

Notes rapide provenant du livre

Charles Jacquet 2781-12-00

January 21, 2015

1 Définition de base

- **Abstraction** : Décomposer un problème jusque dans ses parties les plus fondamentales.
- **Abstract Data Type** : C'est un modèle mathématique de structures de données qui spécifie le type de donnée , les opérations supportées ainsi que les arguments nécessaire pour celles-ci. C'est une sorte de contrat où on dit ce que la structure doit faire mais on ne dit pas comment elle doit le faire. Pour rendre le ADT plus indépendant de son interface, il est préférable de l'implémenter avec une interface plutôt qu'avec une classe.
- **Collection** : Une collection d'objet est un ensemble d'objets stockés avec grâce à une structure particulière.
- **Arbre ordonné** : Un arbre est dit ordonné, si il y a une signification linéaire à l'ordre des enfants. C'est-à-dire que l'ordre à de l'importance dans le parcourt de l'arbre.
- **Arbre binaire impropre** : Un arbre binaire propre est un arbre dont tous les noeuds internes ont soit aucun ou soit deux enfants chacun. Ceci étant dit, un arbre impropre est un arbre qui possède au moins un noeud n'ayant pas exactement deux fils.
- **Hauteur d'un arbre** : c'est le nombre maximum de génération en dessous du noeud. Le niveau n, est l'ensemble des noeuds de hauteur n. La feuille (leaf) la plus basse est de hauteur 0.
- **Profondeur d'un arbre** : C'est le nombre de parents que possède un noeud. La racine est de profondeur 0.
- **Arbre complet** : Un arbre complet est un arbre dont tous les noeuds internes ont exactement deux enfants. Un arbre complet implique que tous les enfants d'un même parent ont la même hauteur. Dès lors, la différence des hauteurs d'enfants ayant le même parent sera toujours égale à 0, donc un arbre complet est toujours équilibré.
- **Arbre essentiellement complet** : C'est lorsque tous les noeuds sauf les noeuds du dernier niveau sont complet, c'est à dire qu'ils possèdent le maximum de noeuds. Et ensuite, tous les noeuds du dernier niveau sont au maximum à gauche. Un arbre essentiellement complet est également toujours équilibré.
- **Graphe directed** : Chaque edge à un sens.
- **Internal node** : Nœud avec au minimum un enfant.
- **External node** : Nœud sans enfant.
- **Position équilibrée** : Une position dans un arbre est dite balancée si la valeur absolue de la différence entre la hauteur de ses enfants est au maximum de 1.
- **Arbre équilibré** : Un arbre équilibré est un arbre pour lequel la valeur absolue de la différence entre la hauteur du sous-arbre de gauche et de celle du sous-arbre de droite est au maximum de 1.
- **Patron de conception** : est un arrangement caractéristique de modules, reconnu comme bonne pratique en réponse à un problème de conception d'un logiciel. Il décrit une solution standard, utilisable dans la conception de différents logiciels.(Wikipédia)
- **Weighted Graphs** : Dans certains graphes, un poids est associé à chaque arrêtes.

2 Les itérateurs

2.1 snapshot iterator

L'itérateur maintient sa propre copie la collection

2.2 lazy iterator

Ne fait que passer d'un éléments à l'autre avec l'opération next

3 Array and linked List

Il existe plusieurs sortes de linked lists:

3.1 Simple array

Attention, si on doit trouver des éléments dans un tableau à 2 dimensions avec un algorithme en $O(n \log(n))$ il suffit de faire un binary search sur les n niveaux. soit $N * \log(n)$.

3.2 Simplement liée

Contient juste un lien vers l'élément suivant. Est utile pour l'implémentation de stack (pile) ou de queue (queue).

3.3 Doublement liée

3.4 Circulaire

4 Stack

La stack est une collection d'objets suivant la règle du LIFO: last-in first-out

- push(e)
- pop()
- – (composantes additionnelles)
- top()
- size()
- isEmpty()

5 Queue

La queue est une collection d'objets suivant la règle du FIFO first-in first-out

- enqueue(e)
- dequeue()
- – (composantes additionnelles)
- first()
- size()
- isEmpty()

6 Tree

C'est un type de données abstraites sous forme d'éléments hiérarchiques. parent ('parent') -> enfant ('children')

Le premier élément d'un arbre est la racine ('root')

6.1 Binary tree

Chaque nœud à au maximum deux enfants

Il y a le 'left child' et le 'right child'

L'enfant de gauche précède l'enfant de droite dans l'ordre de l'arbre

6.1.1 représentation en mémoire

Pour représenter un arbre, on utilise les listes chainées dans le cas où le contenu n'est pas connu à l'avance avec ces liens :

- vers le parent
- vers le left child
- vers le right child

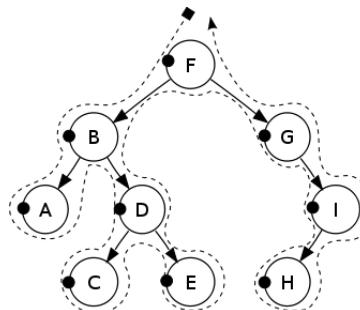
Par contre, lorsqu'on connaît l'arbre et que l'on cherche juste à le stocker, il est parfois plus efficace en terme de mémoire de le stocker dans une array list. (si l'arbre est équilibré)

Attention, Dans le cas d'un arbre complet de taille fixe, il est avantageux de le stocker sous forme d'un tableau.

