S54MA2M7 : Informatique 2
Structures de données

Luigi Santocanale
LIS, Aix-Marseille Université

29 janvier 2019

Plan

Structures de données, abstraites et concrètes

Dictionnaires (classe abstraite)

Listes d'association

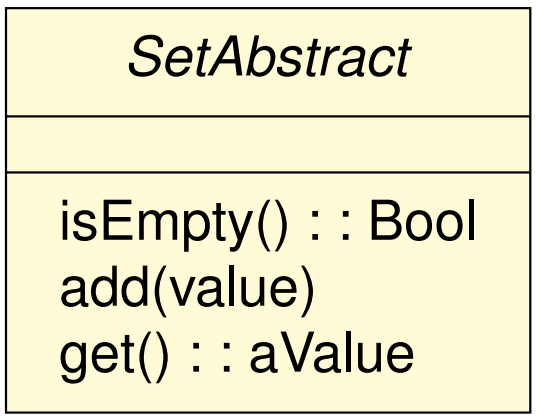
Arbres de recherche binaire

Arbres de recherche binaires equilibrés (AVL trees)

Table d'hachage (chainage linéaire)

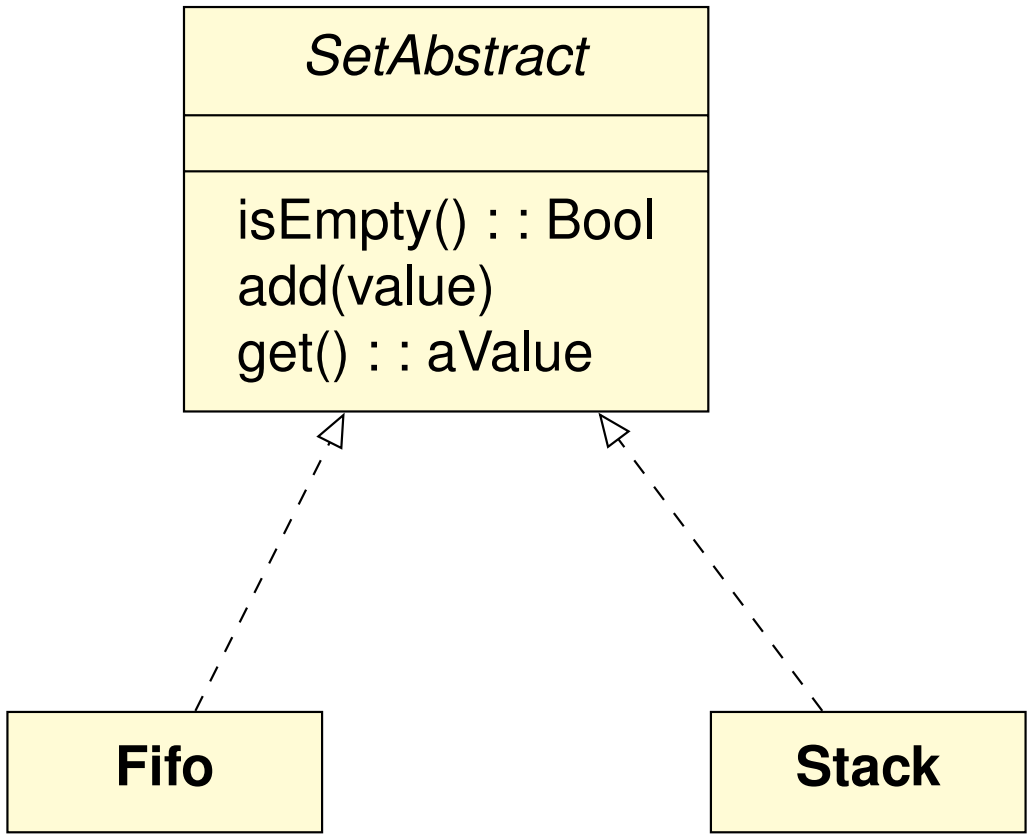
Plan

La structure de donées abstraite `SetAbstract`




```
class SetAbstract():
    def isEmpty(self):
        pass
    def get(self, key):
        pass
    def add(self, key, value):
        pass
```

