Algorithmique Probabiliste

Philippe Duchon

LaBRI - ENSEIRB-Matméca - Université de Bordeaux

2014-15

Structuration de données

- Un problème général : *comment représenter en mémoire les données d'un programme, de manière à être capable d'opérer efficacement dessus ?*

Structuration de données

- Un problème général : *comment représenter en mémoire les données d'un programme, de manière à être capable d'opérer efficacement dessus ?*
- On va se limiter à un cas simple, la représentation d'un **ensemble fini** S de clés

Structuration de données

- Un problème général : *comment représenter en mémoire les données d'un programme, de manière à être capable d'opérer efficacement dessus ?*
- On va se limiter à un cas simple, la représentation d'un **ensemble fini** S de clés
- **Hypothèse :** $S \subset (\Omega, \leq)$, les clés appartiennent à un "univers" *muni d'un ordre total* (pour deux clés possibles x et y, on sait dire si $x \leq y$ ou non ; on ne fait aucune autre hypothèse sur la nature des éléments de Ω)

Opérations souhaitées

- **Insertion** : à partir de S et $x \notin S$, on veut former $S' = S \cup \{x\}$

Opérations souhaitées

- **Insertion** : à partir de S et $x \notin S$, on veut former $S' = S \cup \{x\}$
- **Recherche** : à partir de S et $x \in \Omega$, on veut savoir si $x \in S$ ou pas (et, souvent, récupérer des données attachées à la clé x si $x \in S$)

Opérations souhaitées

- **Insertion** : à partir de S et $x \notin S$, on veut former $S' = S \cup \{x\}$
- **Recherche** : à partir de S et $x \in \Omega$, on veut savoir si $x \in S$ ou pas (et, souvent, récupérer des données attachées à la clé x si $x \in S$)
- **Suppression** : à partir de S et $x \in S$, former $S' = S - \{x\}$

Opérations souhaitées

- **Insertion** : à partir de S et $x \notin S$, on veut former $S' = S \cup \{x\}$
- **Recherche** : à partir de S et $x \in \Omega$, on veut savoir si $x \in S$ ou pas (et, souvent, récupérer des données attachées à la clé x si $x \in S$)
- **Suppression** : à partir de S et $x \in S$, former $S' = S - \{x\}$
- **Union** : à partir de S et S', former $S \cup S'$

Opérations souhaitées

- **Insertion** : à partir de S et $x \notin S$, on veut former $S' = S \cup \{x\}$
- **Recherche** : à partir de S et $x \in \Omega$, on veut savoir si $x \in S$ ou pas (et, souvent, récupérer des données attachées à la clé x si $x \in S$)
- **Suppression** : à partir de S et $x \in S$, former $S' = S - \{x\}$
- **Union** : à partir de S et S', former $S \cup S'$
- **Séparation :** à partir de S et $x \in \Omega$, former $S_{<x}$ et $S_{>x}$ contenant respectivement les éléments de S plus grands ou plus petits que x

Opérations souhaitées

- **Insertion** : à partir de S et $x \notin S$, on veut former $S' = S \cup \{x\}$
- **Recherche** : à partir de S et $x \in \Omega$, on veut savoir si $x \in S$ ou pas (et, souvent, récupérer des données attachées à la clé x si $x \in S$)
- **Suppression** : à partir de S et $x \in S$, former $S' = S - \{x\}$
- **Union** : à partir de S et S', former $S \cup S'$
- **Séparation :** à partir de S et $x \in \Omega$, former $S_{<x}$ et $S_{>x}$ contenant respectivement les éléments de S plus grands ou plus petits que x
- **Parcours :** passer en revue les éléments de S (dans l'ordre croissant, par exemple ; pas forcément nécessaire)

Les solutions usuelles

- Pour ce problème de structuration des données, on a des solutions classiques, déterministes, variées

Les solutions usuelles

- Pour ce problème de structuration des données, on a des solutions classiques, déterministes, variées
- Dans beaucoup de cas, la représentation d'un même ensemble S n'est pas unique : **elle dépend de la séquence d'opérations qu'on a effectuées**

Les solutions usuelles

- Pour ce problème de structuration des données, on a des solutions classiques, déterministes, variées
- Dans beaucoup de cas, la représentation d'un même ensemble S n'est pas unique : **elle dépend de la séquence d'opérations qu'on a effectuées**
- Dans les cas simples, on peut faire une analyse probabiliste : "si les clés sont insérées une par une, dans un ordre aléatoire uniforme, alors l'objet obtenu est tel que l'opération de recherche dans un ensemble de taille n, s'effectue (très probablement) en temps $O(\log n)$"

Les solutions usuelles

- Pour ce problème de structuration des données, on a des solutions classiques, déterministes, variées
- Dans beaucoup de cas, la représentation d'un même ensemble S n'est pas unique : **elle dépend de la séquence d'opérations qu'on a effectuées**
- Dans les cas simples, on peut faire une analyse probabiliste : "si les clés sont insérées une par une, dans un ordre aléatoire uniforme, alors l'objet obtenu est tel que l'opération de recherche dans un ensemble de taille n, s'effectue (très probablement) en temps $O(\log n)$"
- (mais que se passe-t-il si les clés sont insérées dans un "mauvais" ordre ? que se passe-t-il si on mélange insertions et suppressions ?)

