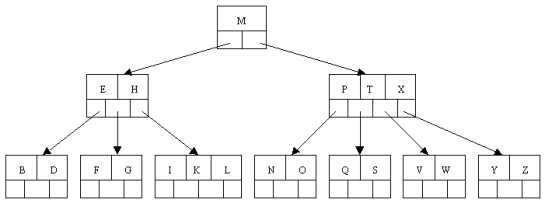
Indexation de fichiers à l'aide de B-Arbres

Présentation du problème (d'après *Introduction à l'algorithmique* – Cormen, Leiserson, Rivest & Stein – Édition Dunod)



B-Arbre d'ordre 2

Les B-arbres d'ordre *k* sont des arbres dont les nœuds internes contiennent une suite ordonnée d'au plus 2*k-1* clefs induisant une suite ordonnée d'au plus 2*k* intervalles, chacun d'eux étant associé à un sous-arbre. Cela ressemble donc à une généralisation des arbres binaires de recherche, cependant, par construction, les B-Arbres d'ordre *k* sont équilibrés, toutes les feuilles sont à la même hauteur, et sont denses puisque, mise à part la racine, tout nœud doit contenir au moins *k-1* clefs et tout nœud interne doit avoir un fils de plus que de clefs. Historiquement, les B-Arbres ont été inventés au début des années 1970 pour accéder rapidement à des informations stockées dans une mémoire secondaire à accès direct comme les disques, un nœud du B-Arbre correspondant alors à un secteur du disque. Les B-Arbres sont aujourd'hui au cœur de toutes les bases de données (les fameux *index*) et interviennent dans des systèmes de fichiers contemporains (ReiserFS, BtrFS). Leurs implémentations sont encore l'objet d'articles scientifiques du plus haut niveau.

Le but de ce projet est de proposer un outil d'indexation de fichiers reposant sur une implémentation générique des B-Arbres. Idéalement, on pourra créer un index des mots et de la position de leur définition dans un dictionnaire fourni sous forme de fichier texte. Pour trouver la définition d'un mot parmi *n* on découvrira sa position en $O(\log_k(n))$ accès au disque puis on accédera directement à la définition recherchée. Ce projet se décompose en 4 étapes :

- **B-Arbres d'entiers :** traduction en langage C d'algorithmes sur les B-Arbres et présentation modulaire.
- **B-Arbres génériques** : généralisation du travail précédent afin de pouvoir créer un B-Arbre travaillant sur un type de clef donné.
- **B-Arbres sur disque :** élimination des pointeurs dans les structures de données et stockage dans un fichier.
- **Indexation de fichiers :** réalisation d'un programme de démonstration.

À titre indicatif, les deux premières étapes pourraient être notées sur 16 points sur 20, la dernière étape serait facultative.

Construction d'un B-Arbre

On va construire un B-Arbre suivant la méthode du livre *Introduction à l'algorithmique* de Cormen *et al.* Cette construction repose sur un algorithme de recherche récursif de l'emplacement de la clef à insérer. Cet algorithme est analogue à celui d'insertion d'une valeur dans un arbre binaire de recherche modulo que l'on *éclate* les nœuds pleins rencontrés *avant* de les traverser. Ainsi tout nœud rencontré contenant *2k-1* clefs est divisé en deux donnant naissance à deux frères l'un contenant les *k-1* plus grandes clefs, l'autre les *k-1* plus petites et on fait remonter le *k-ième* élément (le médian) pour l'insérer dans leur père afin de séparer les deux frères. L'insertion du médian se fait alors sans difficulté puisque l'algorithme assure que le père du nœud éclaté n'est jamais plein. On poursuit ainsi la recherche de l'emplacement en s'enfonçant dans l'arbre jusqu'à rencontrer une feuille où l'on procède à une insertion classique d'une valeur dans un tableau ordonné.

Notons que l'insertion d'une clef se fait toujours au niveau d'une feuille et que seul l'éclatement de la racine provoque l'augmentation de la hauteur de l'arbre. Enfin remarquons que l'éclatement des nœuds pleins rencontrés est parfois inutile, cet algorithme n'est pas donc optimal.