`Stack` et `Fifo` sont concrets

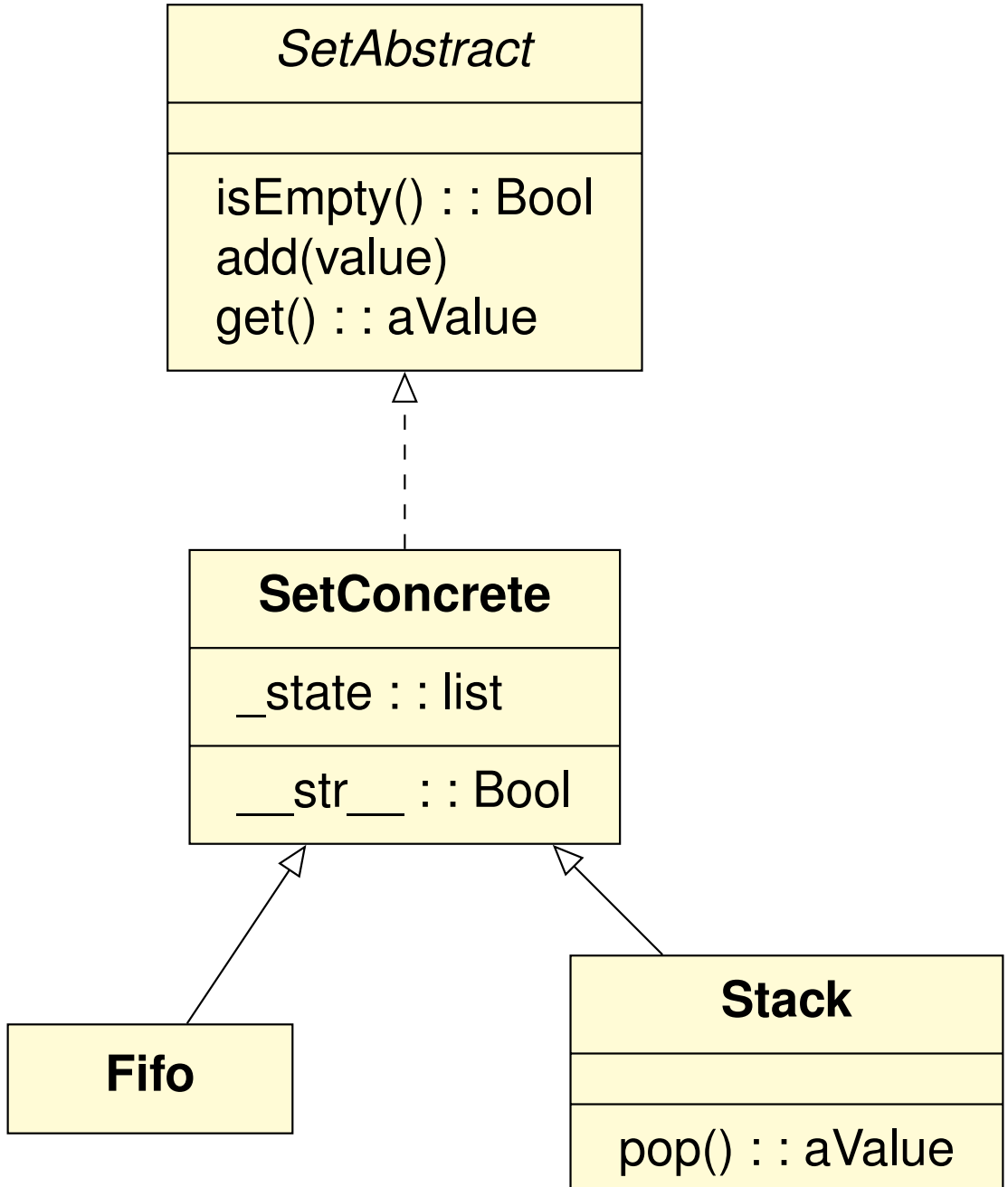


Implémentation en Python

```
from setAbstract import SetAbstract

class Fifo(SetAbstract):
    def __init__(self):
        self.__state = []

    def isEmpty(self):
        return self.__state == []

    def add(self,y):
        self.__state.append(y)
        return self

    def get(self):
        if self.isEmpty():
            raise Exception('Empty
 ↪   Fifo')
        else:
            return
 ↪   self.__state.pop(0)
```

```
from setAbstract import SetAbstract

class Stack(SetAbstract):
    def __init__(self):
        self.__state = []

    def isEmpty(self):
        return self.__state == []

    def add(self,y):
        self.__state.append(y)
        return self

    def get(self):
        if self.isEmpty():
            raise Exception('Empty
 ↪   Stack')
        else:
            return
 ↪   self.__state.pop()
```

Une bonne idée : factosiser le code



Classe intermédiaire

```
from setAbstract import SetAbstract
from random import randint, seed
seed()

class SetConcrete(Set):
    def __init__(self):
        self._state = []
    def __str__(self):
        return self._state.__str__()
    def isEmpty(self):
        return self._state == [] # ou return not self._state
    def add(self,y):
        self._state.append(y)
        return self
    def get(self):
        if self.isEmpty():
            raise Exception('Empty Set')
        else:
            return self._state.pop(randint(0, len(self._state)) -1)
```

Remarque :

- l'attribut `_state` est protégé : un seul (au lieu que 2) tiret du 8
- pas connus hors de l'interface,
  mais connus parmi par les méthodes des classes qui héritent

Deuxième version de `Stack` et `Fifo`

```
from setConcrete import SetConcrete

class Fifo(SetConcrete):
    def get(self):
        if self.isEmpty():
            raise Exception('Empty
 ↪   Fifo')
        else:
            return
 ↪   self._state.pop(0)
```

```
from setConcrete import SetConcrete

class Stack(SetConcrete):
    def get(self):
        if self.isEmpty():
            raise Exception('Empty
 ↪   Stack')
        else:
            return
 ↪   self._state.pop()

    def pop(self):
        return self.pop()
```

Distinction entre `Fifo` et `Stack`

```
from setAbstract import SetAbstract
from setConcrete import SetConcrete
from fifo import Fifo
from stack import Stack

s = SetAbstract()
s.add(1)
print(s)

s = SetConcrete()
s.add(1).add(2).add(3).add(4).get()
print(s)

s = Fifo()
s.add(1).add(2).add(3).add(4).get()
print(s)

s = Stack()
s.add(1).add(2).add(3).add(4).get()
print(s)
```

```
lsantoca$ python code/testSets.py
<set.Set object at 0x108235e80>
[1, 2, 4]
[2, 3, 4]
[1, 2, 3]
```

Si la classe de `s` est `Stack`, alors la loi

$$\texttt{s.add(x).pop()} == \texttt{s}$$

est satisfaite (pour tout `x`).

Structures de données, abstraites v.s. concrètes

Structure de données abstraite, *abstraction* :

- moyen de controller la complexité,
- identifie les aspects les plus importants de la structure,
- détailles omis.
- *Encapsulation :*
  - faire de plusieurs un seul paquet,
  - mettre dedans, cacher les détails,
- Cf. abstraction en maths : algèbre.

Structure de données concrète :

- *utilisable* : donne une implémentation,
- *plusieurs structures* concrète pour (implémentations) pour une seule structure abstraite,
- plusieurs *niveaux d'abstraction*.

Analogies avec les théories (systèmes) algébriques

- Une *théorie algébrique* consiste en
  - un ensemble d'opérations abstraites
  - des axiomes (equiationnels).
- Exemple, théorie algèbrique des groupes :
  - opérations : multiplication, unité, inverse,
  - axiomes : unité, associativité, inverse.