6.2 Les parcours / traversal algo

6.2.1 Preorder traversal (préfixe)

On affiche le root ensuite on affiche toujours le plus en bas à gauche . Et on affiche donc les éléments en descendant de la racine.



6.2.2 Postorder traversal (postfixe)

On affiche le root en dernier.

6.2.3 inorder traversal

Pour afficher dans cet ordre, il suffit de faire une récursion, on affiche le nœud ensuite inorder(left) + inorder (right).

6.2.4 Euler tour

on affiche tous les enfants avant les parents :)

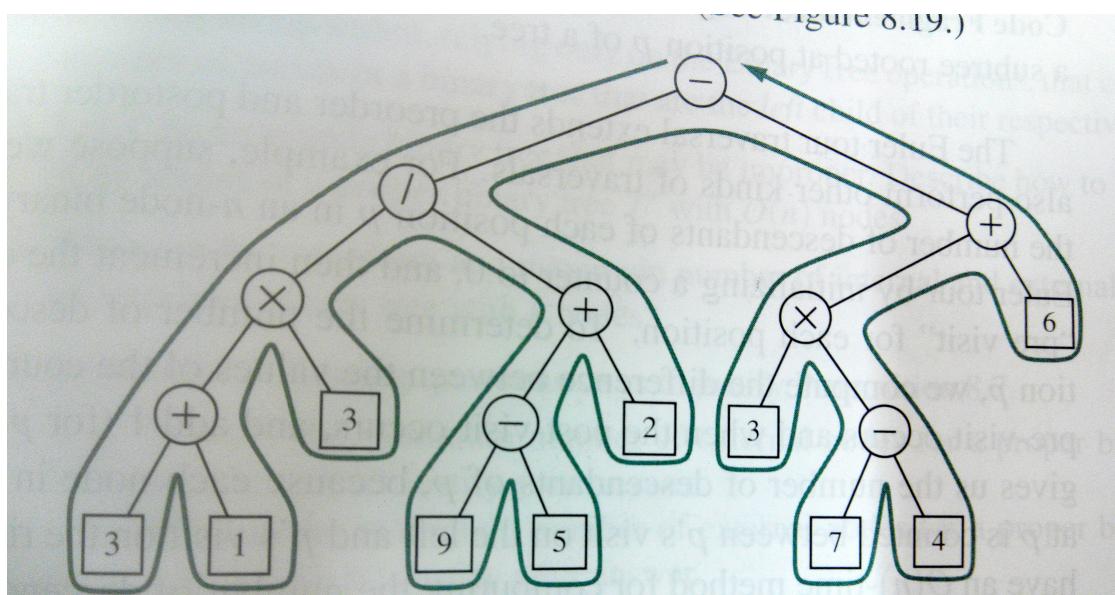
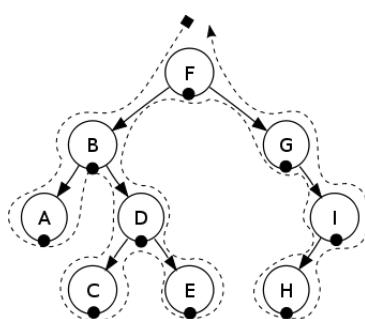
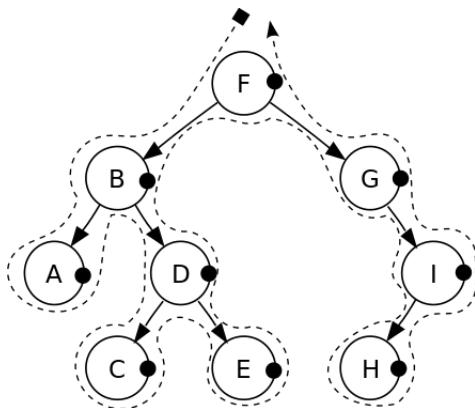


Figure 8.19: Euler tour traversal of a tree.

complexity of the walk is $O(n)$, for a tree with n nodes, because exactly two times along each of the $n - 1$ edges of the tree—one forward along the edge. To n

6.2.5 breadth-first Tree

Contrairement aux algorithme de traversée en profondeur, celui-ci utilise une liste FIFO, soit une file.

