Corrigé de l'examen de programmation avancée

ENSIIE, semestre 2

mercredi 26 mars 2014

Exercice 1 : Modularité (2 points)

1. Encapsulation : seules les fonctions déclarées dans l'interface peuvent être utilisées en dehors du module, ce qui permet de cacher les fonctions auxiliaires ne préservant pas l'invariant.
2. Création d'un espace de nom : il est possible de donner le même nom à des fonctions dans des modules différents.
3. Maintenabilité : le découpage en module permet de circonscrire les modifications.
4. Réutilisabilité : il est possible d'utiliser un même module dans plusieurs projets différents.
5. Séparation des tâches : une fois les interfaces fixées, il est possible de travailler dans les modules indépendamment.
6. Abstraction : en cachant l'implémentation concrète, on ne donne à l'utilisateur du module qu'une vue abstraite plus facile à utiliser.

Exercice 2 : Compilation séparée (3 points)

1. En C :
   ```
   gcc -Wall -ansi -c main.c
   ```
   En OCaml :
   ```
   ocamlc -c main.ml
   ```
2. En C :
   ```
   gcc -Wall -ansi data.o gui.o network.o communication.o main.o
   ```
   En OCaml :
   ```
   ocamlc data.cmo gui.cmo network.cmo communication.cmo main.cmo
   ```
3. Il faut recompiler Network et Communication.
4. En C :
   ```
   prog: data.o gui.o network.o communication.o main.o
         gcc -Wall -ansi -o $@ $^

   data.o: data.h
   ```

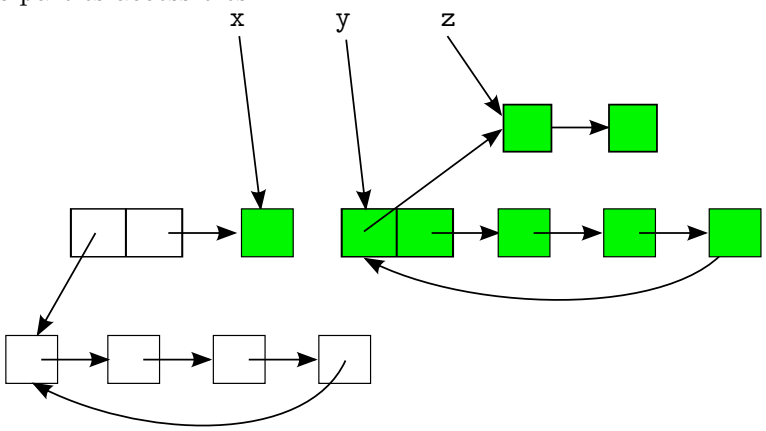
```
gui.o: data.h gui.h
network.o: network.h
communication.o: network.h communication.h
main.o: data.h gui.h communication.h
```

En OCaml :

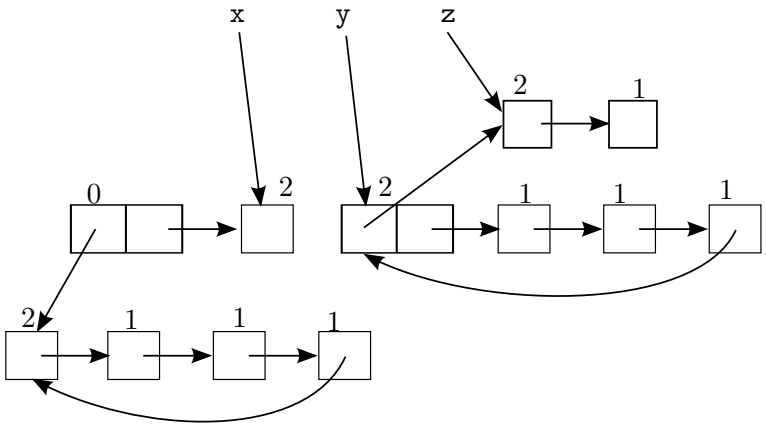
```
prog: data.cmo gui.cmo network.cmo communication.cmo main.cmo
	ocamlc -o $@ $^

%.cmo: %.ml
	ocamlc -c $<

%.cmi: %.mli
	ocamlc -c $<

data.cmo: data.cmi
gui.cmo: data.cmi gui.cmi
network.cmo: network.cmi
communication.cmo: network.cmi communication.cmi
main.cmo: data.cmi gui.cmi communication.cmi
```

