IUT Paris Descartes

DUT 2 Informatique

Algorithmique Avancée – M3103

Cahier de TD / TP

Responsable du module : Camille Kurtz (camille.kurtz@parisdescartes.fr)
Responsables de TD/TPs :

- Jacques Alès-Bianchetti (jacques-ales.bianchetti@parisdescartes.fr)
- Jean-François Brette (jean-francois.brette@parisdescartes.fr)
- Camille Kurtz (camille.kurtz@parisdescartes.fr)

2018 – 2019

Planning 2018–2019

Equipe pédagogique

Responsable du module et CMs : Camille Kurtz (`camille.kurtz@parisdescartes.fr`)
Responsables de TD/TPs :
— Jacques Alès-Bianchetti (`jacques-ales.bianchetti@parisdescartes.fr`)
— Jean-François Brette (`jean-francois.brette@parisdescartes.fr`)
— Camille Kurtz (`camille.kurtz@parisdescartes.fr`)

Plan du module

CM 01 Introduction à la complexité algorithmique

CM 02 Complexité des algorithmes de tri

CM 03 Construction de structures arborescentes

CM 04 et 05 Algorithmique des arbres

CM 06 Collections et tables de hachage, liens avec l'API Java

Évaluation

1. 1 Contrôle Continu (CC) (sous la forme d'un QCM)
2. 1 Projet sur machine "autonome" (à faire en dehors des heures de TD/TPs)
3. 1 DST en fin de bimestre

Planning

Semaine	1 (10/09)	2 (17/09)	3 (24/09)	4 (01/10)	5 (08/10)	6 (15/10)	7 (22/10)
CM	CM 01	CM 02	CM 03 [1]	CM 04	CM 05	CM 06	–
TD	TD 01	TD 02	TD 03	TD 04	TD 05	TD 06	Correction CC
TP	TP 01	TP 02 [2]	TP 03	TP 04	TP 05	TP 06	Éval. projets

1. **Contrôle Continu de 30 min pendant la séance de CM en amphi**
2. **Distribution du sujet de projet en début de séance de TP**

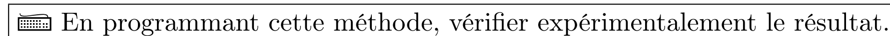
Sujet TD/TP 01 : Notions de base de complexité

Note : les questions comportant le symbole ⌨ seront traitées pendant les séances de TP sur machine.

Exercice 1 – Complexité en temps

Quelle est la complexité en temps (calculée ici en nombre d'appels à `print`) d'un appel de la méthode `loops` ?

```
public static void loops(int n){
  for (int i = 1; i <= n - 1; i++){
    for (int j = i + 1; j <= n; j++){
      for (int k = 1; k <= j; k++){
        System.out.print("x");
      }
    }
  }
}
```


⌨ En programmant cette méthode, vérifier expérimentalement le résultat.

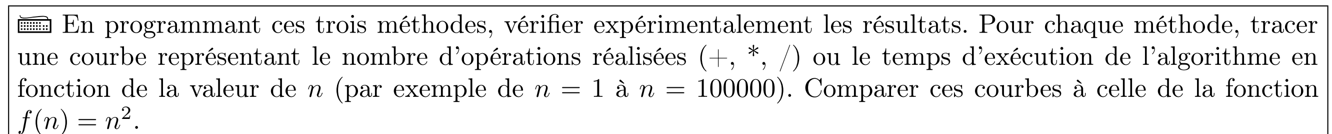
Exercice 2 – Complexité en temps et optimisation

Pour des valeurs de n allant de 1 à 5, réaliser la trace des appels des méthodes `sumSquare1`, `sumSquare2` et `sumSquare3`. Que calculent ces trois méthodes ? Quelle est leur complexité en temps (calculée ici en nombre d'opérations arithmétiques) ?

```
public static int sumSquare1(int n){
  int res = 0;
  for(int i = 1; i <= n; i++){
    for(int j = 1; j <= i; j++){
      res = res + i;
    }
  }
  return res;
}
```

```
public static int sumSquare2(int n){
  int res = 0;
  for(int i = 1; i <= n; i++){
    res = res + i * i;
  }
  return res;
}
```

```
public static int sumSquare3(int n){
  int res = 0;
  res = n * (n + 1) * (2 * n + 1) / 6;
  return res;
}
```


⌨ En programmant ces trois méthodes, vérifier expérimentalement les résultats. Pour chaque méthode, tracer une courbe représentant le nombre d'opérations réalisées (+, *, /) ou le temps d'exécution de l'algorithme en fonction de la valeur de n (par exemple de $n = 1$ à $n = 100000$). Comparer ces courbes à celle de la fonction $f(n) = n^2$.