- Une *structure de donnée abstraite* consiste en
  - un ensemble de prototypes d'opérations/méthodes,
  - des contraintes (que l'on pourra exprimer par la logique du I ordre).
- Exemple, structure abstrate de `Pile` :
  - opérations : `isEmpty`, `add`, `get`,
  - contraintes : `s.add(x).pop() == s`

Analogies (suite)

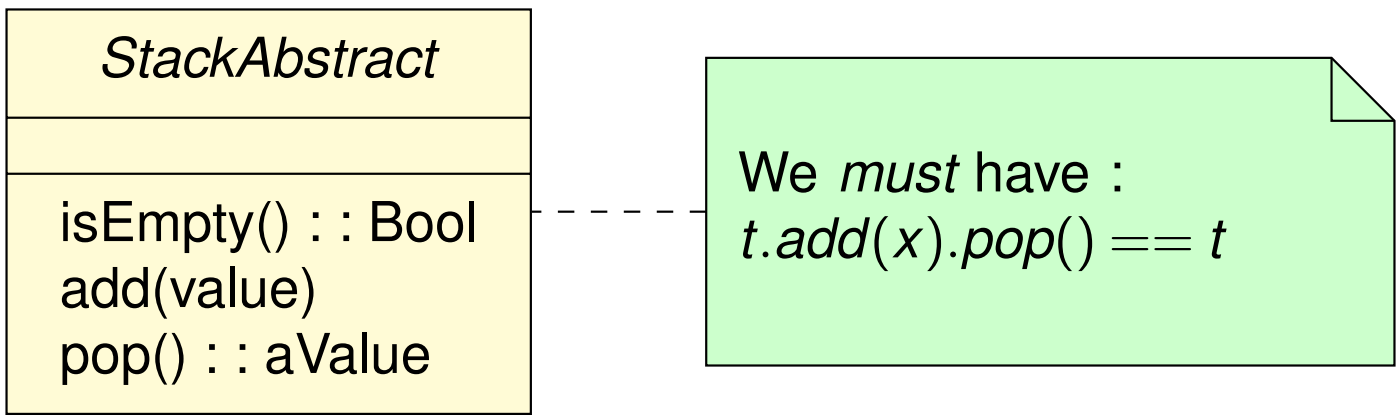
- Un anneau est un groupe commutatif :
  - un anneau a toutes les opérations d'un groupe commutatif, et, en plus, une structure mutlitplicative (t.q. . . .),
  - on ajoute donc, des nouvelles opérations,
  - la classe des anneaux *hérite* depuis celle des groupes commutatifs.

- Un treillis distributif est un treillis modulaire :
  - un treillis distributif satisfait toutes les identités satisfaites par un treillis modulaire
    et, en plus, il satisfait la loi de distributivité de l'inf par rapport au sup,
  - pas de nouvelles équations, mais de nouveau axiomes.
  - la classe des treillis distributifs *hérite* depuis celle des treillis modulaires.

Langages UML et identités algébriques

- Dans le langage UML,
  on spécifie les méthodes d'une classe abstraite,
- on peut spécifier des contraintes par des annotations.

Par exemple :



Chaque algorithme de `parcours` utilisant une structure de données satisfaisant la loi ci-dessus sera considéré un `parcours en profondeur` !!!

En python

Les contraintes peuvent participer à la définition d'une classe abstraite.
Par exemple :

```
import random

class StackAbstract:

    def isEmpty(self):
        pass

    def add(self,y):
        pass

    def get(self):
        pass

    def __eq__(self, other):
        return self.__dict__ == other.__dict__

    def test(self, notrials):
        copy = self.copy()
        for x in range(notrials):
            y = random.randint(1000)
            self.add(y).pop()
            if self != copy:
                return False
        return True
```