1. Prends tous les noeuds d'un niveau dans une liste fifo

2. Un par un, fait la mm chose en listant les noeuds du niveau suivant
3. et ainsi de suite

Pour mieux comprendre, on peut utiliser le pseudo code suivant:

```
ParcoursLargeur(Sommet s):
{
    f = CreerFile();
    f.enfiler(s);
    marquer(s);
    TANT-QUE NON f.vide() FAIRE
        s = f.defiler();
        afficher(s);
        POUR TOUT voisin de s FAIRE
            SI voisin non marqué FAIRE
                f.enfiler(voisin);
                marquer(voisin);
            FIN SI
        FIN POUR TOUT
    FIN TANT QUE
}
```

7 Heaps

Un heap est un binary tree qui stocke des entrées directement à leur bonne positions et qui possède 2 propriétés:

- Dans un heap T, pour chaque positions p autre que la racine, la clé stockée en p est plus grande que la clé des parents de p.
- Un heap avec une hauteur h est un arbre binaire complet si tous les niveaux $1 \rightarrow (h-1)$ ont le maximum de noeuds possible. Et les noeuds du niveau h, sont au maximum à gauche

8 Priority queue

Permet de stocker des objets qui doivent être utilisé dans un ordre bien précis. C'est à dire qu'elle va donner un avantage à certains objets par rapport à d'autres en fonction d'un critère.
On ajoute l'élément avec une clé.

- insert(k, v)
- min()
- removeMin()
- size()
- isEmpty()

8.1 Implémentation

On peut l'implémenter avec une liste non triée .. plus facile pour ajouter des éléments mais trouver le min, et le remove min sont plus lourd soit en $O(n)$.

Pour une implémentation avec une liste triée, on est en $O(n)$ pour ajouter des éléments mais on est en $O(1)$ pour le reste.

On peut également l'implémenter avec un heap. Ceci permet d'avoir une complexité d'ajout et de suppression en $O(\log n)$

9 Map

Map est un type de donnée abstraite qui permet de stocker efficacement des valeurs grâce à une clé. On stocke donc une paire de données, (k,v) avec la clé k et l'objet v.

- size()
- isEmpty()
- get(k) : retourne l'objet stocké avec la clé k
- put(k,v) : rajoute une entrée avec la clé K si elle n'existe pas, sinon, la remplace! (attention ça peut faire des dégâts)
- remove(k)
- keySet() retourne une collection 'itérable' des clés stockées
- values() retourne une collection 'itérable' des entrées stockées
- entrySet() retourne une collection 'itérable' des clés-entrées stockées

9.1 Sorted Map

- firstEntry()
- lastEntry()
- ceilingEntry(k) renvoie la valeur la plus proche de k, et plus grande ou égale à k.
- floorEntry(k) renvoie la valeur la plus proche de k, et plus petite ou égale à k.
- lowerEntry(k) retourne la valeur juste en dessous de k (<)
- higherEntry(k) retourne la valeur juste au dessus de k (>)
- subMap(k1, k2)

10 Hashing

Le but est d'utiliser une fonction de hash pour créer la clé de l'élément dans un tableau.

On considère le tableau utilisé comme un bucket array, c'est à dire que si on ne gère pas les collisions, on peut avoir plusieurs éléments dans une 'case'. Pour mettre un éléments, on a plusieurs étapes

- fonction de hash \rightarrow int
- compression \rightarrow $0 \leq \text{int} < N$ avec N la taille du tableau
- gestion des collisions

Dans une table de hashing, on défini le facteur de charge (load factor) $\lambda = n/N$ avec n le nombre d'éléments dans la bucket array tandis que N sa taille.

10.1 Collisions

10.1.1 Sparate chaining

Le but est de l'utiliser comme une seconde collections dans chaque éléments de la bucket array. Par exemple, un autre tableau, ou une liste chainée.

10.1.2 Linear probing

Si la place est déjà prise on regarde la suivante jusqu'à ce qu'on en trouve une de libre.

10.1.3 Quadratic probing

Si la place $A[h(k)]$ est prise, on regarde en $A[h(k) + i^2]$ avec $i = 1, 2 \dots$

Attention, ça cause des problèmes dès que le tableau est rempli à moitié, on est pas sur de trouver des places :(

10.1.4 Double hashing

A pour intérêt de ne pas causer de clustering mais est bien plus compliqué et implique la dérivée de la table de hash

10.1.5 Attention suppression d'un élément

Attention, lors de la suppression d'un élément qui utilise par exemple le linear probing, on doit transférer tous les éléments suivants dans une autre collection jusqu'à avoir une élément vide. Ensuite il faut les ajouter un à un dans la collection de départ.

11 Dictionnaire

Le dictionnaire est une Map avec des strings comme identifiant. (pas sur)

12 Skip List

La skip list est une amélioration de liste chainée triée.

En fait c'est un ensemble de niveau de liste. chaque élément a une certaine probabilité de se trouver au niveau supérieur.

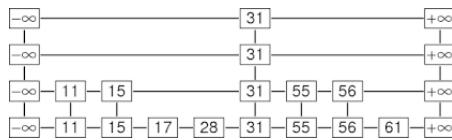


Figure 1: Exemple d'une skip list

Pour ce qui est de l'algorithme de recherche, au début, on commence en "haut à gauche", et lorsqu'on atteint le dernier élément, on descend de niveau par rapport à l'élément juste en dessous de celui recherché. On continue jusqu'à le trouver et si ce n'est pas le cas, on renvoie null.