Exercice 3 – Complexité en temps : meilleur cas, pire cas et cas moyen

Étant donné un tableau `tab` contenant des valeurs de type `boolean`, que calculent les méthodes `containsTrue1` et `containsTrue2` ? Quelle est leur complexité en temps (calculée ici en nombre d'affectations), en meilleur cas, pire cas et cas moyen ?

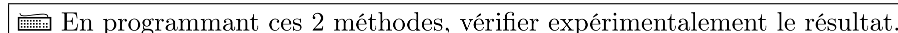
```
public static boolean containsTrue1(boolean tab[]){
  boolean res = false;
  for(int i = 0; i < tab.length; i++){
    res = res || tab[i];
  }
  return res;
}
```

```
public static boolean containsTrue2(boolean tab[]){
  boolean res = false;
  int i = 0;
  while((!res) && (i < tab.length)){
    res = (res || tab[i]);
    i++;
  }
  return res;
}
```

⌨ En programmant ces deux méthodes, vérifier expérimentalement les résultats.

Exercice 4 – Optimisation en temps vs. coût en espace

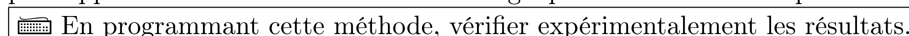
La factorielle d'un entier n positif, notée $n!$ est définie comme le produit de tous les entiers de 1 à n. Quelle est la complexité en temps d'une méthode calculant la factorielle ?
Le coefficient binomial C_n^k se définit, pour $0 \leq k \leq n$, comme $\frac{n!}{k!(n-k)!}$. Quelle est la complexité en temps d'une méthode calculant le coefficient binomial ?

 En programmant ces 2 méthodes, vérifier expérimentalement le résultat.

Pour une valeur N donnée, on souhaite effectuer un traitement requérant un grand nombre de fois le calcul de coefficients binomiaux. Afin d'optimiser la procédure, on décide de précalculer et stocker tous les coefficients binomiaux pour les valeurs $n \leq N$. Combien de coefficients binomiaux cela représente-t-il ? Comment les stocker efficacement (par rapport au coût en espace et au coût en temps de l'accès) ?
Proposer une méthode calculant la structure de données de stockage de ces coefficients, en vous appuyant sur la formule de calcul des coefficients binomiaux précédemment évoquée. Quel est le coût en temps de cette méthode ?
En vous appuyant sur le fait que $C_n^0 = C_n^n = 1$ pour tout n positif, et que $C_k^n = C_k^{n-1} + C_{k-1}^{n-1}$ lorque $0 < k < n$, proposer une nouvelle méthode calculant la structure de données de stockage de ces coefficients. Quel est le coût en temps de cette méthode ?

Exercice 5 – Corrélation entre coûts en espace et en temps

Considérons un tableau `tab` de taille N contenant des valeurs de type `short`. On sait alors que toutes les valeurs de `tab` sont entières et contenues entre deux bornes $a \leq b$. On cherche à trier `tab` de la manière la moins coûteuse possible. Pour ce faire, on crée un tableau `histo` qui contient $b-a+1$ cases, destinées à stocker chacune le nombre d'occurrences de l'une des valeurs entre a et b. Ce tableau est ensuite parcouru afin de générer une version triée de `tab`. Proposer un méthode s'appuyant sur cette stratégie. Quel est le coût en espace de cette méthode ? Quel est son coût en temps ? Quand peut-on affirmer que cette méthode présente une complexité linéaire en temps par rapport à la taille de `tab` ? Cette stratégie peut-elle fonctionner pour trier des tableaux de nombres flottants ?

