Devoir en temps limité n°4 – 3h

Le code doit être commenté dès qu'il dépasse les 5 lignes ou comprend un concept compliqué. Il est possible (et recommandé) d'utiliser des fonctions auxiliaires en Ocaml, en expliquant leur rôle.

Pour maximiser la compréhensibilité des exercices, le sujet est séparé en trois parties. D'abord les questions de cours, puis les exercices à faire en C, puis les exercices à faire en Ocaml. **Les exercices ne sont pas rangés par ordre de difficulté et sont indépendants.**

1 Questions de cours

1. Qu'est-ce que la barrière d'abstraction ?
2. Quelles sont les primitives de la structure de données liste ?
3. Qu'est-ce qu'un maillon dans une structure simplement chainée ?
4. Énoncer une implémentation possible d'une file et rappeler brièvement sont fonctionnement, en utilisant un ou des dessins.
5. Dessiner le graphe de flot de contrôle du programme suivant et donner un jeu de tests permettant de couvrir tous les arcs.
   Remarque : on suppose que n est la longueur du tableau t. m est un entier quelconque.

```c
int f (int* t, int n, int m){
  if (m >= n){
    int a = t[0];
    int i=1;
    while(i<n){
      a+=t[i];
      i+=1;
    }
    return a;}
  else { if(n>=1){return 0;}
         else {return 2;}}}
```

   Quelle erreur peut on manquer, même avec un jeu de test qui couvre tous les arcs ?

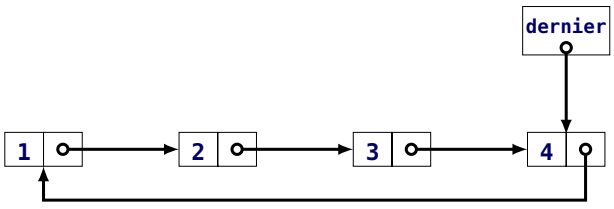
2 Exercices en C

Exercice 1 Implémentation d'une file circulaire en C

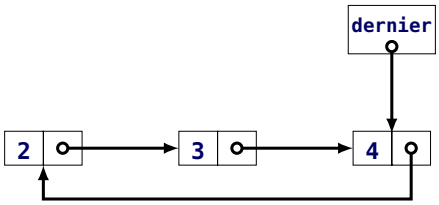
Dans cet exercice on propose d'implémenter un file d'entiers à partir d'une structure chainée circulaire.

L'idée est la suivante : la file est une chaine de maillons, les éléments sont enfilés à droite (après le dernier élément) et défilés à gauche (le premier élément est le premier défilé).

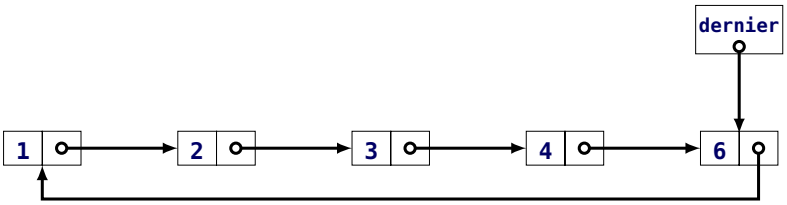
Pour avoir accès à tout moment au premier et au dernier maillon, le dernier élément pointe vers le premier élément et on dispose d'un pointeur vers le dernier élément.



Un exemple de file f_1



Après défilement de f_1



Après enfilement de 6 dans f_1

On propose les types suivants :

```c
typedef struct maillon {int valeur; maillon* suivant;} maillon;
typedef struct file {maillon* dernier;} file;
```

1. Comment représenter la file vide avec cette implémentation ? Écrire la fonction `file* creer()` qui crée et renvoie un pointeur vers une file vide.
2. Écrire la fonction `bool est_vide(file* f)` qui vérifie si la file est vide.
3. Faire un dessin de la file quand elle ne contient qu'un seul élément x. Écrire une fonction `maillon* cree_maillon(int x)` qui crée un maillon de valeur x qui pointe vers lui-même.