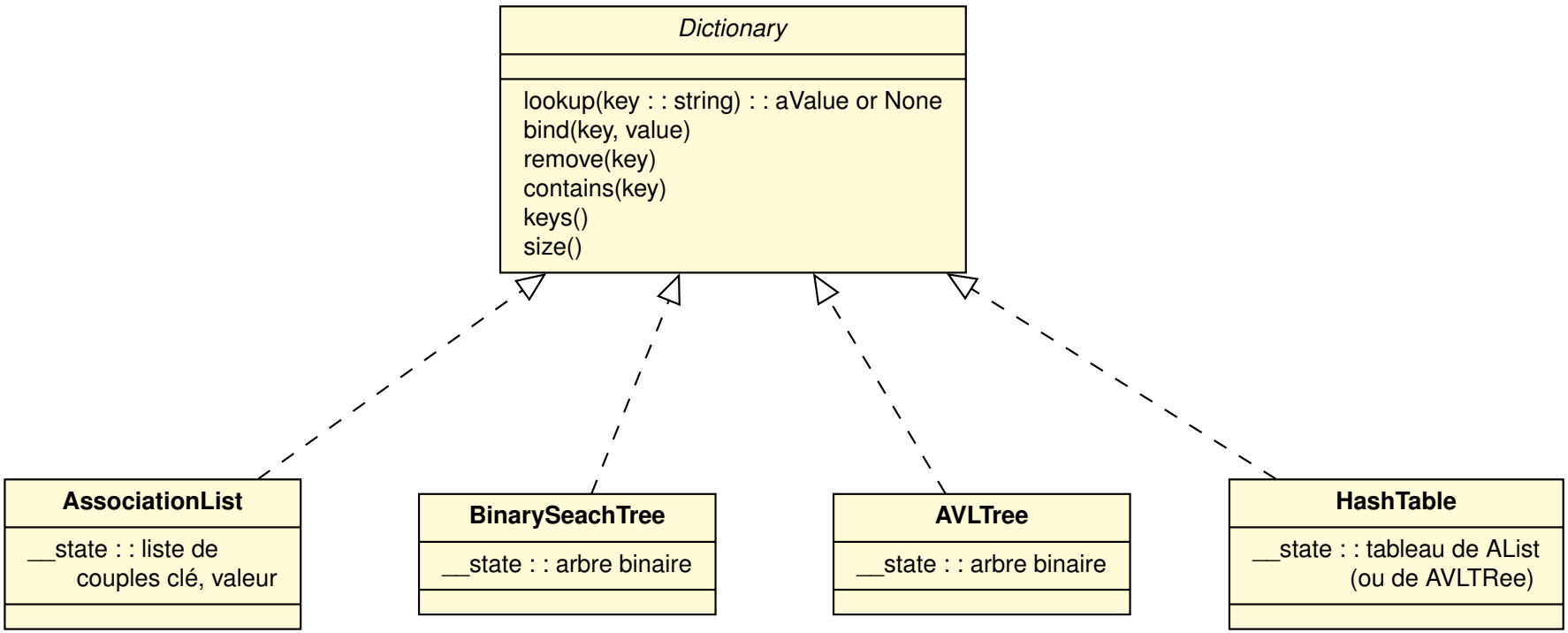
Plan

Comparaison d'implémentation différentes

- En informatique, des paramètres à prendre en compte sont :
  - *correction* d'une implémentation
  - *facilité* d'implémentation
  - *performance* d'une implémention
- Nous allons exemplifier les deux dernier parametres avec des implémentations différentes de la même structure de données abstraite, les *dictionnaires*.
- Que se pass-t'il quand **python** execute des instructions telles que

```
x = mon_dictionnaire['ma_clé'] # lookup
mon_dictionnaire['ma_clé'] = 1 # bind
del mon_dictionnaire['ma_clé'] # remove
'ma_clé' in mon_dictionnaire   # contains
mon_dictionnaire.keys()        # keys
len(mon_dictionnaire)          # size
```

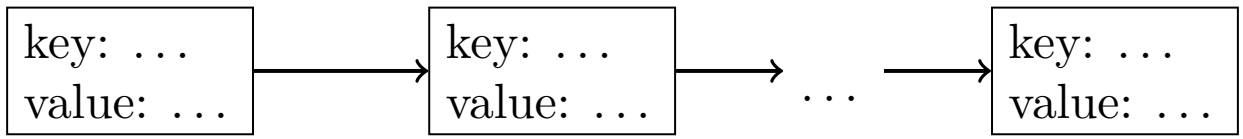
Interface d'un dictionnaire, implémentations proposées



Plan

Liste d'association

- On mantient une liste de couples clé, valeur



- `lookup(needle)` :
  - parcours de la liste en comparant les clés avec `needle`,
  - si `needle` trouvé on retourne sa valeur,
  - si on arrive à la fin de la liste sans rien trouver, on retourne `None`.
  - Hypothèse : la comparaison des clés prend temps $O(1)$.
  - Pire cas : $O(n)$, où n est la longueur de la liste,
  - En moyenne : $O(n)$
- `bind(key, value)` :
  - ajout de `(key, value)` en tête de liste,
  - temps $O(1)$
  - attention duplication possible des clés
- `remove`, `contains` ... : temps $O(n)$

Lookup, calcul du temps moyen pour `lookup`

Hypothèses :

- Les clés $\{1,\ldots,n\}$ sont contenues dans la liste associative.
- La liste d'association ne contient pas de clés repetes, elle a donc longueur n,
- En moyenne, à quelle position de la liste se trouve la clé i (tirée au hasard) ?

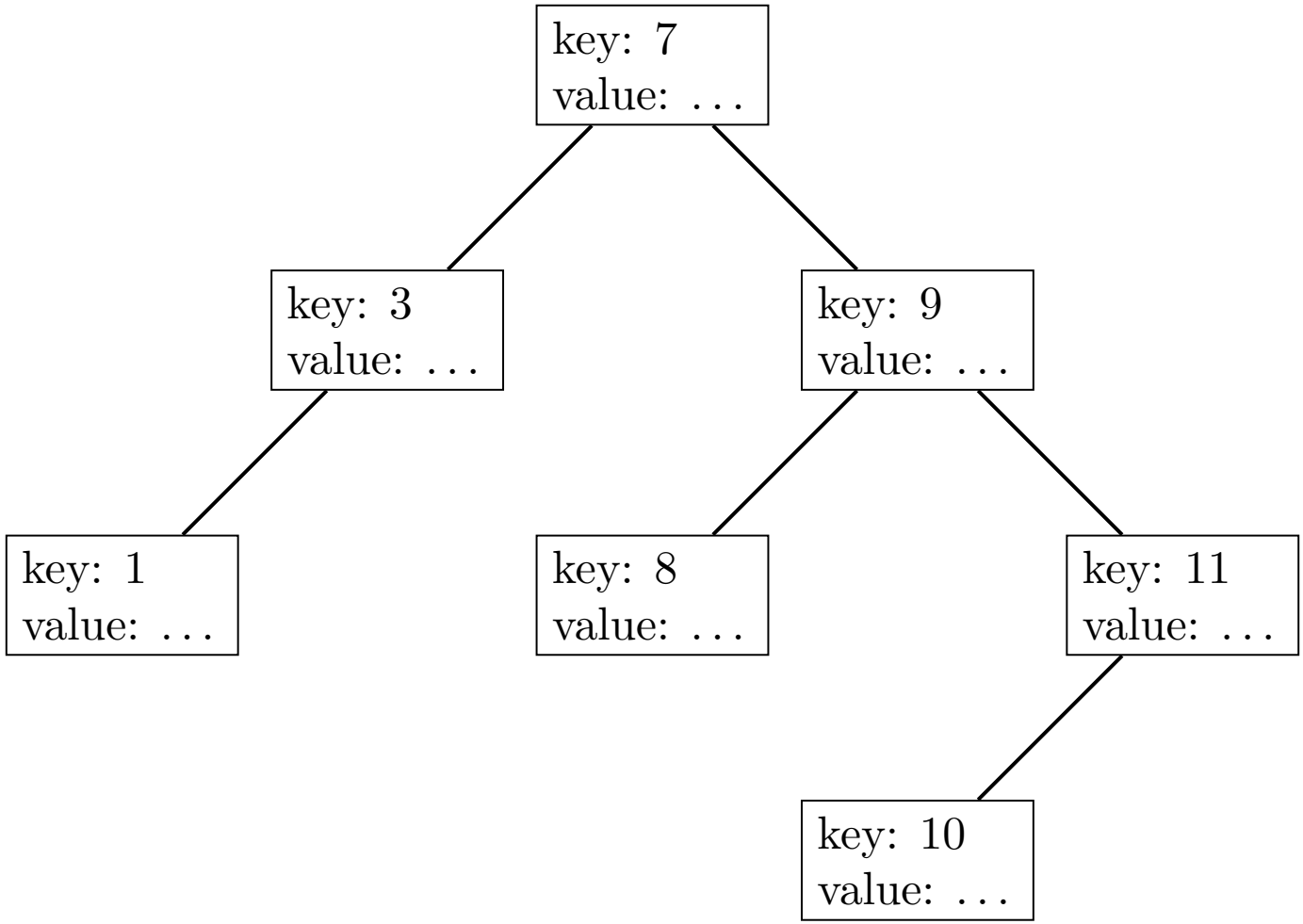
$$
\begin{aligned}
E(pos) &= \sum_{i\in[n]} \sum_{\sigma\in S_n} \sigma^{-1}(i) \frac{1}{n} \frac{1}{n!} \\
&= \sum_{i\in[n]} \sum_{j\in[n]} \sum_{\sigma\in S_n, \sigma^{-1}(i)=j} j \frac{1}{nn!} \\
&= \sum_{j\in[n]} j \frac{1}{nn!} \Big(\sum_{i\in[n]} \sum_{\sigma\in S_n, \sigma^{-1}(i)=j} 1 \Big) \\
&= \sum_{j\in[n]} j \frac{1}{nn!} \Big(\sum_{i\in[n]} \sum_{\sigma\in S_n, \sigma(j)=i} 1 \Big) = \sum_{j\in[n]} j \frac{n!}{nn!} \\
&= \frac{1}{n} \frac{n(n+1)}{2} = \frac{n+1}{2} = O(n)\,.
\end{aligned}
$$

