Chapitre 1

Introduction

Définition 1 (Algorithme)**.** *Un algorithme est une suite finie d'opérations élémentaires constituant un schéma de calcul ou de résolution d'un problème.*

Double problèmatique :

— Trouver une méthode de résolution (exacte ou approchée) du problème
— Trouver une méthode efficace.

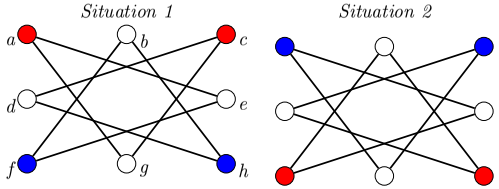
Dans tous les métiers en informatique, il faudra modéliser le problème et fournir une solution efficace. À l'heure des big-data, il est de plus en plus important de pouvoir traiter beaucoup de données en un temps raisonnable. Un film d'animation implémenté avec un algorithme naïf, nécessiterait plus de 70 ans pour la génération de ses images.

L'objectif de ce cours est de fournir un éventail de modélisations possibles et d'algorithmes efficaces associés. Il faut connaître les problèmes déjà répertoriés, la plupart ont déjà été étudiés par de nombreux chercheurs (il y a un très grand nombre de livres sur l'algorithmique). Des modélisations efficaces ont été proposées pour ces problèmes, il faut savoir les reconnaître et connaître les algorithmes classiques pour ces modélisations.

(Bon travail d'informaticien = bonne structure de donnée + bon algorithme + bon schéma de conception.)

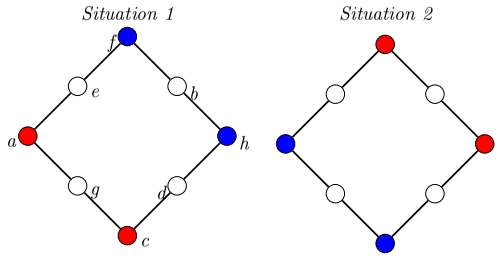
Exemple 1. *Exemple trouvé dans le jeu vidéo Machinarium.*

Problème : passer de la situation 1 à la situation 2 en déplaçant les pions selon les arêtes vers des positions vides.



En fait si on représente autrement (en dépliant le graphe (planaire)) alors on trouve beaucoup

plus facilement la solution !



⚠Importance de la représentation, trouver un algo revient à trouver la bonne modélisation du problème pour laquelle on connaît un algo efficace.

I Problèmes d'optimisation combinatoire

Euler (1707-1783) : "Il n'y a rien dans le monde qui ne se réalise sans la volonté de minimiser ou maximiser quelque chose."

Un problème combinatoire nécessite de trouver un élément noté S parmi un grand nombres d'autres éléments $\mathscr{S}$ tel que S satisfait certaines contraintes.

S peut être un groupement, un ordre, une partition, ou une affectation..., d'un ensemble fini d'objets $\mathscr{O}$.

Candidat $= S$ qui ne vérifie pas nécessairement les contraintes requises.
Solution réalisable $= S$ qui vérifie les contraintes requises.

Définition 2. *Un problème d'optimisation combinatoire est défini par :*
— *un ensemble énumérable $\mathscr{S}' \subseteq \mathscr{S}$ de* ***solutions réalisables***
— *et une fonction de coût/valeur $v : \mathscr{S} \longrightarrow \mathbb{R}$*

On cherche une solution S^* de valeur optimale – maximale (resp. minimale) suivant les problèmes :

$$S^* = \underset{S \in \mathscr{S}'}{\operatorname{argmax}}\{v(S)\} \quad (\textit{resp. argmin...})$$

(c'est à dire, S^* tel que $v(S^*) = \max_{S \in \mathscr{S}'}\{v(S)\}$ (resp. min))

Mauvaise nouvelle : pour certains problèmes on pense qu'il n'y a pas de méthode permettant de trouver efficacement (c'est à dire en temps polynomial par rapport à la taille des données) et à tous les coups la solution optimale (ce sont des problèmes *NP-complets*).

II Bref historique

Le mot "algorithme" provient le la forme latine (*algorismus*) du nom du mathématicien arabe Al Kharezmi ou Al-Khwârizmî (né en 780 en Ouzbékistan puis membre de la maison de la sagesse de Bagdad) auteur entre autre du manuel de calculs "'Abrégé du calcul par la restauration et la comparaison" (contenant le mot "al-jabr" qui a donné algèbre) définissant des algorithmes permettant de réaliser des opérations arithmétiques classiques (addition, soustraction, mulitplication, division, extraction de racines carrées, règle de trois, etc.) Ses écrits,

rédigés en langue arabe, puis traduits en latin à partir du XIIe siècle, ont permis l'introduction des chiffres dits *arabes* et de l'algèbre en Europe.

La théorie des graphes a commencé a être étudiée par Euler avec le problème des ponts de Königsberg (1736). Puis en 1914, König travaille sur les graphes bipartis et propose un théorème sur les couplages parfaits.

D'autre part en 1835, Kirchhoff établit une modélisation des circuits électriques, introduit les notions de tension, courant. Ce n'est que dans les années 1950 que ces travaux donnent naissance à la théorie des flots dans un graphe (théorème de Ford-Fulkerson 1956 liant le flot maximum et la coupe minimum). La théorie des flots s'applique aux problèmes de couplage et sera à l'origine de la programmation linéaire. Notons que les premiers mathématiciens qui se sont intéressés à des problèmes qu'on ne nommait pas encore programmes linéaires furent Laplace (1749-1827) et le baron Fourier. En 1939, c'est le russe Kantorovitch (prix Nobel d'économie en 1975) qui a imaginé une méthode inspirée des multiplicateurs de Lagrange, classiques en mécanique, pour résoudre des « programmes de transport ». La contribution décisive a été l'invention de l'algorithme du Simplexe, développé à partir de 1947 [1] notamment par George Dantzig (d'origine Russe immigré aux U.S.) et le mathématicien Von Neumann. Au milieu des années 80, l'indien Karmarkar a proposé une nouvelle méthode créée aux Bell Laboratories qui permettait de résoudre de très gros problème linéaires, par une démarche « intérieure » au polyèdre des solutions admissibles [2].

Les travaux sur les couplages donneront naissance à la théorie de la complexité : c'est Edmonds en 1965 qui introduit la notion d'algorithme polynomial pour caractériser l'efficacité de son algorithme de couplage, ce qui pousse Cook à montrer l'existence de problème NP-complet (SAT) cette découverte se fait en même temps en Russie : Levin étudiant de Kolmogorov en 1971 montre que le problème de pavage (tiling problem) [3] est NP-complet. Les algorithmes de pavage et de couplage ont progressé récemment afin d'avancer dans les recherches sur l'alignement de séquences d'ADN.

La théorie des flots et la programmation linéaire sont des éléments de l'optimisation combinatoire qui fait partie du domaine de la recherche opérationnelle [4].

1. Les modèles de programmation linéaire furent développés durant la seconde guerre mondiale pour résoudre des problèmes logistiques mais leur usage fut tenu secret jusqu'en 1947.

2. Application après-guerre : Opérations Vittles et Plainfare pour ravitaillement de la trizone pendant le blocus de Berlin par pont aérien (23 juin 1948 - 12 mai 1949), Simplexe exécuté à la main (des milliers de variables), jusqu'à 12 000 tonnes de matériel par jour !

3. Notons que le tiling problem correspond à la recherche d'un couplage maximal minimum (un assignement minimum en nombre d'arcs qui est maximal c'est-à-dire tel qu'on ne peut plus ajouter d'autres arcs) qui est NP-complet alors que trouver un couplage maximum (c'est-à-dire trouver un assignement avec un nombre maximum d'arcs) correspond à rechercher un flot maximum, ce qui est polynomial.

4. La recherche opérationnelle se situe au confluent des mathématiques discrètes (dans le sens où la notion de continuité n'intervient pas. Les objets étudiés en mathématiques discrètes sont des ensembles dénombrables comme celui des entiers. Les mathématiques discrètes incluent habituellement la logique (et l'étude du raisonnement), la combinatoire, les théories des ensembles, des nombres, des graphes, de l'information, de la calculabilité et de la complexité.) et de l'informatique théorique (Dijkstra était informaticien).

   Recherche opérationnelle : méthodes pour élaborer de meilleurs décisions dans des problèmes d'organisation. Elle tire son nom des applications militaires dont elle est issue, elle s'attaque à trois types de problèmes

   — combinatoire (trouver une solution optimale parmi un grand nombre de solutions.Exemple typique : déterminer où installer 5 centres de distribution parmi 30 sites d'implantation possibles, de sorte que les coûts de transport entre ces centres et les clients soient minimum. L'énumération des solutions $C_{30}^{5} = 30x29x28x27x26/(5x4x3x2) = 142\ 506$ doit être évitée.),

   — aléatoire ou stochastique (trouver une solution optimale en présence d'incertitude. Exemple typique : connaissant la distribution aléatoire du nombre de personnes entrant dans une administration en une minute et la distribution aléatoire de la durée de traitement pour une personne, déterminer le nombre minimum de guichets à ouvrir pour qu'une personne ait moins de 5% de chances de devoir attendre plus de 15 minutes.)

III Plan du cours

Que l'algorithme soit polynomial ou non, il faut trouver des moyens d'écrire des programmes qui donnent la solution en un temps raisonnable même si la taille des données est assez grande.