Exemple de construction

Construisons un B-Arbre d'ordre 2 (au moins 1 clef et au plus 3) correspondant à la séquence 50, 20, 10, 80, 70, 90, 55, 25, 15, 85, 75, 95 :

Création de la racine
50
Insertion de 20 puis de 10
10
20
50

Insertion de 80 puis de 70 en 3 étapes :
- éclatement de la racine en trois nœuds

```
        10
20
        50
```

- insertion de 80 puis de 70

```
        10
20
        50
        70
        80
```

Insertion de 90 en deux étapes :
- éclatement du nœud (50 70 80) en deux nœuds et insertion du médian (70) dans le père

```
        10
20
        50
70
        80
```

- puis insertion de 90

```
        10
20
        50
70
        80
        90
```

Insertion de 55, 25, 15 et 85

```
        10
        15
20
        25
        50
        55
70
        80
        85
        90
```

Insertion de 75 par éclatement de (80 85 90)

```
        10
        15
20
        25
        50
        55
70
        75
        80
85
        90
```

Insertion de 95 par éclatement préventif (bien qu'inutile) de la racine en trois nœuds :

```
                10
                15
        20
                25
                50
                55
70
                75
                80
        85
                90
                95
```

Présentation succincte des algorithmes

Par choix pédagogique, nous avons décidé de conserver presque tels quels les structures de données et identifiants du livre de *Cormen et al. ;* ce livre est disponible en grand nombre à votre bibliothèque universitaire.

Structures de données.

Chaque nœud x contient les champs suivants :

x.n : nombre de clefs conservées

x.clé[] : tableau de x.n clefs rangées dans l'ordre croissant

x.feuille : valeur booléenne indiquant si le nœud est une feuille ou non.

x.c[] : tableau de x.n + 1 pointeurs vers les enfants du nœud lorsque celui-ci n'est pas une feuille

Pour tout nœud interne x, les clefs x.clé[i] déterminent les intervalles des nœuds stockés dans chaque sous-arbre, pour toute clef C_i d'un sous-arbre de racine x.c[i] on a :

$$C_1 \leq x.clé[1] \leq C_2 \leq \ldots \leq x.clé[x.n] \leq C_{x.n+1}$$

La racine T est composée du champ suivant :

T.racine : pointeur vers le nœud racine

Algorithmes.

```
Rechercher-B-Arbre (x, e)
i ← 1
tant que i ≤ x.n et e > x.clé[i]
        faire i ← i+1
si i ≤ x.n et e = x.clé[i]
        alors retourner (x, i)
si x.feuille
        alors retourner NIL
        sinon
            Lire-Disque(x.c[i])
            retourner Rechercher-B-Arbre(x.c[i],e)
```

```
Créer-B-Arbre(T)
x ← Allouer-Nœud()
x.feuille ← vrai
x.n ← 0
Ecrire-Disque(x)
T.racine ← x
```

```
Partager-Enfant-B-Arbre(x,i,y)
z← Allouer-Nœud()
z.feuille ← y.feuille
z.n ← k – 1
pour j ← 1 à k-1
        faire z.clé[j] ← y.clé[j+k]
si non y.feuille
        alors pour j ← 1 à k
                faire z.c[j] ← y.c[j+k]
y.n ← k-1
pour j ← x.n+1 jusqu'à i+1
        faire x.c[j+1] ← x.c[j]
x.c[i+1]← z
pour j ← x.n jusqu'à i
        faire x.clé[j+1] ← x.clé[j]
x.clé[i] ← y.clé[k]
x.n ← x.n+1
Ecrire-Disque(y)
Ecrire-Disque(z)
Ecrire-Disque(x)
```

```
Insérer-B-Arbre(T,e)
r ← T.racine
si r.n = 2k-1
 alors
        s ← Allouer-Nœud()
        T.racine ← s
        s.feuille ← faux
        s.n ← 0
        s.c[1] ← r
        Partager-Enfant-B-Arbre(s,1,r)
        Insérer-B-Arbre-Incomplet(s,e)
 Sinon
        Insérer-B-Arbre-Incomplet(r,e)
```

```
Insérer-B-Arbre-Incomplet(x,e)
i ← x.n
si x.feuille alors
        tant que i ≥ 1 et e < x.clé[i] faire
                x.clé[i+1] ← x.clé[i]
                i ← i - 1
        x.clé[i+1] ← e
        x.n ← x.n + 1
        Écrire-Disque(x)
sinon
        tant que i ≥ 1 et e < x.clé[i] faire
                i ← i – 1
        i ← i + 1
        Lire-Disque(x.c[i])
        si x.c[i].n = 2k - 1alors
          Partager-Enfant-B-Arbre(x,i,x.c[i])
          si e > x.clé[i] alors
                i ← i + 1
        Insérer-B-Arbre-Incomplet(x.c[i],e)
```