Les solutions usuelles

- Pour ce problème de structuration des données, on a des solutions classiques, déterministes, variées
- Dans beaucoup de cas, la représentation d'un même ensemble S n'est pas unique : **elle dépend de la séquence d'opérations qu'on a effectuées**
- Dans les cas simples, on peut faire une analyse probabiliste : "si les clés sont insérées une par une, dans un ordre aléatoire uniforme, alors l'objet obtenu est tel que l'opération de recherche dans un ensemble de taille n, s'effectue (très probablement) en temps $O(\log n)$"
- (mais que se passe-t-il si les clés sont insérées dans un "mauvais" ordre ? que se passe-t-il si on mélange insertions et suppressions ?)
- Pour assurer de **bonnes complexités**, on est typiquement obligé de *complexifier* les structures et les algorithmes qui travaillent dessus (cas des arbres AVL ou rouges-noirs par rapport aux arbres binaires de recherche)

Structures de données probabilistes

- Une alternative : **probabiliser** (randomiser) les opérations sur les structures de données

Structures de données probabilistes

- Une alternative : **probabiliser** (randomiser) les opérations sur les structures de données
- Les opérations (au moins certaines) sont probabilistes : le résultat d'une insertion ou d'une suppression n'est pas forcément déterministe

Structures de données probabilistes

- Une alternative : **probabiliser** (randomiser) les opérations sur les structures de données
- Les opérations (au moins certaines) sont probabilistes : le résultat d'une insertion ou d'une suppression n'est pas forcément déterministe
- On conserve la simplicité des algorithmes

Structures de données probabilistes

- Une alternative : **probabiliser** (randomiser) les opérations sur les structures de données
- Les opérations (au moins certaines) sont probabilistes : le résultat d'une insertion ou d'une suppression n'est pas forcément déterministe
- On conserve la simplicité des algorithmes
- Dans les bons cas, on a une propriété très séduisante : *quelle que soit la séquence de mises à jour qu'a subies la structure, la **loi de probabilités** de l'objet final ne dépend **que** de l'ensemble des clés présentes (et pas de la séquence d'opérations)*

Structures de données probabilistes

- Une alternative : **probabiliser** (randomiser) les opérations sur les structures de données
- Les opérations (au moins certaines) sont probabilistes : le résultat d'une insertion ou d'une suppression n'est pas forcément déterministe
- On conserve la simplicité des algorithmes
- Dans les bons cas, on a une propriété très séduisante : *quelle que soit la séquence de mises à jour qu'a subies la structure, la **loi de probabilités** de l'objet final ne dépend **que** de l'ensemble des clés présentes (et pas de la séquence d'opérations)*
- Deux exemples : "treaps" (basés sur les ABR) et "skip lists" (listes hiérarchiques)

Treaps

On considère un ensemble $S = \{(x_i, p_i)\}_{1 \leq i \leq n}$ de couples, chacun formé d'une *clé* x et d'une *priorité* p ; on suppose que les clés et les priorités sont toutes distinctes.
Un *treap* (tree-heap) pour S est un *arbre binaire* (plan), dont les n noeuds sont étiquetés uniquement par les couples de S, avec la double condition :

- L'arbre étiqueté uniquement par les clés, est un arbre binaire de recherche (*i.e.* pour tout noeud v, de clé $c(v)$, *chaque clé* présente dans le sous-arbre gauche (droit) de v est inférieure (supérieure) à $c(v)$)
- L'arbre étiqueté uniquement par les priorités, satisfait une *condition de tas* : pour tout noeud v, de priorité $p(v)$, *chaque enfant* de v a une priorité supérieure à $p(v)$ (les priorités sont *croissantes* le long des branches de l'arbre, de la racine vers les feuilles).

Treaps

On considère un ensemble $S = \{(x_i, p_i)\}_{1 \leq i \leq n}$ de couples, chacun formé d'une *clé* x et d'une *priorité* p ; on suppose que les clés et les priorités sont toutes distinctes.
Un *treap* (tree-heap) pour S est un *arbre binaire* (plan), dont les n noeuds sont étiquetés uniquement par les couples de S, avec la double condition :

- L'arbre étiqueté uniquement par les clés, est un arbre binaire de recherche (*i.e.* pour tout noeud v, de clé $c(v)$, *chaque clé* présente dans le sous-arbre gauche (droit) de v est inférieure (supérieure) à $c(v)$)
- L'arbre étiqueté uniquement par les priorités, satisfait une *condition de tas* : pour tout noeud v, de priorité $p(v)$, *chaque enfant* de v a une priorité supérieure à $p(v)$ (les priorités sont *croissantes* le long des branches de l'arbre, de la racine vers les feuilles).
- (on a besoin que clés et priorités appartiennent à deux univers totalement ordonnés, qui peuvent être totalement différents)

Existence des treaps

Pour tout ensemble de couples de clés/priorités S,

- il existe **au moins un** treap : l'ABR obtenu en insérant (par l'algorithme d'insertion dans les feuilles) les clés dans l'ordre croissant des priorités attachées

Existence des treaps

Pour tout ensemble de couples de clés/priorités S,

- il existe **au moins un** treap : l'ABR obtenu en insérant (par l'algorithme d'insertion dans les feuilles) les clés dans l'ordre croissant des priorités attachées
- il existe **au plus un** treap :

Existence des treaps

Pour tout ensemble de couples de clés/priorités S,

- il existe **au moins un** treap : l'ABR obtenu en insérant (par l'algorithme d'insertion dans les feuilles) les clés dans l'ordre croissant des priorités attachées
- il existe **au plus un** treap :
  - la racine ne peut contenir que le couple de plus faible priorité