 En programmant cette méthode, vérifier expérimentalement les résultats.

Exercice 6 – Optimisation en temps par prétraitement

Considérons un tableau `tab` de taille N contenant des valeurs de type `float`. Proposer une méthode qui vérifie la présence d'un élément donné `x` dans `tab`. Quelle est la complexité pire cas, meilleur cas et cas moyen de cette méthode ?
Considérons maintenant que `tab` a été trié (par exemple, par valeurs croissantes). Proposer une méthode dichotomique qui vérifie la présence d'un élément donné `x` dans `tab`. Quelle est la complexité pire cas de cette méthode ?
Sachant qu'une procédure de tri optimale présente une complexité en temps $\Theta(N \log N)$, à partir de combien de recherches d'éléments devient-il valable de trier préalablement `tab` ?

Pour aller plus loin...

Exercice 7 (subsidiaire)

Soit `n` une variable de type entier (`byte`, `short`, `int` ou `long`) stockée sur 2^k bits ($k = 3$, 6, 12 ou 24, pour les types précédents). Proposer une première méthode qui calcule le nombre de bits à 1 dans `n`, en temps linéaire par rapport à la taille de `n` (c'est-à-dire par rapport à son nombre de bits). Proposer une seconde méthode qui calcule le nombre de bits à 1 dans `n`, en temps logarithmique par rapport à la taille de `n`.

Exercice 8 (subsidiaire)

Soient f et g deux fonctions des entiers naturels dans les réels strictement positifs. Montrer que la relation $f = \mathcal{O}(g)$ est réflexive et transitive. Montrer que la relation $f = \Theta(g)$ est une relation d'équivalence. Montrer que $f = \mathcal{O}(g)$ implique $a.f = \mathcal{O}(g)$ et que $f = \Theta(g)$ implique $a.f = \Theta(g)$, où a est un réel strictement positif quelconque.

Exercice 9 (subsidiaire)

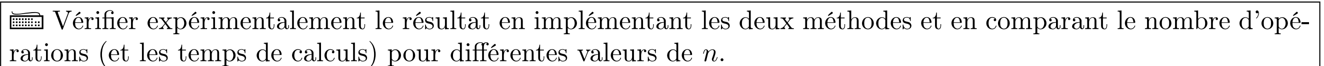
Soient f_1, f_2, g_1 et g_2 des fonctions des entiers naturels dans les réels positifs. Montrer que $f_1 + f_2 = \mathcal{O}(\max(g_1, g_2))$ et $f_1.f_2 = O(g_1.g_2)$ si $f_1 = \mathcal{O}(g_1)$ et $f_2 = \mathcal{O}(g_2)$. Montrer que $f_1 - f_2 = \mathcal{O}(f_1)$ si $f_1 - f_2 \geq 0$. Montrer que $f_1 + f_2 = \Theta(\max(g_1, g_2))$ et $f_1.f_2 = \Theta(g_1.g_2)$ si $f_1 = \Theta(g_1)$ et $f_2 = \Theta(g_2)$.

Sujet TD/TP 02 : Récursivité

Note : les questions comportant le symbole ⌨ seront traitées pendant les séances de TP sur machine.

Exercice 1 – Récursif vs. itératif

Définir deux méthodes `factorielleRec` et `factorielleIter` qui calculent la factorielle $n! = \prod_{k=1}^{n} k = 1 \times 2 \times \ldots \times k$ d'un nombre entier positif n de manière récursive et itérative, respectivement. Comparer ces deux méthodes (taille du code, lisibilité, ...). Calculer leur complexité en temps.


⌨ Vérifier expérimentalement le résultat en implémentant les deux méthodes et en comparant le nombre d'opérations (et les temps de calculs) pour différentes valeurs de n.

Exercice 2 – Récursivité multiple vs. récursivité simple (1/2)

Que calculent les deux méthodes `puissance1` et `puissance2` ?

```
public static int puissance1(int n){
  if (n == 0){
    return 1;
  }
  else{
    return 2 * puissance1(n - 1);
  }
}
```

```
public static int puissance2(int n){
  if (n == 0){
    return 1;
  }
  else{
    return puissance2(n - 1) + puissance2(n - 1);
  }
}
```

Pour $n = 5$, dessiner l'arbre des appels récursifs de ces deux méthodes. Calculer leur complexité en temps.

