Chats IFT2015

Séance 01

I am not sure I understood the lnN for binary trees, can you please elaborate?

I didn't write "ln", I wrote "lg". The first means log to the base e, the second means log to the base 2. In CS we almost always use lg. We will spend a lot of time figuring out how to keep trees BALANCED, which means that their depth is only about lgN. That means that if you have a million keys, since 2^{20} is about a million, replacing N by lgN changes the cost from a million comparisons to about 20. Worth the trouble.

Pourquoi je mets tellement l'accent sur les TAD?

C'est parce que, au début de l'info, l'étude dans ce domaine était très ad hoc. Telle ou telle méthode est bonne pour ceci, une autre est meilleure pour autre chose. Le TAD a permis de mettre un peu d'ordre là-dedans.

Comment exprimer ce que nous avons remarqué en utilisant la terminologie du TAD?

La meilleure méthode peut évidemment changer si le problème change.

C'est quoi qui définit le problème?

C'est le TAD qui définit le problème!

Comment le TAD a changé entre les deux cas que nous avons regardé?

L'ordre était important pour le premier TAD.

Bon, c'est quoi un TAD?

Ce sont des données, avec un certain nombre d'opérations à effectuer.

Ici quand on parle de méthode, c'est seulement de structures ?

Oui, la méthode c'est une structure de données.

Séance 02

Dans le contexte de Hashing, let mot "retrait" est utilisé de deux façons différentes. Aussi, dans notre étude, le mot "liste" est utilisé de 17 façons différentes, parfois pour indiquer une méthode, parfois pour indiquer un problème! Ça vaut mieux de remarquer cela!

La liste TAD c'est différent des listes généralisées?

Un TDA n'est pas une liste. Ce sont des données avec un des opérations spécifiées.

Est-ce que la méthode d'implantation d'un TAD peut en influencer l'efficacité de telle sorte que l'ordre d'un type d'opération puisse changer, ou est-ce que toutes les méthodes d'implantation se comportent environ de la même manière?

Définitivement la différence peut être énorme. Regardez l'exemple #1 qui s'en vient.

L'exemple 2 est-il censé représenter un TAD ou bien un moyen d'organiser des TAD?

L'exemple 2 est une description générale d'une méthode pour résoudre le problème défini par le TAD Table.

Pouvez-vous parler un peu sur la liste ré-entrant, svp ?

Supposons que vous modéliser l'organisation "U de M". Les étudiants de deuxième année en INFO pourraient être dans une sous-liste très grande, avec beaucoup d'informations reliés. Mais cette sous-liste pourrait être intéressante pour le registraire, et aussi pour le directeur du DIRO. Plutôt que d'emmagasiner la sous-liste deux fois, permettons à différents nœuds dans la liste de partager la sous-liste. Moins d'espace utilisé. Mais il faut faire attention: si le directeur décide qu'il n'a plus besoin de la sous-liste, il faut la couper mais le registraire n'aura plus accès! Pour résoudre ce problème, on ajoute par exemple un compteur qui indique combien de nœuds accède à la sous-liste. On ne supprime pas avant que le compteur descende à zéro. Si le directeur n'a plus besoin de la sous-liste, il coupe simplement lien vers la sous-liste. Mais le système ne doit pas éliminer la sous-liste s'il y a d'autre nœuds qui en ont besoin, le registraire par exemple.

On ajoute ce compteur à chaque nœud?

Non, un compteur à la sous-liste.

Donc on déconnecte les connections jusqu'à avoir zéro au compteur?

Oui, c'est ça.

Ok alors si quelqu'un ne veut plus la sous liste, il coupe juste la référence à la sous-liste?

Exactement!

Donc liste de données abstraites peut être aussi le problème qu'on nous pose?

Les choses sont plus simples que vous pensez. Il n'y a rien de compliqué ici! Pour décider quelle est la meilleure méthode pour résoudre tel ou tel problème, il faut préciser soigneusement le problème. Alors, je peux dire vaguement " veux insérer et retirer des items dans une liste de comptes de banque. Quelle est la meilleure méthode?". Mais m'exprimer comme cela est trop vague si je veux faire les choses soigneusement. Qu'est-ce que je veux faire exactement? Dites-moi cela et je vais commencer à comparer les méthodes (les structures de données) soigneusement! À moins de me définir EXACTEMENT le problème, je ne peux pas vous dire PRÉCISÉMENT quelle est la meilleure méthode. Alors, pour définir le problème, j'introduis le TAD Table. Nous allons voir que ce TAD faire référence à des données et plusieurs opérations, avec la fréquence de chaque opération précisée.

Quand on a fait le problème des carrefours la méthode matrice se fait en $O(N^2)$ et la méthode liste chainée en $O(N)$?

Le coût, mémoire ou temps de calcul, est de l'ordre de $O(N^2)$ avec la matrice. De l'ordre de $O(N)$ avec la liste chaînée.

Pour la méthode tableau, au pire des cas, on doit lire N lignes N fois, donc $O(N^2)$, non?

Oui, pour un seul carrefour, le coût, que cela soit en temps de calcul ou en mémoire utilisée, serait de l'ordre $O(1)$ pour la liste chaînée, et de l'ordre $O(N)$ pour le tableau. Mais je dois faire cela pour N carrefours, ce qui veut dire la différence entre N et N^2.

Il est possible d'expliquer plus 2^{K-M} ?

Recherche dichotomique: J'ai fait la comparaison entre l'arbre binaire et une liste non-triée. J'ai constaté que certaines opérations sont de l'ordre $O(lgN)$, plutôt que $O(N)$. Ça rend l'idée d'arbre de recherche intéressante. Ok, fin de la comparaison. Mais vous pourriez vous dire, mais qu'est-ce qui se passe si le tableau est trié? Dans le cas d'un tableau trié, la recherche devient $O(lgN)$. Mais je vous signale qu'il y a d'autres opérations (retrait) qui sont de l'ordre $O(N)$, l'idée d'utiliser l'arbre binaire reste intéressante. Pour votre question même sur la recherche dichotomique: Nous commençons avec un tableau avec $N = 2^k$ éléments. Après avoir fait 1 comparaison, nous savons que l'élément cherché est en haut ou en bas, soit un tableau de longueur 2^{k-1}. Après 2 comparaisons, l'élément se trouve dans un tableau de longueur 2^{k-2}. Après m comparaisons, dans un tableau de longueur 2^{k-m}. Le processus va terminer quand vous avez fait $m = k$ comparaisons, parce que là vous aurez une liste de longueur 1. Mais $k = lgN$, donc ça prend $m = lgN$ étapes.

Assumant que l'on fait les 2^{k-k} comparaisons, ça ne semble pas très rapide comme algorithme non?

Oui, lgN est très bien par rapport à N. Par contre, si vous faîtes la comparaison entre arbre binaire et tableau trié, il faut trier. Cela coute $NlgN$. Aussi: Pour retirer un élément, ça coute N pour le tableau trié, mais nous allons voir que le retrait dans l'arbre binaire est aussi lgN.

Donc $NlgN$ pour trier + lgN pour chercher dans le tableau?

Oui, alors tout dépend combien de retraits faut-il faire, combien de recherches, combien d'insertions… Le TAD Table va lister ces opérations, et si vous avez une idée de la fréquence de chaque opération, vous pouvez décider quelle est la meilleure méthode. Les arbres binaires sont $logN$ pour la recherche, lgN pour le retrait, lgN pour l'insertion. Il n'y a rien qui coûte de l'ordre de N. C'est très bien.

C'est quoi la notation "cf" qui est utilisée après l'exemple 1 dans "3 k n (cf N^2)" dans le vidéo S02S2?

https://en.wikipedia.org/wiki/Cf.#:~:text=The%20abbreviation%20cf.,with%20the%20topic%20being%20discussed

Si on connaît d'avance notre N, et qu'il reste fixe, on peut par contre choisir une structure de type 2^N, en sachant que notre N reste très petit.

Supposons que je propose une méthode 2^N. Si je sais a priori que N sera toujours petit, oui, je peux me servir de l'algo. C'est pour cela que quand nous faisons des analyses nous ignorons ce qui se passe pour N petit. ... Si $N = 10$, la différence entre N et lgN n'est pas grande, utilisez l'algo qui vous plaît. Ça m'est égale, ça vous est égal. Mais si jamais il y a la possibilité que N va être grand, vous êtes mieux de faire la distinction entre N et N^2, ou N et lgN, etc.

Pourquoi est-ce qu'on ne choisirait pas l'arbre binaire?

Souvent l'arbre binaire est effectivement un bon choix. Mais pensez au guichet automatique. Là vous avez besoin de répondre très vite à la question: combien dans mon compte? Et il arrive que la méthode de Hashing puisse répondre à cette question dans un temps de l'ordre constant! Tout dépend de la fréquence des opérations spécifiées dans le TAD.

Alors, on doit considérer le coût pour l'opération la plus fréquente ou la plus importante, c'est ça ?

