Devoir maison n^{o}1 pour le 25 mars 2016 : Implémentation des listes à raccourcis (les *SkipList*)

L2 - Structures de données, 2015-2016

NOTES IMPORTANTES :

Modalités de remise du projet : Vous devez déposer sous Moodle une archive au format tar.gz (et uniquement à ce format-là) nommée : prénom.nom-SD1.tar.gz (remplacez prénom et nom par VOTRE prénom et VOTRE nom). Merci de ne pas mettre d'espaces ou d'accents dans le nom de votre archive. Cette archive, qui se décompressera dans un répertoire nommé prénom.nom-SD1 (sans blanc ni accent) devra contenir UNIQUEMENT :

- les fichiers sources, respectant la norme C99, de votre projet,
- un fichier *Makefile* permettant de compiler et de tester le programme réalisé,
- un document au format PDF contenant les explications demandées dans le sujet et les informations que vous souhaitez communiquer sur votre travail (Spécifications, description des algorithmes, références bibliographiques, ...)

En cas de non respect des consignes de dépôt une pénalité de 2 points sera appliquée sur la note globale. Tout retard par rapport à la date limite entraînera une pénalité de 1 point par heure révolue.

Lutte contre la copie et le plagiat : Un outil de détection de copie et de plagiat sera exécuté sur le code remis. Toute copie ou plagiat détecté sera sévèrement sanctionné. La note 0 sera attribuée au devoir et les étudiants concernés seront convoqués par le directeur des études et le responsable de l'UE.

Modalités de correction :

- Les codes remis seront testés automatiquement sur des machines tournant sous Linux avec la même installation que dans les salles de TP. Le script automatique exécutera en aveugle la commande `make clean && make tests`. Si la compilation échoue, la note 0 sera donnée et le code ne sera pas lu. Votre programme doit donc impérativement compiler et s'exécuter sur les machines de TP.
- 4 séries de 4 tests seront passés automatiquement, correspondant aux tests présents dans l'archive décrite en section 1. Chaque série de tests conditionne l'obtention de la note maximale dans chacune des 4 questions de la section 3. Si, dans une série, tous les tests passent, la note potentielle obtenue est la note maximum. Si tous les tests ne passent pas, la note maximale obtenue sera proportionnelle au nombre de tests passés (75% de la note pour 3 tests sur 4, 50% pour 2 tests sur 4, etc...)
- 3 tests supplémentaires seront ensuite exécutés. Ces tests ne vous sont pas fournis et porteront sur la construction, la recherche, et la suppression dans une liste contenant un très grand nombre de valeurs (de l'ordre d'une centaine de millions). Pour chaque test qui échoue, une pénalité de 1 point sera portée sur la note globale.
- Dans la configuration correspondant à la série de tests numéro 3, et pour chaque opération (construction, recherche, itérateur et suppression, l'analyseur de fuite de mémoire `valgrind` sera exécuté sur votre programme. Si le bilan mémoire de votre programme n'est pas nul, une pénalité de 1 point par opération générant des fuites de mémoire sera appliquée à la note globale. Lorsque vous exécutez `valgrind` sur votre programme, assurez-vous qu'apparaissent en fin de rapport les lignes :

```
    in use at exit: 0 bytes in 0 blocks
All heap blocks were freed -- no leaks are possible
```

- Enfin, la qualité du code, sa lisibilité, son efficacité (en temps et en mémoire) ainsi que sa réutilisabilité seront prises en compte dans l'attribution de la note.

L'objectif de ce devoir est de programmer et de comparer, en termes de complexité et de performance les opérations de dictionnaire (insérer, rechercher, supprimer) sur une structure de données linéaire de type liste doublement chaînée. A partir de l'archive source qui vous est fournie [1], le but de ce devoir est d'implémenter la structure linéaire probabiliste *SkipList* correspondant à la description fonctionnelle donnée dans la section 2.

1. Elle est décrite en section 1 et contient les spécifications du type abstrait de données à implémenter, quelques outils supplémentaires, et des jeux de tests pour évaluer le comportement du code développé.