Pour l'ajout d'un élément, on ajoute au bon endroit, on "randomize" pour trouver la taille de la tour et ensuite on la lie aux différents éléments.

Pour l'ajout d'élément, c'est la même chose ... il faut faire attention à bien refaire les liens.

Une Skip List est une implémentation possible pour un dictionnaire non ordonné mais il n'est pas judicieux de l'utiliser. Car le système de rangement des entrées est basé sur de l'aléatoire. Donc la recherche se fait en $O(\log n)$, l'insertion en $O(\log n)$ et la suppression en $O(\log n)$.

13 Binary search trees

- get(k)
- put(k, v)
- remove(k)

14 Balance Search Trees

Le but est d'utiliser des algorithme pour l'ajout et la suppression de manière à garder un arbre le plus équilibré possible.

14.1 Rotation

La rotation à pour but de changer de place entre la formation de gauche et la formation de droite :

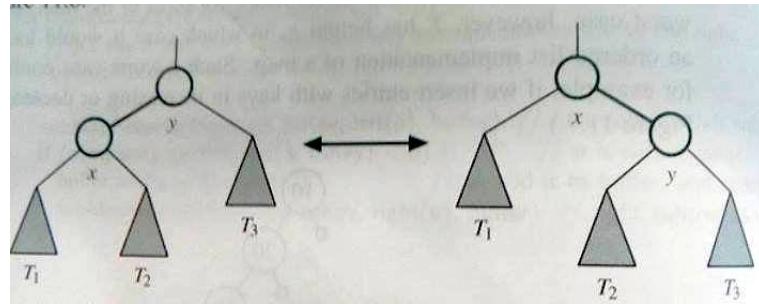


Figure 2: exemple de rotaion

Bien sur ces changements ce font lorsqu'ils sont nécessaire pour re-balancer un arbre. Pour ça en fait on définit 3 fonctions, qu'on verra plus spécifiquement dans les différente type d'arbres balancé

- rebalanceInsert
- rebalanceDelete
- rebalanceAccess

14.2 AVL trees

On ajoute au binary search tree, l'obligation d'avoir une hauteur en $\log(n)$.

14.2.1 Insertion

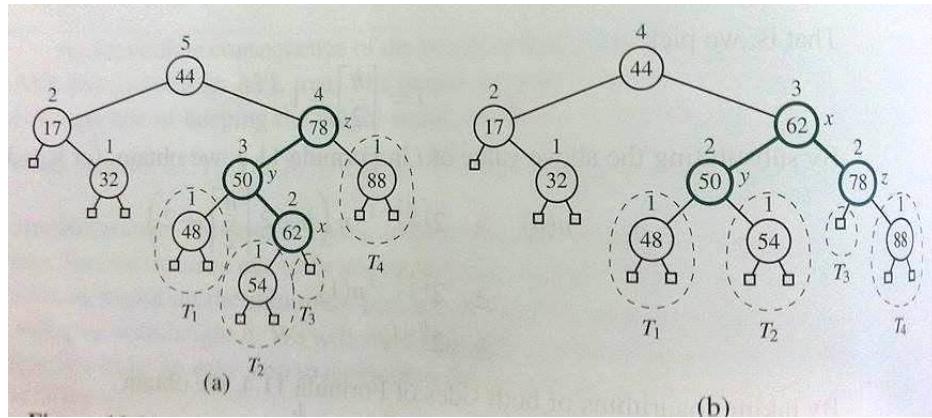


Figure 3: insertion avl ex 1

Lorsqu'on insert un élément, l'arbre risque de ne plus être balancé. On utilise une stratégie de recherche et réparation.

14.2.2 Suppression

C'est semblable pour la suppression.

