TP 7 : Tableaux dynamiques

La structure abstraite de *tableau dynamique* est une extension de la structure abstraite de *tableau* : on peut toujours accéder facilement (et rapidement) à un élément quelconque par son indice, mais il est également possible d'ajouter ou de supprimer un élément. En règle générale, cette opération n'est possible qu'à l'extrémité droite du tableau.

Opération	**Type**	**Effet**
`get`	DYNARRAY × `int` → ELT	Accès à un élément
`set`	DYNARRAY × `int` × ELT → {}	Modification d'un élément
`push`	DYNARRAY × ELT → {}	Ajout d'un élément à la fin du tableau
`pop`	DYNARRAY → ELT	Récupération et suppression de l'élément le plus à droite
`create`	{} → DYNARRAY	Création d'un tableau vide (fonction ne prenant pas d'argument)
`length`	DYNARRAY → `int`	Nombre d'éléments présents

1 RÉALISATION NAÏVE

On utilise la structure suivante :

```
struct int_dynarray {
  int len;
  int capacity;
  int* data;
};

typedef struct int_dynarray int_dynarray;
```

- L'entier `len` représente le nombre d'éléments *actuellement présents* dans le tableau.
- L'entier `capacity` représente la *capacité* du tableau, c'est-à-dire la taille du bloc vers lequel pointe `data`.
- Les éléments sont stockés à partir du début du bloc : il peut y avoir de la place libre à la fin du bloc (si `capacity > len`). Dans ce cas, les valeurs présentes dans les cases « libres » n'ont aucun sens.

Pour gérer les `pop` et les `push`, on peut imaginer procéder ainsi :

- un `pop` fait diminuer `len` de 1, et ne change pas `capacity` (il y a une erreur si `len` vaut zéro) ;
- pour un `push`, il y a deux cas :
  - si $\texttt{len} < \texttt{capacity}$, on écrit le nouvel élément à droite et l'on incrémente `len` ;
  - si $\texttt{len} = \texttt{capacity}$, on alloue un nouveau bloc `data`, de taille $\texttt{capacity} + 1$, on recopie l'ancien dans le nouveau et on y ajoute l'élément supplémentaire.

▷ **Question 1.**

1. Ecrire une fonction `length` qui renvoie la longueur du tableau dynamique dont l'adresse est passée en entrée.

   ```
   int length(int_dynarray* t);
   ```

2. Ecrire une fonction `make_empty` renvoyant un pointeur vers un `int_dynarray` vide. On initialisera `data` à `NULL`.

   ```
   int_dynarray* make_empty(void)
   ```

3. Ecrire les fonctions `get` et `set`.

   ```
   int get(int_dynarray* t, int i);
   void set(int_dynarray* t, int i, int x);
   ```

4. Ecrire la fonction `pop` (qui ne modifiera jamais la capacité du tableau) :

   ```
   int pop(int_dynarray* t);
   ```

5. Ecrire une fonction `resize(int_dynarray_t* t, int new_capacity)` qui alloue un nouveau bloc `data`, copie le contenu de l'ancien bloc dans le nouveau, met à jour les champs de `t` et libère l'ancien bloc.

   ```
   void resize(int_dynarray* t, int new_capacity);
   ```

6. Ecrire la fonction `push` :

   ```
   void push(int_dynarray* t, int x);
   ```

7. Ecrire une fonction `delete(int_dynarray* t)` qui détruit le tableau dynamique pointé par `t` en libérant ce qu'il faut libérer.

   ```
   void delete(int_dynarray* t);
   ```

◁

▷ **Question 2.**
Analyse de la réalisation naïve : On considère une série de n opérations `push` successives sur un tableau dynamique initialement vide. Déterminer le coût total de ces opérations.
On considérera pour simplifier que les opérations `malloc` et `free` peuvent se faire en temps constant, mais cela ne change de toute façon pas le résultat ici.
◁

2 RÉALISATION EFFICACE

Le résultat de la question précédente montre que la réalisation précédente n'est pas satisfaisante. Il faudrait que les opérations `push` et `pop` se fassent rapidement. Il n'est pas vraiment possible d'obtenir une complexité constante dans le pire des cas pour ces fonctions, mais on peut obtenir une complexité *amortie* en $O(1)$ pour `push` en utilisant la stratégie suivante :

- s'il reste de la place libre, on ajoute l'élément dans le tableau.
- sinon, on procède aussi comme pour la solution naïve, sauf que le nouveau bloc alloué est de taille `2 * capacity`. Il faut ajouter un cas particulier si `capacity` est nul.

▷ **Question 3.** Apporter les modifications nécessaires à la fonction `push` pour utiliser la nouvelle stratégie.
◁

▷ **Question 4.**
Analyse amortie sans réduction de taille :

1. Quelles sont les complexités des opérations `pop`, `get` et `set` ?
2. Si t est un tableau dynamique de longueur n, quelle est la complexité d'un `push` dans le pire cas ? dans le meilleur cas ?
3. On considère une série de n opérations `push` et `pop` sur un tableau initialement vide. Il n'y a aucune contrainte sur les opérations effectuées (sauf qu'on ne fait pas de `pop` sur un tableau vide) : on peut avoir n `push`, ou $n/2$ `push` suivis de $n/2$ `pop`, ou une alternance de `push` et de `pop` ...
   Montrer que le coût total de cette série de n opérations est en $O(n)$.