1 Description de l'archive logicielle fournie

L'archive qui vous est fournie pour servir de base à votre travail est constituée des fichiers suivants :

1. Fichers sources
   - `rng.h` et `rng.c` : Définition du type *RNG* et des fonctions de manipulation permettant de générer une sequence de nombres aléatoires suivant une loi de probabilité imposée et permettant ainsi de construire la structure de données.
   - `skiplist.h` : Spécification, selon le format vu en cours et décrit à l'aide du langage *doxygen* du type abstrait à implémenter. Si le programme *doxygen* est installé sur votre machine, vous pouvez générer une version plus lisible de la spécification en tapant la commande *make doc* et en ouvrant dans votre navigateur le fichier `doc/html/index.html`.
   - `skiplisttest.c` : Programme principal, se limitant initialement à l'analyse de la ligne de commande selon les spécifications suivantes :
     usage : *./skiplisttest -id num*
     where *id* is one of the four letters :
     c : construct and print the *SkipList* with data read from file `test_files/construct_`*num*`.txt`.
     s : construct the *SkipList* with data read from file `test_files/construct_`*num*`.txt` and search elements from file `test_files/search_`*num*`.txt`. Print statistics about the searches.
     i : construct the *SkipList* with data read from file `test_files/construct_`*num*`.txt` and search, using an iterator, elements read from file `test_files/search_`*num*`.txt`. Print statistics about the searches.
     r : construct the *SkipList* with data read from file `test_files/construct_`*num*`.txt`, remove values read from file `test_files/remove_`*num*`.txt` and print the list in reverse order.
     where *num* is the file number for the input.
   - `Makefile` : makefile (pouvant éventuellement être modifié mais servant de référence pour la correction) permettant de compiler votre production.
2. Fichiers de test
   - `test_files` : répertoire contenant les jeux de données et les résultats de référence pour effectuer les tests
3. Script de test
   - `test_script.sh` : script shell de test de votre programme. Lorsque vous le lancez, ce script exécute les différentes étapes demandées et fournit un affichage indiquant les tests ayant réussi. Si votre programme est correct, vous obtiendrez l'affichage suivant :
     nom du test (numéro) **[OK]**
     Si un test échoue, vous aurez le symbole **[KO]** à la place du **[OK]** correspondant.

Pour chaque étape du sujet, vous devrez écrire dans votre programme principal la portion correspondant à la ligne de commande associée à l'opération à réaliser.

2 Description du principe de fonctionnement des *SkipLists*

Dans la description fonctionelle des listes à raccourcis que nous fournissons ici, les exemples sont donnés pour une liste simplement chaînée. Il vous est demandé d'implémenter les listes à raccourci sur des l**istes doublement chaînées**.

Les listes à raccourcis, nommées *SkipLists*, sont une alternative aux arbres de recherche équilibrés que nous verrons dans la suite du cours. Inventées par William Pugh en 1990 [2], elles reposent sur une structure de données linéaire construite de manière probabiliste. Les opérations de dictionnaire définies sur les listes à raccourcis sont simples à programmer, élégantes, et l'aspect probabiliste permet de démontrer une complexité moyenne en $O(log(N))$ au lieu des $O(N)$ inhérents aux structures linéaires. Toutefois, l'aspect probabiliste de la structure de données ne permet pas d'assurer la complexité en pire cas, comme on pourra le faire sur les arbres équilibrés, mais donne, en pratique, une structure extrêmement efficace (la probabilité de construire une structure non efficace étant très faible). Nous ne ferons pas de démonstration ici, et il ne vous est pas demandé d'en effectuer une, les étudiants curieux pourront se rapporter à leur cours de complexité du semestre précédent et à l'analyse fournie dans l'article de William Pugh, accessible sur Moodle.

2.1 Structure d'une *SkipList*

Les listes à raccourcis sont décrites par une succession de nœuds inter-connectés. A la manière des nœuds sur les listes (doublement) chaînées, chaque nœud est relié à son successeur dans la structure de données (et à son

2. William Pugh, *Skip lists : a probabilistic alternative to balanced trees* in Communications of the ACM, June 1990, 33(6) 668-676

prédécesseur si la liste est doublement chaînée). Par souci de clarté, nous considérons, dans notre explication, des listes simplement chaînées.

Chaque nœud possède une information associant une clé à une valeur. Sans perte de généralité, nous considérons que les nœuds possèdent une seule information représentant à la fois la clé et la valeur : un entier. Les nœuds sont ordonnés dans la liste par ordre croissant de leur clé. **La structure de données est donc une structure ordonnée**. Un nœud possède aussi une ou plusieurs références (pointeurs) vers des nœuds situés après lui dans la liste. Le nombre de pointeurs est déterminé aléatoirement, ce qui donne l'aspect probabiliste à la structure de données, et définit ce que nous appelons le *niveau* du nœud.