Existence des treaps

Pour tout ensemble de couples de clés/priorités S,

- il existe **au moins un** treap : l'ABR obtenu en insérant (par l'algorithme d'insertion dans les feuilles) les clés dans l'ordre croissant des priorités attachées
- il existe **au plus un** treap :
  - la racine ne peut contenir que le couple de plus faible priorité
  - le sous-arbre gauche doit forcément contenir exactement les couples dont les clés sont inférieures à la clé de la racine, et le sous-arbre droit, ceux dont les clés sont supérieures

Existence des treaps

Pour tout ensemble de couples de clés/priorités S,

- il existe **au moins un** treap : l'ABR obtenu en insérant (par l'algorithme d'insertion dans les feuilles) les clés dans l'ordre croissant des priorités attachées
- il existe **au plus un** treap :
  - la racine ne peut contenir que le couple de plus faible priorité
  - le sous-arbre gauche doit forcément contenir exactement les couples dont les clés sont inférieures à la clé de la racine, et le sous-arbre droit, ceux dont les clés sont supérieures
  - (fin par induction sur le nombre de couples : les sous-arbres d'un treap sont des treaps)

Existence des treaps

Pour tout ensemble de couples de clés/priorités S,

- il existe **au moins un** treap : l'ABR obtenu en insérant (par l'algorithme d'insertion dans les feuilles) les clés dans l'ordre croissant des priorités attachées
- il existe **au plus un** treap :
  - la racine ne peut contenir que le couple de plus faible priorité
  - le sous-arbre gauche doit forcément contenir exactement les couples dont les clés sont inférieures à la clé de la racine, et le sous-arbre droit, ceux dont les clés sont supérieures
  - (fin par induction sur le nombre de couples : les sous-arbres d'un treap sont des treaps)
- On peut donc parler **du** treap associé à un ensemble S donné.

Opérations sur les treaps

- **Recherche** d'une clé : comme dans un ABR

Opérations sur les treaps

- **Recherche** d'une clé : comme dans un ABR
- **Insertion** d'un couple (x, p) : on part du treap pour S, on veut le treap pour $S \cup \{(x, p)\}$

Opérations sur les treaps

- **Recherche** d'une clé : comme dans un ABR
- **Insertion** d'un couple (x, p) : on part du treap pour S, on veut le treap pour $S \cup \{(x, p)\}$
- **Suppression** d'un couple $(x, p) \in S$: on part du treap pour S, on veut le treap pour $S \backslash \{(x, p)\}$ (dans la pratique, on nous donne x et on souhaite supprimer le couple contenant x comme clé)

Opérations sur les treaps

- **Recherche** d'une clé : comme dans un ABR
- **Insertion** d'un couple (x, p) : on part du treap pour S, on veut le treap pour $S \cup \{(x, p)\}$
- **Suppression** d'un couple $(x, p) \in S$: on part du treap pour S, on veut le treap pour $S \backslash \{(x, p)\}$ (dans la pratique, on nous donne x et on souhaite supprimer le couple contenant x comme clé)
- **Séparation** selon une clé x : on part d'une clé x et d'un treap pour l'ensemble S, on souhaite les treaps pour les sous-ensembles

$$S_< = \{(y, q) \in S : y < x\} \text{ et } S_> = \{(y, q) \in S : y > x\}.$$

Opérations sur les treaps

- **Recherche** d'une clé : comme dans un ABR
- **Insertion** d'un couple (x, p) : on part du treap pour S, on veut le treap pour $S \cup \{(x, p)\}$
- **Suppression** d'un couple $(x, p) \in S$: on part du treap pour S, on veut le treap pour $S \backslash \{(x, p)\}$ (dans la pratique, on nous donne x et on souhaite supprimer le couple contenant x comme clé)
- **Séparation** selon une clé x : on part d'une clé x et d'un treap pour l'ensemble S, on souhaite les treaps pour les sous-ensembles

$$S_< = \{(y, q) \in S : y < x\} \text{ et } S_> = \{(y, q) \in S : y > x\}.$$

- **Fusion** de deux treaps : on part des treaps pour des ensembles S_1 et S_2, on souhaite le treap pour $S = S_1 \cup S_2$; cas particulier où *toutes les clés de* S_1 *sont inférieures à toutes les clés de* S_2

Réparation d'un treap

- Définissons un "presque treap pour S, de défaut $(x, p) \in S$" comme un arbre binaire dont les noeuds sont étiquetés par les couples de S,

Réparation d'un treap

- Définissons un "presque treap pour S, de défaut $(x, p) \in S$" comme un arbre binaire dont les noeuds sont étiquetés par les couples de S,
  - dont les clés forment un ABR

Réparation d'un treap

- Définissons un "presque treap pour S, de défaut $(x, p) \in S$" comme un arbre binaire dont les noeuds sont étiquetés par les couples de S,
  - dont les clés forment un ABR
  - dont *tous les couples de noeuds* (v, v') qui "violent" la condition de tas (v ancêtre de v', avec $p(v) > p(v')$), contiennent **le** noeud (x, p)

Réparation d'un treap

- Définissons un "presque treap pour S, de défaut $(x, p) \in S$" comme un arbre binaire dont les noeuds sont étiquetés par les couples de S,
  - dont les clés forment un ABR
  - dont *tous les couples de noeuds* (v, v') qui "violent" la condition de tas (v ancêtre de v', avec $p(v) > p(v')$), contiennent **le** noeud (x, p)
- On souhaite, à partir de ce presque treap, obtenir **le** treap pour S

