Structures de données II.

Xavier Dubuc

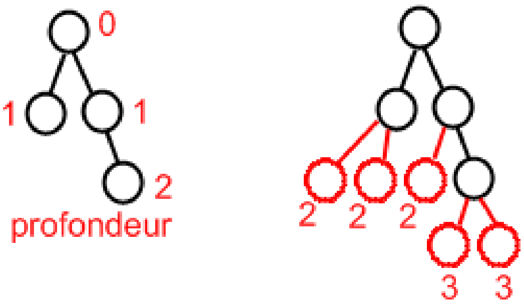
8 octobre 2009

Table des matières

1 Chapitre 2 - Les arbres

1.1 Propriétés des arbres.

1. Si T est un **arbre binaire** à n noeuds et de hauteur h, alors on a :
   - $h \leq n \leq 2^k - 1$
   - $\log_2(n+1) \leq h \leq n$ (Corollaire h est en $O(n)$)
2. Si T est un **arbre binaire** à n noeuds et de hauteur h, soit n_I son nombre de noeuds internes et n_F son nombre de feuilles, alors on a :
   - $h - 1 \leq n_I \leq 2^{k-1} - 1$
   - $1 \leq n_F \leq 2^{k-1}$
3. $C(t)$ complété de T, **arbre binaire** de recherche, si T a n noeuds, alors $C(T)$ a $2n+1$ noeuds (corollaire, T a $n+1$ références vides).
   Définition récursive : Profondeur d'un noeud :
   – Profondeur de la racine : 0
   – Si un noeud a une profondeur p, alors ses fils ont une profondeur de $p+1$.
   Définition : T, **arbre binaire** , $C(T)$ son complété, on définit :
   – *Internal path length* : $I(T)$: somme des profondeurs des noeuds internes de T.
   – *External path length* : $E(T)$: somme des profondeurs des feuilles de T.
   Exemple :



$I(T) = 0 + 1 + 1 + 2 = 4$
$E(T) = 2 + 2 + 2 + 3 + 3 = 12$

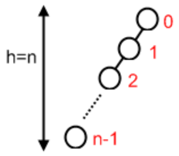
4. $E(T) = I(T) + 2n$ et $(n+1)\log_2\left\{\dfrac{(n+1)}{4}\right\} < I(T) \leq \dfrac{n(n-1)}{2}$

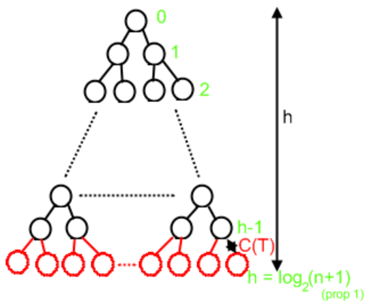
<u>Preuve</u> :
Prouvons que $E(T) = I(T) + 2n$, calculons de 2 façons $\sum_{v\ noeuds\ de\ C(T)} prof(v)$:
$(1) = \sum_{v\ noeuds\ internes\ de\ C(T)} prof(v) + \sum_{v\ feuilles\ de\ C(T)} prof(v)$
$= \sum_{v\ noeuds\ de\ T} prof(v) + \sum_{v\ feuilles\ de\ C(T)} prof(v) = I(T) + E(T)$

$(2) = prof(racine) + \sum_{v\ noeuds\ de\ T} 2\,(prof(v) + 1) = 0 + 2\sum_{v\ noeuds\ de\ T} prof(v) + 2n$
<u>*Explications*</u> : tout noeud de $C(T)$ est le fils d'un noeud interne de $C(T)$ c'est-à-dire un noeud de T (sauf la racine), de plus, tout noeud interne de $C(T)$ a 2 fils par définition.
(1)=(2) $\Rightarrow I(T) + E(T) = 2n + 2I(T) \Rightarrow E(T) = I(T) + 2n$

Prouvons que $(n+1)\log_2\left\{\dfrac{(n+1)}{4}\right\} < I(T) \leq \dfrac{n(n-1)}{2}$,

Comme pour la hauteur, on va s'interesser à un arbre qui a un nombre minimum de noeuds, et à un arbre qui a un nombre maximum de noeuds.





