

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 (1/3)

```
1 type file = {e:int list; s:int list}
2 type graphe = int list array
3
4 let file_vide = {e=[]; s=[]}
5
6 let rec ajoute f liste = match liste with
7   | [] -> f
8   | elt::q -> ajoute {e=(elt::f.e); s=f.s} q
```

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

```
1 | let pop_opt f =
2 |   let rec retourne sub_f =
3 |     match sub_f.e with
4 |     | [] -> sub_f
5 |     | elt::q -> retourne {e=q; s=elt::sub_f.s}
6 |   in let new_f =
7 |     if f.s = [] then
8 |       retourne f
9 |     else f
10 |   in match new_f.s with
11 |   | [] -> file_vide, None
12 |   | elt::q -> {e=new_f.e; s=q}, Some elt
```

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

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

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 : vide et ajoute (1/2)*

```
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 = [|]|}
7
8  let prio couple =
9      let _,b = couple in b
10
11  let redim tbmin new_taille cur_taille =
12      assert (new_taille >= tbmin.nb_elts);
13      let new_data = Array.make new_taille ('\000',0) in
14      for i=0 to tbmin.nb_elts - 1 do
15          new_data.(i) <- tbmin.data.(i)
16      done;
17      tbmin.data <- new_data
```

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

```
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) n;
6
7   (*ajout et percolations vers le haut*)
8   tbmin.data.(tbmin.nb_elts) <- (x,p);
9   tbmin.nb_elts <- tbmin.nb_elts + 1;
10  let rec percole_haut i =
11    let daron = if (i-1)/2 < 0 then 0 else (i-1)/2 in
12    if prio tbmin.data.(daron) > prio tbmin.data.(i) then begin
13      let temp = tbmin.data.(i) in
14      tbmin.data.(i) <- tbmin.data.(daron);
15      tbmin.data.(daron) <- temp;
16      percole_haut daron;
17    end;
18  in percole_haut (tbmin.nb_elts - 1)
```

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

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

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

```
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) n;
```

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

```
1   (*percolations du nouveau premier élément*)
2   let rec percole_bas 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_bas 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_bas fils_g
19      end;
20   in if tbmin.nb_elts > 0 then percole_bas 0;
21   Some res
```