Réparation d'un treap

- Définissons un "presque treap pour S, de défaut $(x, p) \in S$" comme un arbre binaire dont les noeuds sont étiquetés par les couples de S,
  - dont les clés forment un ABR
  - dont *tous les couples de noeuds* (v, v') qui "violent" la condition de tas (v ancêtre de v', avec $p(v) > p(v')$), contiennent **le** noeud (x, p)
- On souhaite, à partir de ce presque treap, obtenir **le** treap pour S
- **Remarque :** soit toutes les violations ont (x, p) comme ancêtre v, soit elles ont toutes (x, p) comme descendant v'

Réparation d'un treap

- Définissons un "presque treap pour S, de défaut $(x, p) \in S$" comme un arbre binaire dont les noeuds sont étiquetés par les couples de S,
  - dont les clés forment un ABR
  - dont *tous les couples de noeuds* (v, v') qui "violent" la condition de tas (v ancêtre de v', avec $p(v) > p(v')$), contiennent **le** noeud (x, p)
- On souhaite, à partir de ce presque treap, obtenir **le** treap pour S
- **Remarque :** soit toutes les violations ont (x, p) comme ancêtre v, soit elles ont toutes (x, p) comme descendant v'
- **Ou encore :** on pourrait transformer le "presque treap" en vrai treap, en changeant la priorité du noeud (x, p) (pour une valeur p' comprise entre la priorité du parent et la plus petite priorité des enfants)

Réparation d'un treap (suite)

- *Cas où* (x, p) *a une priorité inférieure à son père :* **tant que** (x, p) **a une priorité inférieure à son père, faire une rotation du père avec le noeud** (x, p)

Réparation d'un treap (suite)

- *Cas où* (x, p) *a une priorité inférieure à son père :* **tant que** (x, p) **a une priorité inférieure à son père, faire une rotation du père avec le noeud** (x, p)
  - Après une telle rotation, soit on a un treap (et c'est fini), soit on a un presque treap de même défaut

Réparation d'un treap (suite)

- *Cas où* (x, p) *a une priorité inférieure à son père :* **tant que** (x, p) **a une priorité inférieure à son père, faire une rotation du père avec le noeud** (x, p)
  - Après une telle rotation, soit on a un treap (et c'est fini), soit on a un presque treap de même défaut
  - À chaque rotation, la profondeur du noeud (x, p) diminue de 1 (donc au maximum, le nombre de rotations est la hauteur de l'arbre de départ)

Réparation d'un treap (suite)

- *Cas où* (x, p) *a une priorité inférieure à son père :* **tant que** (x, p) **a une priorité inférieure à son père, faire une rotation du père avec le noeud** (x, p)
  - Après une telle rotation, soit on a un treap (et c'est fini), soit on a un presque treap de même défaut
  - À chaque rotation, la profondeur du noeud (x, p) diminue de 1 (donc au maximum, le nombre de rotations est la hauteur de l'arbre de départ)
- *Cas où* (x, p) *a une priorité supérieure à au moins un de ses fils :* **tant que** (x, p) **a une priorité supérieure à au moins un de ses fils, faire une rotation de** (x, p) **avec celui de ses fils qui a la plus faible priorité**

Réparation d'un treap (suite)

- *Cas où* (x, p) *a une priorité inférieure à son père :* **tant que** (x, p) **a une priorité inférieure à son père, faire une rotation du père avec le noeud** (x, p)
  - Après une telle rotation, soit on a un treap (et c'est fini), soit on a un presque treap de même défaut
  - À chaque rotation, la profondeur du noeud (x, p) diminue de 1 (donc au maximum, le nombre de rotations est la hauteur de l'arbre de départ)
- *Cas où* (x, p) *a une priorité supérieure à au moins un de ses fils :* **tant que** (x, p) **a une priorité supérieure à au moins un de ses fils, faire une rotation de** (x, p) **avec celui de ses fils qui a la plus faible priorité**
  - Après une telle rotation, soit on a un treap (et c'est fini), soit on a un presque treap de même défaut

Réparation d'un treap (suite)

- *Cas où* (x, p) *a une priorité inférieure à son père :* **tant que** (x, p) **a une priorité inférieure à son père, faire une rotation du père avec le noeud** (x, p)
  - Après une telle rotation, soit on a un treap (et c'est fini), soit on a un presque treap de même défaut
  - À chaque rotation, la profondeur du noeud (x, p) diminue de 1 (donc au maximum, le nombre de rotations est la hauteur de l'arbre de départ)
- *Cas où* (x, p) *a une priorité supérieure à au moins un de ses fils :* **tant que** (x, p) **a une priorité supérieure à au moins un de ses fils, faire une rotation de** (x, p) **avec celui de ses fils qui a la plus faible priorité**
  - Après une telle rotation, soit on a un treap (et c'est fini), soit on a un presque treap de même défaut
  - À chaque rotation, la profondeur du noeud (x, p) augmente de 1 (donc au maximum, le nombre de rotations est la hauteur de l'arbre **final**).

Algorithme d'insertion

- Insérer la clé x dans l'arbre, en tant qu'ABR (en négligeant la priorité) ; on obtient soit un treap, soit un presque treap de défaut (x, p) ;
- Réparer le treap

Algorithme d'insertion

- Insérer la clé x dans l'arbre, en tant qu'ABR (en négligeant la priorité) ; on obtient soit un treap, soit un presque treap de défaut (x, p) ;
- Réparer le treap
- **Complexité :** l'insertion a un coût borné par la hauteur h de l'arbre avant insertion ; la réparation demande un nombre de rotations au plus égal à la profondeur d'insertion (donc au plus $h + 1$)

Algorithme d'insertion

- Insérer la clé x dans l'arbre, en tant qu'ABR (en négligeant la priorité) ; on obtient soit un treap, soit un presque treap de défaut (x, p) ;
- Réparer le treap
- **Complexité :** l'insertion a un coût borné par la hauteur h de l'arbre avant insertion ; la réparation demande un nombre de rotations au plus égal à la profondeur d'insertion (donc au plus $h + 1$)
- **Variante :** lors de la descente pour l'insertion de x, on peut repérer le premier noeud v dont la priorité est supérieure à p ; on remplace alors le noeud v (et son sous-arbre T_v) par le noeud (x, p), avec comme fils les résultats de la séparation de T_v selon la clé x

Algorithme de suppression

L'algorithme de suppression est en quelque sorte le symétrique de celui d'insertion.