<u>Premier dessin</u> :
$I(T) = 0 + 1 + ... + (n-1) = \dfrac{(n-1)n}{2}$
<u>Second dessin</u> :
$I(T) = E(T) - 2n$
$= \sum_{v\ noeuds\ de\ C(T)} prof(v) = \sum_{v\ feuilless\ de\ C(T)} \log_2(n+1)$
$= (n+1)\log_2(n+1) - 2n$

<u>En général :</u>

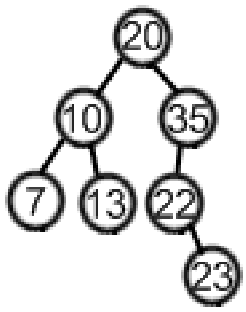
$(n+1)\log_2(n+1) - 2n \leq I(T) \leq \dfrac{(n-1)n}{2}$
$(n+1)\log_2(n+1) - 2(n+1) < I(T)$
$(n+1)\left[\log_2(n+1) - 2\right] < I(T)$
$(n+1)\left[\log_2(n+1) - \log_2(4)\right] < I(T)$
$(n+1)\log_2\left(\dfrac{(n+1)}{4}\right) < I(T)$

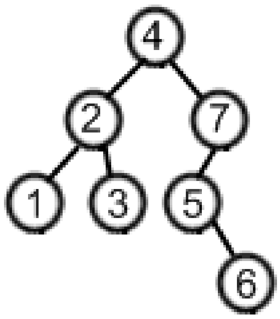
Commentaires :

h a une complexité comprise entre $\log_2 n$ et n et I(T) entre $n \ \log_2 n$ et n^2.

2 Chapitre 3 - Arbres binaires de recherche

Définition : il s'agit d'un arbre binaire(y compris l'arbre vide) tel que pour tout noeud, la donnée qui s'y trouve est < à toutes les données du sous-arbre droit et > à celles du sous-arbre gauche.
Exemple :

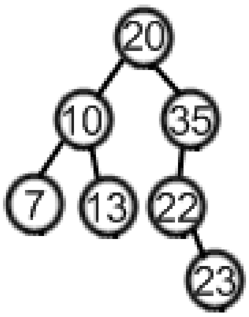




2.1 Hauteur

Au pire, la hauteur h est en $O(n)$ (voir la première propriété)

2.2 Recherche d'une donnée k dans un arbre binaire de recherche.



Recherche avec succès : $k = 22$
A chaque noeud rencontré, on laisse tomber l'un des 2 sous arbres. (recherche dichotomique) Ici, contrairement au calcul de la hauteur, on ne visite qu'une partie des noeuds. (placé sur un chemin partant de la racine).
Recherche avec succès : $k = 12$
Le chemin suivi aboutit à une référence vide $\rightarrow$ *12 n'est pas présent.*

Nous allons développer un algorithme basé sur la définition récursive des arbres binaires de recherche,
Cas de base : T, arbre vide, k n'est donc pas présent.
Cas général : T est un arbre contenant une donnée x et 2 sous-arbres T_1 et T_2.

- $k = x$, k est présent, on l'a trouvé.
- $k > x$, k est présent dans T si et seulement si k est présent dans T_2 (l'arbre de droite).
- $k < x$, k est présent dans T si et seulement si k est présent dans T_1 (l'arbre de gauche).

2.2.1 Algorithme

Algorithme **Recherche**(T,k)
Entrée : *T, **arbre binaire** de recherche.*
k, une donnée.
Sortie : *booléen vrai ssi k est dans T.*

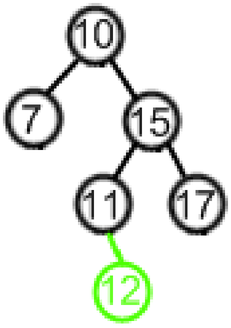
Si $EstVide(T)$ *alors retourner* **faux**
Sinon Si $(T_{data} = k)$ *alors retourner* **vrai**
Sinon Si $(k > T_{data})$ *alors retourner* **Recherche**(T_{right}, k)
Sinon retourner **Recherche**(T_{left}, k)