Implémentation en python

Exercice en TP

Plan

Arbre de recherche binaire



Arbre de recherche binaire, définition

Idées :

- Les clés sont totalement ordonnées
- La liste est substituée par un arbre (enraciné) binaire planaire.

Défs :

- Arbre de recherche binaire : chaque noeud a
  - 0 fils (c'est une feuille),
  - 1 fils (à gauche, ou à droite)
  - 2 fils (un à gauche, l'autre à droite)
- Arbre de recherche : arbre de recherche binaire étiqueté (par les clés) tel que, pour tout noeud etiqueté par k :
  - si k_l est un étiquette du sous-arbre gauche, alors $k_l < k$,
  - si k_r est un étiquette du fils droite, alors $k < k_r$,

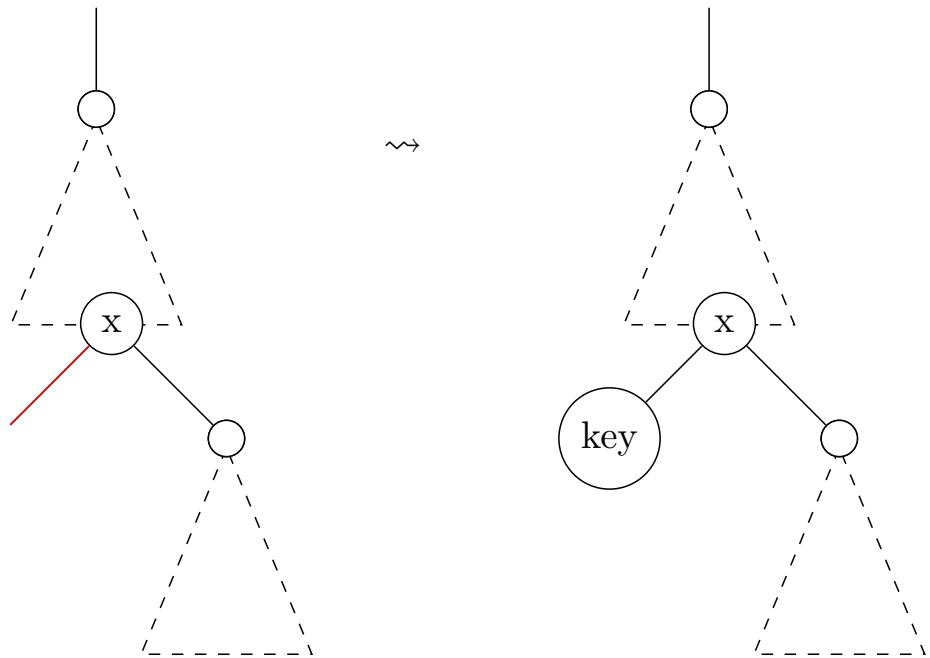
Complexité

- Hypothèse : comparaison des clés en temps $O(1)$.
- Pour chercher une clé il faut parcourir seulement une branche de l'arbre.
- *depth* = max longueur des branches.
- `lookup`, `bind`, `remove` : $O(\mathit{depth})$.
- En moyenne : $O(\log_2 n)$! ! !

Bind/add

`bind(key, value)` :

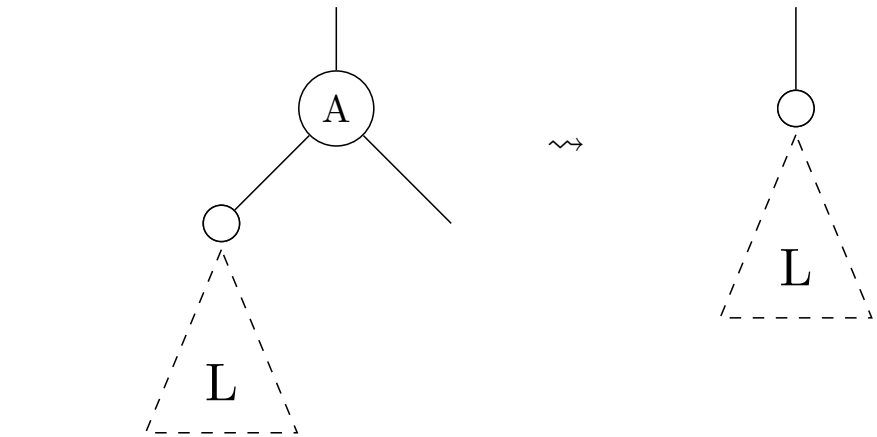
- Si la clé n'est pas trouvée, alors le parcours de branche retourne un noeud feuille sans un fils dans une direction.
- On ajoute une nouveau node dans cette direction,