— On va dans un premier temps présenter des structures de données qui permettent de réduire les temps de traitement des données pour différentes opérations.

— On abordera ensuite les algos dédiés aux problèmes polynomiaux de flots (programmes linéaires particuliers) : Algo Ford-Fulkerson (efficace mais non polynomial (pseudo-poly) mais il existe une variante polynomiale (en prenant la plus courte chaîne augmentante en termes d'arcs). Certains problèmes peuvent être codés en termes de flots (plus courts chemins) alors que d'autres problèmes apparemment similaires ne sont pas traitables par des algos polynomiaux (voyageur de commerce).

— et programmation linéaire (Ensemble fini de solutions (les sommets du polytope convexe défini par les contraintes linéaires sauf si problème dégénéré (voir Chapitre 5 Section V))) : Algo du Simplexe (efficace mais non polynomial mais il existe un algo polynomial pour la prog linéaire)

— puis on étudiera les algos approchés (meta-heuristiques) permettant de traiter des problèmes plus généraux en particulier ceux qui n'ont pas d'algo polynomial (nombre chromatique, voyageur de commerce) .

— ou concurrentiel (trouver une solution optimale en prenant en compte les agissements d'autres décideurs. Exemple typique : fixer une politique de prix de vente, sachant que les résultats d'une telle politique dépendent de la politique que les concurrents adopteront.)

[22].

Première partie

Structures de données efficaces

Chapitre 2

Structures de données fondamentales

Description des structures de données et des algorithmes issues de : Introduction to Algorithms, Thomas H. Cormen et al. (2nd and 3rd edition) – Wikipedia - Tas Binomial – Wikipedia - Binomial Heap

Concepts clés et compétences transmises

Concepts présentés dans cette partie

- Tas et file de priorité
- Arbres binaire de recherche, équilibrage et opérations de dictionnaire

Compétences acquises dans cette partie

- Savoir choisir une structure de donnée élémentaire en fonction de l'utilisation que l'on doit en faire.

I Arbres et Forêts

Définition 3. *Un "arbre" (en algorithmique) est une structure de données récursive : qui est*
- *soit réduite à un seul noeud sans fils (dit feuille),*
- *soit composé d'un noeud (dit noeud interne) relié par des arcs vers des noeuds (dits fils) qui sont des arbres.*

Les arcs représentent une relation père-fils entre les noeuds, on les représente de haut en bas (on peut se passer des flèches). L'ordre des fils a une importance ("arborescence ordonnée"), on représente cet ordre de gauche à droite.

Remarque 1. *Dans le vocabulaire de la théorie des graphes, cette structure est appelée arborescence : graphe connexe et sans cycle, orienté avec une racine.*

Propriété 1. *Un arbre possède une racine unique et chaque noeud a un unique parent (il existe un seul chemin de la racine à tout noeud (sans cycle)).*

Définition 1 (Attributs des arbres)**.**
- *La profondeur d'un nœud est la distance (le nombre d'arcs) entre la racine de l'arbre et le nœud.*
- *La hauteur d'un nœud est la plus grande distance entre le nœud et une feuille de sa descendance.*
- *La hauteur d'un arbre est la plus grande profondeur d'une feuille de l'arbre (nombre de "niveaux" en graphes), c'est la hauteur de la racine.*

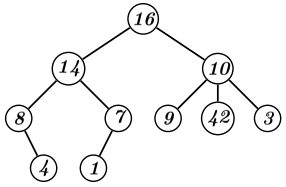
— *La taille d'un arbre est son nombre de nœuds.*
— *La clé (ou étiquette) d'un nœud est tout ou partie de l'information qui est stockée sur le nœud.*

Définition 2 (Principale classes d'arbres)**.**
— *Arbres binaire : chaque noeud à au plus 2 fils.*
— *Arbres k-aires*[1] *: chaque noeud à au plus k fils.*

Définition 3.
— *Un arbre k-aire est dit plein si toutes les feuilles ont la même profondeur et tous les nœuds internes ont le degré k.*
— *Un arbre est dit presque plein*[2] *si tous les niveaux sont remplis, sauf éventuellement le dernier, et que celui ci est rempli de gauche à droite*[3]*. Dans un arbre presque plein, la profondeur d'une feuille est soit la hauteur h de l'arbre, soit h − 1.*
— *Un arbre binaire est dit équilibré si pour chaque noeud, son sous-arbre droit est de même hauteur que son sous-arbre gauche (avec hauteur de l'arbre vide=-1)*

Une forêt est un ensemble d'arbres.

Exemple 2. *Illustrer toutes les notions dessus. racine, père, fils, noeud, feuilles, noeud interne, clé, profondeur du noeud 7=2, hauteur=3, taille=10, k = 3 (arbre ternaire), plein ($(3^4-1)/(3-1) = 40$ noeuds en tout), presque-plein (il faudrait bouger le noeud 7 et ajouter 5 noeuds (1 prof 1, 4 prof 2, 2 prof 3)), équilibré (ajouter 4 feuilles).*



II Gestion de priorité : Tas binaire

Objectif : accéder en $\Theta(1)$ à l'élément le plus prioritaire d'une collection de n éléments.

Définition 4 (tas binaire)**.** *Un tas binaire*
— *est un arbre binaire presque-plein.*
— *est un tas : arbre dont l'ensemble des clés E est muni d'un ordre total $\succ$*[4] *et qui vérifie :*

$$\forall i, j \text{ t.q. } i = pere(j), \quad clé(i) \succ clé(j) \quad \text{“la clé du père est prioritaire sur la clé du fils”}$$

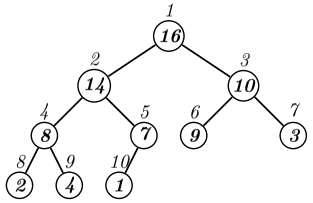
— *Si $\succ$ est la relation $>$, le tas est appelé tas maximal, si $\succ$ est la relation $<$, il est appelé tas minimal.*

1. On réserve n pour la taille.
2. L'article de wikipedia “tas binaire” parle d'”arbre parfait” mais cela semble contredire les définitions habituelles.
3. Les fils d'un noeud qui ont eux-même le plus de fils sont mis en premier
4. Avec $\prec$ defini par $a \prec b \Leftrightarrow b \succ a$

Représentation :

Un tas binaire étant un arbre presque-plein, l'implantation la plus compacte et la plus efficace repose sur un tableau dont la taille (dénommée Length) détermine le nombre maximum de noeuds que l'on peut stocker et auquel on ajoute un attribut (appelé Size) permettant de connaître le nombre effectif de noeuds du tas. Le premier élément du tas (la racine) est à l'indice 1 et le dernier à l'indice $n =$ Size du tableau.

Exemple 3. *Un tas binaire maximal, vu en (a) comme un arbre et en (b) comme un tableau.*



(a)

1	2	3	4	5	6	7	8	9	10	11	12	13
16	14	10	8	7	9	3	2	4	1			

(b) Length=13, Size=10

Avec cette représentation, les arcs de l'arbre sont implicitement représentés par les fonctions suivantes pour un nœud à l'indice i :

Function Parent(i)
return $\lfloor i/2 \rfloor$

Function Left(i)
return $2i$

Function Right(i)
return $2i + 1$

La fonction Left calcule $2i$ en effectuant un décalage à gauche de 1 bit avec insertion d'un 0. De même, Right calcule $2i + 1$ en effectuant un décalage à gauche de 1 bit avec insertion d'un 1 sur le bit de poids faible. La fonction Parent calcule $\lfloor i/2 \rfloor$ en effectuant un décalage à droite de 1 bit.

Opérations :

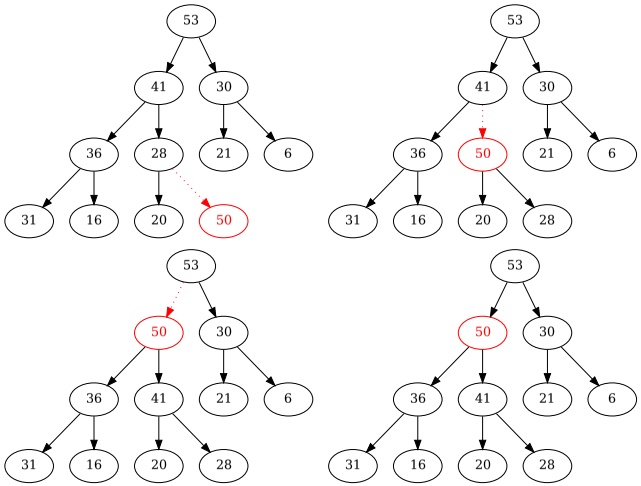
Un tas binaire supporte les opérations suivantes détaillées ci-dessous :
— **Add** : ajout d'un élément dans le tas binaire.
— **Remove** : supprime la racine d'un tas binaire.
— **IncreaseKey** : augmente la valeur d'une clé d'un élément (dont on connait l'indice).
— **DecreaseKey** : diminue la valeur d'une clé d'un élément (dont on connait l'indice).
— **BuildHeap** : construction d'un tas binaire à partir d'un ensemble d'éléments.

⚠ Les algorithmes sont illustrés avec une représentation en arbre mais sont écrits (en Annexe) en fonction de la représentation en tableau avec les attributs Length et Size.

