

# Parcours d'arbre

Quentin Fortier

December 25, 2021

# Parcours en profondeur

Le parcours en **profondeur** va le plus profondément possible dans une branche avant de passer à la prochaine :

- **Parcours préfixe** : d'abord  $r$ , puis tous les sommets de  $g$  (appel récursif), puis les sommets de  $d$  (appel récursif).

# Parcours en profondeur

Le parcours en **profondeur** va le plus profondément possible dans une branche avant de passer à la prochaine :

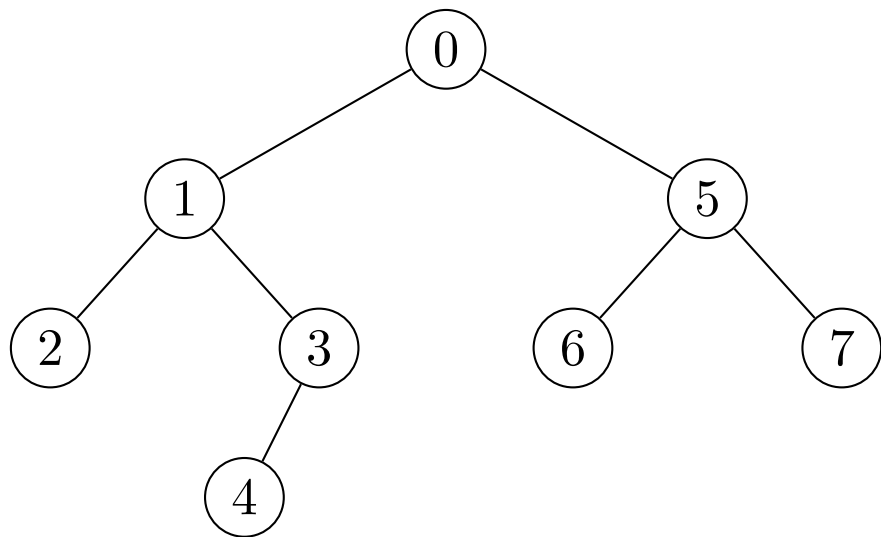
- **Parcours préfixe** : d'abord  $r$ , puis tous les sommets de  $g$  (appel récursif), puis les sommets de  $d$  (appel récursif).
- **Parcours infixé** : d'abord tous les sommets de  $g$  (appel récursif), puis  $r$ , puis les sommets de  $d$  (appel récursif).

# Parcours en profondeur

Le parcours en **profondeur** va le plus profondément possible dans une branche avant de passer à la prochaine :

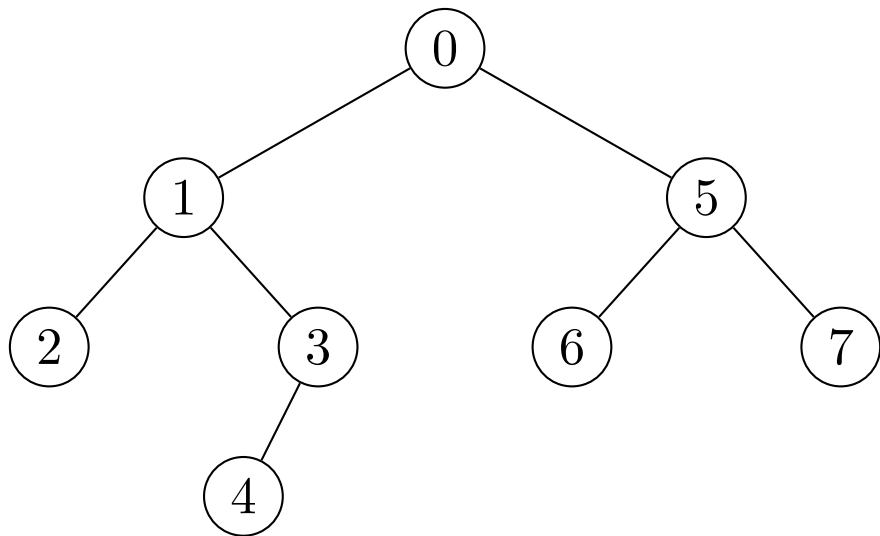
- **Parcours préfixe** : d'abord  $r$ , puis tous les sommets de  $g$  (appel récursif), puis les sommets de  $d$  (appel récursif).
- **Parcours infixé** : d'abord tous les sommets de  $g$  (appel récursif), puis  $r$ , puis les sommets de  $d$  (appel récursif).
- **Parcours suffixe** : d'abord tous les sommets de  $g$  (appel récursif), puis les sommets de  $d$  (appel récursif), puis  $r$ .

