LINFO 1121

DATA STRUCTURES AND ALGORITHMS

UCLouvain

Bilan 3
Arbres de recherche

Pierre Schaus

Binary Search: Vrai ou faux ?

- Dans le meilleur des cas, le nombre de comparaisons entre clefs pour une recherche binaire d'une clef particulière dans un tableau trié de N clefs distinctes est ~log(N) ?

Vrai ou faux ?

- Dans le meilleur des cas, le nombre de comparaisons entre clefs pour une recherche binaire d'une clef particulière dans un tableau trié de N clefs distinctes est ~log(N) ?

- Faux: Cela correspond au pire-cas, dans le meilleur des cas on tombe directement sur la clef donc O(1).

Rappel: algorithmes pour traverser un arbre

```
visiterPréfixe(Arbre A) :
    visiter (A)
    Si nonVide (gauche(A))
          visiterPréfixe(gauche(A))
    Si nonVide (droite(A))
          visiterPréfixe(droite(A))
```

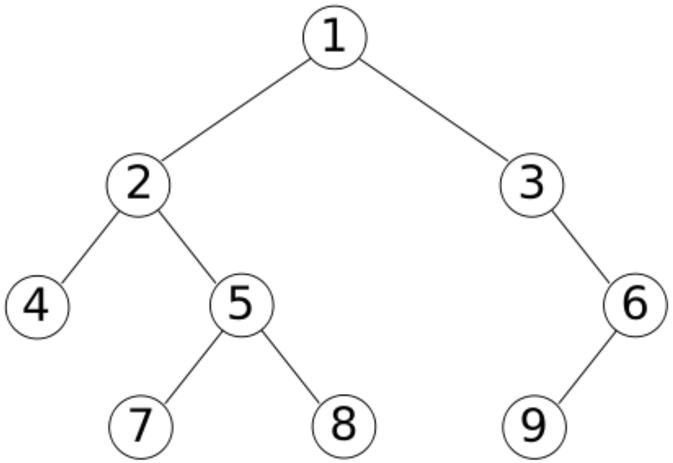
1, 2, 4, 5, 7, 8, 3, 6, 9

```
visiterPostfixe(Arbre A) :
    Si nonVide(gauche(A))
       visiterPostfixe(gauche(A))
    Si nonVide(droite(A))
       visiterPostfixe(droite(A))
    visiter(A)
```

4, 7, 8, 5, 2, 9, 6, 3, 1

```
visiterInfixe(Arbre A) :
    Si nonVide(gauche(A))
       visiterInfixe(gauche(A))
    visiter(A)
    Si nonVide(droite(A))
       visiterInfixe(droite(A))
```

4, 2, 7, 5, 8, 1, 3, 9, 6



!!Not a BST!!

https://fr.wikipedia.org/wiki/Arbre_binaire#Parcours_préfixe

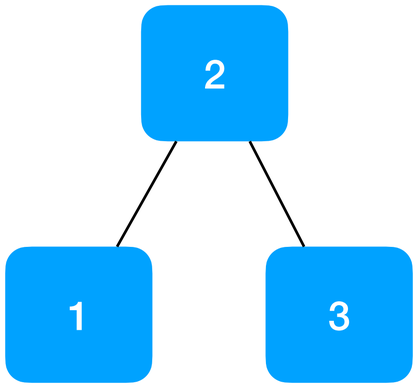
BST Vrai ou Faux ?

- Étant donné un parcours infixe d'un BST contenant N clefs distinctes.
  - Est-il possible de reconstruire la forme du BST sur base du résultat du parcours ?
  - Si oui, écrivez le pseudo-code d'un algorithme pour le faire, si non, donnez un contre-exemple qui justifie votre réponse

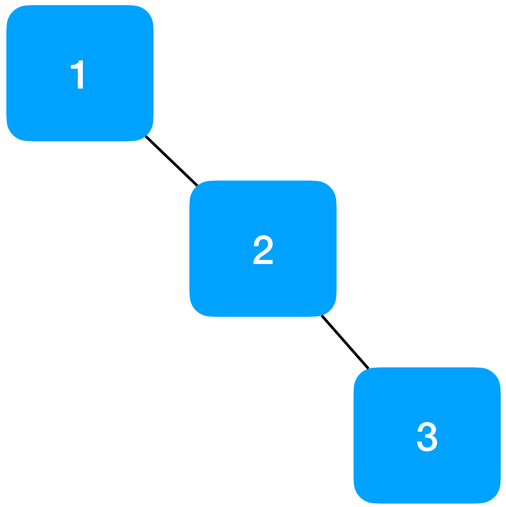
Contre exemple

- infixe(T1) = infixe(T2) = 1-2-3 et T1 != T2

- T1



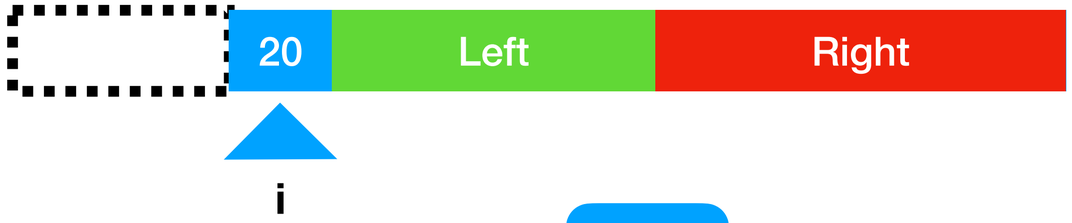
- T2

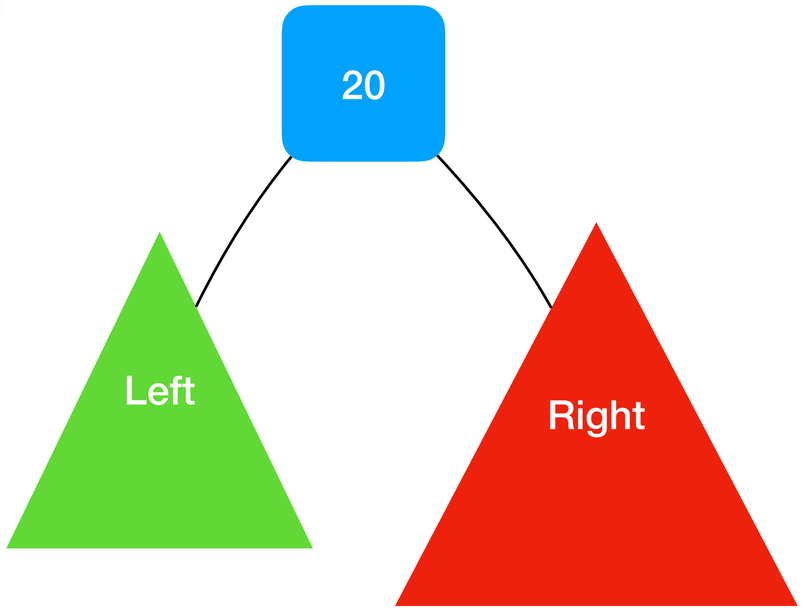


Vrai ou Faux ?

- Étant donné un parcours préfixe d'un BST contenant N clefs distinctes.
  - Est-il possible de reconstruire la forme du BST sur base du résultat du parcours?
  - Si oui, écrivez le pseudo-code d'un algorithme pour le faire, si non, donnez un contre-exemple qui justifie votre réponse.