Exercice 3 : Garbage collection (4 points)

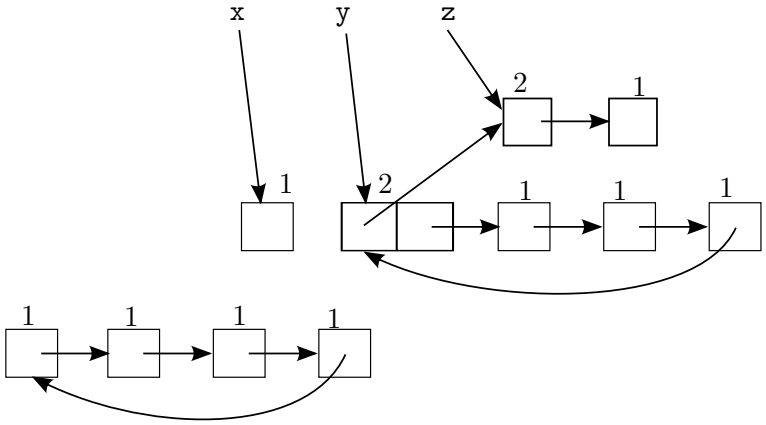
1. En vert, les parties accessibles.



2. Chaque bloc possède un compteur des références qui pointent sur lui :

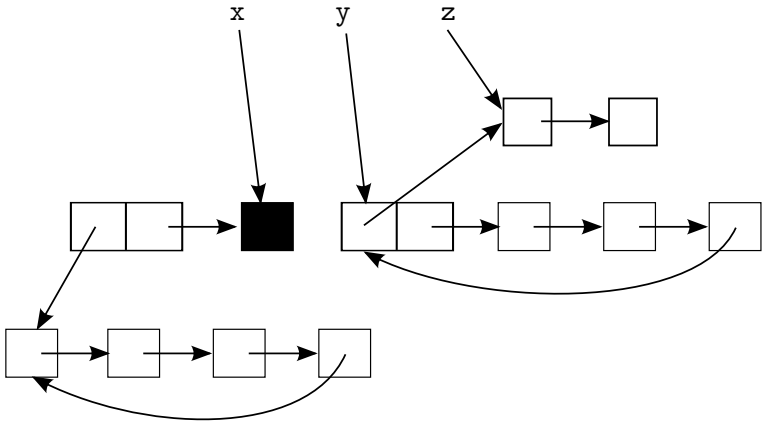


On supprime celles à 0 :

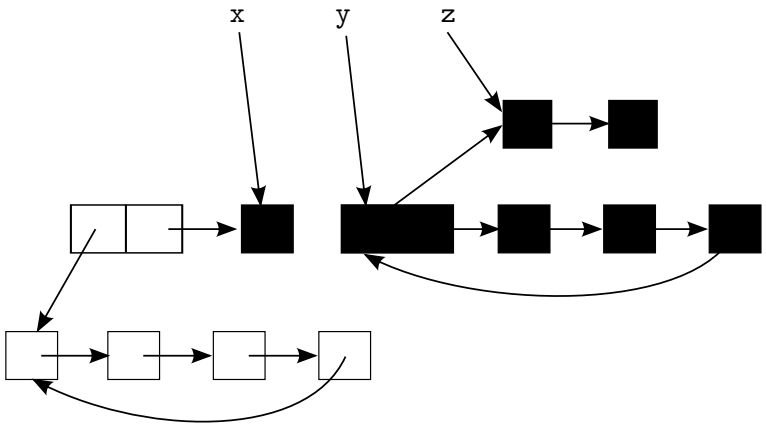


On remarque qu'il reste des parties non accessibles qui n'ont pas été libérées.