B-Arbres d'entiers

Il s'agit de traduire en langage C les algorithmes présentés ci-dessus et très bien commentés dans le livre de Cormen *et al.* **Il ne s'agit pas de traduire mot à mot les algorithmes mais bien d'en proposer une traduction lisible et efficace respectant à la fois la démarche algorithmique des auteurs et le *bon usage* du langage en C.** Par exemple, on conservera l'utilisation de tableaux qui peuvent être écrits / lus directement dans / depuis un fichier, par contre, on s'efforcera d'utiliser autant que possible les fonctions memcpy() et memmove() pour copier et déplacer des zones de données.

Il s'agira de produire un module b_arbre_entier dont l'interface ne révélera pas à l'utilisateur les structures de données utilisées (respect du principe du masquage de l'implémentation). Outre la création et la destruction d'un B-Arbre et l'insertion / la recherche d'un élément dans celui-ci, ce module proposera une fonctionnalité d'*itération* qui permettra d'appliquer une fonction passée en paramètre à chacune des clefs de l'arbre et ce dans l'ordre croissant. On peut imaginer que l'appel itérer(un_arbre, afficher_un_entier) affiche dans l'ordre croissant l'ensemble des clefs conservées. À des fin de débogage, on pourra écrire une fonction qui affichera la structure d'un B-Arbre.

Un programme de tests accompagnera le module. Celui-ci, paramétré par l'ordre du B-Arbre, lira un ensemble d'entiers sur son entrée standard et, une fois la fin de fichier rencontrée affichera le B-Arbre construit.

B-Arbres génériques

Il s'agit de produire une version générique des B-Arbres : on désire pouvoir construire des B-Arbres pout tout type de données contraint (B-Arbre de doubles, B-Arbre de tableaux de 10 caractères). Pour cela vous pouvez choisir entre deux approches possibles : l'approche dynamique où le code fait abstraction du type de données à conserver et l'approche statique où l'on génère automatiquement le code dédié à un type donné. Les deux approches sont complémentaires : si l'approche dynamique est plus souple d'utilisation, l'approche statique offre des facilités pour le débogage et l'optimisation.

Approche dynamique.

Il s'agit de s'inspirer de la technique de programmation de la fonction de tri générique qsort(). Pour travailler cette fonction a besoin du nombre d'éléments contenus dans le tableau à trier, de la taille unitaire des éléments et d'un pointeur de fonction. Pour créer un B-Arbre générique, il faut préciser de façon analogue l'ordre du B-Arbre, la taille des éléments à stocker et la fonction de comparaison à utiliser pour les ordonner.

Pour comparer une clef donnée en paramètre à un élément du tableau, copier ou déplacer un ensemble de clefs, il peut être nécessaire de calculer l'adresse de la ième clef, cela peut se coder ainsi :

```
(void *) (taille_unitaire * i + (char *) première_clef)
```

On aura intérêt à définir une fonction ou macro fils(arbre, nœud, position) qui retournera l'adresse du fils en question. On aura aussi intérêt à modifier le code dédié aux entiers afin de minimiser les différences entre les deux codes : il est bien plus aisé de mettre au point un code dédié qu'un code générique.

