# SINF1121 – Algorithmique et structures de données

# Mission 3

*Arbres binaires de recherche et arbres de recherche équilibrés*

## 1 Contexte général de la mission

Vous travaillez sur un nouveau logiciel de streaming musical permettant aux utilisateurs de parcourir un catalogue, sélectionner un morceau ou une playlist et l'écouter instantanément. Afin d'être en mesure de rivaliser avec la concurrence, vous décidez de vous focaliser sur les fonctionalités de recherche dans le catalogue.

Votre tâche dans cette mission est d'implémenter une structure de données efficace pour stocker toutes les paires (artiste, liste de chansons) afin de pouvoir, par la suite, récupérer l'ensemble de ces paires (ou un sous-ensemble de celles-ci) **ordonnées par nom d'artiste**. La structure de données idéale pour accomplir cette tâche est bien entendue la **map ordonnée**, et on vous demandera de l'implémenter en utilisant un arbre binaire de recherche. Ces deux notions constituent le fil conducteur de cette mission. Vous devrez également, dans un deuxième temps, être capable de trier les chansons correspondant à un certain artiste. Pour ce faire, vous utiliserez l'algorithme bien connu `QuickSort`.

## 2 Objectifs poursuivis

À l'issue de cette mission chaque étudiant sera capable :

— de ***décrire*** avec exactitude et précision les concepts d'***arbres binaires de recherche*** et de ***table de symboles ordonnée***,
— de ***mettre en oeuvre*** des algorithmes basés sur les arbres de recherche,
— d'***évaluer*** et ***mettre en oeuvre*** des représentations classiques d'arbres de recherche,
— d'appliquer à bon escient des principes de programmation orientée-objet tels que modularité, abstraction, généricité afin de pouvoir ***composer*** et ***réutiliser*** au mieux des ***classes*** existantes.

## 3 Prérequis

Les conditions à remplir pour pouvoir aborder cette mission sont :

— avoir rempli avec succès les missions précédentes,
— se débrouiller en anglais technique et scientifique (lecture).

# 4 Ressources

— Livre : *Algorithms-4* : chapitre 3.1, 3.2, 3.3.
— Document : *Spécifications en Java, Préconditions et postconditions : pourquoi ? comment ?*, P. Dupont.
Sur le site Moodle, suivre `Documents et liens/pdf/specif.pdf`
— Document : *Complexité calculatoire*, P. Dupont
Sur le site Moodle, suivre `Documents et liens/pdf/complex.pdf`

# 5 Calendrier

— lundi 19 octobre : deémarrage de la mission
— dimanche 25 octobre, avant 18h00 : envoi re ?ponses questions (groupe) + tests Inginious (individuel)
— lundi 26 octobre : séance interme ?diaire
— vendredi 30 octobre, avant 18h00 : remise des produits (groupe)
— lundi 2 novembre : seéance de bilan

# 6 Questions

1. Laquelle des deux implémentations `SequentialSearchST` ou `BinarySearchST` utiliseriez vous pour une application qui réalise $10^3$ `put()` et $10^6$ `get()` opération dans un ordre aléatoire ? Justifiez.
2. Implémentez la méthode `floor()` de `BinarySearchST`.
3. *Interpolation Search : Exercice 3.1.14*. En supposant que les clefs soient des doubles ou des entiers. Écrivez une version de la recherche binaire qui supposant une répartition uniforme des clefs va d'abord chercher au début d'un dictionnaire un mot qui commence par une lettre proche du début d'alphabet. Plus exactement si la clef recherchée est $k_x$, et que la plus petite clef est $k_{lo}$ et la plus grande est $k_{hi}$, cherchez d'abord à l'indice $\lfloor (k_x - k_{lo})/(k_{hi} - k_{lo}) \rfloor$ et pas au milieu du tableau d'abord. Implémentez `InterpolationSearchST` et comparez celle-ci sur `FrequencyCounter`.
4. *Caching : Exercice 3.1.25*. Il est très fréquent de tester d'abord la présence d'une clef avant d'ajouter ou modifier l'entrée correspondante. Cela engendre successivement plusieurs recherche consécutive de la même clef. L'idée du *caching* est de mémoriser en interne la dernière clef accédée et de l'utiliser de manière opportuniste si celle-ci est toujours valide. Modifiez `BinarySearchST` pour y intégrer cette idée.
5. *Exercice 3.2.31* Ecrivez une méthode `isBST()` qui prend un `Node` comme argument et qui retourne `true` si l'argument est est la racine d'un BST, `false` sinon (il faut donc vérifier que les propriétés d'un BST sont satisfaites).
6. *Exercice 3.2.4* Supposons qu'un certain arbre de recherche possède des clefs entre 1 et 10 et que nous cherchions la clef 5. Quelle(s) sequence(s) ne peut pas correspondre à la séquence des clefs examinées ?