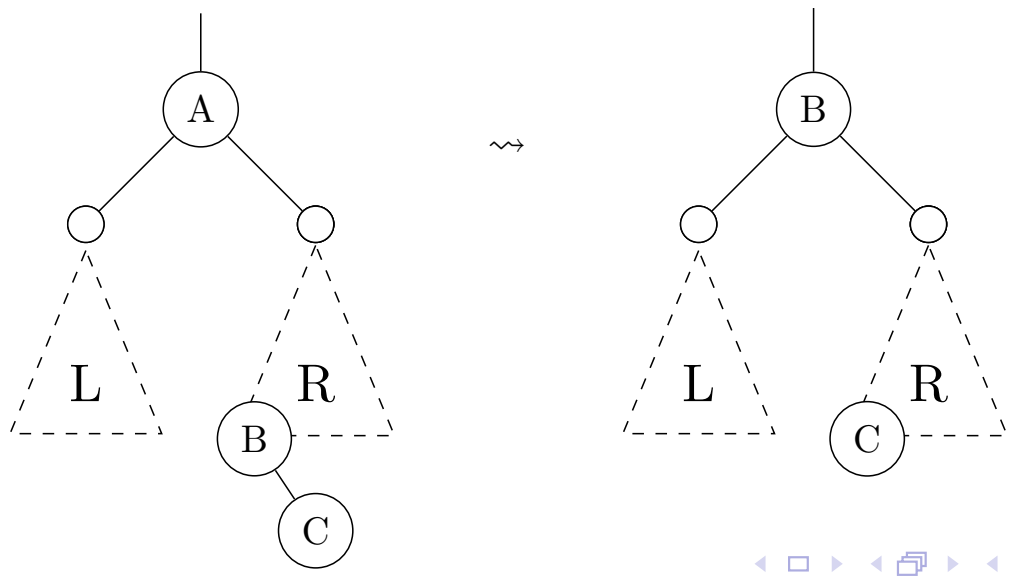
Remove

`Remove(A)` :

- Un fils :



- Deux fils :



Considérations

- Pire cas : $O(n)$
- Car on peut faire des ajouts en résultant avec un arbre avec une seule branche.
- Essayez avec les ajouts successifs des clés $1, 2, 3, 4, 5 \ldots$
- Un tel arbre n'est pas équilibré.

Plan

Arbres de recherche binaire equilibrés

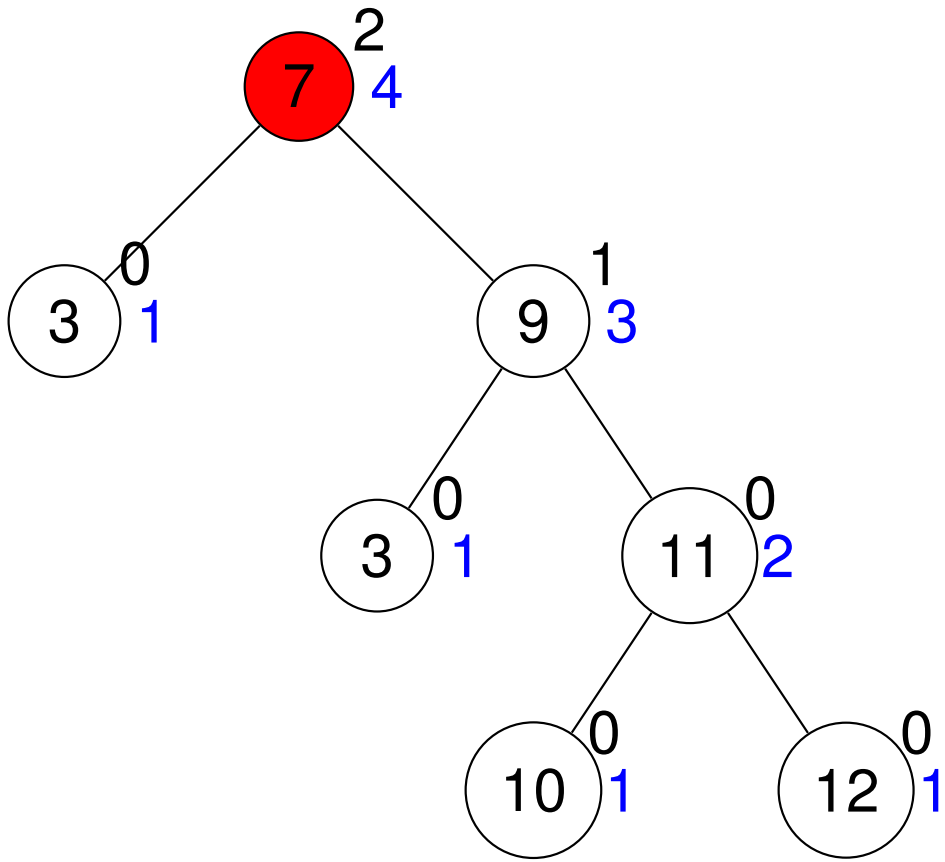
- $Height(v) =$ longueur maximum d'une branche dans le sous arbre enraciné en v.
  - $\ell(v)$ fils gauche de v,
  - $\rho(v)$ fils droit de v.
- Facteur d'équilibrage :