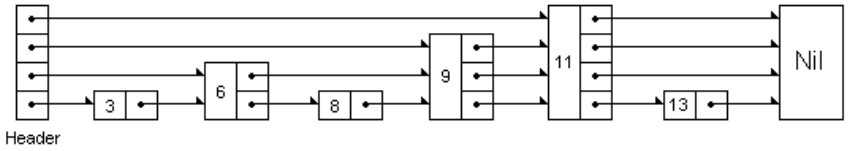

FIGURE 1 – Exemple de liste à raccourcis possédant 6 nœuds.

Un nœud de niveau $1 \leq k \leq nb\_level$ possède k pointeurs vers les nœuds situés après dans la liste. Ainsi, chaque nœud possède au moins un pointeur (le pointeur de niveau 1) désignant l'élément immédiatement suivant dans la liste. Les pointeurs numéro $p > 1$ désignent des nœuds de niveau $l \geq p$ se trouvant une ou plusieurs places plus loin dans la liste. Ces pointeurs sont donc des raccourcis vers les nœuds suivants, d'où le nom donné à ces listes.

Par exemple, dans la figure 1, avec $nb\_level = 4$, le niveau du nœud de clé 3 est 1 et celui du nœud de clé 6 est 2. Dans le nœud de clé 6, le pointeur numéro 1 permet d'accéder au nœud de clé 8 (le suivant immédiat du nœud de clé 6 dans la liste) alors que le pointeur numéro 2 permet d'accéder au nœud de clé 9 (situé plus loin dans la liste et de niveau $3 \geq 2$).

Dans une liste à raccourcis, deux nœuds sont systématiquement présents. Le premier, nommé *header*, est un nœud de niveau *nb_level* possédant donc *nb_level* pointeurs. Il marque le début de la liste. Le second, nommé *NIL*, marque la fin de la liste et est de niveau arbitraire. Il est à noter que ces deux nœuds peuvent être fusionnés en un seul que nous nommons *sentinelle* (*cf* cours de structures de données).

Une liste à raccourcis possède donc deux propriétés importantes : le niveau maximum d'un nœud (noté *nb_level*) et la loi de probabilité [3] permettant de déterminer le niveau que l'on va attribuer à un nœud particulier. Cette loi de probabilité est implémentée dans le générateur de nombres aléatoires dont l'interface vous est fournie dans le fichier `rng.h`.

La figure 1 montre un exemple de liste de niveau maximum 4 et possédant 6 nœuds avec clé.

2.2 Recherche d'un nœud dans une *SkipList*

La recherche d'un nœud, par sa clé, dans une liste à raccourcis débute par l'en-tête (ou la sentinelle), au niveau le plus haut de la liste et en suivant les pointeurs de ce niveau tant que les clés des nœuds accessibles sont inférieures à la clé recherchée. Si la clé d'un nœud accessible par un pointeur de niveau l est égale à la clé recherchée, l'algorithme s'arrête. Si la clé de ce noeud est supérieure à la clé recherchée, la progression continue sur le niveau $l - 1$ jusqu'à ce que l'on ait trouvé la clé recherchée ou que la clé d'un nœud atteint par un pointeur de niveau 1 soit strictement supérieure à la clé recherchée, indiquant que cette clé recherchée est absente de la collection.

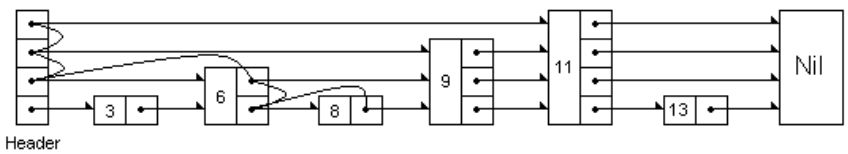

FIGURE 2 – Recherche du nœud 8.

La figure 2 montre le résultat de la recherche du nœud 8 dans la liste à raccourcis de la figure 1. Les liens courbes correspondent aux pointeurs qui ont été examinés pour trouver le nœud. Le nœud 3 n'a pas été visité.