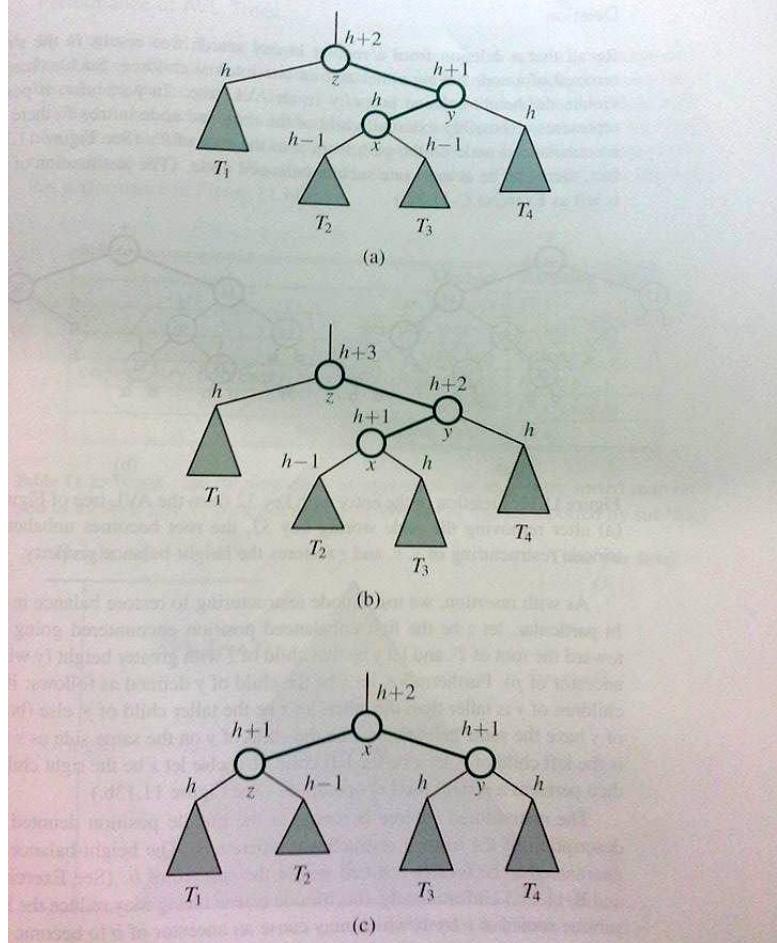


Figure 4: insertion avl ex 2

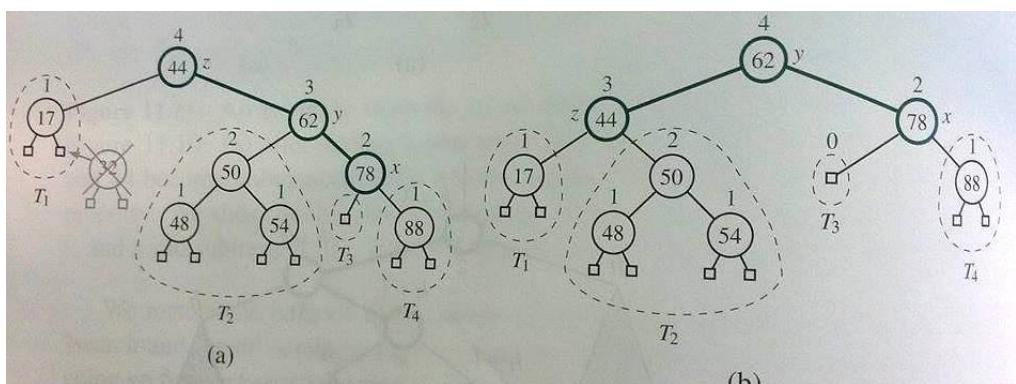


Figure 5: suppression d'un élément avl

14.3 Multiway search tree

Prenons W , un noeud d'un arbre ordonné. On dit de w qu'il est un d -node s'il possède d enfants.

- Chaque noeud interne de T , doit au minimum posséder 2 enfants.
- Si on a un noeud w de T qui possède d enfants, tel que c_i est un enfant du noeud avec $i = 1, \dots, d$

. Alors w doit posséder un set de $(n-1)$ paire de clé-valeur (k, v) .

- On définit deux éléments, $k_0 = -\infty$ $k_d = \infty$ (je n'ai pas la moindre idée de leur utilité :'().
- Pour comprendre le classement dans ce type d'arbre, il faut prendre un noeud w. Ce noeud possède 'd' enfants appelés c_i avec $i = 1, \dots, d$. On définit par k_i les clés contenues dans le noeud w et k toutes clé contenue dans c_i . Le classement est tel que $k_{i-1} \leq k \leq k_i$.

On peut implémenter cet arbre comme une liste simplement chainée et chaque noeud comment une SortedTableMap.

14.4 (2,4)-tree

- chaque noeud interne à au maximum 4 enfants
- Tous les noeuds externes ont la même hauteur

Les noeuds ont donc 2, 3 ou 4 enfants et à pour intérêt d'utiliser moins d'espace mémoire.

14.5 B-tree

Un B-tree d'ordre d est un arbre (a, b) tel que: $a = d/2$ et $b = d$.
Soit un arbre $(\frac{d}{2}, d)$.

14.6 Splay tree

On ajoute aucune règle spécifique à la taille/hauteur de l'arbre mais on va juste modifier la façon d'ajouter, de supprimer ou de rechercher des éléments de manière à garder les éléments les plus recherché au plus près possible du root (splaying) et donc obtenir un algorithme de recherche plus efficace.