⌨ Vérifier expérimentalement le résultat.

Exercice 3 – Récursivité multiple vs. récursivité simple (2/2)

La suite de Fibonacci $(F_k)_{k \geq 0}$ se définit par $F_0 = 1$, $F_1 = 1$ et $F_{k+2} = F_{k+1} + F_k$ pour tout $k \geq 0$. Définir une première méthode récursive `fibonacciRec` qui applique de manière immédiate cette définition. Pour $k = 5$, dessiner l'arbre des appels récursifs de cette méthode. Calculer sa complexité en temps.

⌨ Vérifier expérimentalement le résultat.

Définir une seconde méthode récursive `fibonacciRec2` qui, pour un paramètre $k \geq 1$, retourne le k-ième, mais aussi le $(k-1)$-ième terme de la suite de Fibonacci. Pour $k = 5$, dessiner l'arbre des appels récursifs de cette méthode. Calculer sa complexité en temps.

⌨ Vérifier expérimentalement le résultat.

Exercice 4 – Récursivité et explosion combinatoire

Que calcule la méthode `mystere` ?

```
public static int mystere(int i){
  if (i < 2){
    return 1;
  }
  if (i == 2){
    return 2;
  }
  for(int j = mystere(i - 1); j > 1; j = mystere(j - 1)){
    if ((i % j) == 0){
      return mystere(i - 1);
    }
  }
  return i;
}
```

Quels sont les points critiquables dans cette méthode ? Proposer une méthode itérative qui réalise le même travail.

Exercice 5 – Les classiques de la récursivité (1/2)

Définir une méthode `ackermann` calculant la fonction d'Ackermann $A : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, définie par :

$$\begin{cases} A(0,n) = n+1 & \\ A(m,0) = A(m-1,1) & (m \neq 0) \\ A(m,n) = A(m-1, A(m,n-1)) & (m,n \neq 0) \end{cases}$$

Étudier le comportement de cette fonction pour des petites valeurs de m et n.

Exercice 6 – Les classiques de la récursivité (2/2)

Définir une méthode `syracuse` calculant le nombre de termes r de la suite de Syracuse définie par :

$$\begin{cases} U_0 = n \in \mathbb{N} \\ U_{i+1} = \begin{cases} \frac{U_i}{2} & \text{si } U_i \text{ est pair} \\ 3.U_i + 1 & \text{si } U_i \text{ est impair} \end{cases} \\ U_r = 1 \text{ (dernier terme)} \end{cases}$$

Étudier le comportement de cette méthode pour des petites valeurs de n.

Exercice 7 – Recherche dichotomique

Définir une méthode récursive `boolean dichotomieRec(float[] t, float x, int d, int f)` qui recherche la présence d'un élément `x` dans un tableau `t` trié, entre les bornes `d` et `f` de ce tableau. Une telle méthode pourra être appelée par la méthode

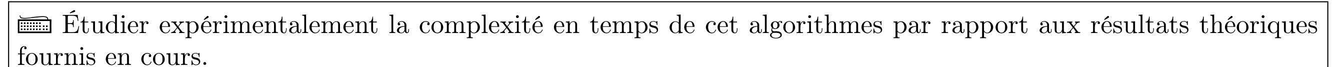
```
public static boolean dichotomie(float[] t, float x){
  return dichotomieRec(t, x, 0, t.length - 1);
}
```

Quelle est la complexité pire cas de cette méthode, en termes de nombre d'appels récursifs ?

Vérifier expérimentalement le résultat.

Exercice 8 – Retour sur les tris (1/2)

Proposer une implémentation du tri rapide.


Étudier expérimentalement la complexité en temps de cet algorithmes par rapport aux résultats théoriques fournis en cours.

Exercice 9 – Retour sur les tris (2/2)

Proposer une implémentation du tri par fusion.

Étudier expérimentalement la complexité en temps de cet algorithmes par rapport aux résultats théoriques fournis en cours.