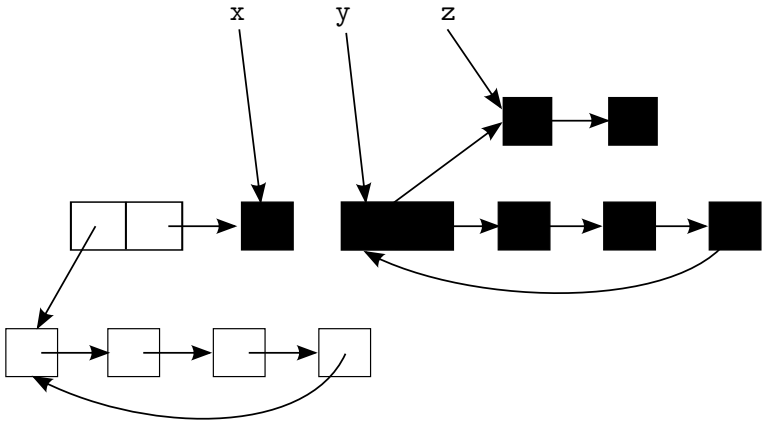
3. On marque à partir de `x` :



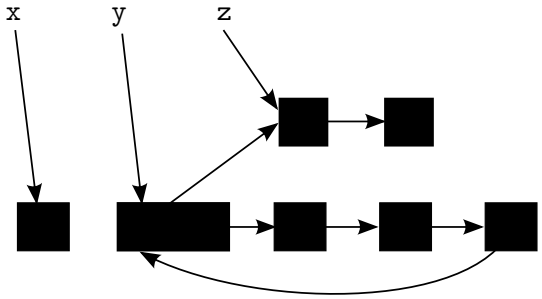
Puis de `y` :



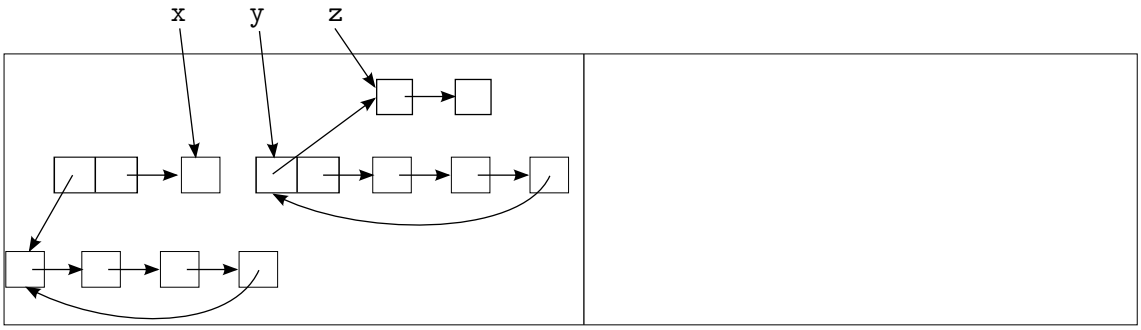
Puis de `z` :