On peut par exemple faire une promotion avec un zig-zig, zig-zag, zig:

*zig-zig: The node x and its parent y are both left children or both right children.
(See Figure 11.33.) We promote x, making y a child of x and z a child of y,
while maintaining the inorder relationships of the nodes in T.*

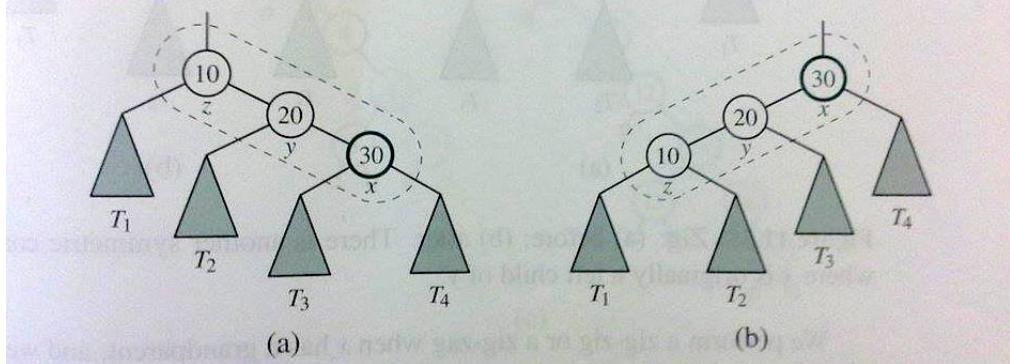


Figure 6: Promotion de x avec un zig-zig

When to splay ??? Si on fait une recherche de k, qu'on le trouve alors on le splay pour qu'il devienne le root. Sinon, on splay le parent de la feuille (si on le trouve pas .. on arrive à une leaf). Lors d'une insertion, de sorte que le numéro inséré devienne le root \rightarrow on l'ajoute et puis on splay. Quand on supprime un élément, on splay le parent de sorte à ce qu'il devienne le root

15 Pattern Matching algorithms on pattern

15.1 Brute force

On test juste pour chaque lettre, cet algorithme est loin d'être efficace

zig-zag: One of x and y is a left child and the other is a right child. (See Figure 11.34.) In this case, we promote x by making x have y and z as its children, while maintaining the inorder relationships of the nodes in T .

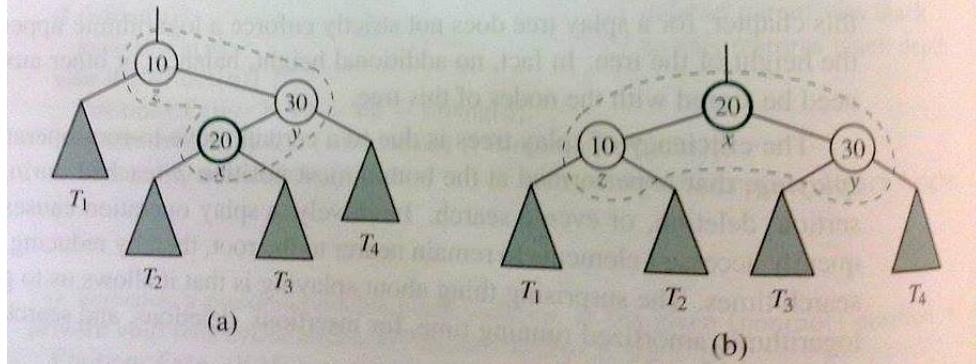


Figure 7: Promotion de x avec un zig-zag

zig: x does not have a grandparent. (See Figure 11.35.) In this case, we perform a single rotation to promote x over y , making y a child of x , while maintaining the relative inorder relationships of the nodes in T .

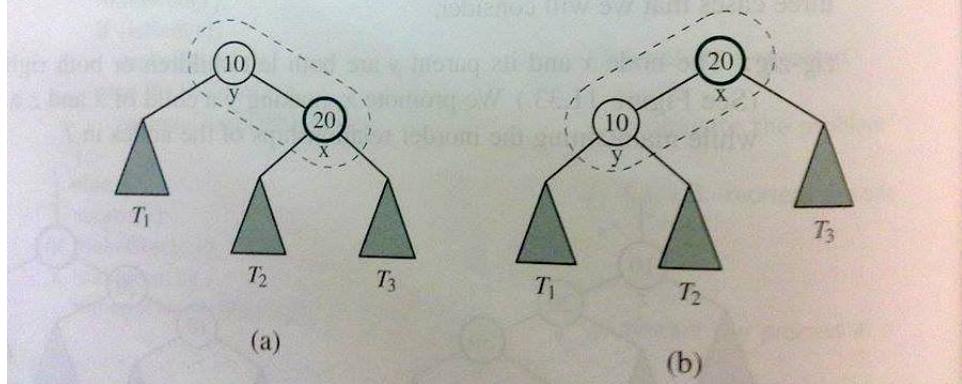


Figure 8: Promotion de x avec un zig

15.2 The Boyer-Moore Algo

<https://www.youtube.com/watch?v=PHXAOKQk2dw> Permet d'éviter une fraction significante du texte:
Si on a une lettre c qui n'apparaît pas du tout dans le pattern, alors on skip le pattern juste après.
Si on a un caractère c , qui apparaît dans le pattern, on skip jusqu'à ce caractère et ensuite on continue.
Pour l'implémentation, c'est un peu différent, On doit créer le tableau de boyer-Moore:
Pour ce faire, en partant de la gauche, pour chaque caractère on calcul $\max(1, \text{length}-1-\text{index})$. Si il y a plusieurs valeurs, on oublie la plus vieille valeur et on la remplace par la nouvelle.
Lors de la comparaison, on commence par la droite et on va vers la gauche, lorsqu'il y a un mismatch, on décale de la valeur trouvée vers l'avant.