Dans la pratique les choses sont un peu moins claires. Mais vous vous dites, *hmmm*, j'ai besoin de faire des consultations vite, mais retirer un enregistrement peut se faire lentement, cela correspond à fermer le compte du client. Je vais donc penser à Hashing.

Séance 03

Maintenant, donnez-moi la preuve que T_{avg} est supérieur ou égal à T_{best}.

$T_{avg} = \frac{1}{n} * \Sigma\, T_i$, qui est $\geq \frac{1}{n} * \Sigma\, (\min T_i)$. Mais cela est égal à $\frac{1}{n} * \Sigma\, T_{best}$, ce qui est exactement égal à T_{best}.

Pour quelle raison prenons-nous souvent Quicksort en pratique malgré que l'analyse asymptotique semble meilleure pour Heapsort?

Il est vrai que l'analyse asymptotique indique que Heapsort serait meilleur. Mais dans la pratique, à cause des constantes cachées dans les analyses asymptotiques, il arrive que Quicksort est typiquement plus vite. Aussi, les cas extrêmes arrivent peu souvent.

Is it possible that we have lower bound of $T_{worst} = N$ and upper bound of $T_{avg} = N\ lgN$? Which means is it possible that the lower bound of T_{worst} is less than the upper bound of T_{avg}?

En général, il est possible d'avoir des chevauchements entre les gammes borne-inf, borne-sup, mais bien sûr, il y a des cas impossibles. Il suffirait de prendre simplement $T_{worst} = N$ et $T_{avg} = N\ lgN$.

Séance 04

Pour le Quicksort, je n'ai pas bien compris comment on peut utiliser un stack au lieu de deux pour l'implantation.

Nous avons deux choses à faire: trier la sous-liste gauche, et trier la sous-liste droite. On peut s'occuper de l'une ou l'autre, mais pas les deux en même temps. Alors on met une sous-liste sur la pile, et on s'occupe de l'autre. Maintenant, quelle sous-liste mettez-vous sur la pile? Combien d'éléments peuvent se trouver dans la pile, maximum? Peut-elle devenir très grande en terme nombre d'éléments? Ce qui m'inquiète est la longueur de la pile même. Oui, dans le plus mauvais cas ça pourrait être $n-1$. Mais si je mets toujours la sous-liste la plus longue sur la pile, le nombre maximum d'éléments dans la pile serait de l'ordre lgN. Mais la réponse la plus importante est la suivante: Quand $front = back$ dans la queue, elle peut être vide ou complète.

C'est très dommage que j'ai dû couper juste ici. Vous pourriez aller déjà regarder le premier segment de la Séance 5 tout de suite, je veux dire avant le prochain cours ...

L'élément à la racine doit avoir une clef qui est plus petite que la clef de ses deux enfants. Alors si la clef à la racine est 3, la clef à gauche pourrait être 5 et la clef à droite pourrait être 13.

J'ai de la difficulté à différencier un arbre binaire "complete" d'un arbre binaire "full".

Un arbre FULL est un arbre binaire où chaque nœud a 0 enfant ou 2 enfants. Jamais 1 enfant. Alors ce n'est pas la même chose qu'un arbre COMPLETE. Toutes les combinaisons sont possibles. Un arbre avec racine et deux enfants est FULL et COMPLETE.

Est-ce que quand on utilise le HEAP pour représenter un arbre binaire, alors on a un problème lorsqu'on arrive à l'élément maximum? On doit recopier tout le tableau dans un tableau plus gros? Ou est-ce qu'on utilise quand même un tableau doublement chaîné?

Dans les cours subséquents, je vais parler de la différence entre FULL et COMPLETE. On va pouvoir insérer des éléments dans le HEAP. Le HEAP est une structure dans un table contiguë qui utilise une certaine méthode pour emmagasiner les enfants et aussi qui respecte la condition que la clef de chaque nœud est plus petite que la clef de ses enfants. Cela va nous permettre de trouver l'élément prioritaire dans un temps lgN, mais aussi de pouvoir INSÉRER des nouveaux éléments dans la queue dans un temps lgN, et aussi de pouvoir RETIRER des éléments de la queue dans un temps lgN. Voyez-vous, si vous avez des éléments dans les cases $1, 2, 3$ et vous décidez d'insérer un nouvel élément alors vous devez mettre l'élément dans la case 4. C'est la contrainte de l'arbre COMPLETE. Mais aussi: IMPORTANT! Vous devez mettre le nouvel élément dans la case 4, et vous devez garantir que la condition sur les clefs reste respectée. Nous ne sommes pas en train de trier des éléments. Nous construisons une queue de priorité. Quel est notre but? Pas de trier des éléments, mais plutôt de pouvoir savoir quel est l'élément prioritaire à tel ou tel moment. Vous pourriez avoir la clef 2 dans la case 1 à la racine et 5 comme enfant gauche, et 13 comme enfant droite, l'élément prioritaire est à la racine. On trouve toujours l'élément prioritaire à la racine. Alors on enlève la racine, avec clef 2, et on fait avec l'élément prioritaire ce qu'on voulait. Mais après avoir retiré l'élément prioritaire 2, il faut combler le trou que vous avez fait dans la structure: la première case est maintenant vide! Le nouvel élément prioritaire sera 5 dans mon exemple, mais je dois restructurer la chose. Il arrive que je puisse faire cette restructuration dans un temps lgN. Alors je peux continuer de prendre les éléments prioritaires, l'un après l'autres, avec un coût de restructuration de lgN chaque fois. Est-ce que vous allez trier votre liste doublement chaînée, ou pas? Peu importe votre réponse, il y aura des opérations de l'ordre de N.

J'ai de la difficulté à comprendre pourquoi, si le but est simplement d'obtenir une stack (et donc d'avoir le dernier élément prioritaire) est-ce qu'on ne le mettrait pas tout simplement dans une liste doublement chaînée avec un pointeur sur le dernier élément?

Distinguez entre le problème est la méthode. Notre but n'est pas de trier une liste. C'est de pouvoir savoir l'élément prioritaire à chaque étape. Ça c'est le PROBLÈME. Mais vous pouvez utiliser comme MÉTHODE (structure de données) une liste chainée triée ou une liste chaînée non-triée. Ça c'est la MÉTHODE. Peu importe votre choix de MÉTHODE, il y aura des opérations d'ordre N.

Mais quand on bâtit la liste, on est forcé de mettre les éléments le plus en bas à gauche possible?

Non, je vais vous montrer où mettre les éléments. L'algorithme pour mettre des nouveaux éléments est vraiment astucieux. Cela vous laisse avec un arbre COMPLETE qui satisfait la condition sur les clefs, et aussi vous pouvez garantir de faire cela avec lgN opérations à chaque étape! Il faut N opérations pour établir la structure au début, qui respecte la condition sur les clefs (algorithmique de Floyd). Mais après cela, toutes les opérations sont lgN. *C'est la clé (du succès) : réussir à faire des opérations de l'ordre lgN à chaque étape?* C'est ça, mais mieux encore : même si pour certaines opérations c'est lgN dans le plus mauvais cas, ça sera seulement (1) dans le cas moyen. C'est vraiment satisfaisant.

Séance 05

Des questions? Sur les queues de priorité par exemple?

La preuve de l'algorithme de Floyd $O(N)$ est intéressante. L'idée est utilisée ailleurs. Faut regarder l'exemple de Weiss, p. 239 $O(N \lg (k0))$. Intéressant aussi.

Je ne comprends pas les concepts de feuilles et nœuds internes. Qu'est-ce qu'ils sont vraiment ? Les nœuds internes sont-ils les nœuds avec enfant(s) ?

Exactement. Les nœuds internes ont des enfants. Sinon, le nœud est une feuille. Les feuilles n'ont pas d'enfants.

Avez-vous compris la preuve $O(N)$ de l'algorithme de Floyd?

Vous commencez par regardez l'énoncé de l'algorithme. Vous devez faire quelque chose pour chaque nœud interne. Vous devez balayer les nœuds internes de droite à gauche, et faire quelque chose chaque fois. Avec le dernier petit résultat, vous savez exactement ce que cela veut dire. Vous savez où se trouve le dernier nœud interne, et vous scannez à gauche pour trouver les autres. Pour trouver le coût total, il faut se demander combien de nœuds internes. On a remarqué qu'il y en a 2^i au niveau i. On a aussi remarqué que, pour chaque nœud interne, il faut descendre $h - i$ étapes. Cela nous permet d'écrire la somme totale d'étapes. Après cela, il ne reste qu'à trouver une formule simple pour la somme. L'astuce de diviser cela en séries géométriques, horizontales et verticales sur la page, nous a permis de donner un estimé simple.

Séance 06

Quand vous parlez d'arbre cousu avec des ficelles, est-ce que c'est semblable à une liste chaînée?

Un peu, mais ce sont des pointeurs supplémentaires.

Pourquoi j'insiste tellement sur l'idée de liens entre des choses qui semblent différentes?