2.2.2 Complexité dans le pire des cas

- Recherche avec succès
  Recherche de k dans une feuille située le plus bas possible dans l'arbre :
  - nombre de noeuds visités : h
  - coût local par noeud : $O(1)$
  - pas de références visitées.
  $\rightarrow h * O(1) = O(h) = O(n)$ la même complexité que pour les listes triées mais en pratique, le comportement est meilleur que celui des listes.
- Recherche avec échec

  Identique mis à part un travail en $O(1)$ pour traiter la seule référence vide visitée, ce qui ne modifie pas la complexité globale, on en tire donc les mêmes conclusions.

2.3 Insertion d'une donnée k dans un arbre binaire de recherche.



1) Rechercher l'endroit où insérer
2) Insertion proprement dit de la donnée à l'endroit repéré.

Raisonnement :
T,k entrées ; T' sortie (**arbre binaire** de recherche résultant de l'insertion de k dans T)
Cas de base : *T arbre vide $\rightarrow T'$ sera un arbre composé d'un seul maillon contenant k.*
Cas de général *T est un arbre contenant une donnée x et 2 sous-arbres T_1 et T_2.*

- $k = x$, k est déjà présent, $T' = T$.
- $x < k$, k doit être inséré à droite.
- $x > k$, k doit être inséré à gauche.

2.3.1 Algorithme

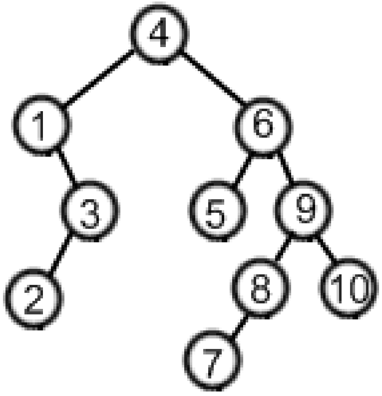
Algorithme **Insertion**(T,k)
Entrée : T, ***arbre binaire** de recherche.*
k, une donnée.
Sortie : T'.
Si $EstVide(T)$ *alors retourner* $CreationNoeud(k)$
Sinon Si $(T_{data} = k)$ *alors retourner* T
Sinon Si $(T_{data} < k)$ *alors* $T_{right} \leftarrow$ **Insertion**(T_{right}, k)
Retourner T
Sinon $T_{left} \leftarrow$ **Insertion**(T_{left}, k)
retourner T

Algorithme **InsertionBis**(T,k)
Entrée : T, ***arbre binaire*** *de recherche.*
k, une donnée.
Sortie : / *(T modifié).*
Si $EstVide(T)$ *alors* $CreerNoeudBis(T, k)$
Sinon Si$(T_{data} < k)$ *alors* **InsertionBis**(T_{right}, k)
Sinon Si $(T_{data} > k)$ *alors* **InsertionBis**(T_{left}, k)

2.3.2 Complexité dans le pire des cas

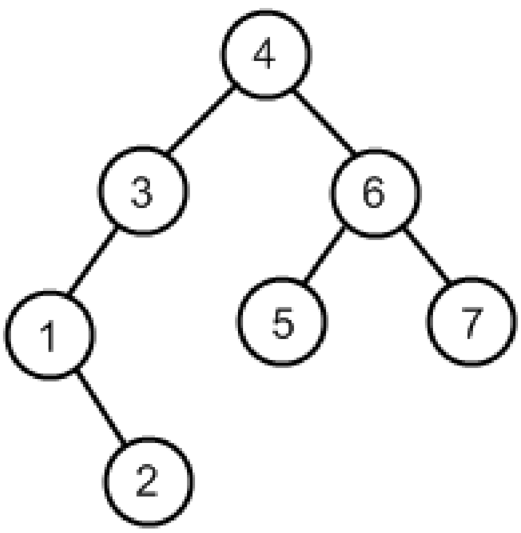
Insertion = recherche + travail complémentaire, en cas de recherche avec succès, il n'y aucun travail complémentaire, on obtient donc du $O(h)$, quant au cas de la recherche avec échec, un travail supplémentaire en $O(1)$ est accompli, ce qui ne modifie pas la complexité globale qui reste en $O(h)$.