15.3 The Knuth-Morris-Pratt Algo

On se rend compte que dans les algorithmes précédents on perd de l'information et on doit la recalculer à chaque fois.

Il faut calculer une fonction d'échec : "failure function" c'est à dire un tableau dans lequel on compte les lettres du début :

k	0	1	2	3	4	5	6	7	8	9	10	11
$P[k]$	a	m	a	l	g	a	m	a	t	i	o	n
$f(k)$	0	0	1	0	0	1	2	3	0	0	0	0

Ensuite on test. Quand on a un mismatch, on regarde la valeur, donc on recommence à cette valeur.

16 Pattern Matching algorithms directly on the text

16.1 Tries

Le premier élément n'est pas un caractère. Ensuite, chaque noeud est un caractère de l'alphabet, tel

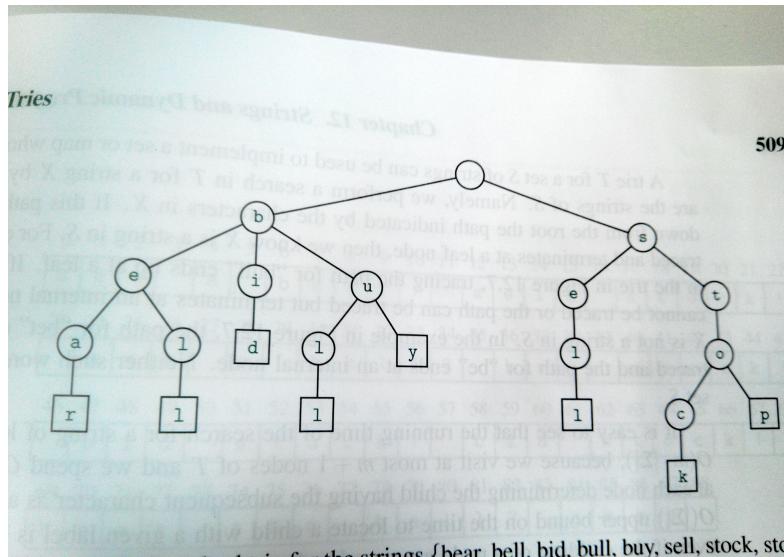


Figure 12.7: Standard trie for the strings {bear, bell, bid, bull, buy, sell, stock, etc}

que chaque leaf représente un mot. Soit pour retrouver ce mots, il suffit de concaténer tous les strings précédents.

16.2 Compressed trie

<https://www.youtube.com/watch?v=ZdooBTdW5bM> Le but est que ça prenne moins de place.
Parfois c'est redondant, donc si il n'y a pas plusieurs choix de lettre, autant en mettre plusieurs ensemble.
Ensuite on peut également stocker une combinaison de 3 index (i,j,k) à la place des string. Ces index sont référencés par rapport à i strings et on en prends les lettres contenues entre l'index j et k. Cette technique est fort efficace pour réduire l'espace mémoire utilisé.

16.3 Suffix Tries

C'est un tries mais uniquement avec les suffix d'un plus grand mot.

17 Greedy method

C'est une méthode d'optimisation, qui a pour but de faire des choix plus petits et en espérant arriver à une optimisation maximale ..

18 Huffman

<https://www.youtube.com/watch?v=ZdooBTdW5bM>

Algo utilisé pour compresser du texte.

1. Premièrement, il faut trouver la fréquence de chaque lettre.
2. On trie le tableau avec comme clé la fréquence et la lettre en valeur.
3. On utilise une priority queue, on crée un binary tree avec les deux plus petits éléments en les retirant de la priority queue.
4. On rajoute ce tree avec comme clé la somme des deux clés.

19 graph

Dans un graphe :

- vertex (vertices au pluriel)
- edges

19.1 Comment stocker le graphe

- **Edge list** : On crée bêtement une list de nœuds non ordonnée, et une liste non ordonnée d'arêtes.
- **Adjacency list** : On crée deux collections, une collection primaire pour stocker les nœuds, et chaque nœud v possède une collection secondaire, $I(v)$ contenant toutes les arrêtes incidentes à ce nœud.
- **Adjacency map** : Stocker pour chaque nœud, une map (vertex, edge). Avec tous les nœuds adjacent à celui-ci.
- **adjacency matrix** : On crée un tableau à 2 dimensions, on détermine un index par nœud. et on mets dans le tableau l'arrête qui lie les 2 noeuds.

19.2 Graph traversal

Bon dans le livre, ils parlent de mythologie grec ... je ne pense pas que ça soit important :) Oh que si jeune padawan!