1 Add $T_{Add}(n) \in \Theta(\log n)$, $S_{Add}(n) \in \Theta(1)$

L'insertion d'un élément dans un tas binaire. S'effectue en deux étapes :
— ajouter l'élément à la première position disponible dans le tas (en T[SIZE+1])
— rétablir la propriété de tas (si violée par l'ajout de l'élément) : cette opération s'appelle percolation ascendante (PercolateUp) qui consiste à remonter l'élément vers la racine tant qu'il est plus prioritaire que son père en l'échangeant avec celui-ci.

Exemple 4. *Étapes de l'insertion de l'élément 50 dans un tas binaire maximal.*



Complexité : Soit h la hauteur du tas binaire, lors du PercolateUp on effectue au plus h échanges. Or comme un tas binaire est un arbre binaire presque-plein on a $h \leq \log_2 n$ (où n est le nombre de nœuds du tas binaire) [5]. Donc complexité temporelle en pire cas, $T_{Add}(n) \in \Theta(\log n)$.

Pour la complexité spatiale, les changements se font en place donc $S_{Add}(n) \in \Theta(1)$.

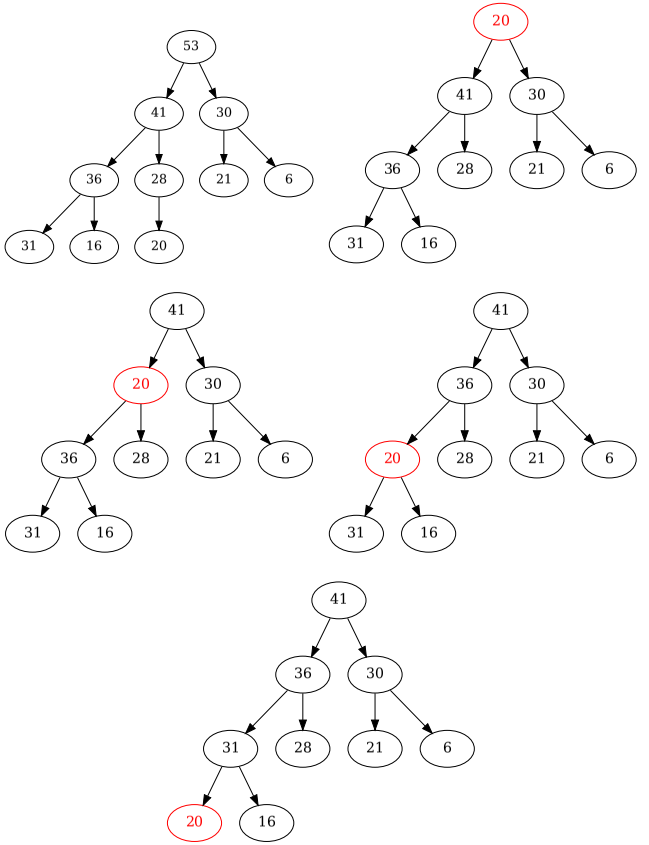
2 Remove $T_{Remove}(n) \in \Theta(\log n)$, $S_{Remove}(n) \in \Theta(1)$

L'accès à la racine (donc l'élément le plus prioritaire du tas) se fait en temps constant mais suppression nécessite le rétablissement de la propriété de tas sur l'arbre résultant, 2 étapes :

- remplacer la racine par le dernier élément du tas (l'élément le plus à droite sur le dernier niveau) c'est à dire diminuer de 1 la taille du tas,
- rétablir la propriété de tas par une opération de percolation descendante(PercolateDown) qui consiste à descendre l'élément tant qu'il est moins prioritaire qu'au moins un de ses fils en le permutant avec le plus grand de ses deux fils.

Exemple 5. *Étapes de suppression de la racine dans un tas binaire maximal.*

5. l'arbre presque plein le plus petit est plein jusqu'au niveau $h-1$ et n'a qu'un noeud au niveau h donc $n \geq$ (nombre de noeuds jusqu'au niveau $h-1$) $+$(1 noeud niveau h) $= \sum_{i=0}^{h-1} 2^i + 1 = S = 2^h - 1 + 1$ (car $2S - S = \sum_{i=1}^{h} 2^i - \sum_{i=0}^{h-1} 2^i$). Donc $n \geq 2^h$ d'où $h \leq \log_2(n)$.



Complexité : la complexité de la fonction PercolateDown (écrite par récurrence) [6], sur un tas de taille n enraciné au nœud d'indice i est en $\Theta(1)$ pour corriger les relations entre les nœuds $T[i]$, $T[\text{Left}(i)]$ et $T[\text{Right}(i)]$, plus le temps d'exécuter PercolateDown depuis un des fils de i.

Comme un tas est un arbre binaire presque-plein, les sous arbres ont une taille [7] inférieure à $\frac{2}{3}n$, le pire cas arrivant lorsque le dernier niveau est exactement rempli à moitié. Le temps de calcul de la fonction PercolateDown peut donc se baser sur la récurrence $T(n) = T(\frac{2}{3}n) + \Theta(1)$. En utilisant le deuxième cas du théorème **??**, la solution de cette récurrence est $T(n) \in \Theta(n^{\log_{3/2} 1} \log n)$ donc Remove $\in \Theta(\log n)$. Pour la complexité spatiale, les changements se font en place donc $S_{Remove}(n) \in \Theta(1)$.

6. On peut aussi faire le même raisonnement sur la hauteur que pour Add en écrivant l'algo avec un while.
7. Taille du sous arbre gauche de hauteur $h-1$ est $sag = 2^h - 1$, taille du tas (sous-arbre droit+gauche+racine) : $n = 2^h - 1 + 2^{h-1} - 1 + 1 = \frac{3}{2}2^h - 1$ donc $sag = \frac{2}{3}n - \frac{1}{3}$.

3 IncreaseKey et DecreaseKey $T(n) \in \Theta(\log n)$, $S(n) \in \Theta(1)$

La valeur d'un élément est représentée par sa clé au sein du tas et il est souvent nécessaire de faire évoluer cette valeur (augmenter IncreaseKey, ou diminuer DecreaseKey), mais en gardant la propriété de tas. Si on augmente la priorité de la clé on fera donc un PercolateUp à partir de l'élément modifié et si on la diminue on fera un PercolateDown.

Dans le pire cas de ces opérations il faudra remonter ou descendre tout l'arbre depuis une feuille ou la racine, la complexité en pire cas est donc l'utilisation des algorithmes de percolation sur toute la hauteur de l'arbre c'est à dire $\Theta(\log n)$.

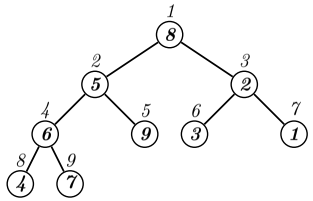
4 BuildHeap $T_{BuildHeap}(n) \in \Theta(n)$, $S_{BuildHeap}(n) \in \Theta(1)$.

Lorsque l'on souhaite construire un tas binaire à partir d'un ensemble connu d'éléments (un tableau), on considère le tableau comme un arbre binaire et on va rétablir la propriété de tas.

Exemple 6.
*Soit le tableau suivant de taille T.*Size$= 9$ *à partir duquel on veut construire un tas maximal :*

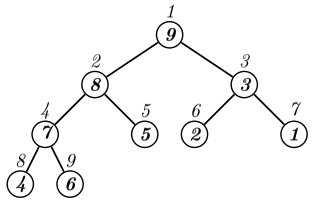
8	5	2	6	9	3	1	4	7

On lui associe l'arbre binaire suivant :



On rétablit la propriété depuis les feuilles vers la racine en utilisant la percolation vers le bas. En pratique, les feuilles ne bougeront pas donc on commence à partir du premier noeud possédant des fils. Comme un tas binaire est un arbre presque-plein, les éléments du sous-tableau $T[(\lfloor n/2 \rfloor + 1)..n]$ seront les feuilles de l'arbre et sont donc des tas binaire à 1 élément valides. L'opération de construction va donc appliquer PercolateDown à partir du précédent élément du tableau $T[\lfloor n/2 \rfloor]$ (père du dernier élément) jusqu'à la racine $T[1]$.

Exemple 6 (suite)
$\lfloor T.Size/2 \rfloor = 4$*, on commence donc le* PercolateDown *sur l'indice 4, ce qui fait échanger 6 et 7, on continue sur l'indice 3 etc. Ce qui donne :*



Chaque appel à l'opération PercolateDown nécessite dans le pire des cas un temps en $O(\log n)$ et BuildHeap effectue $O(n)$ appels à PercolateDown. Nous avons donc la borne

maximale de complexité de l'opération de construction en $O(n \log n)$. Toutefois, bien que cette borne soit correcte, elle n'est pas asymptotiquement précise et surévalue la complexité de l'algorithme.

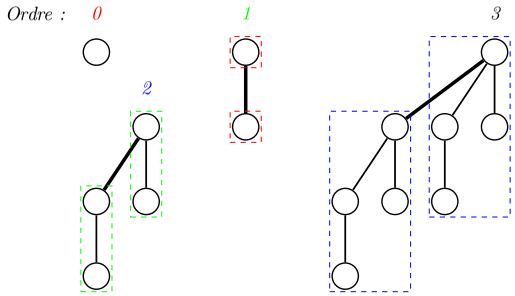
Comme le temps de l'opération PercolateDown appelée sur un nœud de hauteur h est en $\Theta(h)$, en exprimant $T(n)$ en fonction de la somme des traitements des noeuds de différentes hauteurs[8] h avec h qui varie de 1 à $\log n$, on arrive à une complexité en pire cas en $\Theta(n)$ (voir TD). La complexité spatiale est en $\Theta(1)$ puisqu'on effectue la construction en place.