2.4 Suppression d'une donnée dans un arbre binaire de recherche.



Si on veut supprimer **11**, il faut faire une recherche, se rendre compte que **11** n'est pas dans l'arbre et ne rien faire.
Si on veut supprimer **7**, **7** est dans une feuille, il faut supprimer la feuille.
Si on veut supprimer **1**, **1** est dans un noeud ayant un sous arbre droit, il faut donc supprimer le noeud contenant **1** et rattacher le sous-arbre comme sous-arbre gauche du père du noeud contenant **1**.

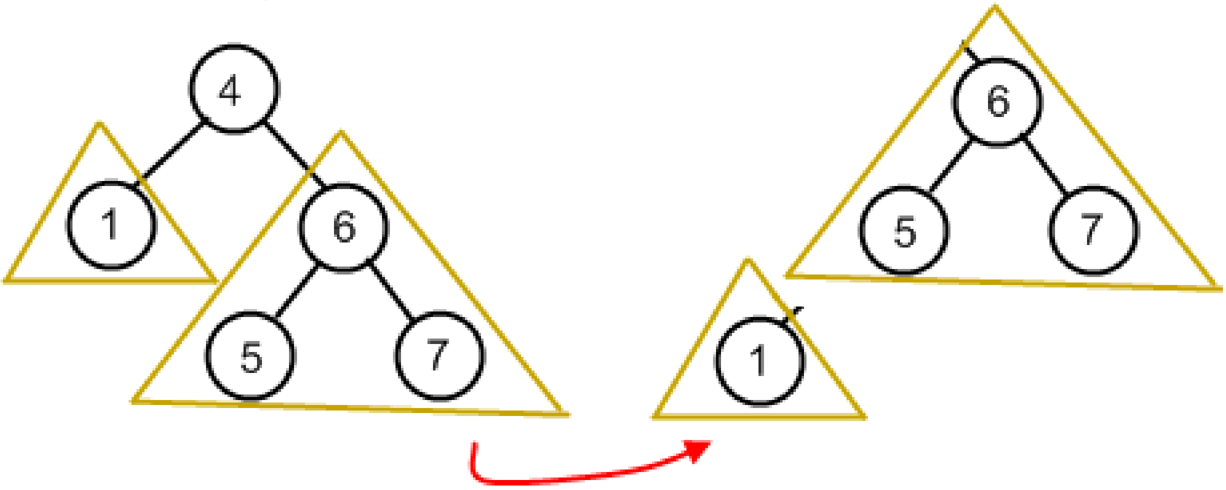
Autre exemple :



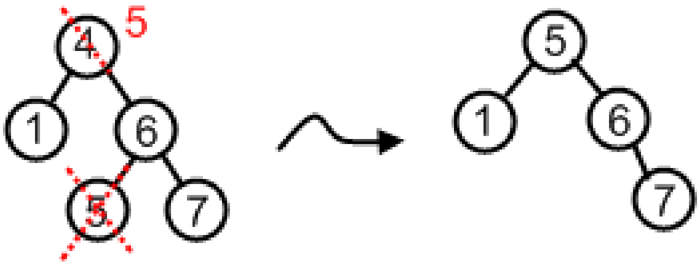
Si on veut supprimer **2**, **2** est dans une feuille, il faut supprimer la feuille.
Si on veut supprimer **3**, **3** est dans un noeud ayant un sous arbre droit, il faut donc supprimer le noeud

contenant **3** et rattacher le sous-arbre comme sous-arbre gauche du père du noeud contenant **3**.
Si on veut supprimer **4**, on peut appliquer 2 idées :

1ère idée : On supprime le noeud contenant **4** et on raccroche son sous-arbre gauche comme sous-arbre gauche de la feuille la plus à gauche du sous-arbre droit.



→ Ce n'est pas une bonne idée, en effet, l'arbre va tendre vers une liste, tout s'ajoutant sur la gauche.
2ème idée : On remplace la donnée à supprimer par une autre donnée. *($max(T_{left})$ ou $min(T_{right})$ en l'occurence)*



Remarque : Le minimum utilisé est une feuille ou un noeud qui possède un sous-arbre droit, la suppression de ce noeud est donc «facile».