Par exemple, la liste pure et l'arbre qui correspond (dans le contexte de la rep "puînée"). C'est parce que cela peut vous être utile! Chaque fois que vous inventez une astuce pour parcourir une liste pure, par exemple, cela vous donne une astuce pour parcourir des arbres. C'est le principe général qui est important. C'est le principe derrière qui est toujours important.

Can you clarify the statement you made at the end of the second recording about how Weiss names chapter 4.3?

For me it is important to think clearly about things and mixing the problem with the method is not thinking clearly! The problem I want to solve is "how do I implement the operations in TAD Table?" One of the METHODS to solve this problem is to use a Binary Search Tree. Another is to use Hashing. Or 1-2-3 Deterministic Skip Lists. These are METHODS. Liste chaînée est une MÉTHODE. J'ai le TAD *Liste*. Un certain nombre d'opérations que je dois implanter.

But Weiss does not seem to refer to any ADT called "Table", or am I missing it?

You are right, Weiss introduces ADTs, that's good, but he doesn't use the language very consistently, that's bad.

Alors un Binary Search Tree n'est pas un TAD ?

Un BST n'est pas un TAD, c'est une méthode pour résoudre TAD Table. J'ai le TAD *Liste*, un certain nombre d'opérations que je dois réaliser. Comment faire? Quelles sont les méthodes possibles? Une méthode serait le tableau contigu. Une autre serait la liste chaînée.

Pourriez-vous différencier TAD Table et méthode Table svp?

Il n'y a pas de méthode "Table". TAD Table est le problème, les opérations à réaliser. Je peux parler de "tableau" comme méthode. C'est une collection de cellules continues, avec accès direct par indexage. Mais cela n'a rien à voir avec TAD Table. TAD Table est un exemple de TAD. Je vous ai donné en haut d'une diapo des exemples de problème: TAD Liste, etc.

Mais le tableau contient des méthodes?

Un tableau contigu EST une méthode, c'est une méthode pour réaliser TAD Liste.

Je ne me rappelle plus de la différence entre Full ou Complete. Je croyais que c'était simplement la traduction.

Il faut donc revoir les présentations, où je dis que je ne voulais pas traduire "Full" et "Complete". Un arbre Complete peut ne pas être Full : ce n'est pas obligatoire d'avoir toujours 0 ou 2 enfants, on pourrait avoir des nœuds avec seulement un enfant, donc il n'est pas Full. On pourrait avoir des arbres Full, ajoutant des enfants à droite, pas à gauche, donc ce ne serait pas Complete.

Et si l'arbre a une seule feuille c'est Complete?

La réponse est "peut-être"! Si vous avez la racine avec un enfant à gauche, c'est Complete. Si vous avez la racine avec enfant à droite, ce n'est pas Complete.

Est-ce que c'est possible de transformer un arbre full pour faire en sorte qu'il soit Complete?

Non, ce sont des propriétés d'arbres fixes. Si chaque nœud a 0 ou 2 enfants, c'est Full. Si tous les nœuds sont présents, sauf peut-être dans la dernière rangée, c'est Complete.

Quelle est la différence claire entre un TAD et une méthode ? Je confonds beaucoup les deux (par exemple entre le TAD table et la Linked List)

TAD est le problème. Ce sont les opérations qui doivent être réalisées. Quelle structure utiliser pour faire cela? Quelle méthode?

Donc le TAD n'est définit que par une liste d'opérations qu'on cherche à effectuer ?

Oui.

J'insiste toujours sur les idées derrières. Pourquoi?

La preuve de la méthode de Floyd par exemple. L'idée n'est pas seulement de savoir comment fonctionne Floyd. Nous voulons comprendre aussi pourquoi ça fonctionne. Je trouve la preuve intéressante, mais pas seulement à cause de l'astuce de sommations horizontales et verticales. Pensez à la forme générale d'un arbre parfaitement balancé, j'ai toujours dessiné un triangle. Qu'est-ce que vous voyez? Une collection de nœuds. En rangées. Mais tiens, combien de nœuds dans la dernière rangée? 1 dans la première rangée, 2 dans la deuxième , 4 dans la troisième… Quelle fraction de nœuds est dans la dernière rangée? La moitié de nœuds, point sur la ligne. Et ensuite, il y a un autre quart dans l'avant dernière ligne.

Mais en quoi ce résultat est utile ?

Une chose, si je me trouve dans un contexte qu'il n'y a rien à faire dans la dernière ligne, dans la méthode de Floyd, par exemple, et si en plus il y a peu à faire dans l'avant-dernière ligne, comme dans l'algorithme de Floyd… Les choses risquent d'être moins chères que j'avais prévu.

Séance 07

Pourquoi N+1 nœuds d'échec? Dans quel arbre faut-il travailler pour démontrer cela?

Parce que les feuilles est $n + 1$.

Dans quel arbre travaillez-vous?

Arbre Complete.

Il y a l'arbre de base. Mais il y a un autre arbre, théorique, dans lequel on travaille ici. Si vous concevez l'arbre avec les nœuds d'échec ajoutés comme un nouvel arbre, un arbre étendu, combien de nœuds a cet arbre?

C'est l'arbre qui a comme nœuds internes l'arbre de base et comme feuilles les nœuds d'échec qui a $N + N + 1$ nœuds. Le nombre de nœuds ajoutés est égal au nombre de feuilles de l'arbre original. L'arbre étendu a évidemment 2N+1 nœuds, ça vient de notre petit théorème. Au total, $2N + 1$ nœuds.

Y a-t-il une différence entre un nœud d'échec et un pointeur nul?

Le pointeur nul est dans l'implantation. Le nœud d'échec est la chose correspondante théorique.

J'ai dit dans la présentation que le nœud prédécesseur doit avoir un pointeur nul à droite. C'est clair, mais quelle est la preuve?

S'il y a un nœud à droite, il existe une valeur plus grande que le prédécesseur et inférieure au nœud initial.

Dans l'esquisse de preuve, j'ai appliqué la règle d'Hôpital à la fonction $\lg N$, $H(N)$ est défini seulement pour les entiers.

Il est facile de montrer que $H(N)$ est très proche de lgN dans le cas spécial quand $N = 2^k$. C'est motivant, rassurant. La preuve générale est plus difficile.

Soulignons ce que nous allons faire dans la suite.

Nous allons prendre un arbre "aléatoire" pour commencer et nous allons montrer que la profondeur en moyenne est lgN. Était-il évident a priori qu'il n'y a pas tellement d'arbres "méchants" pour donner très souvent des profondeurs très grandes? Pour moi non. Cela veut dire que les arbres "méchants" sont rares. Mais on aimerait limiter la profondeur dans le plus mauvais cas aussi.

Wouldn't we assume some kind of "average" tree depth is most likely?

What is the sample space here? What do we MEAN by an average tree? Nous allons prendre comme définition d'un arbre aléatoire quelque chose qui se base sur l'ordre d'insertion.

For any given tree structure, that with a random data set, the depth will tend to some average, rather than something "méchant".

Yes. We are going to take an average over all possible orders of insertion in the tree. After that, as you say, the depth will have some sort of average. Et nous allons voir que la valeur qui sort de l'analyse est $O(lgN)$.

Man Ping Li, j'ai admiré vos efforts de penser à "comment faire la preuve?"

Nous ferons quelque chose de différent, mais c'est très bien d'essayer vous-même. Il y a E, qui est comme I, mais dans l'arbre avec les nœuds d'échec. Pourquoi intéressant? Parce que la Recherche dans le cas d'échec nous intéresse beaucoup. Quel est le coût moyen de recherche dans le cas de recherches échouées (Insertion)? Pour le calcul de E, vous faites la même chose, mais en utilisant seulement les nœuds d'échec. Commençons avec un arbre avec 1 nœud. Il y a 2 nœuds d'échec. La valeur de I est 0. La valeur de E est $1 + 1 = 2$.

Prenons un arbre avec la racine, et 1 enfant à gauche. Quelle est la valeur de E?

5. On a 2 nœuds d'échec à gauche, plus 1 à droite. Les 2 à gauche sont de profondeur 2. Et celui à droite de profondeur 1.

Maintenant, une racine avec 2 enfants. Quelle est la valeur de I, et quelle est la valeur de E?

2 pour I, et 8 pour E.

Comment calculer E, étant donné I et N, sans passer par la définition détaillée? Regardez les trois petits exemples.

$E = I + 2N$.

Quelle serait l'approche générale pour faire la preuve?

On pourrait procéder par induction. Comment procéder? Je connais deux façons, parce que j'ai posé la question plusieurs fois aux étudiants dans le passé, et comme chaque année, il y a des étudiants brillants. Supposons que nous ayons un arbre avec N nœuds, et une certaine valeur de I. Et que nous avons $E = I + 2N$. Quelle est la conséquence pour I d'enlever 1 nœud. Disons que I est réduit de d…

Séance 08

À 3:00 sur la vidéo, je fais référence à la constante. Il est quand même possible de dire quelque chose à ce sujet. Pensez à cela!

Il est mieux de séparer la partie "aléatoire" en deux parties. On commence par choisir la FORME de l'arbre de façon aléatoire, et après on fait la moyenne sur les NOEUDS dans l'arbre, en divisant I par N. On y reviendra.