Comme une série de n opérations (sur un tableau initialement vide) a un coût total en $O(n)$, on dira que la *complexité amortie* d'une opération `push` (ou `pop`) est en $O(1)$.
◁

Cette complexité amortie est satisfaisante, mais notre stratégie a un gros défaut : la mémoire utilisée peut-être arbitrairement plus grande que celle nécessaire à stocker le nombre actuel d'éléments. En effet, la taille du bloc alloué ne diminue jamais lors d'une opération `pop`, et l'on peut donc avoir un tableau vide occupant une place proportionnelle au nombre maximum d'éléments qu'il a contenus par le passé.

▷ **Question 5.** On propose la stratégie suivante :

- si `len` devient strictement inférieur à `capacity / 2` après un `pop`, on ré-alloue le bloc de données en lui donnant une taille `capacity / 2`.
- sinon, on procède comme avant.

1. Apporter les modifications nécessaires à la fonction `pop`.
2. Montrer qu'une série de n opérations successives peut avoir un coût de l'ordre de n^2, et le mettre en évidence expérimentalement. On pourra pour cela considérer les opérations suivantes : on ajoute $2^{k-1}+1$ éléments puis on fait une succession de deux `pop` suivis de deux `push`.

◁

▷ **Question 6.** Pour régler ce problème, on modifie légèrement la stratégie :

- si `len` devient strictement inférieur à `capacity/4`, on ré-alloue un bloc de taille `capacity/2`.
- sinon, on supprime l'élément sans ré-allouer.

Nous allons montrer que la complexité amortie des opérations `pop` et `push` est en $O(1)$.
Considérons Φ une fonction de potentiel définie par :

$$\Phi(t) = |2len(t) - capacity(t)|.$$

On considère $t_0, \dots, t_{n-1}$ les états successifs du tableau (t_0 est donc vide), et l'on note C_i le coût de l'opération qui fait passer de t_i à t_{i+1}. On prend toutes les constantes multiplicatives égales à 1 pour les complexités, ce qui est possible sans perte de généralité (on pourrait multiplier le potentiel par une constante), et l'on distingue les cas suivant la nature de la i-ème opération. Montrer les assertions suivantes :

1. Si c'est un `push` sans redimensionnement, alors :
$$C_i + \Phi(t_{i+1}) - \Phi(t_i) \leq 3$$
2. Si c'est un `push` avec redimensionnement, alors :
$$C_i + \Phi(t_{i+1}) - \Phi(t_i) \leq 3$$
3. Si c'est un `pop` sans redimensionnement, alors :
$$C_i + \Phi(t_{i+1}) - \Phi(t_i) \leq 3$$
4. Si c'est un `pop` avec redimensionnement, alors :
$$C_i + \Phi(t_{i+1}) - \Phi(t_i) \leq 1$$

En déduire, en utilisant un télescopage que :

$$\sum_{i=0}^{n-1} C_i \leq 3n = O(n)$$

On a donc bien une complexité amortie en $O(1)$. ◁

3 Opérations supplémentaires

Les opérations que nous avons définies jusqu'ici sont les seules opérations élémentaires sur un tableau dynamique.

▷ **Question 7.** Dans cet exercice, on considère que les seuls moyens d'interagir avec un tableau dynamique sont les fonctions définies dans la signature donnée dans la partie précédente. Autrement dit, l'implémentation est « cachée » : on ne sait même pas que `int_dynarray` est défini comme une `struct`.

1. Ecrire une fonction permettant d'insérer un nouvel élément à un emplacement arbitraire i du tableau. Les éléments présents aux indices $j \geq i$ seront décalés d'une case vers la droite.

   ```
   void insert_at(int_dynarray* t, int i, int x)
   ```

   Remarque : 1. *Les valeurs acceptables pour i vont de 0 à la longueur du tableau incluse (dans ce cas, l'insertion revient à un* `push`*).*

2. Déterminer la complexité de `insert_at` (en fonction de i et $len(t)$).

3. Ecrire une fonction permettant de supprimer un élément à un emplacement arbitraire, en récupérant sa valeur. Les éléments situés à droite seront décalés vers la gauche.

   ```
   int extract_at(int_dynarray* t, int i)
   ```

   Remarque : 2. *Les valeurs acceptables pour i vont de 0 à $n-1$ (où n est la longueur du tableau). Si $i = n-1$, l'opération équivaut à un* `pop`*.*

4. Déterminer la complexité de cette fonction.

◁

▷ **Question 8.**

Une variante du tri insertion :

On se propose d'écrire une variante du tri insertion sur les `int_dynarray`. Ce tri ne sera pas en place : on renverra un nouveau tableau (trié) sans modifier celui passé en paramètre.

L'idée est la suivante, en notant `in` le tableau à trier :

- on crée un tableau vide `out`, tout au long de l'exécution de l'algorithme, ce tableau sera trié ;
- pour chaque élément de `in` :
  - on détermine à quelle position de `out` il faut l'insérer pour que `out` reste trié ;
  - on effectue l'insertion (à la position déterminée)
- on renvoie le tableau `out`.

1. Ecrire une fonction `position(int_dynarray* t, int x)` qui renvoie le plus grand entier i tel que l'insertion de x en position i laisse le tableau `t` trié (en supposant qu'il était trié avant l'appel).

   ```
   int positon(int_dynarray* t, int x);
   ```

2. Ecrire une fonction `insertion_sort(int_dynarray* t)` qui trie un tableau suivant l'algorithme décrit ci-dessus.

   ```
   int_dynarray* insertion_sort(int_dynarray* t);
   ```

3. Déterminer la complexité de cette fonction (dans le pire cas). On distinguera le nombre de comparaisons entre éléments du tableau effectuées des autres opérations élémentaires.

4. Modifier la fonction `position` pour qu'elle effectue un nombre de comparaisons en $O(\log n)$ (où n est la longueur du tableau dans lequel on insère).

5. Peut-on imaginer une situation où cette amélioration est significative ?

◁