En gros : la suppression se déroule en 3 étapes :
1. Trouver la donnée à supprimer (cf *recherche*)
2. Repérer le cas :
   – Suppression d'une feuille.
   – Suppression d'un noeud possédant 1 fils.
   – Suppression d'un noeud possédant 2 fils.
3. Si on se trouve dans le dernier cas, il faut effectuer la recherche du minimum et le remplacement de données.

Raisonnement général : nous allons résoudre ce problème grace à 3 algorithmes différents, l'un permettant de supprimer la racine d'un arbre, l'un permettant de supprimer le minimum d'un arbre et faire l'échange des données ainsi qu'un dernier supprimant les feuilles et appellant les 2 autres algorithmes dans le cas où l'argument ne serait pas une feuille.

2.4.1 Premier Algorithme

Entrée : *T, **arbre binaire de recherche** et k une donnée.*
Sortie : *T', **arbre binaire de recherche** résultant de la suppression de k dans T.*

Cas de base : T est un arbre vide, ce qui signifie que k n'est pas dans T, T' est donc l'arbre vide.
Cas général : T est un arbre contenant une donnée x et possédant un fils gauche, T_1, et un fils droit, T_2 et on suppose que l'on peut calculer T_1' et T_2' les arbres résultants de la suppression de k dans T_1 et T_2 respectivement. 3 cas de figure :

- $k > x$: T' sera l'arbre contenant x, ayant T_1 comme fils gauche et T_2' comme fils droit.
- $k < x$: T' sera l'arbre contenant x, ayant T_1' comme fils gauche et T_2 comme fils droit.
- $k = x$: On a trouvé k, on appelle le second algorithme qui va s'occuper de le supprimer.

Algorithme **Suppression**(T,k)
Entrée : *T, **arbre binaire de recherche**.*
k, une donnée.
Sortie : / *(T modifié en T').*
*Si (non **isEmpty**(T)) alors*
*Si ($k = T_{data}$) alors **SuppressionRacine**(T)*
*Sinon Si ($k > T_{data}$) alors **Suppression**(T_{right}, k)*
*Sinon **Suppression**(T_{left}, k)*

2.4.2 Second Algorithme

Entrée : *T, **arbre binaire de recherche** non-vide.*
Sortie : *T'', **arbre binaire de recherche** résultant de la suppression de la racine de T.*

Cas de base : T est une feuille, T'' est donc un arbre vide.
Cas général : 3 cas de figure :

a) T est un arbre contenant une donnée k et possédant un sous-arbre **gauche**, T_1, qui est un **arbre binaire de recherche** ; T'' est donc T_1, en effet, si on supprime la racine, il ne reste plus que T_1.
b) T est un arbre contenant une donnée k et possédant un sous-arbre **droit**, T_2, qui est un **arbre binaire de recherche** ; T'' est donc T_2, en effet, si on supprime la racine, il ne reste plus que T_2.
c) T est un arbre contenant une donnée k et possédant **2 fils**, dans ce cas on fera appel au 3ème algorithme.
   On va construire T'', l'arbre dont la racine sera $\min(T_2)$, le fils gauche sera T_1 et le fils droit T_2''' l'arbre résultant de la suppression de $\min(T_2)$ dans T_2. T'' est bien un **arbre binaire de recherche**. En effet :
   Avant la suppression, on a T_1,T_2 2 arbres binaires de recherche, on a donc la relation $T_1 < k < T_2$ ou encore $T_1 < k < min(T_2) < T_2'''$,
   Après la suppression, on a T_1 et T_2''' 2 arbres binaires de recherche *(par le 3ème algorithme)* et on a bien $T_1 < min(T_2) < T_2'''$, ce qui implique que, par définition, T'' est un **arbre binaire de recherche**.

