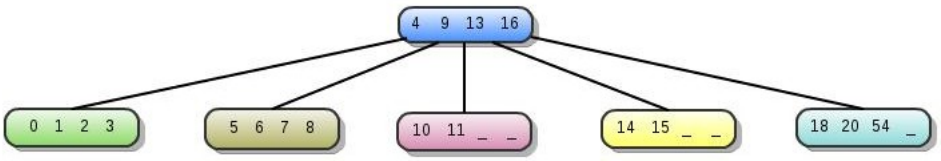

FIGURE 3.1 – Exemple de B-arbre

Étant donné la forme dont sont stockées les données dans les B-arbres (tri) et l'exécution des opérations d'insertion et de suppression en temps amorti logarithmique, cette structure est principalement mise en œuvre dans les mécanismes de gestion des bases de données et des systèmes de fichiers.

3.2 Implémentations

3.2.1 Méthodologie et choix de structure

En considérant les propriétés des B-arbres décrites précédemment, nous avons procédé dans la construction de nos structures/algorithmes de la manière suivante :

1. Mise en place de classes génériques en utilisant le principe des "templates" : L'objectif de cette approche est de générer le code spécifique à chaque type à partir d'un modèle générique. L'intérêt de cette technique est double : nous disposons d'un code optimisé pour chaque type et le code source n'est pas dupliqué car il écrit un méta-code à partir duquel sont générés les codes spécifiques. Nous avons donc paramétré nos arbres par le type de données à stocker et nous avons choisi de fixer l'ordre du B-Arbre.

```
1 //T un type quelconque et k l ordre de l arbre
  template<class T,short k>
```

2. Mise en place de la structure de noeud "`BTreeItem`" : Cette structure comporte les éléments suivants :
   - Deux variables `k` et `nCount` de types entier court représentant respectivement l'ordre de l'arbre et le nombre de clés stockés dans un noeud.
   - Un tableau `data` de $2 \times k$ clés rangées dans l'ordre croissant.
   - Un tableau `subItems` de $2 \times k + 1$ pointeurs vers les enfants du noeud lorsque celui-ci n'est pas une feuille.

   Cette structure possède également deux fonctions qui nous serviront dans la méthode d'insertion dans le B-arbre :
   - Une fonction de recherche dichotomique[Wik] `searchInNode`.
   - Une fonction de "découpage" de noeud `SplitNode`.
3. Mise en place de la structure d'arbre complète `BTree` : cette structure comporte les fonctions nécessaires pour la recherche d'un élément dans l'arbre, la suppression ainsi que l'ajout. Nous verrons ces fonctionnalités plus en détails dans les parties qui suivent.

3.2.2 Méthode de recherche

La recherche est effectuée de la même manière que dans un arbre binaire de recherche, d'où l'utilisation de la fonction de recherche dichotomique citée précedemment. Partant

de la racine, nous parcourons récursivement l'arbre. À chaque nœud, nous choisissons le sous-arbre fils dont les clés sont comprises entre les mêmes bornes que celles de la clé recherchée.

Algorithme Nous avons la structure de noeud suivante :

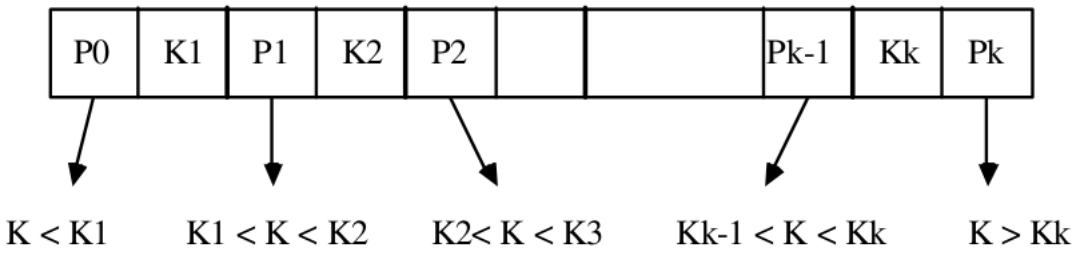

FIGURE 3.2 – Représentation de la structure

avec $K_1 < K_2 < K_3 < \cdots < K_n$ clés et P_0,...,P_k pointeurs vers noeuds enfants.

Pour chercher une clé k dans le B-arbre, l'algorithme récursif a été imaginé à partir des grandes étapes suivantes :

- Une fonction de recherche dichotomique[Wik] `searchInNode`.
- Une fonction de "découpage" de noeud `SplitNode`.

4 Implémentation d'une librairie sur les arbres

Étant donné le retard pris par nos deux binôme dans la réalisation du projet, nous avons préféré effectuer une analyse sur une hiérarchie permettant de représenter les différentes implémentations d'arborescences plutôt que d'essayer de produire du code qui ne serait certainement pas terminé au terme de ce projet.

Dans un premier temps nous proposerons donc un modèle de hiérarchie sur les arborescences qui pourrait permettre, par la suite, une éventuelle implémentation d'une librairie. Puis nous discuterons sur les moyens pouvant être mis en oeuvre pour automatiser le choix des structures.