- Trouver le couple de clé x dans l'arbre (recherche dans un ABR)
- Passer à $+\infty$ (ou à n'importe quelle valeur supérieure aux priorités) la priorité p, et réparer le treap ; le couple (x, ∞) va devenir une feuille
- Supprimer la feuille en question

Algorithme de suppression

L'algorithme de suppression est en quelque sorte le symétrique de celui d'insertion.

- Trouver le couple de clé x dans l'arbre (recherche dans un ABR)
- Passer à $+\infty$ (ou à n'importe quelle valeur supérieure aux priorités) la priorité p, et réparer le treap ; le couple (x, ∞) va devenir une feuille
- Supprimer la feuille en question
- **Complexité :** le coût total (descente, plus nombre de rotations) est au plus égal à la profondeur de la feuille avant suppression, dont 1 de que la hauteur h' de l'arbre final

Algorithme de suppression

L'algorithme de suppression est en quelque sorte le symétrique de celui d'insertion.

- Trouver le couple de clé x dans l'arbre (recherche dans un ABR)
- Passer à $+\infty$ (ou à n'importe quelle valeur supérieure aux priorités) la priorité p, et réparer le treap ; le couple (x, ∞) va devenir une feuille
- Supprimer la feuille en question
- **Complexité :** le coût total (descente, plus nombre de rotations) est au plus égal à la profondeur de la feuille avant suppression, dont 1 de que la hauteur h' de l'arbre final
- **Variante :** faire la recherche du noeud de clé x, puis remplacer le noeud trouvé par la fusion de ses deux sous-arbres

Séparation et fusion

- **Séparation** selon x : on peut insérer dans le treap actuel le couple $(x, -\infty)$: l'arbre qu'on obtient a pour racine le couple $(x, -\infty)$, les deux sous-arbres sont les treaps issus de la séparation

Séparation et fusion

- **Séparation** selon x : on peut insérer dans le treap actuel le couple $(x, -\infty)$: l'arbre qu'on obtient a pour racine le couple $(x, -\infty)$, les deux sous-arbres sont les treaps issus de la séparation
- **Fusion** (cas où les clés de S_1 sont plus petites que celles de S_2) : inversement, on prend une clé x entre $\max S_1$ et $\min S_2$, on forme un presque treap avec x comme racine, les treaps de S_1 et S_2 comme sous-arbre, et on supprime la clé x

Treaps aléatoires

Par “treap aléatoire pour un ensemble E de clés”, on entend :

- Pour chaque clé $x \in E$, on tire indépendamment une priorité p_x, aléatoire, toutes de même loi continue (par exemple : toutes uniformes sur $[0, 1]$)

Treaps aléatoires

Par “treap aléatoire pour un ensemble E de clés”, on entend :

- Pour chaque clé $x \in E$, on tire indépendamment une priorité p_x, aléatoire, toutes de même loi continue (par exemple : toutes uniformes sur $[0, 1]$)
- (l'ordre relatif des priorités est aléatoire uniforme)

Treaps aléatoires

Par “treap aléatoire pour un ensemble E de clés”, on entend :

- Pour chaque clé $x \in E$, on tire indépendamment une priorité p_x, aléatoire, toutes de même loi continue (par exemple : toutes uniformes sur $[0, 1]$)
- (l'ordre relatif des priorités est aléatoire uniforme)
- On prend le treap des couples (x, p_x)

Treaps aléatoires

Par “treap aléatoire pour un ensemble E de clés”, on entend :

- Pour chaque clé $x \in E$, on tire indépendamment une priorité p_x, aléatoire, toutes de même loi continue (par exemple : toutes uniformes sur $[0, 1]$)
- (l'ordre relatif des priorités est aléatoire uniforme)
- On prend le treap des couples (x, p_x)
- C'est **équivalent** à “numéroter” les clés de 1 à $|E|$ dans un ordre aléatoire uniforme, **à ceci près** qu'on peut ajouter une clé, ou en retirer une, en conservant les priorités

Treaps aléatoires

Par “treap aléatoire pour un ensemble E de clés”, on entend :

- Pour chaque clé $x \in E$, on tire indépendamment une priorité p_x, aléatoire, toutes de même loi continue (par exemple : toutes uniformes sur $[0, 1]$)
- (l'ordre relatif des priorités est aléatoire uniforme)
- On prend le treap des couples (x, p_x)
- C'est **équivalent** à “numéroter” les clés de 1 à $|E|$ dans un ordre aléatoire uniforme, **à ceci près** qu'on peut ajouter une clé, ou en retirer une, en conservant les priorités
- **Dynamiquement :** si l'ensemble E évolue au cours du temps, par insertions et suppressions

Treaps aléatoires

Par “treap aléatoire pour un ensemble E de clés”, on entend :