3. A titre d'information, cette loi définit la probabilité $P(l)$ pour qu'un nœud soit de niveau l de la manière suivante.

$$1 \leq l < nb\_level \rightarrow P(l) = \frac{1}{2^l}$$
$$P(nb\_level) = \frac{1}{2^{nb\_level-1}}$$

2.3 Insertion d'un nœud dans une *SkipList*

L'insertion d'un nœud dans une liste à raccourcis se fait de façon très similaire à l'insertion d'un nœud dans une liste chaînée. Toutefois, dans une liste à raccourcis, comme dans toute structure de recherche ordonnée, il ne peut y avoir de collision de clé. Une clé apparaîtra donc une seule fois dans la collection. L'algorithme d'insertion commence par rechercher le nœud avant lequel il faut insérer le nouveau nœud (le premier nœud à avoir une clé supérieure à la clé du nœud à insérer).

Le niveau l du nouveau nœud est déterminé selon la loi de probabilité associée à la liste à raccourcis et les l pointeurs devant désigner ce nouveau nœud (et n'appartenant pas forcément à un même nœud de la liste) sont mis à jour. Il en est de même pour les l pointeurs du nouveau nœud. Si la liste est doublement chaînée, il y a un moyen simple, que vous devrez mettre en œuvre, pour connaître l'ensemble des pointeurs devant être mis à jour.

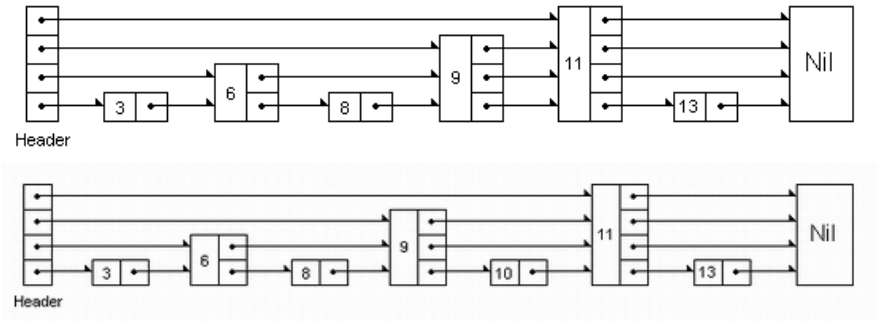

FIGURE 3 – Liste à raccourcis avant et après l'insertion du nœud 10.

La figure 3 montre le résultat de l'insertion du nœud 10 de niveau 1 (déterminé aléatoirement) dans la liste à raccourcis de la figure 1.

2.4 Suppression d'un nœud dans une *SkipList*

La suppression d'un nœud dans une liste à raccourcis se fait de manière symétrique à l'insertion. On commence par chercher le nœud à supprimer et on met à jour les différents pointeurs (à tous les niveaux) influencés par la suppression du nœud.

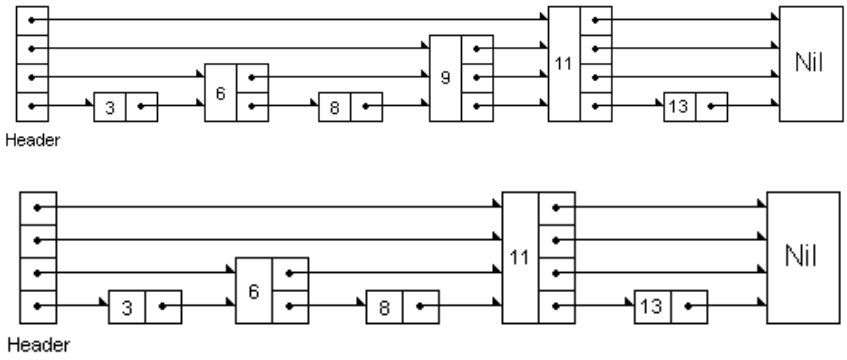

FIGURE 4 – Liste à raccourcis avant et après la suppression du nœud 9.

La figure 4 montre le résultat de la suppression du nœud 9 (de niveau 2) dans la liste à raccourcis de la figure 1.

3 Travail à réaliser

Les étapes ci-dessous sont à réaliser pour votre devoir. **ATTENTION**, nous vous demandons d'implémenter les listes à raccourcis pour des **listes doublement chaînées**. Les raccourcis doivent donc être à double sens. Afin de limiter les cas particuliers et de simplifier le développement, l'utilisation d'une sentinelle est très fortement recommandée.

La documentation de référence présente dans le code fourni doit être exploitée et suivie pour que vos développements respectent la spécification. Lors de la correction de vos devoirs, nous remplacerons vos fichiers d'implémentation par une implémentation de référence pour s'assurer du bon respect de l'interface. Reportez-vous aux modalités de correction pour savoir comment vont être évalués vos travaux.

