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 (1/2)*

```
type graphe = int list array
type file = {entrants:int list; sortants:int list}

let file_vide () = {entrants = []; sortants = []}

let pop_opt f =
    let rec retourne f1 =
        match f1.entrants with
        | [] -> f1
        | e::q -> retourne {entrants=q; sortants = e::f1.sortants}
    in let f2 = if f.sortants = [] then retourne f else f
    in match f2.sortants with
    | [] -> file_vide (), None (* file vide *)
    | e::q -> {entrants=f2.entrants; sortants=q}, Some e
```

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

```
let rec ajoute f liste =
    match liste with
    | [] -> f
    | e::q -> ajoute {entrants = e::f.entrants; sortants = f.sortants} q

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 -> []
        | f1, Some v when non_vus.(v) ->
            non_vus.(v) <- false;
            v::(parcours (ajoute f1 g.(v)))
        | f1, Some v ->
            parcours f1
    in parcours {entrants=[s]; sortants=[]}
```

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 : type et fonction* `redim`

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

let redim tbmin new_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 : fonction* `percole_haut`

```
let percole_haut tbmin i_depart =
    let rec percole 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 daron;
        end
    in if tbmin.nb_elts <> 0 then
        percole i_depart
```

Implémentation - *file de priorité : fonction* `percole_bas`

```
let percole_bas tbmin i_depart =
    let rec percole 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 fils_d
            else
                let temp = tbmin.data.(fils_g) in
                tbmin.data.(fils_g) <- tbmin.data.(i);
                tbmin.data.(i) <- temp;
                percole fils_g
        end
    in if tbmin.nb_elts <> 0 then
        percole i_depart
```

Implémentation - *file de priorité : type et fonction* `file_vide`

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

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

Implémentation - *file de priorité : fonction* `ajoute` *(qui remplace aussi)*

```
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);

    (*vérification de la présence éventuelle de x*)
    let deja_present = ref false in (*indice de x si existence, sinon -1*)
    for i=0 to tbmin.nb_elts-1 do
        match tbmin.data.(i) with
        | (elt,prio) when elt = x ->
            tbmin.data.(i) <- (x, p);
            deja_present := true;
            if prio > p then
            percole_bas tbmin i
            else if prio < p then
            percole_haut tbmin i
        | _ -> ()
    done;

    (*ajout et percolations vers le haut*)
    if not !deja_present then begin
        tbmin.data.(tbmin.nb_elts) <- (x,p);
        tbmin.nb_elts <- tbmin.nb_elts + 1;
        percole_haut tbmin (tbmin.nb_elts - 1)
    end
```

Implémentation - *file de priorité : fonction* ***pop_opt***

```
let tbmin_pop tbmin =
    if tbmin.nb_elts = 0 then None else

 (* 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);

    (*percolations du nouveau premier élément*)
    percole_bas tbmin 0;
    Some res
```

Implémentation - *algorithme de Dijkstra (1/2)*

On suppose implémentée la structure de file de priorité min.

```
let algo_dijkstra (g:graphe) (s:char) (t:char) =
    let non_vus = Array.make 256 true in
 (* le prédécesseur de chaque sommet *)
    let pred = Array.make 256 '\000' in
 (* mémoïsation : distances de s à chaque sommet *)
    let dist = Array.make 256 max_int in
    dist.(int_of_char s) <- 0;
    let file_p = tbmin_vide () in
    tbmin_ajoute file_p s 0;

    let rec reconstruire chemin (sommet:char) =
        match sommet with
        | '\000' -> chemin
        | _ -> reconstruire (sommet::chemin) pred.(int_of_char sommet)
```

Implémentation - *algorithme de Dijkstra (2/2)*

```
    in let rec traitement u voisins =
     (* pour chaque voisin v de u, si favorable,
        on remplace par la distance la plus courte puis on avance,
        sinon on avance directement *)
        match voisins with
        | [] -> ()
        | (v,w)::q  when non_vus.(int_of_char v) &&
          dist.(int_of_char u) + w < dist.(int_of_char v) ->
            dist.(int_of_char v) <- dist.(int_of_char u) + w;
            tbmin_ajoute file_p v dist.(int_of_char v);
            pred.(int_of_char v) <- u;
            traitement u q
        | (v,w)::q -> traitement u q
    in let rec parcours () =
        match tbmin_pop file_p with
        | None -> failwith "chemin inexistant"
        | Some (u,_) when u=t -> reconstruire [] u
        | Some (u,_) when non_vus.(int_of_char u) ->
            non_vus.(int_of_char u) <- false;
            traitement u g.(int_of_char u);
            parcours ()
        | Some _ -> parcours ()
    in parcours ()
```

Implémentation - *structure Unir et Trouver avec forêt, doublement optimisée (1/2)*

```
type pile_spaghetti = {
    mutable parent: pile_spaghetti option;
    mutable rang: int;
    valeur: int;
}
let creer x =
 (* création d'une classe d'équivalence *)
    {parent=None; rang=0; valeur=x}

let rec trouver x =
 (* recherche du représentant
    on fait de la COMPRESSIONS DE CHEMIN *)
    match x.parent with
    | None -> x
    | Some e ->
        let res = trouver e in
        x.parent <- Some res; (* applatissement de l'arbre *)
        res
```

Implémentation - *structure Unir et Trouver avec forêt, doublement optimisée (2/2)*

```
let unir x y =
 (* réunion de deux classes d'équivalence
    on fait de l'UNION PAR RANG *)
    let parent_x = trouver x in
    let parent_y = trouver y in
    if parent_x <> parent_y then begin
        match parent_x, parent_y with
        | a,b when a.rang = b.rang ->
            a.parent <- Some b;
            b.rang <- b.rang + 1
        | a,b when a.rang < b.rang -> a.parent <- Some b
        | a,b when a.rang > b.rang -> b.parent <- Some a
        | _ -> () (* ne sert à rien *)
    end
```

Implémentation - *création du tableau* `[1,2,3]` *en C*

```c
int main(){
    int a[3] = {1,2,3};
    return 0;
}
```