Mais si ce n'est pas le cas dans un arbre méchant, pourquoi est-ce que ce serait vrai pour tous les autres types d'arbres ?

Si quoi n'est pas le cas?

Que la profondeur soit estimable à lgN. Quand vous parliez de constante dans le vidéo, c'était pour faire l'estimation de la profondeur non?

Nous allons commencer par faire un choix d'arbre aléatoire, et après nous allons faire une pondération sur tous les nœuds de l'arbre, en divisant I par N. Il sera quand même possible d'étudier la constante…

Donc la constante est pour définir la valeur de $\frac{I}{N}$ et non la profondeur ?

Vous voyez l'idée de diviser I par N? C'est pourquoi nous avons introduit I. La constante est pour le coût moyen.

La référence au Devoir 2 est de E20. Par contre, dans le Devoir 2 vous allez regarder $D(1) = 0$.

Je trouve que ma présentation est incomplète à ce sujet. Je fais mon autocritique au début du Devoir 2…

À 1:12 de la vidéo, je dis que la constante ne m'intéresse pas plus que cela. Cette année cela m'intéresse, si vous me comprenez.

Quelle était la question que nous avons laissé ouverte du premier chat? Nous commençons ici par regarder la valeur espérée de I. Mais pourquoi nous nous intéressons à I?

C'est parce que, dans une deuxième étape, nous allons vouloir faire la moyenne sur tous les nœuds. Si je vous donne un arbre quelconque, et je vous demande de faire la pondération du nombre de comparaisons sur chaque nœud dans l'arbre.

Qu'est-ce que vous faites?

C'est la longueur I.

Mais divisée par N, n'est-ce pas?

Pour avoir la moyenne oui.

Aussi, il y a un autre petit oubli dans votre suggestion, quelque chose qui est mentionné soit dans le Devoir 2, soit à l'Intra, soit dans les deux.

Alors, c'est la moyenne de comparaissons à faire?

Oui. C'est le nombre de comparaison qu'on peut s'attendre à faire en moyenne

Pourquoi on dit que $d(node\ sans\ enfant) = -1$?

Le choix de -1 pour les arbres vides est un peu bizarre, je suis d'accord. Mais c'est ce qu'il faut pour pouvoir toujours dire la même chose, que la DIFFÉRENCE entre gauche et droite ne peut pas dépasser 1 en valeur absolue.

Dans les chats précédents, vous avez parlé de la constante. C'est bien la constante qui relie le temps de recherche moyenne pour un arbre de recherche binaire à "I"?

Pour commencer, nous sommes dans un nouveau théorème maintenant. Quand je parlais de constante, c'était dans le cas de l'arbre moyen, critère "coût moyen". Nous parlons maintenant d'une nouvelle méthode: AVL. Pour le cas moyen, on n'a rien ajouté. On sait que les méthodes AVL et autres s'en viennent, mais pour établir une base de comparaison, on a commencé avec l'idée de ne rien ajouter à l'arbre. Ce que nous avons constaté est que le comportement n'est pas si pire! On n'ajoute rien, et pourtant nous avons un comportement lgN en moyenne. Surprenant en fait. Si nous ajoutons les astuces comme celles de AVL, nous avons un comportement lgN même dans le plus mauvais cas. Et la longueur maximale maintenant est seulement 1.441 fois plus longue que l'optimum lgN.

Un arbre moyen est un BST sans particularité?

Oui.

Pourquoi -1 , je veux revenir à cela. Ce qu'il faut pour AVL est que la différence ne dépasse jamais 1 en valeur absolue.

Pour ne pas être obligé à dire quelque chose de spécial dans le cas un peu spécial du sous-arbre vide on introduit la définition que $d(\Lambda) = -1$. Cela nous permet d'exprimer ce qu'on veut en utilisant la même phrase "différence ne dépasse 1 en valeur absolue". Dans d'autres contextes, cela va être un peu différent. Voir par exemple la première question du Devoir 2, où vous avez quelque chose de plus "naturel". Mais attention, "balancé" veut dire "presque balancé". Et après la découverte de AV et L il y avait d'autres façons de trouver des arbres "balancés". La première question du devoir n'est pas compliquée, je vous signale que je vous demande d'ajouter un petit quelque chose que j'ai omis au début de la preuve. La question est tellement simple que je ne peux rien dire sans vous donner la réponse. Vous allez voir.

Pour avoir que la balance est toujours $B(H) = H(T_L) - H(T_R)$ c'est bien ça non?

Oui, sauf que j'ai noté par d plutôt que H.

C'est quoi le gain pour utiliser AVL? Je ne vois pas grande amélioration.

C'est le fait que c'est vrai dans le plus mauvais cas! Vous n'aurez pas peur que, au fur et à mesure d'ajouter et retirer des clefs que l'arbre va devenir une liste linéaire, ou presque! Vous pouvez compter sur une réponse dans un temps qui ne dépasse pas 50% plus cher que le maximum. Supposons que par hasard les clefs arrivent dans l'ordre $1, 2, 3, 4, 5, \ldots$ Dans un arbre quelconque, cela va être une liste linéaire, non? Toutes les clefs sont ajoutées à droite chaque fois. Cela donne des recherches $O(N)$. Mais avec AVL, nous allons rebalancer avec chaque insertion, et cela va rester $O(lgN)$. Nous rebalançons l'arbre après chaque insertion, retrait, et ça reste balancé, jamais plus profond que $1{,}441\ lgN$.

$Coût\ de\ rabalancement = O(N)$ non?

C'est $O(N)$ si on n'ajoute pas d'astuces comme AVL, et $O(lgN)$ avec AVL.

Séance 09

L'idée de minimiser N avec d fixe, plutôt que de maximiser d avec N fixe, devient claire si vous regardez la vidéo entre 40 et 70 seconds. $1/2$ est remplacé par 2, cela montre l'idée intuitive.

Ce que je dis à 6:15 est correct, mais j'indique la mauvaise chose avec la souris! Assurez-vous de comprendre mon erreur: ce n'est pas le fait qu'il y a deux arbres dans l'ensemble, c'est le fait que c'est deux arbres ont deux nœuds!

Enfin, 14:42. La hauteur de T_G doit être $d-2$. Intuitivement, c'est très clair. Mais comment faire la preuve? J'ai posé la question dans l'Intra en E20

D'où vient le $\frac{\phi^{d+3}}{\sqrt{5}}$?

Nous avons commencé par établir une récurrence sur G. Une récurrence linéaire, homogène de degré 2. On a remarqué que la récurrence pour G était très étroitement reliée à la récurrence de Fibonacci. Nous savons résoudre la récurrence de Fibonacci. On a donc pu trouver une solution pour G. Parce que G est une borne inférieure (nous avons choisi les "pires" arbres à chaque étape) nous avons l'inégalité que vous mentionnez.

Avez-vous compris pourquoi nous pouvons fixer d, et minimiser N, plutôt que de fixer N et maximiser d?

Prenez le cas simple. Si je montre que $d \leq \frac{N}{2}$, c'est la même chose que montrer que $N \geq 2d$. Alors nous décidons de plutôt faire cela. Fixons d, minimisons N. Nous construisons l'arbre qui minimise N. Arbre AVL de hauteur $d-1$ à droite, arbre AVL de hauteur $d-2$ à gauche. Ça c'est l'arbre qui minimise N. Si ce n'était pas cela, on pourrait remplacer l'arbre qui prétend être optimal par un arbre avec moins de nœuds. Contradiction. L'arbre qui minimise N doit être AVL $d-2$ à gauche, AVL $d-1$ à droite. Voyez-vous que l'un des sous-arbres doit avoir la hauteur $d-1$? Ok, c'est parce que nous supposons que nous commençons avec un arbre dans P_d. Maintenant, nous voulons minimiser le nombre de nœuds. À gauche il faut que cela soit $d-2$, pour que cela reste AVL. Et les sous-arbres de hauteur $d-1$ et $d-2$ doivent eux aussi minimiser le nombre de nœuds, vous êtes d'accord? Ils doivent donc être dans P_{d-2} et P_{d-1} respectivement. Voyez-vous que les sous-arbres doivent eux-mêmes minimiser le nombre de nœuds? Sinon, l'arbre original n'aurait pas été optimal. Cela nous donne une récurrence pour G. On peut résoudre la récurrence en comparant avec Fibonacci. Je n'ai pas discuté, dans le deuxième cas où deux rotations sont nécessaires, pourquoi $T2$ et $T3$ doivent avoir les hauteurs indiquées. Ce sont les mêmes arguments que nous avons utilisés dans le premier cas: l'arbre original était AVL, et le nœud S était la PREMIÈRE fois qu'il y avait un manque de balancement. Notre intuition nous dit que l'arbre le plus petit doit avoir une hauteur plus petite.

Les ϕ sont les solutions de la relation de récurrence de Fibonacci. Comme notre récurrence G est très proche, on s'en sert pour exprimer G_d, c'est bien ça?