Pour aller plus loin...

Exercice 10 – Sudoku

Définir une classe `Sudoku` permettant de gérer une grille de sudoku classique (9×9). Définir un constructeur `sudoku()` générant une grille vide. Définir un constructeur `sudoku(int n)` générant une grille de sudoku *valide*, dont n cases sur les 81 ont été préremplies aléatoirement. Surcharger la méthode `toString()`. Définir une méthode `resoudre()` qui retourne la liste de toutes les grilles solutions associées à une grille de sudoku. (Cette méthode pourra renvoyer 0, 1 ou plusieurs grilles, suivant que la grille de sudoku n'autorise pas de solution, une solution unique, ou bien plusieurs solutions.)

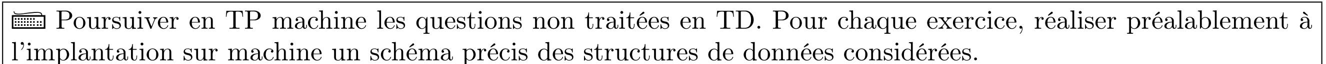
Sujet TD/TP 03 : Arbres : structures et manipulation

Note : les questions comportant le symbole ⌨ seront traitées pendant les séances de TP sur machine.

Remarques préliminaires

Dans tout ce sujet, on ne s'intéressera qu'à des arbres contenant des valeurs entières (`int` ou bien `Integer`). Le principe de la plupart des exercices resterait le même en changeant la nature des valeurs contenues dans ces arbres, notamment par le biais d'une paramétrisation des classes (*template*).

Chacune des classes créées dans les exercices 1 à 3 devra implémenter l'interface `BTree.java` décrite en fin de sujet. Chacune devra également disposer : d'un constructeur par défaut (sans paramètre) créant un arbre vide ; d'un constructeur à un seul paramètre `int` construisant un arbre formé d'une racine unique ; d'un constructeur à trois paramètres (un `int` et deux arbres) construisant un arbre par enracinement des deux arbres sur la racine.



⌨ Poursuiver en TP machine les questions non traitées en TD. Pour chaque exercice, réaliser préalablement à l'implantation sur machine un schéma précis des structures de données considérées.

Exercice 1 – Arbres binaires : implantation chaînée

Un arbre binaire peut être implanté à partir d'une structure de nœud contenant un élément de l'arbre (la "valeur" au nœud), et deux références sur des "sous-arbres" binaires (fils gauche et fils droit). Cette implantation permet donc de définir un arbre binaire depuis sa racine vers ses feuilles, via les nœuds fils successifs. Créer une classe `BTreeCA.java` implantant ce schéma. Tester les différentes méthodes via le `main` de la classe.

Créer une classe `BTreeCS.java` qui hérite de `BTreeCA.java`, implantant un schéma symétrique qui permet à un nœud père d'accéder à ses fils et vice versa. (On considérera que le nœud racine est son propre père.) Tester les différentes méthodes via le `main` de la classe.

Exercice 2 – Arbres binaires : implantation par tableau 2D

On considère maintenant une implantation des arbres binaires par tableau bidimensionnel régulier. Ce tableau contient autant de colonnes que d'éléments dans l'arbre, chaque colonne codant un nœud de l'arbre. Il contient par ailleurs 4 lignes. Pour un nœud / une colonne donné(e), la première ligne / case correspond à la valeur contenue au nœud. Les seconde et troisième lignes / cases contiennent les positions (c'est-à-dire l'indice de colonne) des nœuds correspondant respectivement aux fils gauche et droit. La quatrième ligne / case correspond à la position du nœud père. Créer une classe `BTreeT2.java` implantant ce schéma. Tester les différentes méthodes via le `main` de la classe.