19.2.1 Depth-First Search (DFS)

Pour ce faire, on se muni d'une corde, (non on ne vas pas se pendre pour un cours pareil ;)). On choisit un nœud au hasard. On le colorie pour dire qu'on est passé, ensuite, on prend un des chemins possible (edge) en déroulant la corde, on arrive à une autre nœud, qu'on colorie également.

Si l'arrête mène à un nœud déjà visité, on y va pas. Dans le cas, où il n'y a plus d'arêtes non visitées pour un nœud, alors, on revient sur son chemin en rebobinant la corde. On arrive au nœud précédent, on regarde s'il y a des nœuds non visités. Ainsi de suite jusqu'à revenir au nœud de départ.

Cet algorithme permet de :

1. Voir si un graphe est connexe.
2. Trouver tous les composants connexe d'un graphe.
3. Voir si un graphe possède des cycles ou pas.

19.2.2 Breadth-First Search (BFS)

C'est le même algorithme que le DFS sauf qu'on considère qu'on a plusieurs explorateurs.

19.3 Transitive Closure

Le but est de créer un graphe G^* en partant de G . Uniquement en ajoutant des arcs (edge) entre le noeud de départ et tout les noeuds qu'il peut joindre de façon indirecte.

19.4 Directed Acyclic Graphs & topological ordering

Premièrement dans beaucoup d'application, il peut être intéressant d'avoir un graphe dirigé ne possédant pas de cycle.

De plus, on a une topologie ordonnée lorsque que pour chaque edge (v_i, v_j) on a que $i < j$.

19.5 Shortest Paths

Pour ce faire, on travail usuellement avec des "weighted Graphs".

19.5.1 Dijkstra

<https://www.youtube.com/watch?v=gdmf0wyQlcI> Cette méthode montre la puissance du "greedy method pattern".

On définit V comme étant le set des Vertices du graphe.

1. On définit $s \in V$ comme étant le noeud de départ.
2. Pour $v \in V$, on définit $D[v]$ comme étant la distance approximative entre s et V . Soit $D[s] = 0$ et $D[u \in V \text{ et différent de } s] = \infty$.
3. On définit C comme étant le "Cloud" qui initialement ne contient que s .
4. On choisit u tq $u \in V$ et $u \notin C$ et que $D[u]$ soit le plus petit possible. On ajoute u à C .
Attention, u ne doit pas obligatoirement être adjacent au dernier noeud utilisé, mais doit avoir la plus petite valeur de D et ne pas encore avoir été visité.
5. On met à jour $D[]$ car il y a peut-être un meilleur chemin. Ce principe d'amélioration de l'approximation à chaque itération s'appelle la "procédure de relaxation".

```
if D[u] + w[u,v] < D[u] then  
    D[u] = D[u] + w(u,v)
```

Pour ce faire, on a le choix entre 2 implémentations, avec une priority queue ou avec un heap.

S'il y a peu de noeuds → on utilise un heap

s'il y a beaucoup de noeuds → on utilise une priority queue.

Par contre, cette technique calcule la distance mais pas le trajet ... pour pouvoir se souvenir du trajet, on utilise un "shortest-path tree".

19.6 Minimum spanning tree

Par exemple, si nous voulons connecter tous les ordinateurs d'un immeuble tout en utilisant le moins de câble possible.

Pour définir cela, on dit juste que la somme des poids des différentes arrêtes doit être minimale. c'est le MST problem.

19.6.1 Prim-Jarnik's algorithm

<https://www.youtube.com/watch?v=bxmnnOFxKXY0>

Cet algorithme est aussi basé sur la greedy method.

1. On choisit un noeud.
2. On regarde tous les chemins possible de prendre.
3. On prend celui avec le poids minimum.

4. On regarde tous les chemins non utilisé et adjacent à au moins un nœud déjà visité.
5. On prend celui avec le poids minimum.
6. On réitère l'opération jusqu'à ce que tous les nœuds aient été visités.

19.6.2 Kruskal's algorithm

<https://www.youtube.com/watch?v=71UQH7Pr9kU>

Pour ce faire on ne part pas d'un noeud mais on utilise une priority queue sur les vertex.

A chaque itération, on prends l'arrête avec le poids minimum. Cette arrête ne peut pas créer de cycle. soit si e crée un lien $\in S$, le set des nœuds déjà visités, alors on ne le prends pas. On continue d'itérer jusqu'à ce que chaque nœud soit connecté ensemble (soit avoir un graphe connexe).

Disjoint partitions

Pour implémenter l'algorithme de kruskal, il est parfois intéressant de savoir manipuler des sets disjoints. Pour ce faire, on crée des sous arbre contenant des subgraphs, un élément doit se trouver dans un et un seul set.

On crée deux grandes fonctions importantes :

- **find(x)** : Qui donne le set qui contient x.
- **union(T1,T2)** : Qui fait l'union de deux sets disjoints.