$$FE(v) := Height(\rho(v)) - Height(\ell(v)).$$

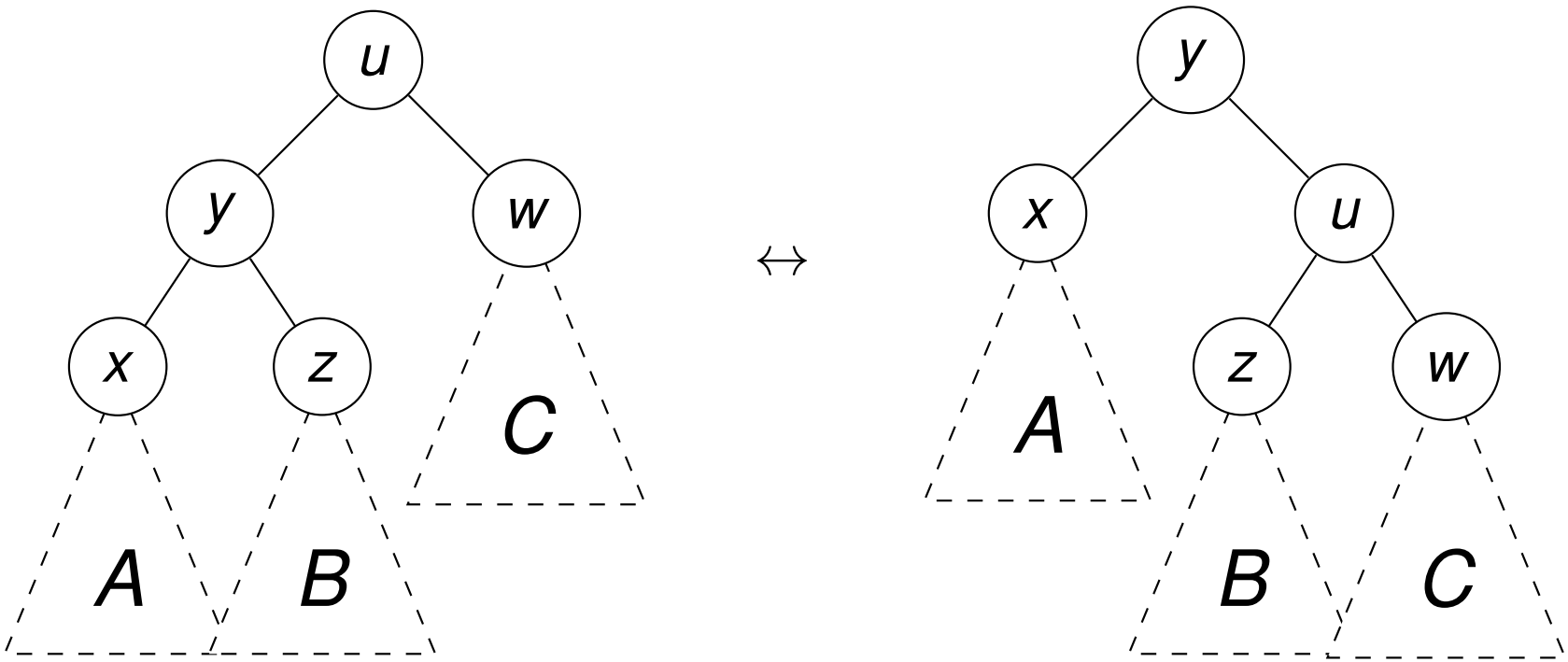
- Un arbre de recherche binaire est equilibré si, pour tour noeud v,

$$FE(v) \in \{-1, 0, 1\}.$$

Un arbre non équilibré

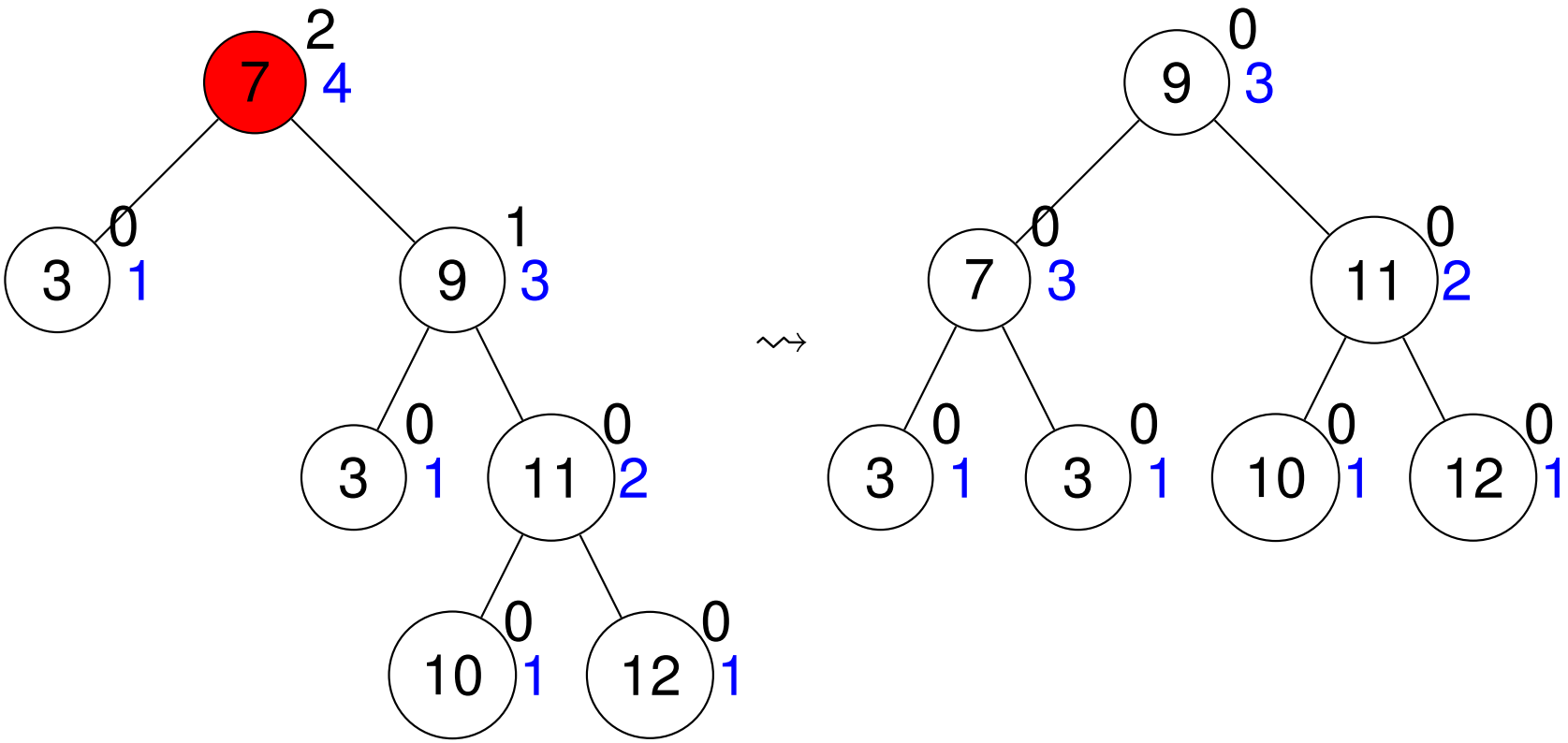


Rotations



(avec $x < y < z < u < w$).

