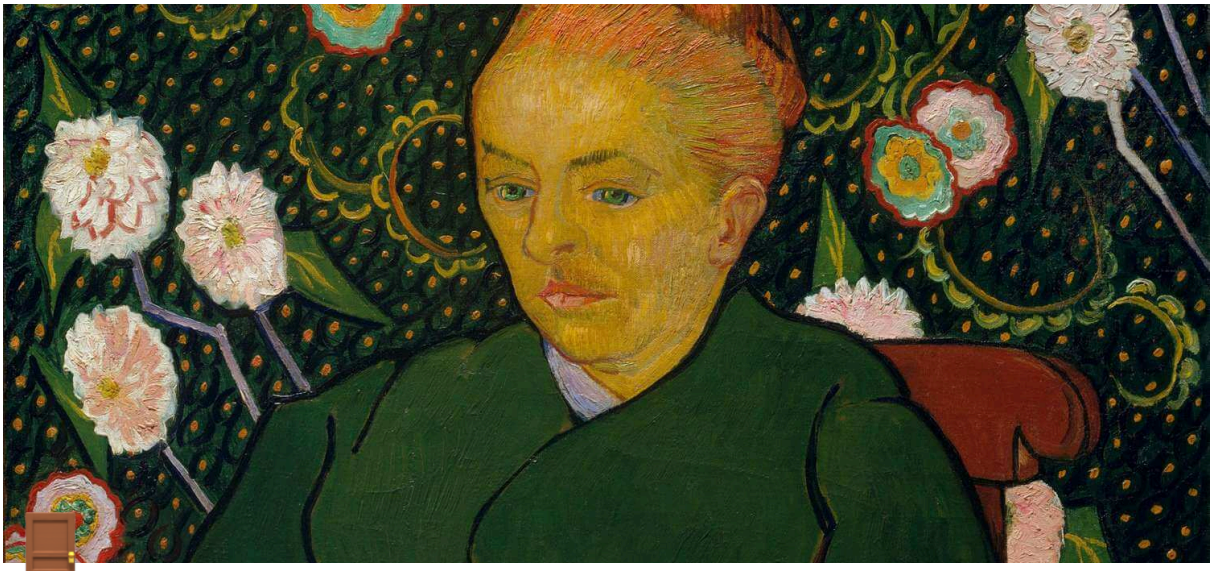


Projet - UE Ouverture

1 - Présentation :

1.1 - Algorithmes de Rémy :

1.1 - Définir une structure de données adéquate pour manipuler les arbres binaires :

Nous allons construire un module représentant un arbre binaire qu'on nommera `Tree` et est sous la forme suivante :

```
type 'a tree =
  | Empty of 'a
  | Node of 'a * 'a tree * 'a tree

let empty = Empty 0

let getValue t =
  match t with
  | Empty a -> a
  | Node (v, _, _) -> v
```

1.2 - Encoder un prototype de l'algorithme de Rémy :

```
open Tree

let splitTree tree nb =
  let piece = Random.int 2 in
  if piece = 0 then Node (nb, tree, Empty (nb + 1))
  else Node (nb, Empty (nb + 1), tree)

let rec createTree tree nb r =
```

```
  let value = getValue tree in
  if value = r then splitTree tree nb
  else
    match tree with
    | Empty _ -> tree
    | Node (v, g, d) ->
        Node (v, createTree g nb r, createTree d nb r)

let rec algoRemy tree n i =
  if n = 0 then tree
  else
    let value = Random.int i in
    algoRemy (createTree tree i value) (n - 1) (i+2)
```

1.4 - Soit votre prototype n'est pas suffisamment efficace. On s'autorise à stocker un peu plus d'information que nécessaire lors de la construction afin d'aboutir à une nouvelle complexité temporelle linéaire. Définir une nouvelle implémentation et justifier de son efficacité en temps linéaire dans le pire des cas :

Le problème avec notre codage est que pour trouver le noeud qui a été sélectionné dans notre algorithme, nous somme obligé de parcourir tout l'arbre. Ce qui nous fait une complexité en $O(n^2)$ (On applique n fois l'algorithme et au pire des cas, on va parcourir tous les noeuds ce qui nous fait un $O(n)$ car on aura à chaque étape 1 + 2 + 3 + ... + n ~ $n(n+1)/2$).

Pour améliorer cela, on pourrait avoir un pointeur directement sur nos noeuds qu'on stockera dans un tableau pour avoir un accès en $O(1)$. On doit pour cela savoir le nombre d'élément qu'on aura dans notre tableau.

On remarque qu'à chaque étape, on créer 2 nouveaux noeuds et donc qu'on aura au total $2 \cdot n + 1$ noeuds (feuilles inclus) au final. Donc on initialise notre tableau comme :

```
  let list_node = Array.init (n+1) (fun _ -> ref Empty)
```

Maintenant, on modifie la structure de notre arbre pour avoir des références de noeuds pour nos fils gauche et droite pour que les modifications s'applique dans l'arbre final :

```
type 'a tree =
  | Empty
  | Node of 'a tree ref * 'a tree ref

let empty = Empty

(* Pretty printing d'un arbre binaire avec indentation *)
let rec print_tree_pretty ?(indent="") ?(max_depth=10) depth tree =
  if depth > max_depth then
    Printf.printf "%s...\n" indent
  else
    match tree with
    | Empty -> Printf.printf "%sEmpty\n" indent
    | Node (left, right) ->
        Printf.printf "%sNode\n" indent;
        Printf.printf "%s├─ left: " indent;
```

```
        print_tree_pretty ~indent:(indent ^ " │   ") ~max_depth (depth+1) !left;
        Printf.printf "%s└── right: " indent;
        print_tree_pretty ~indent:(indent ^ "    ") ~max_depth (depth+1) !right
```