Exercice 3 – Arbres binaires : implantation par tableau 1D

Il est enfin possible de proposer une implantation des arbres binaires par tableau monodimensionnel. Pour un arbre de profondeur p, le nombre de cases du tableau est alors de $2^p - 1$; la racine est stockée en position $2^{p-1} - 1$, sachant que l'indexation se fait de 0 à $2^p - 2$. Les nœuds du sous-arbre gauche (resp. droit) sont alors stockés dans les cases d'indices 0 à $2^{p-1} - 2$ (resp. 2^{p-1} à $2^p - 2$). Créer une classe `BTreeT1.java` implantant ce schéma. Tester les différentes méthodes via le `main` de la classe. N.B. : Il faudra alors représenter l'arbre comme un arbre complet, quitte à gérer des cases lacunaires dans le tableau. Il faudra également gérer l'augmentation de capacité du tableau en cas de dépassement dû à une augmentation de la profondeur ; pour ce faire, on réfléchira à la manière dont s'intercalent les valeurs des nœuds.

Annexe – Interface `BTree.java`

```java
interface BTree{

  boolean isEmpty();
  /* accesseur : renvoie vrai si l'arbre ne contient aucun element */

  BTree getRoot() throws Exception;
  /* accesseur : renvoie la racine de l'arbre */

  int getValue() throws Exception;
  /* accesseur : renvoie la valeur contenue a la racine de l'arbre */

  BTree getLeftTree() throws Exception;
  /* accesseur : renvoie le sous-arbre gauche de l'arbre */

  BTree getRightTree() throws Exception;
  /* accesseur : renvoie le sous-arbre droit de l'arbre */

  int getLeftValue() throws Exception;
  /* accesseur : renvoie la valeur a la racine du sous-arbre gauche */

  int getRightValue() throws Exception;
  /* accesseur : renvoie la valeur a la racine du sous-arbre droit */

  void setLeftTree(BTree leftTree) throws Exception;
  /* modificateur : ajoute un sous-arbre gauche a la racine de l'arbre (si libre) */

  void setRightTree(BTree rightTree) throws Exception;
  /* modificateur : ajoute un sous-arbre droit a la racine de l'arbre (si libre) */

  void setLeftValue(int leftSubRoot) throws Exception;
  /* modificateur : ajoute une valeur en fils gauche de la racine (si libre) */

  void setRightValue(int rightSubRoot) throws Exception;
  /* modificateur : ajoute une valeur en fils droit de la racine (si libre) */

}
```

Pour aller plus loin. . .

Exercice 4 – Arbres généraux : implantation par tableaux chaînés

Par opposition à un arbre binaire, où chaque nœud possède de 0 à 2 fils, un arbre général peut contenir de 0 à un nombre fini, mais arbitrairement grand, de fils. Par ailleurs, les fils d'un nœud dans un arbre général sont ordonnés, du premier au dernier, et la notion d'orientation (gauche, droite) des arbres binaires n'a donc plus cours. Il est possible de proposer une implantation des arbres généraux, par implantation chaînée asymétrique, c'est-à-dire avec chaque nœud référençant ses nœuds fils. Une implantation présentant des similitudes avec celle de l'exercice 1 peut alors être considérée. La différence réside dans le fait que les attributs "fils gauche" et "fils droit" sont remplacés par un tableau ou une liste de fils. Proposer une implantation des arbres généraux basée sur ce schéma.

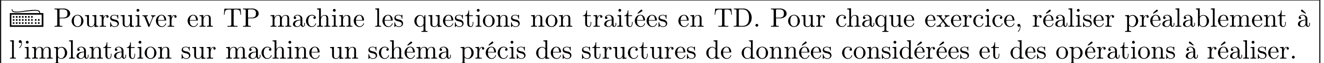
Exercice 5 – Arbres généraux : implantation purement chaînée

Il est possible d'éviter de passer par l'utilisation d'un tableau ou d'une liste de fils, en faisant en sorte que chaque nœud référence d'une part son premier fils et d'autre part son frère "cadet" immédiat. Proposer une implantation des arbres généraux basée sur ce schéma.

Sujet TD/TP 04 : Forêt d'arbres généraux : dictionnaire

Note : les questions comportant le symbole ⌨ seront traitées pendant les séances de TP sur machine.

Remarques préliminaires



⌨ Poursuiver en TP machine les questions non traitées en TD. Pour chaque exercice, réaliser préalablement à l'implantation sur machine un schéma précis des structures de données considérées et des opérations à réaliser.