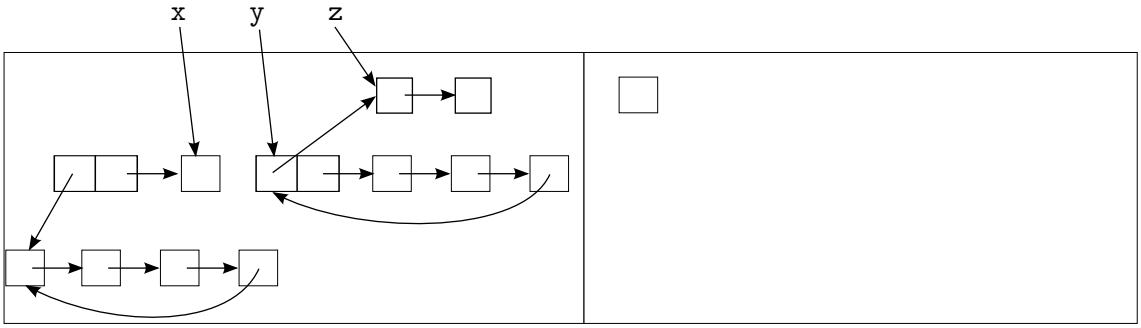
Ensuite on parcours la mémoire et on libère les zones non marquées :



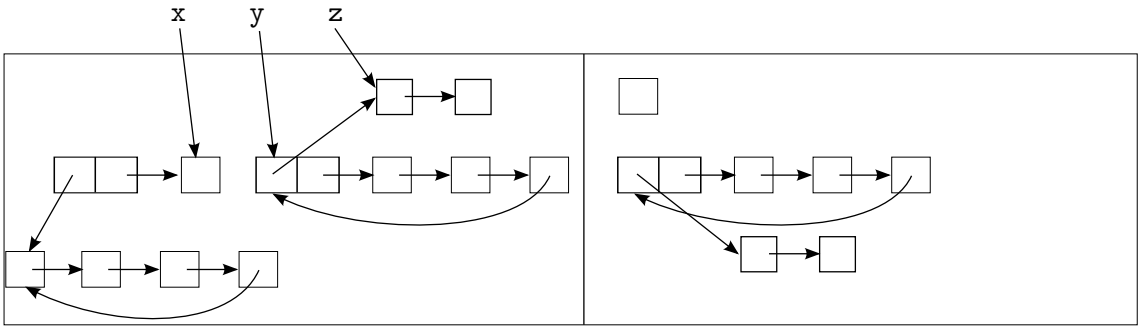
4. Avant lancement du garbage collector, on n'utilise que la moitié de la mémoire :



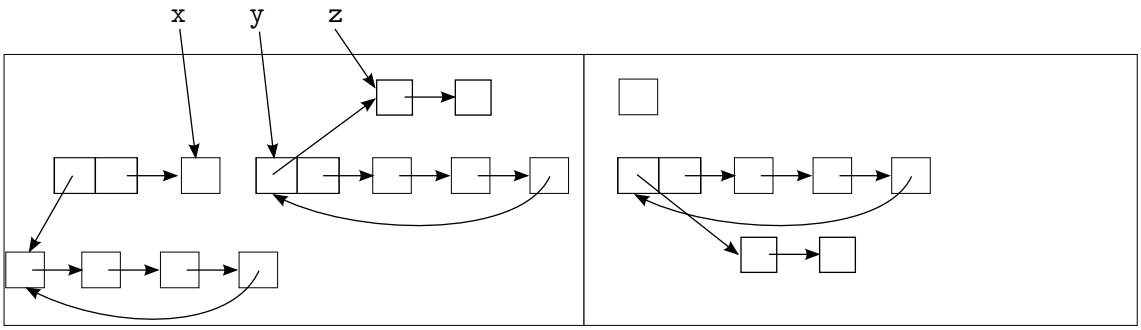
On copie ce qui est accessible depuis `x` dans l'autre moitié :