- Pour chaque clé $x \in E$, on tire indépendamment une priorité p_x, aléatoire, toutes de même loi continue (par exemple : toutes uniformes sur $[0, 1]$)
- (l'ordre relatif des priorités est aléatoire uniforme)
- On prend le treap des couples (x, p_x)
- C'est **équivalent** à “numéroter” les clés de 1 à $|E|$ dans un ordre aléatoire uniforme, **à ceci près** qu'on peut ajouter une clé, ou en retirer une, en conservant les priorités
- **Dynamiquement :** si l'ensemble E évolue au cours du temps, par insertions et suppressions
  - Chaque fois qu'une nouvelle clé x est insérée, on lui tire une priorité p_x

Treaps aléatoires

Par “treap aléatoire pour un ensemble E de clés”, on entend :

- Pour chaque clé $x \in E$, on tire indépendamment une priorité p_x, aléatoire, toutes de même loi continue (par exemple : toutes uniformes sur $[0, 1]$)
- (l'ordre relatif des priorités est aléatoire uniforme)
- On prend le treap des couples (x, p_x)
- C'est **équivalent** à “numéroter” les clés de 1 à $|E|$ dans un ordre aléatoire uniforme, **à ceci près** qu'on peut ajouter une clé, ou en retirer une, en conservant les priorités
- **Dynamiquement :** si l'ensemble E évolue au cours du temps, par insertions et suppressions
  - Chaque fois qu'une nouvelle clé x est insérée, on lui tire une priorité p_x
  - La priorité p_x reste inchangée tant que la clé est présente dans le treap ; si la clé est supprimée, puis réinsérée, on tire une nouvelle priorité (pas besoin de “se souvenir” des priorités des clés supprimées)

Propriété fondamentale des treaps aléatoires

- **Propriété fondamentale** des treaps aléatoires : pour *toute* séquence s d'insertions et de suppressions dans un treap aléatoire initialement vide, la **loi** du treap obtenu ne dépend **que** de l'ensemble $E(s)$ des clés présentes à la fin de la séquence ; c'est aussi la loi de l'ABR qu'on obtiendrait en insérant, par l'algorithme standard, les clés de $E(s)$ dans un ABR initialement vide, dans un ordre aléatoire uniforme.

Propriété fondamentale des treaps aléatoires

- **Propriété fondamentale** des treaps aléatoires : pour *toute* séquence s d'insertions et de suppressions dans un treap aléatoire initialement vide, la **loi** du treap obtenu ne dépend **que** de l'ensemble $E(s)$ des clés présentes à la fin de la séquence ; c'est aussi la loi de l'ABR qu'on obtiendrait en insérant, par l'algorithme standard, les clés de $E(s)$ dans un ABR initialement vide, dans un ordre aléatoire uniforme.
- (c'est aussi la loi de l'arbre d'exécution de **RandQuickSort** sur un tableau contenant les clés de S - en oubliant les priorités)

Propriété fondamentale des treaps aléatoires

- **Propriété fondamentale** des treaps aléatoires : pour *toute* séquence s d'insertions et de suppressions dans un treap aléatoire initialement vide, la **loi** du treap obtenu ne dépend **que** de l'ensemble $E(s)$ des clés présentes à la fin de la séquence ; c'est aussi la loi de l'ABR qu'on obtiendrait en insérant, par l'algorithme standard, les clés de $E(s)$ dans un ABR initialement vide, dans un ordre aléatoire uniforme.
- (c'est aussi la loi de l'arbre d'exécution de **RandQuickSort** sur un tableau contenant les clés de S - en oubliant les priorités)
- **Exemple :**

Propriété fondamentale des treaps aléatoires

- **Propriété fondamentale** des treaps aléatoires : pour *toute* séquence s d'insertions et de suppressions dans un treap aléatoire initialement vide, la **loi** du treap obtenu ne dépend **que** de l'ensemble $E(s)$ des clés présentes à la fin de la séquence ; c'est aussi la loi de l'ABR qu'on obtiendrait en insérant, par l'algorithme standard, les clés de $E(s)$ dans un ABR initialement vide, dans un ordre aléatoire uniforme.
- (c'est aussi la loi de l'arbre d'exécution de **RandQuickSort** sur un tableau contenant les clés de S - en oubliant les priorités)
- **Exemple :**
  - On part d'un treap vide

Propriété fondamentale des treaps aléatoires

- **Propriété fondamentale** des treaps aléatoires : pour *toute* séquence s d'insertions et de suppressions dans un treap aléatoire initialement vide, la **loi** du treap obtenu ne dépend **que** de l'ensemble $E(s)$ des clés présentes à la fin de la séquence ; c'est aussi la loi de l'ABR qu'on obtiendrait en insérant, par l'algorithme standard, les clés de $E(s)$ dans un ABR initialement vide, dans un ordre aléatoire uniforme.
- (c'est aussi la loi de l'arbre d'exécution de **RandQuickSort** sur un tableau contenant les clés de S - en oubliant les priorités)
- **Exemple :**
  - On part d'un treap vide
  - $I(1), I(2), \ldots I(1000)$

Propriété fondamentale des treaps aléatoires

- **Propriété fondamentale** des treaps aléatoires : pour *toute* séquence s d'insertions et de suppressions dans un treap aléatoire initialement vide, la **loi** du treap obtenu ne dépend **que** de l'ensemble $E(s)$ des clés présentes à la fin de la séquence ; c'est aussi la loi de l'ABR qu'on obtiendrait en insérant, par l'algorithme standard, les clés de $E(s)$ dans un ABR initialement vide, dans un ordre aléatoire uniforme.
- (c'est aussi la loi de l'arbre d'exécution de **RandQuickSort** sur un tableau contenant les clés de S - en oubliant les priorités)
- **Exemple :**
  - On part d'un treap vide
  - $I(1), I(2), \ldots I(1000)$
  - $S(100), S(101), \ldots, S(1000)$

