TD5: listes et structures de données dynamiques

Exemple d'utilisation d'une pile : vérification des parenthèsages

On s'intéresse à des textes qui contiennent des délimiteurs ouvrants et fermants qui sont des parenthèses, accolades et crochets. On voudrait savoir si ces textes sont bien parenthésés c'est à dire que chaque délimiteur ouvert est fermé de façon cohérente. Par exemple, "([au (ha ha)][[123]])" est bien parenthésé.

Récupérez l'archive contenant les classes à compléter. La classe "Perenthesage" contient déjà quelques méthodes élémentaires pour savoir si un caractère est un délimiteur, ouvrant ou fermant.

1. On commence par un cas simple où l'on vérifie uniquement un type de délimiteur. Par exemple, "verificationParen" vérifie si les parenthèses sont ouvertes et fermées correctement, en ignorant les autres délimiteurs.

   Ce cas est très simple, il suffit de regarder les caractères les uns après les autres et de gérer un compteur. S'il s'agit d'un délimiteur, et que c'est une parenthèse ouvrante, on incrémente le compteur, si c'est une parenthèse fermante, on décrémente le compteur. Pour savoir si le parenthésage est correct, il faut utiliser ce compteur. Nota: la méthode doit s'arrêter dès qu'on est sûr que le parenthésage est incorrect.

2. On considère maintenant les trois délimiteurs mais on ignore les erreurs d'inter-croisement des délimiteurs. Donc par exemple, "(([))]" sera considéré comme correct car chaque délimiteur pris séparément est bien parentésé mais les "((" sont fermées avant le "[". Ce cas est encore simple, il suffit de gérer trois compteurs, un pour chaque délimiteur, comme précédemment.

3. On veut maintenant tenir compte des inter-croisements. C'est là que la pile intervient. La bonne solution à ce problème est en effet d'utiliser une pile. Quand le caractère courant est un délimiteur ouvrant, on l'ajoute à la pile. Quand c'est un délimiteur fermant, on récupère ce qui est sur le sommet de la pile. Si c'est le délimiteur ouvrant qui correspond au fermant, on continue, sinon, il y a une erreur de parenthésage. On vous laisse chercher quel doit être l'état de la pile à la fin du traitement ...

   Utilisez la pile définie dans java dont la description est ici :

   https://docs.oracle.com/javase/10/docs/api/java/util/Stack.html (voir explications dans le support de cours).

4. Votre propre pile maison. A vous maintenant d'implémenter une classe "Pile<K>". Les éléments sont stockés dans une ArrayList java, vous implémentez les opérations de base : pop, push, empty, peek. Coupez/collez votre méthode "verification" pour en faire une méthode "verificationMaison" qui utilise votre propre pile et vérifiez que cela fonctionne toujours bien.

Listes génériques

Dans un premier temps, lisez et comprenez le support de cours qui vous est fourni puis téléchargez les deux classes "ListeGenerique" et "ElementListe" que vous devez compléter. Testez vos classes avec moodle.

1. **Méthodes élémentaires.** Complétez la méthode "boolean existe(E elt)" qui renvoie *true* si *elt* est dans la liste et $false$ sinon et "int rang(E elt)" qui renvoie l'indice de *elt* dans la liste, en numérotant à partir de 0, et renvoie -1 si *elt* n'est pas dans la liste. Ces deux méthodes suivent exactement le schéma de la méthode "int size()" qui vous est fournie: la méthode de la classe "ListeGenerique" traite le cas de la liste vide (i.e. $premier == null$) ou fait appel à la méthode correspondante de la classe "ElementListe". La méthode de la classe "ElementListe", traite l'élément *element* et continue à traiter les éléments à partir de *suivant* si $suivant != null$.

2. **Suppression d'un élément.** Supprimer un élément est un peu plus délicat. En effet, il faut se déplacer jusqu'à trouver l'élément, mais il faut garder le lien sur l'élément précédent, pour ne pas rompre la chaîne, comme expliqué dans le cours.

ArrayList (pour les téméraires ...)

On veut maintenant réaliser notre propre version des *ArrayList* java. Une *ArrayList* java combine les avantages d'un tableau et d'une liste : l'accès direct aux éléments et l'allocation dynamique. En effet, une *ArrayList* java est en fait une liste chaînée de tableaux. L'élément de base est un tableau, à chaque fois que la liste se remplit et qu'il n'y a plus assez de places dans les tableaux qui sont actuellement chaînés, alors un nouveau tableau est alloué et chaîné en fin de liste.

On vous propose d'écrire les *ArrayList* de façon similaire aux listes génériques que vous venez de créer en utilisant deux classes "ArrayListeGenerique<E>" et "ElementArrayListe<E>".

La classe "ArrayListeGenerique" contient un lien vers le début de la liste des tableaux. Elle contient aussi un lien vers le dernier tableau de cette liste. En effet, quand on ajoute un élément, on l'ajoute dans le dernier tableau s'il y a encore de la place, sinon, on créé un nouveau tableau que l'on chaîne avec le dernier et qui devient lui-même le dernier tableau. Il y a aussi l'attribut $size'$ qui contient le nombre d'éléments (mis à jour à chaque ajout/suppression) et l'attribut $initialCapacity$ qui donne la taille des tableaux.

La classe "ElementArrayListe" contient un tableau (de taille $initialCapacity$) et un lien vers le prochain tableau (si ce n'est pas le dernier). Il y a aussi l'indice courant de la première place disponible dans le tableau (cet indice est $initialCapacity$ pour tous les tableaux sauf le dernier).