C'est bien ça, mais une chose à la fois. Il faut commencer par voir que l'arbre qui minimise N doit avoir la forme mentionnée.

Regardez la question sur la hauteur de P_{d-2} dans l'Intra de E20.

Pourquoi j'ai posé cette question? Pour commencer, ce n'est pas si facile que cela, et pourtant, il est toujours pris comme évident! C'est une bonne question pour un examen à la maison, parce que je n'ai jamais vu la question discutée dans les preuves présentées. Mais la preuve n'est pas si facile. Beaucoup d'étudiants ont essayé de démontrer cela en utilisant la récurrence sur G, mais ça ne fonctionne pas. What's the method after all? Regardons l'arbre P_{d-2} qui est supposé avoir le nombre minimal de nœuds. Une façon de faire serait de prendre le chemin le plus long dans cet arbre. Si cela a longueur supérieure à $d-2$, on pourrait l'enlever, l'arbre resterait AVL, et pourtant il aurait moins de nœuds. Contradiction.

Mais la hauteur de l'arbre le plus petit doit être limitée pour que l'arbre complet soit balancé (AVL) : la hauteur $T_G - 1$.

Oui, j'ai parlé seulement de la partie la plus difficile. On peut éliminer d pour commencer, mais pourquoi pas $d-1$, c'est ça la question difficile.

Si les deux sous-arbres (g, d) sont $d-1$, ça ne minimise pas N. Alors, un sous-arbre doit être $d-2$.

Oui, cela ne minimise pas N, mais démontrez-moi cela!

Comment on peut éliminer les possibilités autres que $d-2$, $d-1$ et d?

Alors, l'approche générale dans le contexte d'une preuve comme ça est de supposer que l'énoncé est faux. Si on avait $d-1$, par exemple, le chemin de longueur maximale serait de longueur $d-1$. Comment procéder pour faire une telle preuve? Comment éliminer les possibilités que vous avez mentionnées? Faites l'hypothèse que l'énoncé est faux, trouvez une contradiction. J'avais dit dans l'énoncé qu'il fallait une preuve dans le même style.

Fin de la matière de l'intra

Buddy System

Ok, pour le Buddy System, avant la présentation principale, je veux m'assurer que vous comprenez le contexte. Vous vous mettez à la place de la personne qui implante un système pour allouer la mémoire aux usagers. Alors il y a des usagers qui viennent, ils ont besoin d'un bloc de mémoire, le système doit fournir un bloc de taille assez élevée pour permettre l'usager de faire ce qu'il veut.

Pouvez-vous définir $Disp(j)$ svp?

Le système dont je viens de vous parler doit garder trace des blocs qui sont disponibles, des blocs qu'il peut allouer aux usagers pour satisfaire leurs requêtes. Dans le Buddy System, les blocs vont toujours être de taille 2 exposant quelque chose, 8, 16, 32, etc. $Disp(j)$ est une liste des blocs disponibles de taille 2^j. Ok, mais on va voir que c'est très astucieux la façon de garder ces listes, et la façon de fusionner deux blocs adjacents (des "buddy" , des copains).

Disponible et libre sont-ils utilisés de façon équivalente dans votre explication?

Oui, disponible et libre sont équivalents.

Deux blocs disponibles de taille 2^j vont être fusionnés pour avoir un bloc de taille 2^{j+1}. Vous voyez pourquoi on veut fusionner des blocs. Sinon, à la longue vous pourriez avoir beaucoup de petits blocs disponibles ou libres, et pourtant incapable d'allouer un bloc de plus grande taille. Cela s'appelle la "fragmentation". Est-ce clair? Peut-être une centaine de blocs de taille 50 sont libres, mais éparpillés partout dans le bloc principal, et donc le système n'est pas capable de rendre un bloc de plus grande taille.

Comment on détermine la grandeur du bloc à allouer à l'usager?

L'usager spécifie cela dans sa requête.

Peut-il être fusionné ?

Nous allons voir tout cela en détail, mais mieux vaut comprendre l'idée générale avant la présentation. L'usager vient et il dit « j'ai besoin d'un bloc de $n = 5$ cellules ». Parfois on dit « $n = 5$ mots » mais il faut voir que les cellules ou mots alloués peuvent être des blocs de 1000 octets, ça m'est égal.

Le Buddy System fusionne seulement des blocs adjacents?

Oui, seulement des blocs adjacents. En fait, même pas TOUS les blocs adjacents. Nous allons voir que le système peut fusionner uniquement des blocs adjacents qui sont des "buddy", des copains. Cela veut dire qu'il y aura parfois un peu de "fragmentation", c'est-à-dire des blocs adjacents de 8 cellules qui ne peuvent pas être fusionnés, incapable donc de fournir un bloc de taille 16. Mais le fait de faire cela permet une très grande vitesse d'accès aux blocs. Voici une définition intuitive de "buddy": Les blocs doivent être alignés sur les frontières de mémoire 2^j. Cela veut dire quoi? Supposons 8 blocs de mémoire de taille 2 cellules. Cellule 1 et Cellule 2 font un bloc. Son buddy c'est le bloc des deux cellules 3 et 4.

Qu'est-ce qui arrive si l'usager demande trop d'espace?

La réponse à ta question est facile: on répond qu'on ne peut pas satisfaire à la requête.

At the end, I didn't understand the relationship between the doubly-linked list and the memory blocks we're trying to manage. That is, we have this idea of contiguous memory that we're trying to manage efficiently, but we're managing it with a doubly-linked list, (which itself need not be contiguous in memory), but we're accessing items in that doubly-linked list "quickly" simply by flipping bits.

You've got 64 cells available, total $N = 64$. They are just sitting there! Certain blocks inside this set of cells are available. For example, maybe 32 and 33 are available as a block of size 2. The linked list is formed by putting the link pointers WITHIN the available size. There is no separate linked list! There is lots of room for pointers within the cell itself. Voyez-vous ce que je viens de dire. Les listes $Disp[\,]$ ne sont pas séparées. Les listes sont faites en mettant les pointeurs de liste à l'intérieur des blocs qui sont disponibles.

Alors selon le Buddy System on alloue seulement des blocs de taille 2^k... Toujours exponentiel.

Oui, on se contente d'allouer des blocs uniquement de taille 2^k parce que cela nous permet une grande efficacité critère temps.

Donc la mémoire se décrit elle-même ?

Michael, je ne comprends pas ce que vous entendez par "se décrit" elle-même. La mémoire est là, mettons $N = 64$ cellules. Nous devons rendre certains segments de la mémoire temporairement à l'usager qui en a besoin. L'usager nous dit « J'ai besoin de 5 cellules ». Nous devons trouver 5 cellules qui sont libres. Pour faire cela, on garder trace des cellules qui ne sont pas déjà en utilisation.

Je veux dire, la liste qui décrit la division des blocs est stockée elle-même dans les blocs, donc l'allocation de la mémoire est définie par la mémoire qui est allouée.

Oui, le système se sert du bloc qui n'est pas en utilisation en ce moment pour emmagasiner les pointeurs qui relient les blocs qui ne sont pas en utilisation par les usagers. C'est astucieux! Il y a beaucoup de petites choses astucieuses là-dedans. Pas de listes séparées: les blocs libres sont reliés par des pointeurs emmagasinés à l'intérieur des blocs non-utilisés. On n'est pas obligé de passer à travers des listes pour trouver des blocs à fusionner: on y va en changeant un bit. Aussi astucieux! On a utilisé une liste doublement chainée. On peut facilement supprimer un bloc qui sera fusionné. Astucieux!

Est-ce que vous pouvez donner un exemple d'application du Buddy System dans l'industrie qui est fréquent?

Non, mon domaine de recherche n'est pas du tout dans "systèmes". Par contre, je parle informellement avec des collègues pour savoir ce qui est utilisé couramment dans l'industrie. Quand je dis "collègues" je veux dire des gens que je rencontre dans des comités qui travaillent dans le domaine "systèmes". Ce qui est clair, c'est que beaucoup de systèmes de "Allocation de mémoire" sont maintenant rendus caduques, et je n'en parle pas. Faut suivre des cours en Système pour connaître les méthodes courantes, je dirais que cela ne fait pas partie de "Structures de données" maintenant. Par contre, on me dit informellement que les Buddy Method sont toujours utilisés.

Pas besoin de déplacer le bloc pour fusionner, et seul le pointeur fonctionne-t-il ?

Non les blocs ne se déplacent pas. Ce sont des blocs de mémoire fixes. C'est notre interprétation de chaque cellule qui change.

Séance 10 – Hashing

J'ai une question toute simple : $P(T) = 20 \rightarrow H(T) = 6$... Je n'ai pas bien entendu l'opération pour arriver à 6.

$20\ mod\ 7 = 6$.

Je vois que le Hashing est plus facile pour enregistrer (avec $h(x)$), mais n'est pas pratique pour rechercher ou récupérer les données dans l'ordre... alors pourquoi est-ce que le Hashing est "intéressant" ? Quels sont les cas pratiques d'application?