Propriété fondamentale des treaps aléatoires

- **Propriété fondamentale** des treaps aléatoires : pour *toute* séquence s d'insertions et de suppressions dans un treap aléatoire initialement vide, la **loi** du treap obtenu ne dépend **que** de l'ensemble $E(s)$ des clés présentes à la fin de la séquence ; c'est aussi la loi de l'ABR qu'on obtiendrait en insérant, par l'algorithme standard, les clés de $E(s)$ dans un ABR initialement vide, dans un ordre aléatoire uniforme.
- (c'est aussi la loi de l'arbre d'exécution de **RandQuickSort** sur un tableau contenant les clés de S - en oubliant les priorités)
- **Exemple :**
  - On part d'un treap vide
  - $I(1), I(2), \ldots I(1000)$
  - $S(100), S(101), \ldots, S(1000)$
  - On a un "treap aléatoire" pour les clés 1 à 100 (la loi est la bonne)

Arbres binaires aléatoires

- La propriété fondamentale des treaps justifie l'étude des "arbres binaires aléatoires" (forme des arbres binaires obtenus par insertion dans un ordre aléatoire)

Arbres binaires aléatoires

- La propriété fondamentale des treaps justifie l'étude des "arbres binaires aléatoires" (forme des arbres binaires obtenus par insertion dans un ordre aléatoire)
- En particulier, au vu des algorithmes d'insertion et de suppression, la statistique *hauteur* des arbres est manifestement importante.

Arbres binaires aléatoires

- La propriété fondamentale des treaps justifie l'étude des "arbres binaires aléatoires" (forme des arbres binaires obtenus par insertion dans un ordre aléatoire)
- En particulier, au vu des algorithmes d'insertion et de suppression, la statistique *hauteur* des arbres est manifestement importante.
- (ça tombe bien, les combinatoristes savent plein de choses sur les arbres binaires aléatoires)

Arbres binaires aléatoires

- La propriété fondamentale des treaps justifie l'étude des "arbres binaires aléatoires" (forme des arbres binaires obtenus par insertion dans un ordre aléatoire)
- En particulier, au vu des algorithmes d'insertion et de suppression, la statistique *hauteur* des arbres est manifestement importante.
- (ça tombe bien, les combinatoristes savent plein de choses sur les arbres binaires aléatoires)
- **Indépendance par rapport aux clés :** étant donnée une forme d'arbre binaire de taille n, et un ensemble de n clés, il y a un seul ABR de cette forme pour cet ensemble de clés

Arbres binaires aléatoires

- La propriété fondamentale des treaps justifie l'étude des "arbres binaires aléatoires" (forme des arbres binaires obtenus par insertion dans un ordre aléatoire)
- En particulier, au vu des algorithmes d'insertion et de suppression, la statistique *hauteur* des arbres est manifestement importante.
- (ça tombe bien, les combinatoristes savent plein de choses sur les arbres binaires aléatoires)
- **Indépendance par rapport aux clés :** étant donnée une forme d'arbre binaire de taille n, et un ensemble de n clés, il y a un seul ABR de cette forme pour cet ensemble de clés
- Pour une **forme d'arbre** T donnée, de taille n, la probabilité qu'un arbre aléatoire ait cette forme, est exactement $p(T) = \mathrm{ins}(T)/n!$, où $\mathrm{ins}(T)$ est le nombre d'ordres d'insertion qui donnent cet arbre

Probabilité d'un arbre

La loi p_n sur les arbres binaires de taille n satisfait les deux conditions suivantes :

- la taille du sous-arbre gauche est *uniforme* sur $[[0, n-1]]$;

Probabilité d'un arbre

La loi p_n sur les arbres binaires de taille n satisfait les deux conditions suivantes :

- la taille du sous-arbre gauche est *uniforme* sur $[[0, n-1]]$;
- conditionnellement au fait que la taille du sous-arbre gauche soit k, les deux sous-arbres sont *indépendants* et *de lois respectives* p_k et p_{n-1-k}

Probabilité d'un arbre

La loi p_n sur les arbres binaires de taille n satisfait les deux conditions suivantes :

- la taille du sous-arbre gauche est *uniforme* sur $[[0, n-1]]$;
- conditionnellement au fait que la taille du sous-arbre gauche soit k, les deux sous-arbres sont *indépendants* et *de lois respectives* p_k et p_{n-1-k}
- De plus, p_n **est la seule loi de probabilités** sur les arbres binaires de taille n qui satisfait ces deux conditions

Probabilité d'un arbre

La loi p_n sur les arbres binaires de taille n satisfait les deux conditions suivantes :

- la taille du sous-arbre gauche est *uniforme* sur $[[0, n-1]]$;
- conditionnellement au fait que la taille du sous-arbre gauche soit k, les deux sous-arbres sont *indépendants* et *de lois respectives* p_k et p_{n-1-k}
- De plus, p_n **est la seule loi de probabilités** sur les arbres binaires de taille n qui satisfait ces deux conditions
- **Autrement dit :** si T a pour sous-arbres T_g et T_d, de tailles respectives k et $n-k-1$, alors

$$p_n(T) = \frac{1}{n} p_k(T_g) p_{n-k-1}(T_d).$$

Probabilité d'un arbre

La loi p_n sur les arbres binaires de taille n satisfait les deux conditions suivantes :