III Gestion de priorité et union : Tas binomial

Sur la structure de données précédente décrivant un tas binaires, l'opération d'union à une complexité en $\Theta(n)$ pour fusionner 2 tas binaire représentant n éléments au total. Objectif ramener cette complexité à $\Theta(\log n)$ dans le pire des cas.

Définition 5 (Arbre binomial)**.** *Un arbre binomial est défini par récurrence sur son "ordre" :*
- *Un arbre binomial d'ordre 0 possède 1 seul nœud, sa racine, qui est aussi une feuille.*
- *Un arbre binomial d'ordre $k > 0$ est constitué de deux arbres binomiaux d'ordre $k-1$ reliés par un arc père fils entre leurs racines.*

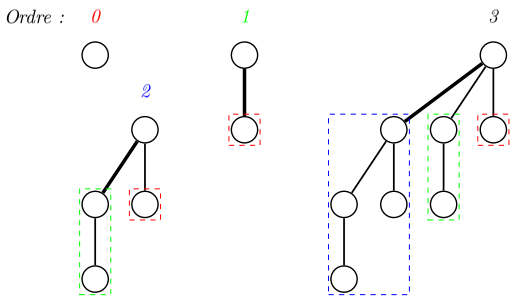
Exemple 7. *Exemples :*



Propriété 2.
- *Un arbre binomial d'ordre k possède 2^k nœuds.*
- *Un arbre binomial d'ordre k est de hauteur k.*
- *Dans un arbre binomial d'ordre k, il y a exactement C_k^i nœuds de profondeur i (avec $0 \leq i \leq k$), avec C_n^p le coefficient binomial donnant le nombre de parties à p éléments dans un ensemble à n éléments*[9]*.*
- *La racine d'un arbre binomial d'ordre k possède k fils qui sont les racines d'arbres binomiaux d'ordre respectifs $k-1$, $k-2$, $\ldots$, 0.*

Exemple 8. *Illustration de la prop.*

8. Ne pas confondre "hauteur" et "profondeur".
9. ça se démontre par récurrence.



Définition 6 (Tas binomial)**.** *Un tas binomial est une forêt d'arbres binomiaux satisfaisant les propriétés :*

- *Chaque arbre du tas binomial est un tas minimal :*
  La clé du père est inférieure ou égale à celle de son fils.
- *Pour tout $k >= 0$, il y a au plus un arbre binomial d'ordre k dans le tas.*

Remarque 2. *La racine de chaque arbre du tas contient la valeur minimale de cet arbre.*

Propriété 3. *Un tas binomial avec n éléments contient au plus $\log_2 n + 1$ arbres binomiaux.*

Démonstration : Comme un arbre d'ordre i possède 2^i nœuds, alors un tas de taille n peut contenir un arbre d'ordre maximal $\lfloor \log_2 n \rfloor$. D'après la définition, il ne contient au plus qu'un tas de chaque ordre entre 0 et son ordre max. De 0 à $\lfloor \log_2 n \rfloor$, il y a $\lfloor \log_2 n \rfloor + 1$ éléments. Il ne peut donc contenir au max que $\lfloor \log_2 n \rfloor + 1$ arbres. □

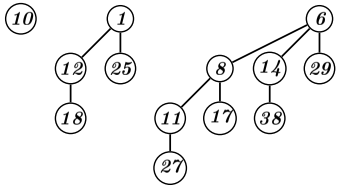
En utilisant la preuve de la propriété précédente on déduit que :

$$n = \sum_{i=0}^{\lfloor \log_2 n \rfloor} b_i 2^i$$

avec $b_i = 1$ si l'arbre d'ordre i est dans le tas, 0 sinon.

Corollaire 1. *Un tas binomial contient un arbre d'ordre i ssi le ième bit (en partant de la droite) pour coder son nombre d'éléments est un 1.*

Exemple 9. *Le nombre* 13 *s'écrit en binaire comme* 1101 *et un tas binomial possédant* 13 *éléments est composé des arbres binomiaux d'ordre* 3, 2 *et* 0. *La figure ci-dessous donne un exemple d'un tel tas.*



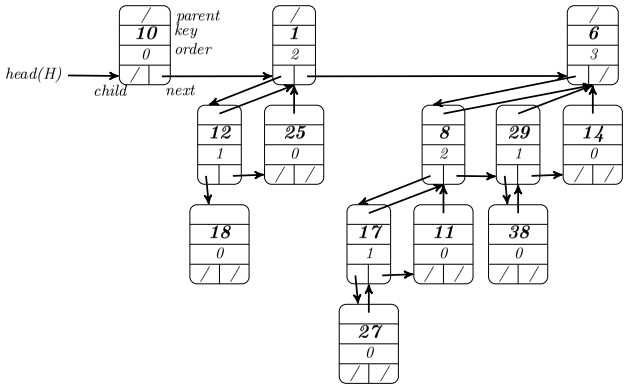
Représentation :

Un nœud est représenté par une structure contenant :

- un lien vers le père,
- la clé,
- l'ordre de l'arbre binomial,
- un lien vers le fils et un lien vers le frère (ou racine suivante).

Les noeuds contenant les racines (père = NIL) des arbres du tas binomial sont stockés dans une liste chaînée ordonnée selon les "ordres des arbres" croissants.

Exemple 9 (suite) *Représentation d'un tas binomial.*



Opérations :

Un tas binomial supporte les opérations suivantes, toute de complexité temporelles $\Theta(\log n)$ en pire cas, mais, pour certaines, peuvent être ramenées en $\Theta(1)$ par une implantation adéquate :

- **Add** : ajout d'un élément dans le tas binomial.
- **Find** : recherche de l'élément ayant la clé minimum.
- **RemoveMin** : supprime l'élément minimum d'un tas binomial.
- **DecreaseKey** : diminue la valeur d'une clé d'un élément.
- **Remove** : supprime un élément d'un tas binomial.
- **Merge** : réalise l'union de deux tas binomiaux.

1 Merge ($T_{Merge}(n) \in \Theta(\log n)$, $S_{Merge}(n) \in \Theta(1)$)

L'opération d'union de deux tas binomiaux est l'opération centrale (justifie cette structure et utilisée par les autres opérations).

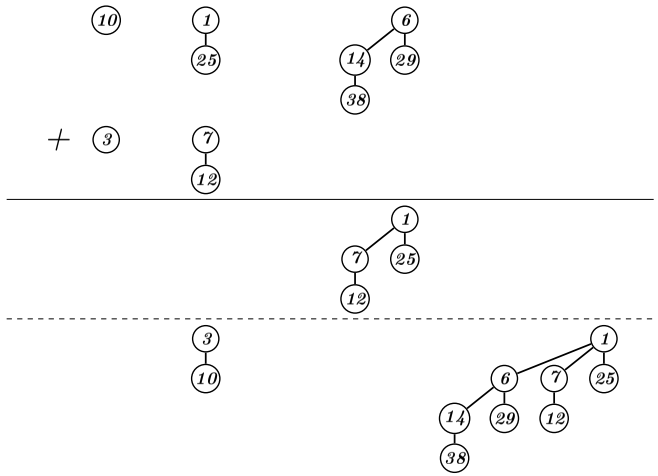
L'algorithme d'union de deux tas binomiaux se déroule en deux étapes :

- fusionner les deux listes de racines d'arbres binomiaux en les ordonnant selon leur degré [10] (fonction RootMerge). La liste résultante ne peut donc contenir, au plus, que 2 arbres binomiaux de même ordre, qui seront consécutifs dans la liste.

10. Le degré des racines est aussi l'ordre des arbres.

— fusionner les arbres de même ordre (celui de racine minimale en haut) jusqu'à ce qu'il ne reste au plus qu'un seul arbre par ordre. Dans le cas où trois arbres de même ordre se suivent (à cause d'une union au niveau inférieur) alors on saute le premier et on fusionne les deux suivants.

Exemple 10. *Exemple de fusion de deux tas binomiaux.*



Complexité : Comme la liste des racines est triée par degré croissant, cette opération se réalise en un seul parcours avec une complexité en $\Theta(m)$ si les deux listes contiennent au total m racines. Il est à noter au passage que $m \leq \lfloor \log_2 n \rfloor + 1$. La fusion de deux arbres binomiaux étant réalisée avec une complexité en $\Theta(1)$, la fusion de tas binomiaux, consistant à fusionner au plus $\lfloor \log_2 n \rfloor + 1$ arbres est donc réalisée en $\Theta(\log n)$. En espace, en réalité la fonction RootMerge ne crée pas une nouvelle liste d'arbres binomiaux mais change directement les pointeurs des deux listes d'arbres existants pour en faire une seule liste chaînée ordonnée selon les ordres des arbres, donc $S_{Merge}(n) \in \Theta(1)$

2 Add (T en $\Theta(\log n)$ et S en $\Theta(1)$)

L'ajout d'un nouvel élément dans un tas binomial :
— créer un tas binomial ne contenant que l'élément à insérer
— fusionner le tas obtenu avec le tas de destination.

En raison de la complexité de l'opération d'union, insérer un élément dans un tas de n éléments se fait donc en $\Theta(\log n)$.

3 Find (T en $\Theta(\log n)$ et S en $\Theta(1)$)

Rechercher l'élément minimum dans un tas binomial revient à rechercher l'élément minimum dans la liste des racines.