Exemple classique: guichet automatique à la banque. Les requêtes de fouille doivent être instantanées. Je ne veux pas attendre 2 minutes pour savoir si j'ai $200 dans mon compte. Je veux une réponse immédiate. Par contre, si je décide de fermer mon compte (retrait dans la structure de données) je suis prêt à attendre. La fouille est très très vite. Retirer un élément est plus lent. Here is another example. Twenty years ago, when PC's were slow. I could search for a file on a PC, KNOWING THE NAME OF THE FILE. The little dog would be there, wagging its tail while the search was proceeding, and I could go for lunch. Là c'était dans mon propre ordinateur, en connaissant le nom du fichier. Et pourtant, aujourd'hui Google peut me trouver n'importe document sur la planète en quelques secondes. Comment faire?

Il y avait une idée intéressante à 5:30. Qu'est-ce que ça donne pour TRIER les listes externes? Est-ce que la réponse dépend du cas? Clé présente ou clé absente?

Si je sais a priori que la clef est absente? Si, a posteriori il arrive que la clef n'était pas présente. Qu'est-ce que ça donne de trier les listes externes? Dans les circonstances différentes? Peut-être qu'il y a un avantage. Quel est l'avantage? Quand on saura cela on peut décider si cela vaut la peine de trier. It "fails faster". More efficient. But how to decide whether the cost of sorting (and keeping sorted) is less overall than sequential search every time?

But we should prioritize the cost associated to the most common operation, don't we ?

Oui, après avoir estimé les coûts, on peut donner la priorité selon nos besoins.

Séance 11 – Hashing, Graphes

À 5:00 à peu près j'ai noté l'ensemble d'entiers $0, \ldots, M-1$ par $[0,\ M-1]$. Mauvaise notation, bien sûr, ce dernier est un ensemble de nombres réels. Je devrais avoir honte.

Pouvez-vous définir M svp?

M est la taille de la table. Nous commençons avec 26^{20} possibilités pour des clefs, et $h(x)$ envoie tout cela dans une petite table avec seulement M cases.

Une autre chose mal dite quand je parlais de hachage double : Pour expliquer l'avantage, j'ai dit que cela serait peu probable que $p(x)$ soit égal à 1. D'où vient cela, voyons?!

Plutôt: ça serait peu probable que $h(x) = h(y)$, et aussi $p(x) = p(y)$. C'est-à-dire que deux clefs distinctes vont suivre le même chemin de collision. Voyez-vous?

Au début de la deuxième partie du segment, j'ai utilisé h comme nom de clé. Pas un très bon choix de nom de clé! Possibilité de confusion entre clé et la fonction h.

Voici une idée. Pour sauver du temps, peut-on utiliser la même fonction de hachage pour p qu'on a utilisée pour h?

Pour sauver le temps de calculer p, c'est ça l'idée.

Je n'ai pas tout à fait compris comment la *tombstone* nous aide à trouver la prochaine clé.

Vous avez un chemin de collision que vous suivez pour trouver les clés dans chaque cas. Vous décidez de retirer une clé, et cela peut briser le chemin de collision pour d'autres clés. Cela dit à l'algorithme qui cherche une clé qu'il ne faut pas arrêter la recherche! Normalement ces recherches terminent quand elles tombent sur une case vide. On met une étoile pour dire à l'algo « cette case ne devrait pas être considérée vide pour fins de terminer la recherche d'une clef. Continuez. » Voyez-vous? Cela étant dit, l'algorithme d'insertion peut remplacer les étoiles par une nouvelle (vraie) clé. Mais, CELA étant dit, si vous faites cela, il faut faire attention au moment de l'insertion: Quand vous insérer, normalement vous cherchez pour voir si la clé est déjà présente. Il ne faut pas arrêter cette recherche avant d'avoir continué jusqu'à la fin du chemin, après une case étoilée il faut continuer, n'est-ce pas?

À 7:22 de la première partie du segment 3, j'ai fait une "correction" que je n'aurais pas dû faire:

J'avais simplement indiqué que w_i et w_{i+1} sont éléments de E. C'était parfait. Mais je ne sais pas pourquoi, pour telle ou telle raison j'ai décidé d'ajouter des parenthèses ( ) autour. Je n'aurais pas dû.

In an undirected graph, can we assume that edge $\{E, V\}$ always equals to $\{V, E\}$?

Je préfère changer votre notation. Dans un graphe orienté, les arêtes n'ont pas d'orientation. C'est comme une ville sans rues à sens unique. Si vous avez une ville sans rues à sens unique, toutes les rues ont deux sens, c'est logiquement la même chose qu' « aucune orientation ». Dans ce cas, si $i - j$ est une arête, alors $j - i$ est aussi une arête. Vous pouvez représenter avec les deux arêtes explicitement présentes, ou pas, dépendant de vos besoins.

Séance 12 – Tri topologique, Prim

Dans le segment 2 il y une erreur bête, le graphe pour le problème de l' « arbre sous-tendant minimal » est NON-orienté. Ça sera le problème intéressant de notre point de vue.

L'algo de base est N^2. Mais nous avons une idée brillante: Pourquoi pas ajouter une que de priorité pour avoir un algorithme $NlgN$? Nous faisons cela, et nous constatons que cette idée nous donne bel et bien une amélioration, MAIS seulement pour les graphes creux. Pour les graphes denses, cela n'aide pas. En fait pour les graphes denses cela donne un algorithme moins bon. En tout cas, arbre s t avec queue de priorité est un exemple intéressant pour des gens comme vous qui s'intéressent beaucoup aux structures de données!

Comment la queue de priorité est-elle implémentée dans cette version de l'algorithme?

Cela ne s'explique pas en 20 mots. Ça s'en vient très bientôt.

Bon, je répète, le graphe est non-orienté dans le cas de Prim.

Pour concevoir un arbre de longueur minimale, nous utilisons très suivant Kruskal. Est-ce que vous avez idée de l'ordre d'un tel algorithme? Comment l'implémenter?

Oui, Kruskal et Prim sont concurrents, je choisis Prim parce que c'est intéressant du point de vue de structure de données, c'est tout.

Ok, c'est quoi l'idée pour une amélioration de l'algorithme?

C'est que nous n'avons pas parlé de comment trouver le minimum des coûts à chaque étape. C'est-à-dire que j'ai parlé vite, j'ai dit, prenons le coût minimum. Comment trouver? Passer à travers les nœuds non-visités? Qu'est-ce que ça va donner comme coût total? N^2. L'idée brillante est la suivante: Pourquoi pas utiliser une queue de priorité pour emmagasiner les coûts pour ajouter un nouveau nœud? C'est cette idée que nous allons regarder, et qui va donner une amélioration, mais uniquement dans le cas de graphes creux.

Est-ce que l'idée d'utiliser la queue de priorité est de parcourir les nœuds en les emmagasinant dans le Min-Heap?

Oui, exactement cela. Mais il y a des choses qu'il faut regarder quand même: Comment garder ce Min-Heap à jour?

Pourquoi ça marche uniquement avec les graphes creux?

L'idée que j'ai déjà mentionnée: nous allons voir que cela donne un algorithme qui est $elog(e)$, à cause des mises-à-jour des éléments dans la queue. N'oubliez-pas que les coûts sont toujours en train de s'améliorer, le coût du meilleur point d'attache est toujours en train de changer, cela veut dire que nous devons mettre le Min-Heap à jour. À la fin c'est $elog(e)$ et pour un graphe dense, $e = N \times N$. $e \log(e)$ est moins bon que $N \times N$ dans ce cas. Dans le cas de graphe creux, $elog3$ est $NlogN$, c'est meilleur.

Séance 13 – Prim, Threaded trees/lists

Pour Dijkstra, il faut changer les étapes de l'algo.

Si $T[v].d + c_{vw} < T[w].d$ alors (deux choses)

$T[w].d \leftarrow T[v].d + c_{vw}$

$T[w].p \leftarrow v$

C'est tout.

Je ne comprends pas bien quand vous dites "donc le coût"... le coût mélangé de chercher minimums et de mettre à jour (Prim) ?

Voyez-vous, on garde trace maintenant du coût pour SE RENDRE au nœud et pas seulement le coût d'attache. C'est l'algorithme de plus court chemin, une origine, N destinations.

Pointeurs "inversés" dans l'algorithme de Prim. Bonne idée. Pour moi, cela me rappelle autre chose. À l'époque je parlais d'algorithmique pour parcourir un arbre sans pile, et sans ficelle. C'est possible de parcourir un arbre sans utiliser une pile, et sans ajouter des pointeurs "ficelles". Voyez-vous comment?

L'algo c'est l'algorithme de Deutsch-Schorr-Waite. Inventé pour être utilisé pour parcours une liste généralisée pour récupérer l'espace. L'algo devait tourner exactement au moment quand il ne restait plus de mémoire, il ne fallait pas utiliser une pile.

Comment faire cela?