## Parcours en profondeur : Parcours préfixe



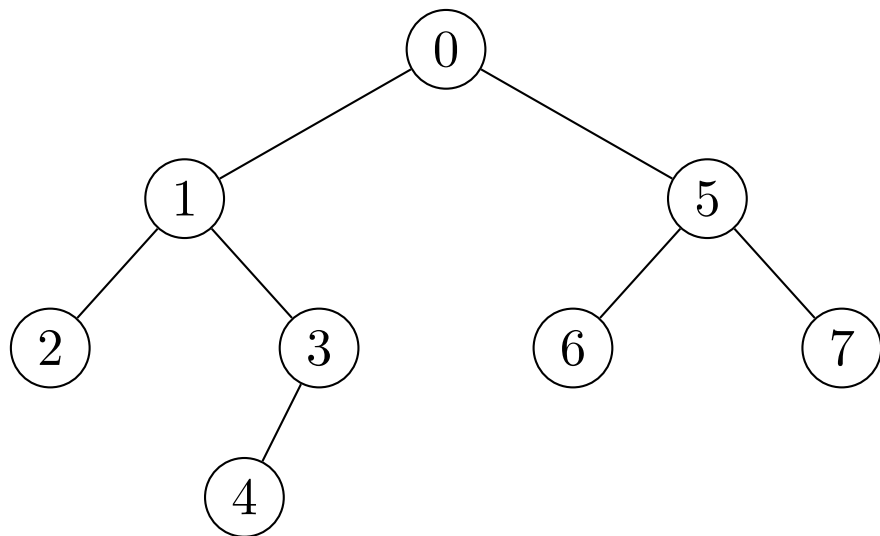
Parcours préfixe : 0, 1, 2, 3, 4, 5, 6, 7

## Parcours en profondeur : Parcours infixe



Parcours infixe : 2, 1, 4, 3, 0, 6, 5, 7

## Parcours en profondeur : Parcours suffixe



Parcours suffixe : 2, 4, 3, 1, 6, 7, 5, 0

# Parcours en profondeur : En OCaml

Parcours préfixe en OCaml :

---

```
1  let rec prefix = function
2      | E -> []
3      | N(r, g, d) -> r::(prefix g)@(prefix d)
```

---

Complexité :



# Parcours en profondeur : En OCaml

Parcours préfixe en OCaml :

---

```
1  let rec prefix = function
2      | E -> []
3      | N(r, g, d) -> r::(prefix g)@(prefix d)
```

---

Complexité :  $O(n^2)$  à cause de @ (où  $n$  est le nombre de noeuds)

# Parcours en profondeur : En OCaml

Parcours préfixe en  $O(n)$ , en ajoutant un accumulateur pour éviter d'utiliser @ :

---

```
1  let rec prefix acc = function
2      | E -> acc
3      | N(r, g, d) -> r::prefix (prefix acc d) g;;
```

---

# Parcours en profondeur : En OCaml

Parcours infixe en  $O(n^2)$  :

---

```
1  let rec infix = function
2      | E -> []
3      | N(r, g, d) -> (infix g)@[r]@(infix d)
```

---

# Parcours en profondeur : En OCaml

Parcours infixe en  $O(n^2)$  :

---

```
1  let rec infix = function
2      | E -> []
3      | N(r, g, d) -> (infix g)@[r]@(infix d)
```