Comme un tas binomial de n éléments possède au plus $\lfloor \log_2 n \rfloor + 1$ arbres binomiaux, une recherche naïve dans la liste permet de réaliser cette opération en $\Theta(\log n)$.

Toutefois, en gérant explicitement une référence vers l'élément minimum, cette opération sera réalisable en $\Theta(1)$. Il faudra alors mettre à jour cette référence lors de toute opération modifiant la valeur du minimum.

4 RemoveMin (T en $\Theta(\log n)$ et S en $\Theta(1)$)

Supprimer l'élément minimum d'un tas binomial :
— construire la liste des fils de l'élément minimum, dans l'ordre inverse de ceux-ci (pour avoir les racines dans l'ordre des degrés croissants), et vider les champs père de ceux-ci, on obtient un tas binomial particulier avec exactement un arbre de chaque ordre de 0 à l'ordre du père -1. (la création de la liste chainée des fils se fait en place on change juste les pointeurs frères et pères)
— fusionner le tas binomial résultant de la suppression avec le tas des fils.

Comme un tas binomial contient au plus $\lfloor \log_2 n \rfloor + 1$ arbres binomiaux d'ordre maximum $\lfloor \log_2 n \rfloor$, cette opération est réalisable en pire cas en $\Theta(\log n)$ (somme des deux).

5 DecreaseKey (T en $\Theta(\log n)$ et S en $\Theta(1)$)

Diminuer la clé d'un élément dont on a l'adresse dans un tas binomial. Cela peut rompre la propriété de tas.

DecreaseKey nécessite donc d'effectuer ensuite une opération de percolation vers la racine de façon à faire remonter cet élément si sa valeur se trouve inférieure à celle de ses parents.

La hauteur d'un arbre binomial dans un tas binomial étant au plus $\lfloor \log n \rfloor$, cette opération se fait donc en pire cas avec une complexité en $\Theta(\log n)$.

6 Remove (T en $\Theta(\log n)$ et S en $\Theta(1)$)

Supprimer un élément d'un tas binomial :
— diminuer la clé de l'élément à $-\infty$ par l'opération DecreaseKey. L'élément devient alors l'élément minimal du tas.
— supprimer le min par l'opération RemoveMin.

IV Structures de données pour la recherche d'information

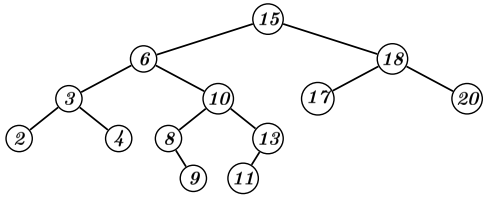
1 Arbres binaires de recherche (BST)

Tas : rechercher un élément quelconque reste en $\Theta(n)$ en pire des cas. Objectif : $\Theta(h)$ avec h la hauteur de l'arbre.

Définition 7. *Un arbre binaire de recherche est un arbre binaire, avec les fonctions d'accès parent(x), left(x), right(x), key(x) et pour tout noeuds x et y,*

Si y est un nœud du sous arbre gauche de x, alors $key(y) \leq key(x)$
Si y est un nœud du sous arbre droit de x, alors $key(x) \leq key(y)$

Exemple 11. *La clé de la racine est 15. Les clés des éléments du sous arbre gauche sont toutes inférieures à la clé de la racine. Les clés du sous arbre droit sont toutes supérieures à la racine. Cette propriété est vraie pour tous les sous arbre enracinés sur un nœud quelconque de l'arbre.*



Opérations :

Search, **Minimum**, **Maximum**, **Predecessor**, **Successor**, **Insert** et **Delete** : toutes de complexité en $\Theta(h)$ en pire cas avec h la hauteur de l'arbre binaire. Cela permet d'utiliser les arbres binaire de recherche à la fois comme dictionnaires et comme des files de priorité.

La propriété des arbres binaire de recherche permet de parcourir les noeuds de l'arbre dans l'ordre croissant des clés avec un algorithme récursif simple : c'est le parcours Infixe (cf. InOrderTreeWalk en Annexe). Les autres parcours sont :

Parcours Infixe	Parcours Préfixe	Parcours Postfixe
fils gauche	noeud	fils gauche
nœud	fils gauche	fils droit
fils droit	fils droit	nœud

Les opérations de parcours d'un arbre binaire de recherche s'effectuent toutes en $\Theta(n)$.

Search ($\mathbf{T} \in \Theta(h)$, $S \in \Theta(1)$

La recherche d'une clé dans un arbre binaire de recherche utilise la propriété d'ordonnancement total des nœuds. L'algorithme récursif suivant retourne le nœud de l'arbre si la clé recherchée existe et retourne NIL sinon.

Function TreeSearch(x, k)

if $(x = NIL)$ *or* $(k = key(x))$ **then return** x
if $(k < key(x))$ **then return** TreeSearch$(left(x), k)$
else return TreeSearch$(right(x), k)$

Seul 1 sous arbre est examiné pour un nœud dont la clé n'est pas égale à la clé recherchée. Ainsi, la complexité de cette fonction en pire cas est bien $\Theta(h)$ avec h la hauteur de l'arbre. [11]

Minimum et Maximum $\Theta(h)$ ou $\Theta(1)$

L'élément minimal est l'élément le plus à gauche dans l'arbre, l'élément maximal est le plus à droite.

Bien que les complexité de ces deux opérations soient en $\Theta(h)$, il est possible d'étendre la structure de donnée afin d'accéder en $\Theta(1)$ à l'élément minimum et à l'élément maximum. Une telle extension est réalisée dans la structure de donnée Arbre binaire cousu (binary threaded tree).

11. Afin de rendre cette fonction plus efficace sur la plupart des architectures, il est possible d'écrire une version itérative en déroulant simplement la récursivité terminale (cf. Annexe).

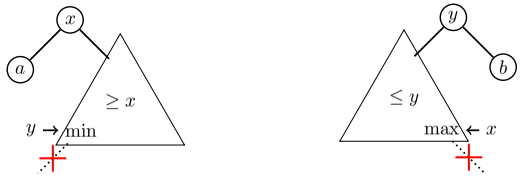
Predecessor et Successor $\Theta(h)$

Il est possible, via un parcours infixe, de parcourir de façon linéaire l'ensemble des éléments ordonnés mais pour chercher le successeur d'un élément x depuis cet élément, il y a deux cas :

a) si le sous arbre droit du nœud x n'est pas vide, son successeur est le minimum dans ce sous-arbre ($succ(x) = \min_y\{y \geq x\}$).

b) si le sous arbre droit du nœud x est vide et que x à un successeur y alors, y est le premier ancètre de x qui a un fils gauche ancètre de x. On remonte l'arbre jusqu'à ce que l'on accède à un nœud y depuis son fils gauche. ($pred(y) = \max_z\{z \leq y\}$) est le max d'un sous-arbre gauche du premier y supérieur à x)

Exemple 11 (suite) *Successeur de 6 : 8 cas a). Successeur de 13 :15 cas b).*



a) x a un fils droit b) x n'a pas de fils droit

FIGURE 2.1 – Successeur de x les deux cas

La fonction pour SUCCESSOR est la symétrique de PREDECESSOR.

Insert $\Theta(h)$

L'algorithme commence par rechercher le nœud y qui sera le parent (même principe que Search mais en mémorisant les pères des noeuds) du nouveau nœud z. Si y n'existe pas, alors z devient la racine de l'arbre, sinon, z est attaché comme fils gauche ou fils droit de son père en fonction de la valeur de sa clé (on ajoute toujours en créant une nouvelle feuille).

La recherche du parent du nœud inséré consistant à parcourir l'arbre en profondeur, la complexité de l'opération d'insertion est en $\Theta(h)$ si h est la hauteur de l'arbre.

Delete $\Theta(h)$

Lorsque l'on supprime un nœud x de l'arbre, trois cas :

a) x est une feuille. Dans ce cas, la suppression est triviale et il suffit de modifier le père de x pour qu'il ne possède plus x comme fils mais NIL.

b) x ne possède qu'un seul fils. Dans ce cas, il suffit de modifier le père de x pour qu'il possède comme fils le fils de x.

c) x a 2 fils, dans ce cas, le successeur y de x est calculé comme dans le cas a) il est donc soit une feuille, soit ne possède qu'un seul fils. L'algorithme remplace dans l'arbre le nœud x par le nœud y. Le fils droit éventuel de y remonte d'un cran (y n'a pas de fils gauche).

Exemple 11 (suite) *Cas b) suppression de 8 : on remonte 9 comme fils de 10. Cas c) suppression de 6 : on remplace 6 par 8 et on remonte 9.*

En raison de la recherche du successeur du nœud à supprimer, la complexité de l'opération TREEDELETE est en $\Theta(h)$ avec h la hauteur de l'arbre.

Rotations $\Theta(1)$

Une rotation est une opération locale, qui s'exécute en $\Theta(1)$ et qui préserve l'ordonnancement des nœuds tout en changeant la structure de l'arbre.

— Rotation gauche sur le nœud x, on suppose que son fils droit y n'est pas $nil(T)$. La rotation gauche fait pivoter l'arbre vers la gauche autour de x. y, le fils droit de x, devient la racine du sous arbre (à la place de x), x devient le fils gauche de y et le fils gauche de y devient le fils droit de x.

— La rotation droite est symétrique de la gauche.