Idée de l'algorithme



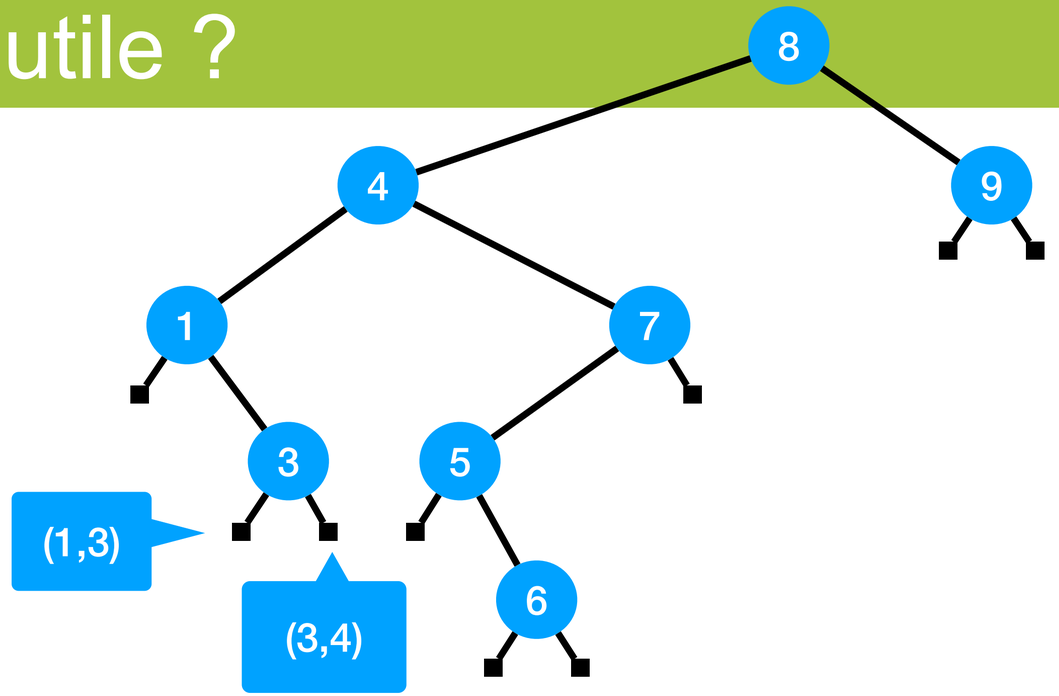


Node[Integer]

Node left
Node right
Integer key
Int size

```
private Node readBSTFromPreorder(int [] input, int i, …)  {
  Node left = readBSTFromPreorder(input,i+1,..);
  Node right = readBSTFromPreorder(input,i+left.size+1,..);
  return new Node(input[i],left,right,left.size+right.size+1);
}
```

Pourquoi est-ce un résultat utile ?



- Et pour les noeuds feuille ?

8,4,1,3,7,5,6,9

- Lorsque i =3, on va lancer récursivement au départ de i=4 (key=7) la lecture de l'arbre de gauche:
  - Pourquoi 7 ne peut-il pas être un enfant droit de 3 sachant que le préfixe 8,4,1,3 a déjà été lu ?
  - Car on s'attend à une clef entre 1 et 3 sur la gauche, et entre 3 et 4 sur la droite

Algo: Recursive Procedure

```
private static Node readBSTFromPreorder(int [] input) {
   return readBSTFromPreorder(input,0,Integer.MIN_VALUE, Integer.MAX_VALUE);
}
```

Lecture du plus long BST possible enraciné au noeud input[i] ayant toutes ses clefs comprises entre min et max

@param input = les clefs d'un BST ordonnées selon un parcours preordre
@param i = une position dans input
@param min la valeur de clef minimum
@param max la valeur de clef maximum

```
public Node preorderRead(int [] preOrderInput, int i, int min, int max) {
    if (i >= preOrderInput.length) return null;
    if (preOrderInput[i] > max || preOrderInput[i] < min) return null;
    Node left = preorderRead(preOrderInput, i + 1, min, preOrderInput[i]);
    int leftSize = left != null ? left.size : 0;
    Node right = preorderRead(preOrderInput, i + 1 + leftSize, preOrderInput[i], max);
    return new Node(left, right, preOrderInput[i]);
}
```

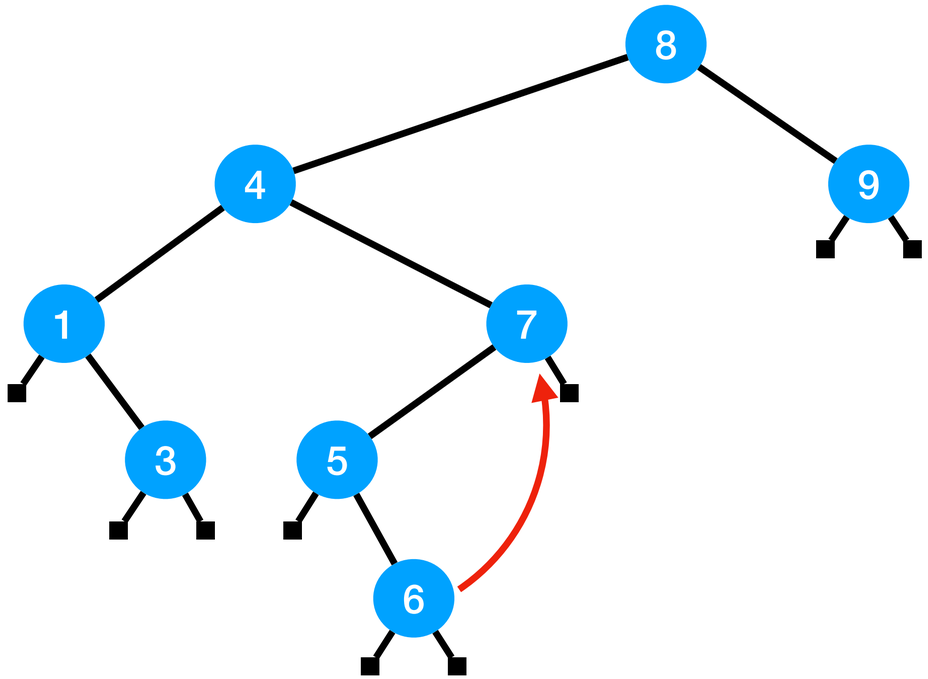
What about this solution ?

```
public PreorderToBST(int [] preOrderInput) {
    Node head = new Node(null,null,preOrderInput[0]);
    for (int e: preOrderInput) {
        put(head, e);
    }
    return head;
}
```

What is the time complexity for this input:

[1,2,3,4,…,n] ?

- Étant donné un arbre ordonné de N clefs distinctes et une clef x, est-il possible de trouver la plus petite clef strictement plus grande que x en temps logarithmique dans le pire cas ?



- Étant donné un arbre ordonné de N clefs distinctes et une clef x, est-il possible de trouver la plus petite clef strictement plus grande que x en temps logarithmique dans le pire cas ?

- FAUX. La hauteur attendue d'un BST résultant de l'insertion de N clefs distinctes dans un ordre aléatoire dans un arbre initialement vide est en moyenne logarithmique mais ici rien ne dit que les clefs ont été insérées dans un ordre aléatoire, ni que l’arbre est équilibré. Donc le O(h) peut être égal à O(n) sans autre précision,

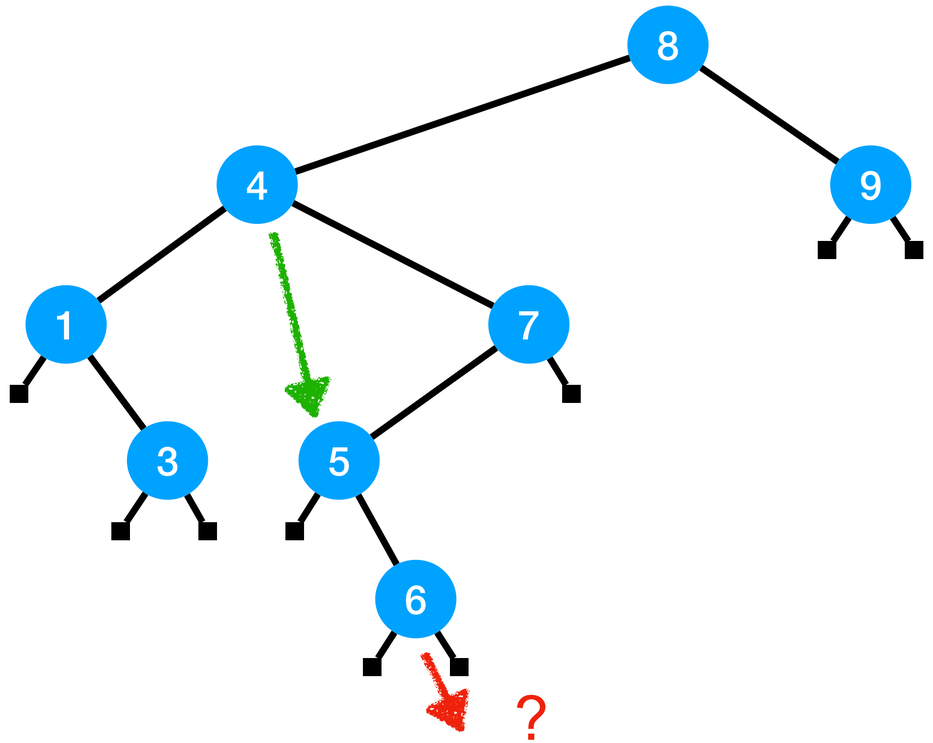
Frai Faux ?