Approche statique.
L'objectif est de générer le code spécifique à chaque type à partir d'un modèle générique (gabarit ou *template*, en anglais) à l'aide d'un préprocesseur. Cette technique de programmation fut utilisée aux origines du langage C++ et est reprise en Java. L'intérêt de cette technique est double : on dispose d'un code optimisé pour chaque type et le code source (celui écrit par le programmeur) n'est pas dupliqué car il écrit un méta-code à partir duquel sont générés les codes spécifiques.
On trouvera sur le site d'Achille Braquelaire http://dept-info.labri.fr/~achille/enseignement/INF-153/src/Chapitre-09/ un code source mettant en œuvre cette technique (on pourra comparer le méta-code au code présenté au Chapitre-06).
Dans notre cas, il s'agit de paramétrer le module par le type de données à stocker, la fonction de comparaison et, pourquoi pas, en profiter pour fixer l'ordre du B-Arbre ce qui autorise quelques simplifications et optimisations supplémentaires.

On proposera un programme de test mettant en valeur le travail réalisé.

B-Arbres sur disque
Il s'agit d'ajouter les fonctionnalités de chargement et de sauvegarde d'un B-Arbre dans un fichier donné en paramètre. Pour d'évidentes raisons de portabilité, il est illusoire de stocker directement une structure de données contenant des pointeurs dans un fichier : cela ne marche pas. Aussi, il s'avère nécessaire de faire disparaître tout pointeur des données à stocker dans un fichier. Dans cette optique, on utilise un *buffer,* tableau alloué (voire réalloué) dynamiquement, pour y puiser la mémoire nécessaire aux nœuds. On peut alors utiliser les indices de ce tableau pour référencer les nœuds et il suffira de sauvegarder ce tableau et l'indice de la racine pour sauvegarder le B-Arbre dans son intégralité. Pour le B-Arbre précédent on obtient la distribution suivante :

```
                10 (1)
                15 (1)
        20 (2)
                25 (3)
                50 (3)
                55 (3)
70 (6)
                75 (4)
                80 (4)
        85 (7)
                90 (5)
                95 (5)
```

On retrouve directement l'ordre d'allocation des différents nœuds et l'on remarque que la racine est ici située sur la 6ième composante du tableau.
L'adaptation du méta-code de la méthode statique est en fait assez aisée pourvu que l'on fixe l'ordre du B-Arbre : en gros, il s'agit de modifier la procédure d'allocation mémoire des nœuds, de remplacer tout pointeur x par un indice i correspondant et de substituer x→champ par tab[i].champ ou encore *x par tab[i].

Les modifications à apporter au code de la méthode dynamique sont plus délicates : il s'agit de calculer la taille de la structure et l'emplacement de chaque objet dans celle-ci. Il est vivement conseillé d'utiliser des fonctions ou macros pour accéder à tout élément de la structure et de bien vérifier leur correction. Une fois ce travail réalisé, la transformation du code est aisée mais tout bug est difficile à résoudre : par principe il faut s'écarter le moins possible du code générique et du code spécifique aux entiers.

Enfin, à l'aide de l'appel système mmap(), on *projettera* le fichier d'index en mémoire. C'est une méthode très efficace pour lire et écrire dans un fichier aussi simplement que l'on manipule un tableau. À cet effet, un code vous sera donné pour illustrer le procédé de projection du fichier d'index comme tableau de nœuds. Techniquement, on réservera le premier nœud du fichier au stockage de l'emplacement de la racine, de l'ordre du B-Arbre et de la taille d'une clef. D'autre part, pour des raisons de performance on aura intérêt à faire en sorte que la taille d'un nœud soit une puissance de 2.

Indexation de fichiers
On pourra illustrer le concept en indexant, par exemple, des dictionnaires disponibles sur :
http://xdxf.revdanica.com/down/

On aura intérêt à définir une fonction de hachage pour accélérer les comparaisons mais aussi pour disperser les clefs : créer un B-Arbre à partir d'une suite ordonnées de clefs n'est pas optimal.