Implémentation - *tri par fusion*

```
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 - *parcours en largeur d'un graphe (1/3)*

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

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
```

Implémentation *- parcours en largeur d'un graphe (2/3)*

```
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
```

Implémentation *- parcours en largeur d'un graphe (3/3)*

```
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 - *liste chainée en C (1/3)*

```c
typedef int elemtype;

struct Maillon{
    elemtype val;
    struct Maillon* suivant;
};
typedef struct Maillon maillon;
```

Implémentation - *liste chainée en C (2/3)*

```c
maillon* ajoute(elemtype x, maillon* c){
    maillon* res = malloc(sizeof(maillon));
    assert(res != NULL);
    res->val = x;
    res->suivant = c;
    return res;
};
```

Implémentation - *liste chainée en C (3/3)*

```c
int main(){
    maillon* a = ajoute(1,NULL);
    a = ajoute(2,a);
    a = ajoute(3,a);
    return 0;
};
```

Implémentation - *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;
}
```

Implémentation - *file de priorité :* `vide` *et* `ajoute` *(1/2)*

```
type tas_binaire_min = {
    mutable nb_elts: int;
    mutable data: (char*int) array
}

let tbmin_vide () = {nb_elts = 0; data = [||]}

let prio couple =
    let _,b = couple in b

let redim tbmin new_taille cur_taille =
    assert (new_taille >= tbmin.nb_elts);
    let new_data = Array.make new_taille ('\000',0) in
    for i=0 to tbmin.nb_elts - 1 do
        new_data.(i) <- tbmin.data.(i)
    done;
    tbmin.data <- new_data
```

Implémentation - *file de priorité :* `vide` *et* `ajoute` *(2/2)*

```
let tbmin_ajoute tbmin x p  =
    (*redimensionnement*)
    let n = Array.length tbmin.data in
    if tbmin.nb_elts >= n then
        redim tbmin (2*n+1) n;

    (*ajout et percolations vers le haut*)
    tbmin.data.(tbmin.nb_elts) <- (x,p);
    tbmin.nb_elts <- tbmin.nb_elts + 1;
    let rec percole_haut i =
        let daron = if (i-1)/2 < 0 then 0 else (i-1)/2 in
        if prio tbmin.data.(daron) > prio tbmin.data.(i) then begin
            let temp = tbmin.data.(i) in
            tbmin.data.(i) <- tbmin.data.(daron);
            tbmin.data.(daron) <- temp;
            percole_haut daron;
        end;
    in percole_haut (tbmin.nb_elts - 1)
```

Implémentation - *file de priorité :* `pop` *(1/3)*

```
type tas_binaire_min = {
    mutable nb_elts:int;
    mutable data: (char*int) array
}

let prio couple =
    let _,b = couple in b

let redim tbmin new_taille cur_taille =
    assert (new_taille >= tbmin.nb_elts);
    let new_data = Array.make new_taille ('\000',0) in
    for i=0 to tbmin.nb_elts - 1 do
        new_data.(i) <- tbmin.data.(i)
    done;
    tbmin.data <- new_data
```

Implémentation - *file de priorité : `pop` (2/3)*

```
let tbmin_pop tbmin =
    (*formalité*)
    assert (tbmin.nb_elts > 0);

    (*remplacement de la première case par la dernière*)
    let res =  tbmin.data.(0) in
    tbmin.data.(0) <- tbmin.data.(tbmin.nb_elts-1);
    tbmin.nb_elts <- tbmin.nb_elts - 1;

    (*redimensionnement*)
    let n = Array.length tbmin.data in
    if tbmin.nb_elts <= n/2 then
        redim tbmin (n/2) n;
```

Implémentation - *file de priorité : `pop` (3/3)*

```
    (*percolations du nouveau premier élément*)
    let rec percole_bas i =
        let max = tbmin.nb_elts - 1 in
        let fils_g = if 2*i+1 <= max then 2*i+1 else max in
        let fils_d = if 2*i+2 <= max then 2*i+2 else max in
        if (prio tbmin.data.(fils_d) < prio tbmin.data.(i)||
            prio tbmin.data.(fils_g) < prio tbmin.data.(i)) then begin
            (*on va percoler le fils de plus basse priorité*)
            if prio tbmin.data.(fils_d) < prio tbmin.data.(fils_g) then
                let temp = tbmin.data.(fils_d) in
                tbmin.data.(fils_d) <- tbmin.data.(i);
                tbmin.data.(i) <- temp;
                percole_bas fils_d
            else
                let temp = tbmin.data.(fils_g) in
                tbmin.data.(fils_g) <- tbmin.data.(i);
                tbmin.data.(i) <- temp;
                percole_bas fils_g
            end;
    in if tbmin.nb_elts > 0 then percole_bas 0;
    res
```