- Soit x un noeud dans un BST.

- Le successeur de x (le noeud contenant la clef suivante dans l'ordre croissant) est le noeud le plus à gauche dans l'arbre de droite de x ?

- Soit x un noeud dans un BST.

- Le successeur de x (le noeud contenant la clef suivante dans l'ordre croissant) est le noeud le plus à gauche dans l'arbre de droite de x ? Faux! Successeur de 6.



Vrai-Faux

- La hauteur max d'un 2-3 tree avec N clefs est ~$\log_3 N$?

Vrai-Faux

- La hauteur max d'un 2-3 tree avec N clefs est ~$\log_3 N$

- La hauteur d'un 2-3 tree est entre

  - $\lfloor \log_3(N) \rfloor \leq h(2-3\text{tree}) \leq \lfloor \log_2(N) \rfloor$

    All 3 nodes

    All 2 nodes

  - C'est donc FAUX, ~$\log_3 N$ est la hauteur min d'un 2-3 tree. La notation ~ tient compte de la constante multiplicative et donc ~$\log_3 N$ n'est pas ~$\log_2 N$

  - $\log_2 N = \log_3(N)/\log_3(2)$ (pour passer d'un à l'autre on introduit une constante).

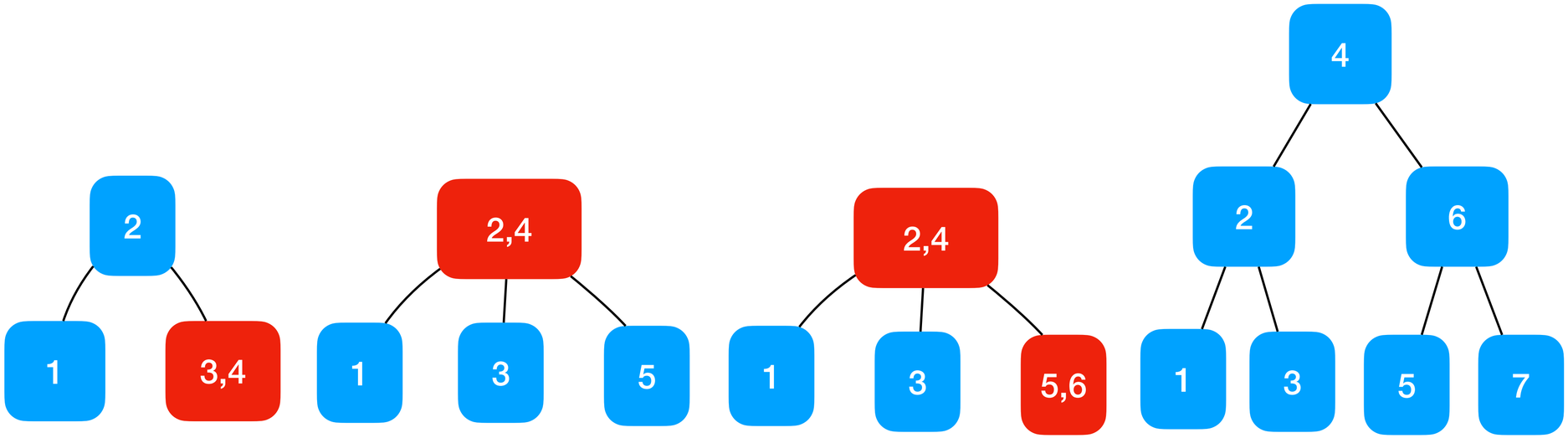
  - **Par contre, hauteur max d'un 2-3 tree avec N clefs est $\Theta(\log_3(N))$**

- Pour l'insertion de N clefs dans l'ordre croissant dans un red-black BST initialement vide. Le nombre de changements de couleurs de la dernière insertion est au plus 3. Le nombre de changements s'entend comme la somme des différences en valeur absolue entre le nombre de rouges après insertion moins le nombre de rouges avant insertion.

- *On traduit la question en arbre 2-3*: Pour l'insertion de N clefs dans l'ordre croissant dans un arbre 2-3 initialement vide. Lors de la dernière insertion, j’ai supprimé au plus 3 3-nodes.

Vérifions

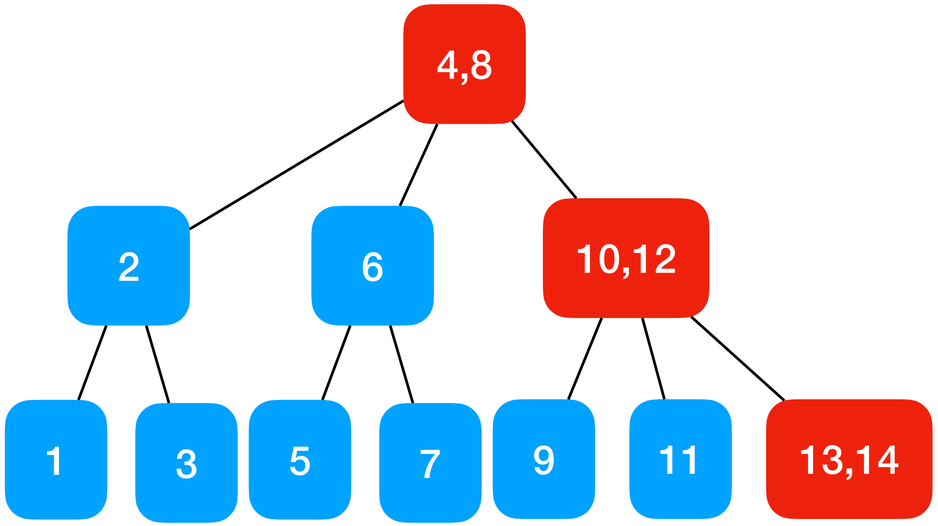
- On insère 1,2,3,4,5,…

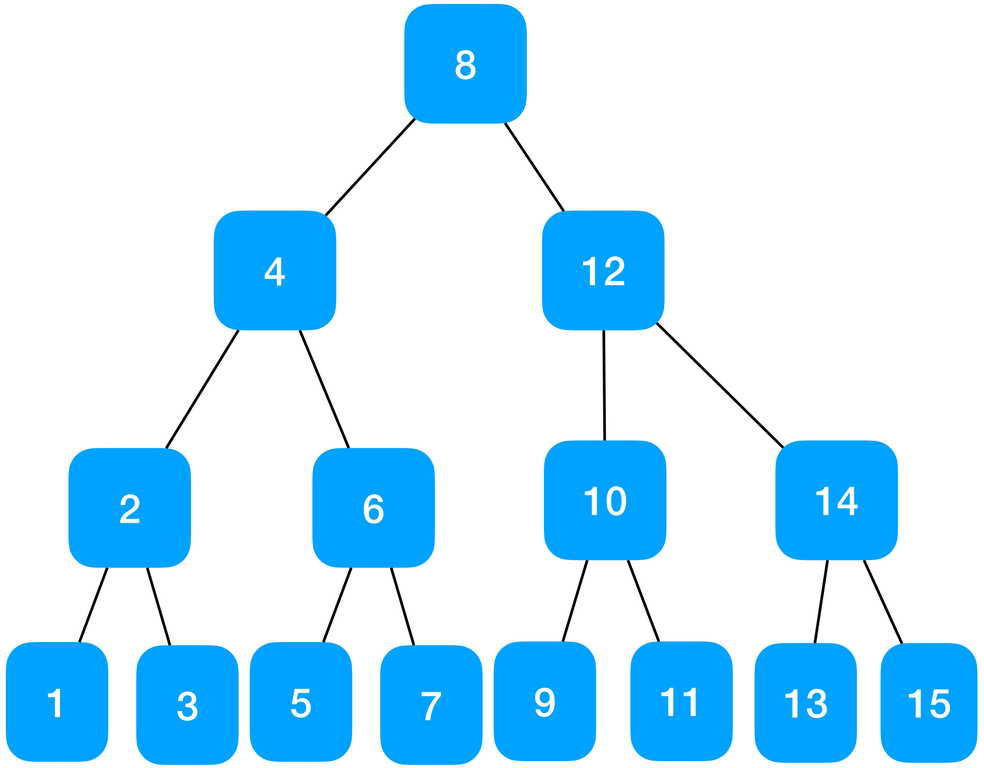


- Essayons de trouver un contre exemple:

- On insère 1,2,3,4,5,6,**7 cette dernière insertion supprime 2 noeuds 2-3. A l'insertion de 15, le nombre de changements sera 3, car la branche de droite seront toutes saturées en 3 nodes et la "bulle" doit remonter jusqu'à la racine.**

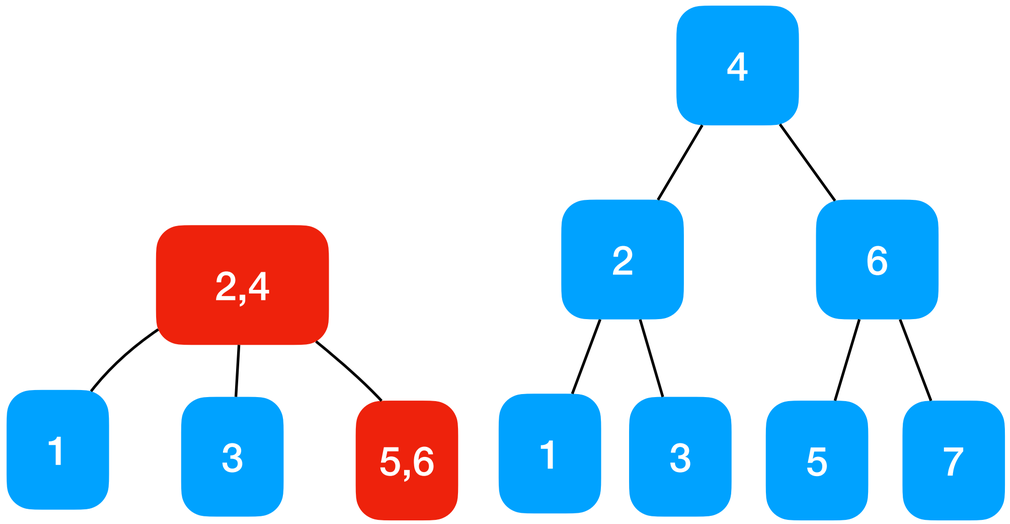
- Après 1,2,3,4,5,6,7,8,9,10,11,12,13,14





- Un red-black BST obtenu après insertion de N > 1 clefs dans un arbre initialement vide possède au moins un lien rouge.

- Traduction: Il y a au moins un 3-node dans un arbre 2-3 ?



Frai/Faux ?

- Dans un red-black BST de N noeuds, la hauteur noire (i.e. le nombre de liens noirs de chaque chemin depuis la racine vers un lien null) est maximum log N ?

- Traduction: la hauteur d'un arbre 2-3 est maximum log N.
  - Oui puisque
  - $\lfloor \log_3(N) \rfloor \leq h(2-3\text{tree}) \leq \lfloor \log_2(N) \rfloor$

Trier avec un BST 🤔

- Imaginez un algorithme de tri utilisant un BST. A quoi ressemblerait cet algorithme ? Quelle est la complexité ?

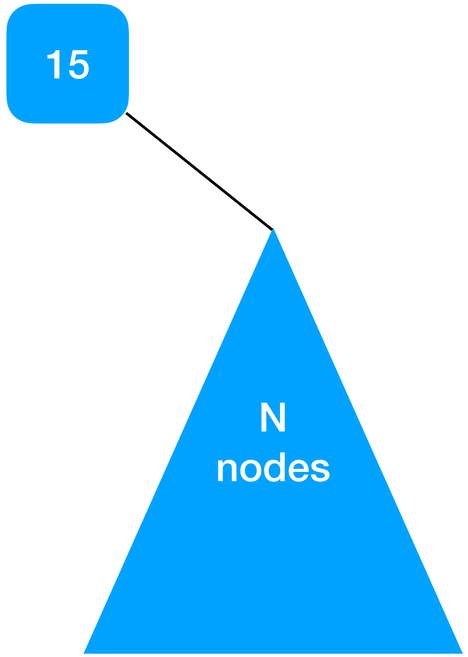
Trier avec un Red-Black Tree ? 🤔

- Quelle serait la complexité de votre algorithme si le BST est remplacé par un red-black BST ?

  - BST: O(n^2) pour la construction du BST car l'insertion prend O(n), ensuite O(n) pour faire le parcours infixe => O(n^2) en tout.

  - red-black: O(n log(n)) pour la construction du red-black car l'insertion prend O(log(n)). Ensuite O(n) pour faire le parcours infixe => O(n log(n)) en tout

Complexity pour supprimer la valeur 15 ?

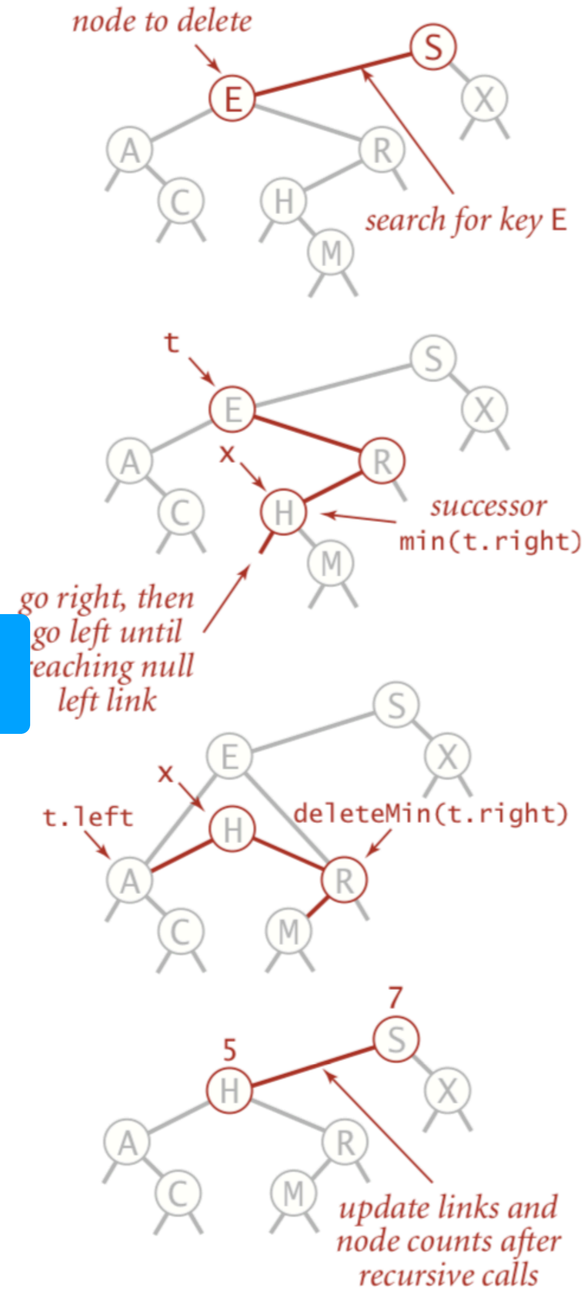


Rappel sur le delete

```
public void delete(Key key)
{  root = delete(root, key);  }

private Node delete(Node x, Key key)
{
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
   if      (cmp < 0) x.left  = delete(x.left,  key);
   else if (cmp > 0) x.right = delete(x.right, key);
   else
   {
      if (x.right == null) return x.left;
      if (x.left == null) return x.right;
      Node t = x;
      x = min(t.right);  // See page 407.
      x.right = deleteMin(t.right);
      x.left = t.left;
   }
   x.N = size(x.left) + size(x.right) + 1;
   return x;
}
```