Exercice 1 – Dictionnaire : structure de données

On souhaite développer une classe `Dictionnaire` capable de stocker des mots, définis sur l'alphabet à 26 lettres allant de *a* à *z*. La première idée envisagée est généralement de développer une liste triée de ces mots, par ordre lexicographique. Toutefois, afin de tirer parti de la redondance des mots, il est alternativement possible de les stocker dans un arbre général, ou plus précisément dans une forêt d'arbres généraux.

Concrètement, cette forêt se compose d'une structure linéaire de 26 éléments (au plus) allant de la lettre *a* à la lettre *z*. Chaque élément / lettre est lié(e) à un arbre général, qui contient tous les mots débutant par cette lettre. Cet arbre code plus précisément les suffixes de ces mots, amputés de cette première lettre. Pour ce faire, il contient lui-même une liste de 26 éléments (au plus) allant de la lettre *a* à la lettre *z*, etc.

À titre d'exemple, les mots *bassin*, *bas* et *bidon*, sont « stockés » dans l'élément *b* de la forêt. La lettre initiale *b* de ces trois mots y est factorisée, et les sous-mots *assin*, *as* et *idon* sont ensuite codés dans un arbre général, contenant une liste qui répartit les sous-mots *assin*, *as* dans l'élément de la lettre *a*, et *idon* dans celui de la lettre *i*, et ainsi de suite. On remarque, au passage, qu'un mot (par exemple *bas*) peut être un préfixe d'un autre (par exemple *bassin*) : ce point aura son importance par la suite.

On propose d'implanter une telle structure de dictionnaire par une classe `Dictionnaire.java` ayant comme attribut un *tableau de 26 éléments* : un pour chaque lettre. Chacun de ces éléments sera une instance de la classe `NoeudLettre.java` et contiendra, entre autres, une lettre de l'alphabet et un sous-dictionnaire tel que celui qui vient d'être évoqué. Il est donc requis de développer deux classes mutuellement récursives, chacune faisant appel à l'une à l'autre dans leurs attributs.

Définir deux telles classes, leurs attributs et des constructeurs de base. Comment se caractérise un mot « valide » par rapport à des « sous-mots » dans la structure de données associée ? Comment traduire ceci en termes d'attributs ?

Exercice 2 – Dictionnaire : opération d'ajout

Définir une méthode permettant d'ajouter un mot dans le dictionnaire. On fera en sorte de garantir la validité du mot saisi (taille et contenu valides).

Exercice 3 – Dictionnaire : opérations d'observation

Définir une méthode `boolean estPresent(String mot)` qui teste la présence d'un mot dans le dictionnaire. Définir des méthodes `int nombreMots()` et `int nombreLettres()` qui calculent le nombre de mots présents dans le dictionnaire, ainsi que le nombre de lettres associé. Définir une méthode `int nombreLettresPhy()` qui calculent le nombre de lettres effectivement codées dans le dictionnaire. En déduire une méthode calculant le taux de compression de cette structure, par rapport à un dictionnaire par liste « classique ».

Exercice 4 – Dictionnaire : affichage

Définir une méthode affichant tous les mots du dictionnaire dans l'ordre lexicographique.

Pour aller plus loin...

Exercice 5 – Dictionnaire : optimisation (1/2)

L'utilisation d'un tableau statique de 26 éléments n'est pas très efficace, dans la mesure ou de nombreuses cases du tableau restent inutil(isé)es, en particulier à mesure que l'on progresse dans les étages inférieurs de la forêt. Une première alternative consiste à remplacer ces tableaux statiques par des listes chaînées, afin de ne représenter effectivement que les éléments existants. Proposer une variante de la structure de dictionnaire sur cette base.

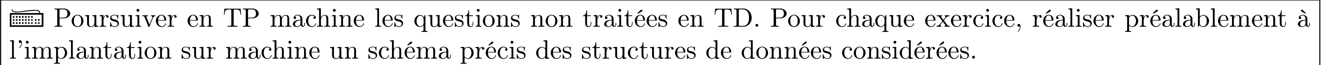
Exercice 6 – Dictionnaire : optimisation (2/2)

L'inconvénient majeur de la liste chaînée est l'impossibilité de rechercher un élément, ou d'en ajouter un, en temps logarithmique. Pour résoudre ce problème, proposer une variante de la structure de dictionnaire en remplaçant la liste chaînée par un arbre binaire de recherche.