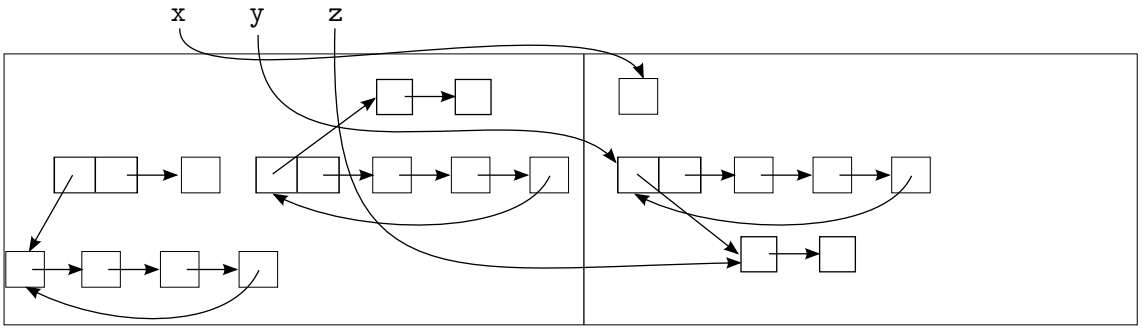
puis ce qui est accessible depuis `y`



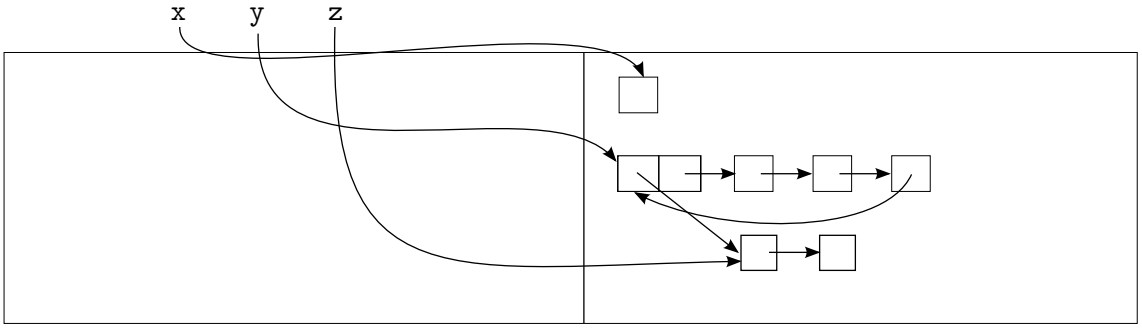
puis ce qui est accessible depuis `z`



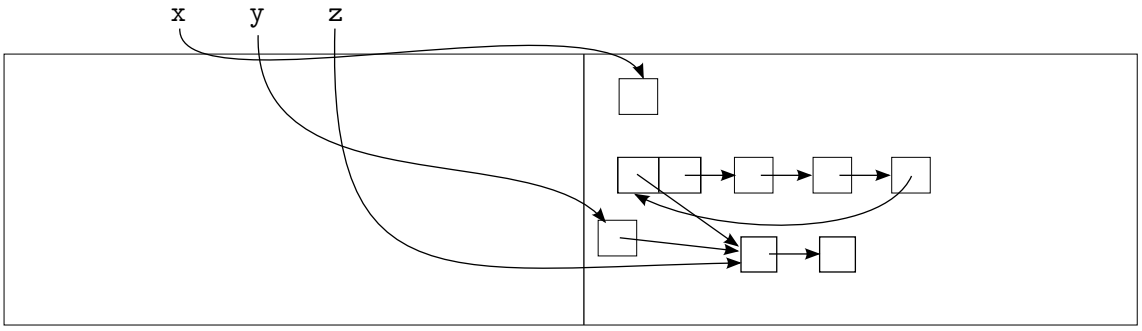
On met à jour les pointeurs des variables :