Pour vous aider on propose la fonction `int defile(file* f)` suivante :

```
int defile(file* f){
 maillon* premier = f->dernier->suivant;
 int v = premier->valeur;
 if(premier->suivant == premier){
   f->dernier = NULL;
 }
 else{
   f->dernier->suivant = premier->suivant;
 }
 free(premier);
 return v;
 }
```

4. Qu'a t-on oublié de vérifier dans la fonction `defile` proposée ? Proposer des commentaires pour la fonction.
5. Écrire la fonction `int coupdoeil(file* f)` qui regarde le prochain élément sans le défiler.
6. Écrire la fonction `void enfile(file* f, int x)`. On fera attention au cas où la file est vide. On rappelle qu'on enfile à droite, donc l'élément enfilé devient le dernier élément.
7. Écrire la fonction `void detruire(file* f)` qui libère toute la mémoire affectée à la file.
8. Donner la complexité de chacune des fonctions précédentes.

Exercice 2 Les palindromes

> Dans cet exercice on suppose une structure de pile mutable de caractères déjà implémentée en C. Les primitives `pile* creer()`, `bool est_vide(pile* p)`, `void empiler(pile* p, char c)`, `char depiler(pile* p)`, `char coupdoeil(pile* p)` et `void detruire(pile* p)` sont définies et utilisables. **On ne suppose pas une implémentation particulière de la pile**.

On rappelle qu'un palindrome est une chaine de caractère qui se lit de la même manière de droite à gauche ou de gauche à droite, comme "kayak".

Mathématiquement parlant, une chaine $s = c_0c_1...c_{n-1}$ est un palindrome si pour tout $i \in [|0, n-1|]$, $c_i = c_{n-1-i}$.

L'algorithme suivant permet de déterminer si une chaine de caractères s est un palindrome en utilisant une pile :

i) Créer une pile vide
ii) Empiler les lettres du mot une à une jusqu'à atteindre le milieu du mot.
iii) Si le mot est de taille impaire, on ignore le caractère central.
iv) Pour chaque lettre c après le milieu, dépiler un élément c' la pile et comparer avec c. Conclure s'il faut continuer ou pas.
v) Renvoyer vrai si la chaine était un palindrome et faux sinon.

1. À l'étape iv), que peut on dire si $c \neq c'$? Si $c = c'$?
2. On rappelle qu'en C les chaines de caractères du type `char*` se terminent par le caractère `'\0'`. Rappeler comment écrire une fonction `int strlen(char* s)` qui calcule la taille de la chaine `s`.
3. Programmer la fonction `bool est_palindrome(char* s)` qui suit le principe expliqué précedemment.

On va maintenant procéder à la preuve de ce programme.

4. Justifier que votre fonction termine.

Pour la correction, on note n la taille de la chaine de caractère en entrée. L'algorithme présenté contient clairement deux boucles indépendantes, une pour l'étape ii) et une pour l'étape iv). (et j'espère que votre programme est bien structuré de cette manière.)

Un invariant pour la première boucle est "À la fin de l'étape i, la pile contient toutes les lettres de i à 0, dans l'ordre.". La preuve est évidente, on pourra donc supposer qu'au début de l'étape iii) la pile contient toutes les lettres de $\lfloor \frac{n}{2} \rfloor - 1$ à 0, dans l'ordre.

5. Proposer un (ou des) invariant(s) utile(s) pour la deuxième boucle. Conclure quant à la correction totale de la fonction.

3 Exercices en Ocaml

Exercice 3 Le tri à bulles

Le tri à bulles est un algorithme de tri qui permet de trier un tableau t dans l'ordre croissant. Son principe est le suivant :

i) On parcourt le tableau de gauche à droite, en comparant chaque élément $t.(i)$ à son voisin de droite $t.(i+1)$: s'ils ne sont pas dans le bon sens, on les échange.

ii) À la fin du parcours :
- si durant le parcours on a fait au moins un échange, on retourne à l'étape i),
- sinon, le tableau est trié.

Un exemple du fonctionnement est donné après les questions pour ne pas les noyer, n'hésitez pas à le regarder.

1. Écrire une fonction `parcours : 'a array -> bool` qui prend en entrée un tableau t et fait **un parcours** du tableau en faisant les échanges nécessaires. La fonction renvoie un booléen indiquant si un échange a été fait lors du parcours.
2. Écrire une fonction `tri_bulles : 'a array -> unit` qui trie un tableau avec la méthode décrite.
3. Montrer qu'un parcours n'effectue aucun échange si et seulement si t est trié.