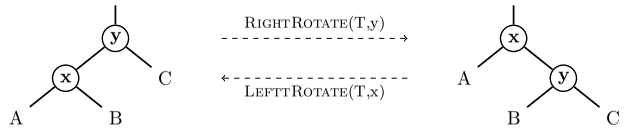
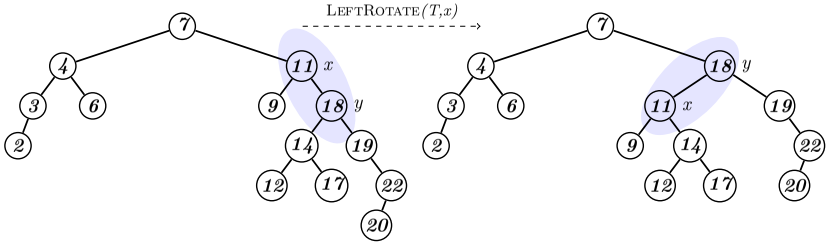


FIGURE 2.2 – Rotations sur un sous-arbre, left-rotate(x) :$y \neq$ NIL, right-rotate(y) : $x \neq$ NIL

Rééquilibrer signifie diminuer la longueur de la branche la plus longue, cela peut se faire par rotation.

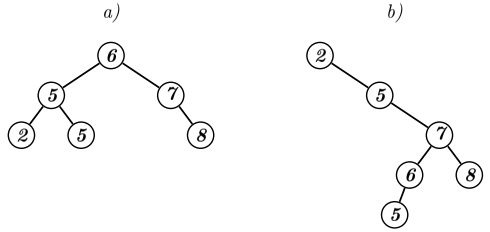
Exemple 12. *Effet d'une rotation gauche sur le rééquilibrage d'un arbre.*



2 Arbres presque-équilibrés : arbres rouge/noir et arbres AVL

Les opérations sur les arbres binaires de recherche sont en $\Theta(h)$. Mais si l'arbre est dégénéré (ex : insertions dans l'ordre croissant des clés), la hauteur est n donc $\Theta(n)$, ce qui ne correspond pas aux objectifs visés.

Exemple 13. *Exemples de 2 arbres binaires de recherche construits sur un même ensemble de clés. L'ordre d'ajout des éléments dans l'arbre a un impact sur la profondeur de l'arbre, et donc sur le temps de calcul des opérations élémentaires.*



On peut montrer que la construction d'un arbre en insérant les éléments dans un ordre aléatoire [12] donnera, en moyenne, un arbre de profondeur $\log n$ qui sera donc optimal pour les différentes opérations.

12. Une permutation possible parmi les $n!$.

Efficacité maximale implique hauteur minimale de l'arbre, c'est à dire $\log_2 n$ avec n le nombre de noeuds. Arbre **équilibré** : toutes les branches ont la même longueur implique que n est une puissance de 2 : pas toujours possible.

Définition 4. *Un arbre presque-équilibré si sa profondeur est en* $O(\log n)$.

Les arbres rouge/noir et les arbres AVL (du nom de leurs inventeurs Georgii Adelson-Velsky et Evguenii Landis) (voir TD) sont des structures d'arbres presque-équilibrés.

Définition 8. *Un arbre Rouge/Noir est un arbre binaire de recherche t.q. :*

1. *Chaque nœud de l'arbre est soit rouge, soit noir.*
2. *La racine de l'arbre est noire.*
3. *Si un noeud n'a pas de fils gauche ou droite alors ce fils inexistant est noté NIL, une feuille NIL est de couleur noire.*
4. *Si un nœud est rouge, alors ses deux fils sont noirs.*
5. *Pour chaque nœud, tous les chemins de ce nœud vers une feuille descendante possèdent le même nombre de nœuds noirs.*

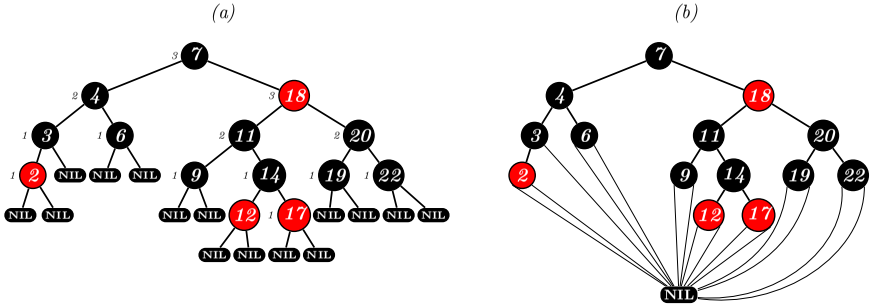
Propriété 4. *Un arbre Rouge/Noir vérifie*
- *aucune branche n'est plus de 2 fois plus longue qu'une autre.*
- *sa hauteur maximale est* $2\log_2(n+1)$ *s'il posséde n nœuds internes (presque-équilibré).*

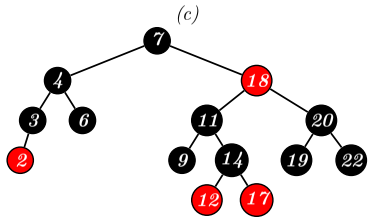
Le presque-équilibre peut être démontré en utilisant la notion de hauteur noire $h_b(x)$ d'un nœud x, qui indique le nombre de nœuds noirs sur la branche reliant le nœud x à une feuille.

Représentation

Un noeud est représenté par une structure avec les fonctions color, key, left, right et parent. En pratique la valeur NIL (parent inexistant ou fils inexistant) désigne un nœud unique de couleur noire appelé sentinelle et notée $nil(T)$ pour un arbre T.

Exemple 14. *La figure ci-dessous montre 3 visualisations différente du même arbre Rouge/Noir (c'est l'arbre précédent avec un* RightRotate *en 22 puis* LeftRotate *en 19) en (a) : l'arbre avec toutes les feuilles distinctes et les hauteurs noires, en (b) : l'arbre avec la sentinelle NIL et en (c) : la visualisation classique qui omet la sentinelle.*





Opérations

Les opérations disponibles sur les arbres Rouge/Noir sont les même que sur les arbres binaires de recherche : **Search**, **Minimum**, **Maximum**, **Predecessor**, **Successor**, **Insert**, **Delete**.

Seules les opération Insert et Delete doivent être réécrites de façon spécifique pour les arbres Rouge/Noir car elles entraînent la nécessité de recoloriage et de modification de la structure de l'arbre.

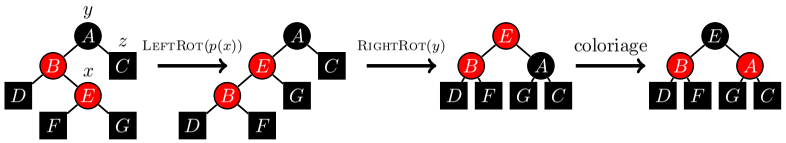
Insertion $\Theta(\log n)$

L'insertion d'un nœud en $\Theta(\log n)$ puisque la longueur maximale d'une branche dans un tel arbre est de $2\log_2(n+1)$. On choisit d'insérer un noeud en lui affectant systématiquement la couleur rouge pour ne pas changer la hauteur noire. Il faut ensuite rectifier l'arbre avec RBTreeInsertFixup(T, z) car on pourrait obtenir deux noeuds rouges consécutifs (algorithme en $\Theta(\log n)$).

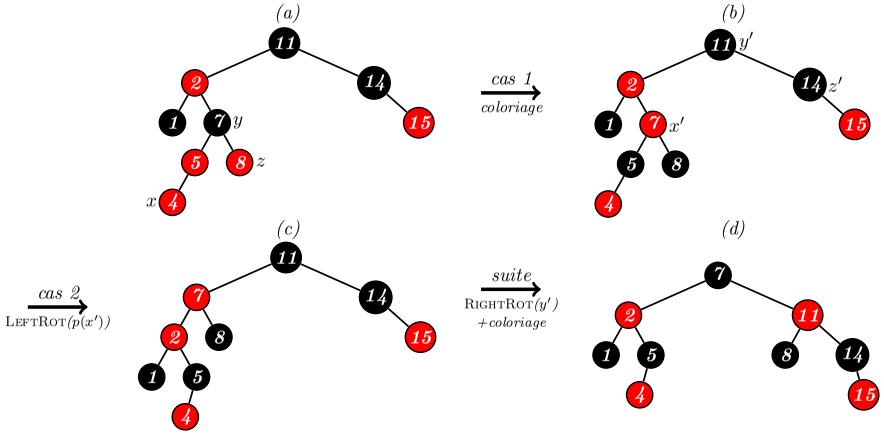
Seules les propriétés 2 et 4 sont concernées. La violation de la propriété 2 seule ne pose pas de problème dans la mesure où le coloriage de la racine en noir la rétablit sans violer les autres propriétés. Analysons donc le cas où seule la propriété 4 est violée. Un nœud rouge x possède donc un père rouge. Considérons le sous-arbre enraciné en le grand-père y de x. Supposons que x est dans le sous-arbre gauche de y (l'autre cas se traite de façon symétrique). Il y a deux possibilités selon que l'oncle z de x est rouge ou noir.

Cas 1 : z est rouge Il existe deux sous-cas qui sont traités de la même manière, selon que x est un fils gauche ou droit. Le premier sous-cas est illustré par la figure ci-dessous. Un recoloriage rétablit les propriétés d'arbre rouge-noir sauf éventuellement la 2 et aucune hauteur noire n'est modifiée. En revanche, il est possible que le père de y soit rouge, ou que y soit la racine. La correction devra donc se poursuivre au niveau supérieur depuis y.

