

## Implémentation - tri par fusion

```
1 | let rec casser l =
2 |   match l with
3 |   | [] -> [], []
4 |   | [e1] -> [e1], []
5 |   | e1::e2::q ->
6 |     let l1, l2 = casser q in
7 |     e1::l1, e2::l2
8 |
9 | let rec fusion l1 l2 =
10 |   match l1, l2 with
11 |   | [], _ -> l2
12 |   | _, [] -> l1
13 |   | e1::q1, e2::q2 ->
14 |     if e2 > e1 then
15 |       e1::(fusion q1 l2)
16 |     else
17 |       e2::(fusion l1 q2)
18 |
19 | let rec tri_fusion l =
20 |   match l with
21 |   | [] -> []
22 |   | [e1] -> [e1]
23 |   | _ ->
24 |     let l1, l2 = casser l in
25 |     fusion (tri_fusion l1) (tri_fusion l2)
```

**Implémentation** - *parcours en largeur d'un graphe (1/2)*

```
1 | type graphe = int list array
2 | type file = {entrants:int list; sortants:int list}
3 |
4 | let file_vide () = {entrants = []; sortants = []}
5 |
6 | let pop_opt f =
7 |     let rec retourne f1 =
8 |         match f1.entrants with
9 |         | [] -> f1
10 |         | e::q -> retourne {entrants=q; sortants = e::f1.sortants}
11 |     in let f2 = if f.sortants = [] then retourne f else f
12 |     in match f2.sortants with
13 |     | [] -> file_vide (), None (* file vide *)
14 |     | e::q -> {entrants=f2.entrants; sortants=q}, Some e
```

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

```
1 | let rec ajoute f liste =
2 |     match liste with
3 |     | [] -> f
4 |     | e::q -> ajoute {entrants = e::f.entrants; sortants = f.sortants} q
5 |
6 | let parcours_largeur g s =
7 |     let n = Array.length g in
8 |     let non_vus = Array.make n true in
9 |     let rec parcours f =
10 |         match pop_opt f with
11 |         | _, None -> []
12 |         | f1, Some v when non_vus.(v) ->
13 |             non_vus.(v) <- false;
14 |             v::(parcours (ajoute f1 g.(v)))
15 |         | f1, Some v ->
16 |             parcours f1
17 |     in parcours {entrants=[s]; sortants=[]}
```

**Implémentation** - *liste chaînée en C (1/3)*

```
1 | typedef int elemtype;
2 |
3 | struct Maillon{
4 |     elemtype val;
5 |     struct Maillon* suivant;
6 | };
7 | typedef struct Maillon maillon;
```

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

```
1 | maillon* ajoute(elemtype x, maillon* c){
2 |     maillon* res = malloc(sizeof(maillon));
3 |     assert(res != NULL);
4 |     res->val = x;
5 |     res->suivant = c;
6 |     return res;
7 | };
```

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

```
1 | int main(){
2 |     maillon* a = ajoute(1, NULL);
3 |     a = ajoute(2, a);
4 |     a = ajoute(3, a);
5 |     return 0;
6 | };
```

### Implémentation - file d'entiers

```
1 struct Maillon{
2     int val;
3     struct Maillon* suivant;
4 };
5 typedef struct Maillon maillon;
6
7 struct File{
8     maillon* e; //maillon d'entrée
9     maillon* s; //maillon de sortie
10 };
11 typedef struct File file;
12
13 file* file_vide(){
14     file* res = malloc(sizeof(file));
15     assert(res != NULL);
16     res->e = NULL;
17     res->s = NULL;
18     return res;
19 }
```

### Implémentation - file de priorité : type et fonction redim

```
1 type tas_binaire_min = {
2     mutable nb_elts:int;
3     mutable data: (char*int) array
4 }
5
6 let redim tbmin new_taille =
7     assert (new_taille >= tbmin.nb_elts);
8     let new_data = Array.make new_taille ('\000',0) in
9     for i=0 to tbmin.nb_elts - 1 do
10         new_data.(i) <- tbmin.data.(i)
11     done;
12     tbmin.data <- new_data
```

**Implémentation** - file de priorité : fonction *percole\_haut*

```
1 let percole_haut tbmin i_depart =
2   let rec percole i =
3     let daron = if (i-1)/2 < 0 then 0 else (i-1)/2 in
4     if prio tbmin.data.(daron) > prio tbmin.data.(i) then begin
5       let temp = tbmin.data.(i) in
6       tbmin.data.(i) <- tbmin.data.(daron);
7       tbmin.data.(daron) <- temp;
8       percole daron;
9     end
10  in if tbmin.nb_elts <> 0 then
11    percole i_depart
```

