Les listes comme structure de données de bases en OCaml

Comme vu en cours, les structures de données de base en OCaml sont les listes et les tableaux.

— Les listes s'écrivent `[x1; x2; ...; xn]` (avec n valeurs), ou `tete :: queue` (avec `tete` une valeur et `queue` une liste), et elles sont de types homogènes (les valeurs ont le même type). Le module `List` contient toutes les fonctions utiles pour les listes, ex. `List.hd` renvoie la tête d'une liste et `List.tl` renvoie la queue de la liste. En OCaml, généralement on parcourt la liste dans une fonction *récursive*, et sans utiliser `List.nth li n` qui donne le `n`-ième élément en temps linéaire en `n` (car il faut reparcourir toute la liste chaînée).
La documentation est en ligne sur `https://ocaml.org/api/List.html`.

Réimplémentation des fonctions sur les listes

Pour chacune des fonctions suivantes qui sont dans le module `List`, écrivez votre propre version en suivant la spécification, c'est à dire son typage et sa documentation) :

1. `val hd : 'a list -> 'a` : *Return the first element of the given list. Raise Failure hd if the list is empty.* Vous pouvez utiliser un filtrage (`match liste with | head :: _ -> ...` par exemple) ou une déconstruction manuelle (`let head :: _ = liste in ...` par exemple).
2. `val tl : 'a list -> 'a list` : *Return the given list without its first element. Raise Failure tl if the list is empty.* Vous pouvez utiliser un filtrage (`match liste with | _ :: tail -> ...` par exemple) ou une déconstruction manuelle (`let _ :: tail = liste in ...` par exemple).
3. `val nth : 'a list -> int -> 'a` : *Return the n-th element of the given list. The first element (head of the list) is at position 0.*
4. `val nth_opt : 'a list -> int -> 'a option` : *Return the n -th element of the given list. The first element (head of the list) is at position 0. Return None if the list is too short. Raise Invalid_argument List.nth if n is negative.* La syntaxe pour lever une exception est `raise (Invalid_argument "message")`. Le type `'a option` est soit `None` pour indiquer une absence de résultat, soit `Some x` pour indiquer la présence d'un résultat `x`.
5. `val rev : 'a list -> 'a list` : *List reversal.* Transforme `[a1; a2; ...; an]` en `[an; an-1; ...; a2; a1]`.
6. `val append : 'a list -> 'a list -> 'a list` : *Concatenate two lists.* Vous pouvez utiliser la fonction précédente `rev`.
7. `val iter : ('a -> unit) -> 'a list -> unit` : *iter f [a1 ; ... ; an] applies function f in turn to a1 ; ... ; an. It is equivalent to begin f a1 ; f a2 ; ... ; f an ; () end.* `iter f liste` consiste à appliquer la fonction `f : 'a -> unit` successivement à toutes les valeurs de la liste. Par exemple `let affiche_liste_entier liste = List.iter (fun x -> print_int x; print_string " -> ") liste` définie une fonction `affiche_liste_entier : 'a list -> unit` qui affiche une liste d'entier comme cela :

```
# let affiche_liste_entier = List.iter (fun x -> print_int x; print_string " -> ") ;;
val affiche_liste_entier : int list -> unit = <fun>
# affiche_liste_entier [1; 2; 3; 4] ;;
1 -> 2 -> 3 -> 4 -> - : unit = ()
```


Manipulation de piles et de files

Les piles sont utilisables directement avec le module `Stack` et les files avec le module `Queue`. Leur interface est quasiment identique, et leur documentation se trouve en ligne sur `https://ocaml.org/api/Stack.html` et `https://ocaml.org/api/Queue.html`.

— Pour manipuler une pile il faut la créer avec `let pile = Stack.create ()`, puis la remplir et la vider progressivement avec respectivement les deux fonctions `Stack.push valeur pile` et `let sommet = Stack.pop pile`. La fonction `Stack.top` permet de regarder le sommet sans le dépiler, et `Stack.length` donne la longueur de la pile.

```
# let pile = Stack.create () ;;
val pile : '_a Stack.t = <abstr>

# Stack.push 10 pile ;;
- : unit = ()
# Stack.push 15 pile ;;
- : unit = ()
# Stack.push 20 pile ;;
- : unit = ()

# Stack.peek pile ;;
- : int = 20
# Stack.peek pile ;;
- : int = 20

# Stack.pop pile ;;
- : int = 20
# Stack.pop pile ;;
- : int = 15
# Stack.pop pile ;;
- : int = 10
# Stack.pop pile ;;
Exception: Stack.Empty.
# Stack.pop pile ;;
Exception: Stack.Empty.
```

— (challenge) La fonction `Stack.iter` fait comme `List.iter` mais sur une pile, et cela, sans la détruire. Utilisez la pour afficher une pile d'entiers sous la forme suivante :

```
# afficher_pile_entiers pile ;;
| 20 |
| 15 |
| 10 |
|____|
```

— Faites les mêmes expériences avec une file, avec le module `Queue` et exactement les mêmes fonctions, et vérifiez que le comportement est celui auquel on s'attend.