Algorithme **SuppressionRacine**(T)
Entrée : *T, **arbre binaire de recherche**non-vide.*
Sortie : / *(T modifié en T'').*
*Si **isLeaf**(T) alors $T \leftarrow$ arbre vide*
*Sinon Si (**isEmpty**(T_{right}) alors $T \leftarrow T_{left}$*
*Sinon Si (**isEmpty**(T_{left}) alors $T \leftarrow T_{right}$*
*Sinon min $\leftarrow$ **SuppressionMin**(T_{right})*
$T_{data} \leftarrow min$

2.4.3 Troisième Algorithme

Entrée : *T, **arbre binaire de recherche** non-vide.*
Sortie : $\min(T)$ *(T modifié en T''' **arbre binaire de recherche** résultant de la suppression du minimum de T.*

Cas de base : T est une feuille et contient une donnée x, $\min(T) \leftarrow x$ et $T''' \leftarrow$ arbre vide.
Cas général : 3 cas de figure :

a) T est un arbre contenant une donnée x et possédant un sous-arbre **gauche**, T_1, qui est un **arbre binaire de recherche** ; on a donc $\min(T) \leftarrow \min(T_1)$ et $T''' \leftarrow T$ (T sera modifié par l'appel récursif sur T_1).

b) T est un arbre contenant une donnée x et possédant un sous-arbre **droit**, T_2, qui est un **arbre binaire de recherche** ; on a donc $\min(T) \leftarrow x$ et $T''' \leftarrow T_2$

c) T est un arbre contenant une donnée x et possédant **2 fils**, $\rightarrow$ idem que le cas a).

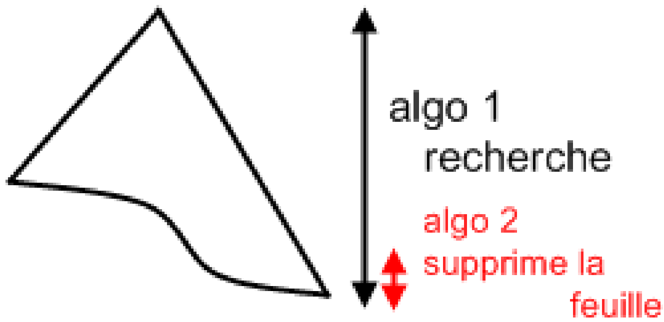
Remarque : *On peut regrouper les cas a) et c) ainsi que le cas de base et le cas b).*

Algorithme **SuppressionMin**(T)
Entrée : T, ***arbre binaire de recherche*** *non-vide.*
Sortie : *Minimum de T (T modifié en T''').*
*Si **isEmpty**(T_{left}) alors min* $\leftarrow T_{data}$
$T \leftarrow T_{right}$
Sinon min $\leftarrow$ ***SuppressionMin***(T_{left})
Retourner min.

2.4.4 Complexité dans le pire des cas

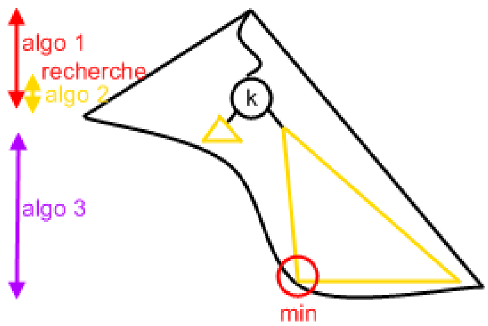
1. Pire des cas où seul le premier algorithme est exécuté.
   Vu que le second algorithme ne doit pas être exécuté, on a que $k \notin T$, on effectue donc une recherche avec échec, et dans le pire des cas la complexité d'un tel algorithme est en $O(h)$ (Vu plus tôt dans le cours).

2. Pire des cas où seuls les 2 premiers algorithmes sont exécuté.



   Du fait que le second algorithme s'exécute, on peut conclure que $k \in T$; le pire des cas sera le cas où on visite le plus de noeuds car le coût est le même pour chaque cas. On considère donc le cas où k est la feuille la plus basse de T. Pour ces 2 algorithmes, on visite h noeuds avec une complexité en $O(1)$ et aucune références vides. Au total, on a donc une complexité dans le pire des cas en $O(h)$.

