TD 3 : listes, piles, files, implantation

Initiation au développement

BUT informatique, première année

Ce TD vise à mettre en œuvre les concepts vus dans le cours *STR1 : listes, piles, files*. Pour cela nous allons ajouter au module créé lors des précédents TDs (sur la recherche et le tri) des types définissant des listes, piles et files d'entiers, ainsi que les méthodes et fonctions associées. Nous mettrons en place des jeux de tests pour chacune de nos fonctions et nous évaluerons leurs performances.

Un texte avec cet encadrement contient une information importante.

Un texte avec cet encadrement est une remarque.

Un texte avec cet encadrement est un travail que vous avez à faire.

1 Organisation de la bibliothèque

Nous étendons la bibliothèque créé au dernier TD.

Dans ce troisième TD nous allons implanter quatre structures de données avec les méthodes et fonctions associées :

— listes chaînées (*head*, *tail*, *append*, *isEmpty*, *nil*),
— listes doublement chaînées (*head*, *tail*, *append*, *isEmpty*, *nil*),
— piles sur le modèle des listes (*push*, *pop*, *isEmpty*, *nil*),
— files sur le modèle des listes (*push*, *pull*, *isEmpty*, *nil*).

Plusieurs organisations en paquets sont imaginables, il va vous falloir en choisir une. Par ailleurs, c'est à vous de choisir pour chaque structure de données les fonctionnalités que vous implantez comme des méthodes et celles que vous implantez comme de fonctions. Vous devez pouvoir justifier ce choix.

Par ailleurs, nous implanterons aussi une nouvelle fonctionnalité :

— tri par insertion sur les listes doublement chaînées.

Proposez et justifiez une organisation en paquets. Créez les dossiers pour recevoir ces paquets.

2 Implantation et test d'une structure de données

Ce travail est à faire pour chaque structure de données à implanter.

Le développement d'une structure de données se fait par des aller-retours réguliers entre la définition de tests et le codage de la structure et de ses fonctionnalités : les tests peuvent nous indiquer des erreurs dans le codage, la réflexion sur le codage peut nous suggérer de nouveaux tests à réaliser.

2.1 Phase de réflexion

Avant de se lancer dans le codage, il est important d'avoir une phase de réflexion : bien réfléchir aux différents champs nécessaires pour notre structure de données, bien définir les entrées et sorties de notre (ou de nos) fonction(s) et réfléchir aux premiers tests nécessaires.

Parfois, parmi les sorties d'une fonction, on peut avoir besoin de retourner une erreur. Pour cela, on peut utiliser la fonction New du paquet errors [1] (avec un s) et définir une variable de type error (sans s).

Définissez les champs de votre structure de données.

Pour chaque fonction ou méthode de l'interface de votre structure de données, définissez les paramètres et les valeurs de retour. En particulier, demandez-vous s'il faut pouvoir retourner une erreur ou si on pourra toujours retourner une valeur correcte quelles que soient les paramètres.

Définissez un premier jeu de tests. Demandez-vous pour cela quelles sont les valeurs limites à tester (tableau vide, élément absent ou présent, etc).

2.2 Codage et test

Une fois la phase de réflexion préliminaire réalisée, on peut commencer à coder. Puis tester avec notre jeu de tests.

Si nos fonctions sont un peu complexes (plus de 5 à 10 lignes de codes) il est important de faire des petits tests au fur et à mesure de leur développement. On ne parle pas ici de définir un jeu de tests, mais d'essayer les fonctions sur des valeurs simples et de faire des affichages à des passages clés du code pour bien s'assurer que ce qui se passe est ce qu'on avait prévu. Pour cela vous pouvez éventuellement créer un main à la racine de votre module et l'utiliser pour appeler vos fonctions.

Codez votre structure de données et ses fonctionnalité puis, une fois que vous pensez avoir terminé, testez-la avec votre jeu de tests. Si vous vous rendez compte en codant que vous avez oublié des tests importants, ajoutez-les à votre jeu de tests immédiatement.

2.3 Couverture

Comme nous l'avons vu en cours, ce n'est pas parce que nos fonctionnalités passent tous nos tests qu'elles sont correctes. On peut oublier des tests importants. La commande go test permet de détecter automatiquement une partie de ces oublis en évaluant la couverture du code, c'est-à-dire en indiquant les parties du code (fonctions, branches de conditionnelles) qui ne sont utilisées par aucun test. Un code non couvert par les tests peut être inutile à la fonctionnalité (et dans ce cas il faut le supprimer) ou bien il doit être testé (et dans ce cas il faut ajouter des tests dans le jeu de tests).

1. `https://pkg.go.dev/errors`

Calculer la couverture. Pour calculer la couverture d'un jeu de tests il suffit d'utiliser la commande go test avec l'option -cover, c'est-à-dire de faire go test -cover. On voit alors un pourcentage s'afficher à l'écran, qui indique la proportion de notre code qui est couvert par les tests.

> Calculez la couverture de votre code par vos tests.

Visualiser la couverture. Si le code n'est pas complètement couvert par les tests, il peut être intéressant de savoir quelle partie de celui-ci n'est jamais utilisée. Ceci peut se faire en utilisant la commande go test -coverprofile=out, qui crée un fichier out contenant des informations sur le code non couvert.

On peut directement lire ce fichier, mais ce n'est pas toujours pratique, surtout pour des codes un peu longs. Une autre solution consiste à utiliser ensuite la commande go tool cover -html=out qui va permettre de visualiser dans un navigateur internet la couverture du code.

> Si nécessaire, visualisez la couverture de votre code par vos tests. Déterminez alors si le code non couvert est utile ou non et ajoutez si besoin des cas de test pour le couvrir.