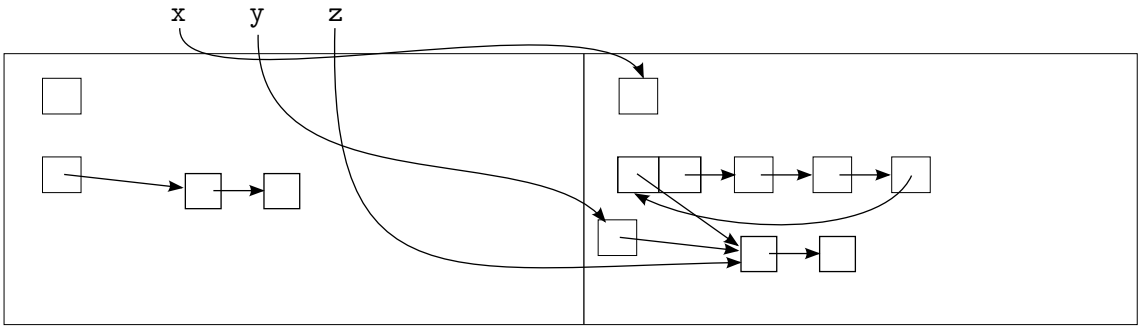
et on libère la première moitié de la mémoire :



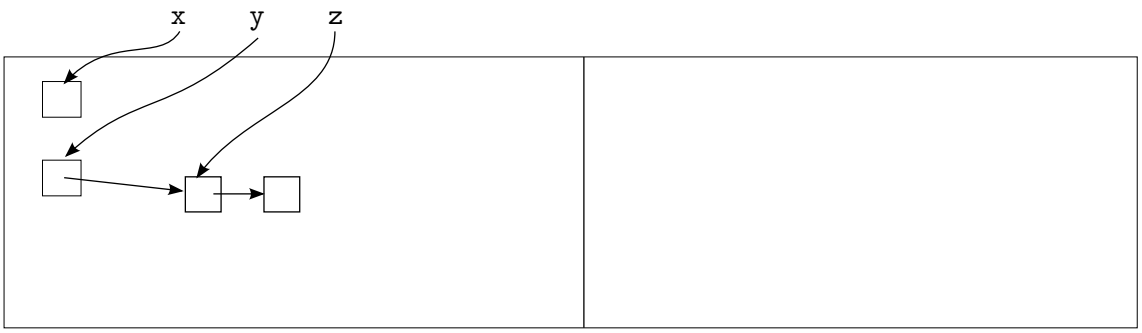
5. On alloue la nouvelle zone dans la moitié dans laquelle on est en train de travailler :



Quand on applique le GC copiant, on copie ce qui est accessible depuis les variables :



puis on met à jour les pointeurs et on libère l'autre moitié :

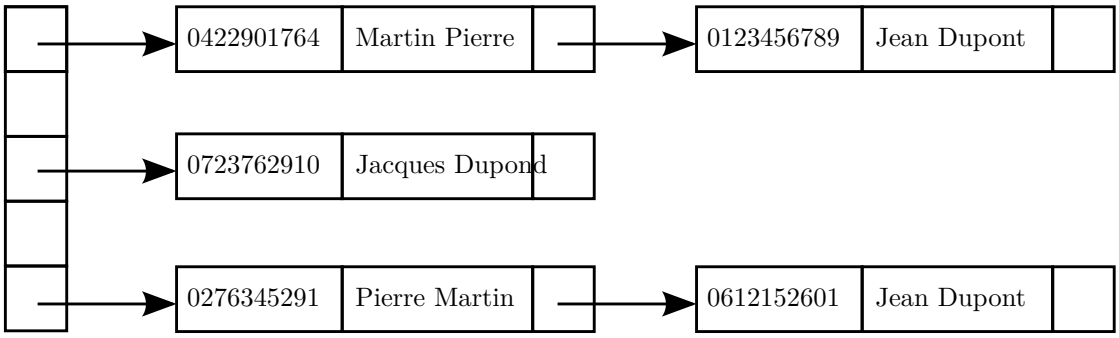


Exercice 4 : Annuaire inversé (4 points)

