Feuille de travaux pratiques n° 4
Tableaux dynamiques et structures chaînées

1 Tableaux dynamiques

Le but de cet exercice est de vous faire comprendre les tableaux dynamiques, autrement dit des tableaux dont la taille augmente selon les besoins. On va simuler leur fonctionnement.

1.1 Principe de l'algorithme

1. On crée un dynamic array de n cases.
2. On reçoit des valeurs et on les place successivement dans le dynamic array. Le nombre de valeurs à placer n'est pas connu à l'avance. Si le nombre de valeurs est supérieur à n, alors on agrandit le dynamic array de la façon suivante :
   — on crée un autre dynamic array dont la taille est doublée ;
   — on copie les valeurs de l'ancien dynamic array dans le nouveau ;
   — on va à l'étape 2.

1.2 Exercices

1. Écrivez une fonction `doublerTableau` qui prend en paramètre un tableau d'entier `t` et sa taille `n` et qui retourne un tableau contenant deux fois plus d'élément que `t` et qui contient les `n` premiers éléments de `t`.
2. Qui doit gérer la mémoire de `t` ?
3. On aimerait ne pas avoir à ce soucier de la mémoire prise par `t` et que la fonction `doublerTableau` le fasse à notre place. Que se passe t'il si on libère la mémoire prise par `t` dans la fonction précédente ?
4. Proposez une solution qui prend `t` en paramètre et qui le ré-alloue. Attention, il faut libérer la mémoire prise auparavant.
5. Supposez qu'à la fin vous ayez `m` éléments en ayant commencé avec `n = 1`. Combien de fois avez-vous allouer un nouveau tableau ? Calculez le nombre total de cases mémoires consommées. Combien de copies de valeurs ont été faites ?
6. Que pensez-vous des dynamic arrays ?
7. On utilise maintenant une structure pour représenter un tableau. Appelons cette structure dynamicArray. Elle contiendra : un entier qui définit la taille allouée pour le tableau et un pointeur vers un entier qui représente le tableau. Écrivez la fonction `reallouer` qui prend en paramètre un dynamic array et qui double sa taille comme proposée précédemment. N'oubliez pas de bien gérer la mémoire.

2 Structures Chaînées

Le but de cet exercice est de programmer les fonctions de base permettant de définir et de manipuler des listes chaînées.

Commencez par créer deux structures. L'une appelée `elementListe` contenant un entier nommé `contenu` et un pointeur, nommé `suivant`, vers la structure `elementListe`. L'autre est appelée `liste` et contient un pointeur, nommé `premier`, vers `elementListe`.

2.1 Exercices

Écrivez les fonctions suivantes, en faisant bien attention à la gestion mémoire :

1. `donnée(x)` : renvoie la donnée associée à l'élément `x`.
2. `suivant(x)` : renvoie l'élément suivant l'élément `x`.
3. `premier(L)` : renvoie le premier élément de la liste, renvoie `nil` si la liste est vide.
4. `estVide(L)` : renvoie vrai si la liste est vide et faux sinon.
5. `supprimerPremier(L)` : supprime le premier élément de `L`.
6. `vider(L)` : supprime tous les éléments de `L`.
7. `ajouterEnTête(x, L)` : ajoute `x` au début de `L`.
8. `ajouterAprès(x, y, L)` : ajoute `x` dans `L` après `y`.
9. `supprimerSuivant(x, L)` : supprime le suivant de `x` dans `L`.
10. `numÉlément(L)` : renvoie le nombre d'éléments de `L`. Quelle est la complexité de cette fonction ?

2.2 Comptage des éléments

On veut améliorer la complexité de la fonction `numÉlément(L)`. Pour ce faire, on introduit une nouvelle donnée dans la structure `liste` : `num` qui contient le nombre d'éléments de la liste. Réécrivez toutes les fonctions précédentes en tenant compte de cette primitive.

2.3 Ajouts

De quelles données devrait-on disposer pour pouvoir ajouter un élément à la fin de la liste ? Quelles sont les modifications à apporter aux fonctions précédentes dans ce cas ? Attention toutes les fonctions peuvent être touchées.

