Implémentation 1 *- tri par fusion*

```ocaml
let rec casser l =
    match l with
    | [] -> [], []
    | [e1] -> [e1], []
    | e1::e2::q ->
        let l1, l2 = casser q in
        e1::l1, e2::l2

let rec fusion l1 l2 =
    match l1, l2 with
    | [], _ -> l2
    | _, [] -> l1
    | e1::q1, e2::q2 ->
        if e2 > e1 then
            e1::(fusion q1 l2)
        else
            e2::(fusion l1 q2)

let rec tri_fusion l =
    match l with
    | [] -> []
    | [e1] -> [e1]
    | _ ->
        let l1, l2 = casser l in
        fusion (tri_fusion l1) (tri_fusion l2)
```

Implémentation 2 - *parcours en largeur d'un graphe*

```
type file = {e:int list; s:int list}

let file_vide = {e=[]; s=[]}

let rec ajoute f liste = match liste with
    | [] -> f
    | elt::q -> ajoute {e=(elt::f.e); s=f.s} q

let pop_opt f =
    let rec retourne sub_f =
        match sub_f.e with
        | [] -> sub_f
        | elt::q -> retourne {e=q; s=elt::sub_f.s}
    in let new_f =
        if f.s = [] then
            retourne f
        else f
    in match new_f.s with
    | [] -> file_vide, None
    | elt::q -> {e=new_f.e; s=q}, Some elt



type graphe = int list array

let parcours_largeur g s =
    let n = Array.length g in
    let non_vus = Array.make n true in
    let rec parcours f =
        match (pop_opt f) with
        | _, None -> ()
        | new_f, Some v when non_vus.(v) ->
            non_vus.(v) <- false;
            print_int v;
            parcours (ajoute new_f g.(v))
        | new_f, Some v ->
            parcours new_f
    in parcours {e=[]; s=[s]}
```

Implémentation 3 - *file d'entiers*

```c
struct Maillon{
    int val;
    struct Maillon* suivant;
};
typedef struct Maillon maillon;

struct File{
    maillon* e; //maillon d'entrée
    maillon* s; //maillon de sortie
};
typedef struct File file;

file* file_vide(){
    file* res = malloc(sizeof(file));
    assert(res != NULL);
    res->e = NULL;
    res->s = NULL;
    return res;
}
```