O(1)

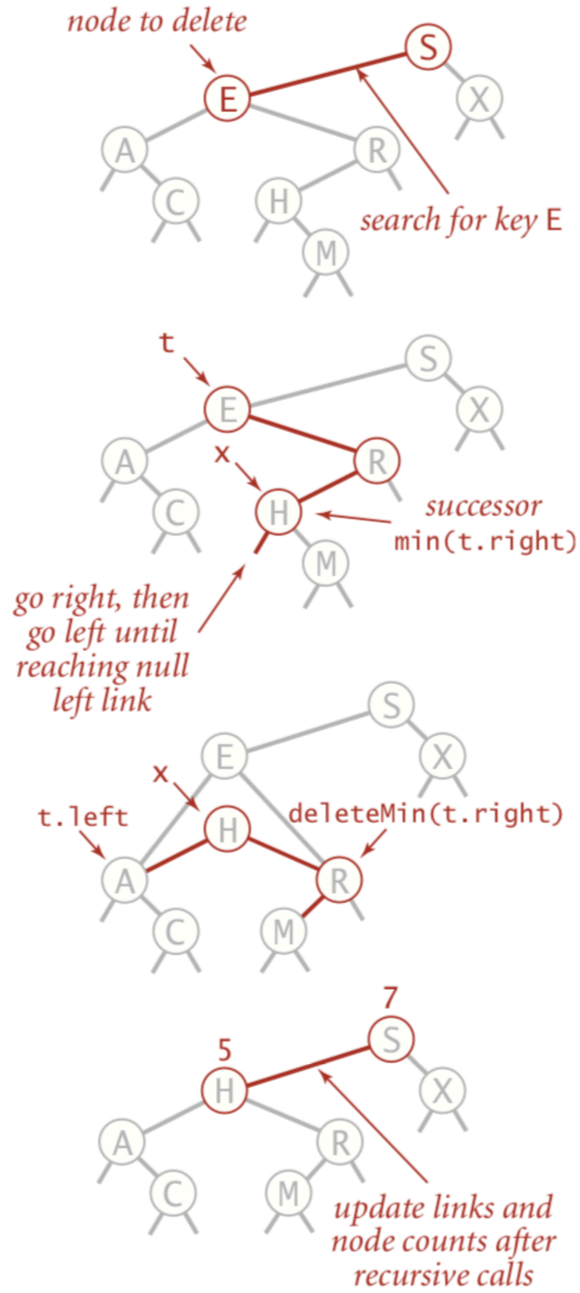


Rappel sur le delete

```
public void delete(Key key)
{  root = delete(root, key);  }

private Node delete(Node x, Key key)
{
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
   if      (cmp < 0) x.left  = delete(x.left,  key);
   else if (cmp > 0) x.right = delete(x.right, key);
   else
   {
      if (x.right == null) return x.left;
      if (x.left == null) return x.right;
      Node t = x;
      x = min(t.right);  // See page 407.
      x.right = deleteMin(t.right);
      x.left = t.left;
   }
   x.N = size(x.left) + size(x.right) + 1;
   return x;
}
```

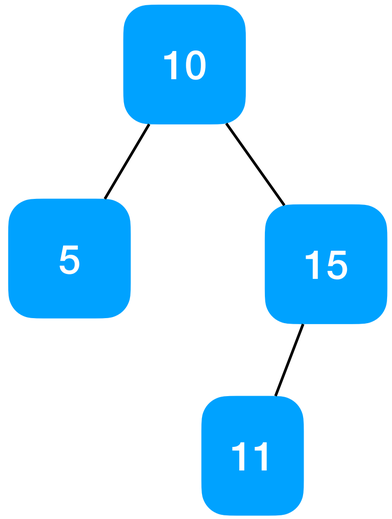
Successeur

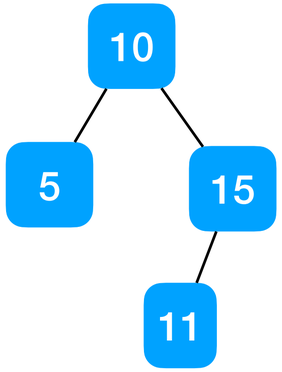


- Est-ce que l'opération de suppression dans un BST est "commutative" ? C'est à dire que supprimer x et ensuite directement y d'un BST (tel qu'implémenté dans le livre) laisse l'arbre dans même état que si on avait d'abord supprimé y et puis x ?

- Donner un contre-exemple ou argumenter pourquoi c'est effectivement toujours le cas.

- Pour vous aider, considérez l'arbre suivant et les opérations de suppression de 5 et 10.



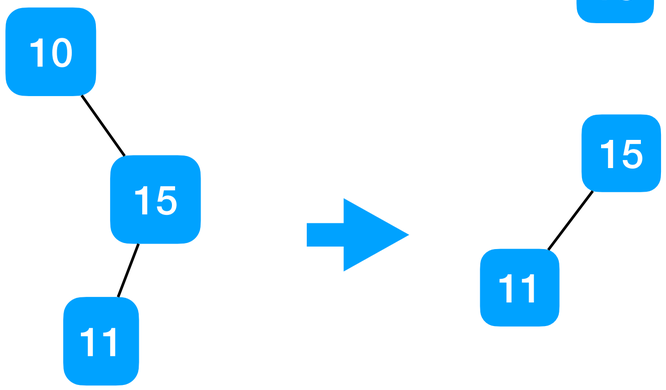



```
public void delete(Key key)
{  root = delete(root, key);  }

private Node delete(Node x, Key key)
{
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
   if      (cmp < 0) x.left  = delete(x.left,  key);
   else if (cmp > 0) x.right = delete(x.right, key);
   else
   {
      if (x.right == null) return x.left;
      if (x.left == null) return x.right;
      Node t = x;
      x = min(t.right);  // See page 407.
      x.right = deleteMin(t.right);
      x.left = t.left;
   }
   x.N = size(x.left) + size(x.right) + 1;
   return x;
}
```