La spécification du TAD et des opérations à implémenter est fournie dans le fichier `skiplist.h` de l'archive logicielle téléchargée sur Moodle (voir section 1). L'implémentation des opérateurs du TAD devra se faire dans le fichier `skiplist.c` que vous créerez. L'implementation des opération de test devra se faire dans le fichier `skiplisttest.c` contenant le programme principal.

3.1 Définition et construction d'une liste à raccourcis (8 points)

1. Définissez la structure de données

   struct s_SkipList

   définissant la représentation interne d'une liste doublement chaînée à raccourcis. Définissez l'ensemble des structures de données et opérateurs y afférant dont vous avez besoin pour représenter votre liste.

2. Programmer les fonctions

   SkipList skiplist_create (**int** nb_levels) et

   void skiplist_delete (**SkipList** d)

   permettant respectivement d'allouer-initialiser une liste doublement chaînée à raccourcis et de détruire-libérer la mémoire utilisée par une telle liste. Dans votre constructeur, vous prendrez soin d'initialiser le générateur de nombres aléatoires associé à la liste en lui fournissant une graine de 0 (*rng_initialize(0) ;*).

3. Programmer les fonctions

   unsigned int skiplist_size (**SkipList** d)

   int skiplist_ith (**SkipList** d, **unsigned int** i) et

   void skiplist_map(**SkipList** d, **ScanOperator** f, **void** *user_data).

   Vous prendrez soin d'avoir une implémentation en $O(1)$ pour la fonction

   unsigned int skiplist_size (**SkipList** d)

   et en $O(n)$ pour les autres fonctions.

4. Programmer la fonction

   SkipList skiplist_insert (**SkipList** d, **int** value)

   qui ajoute une valeur à la liste doublement chaînée à raccourcis. Le niveau du nouveau nœud sera déterminé en appelant la fonction rng_get_value() avec le générateur associé à la liste.

5. Dans le fichier `skiplisttest.c`, programmez la fonction

   void test_construction(**int** num)

   qui lit dans le fichier `test_files/construct_`*num*`.txt` (avec *num* la valeur du paramètre num) les données de la liste à raccourcis à construire.

   Les fichiers `test_files/construct_`*num*`.txt` fournis ont un format simple composé d'une succession de n valeurs.

   — La première valeur est un entier non signé définissant le nombre de niveaux de la liste à construire.
   — La deuxième valeur est un entier non signé donnant le nombre de valeurs à insérer dans la liste.
   — les valeurs suivantes sont des entiers signés devant être ajoutés à la liste.

   Après avoir construit votre liste, vous afficherez, **dans cette fonction**, la liste dans l'ordre croissant selon le format correspondant à l'exemple ci-dessous dans lequel `13` est le nombre de valeurs de la liste. Les valeurs doivent être affichées séparées par un espace. Attention, il ne doit pas y avoir un seul autre affichage sur la sortie standard pendant l'exécution de votre programme.

   ```
   Skiplist (13)
   0 1 2 3 4 5 6 7 8 9 11 12 18
   ```

   L'exemple ci-dessus correspond au résultat attendu pour l'exécution de la commande *./skiplisttest -c 1*

6. Vérifiez la bonne implémentation de cette partie en lançant la commande *make tests*. Si votre programme est bon, vous aurez le symbole [OK] en face des tests de construction.

3.2 Recherche d'une valeur dans une liste à raccourcis (4 points)

1. Programmez la fonction

   bool skiplist_search (**SkipList** d, **int** value, **unsigned int** *nb_operations)

   recherchant la valeur value dans la liste. Cette fonction doit aussi renvoyer, dans le paramètre nb_operations, le nombre de nœuds qui ont été visités pendant la recherche.

2. Dans le fichier `skiplisttest.c`, programmez la fonction

   void test_search(**int** num)

   qui, comme dans la question 3.1.5, lit dans le fichier `test_files/construct_`*num*`.txt` les données de la liste à raccourcis à construire. Une fois la liste construite, la fonction

   void test_search(**int** num)