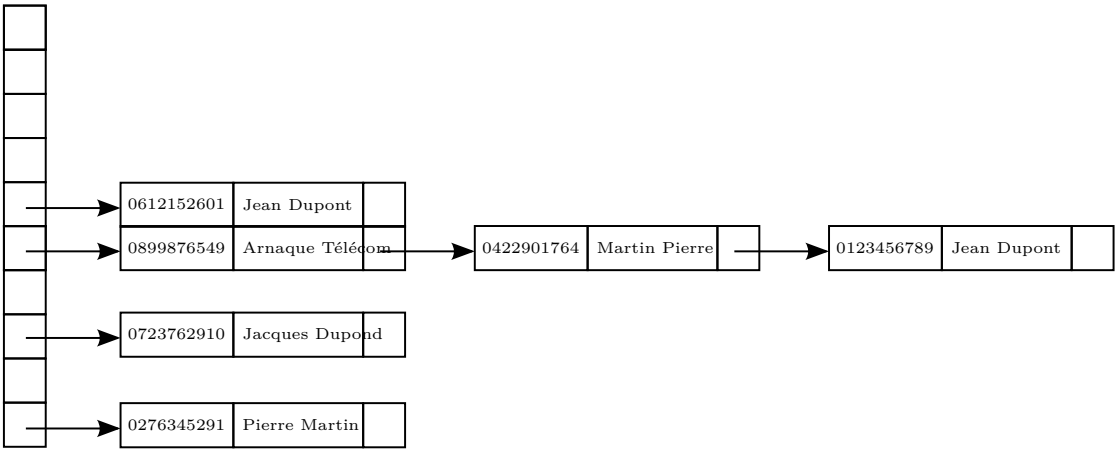
1. Les tables de hachage ont une complexité de recherche en moyenne en $O(1)$ (temps constant). C'est donc mieux que les autres structures vues en cours, et il faut les utiliser dans ce cas.
2. Si on utilisait une telle fonction de hachage, seules les cases d'indices 1 à 8 du tableau serait utilisées, et on aurait beaucoup de collisions (au moins $\left\lceil \frac{100}{8} \right\rceil = 13$ pour une des cases).
3. La fonction de hachage donne les résultats suivants :

Numéro	Somme	Hachage
0123456789	45	0
0612152601	24	4
0276345291	39	4
0723762910	37	2
0422901764	35	0

ce qui conduit à la table de hachage suivante :



4. Le nombre d'éléments insérés devient plus grand que la taille de la table, on redimensionne. On obtient :



Exercice 5 : Piles (7 points)

1. En C :

```
typedef struct donnee_pile* pile;
pile faire_vide();
int est_vide(pile);
pile empiler(int, pile);
int dessus(pile);
pile depiler(pile);
```

En OCaml :

```
type pile
val faire_vide : unit -> pile
val est_vide : pile -> bool
val empiler : int -> pile -> pile
val dessus : pile -> int
val depiler : pile -> pile
```

2. En C :

```
struct donnee_pile {
  int* tableau;
  int tete;
  int taille;
};
```

En OCaml :

```
type pile = int array * int * int
```

(Pour ceux qui connaissent, il est également possible d'utiliser des enregistrements.)

3. En C :

```
pile faire_vide() {
  pile res = malloc(sizeof(struct donnee_pile));
```

```
  res->tableau = malloc(sizeof(int));
  res->tete = -1;
  res->taille = 1;
  return res;
}

int est_vide(pile p) {
  return (p->tete == -1);
}

pile empiler(int i, pile p) {
  int j, newtaille;
  int* newtab;
  p->tete = p->tete + 1;
  if (p->tete < p->taille)
    p->tableau[p->tete] = i;
  else {
    newtaille = p->taille * 2;
    newtab = calloc(newtaille, sizeof(int));
    for(j = 0; j < p->taille; j = j + 1)
      newtab[j] = p->tableau[j];
    free(p->tableau);
    newtab[p->tete] = i;
    p->tableau = newtab;
    p->taille = newtaille;
  };
  return p;
}

int dessus(pile p) {
  return (p->tableau[p->tete]);
}

pile depiler(pile p) {
  p->tete = p->tete - 1;
  return p;
}
```