Cas 2 : z est noir Ce cas est illustré par la figure ci-dessous. On se ramène d'abord au cas où x est un fils gauche par une rotation gauche (première flèche), puis on effectue une rotation droite (seconde flèche) et un recoloriage (troisième flèche). À l'issue de cette opération, le sous-arbre obtenu est un arbre rouge-noir, et aucune hauteur noire n'a été modifiée : la correction est donc terminée. Notons que si x était déjà un fils gauche alors on ne fait que les deux dernières opérations.



Exemple 15. *Correction d'un arbre Rouge/Noir après insertion du nœud* 4 *:*



Suppression $\Theta(\log n)$

Même algorithme que pour les arbres binaires de recherche + correction de la structure de l'arbre si les propriétés des arbres Rouge/Noir ne sont plus vérifiées.

- Si le nœud qui porte la valeur à supprimer possède zéro ou un fils, c'est ce nœud qui est supprimé et son éventuel fils prend sa place.
- Si, au contraire, ce nœud possède deux fils, il n'est pas supprimé. La valeur qu'il porte est remplacée par son successeur et c'est le nœud qui portait le successeur qui est supprimé (Ce nœud n'a pas de fils gauche).

Dans tous les cas, le noeud supprimé n'a au plus qu'un fils x, si le nœud supprimé est rouge, la propriété 5 reste vérifiée. Si le nœud supprimé est noir, il manque un noir sur une branche donc le principe de l'algorithme RBTreeDeleteFixup(T, x) est de considérer que le fils x du sommet supprimé [13] est un noeud bicolore, ce qui fait que toute branche contenant x aura une hauteur noire augmentée de 1 (cela rétablit la prop 5). x est alors de couleur noir-noir ou rouge-noir, et, par conséquent, n'est plus ni noir, ni rouge, ce qui fait que l'arbre ne vérifie plus la propriété 1. À partir de cette considération, la fonction RBTreeDeleteFixup(T, x) va déplacer cette couleur noire supplémentaire (par des rotations et recolorations) jusqu'à la racine de l'arbre pour pouvoir ensuite la supprimer sans rompre les propriétés des arbres Rouge/Noir.

13. et déplacé à sa place

Chapitre 3

Structure arborescente avancée : B-arbres

Description et algorithmes issues de : Introduction to Algorithms, Thomas H. Cormen et al. (2nd and 3rd edition) – GeeksForGeeks - Advanced Data Structures - BTree 1
GeeksForGeeks - Advanced Data Structures- BTree 2
GeeksForGeeks - Advanced Data Structures - BTree 3

I Définition

Un B-Tree [1] est un arbre de recherche équilibré, conçu pour fonctionner efficacement sur disque ou autre technologie secondaire de stockage, la lecture de nombreuses clés se fera en ne lisant qu'une page sur le disque... (exemple sur Mac le gestionnaire de fichier utilise un B-Tree). De très nombreux systèmes de gestion de base de données utilisent les B-Tree pour stocker les informations.

Un B-Tree est très similaire à un arbre Rouge/Noir mais diffère de celui-ci par son facteur de branchement. La hauteur d'un B-Tree contenant n clés est comme pour tout arbre de recherche équilibré, de l'ordre de $\log n$. Toutefois, en raison du nombre de fils que peut avoir chaque nœud, la base du logarithme peut être grande et, en pratique, la hauteur d'un B-Tree peut être considérablement moins importante que celle d'un arbre Rouge/Noir.

Définition 9. *Un B-Tree est un arbre de recherche dont la racine est notée* $root(T)$*) qui vérifie :*

1. *Chaque nœud* x *de l'arbre possède les attributs suivants :*
   - $numkeys(x)$*= nombre de clés stockées dans* x *(abrégé* $nk(x)$*)*
   - *les* $nk(x)$ *clés stockées par ordre croissant large*

     $$key(x,1) \leq key(x,2) \leq ... \leq key(x,nk(x))$$

   - $leaf(x)$*= booléen* $TRUE$ *si* x *est une feuille,* $FALSE$ *sinon.*
2. $nk(x)+1$ *références vers les fils* $child(x,1)$*,* $child(x,2)$*, ...,* $child(x,nk(x)+1)$*. Pour une feuille, ces références sont non définies.*
3. *Les clés du nœud sont t.q., si* c_i *est une clé stockée dans un des sous-arbre de* x *enraciné en* $child(x,i)$ *alors :*

   $$c_1 \leq key(x,1) \leq c_2 \leq key(x,2) \leq ... \leq key(x,nk(x)) \leq c_{n+1}$$

1. Le créateur des arbres B, Rudolf Bayer, n'a pas explicité la signification du « B ». L'explication la plus fréquente est que le B correspond à « balanced » (en français : « équilibré »).

4. *Toutes les feuilles sont à la même profondeur : la hauteur de l'arbre h.*
5. *Les nœuds de l'arbre ont un degré minimum $t \geq 2$ tel que :*
   - *Si l'arbre n'est pas vide, la racine a au moins une clé donc deux fils.*
   - *Chaque nœud autre que la racine possède au moins t fils ($t-1$ clés).*
   - *Chaque nœud peut posséder au plus $2t$ fils ($2t-1$ clés).*
   - *On dira qu'un nœud est plein s'il possède exactement $2t$ fils ($2t-1$ clés).*

Le nombre d'accès disque (en lecture ou écriture) de la plupart des opérations sur un B-Tree est proportionnel à la hauteur de l'arbre. Il est donc important de minimiser cette hauteur afin d'augmenter les performances de traitement.

Théorème 1. *Pour tout B-Tree T avec n clés, de hauteur h et de degré minimum t (≥ 2), on a : $h \in \Theta(\log_t n)$.*

Démonstration : La racine d'un B-Tree T contient au moins 1 clé et tous les autres nœuds contiennent au moins $t-1$ clés. T, de hauteur h a donc au moins 2 nœuds de profondeur 1, $2t$ nœuds de profondeur 2, $2t^2$ nœuds de profondeur 3 et ainsi de suite. A la profondeur h, il y a donc $2t^{h-1}$ nœuds. Le nombre total n de clés satisfait donc :

$$\begin{aligned} n &\geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} \\ &= 1 + 2(t-1)\left(\frac{t^h-1}{t-1}\right) \\ &= 2t^h - 1 \end{aligned}$$

On a donc $t^h \leq \frac{n+1}{2}$. En prenant le logarithme en base t des deux termes de l'égalité, nous prouvons le théorème.

De la même manière on peut calculer le nombre maximum de clés : $n \leq (2t)^{h+1} - 1$, ce qui nous permet d'écrire $h+1 \geq \frac{log_t\, n}{1+log_t\, 2} \geq \frac{log_t\, n}{2}$ puisque $t \geq 2$.

D'où,

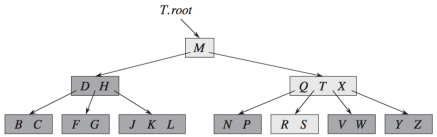
$$\frac{\log_t n}{2} - 1 \leq h \leq \log_t\left(\frac{n+1}{2}\right)$$

et donc $h \in \Theta(\log_t n)$. □

Nous avons ici une illustration de l'avantage d'un B-Tree par rapport à un arbre Rouge/Noir/ Si la hauteur de l'arbre est en $O(\log n)$, la base du logarithme pour le B-Tree est bien supérieure à 2 en pratique. Dans un B-Tree, on économisera donc un facteur $\log t$ par rapport aux arbres Rouge/Noir sur le nombre de nœuds examinés pour réaliser une opération.

Le B-Tree le plus simple correspond à la valeur $t = 2$ et est illustré dans l'exemple 16. Chaque nœud interne possède 2, 3 ou 4 fils.

Exemple 16. *Un B-Tree avec $t = 2$:*



II Opérations

Dans les applications pratique des B-Tree, les données à stocker sont de telles tailles qu'elles ne tiennent pas en mémoire centrale. Les algorithmes des opérations sur les B-Tree copient donc les pages [2] de données nécessaires depuis le disque, réalisent les opérations nécessaires et, en

2. Les informations stockées sur les disques sont divisées en pages de taille identique, chaque page étant stockée de façon linéaire sur une piste du plateau (cf. Annexe). Ainsi une fois la tête de lecture positionnée en début de page, la lecture de toutes les informations de la page se fait de façon rapide et séquentielle.

cas de changement dans les pages lues, écrivent les pages modifiées sur le disque. Le nombre de pages présentes en même temps en mémoire centrale est constant et peut être géré par un système de cache.

Dans les algorithmes, nous modélisons les opérations de lecture/écriture sur disque de la manière suivante. Si x est une référence à un objet. Si cet objet est présent en mémoire, nous pouvons accéder à ses attributs (*numkeys*, *child*, ...). Si l'objet est sur disque, il faut alors effectuer l'opération DiskRead(x) pour charger l'objet x en mémoire centrale avant de pouvoir l'utiliser. Afin de simplifier l'écriture des algorithmes, nous supposons que si l'objet x est déjà en mémoire, l'opération DiskRead(x) ne fait rien. De même, l'opération DiskWrite(x) écrit sur disque l'objet x. Ainsi, la manière de travailler avec un objet peut s'exprimer par

```
x = un pointeur vers l'objet x
DiskRead(x)
faire des opérations qui accèdent aux champs de x et/ou les modifient
DiskWrite(x) /* Omis si aucun champ de x n'a été modifié          */
faire des opérations qui accèdent aux champs de x sans les modifier
```

Nous ne présentons ici que les opérations BTreeSearch, BTreeCreate, BTreeInsert avec les conventions suivantes :

- La racine d'un B-Tree est toujours en mémoire centrale. Il n'y a donc pas besoin d'effectuer une opération DiskRead sur la racine mais toute modification de la racine sera suivie d'une opération DiskWrite.
- Tout nœud passé en paramètre d'une fonction aura auparavant été lu depuis le disque.