Lemme (admis) : On note $a_0 \leq a_1 \leq \ldots \leq a_{n-1}$ les éléments d'un tableau t, ici numérotés selon leur valeur, pas selon leur position dans le tableau, qui peut être quelconque.

Si le sous tableau de t entre les indices $k \in [|0, n-1|]$ et $n-1$ est exactement $[|a_k; a_{k+1}; \ldots; a_{n-1}|]$, alors après un itération de `parcours` sur t, le sous tableau de t entre les indices $k-1$ et $n-1$ est exactement $[|a_{k-1}; a_k; a_{k+1}; \ldots; a_{n-1}|]$.

En particulier si le maximum a_{n-1} du tableau n'est pas initialement en derniere position, alors le parcours met a_{n-1} dans la case $n-1$.

4. (question difficile) Montrer la correction totale de votre algorithme.
   Indication : un invariant utile est qu'à la fin de chaque parcours un certain sous tableau t' de t est trié et que les éléments de t' sont les plus grands éléments de t.

Exemple pour `t=[|3;5;12;1;0|]` :

Durant le premier parcours : on compare 3 et 5, qui sont dans le bon sens, on compare 5 et 12, qui sont dans le bon sens, on compare 12 et 1, qui ne sont pas dans le bon sens. On obtient **`t=[|3;5;1;12;0|]`** puis on compare 12 et 0, qui ne sont pas dans le bon sens. On obtient **`t=[|3;5;1;0;12|]`**. Fin du parcours, on recommence au début.

Durant le deuxième parcours (on ne cite plus les paires qu'on échange pas) : on échange 5 et 1 puis 5 et 0. On obtient **`t=[|3;1;0;5;12|]`** à la fin du parcours.

Durant le troisième parcours : on échange 3 et 1 puis 3 et 0. On obtient **`t=[|1;0;3;5;12|]`** à la fin du parcours.

Durant le quatrième parcours, on échange 1 et 0. On obtient **`t=[|0;1;3;5;12|]`** à la fin du parcours.

Durant le cinquième parcours, on effectue aucun échange. L'algorithme prend fin.

Dans le reste de cette partie on utilisera le module `Stack` (comme une pile en anglais) pré-implémenté de Ocaml : Les primitives ont les noms suivants, le type **`'a Stack.t`** désignant une pile d'éléments de type **`'a`** :

- `Stack.create : unit -> 'a Stack.t` qui crée une pile vide
- `Stack.is_empty : 'a Stack.t -> bool` qui vérifie si une pile est vide
- `Stack.push : 'a -> 'a Stack.t -> unit` qui ajoute un élément (equivalent de empile)
- `Stack.pop : 'a Stack.t -> 'a` qui retire et renvoie l'élément le plus récent (équivalent de depile)
- `Stack.peek : 'a Stack.t -> 'a` qui renvoie sans retirer l'élément le plus récent (équivalent de coup-doeil)

Les fonctions `peek` et `pop` lèvent l'exception `Empty` si la pile est vide.
On suppose que les fonctions `create`, `is_empty`,`push`, `pop` et `peek` s'effectuent en temps constant.
Vous avez le droit d'utiliser des piles auxiliaires.

Exercice 4 Trier les copies

Une professeure de maths a donné deux sujets à ses élèves de MPI : un sujet Mines-Ponts et un sujet CCINP.

Elle a arrangé ses copies dans l'ordre alphabétique, mais avant de commencer à corriger, elle préfère réarranger sa pile de copies pour que les copies du sujet Mines-Ponts soient sur le dessus et les copies CCINP en-dessous mais que les copies Mines-Ponts soient toujours triées dans l'ordre alphabétique, tout comme les copies CCINP.

On modélise les copies par des objets du type énumération `type sujet = CCINP of string | MP of string;;`. Par exemple la copie de Louis, qui a fait le sujet CCINP peut être définie par `let copie = CCINP "Louis";;`.