- Delete 10 et puis 5

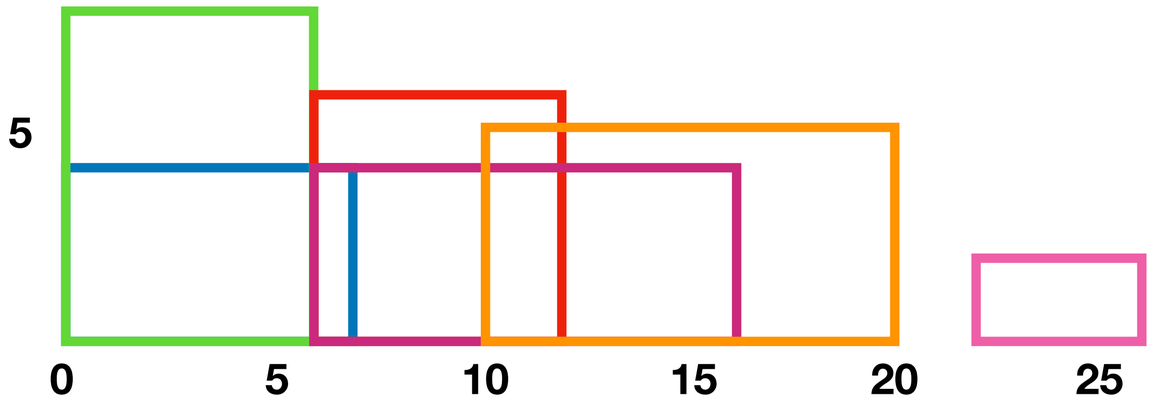
- Delete 5 et puis 10



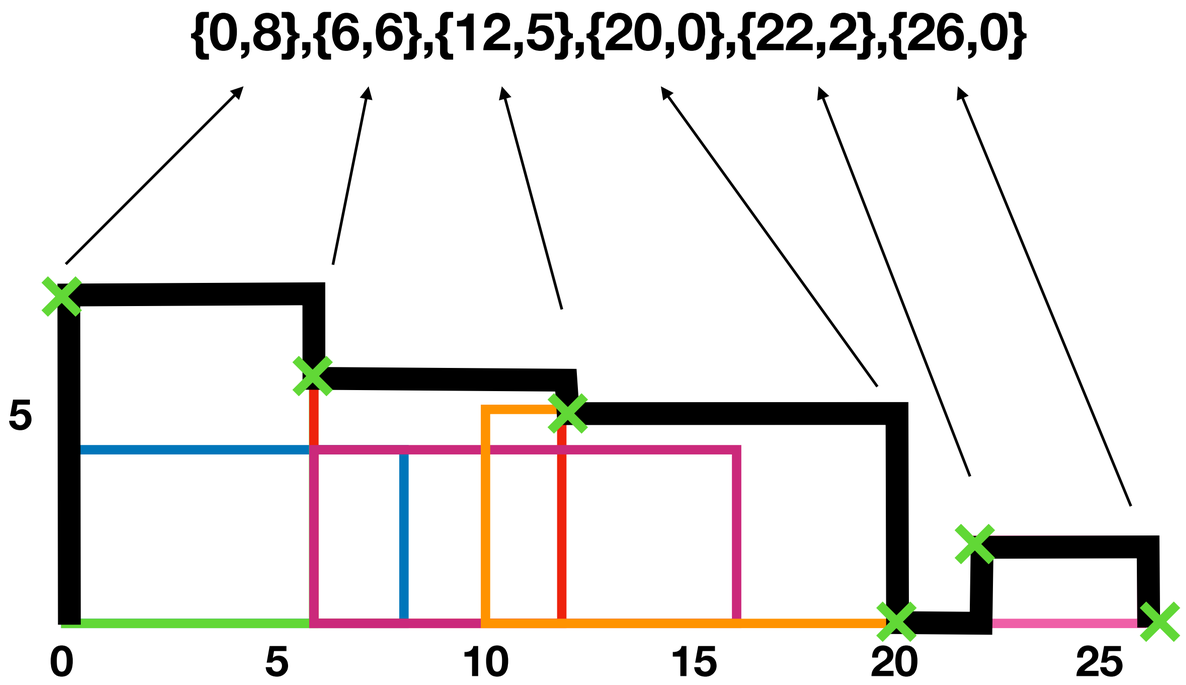
Skyline

Sklyline: Input

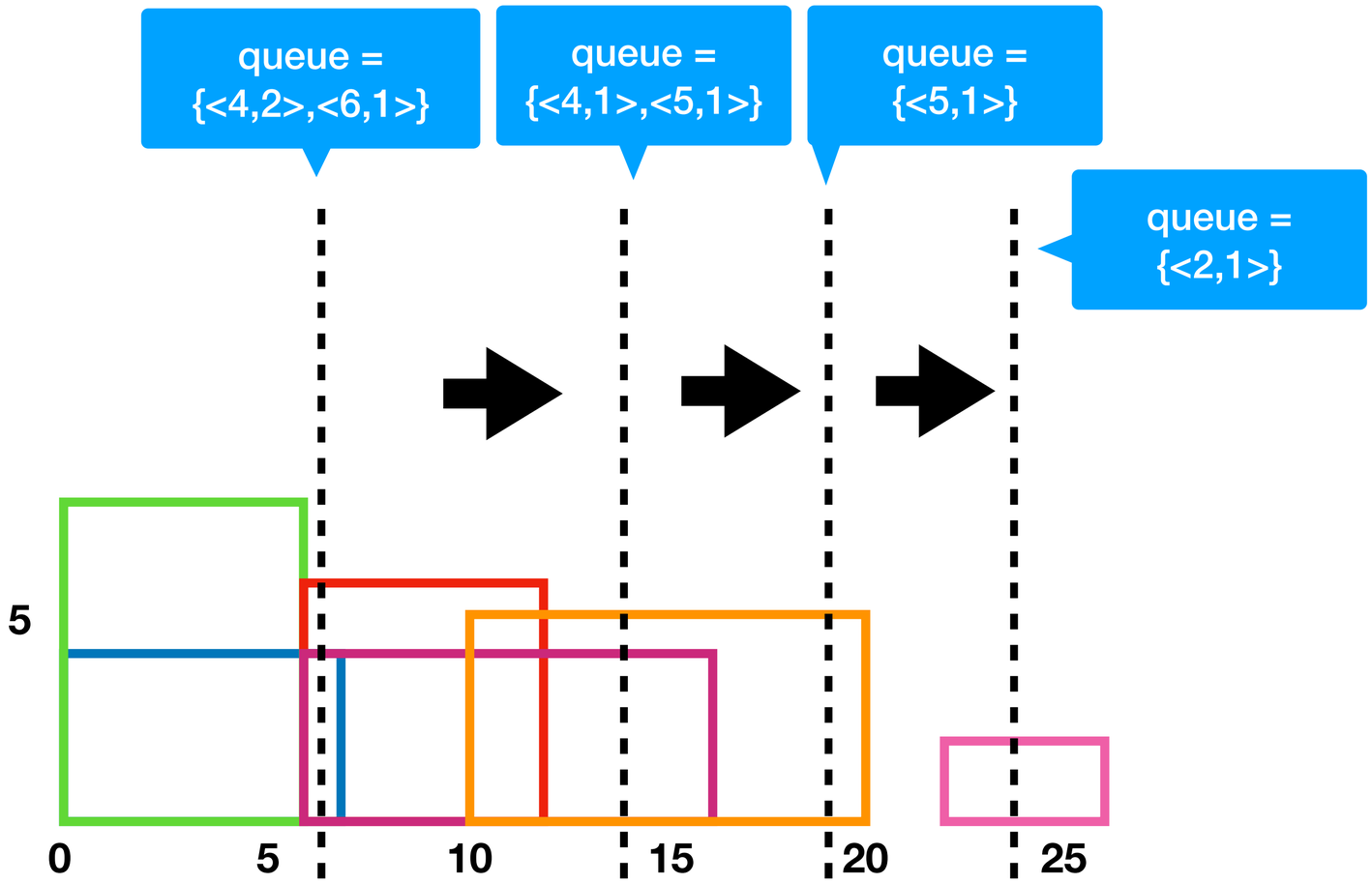
{0,4,7},{0,8,6},{6,6,12},{6,4,16},{10,5,20},{22,2,26}



Sklyline: Output



Sweep Line Algorithm



Observation: the state of the queue can only change at the beginning or at the end of a rectangle

{0,4,7},{0,8,6},{6,6,12},{6,4,16},{10,5,20},{22,2,26}

Sweep Line for Skyline

1. Sort the points (x,h,beginning/end) of the beginning and the end of each building and process each (2xn points)

2. Whenever a new point is met with height h:

    1. Beginning: add in the queue an new entry with key=h as corresponding height, or increment the counter if key already present

    2. End: decrement the counter associated with the key=h (already present)

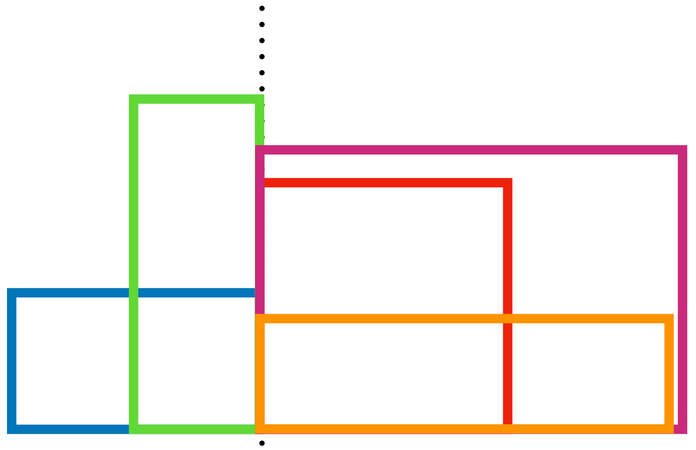
3. A new point of the skyline is created if a new height is detected (when you have processed all the events at this time)