1 BTreeSearch : recherche dans un B-Tree $\Theta(t \log_t n)$

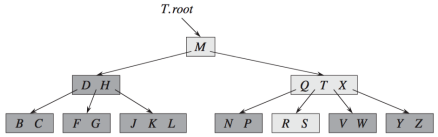
Chercher un élément (une clé) dans un B-Tree est très similaire à une recherche dans un arbre binaire de recherche mais, au lieu de faire une recherche binaire du sous arbre à examiner, il faut faire une recherche parmi n éléments. Cette recherche peut être effectuée de façon efficace en utilisant la propriété d'ordonnancement des clés mais, par clarté, nous écrivons nos algorithmes avec une recherche linéaire parmi les clés.

BTreeSearch est une généralisation de la fonction de recherche dans un arbre binaire. Cette fonction prend en entrée un nœud racine x de l'arbre à explorer et une clé c. L'appel principal, pour chercher la clé c dans l'arbre T est donc BTreeSearch($root(T), c$). Si la recherche est fructueuse, la fonction renvoie un couple (y, i) contenant le nœud y et l'indice de la clé i dans ce nœud tel que $key(y, i) = c$. Si la recherche est infructueuse l'algorithme renvoie NIL.

Function BTreeSearch(x, c)

```
i ← 1
while i ≤ numkeys(x) and key(x, i) < c do
    i ← i + 1
if i ≤ numkeys(x) and c = key(x, i) then return (x, i)
else if leaf(x) then return NIL
else
    DiskRead(child(x, i))
    return BTreeSearch(child(x, i), c)
```

Exemple 17. *La figure qui suit illustre l'opération de recherche de la clé S dans un B-Tree dont les clés sont les consonnes. Les nœuds gris clair sont les nœuds examinés lors de la recherche de la clé S.*



Complexité : Comme dans le cas de la recherche dans un arbre binaire, les nœuds examinés pendant la recherche forment un chemin simple de la racine au nœud recherché. La fonction BTreeSearch examine donc au plus h ($\in \Theta(\log_t n)$) nœuds avec h la hauteur de l'arbre et n son nombre de clés. Comme $numkeys(x) < 2t$, la recherche linéaire dans la liste des clés d'un nœud se fait au pire en $\Theta(t)$ dans chaque nœud. La complexité de la fonction BTreeSearch est donc au pire de $\Theta(t \log_t n)$.

2 Insertion BTreeInsert $\Theta(t \log_t n)$

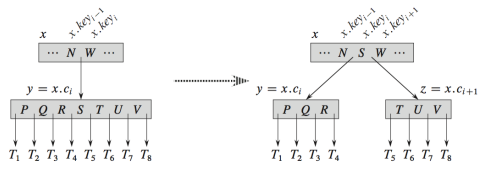
La construction d'un B-Tree (BTreeCreate) se fait d'abord en créant un arbre vide puis en y ajoutant les éléments au fur et à mesure. Ces deux fonctions reposent sur la fonction AllocateNode qui crée un nœud vide en allouant une page sur le disque en $\Theta(1)$.

Insérer (BTreeInsert) une clé dans un B-Tree est sensiblement plus complexe que l'insertion dans un arbre de recherche. Comme pour les arbres de recherche, la nouvelle clé doit être insérée sur une feuille de l'arbre. Toutefois, on ne peut pas, dans le cas des B-Tree, créer un nouveau nœud feuille et l'ajouter à l'arbre puisque cela augmenterait la hauteur de la branche correspondante et l'arbre ne vérifierait plus les propriétés des B-Tree. Il faut donc ajouter la clé dans une feuille existante afin de ne pas augmenter la hauteur de la branche d'insertion.

Pour cela, comme on ne peut pas insérer de clé dans un nœud feuille qui est plein, on introduit la fonction BTreeSplitChild qui va diviser un nœud feuille y plein (qui possède donc $2t-1$ clés) par sa clé médiane $key(y,t)$ en deux nœuds possédant chacun $t-1$ clés. La clé médiane est ajoutée dans le parent de y pour identifier le point de branchement des deux nouveaux sous-arbres. Si le nœud parent de y est plein aussi, il est découpé et la clé de découpe progresse vers la racine.

Afin d'éviter à avoir à parcourir l'arbre 2 fois (une en descendant, l'autre en remontant), lors de la recherche de la feuille d'insertion, chaque fois qu'un noeud plein est rencontré, il est découpé en sa médiane. Ainsi, en faisant cela en descendant dans l'arbre, si un noeud doit être coupé, on est sûr que son parent n'est pas plein.

Exemple 18. *Subdivision d'un nœud plein en sa médiane. Ici, $t = 4$ et le nœud y est coupé en deux nœuds y et z. La clé médiane, S du nœud est ajoutée à son parent.*



Cette fonction effectue simplement un "couper-coller" des informations des nœuds. En raison des boucles de recopie des données dans le nœud z, cette fonction à une complexité temporelle de $\Theta(t)$ et effectue $\Theta(1)$ opération sur le disque.

Dans la fonction d'insertion, la sélection gère le cas où la racine de l'arbre dans lequel on veut insérer le nœud est pleine ou pas. Si elle est pleine, une nouvelle racine est crée. Ensuite, la fonction BTreeInsertNonFull ajoute la clé dans le sous arbre, sachant qu'il n'est pas plein.

Complexité : Pour un B-Tree de hauteur h, la fonction d'insertion effectue au pire $\Theta(h)$ accès disque puisque 1 seul DiskRead ou DiskWrite est effectué avant l'appel récursif à BTreeInsertNonFull. La complexité temporelle totale au pire est de $\Theta(t.h) = \Theta(t \log_t(n))$. De plus, comme la fonction BTreeInsertNonFull est récursive terminale, elle peut être implantée de façon itérative avec une seule boucle while et il est alors facile de montrer que le nombre de pages devant résider en mémoire est au pire $\Theta(1)$.

3 Suppression d'une clé dans un B-Tree $\Theta(t \log_t n)$

La suppression d'une clé dans un B-Tree est analogue (symétrique) à l'opération d'insertion mais est légèrement plus complexe. En effet, une clé peut être supprimée de n'importe quel nœud de l'arbre et non pas d'une feuille uniquement et, en cas de suppression d'une clé dans un nœud interne, il peut être nécessaire de réorganiser les fils de ce nœud.

Lors de la suppression d'une clé, il faut conserver les propriétés des B-Tree : s'assurer qu'un nœud ne devienne pas trop petit (seule la racine peut posséder moins de $t-1$ clés). De plus, afin d'éviter d'avoir à parcourir 2 fois l'arbre pour supprimer la clé/rétablir les propriétés, un algorithme en une seule passe peut être défini comme dans le cas de la fonction d'insertion.

La fonction récursive BTreeDelete supprime la clé c d'un B-Arbre T enraciné en x en prenant soin, lors de l'appel récursif sur un nœud x que le nombre de clé de x est au moins égal au degré minimum t de l'arbre.

BTreeDelete :

1. Si la clé c est dans le nœud x et que x est une feuille, supprimer la clé de la feuille.
2. Si la clé c est dans le nœud x et que x est un nœud interne
   (a) Si le fils y qui précède (respectivement suit) c dans le nœud x possède au moins t clés, trouver le prédécesseur (respectivement successeur) c' de c dans le sous-arbre enraciné en y. Supprimer récursivement la clé c' dans y et remplacer c par c' dans x. La recherche et suppression du prédécesseur (respectivement successeur) peut se faire en une seule passe sur l'arbre.
   (b) Si le fils y qui précède x et le fils z qui suit x ont tous deux exactement $t-1$ clé, fusionner c et les clés de z dans y de telle sorte que x perde à la fois la clé c et la référence au fils z. y contient maintenant $2t-1$ clés. le nœud z peut être détruit et la suppression de c continue récursivement sur y
3. Si la clé c n'est pas présente dans le nœud interne x, déterminer la racine $child(x,i)$ du sous-arbre devant contenir la clé. Si $child(x,i)$ ne possède que $t-1$ clés, exécuter une des deux sous étapes suivantes pour garantir que l'on puisse descendre dans un nœud possédant au moins t clés. Continuer la suppression récursive sur le bon fils de x.
   (a) Si $child(x,i)$ possède exactement $t-1$ clé mais à un frère immédiat possédant au moins t clés, ajouter une clé à $child(x,i)$ en descendant une clé de x dans $child(x,i)$, en montant une clé du frère immédiat gauche ou droite de $child(x,i)$ dans le nœud x et en déplaçant le fils concerné du frère dans $child(x,i)$.
   (b) Si $child(x,i)$ et ses deux frères immédiats ont $t-1$ clés, fusionner $child(x,i)$ avec un de ses frères, ce qui nécessite de faire descendre un clé de x en tant que clé médiane dans $child(x,i)$