---

Parcours infixe en  $O(n)$  :

---

```
1  let rec aux acc = function
2      | E -> acc
3      | N(r, g, d) -> aux (r::aux d) g;;
4
5  let infix = aux [];;
```

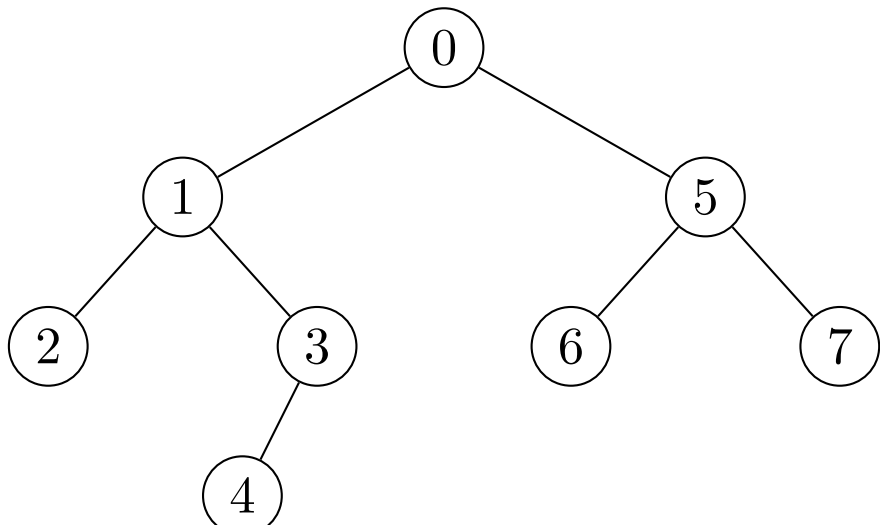
---

# Parcours en largeur

Le parcours en largeur consiste à visiter les noeuds par couche (distance croissante à la racine) : d'abord la racine, puis les noeuds à distance 1, puis 2...

# Parcours en largeur

Le parcours en largeur consiste à visiter les noeuds par couche (distance croissante à la racine) : d'abord la racine, puis les noeuds à distance 1, puis 2...



# Parcours en largeur : Avec récursivité

Fonction qui renvoie la liste des noeuds dans l'ordre d'un parcours en largeur :

---

```
1  let bfs a =
2      let rec aux cur next = match cur with
3          (* cur : liste des noeuds dans la couche en cours *)
4          (* next : liste des noeuds dans la couche suivante *)
5              | [] -> if next = [] then [] else bfs next []
6              | a::q -> match a with
7                  | E -> bfs q next
8                  | N(r, g, d) -> r::bfs q (g::d::next) in
9      aux [a] []
```

---

# Parcours en largeur : avec file immutable

---

```
1 create : unit -> 'a queue
2 is_empty : 'a queue -> bool
3 add : 'a -> 'a queue -> 'a queue
4 peek : 'a queue -> 'a
5 take : 'a queue -> 'a queue
```

---



## Parcours en largeur : avec file immutable

---

```
1  create : unit -> 'a queue
2  is_empty : 'a queue -> bool
3  add : 'a -> 'a queue -> 'a queue
4  peek : 'a queue -> 'a
5  take : 'a queue -> 'a queue
```

---

---

```
1  let bfs a =
2      let rec aux f =
3          if is_empty f then []
4          else let q = take f in
5              match peek f with
6                  | E -> aux q
7                  | N(r, g, d) -> r::(add g q |> add d |> aux)
8      in aux (create () |> add a);;
```

---

## Parcours en largeur : avec file mutable

---

```
1  let bfs a =
2      let l = ref [] in
3      let f = Queue.create () in
4      Queue.add a f;
5      while not (Queue.is_empty f) do
6          let next = Queue.take f in
7          match next with
8          | E -> ()
9          | N(r, g, d) -> l := r::!l; Queue.(add g f; add d f)
10     done;
11     List.rev !l;;
```

---

## Exercice

Écrire les fonctions de parcours d'arbre binaire en C.