Building Point

```
public static class BuildingPoint implements Comparable<BuildingPoint> {
    int x;
    boolean isStart;
    int height;

    public BuildingPoint(int x, boolean isStart, int height) {
        this.x = x;
        this.isStart = isStart;
        this.height = height;
    }

    @Override
    public int compareTo(BuildingPoint other) {
        if (this.x != other.x) {
            return this.x - other.x;
        } else {
            // always put start points before end points
            // if both are start points, order by decreasing height
            // if both are end points, order by increasing height
            return (this.isStart ? -this.height : this.height) - (other.isStart ? -other.height : other.height);
        }
    }
}
```



Full Algo

```
public static List<int[]> getSkyline(int[][] buildings) {
    // Create a list of points.
    BuildingPoint[] points = new BuildingPoint[buildings.length * 2];
    // Fill in the points
    Arrays.sort(points); // Sort the points.
    // Use a tree map to represent the active buildings.
    TreeMap<Integer, Integer> queue = new TreeMap<>();
    queue.put(0, 1); // Add a ground level (height 0).
    int prevMaxHeight = 0;
    List<int[]> result = new ArrayList<>();
    for (BuildingPoint point : points) {
        if (point.isStart) {
            // If it's a start point, add the height to the map, or increment the existing height's count.
            queue.compute(point.height, (key, value) -> {
                if (value != null) return value + 1;
                return 1;
            });
        } else {
            // If it's an end point, decrement or remove the height from the map.
            queue.compute(point.height, (key, value) -> {
                if (value == 1) return null;
                return value - 1;
            });
        }
        // Get current max height after the addition/removal above.
        int currentMaxHeight = queue.lastKey();
        // If the current max height is different from the previous one, we have a critical point.
        if (prevMaxHeight != currentMaxHeight) {
            result.add(new int[]{point.x, currentMaxHeight});
            prevMaxHeight = currentMaxHeight;
        }
    }
    return result;
}
```

```
// Fill in the points
int index = 0;
for (int[] building : buildings) {
  points[index] = new BuildingPoint(building[0], true, building[1]);
  points[index + 1] = new BuildingPoint(building[2], false, building[1]);
  index += 2;
}
```

Time Complexity ?

Solution Interro

LINFO 1121

DATA STRUCTURES AND ALGORITHMS

UCLouvain

Les tables de hachage (Hash tables)

Pierre Schaus

Rappel: les tables de symboles

- Type abstrait de données permettant:
  - D'**insérer** une valeur avec une clef 🗝
  - Etant donné la clef 🗝 de **retrouver** la valeur correspondante

Nous avons vu:

- Les arbres de recherche équilibrés (red-black trees)

- Ils permettent d'insérer/retirer une clef en O(log(N)) et ils permettent surtout d'obtenir des méthodes relatives à l'ordre sur les clefs:
  - Énumération en ordre croissant
  - Retrouver le plus petit, le successeur, ceil, floor, etc
  - Mais cela nécessite d'avoir des clefs comparables entre elles (ordre total). Ce n'est pas toujours possible et on n'a pas toujours besoin de ces méthodes.

Hashtable aussi appelée Hashmap

- Une implémentation spécifique de dictionnaire lorsque nous souhaitons seulement faire des push(key,value) et get(key)

- Ce n’est donc pas un dictionnaire ordonné.

- La grosse différence avec les red-black est que le push/get se fait en O(1) expected au lieu de O(log(n)) pire cas. Grosse différence en pratique ?

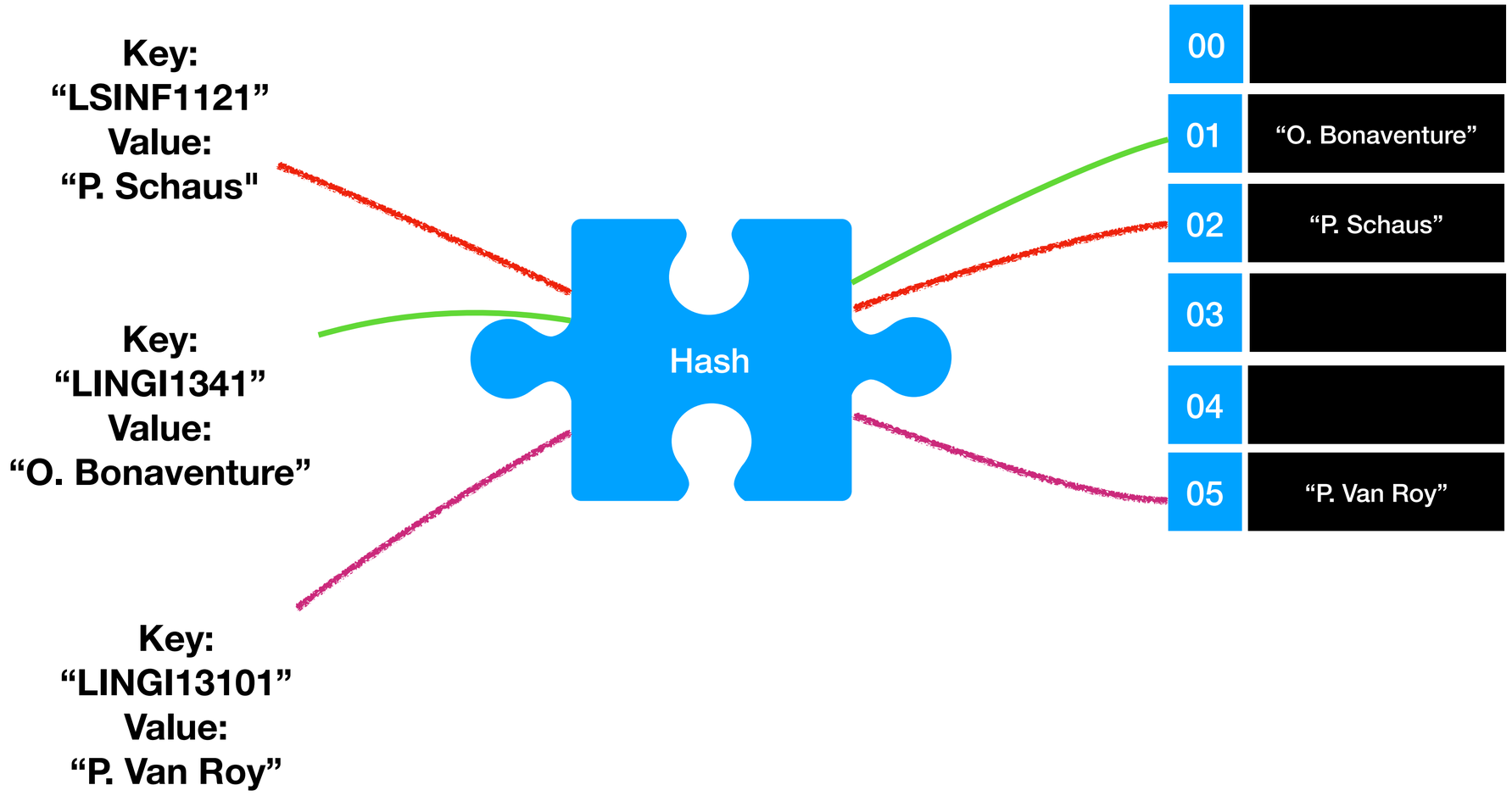
- Speed O(1) vs O(log(N))

```
int size = 10000000;
Random r = new Random();
Integer[] o = new Integer[size];
for (int i=0; i<size; ++i)
    o[i] = Integer.valueOf(r.nextInt());
Map map = new HashMap() ;
map = new TreeMap(); // red-black
long l = System.currentTimeMillis();
for (int i = 0; i < size; i++) {
    map.put(o[i], o[i]);
}
for (int i = size - 1; i > -1; i--) {
    map.remove(o[i]);
}
l = System.currentTimeMillis() - l;
System.out.println("time = " + l);
```

- Résultat sur les implémentations de Java:
  - 4s HashMap
  - 27s TreeMap (red-black in java)