**Implémentation** - file de priorité : fonction *percole\_bas*

```
1 let percole_bas tbmin i_depart =
2   let rec percole i =
3     let max = tbmin.nb_elts - 1 in
4     let fils_g = if 2*i+1 <= max then 2*i+1 else max in
5     let fils_d = if 2*i+2 <= max then 2*i+2 else max in
6     if (prio tbmin.data.(fils_d) < prio tbmin.data.(i) ||
7        prio tbmin.data.(fils_g) < prio tbmin.data.(i)) then begin
8       (*on va percoler le fils de plus basse priorité*)
9       if prio tbmin.data.(fils_d) < prio tbmin.data.(fils_g) then
10        let temp = tbmin.data.(fils_d) in
11        tbmin.data.(fils_d) <- tbmin.data.(i);
12        tbmin.data.(i) <- temp;
13        percole fils_d
14      else
15        let temp = tbmin.data.(fils_g) in
16        tbmin.data.(fils_g) <- tbmin.data.(i);
17        tbmin.data.(i) <- temp;
18        percole fils_g
19      end
20  in if tbmin.nb_elts <> 0 then
21    percole i_depart
```

**Implémentation** - *file de priorité : type et fonction file\_vide*

```
1 | type tas_binaire_min = {
2 |   mutable nb_elts:int;
3 |   mutable data: (char*int) array
4 | }
5 |
6 | let tbmin_vide () = {nb_elts = 0; data = [|]|}
```

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

```
1 | let tbmin_ajoute tbmin x p =
2 |   (*redimensionnement*)
3 |   let n = Array.length tbmin.data in
4 |   if tbmin.nb_elts >= n then
5 |     redim tbmin (2*n+1);
6 |
7 |   (*vérification de la présence éventuelle de x*)
8 |   let deja_present = ref false in (*indice de x si existence, sinon -1*)
9 |   for i=0 to tbmin.nb_elts-1 do
10 |     match tbmin.data.(i) with
11 |     | (elt,prio) when elt = x ->
12 |       tbmin.data.(i) <- (x, p);
13 |       deja_present := true;
14 |       if prio > p then
15 |         percole_bas tbmin i
16 |       else if prio < p then
17 |         percole_haut tbmin i
18 |     | _ -> ()
19 |   done;
20 |
21 |   (*ajout et percolations vers le haut*)
22 |   if not !deja_present then begin
23 |     tbmin.data.(tbmin.nb_elts) <- (x,p);
24 |     tbmin.nb_elts <- tbmin.nb_elts + 1;
25 |     percole_haut tbmin (tbmin.nb_elts - 1)
26 |   end
```

### Implémentation - file de priorité : fonction *pop\_opt*

```
1 let tbmin_pop tbmin =
2   if tbmin.nb_elts = 0 then None else
3
4   (* remplacement de la première case par la dernière *)
5   let res = tbmin.data.(0) in
6   tbmin.data.(0) <- tbmin.data.(tbmin.nb_elts-1);
7   tbmin.nb_elts <- tbmin.nb_elts - 1;
8
9   (* redimensionnement *)
10  let n = Array.length tbmin.data in
11  if tbmin.nb_elts <= n/2 then
12    redim tbmin (n/2);
13
14  (*percolations du nouveau premier élément*)
15  percole_bas tbmin 0;
16  Some res
```

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

On suppose implémentée la structure de file de priorité min.

```
1 let algo_dijkstra (g:graphe) (s:char) (t:char) =
2   let non_vus = Array.make 256 true in
3   (* le prédécesseur de chaque sommet *)
4   let pred = Array.make 256 '\000' in
5   (* mémorisation : distances de s à chaque sommet *)
6   let dist = Array.make 256 max_int in
7   dist.(int_of_char s) <- 0;
8   let file_p = tbmin_vide () in
9   tbmin_ajoute file_p s 0;
10
11  let rec reconstruire chemin (sommet:char) =
12    match sommet with
13    | '\000' -> chemin
14    | _ -> reconstruire (sommet::chemin) pred.(int_of_char sommet)
```

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

```
1  in let rec traitement u voisins =
2    (* pour chaque voisin v de u, si favorable,
3       on remplace par la distance la plus courte puis on avance,
4       sinon on avance directement *)
5    match voisins with
6    | [] -> ()
7    | (v,w)::q when non_vus.(int_of_char v) &&
8       dist.(int_of_char u) + w < dist.(int_of_char v) ->
9       dist.(int_of_char v) <- dist.(int_of_char u) + w;
10       tbmin_ajoute file_p v dist.(int_of_char v);
11       pred.(int_of_char v) <- u;
12       traitement u q
13    | (v,w)::q -> traitement u q
14  in let rec parcours () =
15    match tbmin_pop file_p with
16    | None -> failwith "chemin_inexistant"
17    | Some (u,_) when u=t -> reconstruire [] u
18    | Some (u,_) when non_vus.(int_of_char u) ->
19       non_vus.(int_of_char u) <- false;
20       traitement u g.(int_of_char u);
21       parcours ()
22    | Some _ -> parcours ()
23  in parcours ()
```



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

```
1 type pile_spaghetti = {
2     mutable parent: pile_spaghetti option;
3     mutable rang: int;
4     valeur: int;
5 }
6 let creer x =
7     (* création d'une classe d'équivalence *)
8     {parent=None; rang=0; valeur=x}
9
10 let rec trouver x =
11     (* recherche du représentant
12     on fait de la COMPRESSIONS DE CHEMIN *)
13     match x.parent with
14     | None -> x
15     | Some e ->
16         let res = trouver e in
17         x.parent <- Some res; (* applatissage de l'arbre *)
18     res
```

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

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