L'idée est d'inverser les pointeurs gauches et droits dans l'arbre par des pointeurs inversés, des pointeurs qui indiquent le parent. (!) À chaque moment, quand vous êtes dans un nœud, il y a le chemin vers la racine qui est indiqué par les pointeurs inversés. Il faut bien sûr remettre ces pointeurs aux valeurs originales, au fur et à mesure de traverser l'arbre. C'est possible, avec trois pointeurs auxiliaires. Moins d'intérêt aujourd'hui, mais peut-être vous voulez mettre cela dans votre "sac de trucs".

Quand je parlais de l'insertion dans les arbres cousus j'aurais dû, à la première étape, i.e. premier pointeur ajouté, j'aurais dû indiquer comme ficelle, pas un lien ordinaire.

Quel est le sens du champ p dans l'algorithme de Dijkstra? Pour Prim c'était le "point d'attache". Pour Dijkstra c'est quoi?

Dans Dijkstra, nous regardons plutôt à chaque étape le meilleur choix jusqu'ici pour se rendre au nœud. Quand un nœud est ajouté à l'arbre que nous sommes en train de construire, nous notons cela dans le champ p. Ce qui veut dire qu'on peut interpréter p comme "prochain". p nous donne le prochain nœud sur le plus court chemin depuis le nœud de départ choisi au début. Vous pourriez écrire un petit segment de code pour tracer le chemin le plus court depuis ce nœud de départ en utilisant à chaque étape le champ p.

Séance 14 – Skip lists, Arbres 2-3

Je ne comprends pas encore comment choisir le NIV_MAX de façon aléatoire.

Ça s'en vient dans le prochain segment!

En général, combien de mémoire d'extra est requis pour un skip-list?

Bonne question. C'est lgN pointeurs par noeud. Pas gratuit en terme d'espace. Par contre, comme je l'ai dit, peut-être pas la fin du monde. Cela dépend de l'encombrement des données par rapport à la taille des pointeurs.

Je suppose qu'on peut le calculer facilement.

Oui. Mais si vos données prennent plusieurs M octets, c'est peut-être infime que mémoire supplémentaire. Ça dépend.

Quand vous parlez de probabilité, c'est la probabilité de quoi?

Vous allez mieux comprendre au début du segment 2. Nous choisissons le nombre de pointeurs avec une certaine probabilité. C'est la probabilité que la nouvelle clé est mise dans i listes.

Est-ce que la skip-list est toujours triée?

Oui, nous insérons les clés au fur et à mesure dans l'ordre trié. C'est comme créer un arbre binaire de recherche en ordre RGD. Nous insérons les clés de façon à ce que l'ordre soit respecté.

Monologue de Neil au chat 2 :

DANGER de malentendu! Il y a un changement de contexte qui arrive à peu près ici. Comprenez ce que j'essayais de faire. Je vous vous expliquer d'où viennent les règles bizarres sur le nombre de clés dans chaque nœud d'un arbre B. J'introduis donc les arbres-B, avec une grande valeur de M. Le but est de minimiser le nombre d'accès au disque. On peut voir à ce moment d'où viennent les règles bizarres. On veut que M soit grand pour minimiser le nombre d'accès au disque. Par contre, on ne peut pas insister sur M branchements partout, ça serait impossible de retirer des noeuds. Il faut comprendre cette idée. $\left\lceil \frac{M}{2} \right\rceil$ jusqu'à M, comme gamme de possibilités, vous permet de retirer les noeuds. Ok, nous avons compris cela. Maintenant, on remarque que ça ne donne rien de diviser les arbres en page en mémoire interne, alors NOUS OUBLIONS ce critère de minimiser le nombre d'accès. MAIS: même si l'idée de B-tree ne fonctionne pas en mémoire interne pour la raison que ça fonctionnait dans le cas de données externes, rien ne nous empêche de regarder ce que ça donne pour des petites valeurs de M, $M = 3$ ou $M = 4$ par exemple. Il arrive par hasard que cela donne quelque chose d'utile dans le contexte de données internes POUR DES RAISONS COMPLÈTEMENT DIFFÉRENTES. Ces raison différentes sont les suivantes. Cela donne une méthode qui est un peu comme AVL à cette différence près. Avec AVL on laissait varier les hauteurs des sous-arbres un peu. Cela nous a laissé le jeu pour faire les mises-à-jour. Le nombre de pointeurs dans chaque nœud pour AVL était 2. Les arbres avec $M = 3$ nous donne une autre façon d'avoir le jeu de faire les mises-à-jour. Ils gardent les hauteurs des sous-arbres constants, mais ils laissent un peu de jeu dans le nombre de pointeurs dans chaque nœud. Alors finalement cela n'a rien à voir avec les critères de l'arbre B, c'est d'autre raisons complètement. Mais ce sont les arbres B qui permettent à comprendre d'où cette idée est venue. Comprenez qu'il y a un changement de motivation entre arbre-B et arbre 2-3.

Pourquoi on doit insister pour avoir au moins $\left\lceil \frac{M}{2} \right\rceil$, mais on ne peut pas insister sur M partout?

Cela étant compris, nous pouvons OUBLIER ce critère, et regarder quand même ce qui se passe en mémoire interne avec $M = 3$. Pourquoi pas? En regardant $M = 3$, on constate que nous avons trouvé une structure

intéressante, intéressante pour d'autres raisons. À savoir, cela donne une structure un peu analogue à AVL. Et on constate qu'on peut établir des algorithmes pour insérer, retirer. Tout fonction convenablement. En E20 il y avait de la confusion sur ce changement de contexte, alors j'avais préparé une vidéo supplémentaire. J'ai mis la vidéo sur le site, vous pouvez regarder si vous voulez. Cela répète les mêmes choses. Je répondais à des questions des étudiants en E20.

Sur les skiplists, comment faire la recherche quand les pointeurs (NIV_MAX) ont été créés de façon aléatoire ? Dans ce cas on devrait recommencer chaque fois la recherche dans la liste du niveau inférieur quand on ne trouve pas la clé?

Oui, pour les détails de la recherche même, regardez dans Weiss. L'idée générale est claire: Commencez par décider si l'élément est plus loin que le milieu. Après, descendez d'un niveau. Il faut faire ce processus pour chaque recherche.

Dans l'algorithme de suppression, on n'a pas discuté de ce qui déclenche le rebalancement de l'autre sous-arbre.

Il n'y a pas de rebalancement à faire! Il faut comprendre cela! Si vous regardez les opérations d'insertion, ou de retrait, la structure garde toujours sa forme de profondeurs égales! Même si la hauteur de l'arbre change, elle garde toujours sa forme de profondeurs égales!

Réduction de hauteur je devrais dire.

Oui, la hauteur change. Ce qui ne change pas est la chose suivante: Le nombre de liens entre la racine et la feuille au plus bas niveau est le même pour chaque feuille. Voyez-vous?

Je cherche seulement à comprendre ce qui déclenche la correction de l'arbre gauche, la réduction de hauteur. Comment l'algorithme traite cela?

(Rafael) Je trouve que la hauteur change lorsqu'on ne peut pas trouver de sibling ou parent où voler, alors on va jusqu'à la racine... et la correction affecte tout l'arbre (gauche et droit). Il faut revoir l'exemple.

(Neil) Dessinez cela sur papier, et regardez ce qui se passe. Vous essayez de voler une clé à gauche, ça ne marche pas. Alors vous volez du parent, ça fait disparaître le parent, et la hauteur de l'arbre vient de diminuer. Comme dit Rafael, la correction affecte tout l'arbre, gauche et droit.

Est-ce que ce les arbres 2-3 sont semblable aux arbres AVL à cause que les noeuds (non-leaf) ont au maximum $M - 1$ clefs?

Si je mentionne AVL, c'est juste ceci: In fact, AVL et 2-3 trees come from complete different motivations, as I've explained. But looking at them, after the fact, we can compare them. We notice that AVL gets the slack it needs to do updates by permitting small differences in the height of the subtrees. 2-3 trees get the slack they need in a different way. They keep the height of the subtrees constant, but they permit a bit of play in the number of keys in each node. Je crois avoir déjà dit cela en français. Les arbres AVL et 2-3 viennent de sources complètement différentes. Mais après coup on peut les comparer: il s'agit de 2 façons différentes d'avoir un peu de jeu, pour permettre les mises-à-jour. Si vous regardez la vidéo supplémentaire de E20, vous allez voir les mêmes remarques répétées. J'ai insisté sur le changement de contexte parce que j'ai réalisé que je n'avais pas assez bien expliqué dans la vidéo même.

Séance 15 – Arbres 2-4, Arbres rouge-noir

Dans la preuve (black height etc) c'est le nombre de noeuds dans tout le sous-arbre, y compris la racine du sous-arbre.

Ça serait très utile de motiver la discussion par un exemple concret. Ce n'est pas tout à fait clair pourquoi ils sont si utiles.