4.1 Une possible hiérarchie sur les arbres

Introduction Dans cette partie, nous allons détailler une hiérarchie d'arborescence qui nous paraît plausible. Pour vous permettre de visualiser cette dernière plus aisément reportez vous à la figure 4.1

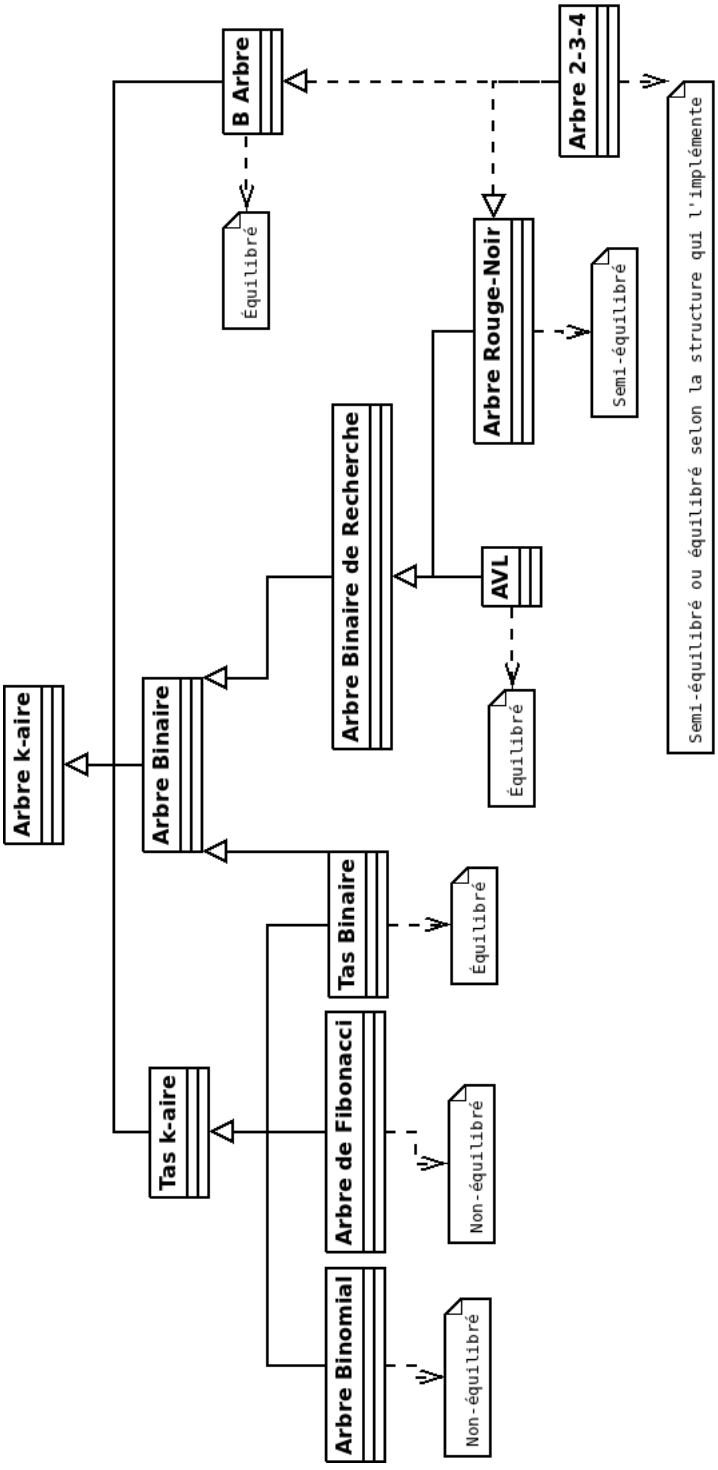

FIGURE 4.1 – Une hiérarchie sur les arbres

Justification du modèle

Arbre k-aire Dans notre modèle, l'arbre k-aire est la classe/interface père de toutes les autres. Cette appellation n'est peut-être pas la plus pertinente puisqu'il s'agit en fait d'une structure arborescente ou chaque nœud peut avoir f fils et contenir e éléments. Cette structure peut être une interface comme une classe concrète mais elle ne présente que peu d'intêret pour ce dernier cas puisqu'elle n'offre aucun avantage algorithme (complexité etc...)

Arbre binaire Il s'agit d'une spécification de l'arbre k-aire ou le nombre de nœuds ne peut contenir qu'un élément et ne posséder que deux fils (en excluant le cas des feuilles). Tout comme l'arbre k-aire, il sera sûrement implémenter comme une interface ou tout du moins comme une classe abstraite.

Tas k-aire La structure du tas spécifie l'arbre k-aire en forçant le nombre d'éléments dans chaque nœud à un et en ajoutant la relation d'ordre entre la valeur du nœud père et celle de ces fils. Là encore on peut imaginer qu'il s'agira d'une classe abstraite.

B arbre Le B arbre spécifie l'arbre k-aire en permettant de contenir dans un nœud un nombre d'élément nécessairement compris entre e et $\frac{e}{2}$. De plus le nombre de fils est dépendant du nombre e' d'éléments présents à un moment donné dans le nœud puisqu'il est au maximum de $e' + 1$. Il s'agit d'ailleurs d'une classe concrète implémentée dans la première partie du sujet.

Toutes les structures suivantes peuvent être considéré comme étant à implémenter en tant que classes concrètes :

Arbre binomial et de Fibonacci Il s'agit de structures très proches du tas k-aire permettant, respectivement, l'implémentation d'un tas binomial et de Fibonacci.

Tas binaire Le tas binaire est une implémentation concrète possédant à la fois les propriétés des arbres binaires et celle des tas.

Arbre binaire de recherche Cette structure de données spécifie l'arbre binaire en imposant une organisation des valeurs dans l'arbre.

AVL L'AVL est une structure spécifiant l'arbre binaire de recherche en forçant celui-ci à être équilibré.

Arbre rouge-noir Spécification particulière de l'arbre binaire de recherche.

Arbre 2-3-4 Il s'agit d'une structure de données pouvant être implémentée soit par un B arbre avec le nombre maximal d'élements dans un nœud fixé à 4 ; soit par un arbre rouge-noir, ce dernier étant isomorphe à l'arbre 2-3-4.

Table des figures

Bibliographie

[Wik] Wikipedia. Dichotomie.