(a) 10,9,8,7,6,5

(b) 4,10,8,6,5

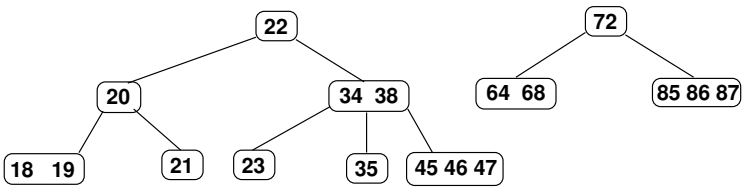
(c) 1,10,2,9,3,8,4,7,6,5

(d) 2,7,3,8,4,5

(e) 1,2,10,4,8,5

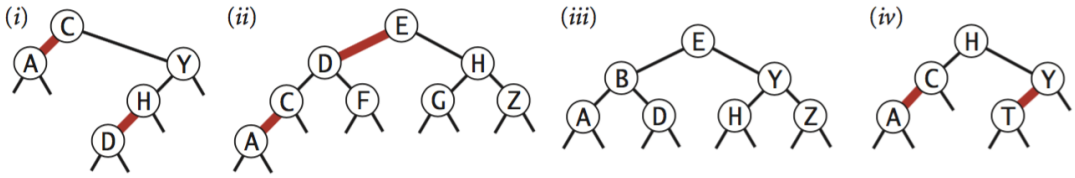
7. *Exercice 3.3.33* Ecrivez une méthode `is23()` dans `RedBlackBST` qui vérifient respectivement qu'aucun noeud n'est connecté à deux liens rouges et qu'il n'y a pas de lien rouge vers la droite. Ecrivez aussi une méthode `isBalanced()` qui vérifie que tout chemin depuis la racine vers vers un lien null a le même nombre de liens noirs. Finalement combinez `isBST(),is23()` et `isBalanced()` pour implémenter `isRedBlackBST()`.
8. Comment faire pour énumérer en ordre croissant toutes les clés mémorisées dans un arbre binaire de recherche ? Quelle est la complexité temporelle de cette opération ? Justifiez votre réponse.
9. Partant d'un arbre binaire de recherche initialement vide, comment se présente l'arbre après y avoir inséré les clés 12, 5, 10, 3, 13, 14, 15, 17, 18, 15 ? Pour les mêmes données comment se présenterait l'arbre finalement obtenu s'il s'agissait d'un 2-3 arbre ? Pour les mêmes données comment se présenterait l'arbre finalement obtenu Cet exemple illustre-t-il les avantages ou inconvénients de ces différentes structures de données ? Pourquoi ?
10. ( Question posée à l'examen de janvier 2009. ) Nous considérons $T$ et $U$ deux 2-3 arbres mémorisant respectivement $n$ et $m$ clés tels que toutes les clés dans $T$ sont strictement inférieures à tous les clés dans $U$. Proposez un algorithme pour fusionner $T$ et $U$ en un seul 2-3 arbre. Ce nouvel 2-3 arbre doit donc contenir toutes les clés de $T$ et toutes les clés de $U$. La complexité temporelle de votre algorithme ***doit*** être en $\mathcal{O}(\log n + \log m)$.

    - Appliquez votre algorithme sur les deux arbres illustrés ci-dessous. Illustrez graphiquement la construction de l'arbre résultat pour chaque étape principale de votre algorithme. Note : votre algorithme doit pouvoir s'appliquer à n'importe quelle paire de 2-3 arbres satisfaisant les conditions reprises dans l'énoncé de la question. On vous demande simplement d'illustrer le fonctionnement de votre algorithme général sur un cas particulier.




    - Justifiez **pourquoi** la complexité temporelle de votre algorithme est bien en $\mathcal{O}(\log n + \log m)$ , où $n$ et $m$ correspondent aux nombres de clés respectivement dans $T$ et $U$.

11. Lequel de ces arbres est un red-black tree ? Pour chacun d'eux dessiner la correspondance vers le 2-3 tree (décrite p432) ?




12. *Spécifique à la mission :* Implémentez la classe *RBEntry* qui implémente `java.util.Map.Entry`. Écrivez les fonctions `equals(Object o)` et `hashCode()`. Insérez des éléments `RBEntry` dans un set et observez leur ordre. Que se passe-t-il si vous changez les méthodes equals et hashCode ? Pourquoi ? (voir la discussion Q&A p387, voir API Java dans `Map.Entry`).

Les réponses aux questions et les parties d'implémentation individuelles doivent être soumises sur INGInious ***avant*** la séance ***intermédiaire***[a]. Cela suppose une étude individuelle et une mise en commun en groupe (sans tuteur) préalablement à cette séance. Un document (au format PDF) reprenant les réponses aux questions devra être soumis sur INGInious ***au plus tard*** pour le dimanche 25 octobre à ***18h00***. Les réponses seront discutées en groupe avec le tuteur durant la séance intermédiaire. Ces réponses ne doivent pas explicitement faire partie des produits remis en fin de mission. Néanmoins, si certains éléments de réponse sont essentiels à la justification des choix d'implémentation et à l'analyse des résultats du programme, ils seront brièvement rappelés dans le *rapport de programme*.