On se permet d'insérer une couche max de noeuds entre chaque niveau de noeuds noirs. Si on regarde de loin, ce sont des arbres un peu comme les arbres AVL. La borne sur la hauteur maximale est un peu moins satisfaisante que AVL. Par contre, nous allons regarder deux versions de l'arbre rouge-noir. Normalement, pour AVL, et rouge-noir "bottom up", ce que vous faites est le suivant: Vous descendez dans l'arbre pour chercher ou mettre la nouvelle clé, ou pour chercher la clé à retirer. Après cela, vous remontez dans l'arbre pour corriger les facteurs de balancement (AVL) ou pour corriger le problème de > 2 noeuds rouge. L'avantage de rouge-noir (Top-down) est qu'il ne sera pas nécessaire de remonter dans l'arbre.

À 2:27 je vous demande "êtes-vous certain qu'il n'y a pas de problème?"

Vous devriez être certain! Il n'y a pas de problème! Je vous demande d'y penser, c'est tout!

Alors le processus en deux parties.

D'abords, décidez où mettre la clef. Après: Remonter dans l'arbre pour corriger le problème d'avoir deux couches rouges de suite. Le problème arrive dans le cas de l'oncle rouge. Le problème peut se corriger en répétant le processus que j'ai décrit, mais il faut répéter à chaque niveau, en remontant vers la racine. C'est très semblable aux arbres AVL finalement, ce processus de remonter pour corriger.

A-t-elle les arbres 2-4 à l'esprit, et modifié cela pour inventer les arbres rouge-noir?

Normalement on voit les arbres rouge-noir présenté comme idée indépendante, et on remarque après que c'est équivalent aux arbres 2-4. En tout cas, la correspondance est intéressante.

Je n'ai pas bien compris la correspondance entre les arbres 2-4 et les skip-list déterministes. Pouvez-vous l'expliquer?

Il faut attendre la présentation de la semaine prochaine pour vraiment comprendre. Mais rien n'empêche de faire une description préliminaire maintenant. Cela vous prépare pour la présentation: Pugh a suggéré une façon de réaliser les SkipLists. Cela permettait une structure qui n'était pas parfaitement de la bonne forme, mais assez proche pour assurer un comportement $logN$. Il y a 20 ans à peu près, une approche déterministe a été suggérée à la place de la méthode de Pugh. On commence par regarder le nombre de noeuds dans chaque "écart" à chaque niveau. C'est quoi un "écart"? C'est ma traduction du mot anglais "gap". Regardons dans un SkipList au niveau h (quelconque). Entre deux noeuds de niveau h, vous avez un certain nombre d'autres noeuds reliés aux niveaux plus bas. Il y a un "gap" entre les deux noeuds. J'ai traduit cela en disant il y a un "écart" entre les deux noeuds. Ok, quelle est l'idée de la méthode? Dans le gap entre deux noeuds, dans l'écart, vous acceptez un certain minimum et un certain maximum pour le nombre de noeuds de niveau $h - 1$. Ça doit être 1, 2 ou 3, et c'est pour cela que la méthode s'appelle « 1-2-3 deterministic skip-list ». On est capable d'insérer et retirer, en respectant toujours cette condition 1-2-3.

Les nœuds peuvent être déplacés.

Oui, vous avez raison. Mais après coût, chaque fois qu'on fait quelque chose en respectant cette condition, on voit qu'on peut interpréter ce qu'on vient de faire comme une manipulation des noeuds d'un arbre 2-4. La

correspondance est biunivoque. Mais pour vraiment comprendre, il faut attendre la présentation des 1-2-3 deterministic skip lists.

Je trouve intéressantes ces correspondances, mais je veux souligner que la chose la plus importante dans la suite est l'idée de Splay Tree et méthodes semblables.

C'est une idée complètement différente pour la manipulation d'arbres binaires. AVL et ROUGE-NOIR sont semblables, d'un certain point de vue. Splay Tree est différent. J'aime présenter comme "motivé par l'idée de self-organizing list". C'est une façon de voir, mais je souligne à maintes reprises, que les avantages du Splay tree vont beaucoup plus loin que l'idée de "self-organizing list".

Séance 16 – 1-2-3 deterministic skiplists

Vous voyez l'idée de base de l'insertion?

Il n'y a pas de problème à insérer si l'écart est inférieur ou égal à 2. Si l'écart est égal à 3, on s'en occupe en montant le noeud au milieu.

Oui, mais j'ai un peu de confusion par rapport à l'algorithme quand l'écart est 3. Dans un exemple, vous avez monté le 25, mais à la fin de la vidéo, vous avez parlé de monter la cellule dans le milieu des trois, c'est-à-dire le 35.

Commençons par l'autre bout. Le cas ≤ 2 est clair. Ok, dans mon exemple, il y avait des écarts de 3 à deux niveaux différents. Dans le premier cas, il a fallu monter le 25 pour pouvoir descendre. On l'a fait. Cela nous a donné, au plus haut niveau, deux écarts de 1, à gauche et à droit. Après, on descend dans l'un ou l'autre. En descendant on a trouvé un autre écart de 3. Alors, on a dû faire la même chose au plus bas niveau. J'ai aussi mentionné, dans le cas de l'insertion dans un écart de 2, que cela donne maintenant un écart de 3. Ce qui veut dire que si plus tard vous voulez insérer une autre clef dans ce "gap", alors il va falloir remonter le noeud au milieu. Je veux vite motiver ce qui se passe pour le retrait, dans le prochain segment. Je commence par expliquer ce qu'il faut faire pour retirer une clef du premier écart, ou du "first gap". Ça se fait, mais pour le faire, il faut utiliser le deuxième gap. Problème: je vois quoi faire pour insérer dans le 2ième gap, je me sers du 3ième. Et ainsi de suite. Mais qu'est-ce que je fais pour traiter le dernier écart? Il n'y a pas un gap plus loin à utiliser. Réponse: Je fais demi-tour, et j'utilise exactement la même méthode, mais en allant de droite à gauche. En fait, les auteurs utiliser gauche à droite pour le premier, et ils vont en sens inverse pour tous les autres. Peu importe.

Monologue de Neil au chat 2 :

Vous voyez l'idée? On commence par vous dire comment retirer une clé qui est, par hasard, dans le premier "gap". Cette partie est claire? S'il y a au moins 2 noeuds de niveau $h - 1$ au prochain niveau, il n'y a rien à faire. C'est quand il y avait seulement un noeud au niveau $h - 1$ qu'on a un problème. Pour régler le problème, on descend le "prochain" noeud, et on remonte celui qui vient après, selon le cas. Cela nous permet de descendre dans le premier gap avec au moins deux noeuds, sans gâcher les choses à droite. Avec toujours 2 noeuds à gauche, on peut continuer de descendre au besoin. On pourrait prendre ces règles pour inventer une méthode pour retirer une clef dans un arbre 2-4!

Bon, ok, il nous reste un sujet, une nouvelle méthode complètement, les Splay Trees. On change notre critère un peu. Si vous acheter une voiture, quel est le coût de l'exploitation de cet appareil? $X par année. Mais cela ne tient pas compte du coût d'achat. Vous dire, ok, cela me coût $Y par année pour "déprécier" la voiture, mettons $2,500 par année si cela vous a couté $25,000 et vous allez garder 10 ans. Mais une autre façon de s'exprimer est de parler du coût amorti. En anglais, "what is the amortized cost of the vehicle over 10 years"? C'est le coût total, prix d'achat plus tous les autre frais pendant 10 ans. Et après, vous pouvez prendre le coût total et diviser par 10. C'est une autre façon de mesurer le coût par année.

Séance 17 – Splay Trees

J'ai peut-être égaré avec ma comparaison avec le plus mauvais cas dans QuickSort.

Dans le cas où nous entrons les clés dans l'ordre pour créer une liste linéaire, combien de rotations pour créer cette liste linéaire, N ou N^2? N, pourquoi? C'est parce qu'il n'y a qu'une seule rotation chaque fois qu'on ajoute un noeud, il ne s'agit pas du tout de $1 + 2 + \ldots + N$. Parce que dans tous les cas (pour ce cas mauvais), on doit simplement insérer chaque nouvel élément avec une rotation à la racine. Comme je l'ai dit, j'avais simplement peur que j'ai égaré en comparant avec Quicksort. Ce qui est impressionnant avec les Splay trees c'est la simplicité de la chose. Finalement, je n'ai jamais fait un exemple de rotation ZIG-ZAG, mais ce n'est pas plus compliqué.

Is there a resource that describes the algorithm for insertion and deletion somewhere? It is not in Weiss, and the question from last year's final is confusing without one.

Youtube.

Une chose que les étudiants ont trouvé compliqué dans le passée est le truc de "choix de direction". Et pourtant c'est simple. L'algorithme pour "tomber dans le premier intervalle" compte sur le fait qu'il y a un deuxième intervalle. Très bien. Si on doit tomber dans le dernier intervalle, on pourrait tout simplement faire demi-tour, et faire la même chose, mais de droite à gauche. Vous êtes d'accord? Cela veut dire que nous pourrions faire tous sauf le dernier de gauche à droite, et le dernier de droite à gauche, ou bien tous sauf le premier de droite à gauche, et le premier de gauche à droite. Peu importe… Quant à l'algorithme pour descendre, il faut regarder les exemples que j'ai donnés.