   effectuera n recherches dans la liste selon les données à lire dans le fichier `test_files/search_`*num*`.txt`. Les fichiers `test_files/search_`*num*`.txt` fournis ont un format simple composé d'une succession de n valeurs.
   — La première valeur de ces fichiers est un entier non signé indiquant le nombre de valeurs à chercher.
   — les valeurs suivantes sont des entiers signés devant être recherchés dans la liste.
   Cette fonction doit afficher, en respectant l'ordre des valeurs lues dans le fichier, le résultat de la recherche et des statistiques globales sur le nombre d'opérations effectuées. Ainsi, pour chaque valeur `v` recherchée dans la liste, une ligne correspondant au modèle suivant doit être affichée sur la sortie standard.
   Si la valeur `v` est trouvée dans la liste :
   `v -> true`
   Si la valeur `v` n'est pas trouvée dans la liste :
   `v -> false`
   Une fois les résultats de recherche affichés, vous afficherez les statistiques sur le nombre de nœuds visités pendant les recherches avec le format suivant. Les statistiques incluent la taille de la liste, le nombre de valeurs cherchées, le nombre de valeurs trouvées, le nombre de valeurs non trouvées, le nombre minimum de nœuds visités pendant une recherche, le nombre maximum de nœuds visités pendant la recherche et le nombre moyen (un entier) de nœuds visités pendant la recherche. Attention, il ne doit pas y avoir un seul autre affichage sur la sortie standard pendant l'exécution de votre programme.

```
Statistics :
    Size of the list : 13
Search 20 values :
    Found 13
    Not found 7
    Min number of operations : 1
    Max number of operations : 8
    Mean number of operations : 5
```

   L'exemple ci-dessus correspond au résultat attendu pour l'exécution de la commande *./skiplisttest -s 1*

3. Vérifiez la bonne implémentation de cette partie en lançant la commande *make tests*. Si votre programme est bon, vous aurez le symbole [OK] en face des tests de recherche.

3.3 Itérateur et recherche linéaire d'une valeur dans une liste (4 points)

1. Implémenter le TAD

   SkipListIterator

   tel que spécifié dans le fichier `skiplist.h` de façon à pouvoir parcourir, du côté client, toutes les valeurs d'une liste à raccourcis par le code suivant :

```
void iterate_on_skiplist(SkipList d) {
    SkipListIterator e = skiplist_iterator_create(d, DIRECTION_ITERATOR);
    for (    e = skiplist_iterator_begin(e);
             !skiplist_iterator_end(e);
             e = skiplist_iterator_next(e)
        ) {
        Do_somehing_with(skiplist_iterator_value(e));
    }
}
```

   où DIRECTION prend une des deux directions FORWARD, BACKWARD de parcours.

2. Dans le fichier `skiplisttest.c`, programmez la fonction

   void test_search_iterator (**int** num)

   qui, comme dans la question 3.2.2, utilise le fichier `test_files/construct_`*num*`.txt` pour construire une liste à raccourcis et cherche dans la liste les valeurs contenues dans le fichier `test_files/search_`*num*`.txt`. La recherche s'effectuera ici du côté du client en utilisant un itérateur. Vous afficherez de la même manière que dans la question 3.2.2 les statistiques sur les recherches.

3. En vous fondant sur les statistiques issues des questions 3.2.2 et 3.3.2, expliquez dans votre rapport de devoir les raisons de la différence en temps de calcul constatée, particulièrement pour le test numéro 4.
4. Vérifiez la bonne implémentation de cette partie en lançant la commande *make tests*. Si votre programme est bon, vous aurez le symbole **[OK]** en face des tests d'itérateur.

3.4 Suppression d'une valeur dans une liste (4 points)

1. Programmez la fonction

   bool skiplist_remove(**SkipList** d, **int** value)

   supprimant la valeur value de la liste.
2. Dans le fichier `skiplisttest.c`, programmez la fonction

   void test_remove(**int** num)

   qui, comme dans la question 3.1.5, lit dans le fichier `test_files/construct_`*num*`.txt` les données de la liste à raccourcis à construire. Une fois la liste construite, la fonction

   void test_remove(**int** num)

   effectuera n suppressions dans la liste selon les données à lire dans le fichier `test_files/remove_`*num*`.txt`. Les fichiers `test_files/remove_`*num*`.txt` fournis ont un format simple composé d'une succession de n valeurs.
   — La première valeur de ces fichiers est un entier non signé indiquant le nombre de valeurs à supprimer.
   — Les valeurs suivantes sont des entiers signés devant être supprimés de la liste.
   Comme dans la question 3.1.5, cette fonction affichera la liste sur la sortie standard (avec le même format) mais dans l'ordre décroissant de ses valeurs.
3. Vérifiez la bonne implémentation de cette partie en lançant la commande *make tests*. Si votre programme est bon, vous aurez le symbole **[OK]** en face des tests de suppression.