*a.* à l'exception, le cas échéant, de question(s) spécifiquement liée(s) au problème traité.

# 7 Problème

Votre mission est donc de réaliser un programme permettant de regrouper des morceaux de musique par artiste, et de trier ces artistes (ainsi que les morceaux correspondant à un certain artiste) par ordre alphabétique, afin de pouvoir récupérer le catalogue complet (ou, le cas échéant, une partie de ce catalogue) de manière efficace.

Concrètement, votre devez implémenter l'interface `OrderedMap` disponible sur INGInious en utilisant un arbre binaire de recherche (vous êtes libres d'utiliser l'arbre de votre choix). Votre classe, `SearchTree`, disposera d'un constructeur sans argument (pour construire un arbre initialement vide), mais également d'un constructeur prenant comme argument un String correspondant au chemin vers un fichier contenant des paires (artiste, chanson). La structure du fichier [1] est la suivante : le nom d'un artiste, suivi d'une tabulation, suivi du nom d'une chanson de cet artiste, suivi d'un retour à la ligne. Parmi les méthodes de l'interface `OrderedMap` que vous devrez implémenter, voici les plus intéressantes :

1. par exemple, *songs.txt*, disponible sur Moodle et tiré du *Top 1,000 Classic Rock Songs* sur `http://www.rocknrollamerica.net/Top1000.html`

— `get(String key)` : retourne un `Set` de `Strings` représentant l'ensemble des morceaux de musique correspondant à l'artiste `key`. Complexité requise : $O(\log n)$ où n est le nombre total d'artistes dans l'arbre. Cela sous-entend que votre arbre devra être équilibré !
— `getOrdered(String key)` : même méthode que la précédente, mais retourne cette fois les morceaux de l'artiste `key` **ordonnés** par ordre alphabétique en utilisant l'algorithme `QuickSort`. Les morceaux de musique correspondant à un certain artiste ne doivent donc **pas** être ordonnés dans votre structure de données, mais ils seront triés dynamiquement sur demande.
— `entriesBetween(String lowest, String highest)` : retourne une liste contenant l'ensemble des entrées dont la clé se trouve entre `lowest` et `highest`, ordonnée par nom d'artiste. La complexité est ici un point très important : s'il y a $n$ artistes au total et $x$ artistes entre `lowest` et `highest`, vous devrez parvenir à une complexité $O(x + \log n)$, ce qui implique une méthode un peu plus intelligente que simplement utiliser `ceilingEntry(lowest)` suivi d'une succession de `higherEntry` sur la clé du résultat, ce qui donnerait une complexité de $O(x \log n)$.

Vous veillerez à justifier les complexités de ces 3 méthodes (ainsi que des autres méthodes que vous jugeriez intéressantes) dans votre *rapport de programme*, en expliquant comment vous êtes parvenus à cette complexité et si vous pensez qu'il est possible de faire mieux.

## Indications pour la bonne réussite de la mission

- Chaque étudiant préparera la réponse à la (aux) question(s) dont il est responsable. Le rapport mentionnera pour chaque question ***qui*** en a été responsable.

- Les réponses aux questions posées ne se trouvent pas toutes dans le chapitre du livre de référence concerné par cette mission. N'oubliez pas de consulter l'index de cet ouvrage ou toute autre ***ressource complémentaire*** mentionnée dans ce document ou sur le site WEB du cours. Veillez toujours à ***citer*** précisément ***vos sources*** lorsque vous répondez aux questions.

- Veillez à vous concentrer d'abord sur les questions avant de passer au problème à résoudre. Veillez à y répondre de manière précise et concise. Quand vous attaquez le problème, gardez à l'esprit les concepts abordés dans les questions.

- Pendant le travail d'analyse du problème et de conception d'une solution, l'écriture d'un ***diagramme de classes*** est très utile.

- Chaque étudiant d'un groupe doit être en charge d'une partie du travail de programmation. Vous indiquerez ***toujours*** en commentaires d'une classe, le ou les auteur(s) de la classe.

- Chaque étudiant est responsable de la bonne organisation de la mission et de l'équilibre entre son travail personnel et sa participation active au groupe.

# 8 Remise des produits

Les produits de la mission (code, tests et rapport) sont à soumettre sur INGInious (https://inginious.info.ucl.ac.be/course/LSINF1121-2015) pour le vendredi 30 octobre, à ***18h00*** **dernier délai**. Passé ce délai, il sera impossible d'effectuer une nouvelle soumission.