En OCaml :

```
let faire_vide () =
  Array.make 1 0, -1, 1

let est_vide (_, tete, _) =
  tete = -1
```

```
let empiler i (tableau, tete, taille) =
  let newtete = tete + 1 in
  if newtete < taille then begin
    tableau.(newtete) <- i;
    tableau, newtete, taille
  end
  else begin
    let newtaille = taille * 2 in
    let newtab = Array.make newtaille 0 in
    for j = 0 to taille - 1 do
      newtab.(j) <- tableau.(j)
    done;
    newtab.(newtete) <- i;
    newtab, newtete, newtaille
  end

let dessus (tableau, tete, _) =
  tableau.(tete)

let depiler (tableau, tete, taille) =
  tableau, tete - 1, taille
```

4. Il n'y a qu'un nombre constant d'opérations effectuées, d'où une complexité en $O(1)$, en moyenne et dans le pire des cas.

5. Le pire des cas est celui où il y a redimensionnement. Alors, il faut recopier le tableau de départ, ce qui prend n étapes, plus un nombre constant d'autres opérations. On a donc une complexité dans le pire des cas en $O(n)$.

6. a) `tableau` = $\boxed{}$, `tete` = -1, `taille` = 1

   `tableau` = $\boxed{1}$, `tete` = 0, `taille` = 1

   `tableau` = $\boxed{1}\boxed{2}$, `tete` = 1, `taille` = 2

   `tableau` = $\boxed{1}\boxed{2}\boxed{3}\boxed{}$, `tete` = 2, `taille` = 4

   `tableau` = $\boxed{1}\boxed{2}\boxed{3}\boxed{4}$, `tete` = 3, `taille` = 4

   `tableau` = $\boxed{1}\boxed{2}\boxed{3}\boxed{4}\boxed{5}\boxed{}\boxed{}\boxed{}$, `tete` = 4, `taille` = 8

   b) Pour insérer le i^{e} élément, on ne redimensionne que si i vaut $2^p + 1$ pour un certain p. Si on ne redimensionne pas, on a besoin d'un nombre constant d'opération, mettons 1, sinon on a besoin d'un nombre constant plus la complexité de la recopie de l'ancien tableau, soit au total $1 + (i - 1)$ opérations.

La complexité totale est donc de

$$\begin{aligned}
&\left(\sum_{i=1, i\neq 2^p+1}^{k} 1\right) + \left(\sum_{i=1, i=2^p+1}^{k} 1 + (i-1)\right) \\
&= \left(\sum_{i=1}^{k} 1\right) + \left(\sum_{i=1, i=2^p+1}^{k} i - 1\right) \\
&= k + \sum_{p=0}^{\lfloor \log_2(k) \rfloor} 2^p \\
&= k + 2^{\lfloor \log_2(k) \rfloor + 1} - 1 \\
&\leq k + 2k \\
&= O(k)
\end{aligned}$$

c) Il faut $O(k)$ opération pour insérer k éléments, la complexité en moyenne est donc en $O(\frac{k}{k}) = O(1)$.

7. Les complexités en moyenne sont les mêmes dans les deux implémentations.

   Un inconvénient de l'utilisation d'un tableau est la complexité de l'empilement dans le pire des cas ($O(n)$ contre $O(1)$).

   Un avantage de l'utilisation de tableau est le plus faible recours à des allocations dynamiques : il en faut uniquement pour doubler la taille des tableaux, donc au maximum $\log_2(k)$ fois pour k insertions, et une fois la taille « de croisière » du tableau atteinte, on n'a plus besoin d'allocation dynamique. Pour les listes chaînées, on a une allocation dynamique à chaque insertion.

   Un autre avantage de l'utilisation de tableau est d'avoir une représentation en mémoire plus compacte : tous les éléments de la pile sont dans la même partie de la mémoire (puisqu'ils sont dans le tableau), tandis que dans le cas des listes chaînées on peut avoir des morceaux de la pile éparpillés dans la mémoire. Ceci peut avoir une influence sur les optimisations liées au cache.