Écrire une fonction `arrange : sujet Stack.t -> unit` qui prend en entrée la pile de copies et les réarrange. Par exemple si la pile contient initialement `MP "Anna", CCINP "Beatrice", CCINP "Louis", MP "Xavier"` alors à la fin elle doit contenir `MP "Anna", MP "Xavier", CCINP "Beatrice", CCINP "Louis"`.

Remarque : il n'est pas nécessaire de vérifier que la pile est initialement bien dans l'ordre alphabétique.

Exercice 5 Trier une pile

Trier une pile, c'est modifier la pile pour que les éléments les plus grands soient les plus proches de la sortie.

On va suivre un principe similaire au tri par insertion : on sort tous les éléments de la pile, puis on les remet tous, en les insérant chacun à sa place.

Il est autorisé d'utiliser des piles auxilaires, **mais pas des listes ou des tableaux**.

1. Écrire une fonction `insere : 'a -> 'a Stack.t -> unit` qui prend en entrée un élément `x` et une pile `p` triée et insère l'élément `x` dans `p` de sorte à ce qu'elle reste triée. La pile doit contenir à la fin les éléments qu'elle contenait déjà, avec en plus `x`.
2. En déduire une fonction `tri_pile : 'a Stack.t -> unit` qui trie la pile.
3. Quelle est la complexité du tri ? Préciser le pire cas et le meilleur cas.

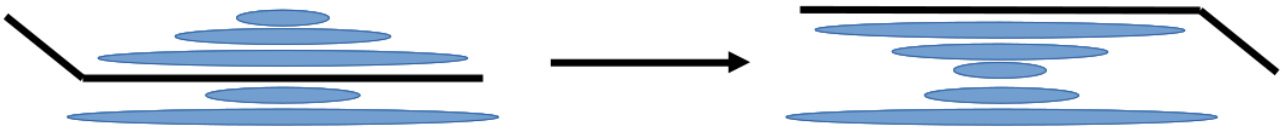
Exercice 6 Le tri de crêpes

Dans cet exercice on a préparé une pile de crêpes de diamètres différents. On souhaite trier la pile pour que la plus petite crêpe soit sur le dessus et la plus grande en dessous. Une crêpe sera représentée par un entier : son diamètre, donc une pile de crêpes est représentée par une pile d'entiers.

Par exemple la pile de gauche doit devenir la pile de droite :

On dispose uniquement d'une spatule que l'on peut insérer dans le pile de crêpes de façon à retourner l'ensemble des crêpes qui sont au-dessus de la spatule.

Par exemple le retournement suivant :



1. Avant de commencer le tri il nous faut une fonction qui effectue le travail de la spatule. Écrire une fonction `retourne : int Stack.t -> int -> unit` qui prend en entrée un pile p et un entier i et retourne les i premières crêpes.
   Par exemple si on a du haut vers le bas des crêpes de diamètres 1,6,2,12,3 et **i = 3**, la pile doit devenir 2,6,1,12,3.

Le principe du tri est le suivant. On remarque qu'on ne crée pas d'autre pile (ni tableau, ni liste) :

- on recherche la plus grande crêpe ;
- on retourne la pile à partir de cette crêpe de façon à mettre cette plus grande crêpe tout en haut de la pile ;
- on retourne l'ensemble de la pile de façon à ce que cette plus grande crêpe se retrouve tout en bas ;
- la plus grande crêpe étant à sa place, on recommence le principe avec le reste de la pile.

On suppose écrite une fonction `taille_pile : int Stack.t -> int` qui calcule la taille de la pile.

2. Écrire une fonction `indice_plus_grande_crepe : int Stack.t -> int -> int` qui prend en entrée la pile et un indice i, cherche et renvoie quelle crepe est la plus grande parmi les i premières (i inclus). La crepe sur le dessus est numérotée 0, celle juste en dessous est numérotée 1, etc... Cette question peut utiliser une pile auxiliaire.
3. Écrire une fonction `tri_crepes : int Stack.t -> unit` qui trie la pile de crêpes **en suivant le principe décrit**.

4. Quelle est la complexité du tri ? Préciser le pire cas et le meilleur cas.
5. Faire la preuve de correction totale de l'algorithme de tri de crêpes. On pourra supposer que la fonction `retourne` est correcte.