Les hash table sont basée sur les fonction de Hash

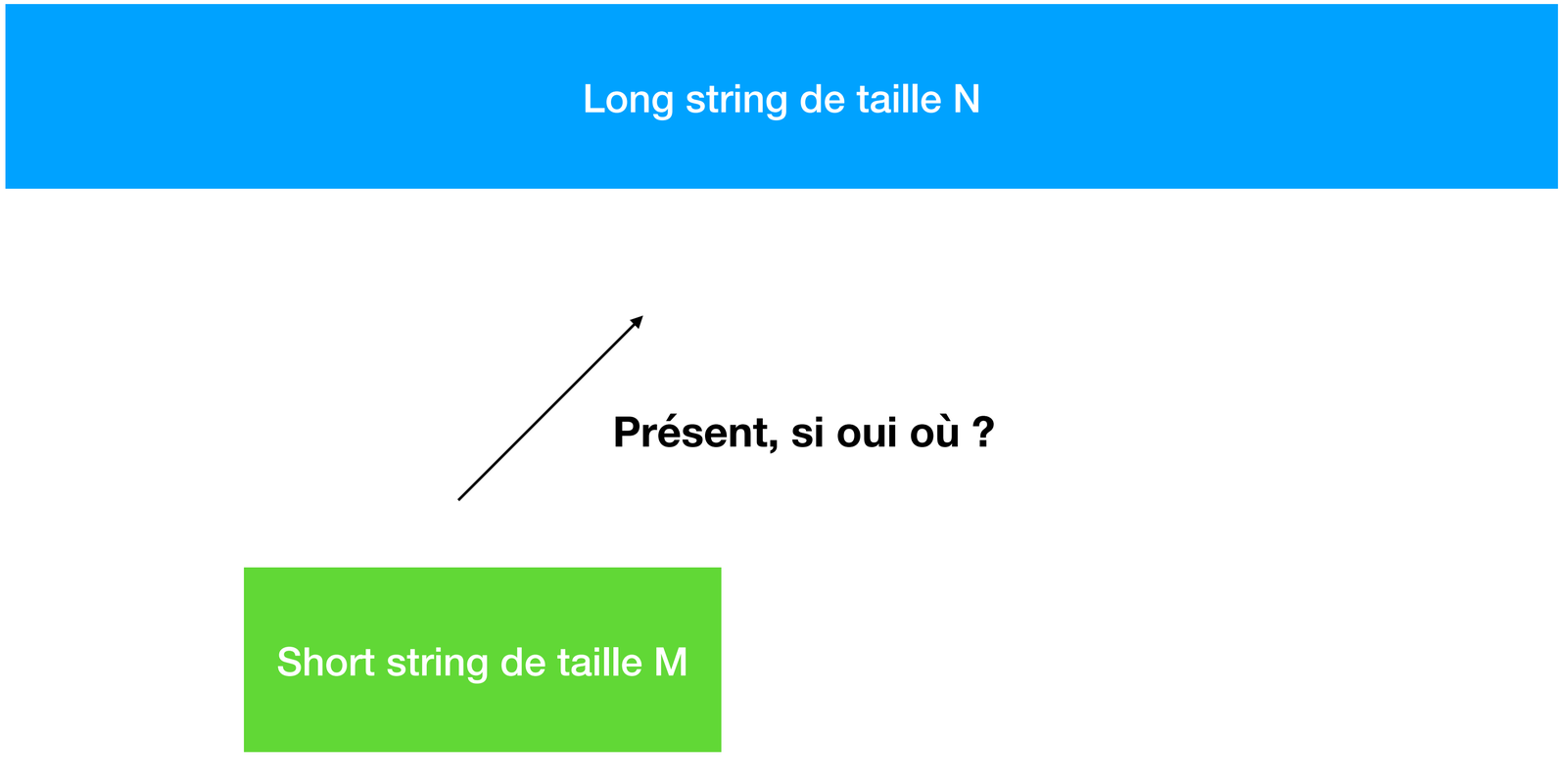
- Une fonction qui transforme l'objet en entier qui semble "aléatoire" mais en fait cette transformation **est déterministe (algorithme).**



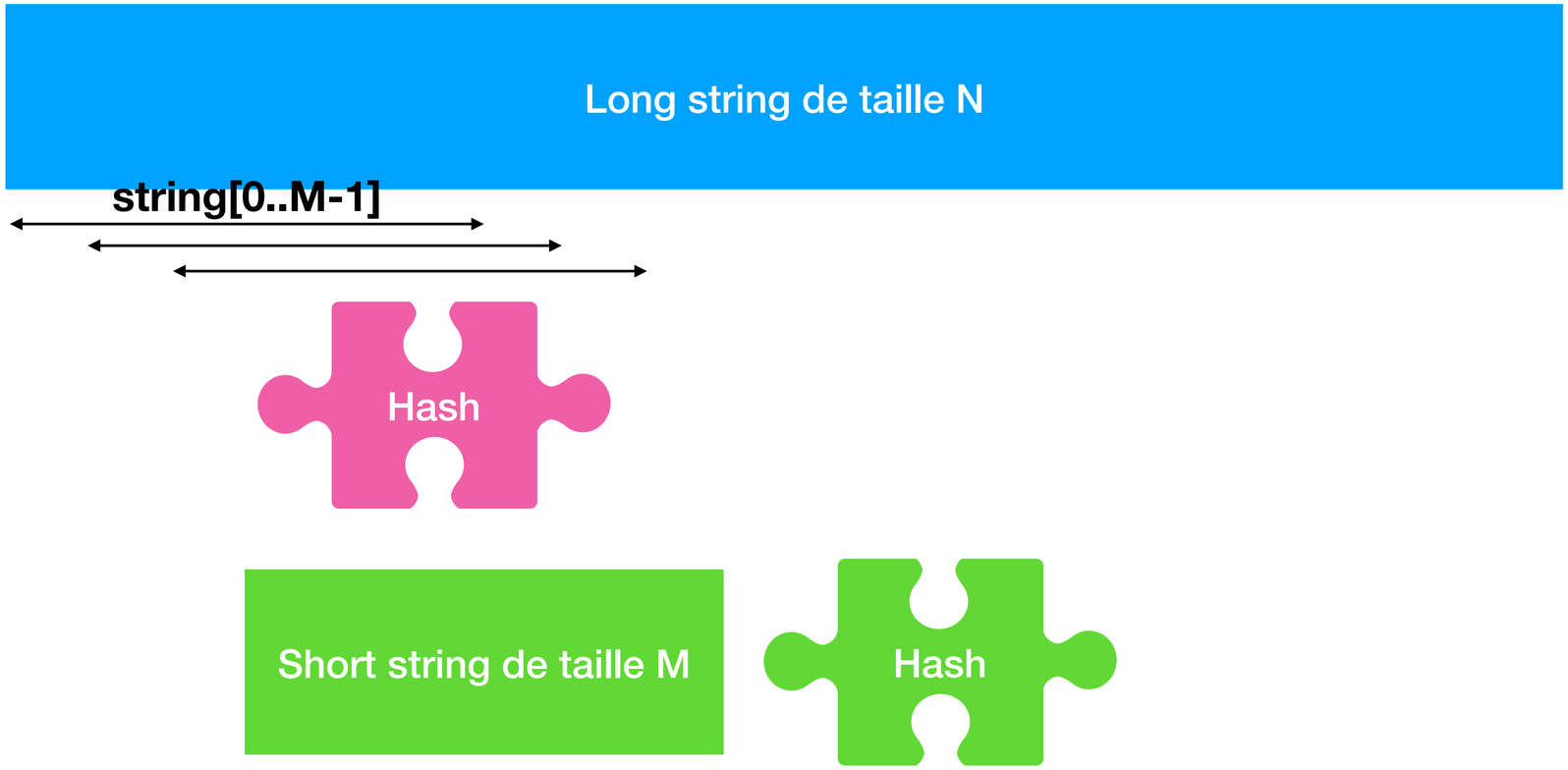
Algorithm de Rabin-Karp

- Une bonne fonction de Hash devrait avoir le comportement d’un dé

Rabin Karp


Long string de taille N
Présent, si oui où ?
Short string de taille M

Rabin Karp


Long string de taille N
string[0..M-1]
Hash
Short string de taille M
Hash