3. Pire des cas où les 3 algorithmes sont exécutés.



   Dans cette configuration, on a que $k \in T$ et que k possède 2 fils non-vides. Le pire des cas sera lorsque le sous-arbre droit de k possèdera son minimum le plus en bas à gauche possible. Pour ces 3 algorithmes, on visitera h noeuds avec un coup en $O(1)$, ce qui au final reviendra à une complexité en $O(h)$.

Dans tous les cas de figure, l'algorithme a une complexité en $O(h) = O(n)$.

2.5 Tri de données

Il existe une façon de trier des données à l'aide des **arbre binaire de recherche**, la voici :

1. Insérer une à une les données dans un **arbre binaire de recherche** initialement vide.
2. Lire cet arbre de manière infixe.

Complexité

1. Création **arbre binaire de recherche** vide. $\rightarrow \mathbf{O(1)}$
2. Insertions $\rightarrow n * O(h)$ (h est la hauteur courante, elle est majorée par h_{fin} la hauteur de l'arbre final), on a donc comme complexité : $\mathbf{O(n * h_{fin}) = O(n^2)}$.
3. Lecture infixe $\rightarrow \mathbf{O(n)}$.

Au total : $O(1) + O(n^2) + O(n) = \mathbf{O(n^2)}$ pour trier, ce qui n'est pas mieux qu'une liste.

Résumé :

Algorithme	Listes triées	Arbres binaires de recherche
Recherche	$O(n)$	$O(h)$
Insertion	$O(n)$	$O(h)$
Suppression	$O(n)$	$O(h)$
Tri	$O(n^2)$	$O(n * h)$

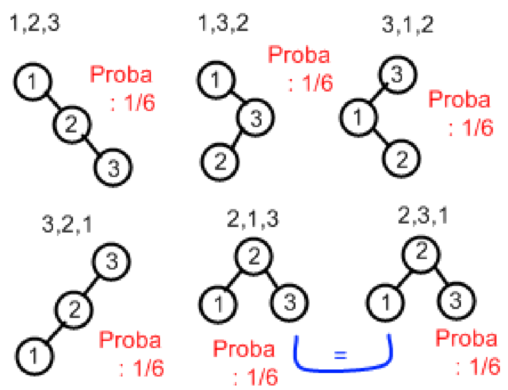
(Au pire, $h = O(n)$)

En pratique, $h \neq O(n)$, on va s'interesser à la complexité en moyenne; on était arrivé à la conclusion qu'en cas de recherche dans des listes triées, que ce soit avec succès ou échec, le nombre de comparaisons en moyenne était de $\frac{n}{2}$. On va maintenant voir que dans le cas des **arbre binaire de recherche**, $h \sim \log_2 n$, ce qui est un résultat positif car on a vu que $\log_2(n+1) \leq h \leq n$ et donc on est plus proche de la borne inférieure que la borne supérieur (en moyenne).
Si on compte le nombre de comparaisons en moyenne, lors d'une recherche avec succès on compte $2\ln(n)$ et lors d'une recherche avec échec $2\ln(n) + 2$. Ce qui est très inférieur au $\frac{n}{2}$ des listes triées !

Hypothèses de travail pour les **arbre binaire de recherche** .

On va considérer n données **différentes**. Lors de l'insertion dans l'arbre binaire, selon l'ordre des données, on peut les insérer de $n!$ façons différentes ce qui implique $n!$ façon de créer un **arbre binaire de recherche**. On émet dès lors l'hypothèse que les probabilités de travailler avec l'un des $n!$ **arbre binaire de recherche** ainsi créés sont toutes les mêmes.

Exemple : $n = 3$



Observation : La probabilité d'avoir k comme racine $= \dfrac{1}{n}$ (car être la racine de l'arbre est équivalent au fait d'être la première donnée parmis les n). En effet, dans le cas où k est en tête de liste, le flot de données peut être vu comme : (k, *une des* $(n-1)!$ *permutations possibles entre les éléments restants*) ; de plus, la probabilité de choisir l'un des arbres est de $\dfrac{1}{n!}$. En multipliant ces 2 probabilités, on obtient la probabilité pour que k soit racine $\rightarrow \dfrac{(n-1)!}{n!} = \dfrac{1}{n}$.