On a à la fin l'algorithme de Rémy en $O(n)$ grâce au fait que chaque noeud est accessible en $O(1)$ et plus besoin de reparcourir l'arbre :

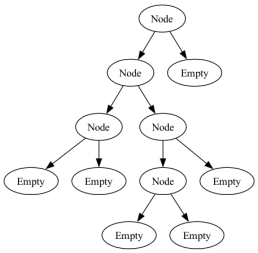
```
open TreeRef

let splitTreeRef tree i list_node =
  let piece = Random.int 2 in
  let enfant = ref Empty in
  match !tree with
  | Empty →
      tree := Node (enfant, ref Empty);
      list_node.(i) ← tree;
      list_node.(i+1) ← enfant
  | Node (_, _) →
      if piece = 0 then (
        tree := Node (ref !tree, enfant);  (* copie de l'ancien noeud dans left *)
        list_node.(i) ← tree;
        list_node.(i+1) ← enfant
      ) else (
        tree := Node (enfant, ref !tree);  (* copie de l'ancien noeud dans right *)
        list_node.(i) ← enfant;
        list_node.(i+1) ← tree
      )



let rec algoRemyRef n i list_node =
  if n = 0 then ()
  else (
    let value = Random.int i in
    (
    splitTreeRef list_node.(value) i list_node;
    algoRemyRef (n - 1) (i+2) list_node
    )
  )
```

Dans le main, on demande à l'utilisateur d'entrer un entier `n` et on aura le main suivant :

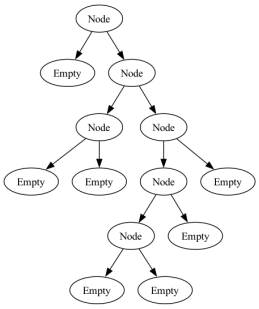
```
let () =
    print_string "Entrez un nombre n : ";
    let n = read_int () in
    let list_node = Array.init (2 * n+1) (fun _ → ref Empty) in
    algoRemyRef n 1 list_empty;
    export_tree_to_dot "arbre.dot" !(list_empty.(0));
    print_endline "Fichier arbre.dot généré !"
```

1.5 - Tester l'algorithme de génération sur de petits arbre :

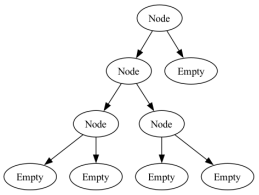
- Pour `n = 5` :



- Pour `n = 6` :



Pour `n = 4` :



1.7 - Proposer un algorithme de complexité linéaire dans le pire des cas pour générer un arbre binaire de type ABR :

Maintenant, nous n'avons plus besoin d'avoir une aussi grande liste, nous allons stocker seulement les feuilles dans le tableau et non les noeuds (feuilles exclus).

On remarque qu'on créer à chaque fois 2 feuilles et que dans notre tableau on retire le noeud courant. Pour gagner de la mémoire, nous pouvons seulement écraser la référence de l'ancienne feuille, devenu noeud, avec l'une des 2 nouvelles feuilles.

```
let list_empty = Array.init (n+1) (fun _ -> ref Empty)
```

On créer une liste de $n + 1$ feuilles car elle va stocker la toute dernière feuilles que l'algorithme de Rémy créera.

On fait tout cela pour avoir un accès au feuilles en $O(1)$.

On a le code modifié de Rémy suivant :

```
open TreeRef

let splitTreeRefABR tree valeur nb_leaf list_empty =
  let gauche = ref Empty in
  let droit = ref Empty in
  tree := Node(gauche, droit);
  list_empty.(valeur) ← gauche;
  list_empty.(nb_leaf) ← droit


let rec algoRemyRefABR n nb_leaf list_empty =
  if n = 0 then ()
  else
    let valeur = Random.int nb_leaf in
    splitTreeRefABR list_empty.(valeur) valeur nb_leaf list_empty;
    algoRemyRefABR (n-1) (nb_leaf+1) list_empty
```

Ici, plus besoin de choisir de faire un pile ou face car le `tree` passé en argument à `splitTreeRefABR` est forcément une feuille et on l'écrase bien dans `list_empty` lorsqu'on créer les 2 nouveaux `Empty` (feuille).

On a le main suivant :

```
let () =
  Random.self_init ();

  print_string "Entrez un nombre n : ";
  let n = read_int () in
  let list_empty = Array.init (n+1) (fun _ -> ref Empty) in
  let arbre = list_empty.(0) in
  algoRemyRefABR n 1 list_empty;
  export_tree_to_dot "arbre.dot" !(arbre);
  print_endline "Fichier arbre.dot généré !"
```

La complexité reste toujours en $O(n)$ car :

- On possède un tableau de taille $n+1$ représentant nos feuilles :
  - On va faire un accès à un élément du tableau n fois
  - La construction du noeud, la suppression de l'ancienne feuille dans notre table et l'ajout des 2 nouvelles feuilles dans notre noeud est en $O(1)$

On a donc bien une complexité en $O(n)$ car chaque opération fait à chaque étape est en $O(1)$.