- la taille du sous-arbre gauche est *uniforme* sur $[[0, n-1]]$;
- conditionnellement au fait que la taille du sous-arbre gauche soit k, les deux sous-arbres sont *indépendants* et *de lois respectives* p_k et p_{n-1-k}
- De plus, p_n **est la seule loi de probabilités** sur les arbres binaires de taille n qui satisfait ces deux conditions
- **Autrement dit :** si T a pour sous-arbres T_g et T_d, de tailles respectives k et $n-k-1$, alors

$$p_n(T) = \frac{1}{n} p_k(T_g) p_{n-k-1}(T_d).$$

- **Ou encore :**

$$p_n(T) = \prod_v \frac{1}{t_v(T)},$$

où le produit porte sur les sommets de l'arbre, et $t_v(T)$ désigne la taille du sous-arbre de T de racine v

Profondeur d'un noeud dans un arbre binaire aléatoire

- On considère, dans un ABR aléatoire pour l'ensemble $\{1, \ldots, n\}$, la profondeur $P_{k,n}$ du noeud k (la k-ème plus petite clé) (0 pour la racine)

Profondeur d'un noeud dans un arbre binaire aléatoire

- On considère, dans un ABR aléatoire pour l'ensemble $\{1, \ldots, n\}$, la profondeur $P_{k,n}$ du noeud k (la k-ème plus petite clé) (0 pour la racine)
- On décompose $P_{k,n} = G_{k,n} + D_{k,n}$, où $G_{k,n}$ désigne le nombre d'ancêtres de k, plus grands que k (ceux dont k est un descendant gauche), et $D_{k,n}$ le nombre d'ancêtres de k, plus petits que k (ceux dont k est un descendant droit)

Profondeur d'un noeud dans un arbre binaire aléatoire

- On considère, dans un ABR aléatoire pour l'ensemble $\{1, \ldots, n\}$, la profondeur $P_{k,n}$ du noeud k (la k-ème plus petite clé) (0 pour la racine)
- On décompose $P_{k,n} = G_{k,n} + D_{k,n}$, où $G_{k,n}$ désigne le nombre d'ancêtres de k, plus grands que k (ceux dont k est un descendant gauche), et $D_{k,n}$ le nombre d'ancêtres de k, plus petits que k (ceux dont k est un descendant droit)
- (On a vu dans l'analyse de **RandQuickSort** une formule pour la probabilité que ℓ soit un ancêtre de k)

- **Record** dans une suite $(y_1, \ldots, y_k)$: le i-ème élément est un record si y_i est plus grand que tous les éléments qui le précèdent $y_1, \ldots, y_{i-1}$

- **Record** dans une suite $(y_1, \ldots, y_k)$: le i-ème élément est un record si y_i est plus grand que tous les éléments qui le précèdent $y_1, \ldots, y_{i-1}$
- Pour $i < k$: i est un ancêtre droit de k, si et seulement si p_i est plus petit que $y_{i+1}, \ldots, y_k$, *i.e.* $1 - p_i$ est un record de la séquence $(1 - p_k, 1 - p_{k-1}, \ldots, 1 - p_1)$

- **Record** dans une suite $(y_1, \ldots, y_k)$: le i-ème élément est un record si y_i est plus grand que tous les éléments qui le précèdent $y_1, \ldots, y_{i-1}$
- Pour $i < k$: i est un ancêtre droit de k, si et seulement si p_i est plus petit que $y_{i+1}, \ldots, y_k$, *i.e.* $1 - p_i$ est un record de la séquence $(1 - p_k, 1 - p_{k-1}, \ldots, 1 - p_1)$
- Les $(1 - p_i)$ sont indépendants, uniformes sur $[0, 1]$

- **Record** dans une suite $(y_1, \ldots, y_k)$: le i-ème élément est un record si y_i est plus grand que tous les éléments qui le précèdent $y_1, \ldots, y_{i-1}$
- Pour $i < k$: i est un ancêtre droit de k, si et seulement si p_i est plus petit que $y_{i+1}, \ldots, y_k$, *i.e.* $1 - p_i$ est un record de la séquence $(1 - p_k, 1 - p_{k-1}, \ldots, 1 - p_1)$
- Les $(1 - p_i)$ sont indépendants, uniformes sur $[0, 1]$
- Dans une suite $(X_1, \ldots, X_m)$ de variables indépendantes uniformes sur $[0, 1]$, les événements $\mathcal{E}_i$: "le i-ème élément est un record" sont indépendants, de probabilités respectives $1/i$.

- **Record** dans une suite $(y_1, \ldots, y_k)$: le i-ème élément est un record si y_i est plus grand que tous les éléments qui le précèdent $y_1, \ldots, y_{i-1}$
- Pour $i < k$: i est un ancêtre droit de k, si et seulement si p_i est plus petit que $y_{i+1}, \ldots, y_k$, *i.e.* $1 - p_i$ est un record de la séquence $(1 - p_k, 1 - p_{k-1}, \ldots, 1 - p_1)$
- Les $(1 - p_i)$ sont indépendants, uniformes sur $[0, 1]$
- Dans une suite $(X_1, \ldots, X_m)$ de variables indépendantes uniformes sur $[0, 1]$, les événements $\mathcal{E}_i$: "le i-ème élément est un record" sont indépendants, de probabilités respectives $1/i$.
- **Donc** la profondeur droite $P_{k,n}$ est distribuée comme la somme de $k - 1$ Bernoulli **indépendantes**, de paramètres $1/i$ $(i = 2 \ldots k)$