2.4 Fusion de listes triées

On suppose que les deux listes L_1 et L_2 sont triées par ordre croissant et que l'on dispose de la fonction booléenne `donnéePlusGrande(x, y)` qui est vraie si la donnée associée à `x` est plus grande que la donnée associée à `y`. Le but est d'arriver à fusionner les deux listes de façon à obtenir une seule liste triée.

1. Considérez les deux listes dont les données sont les lettres suivantes : L_1 `: a, c, g, j, k` et L_2 `: b, h, m, n`. Représentez graphiquement les listes.
2. Proposez une méthode pour insérer les éléments de L_2 dans L_1 pour que L_1 soit triée à la fin. Donnez deux versions, une qui vide la liste L_2 et une autre qui garde intacte la liste L_2.

3 Listes Doublement chaînées

Une liste doublement chaînée est une liste qui, en plus de permettre l'accès au suivant d'un élément, permet l'accès au précédent d'un élément. Pour représenter cette nouvelle liste, on introduit la donnée `précédent` dans la structure `elementListe`.

1. Quel est l'intérêt de ce type de liste par rapport aux listes simplement chaînées ?
2. Écrivez les fonctions suivantes en mettant éventuellement à jour les primitives précédentes :
   (a) `supprimerPremier(L)` : supprime le premier élément de `L`.
   (b) `vider(L)` : supprime tous les éléments de `L`.
   (c) `ajouterAprès(x,y,L)` : ajoute `x` dans `L` après `y`.
   (d) `supprimer(x,L)` : supprime `x` dans `L`.

Ajout en Fin

On veut maintenant gérer le dernier élément de la liste. On pourrait introduire une donnée contenant le dernier élément, mais comme vous l'avez montré cela complique les fonctions.

Pour résoudre ce problème, on peut travailler avec des listes circulaires : on connaît le premier et le précédent du premier est le dernier élément de la liste.

Afin de résoudre les problèmes de l'existence d'une donnée dans la liste, on introduit un élément fictif que l'on appelle sentinelle. Le premier élément réel de la liste devient donc le suivant de la sentinelle et le dernier élément de la liste est le précédent de la sentinelle. On peut donc supprimer la donnée `premier`. On la remplace par la donnée `sentinelle`.

1. Faites un dessin qui montre cela.
2. Écrivez les fonctions suivantes en mettant éventuellement à jour les primitives précédentes :
   (a) `premier(L)` : renvoie le premier élément de `L`.
   (b) `dernier(L)` : renvoie le dernier élément de `L`.
   (c) `estVide(L)` : renvoie vrai si la liste est vide et faux sinon.
   (d) `supprimerPremier(L)` : supprime le premier élément de `L`.
   (e) `vider(L)` : supprime tous les éléments de `L`.
   (f) `ajouterAprès(x,y,L)` : ajoute `x` dans `L` après `y`.
   (g) `supprimer(x,L)` : supprime `x` dans `L`.

4 Opérations sur les listes

On considère que $\mathtt{L}_1$ et $\mathtt{L}_2$ sont deux listes doublement chaînées circulaires avec sentinelle.

1. Faites un dessin qui montre comment vous allez ajouter $\mathtt{L}_2$ à la fin de $\mathtt{L}_1$.
2. Écrivez ensuite la fonction `ajouterEnFin(`$\mathtt{L}_1$`,`$\mathtt{L}_2$`)`.
3. Faites la même chose pour ajouter $\mathtt{L}_2$ au début de $\mathtt{L}_1$. On nommera la fonction `ajouterAuDébut(`$\mathtt{L}_1$`,`$\mathtt{L}_2$`)`.
4. Faites une fonction qui supprime de $\mathtt{L}_1$ tous les éléments dont la donnée associée est impaire (on utilisera la fonction `donnéeEstPaire(x)`) et qui ajoute tous ceux de $\mathtt{L}_2$ dont la donnée associée est paire.