Adjustement



Opérations

Les opérations `lookup`, `bind`, `remove` sont

- comme pour les ARBs,
- suivies par une phase de réequilibrage après :
  - ajout d'une clé,
  - avoir retiré une clé.

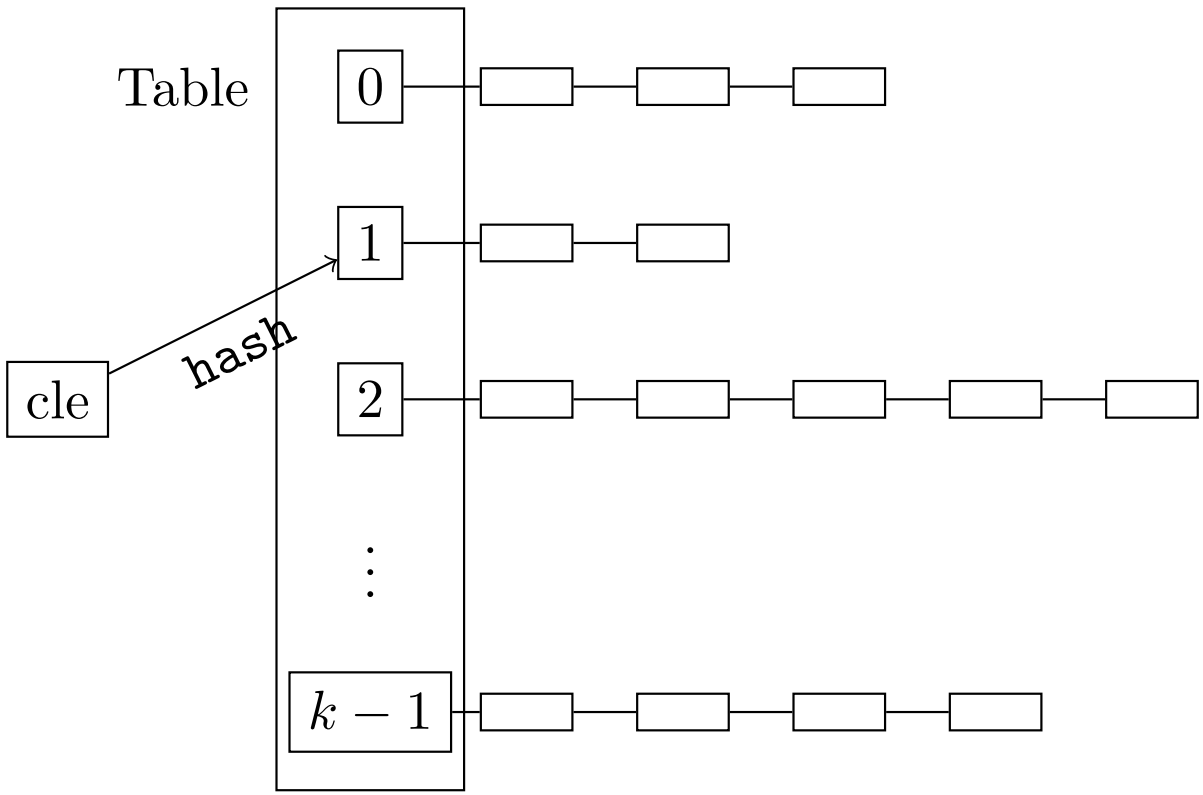
- Les rotations sont utilisées pour rééquilibrer un arbre.

Performance des arbres AVL

- $O(log_2(n))$ au pire cas ! ! !
- Implémentation pratique difficile, “Overhead”.
- Des structures de données similaires/derivées (B-trees, $B+$-trees) utilisées pour les
  - systèmes de gestion des fichiers,
  - systèmes de gestion des bases de données.

Plan

Intuition



Résumé

- On dispose d'un tableau de taille k.
- On pourra, au besoins, doubler la taille du tableau.
- Chaque case du tableau contient (ou pointe à) une liste d'association
  (ou, plus en géneral à un dictionnaire, implémenté n'importe comment).
- On dispose d'une fonction

$$\texttt{hash} : \texttt{Keys} \to \{0, \ldots, k-1\}$$

  où `Keys` est l'ensemble de toutes les clés.
- La fonction `hash` permet de répondre à cette question :
  dans quelle liste on va regarder, ajouter, retirer une clé ?

- La plupart du temps, on a

$$\texttt{hash}(\mathit{key}) = h(\mathit{key}) \bmod k \qquad \text{où} \qquad h : \texttt{Keys} \to \mathbb{N}.$$

Algorithmique

Pour chacun de

- `lookup`
- `put`
- `remove`

faire :

1. calculer $i = \texttt{hash}(cle)$,
2. faire la même opération avec la liste d'association qui se trouve à l'index i du tableau.

Peformance

Hypothèses :

- Calculer `hash`(*cle*) prend temps $O(1)$.
- Accéder à la case i de la table prend temps $O(1)$.
- La distribution de *hash* est uniforme.

Paramètres :

- k taille de la table.
- n numéro de clés présentes dans la table.
- Facteur de charge :

$$fc = \frac{n}{k}$$

  c'est-à-dire, la taille moyenne d'une liste.

Performances (`lookup`, `remove`) :

- Pire cas : $O(n)$.
- En moyenne : $O(fc)$.

Pour `put` ?

Discussion

- Si n est “petit” par rapport à k, alors on considère $fc = \frac{n}{k}$ une constante.
- En pratique, cela donne de temps de `lookp` $O(1)$.
- Au cas le facteur de charge devient trop grand, on peut doubler la taille de la table (et réorganiser les clés).
- Cette opération ne se fera que très rarement (croissance exponentielle de la table, vs. ajout un par un des clés).
- **python** (jsuqu’à la version 3.6) implémente les dictionnaires par table d’hachage avec sondage (probing), voir par exemple ici :

  `https://stackoverflow.com/questions/327311/how-are-pythons-built-in-dictionaries-implemented`
- Une description plus approfondie des techniques de hachages :

  `https://openclassrooms.com/fr/courses/1241676-les-tables-de-hachage`