Sujet TD/TP 05 : Arbres binaires de recherche

Note : les questions comportant le symbole ⌨ seront traitées pendant les séances de TP sur machine.

Remarques préliminaires


⌨ Poursuiver en TP machine les questions non traitées en TD. Pour chaque exercice, réaliser préalablement à l'implantation sur machine un schéma précis des structures de données considérées.

Exercice 1 – Arbres binaires de recherche

Créer une classe ABR.java implantant les arbres binaires de recherche pour les nombres flottants. Cette classe devra notamment permettre de créer un arbre binaire de recherche vide, et d'ajouter un élément. Il devra également surcharger la méthode toString, afin de permettre un affichage ordonné des valeurs contenues dans l'arbre par un parcours adapté. Tester les différentes méthodes via le main de la classe.

Exercice 2 – Arbres binaires de recherche équilibrés

Un arbre est équilibré si la différence de hauteur entre les deux sous-arbres à chaque nœud de l'arbre n'est jamais supérieure à 1. Compléter la classe ABR.java en définissant une méthode calculant la hauteur d'un arbre, ainsi qu'une méthode indiquant si l'arbre est équilibré. Créer une méthode d'ajout d'élément afin que chaque ajout aboutisse à un arbre qui reste équilibré. Tester les différentes méthodes via le main de la classe.

Pour aller plus loin…

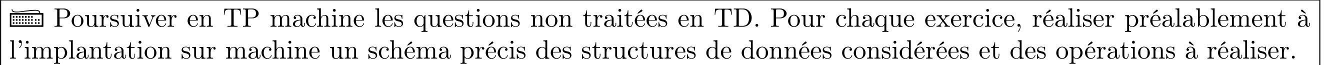
Exercice 3 – Tri par arbres binaires de recherche

Compléter la classe ABR.java par une méthode ArrayList<Double> TriABR(ArrayList<Double> t) qui crée une liste triée à partir de la liste passée en paramètre. Comparer sa complexité en temps avec celles d'autres méthodes de tri préalablement implantées.

Sujet TD/TP 06 : Collections et tables de hachage

Note : les questions comportant le symbole ⌨ seront traitées pendant les séances de TP sur machine.

Remarques préliminaires



⌨ Poursuiver en TP machine les questions non traitées en TD. Pour chaque exercice, réaliser préalablement à l'implantation sur machine un schéma précis des structures de données considérées et des opérations à réaliser.

Exercice 1 – Application d'une fonction de hachage

On souhaite insérer des éléments de type `entier` dans une structure de données implantée par le biais d'une table de hachage. On rappelle que la méthode de la division permet d'obtenir une fonction de hachage de l'ensemble des clefs vers $[0,\ldots,m-1]$:

$$h_{div}(k) = k \ \% \ m$$

où % représente l'opération modulo.On considère ici une table de hachage avec 13 adresses ($m = 13$).

12	
11	
10	
9	
8	
7	
6	
5	
4	
3	
2	
1	
0	

1. Insérer les clés 26, 37, 24, 30, et 11 dans la table de hachage ci-dessus en utilisant la résolution des collisions par adressage ouvert et sondage linéaire avec la fonction $h_i(k) = (h(k) + i) \ \% \ m$.
2. Rajouter maintenant les clés 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Quel problème rencontrez-vous ? Quelle solution proposez-vous ?
3. On considère une nouvelle table de hachage, et cette fois-ci on résout les collision par chaînage externe. Insérer les valeurs 26, 37, 24, 30, 11 ainsi que 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 dans cette table de hachage, et ébauchez la mémoire d'une manière compacte.

Exercice 2 – Fonction de hachage

On se place ici dans le cas d'une fonction de hachage simple par division h_{div}.

1. On suppose que les clés sont des entiers répartis uniformément entre 0 et n, avec $n \gg m$. Montrer que la fonction h_{div} vérifie l'hypothèse d'uniformité simple (on considère que le succès ou l'échec du hachage est identique pour toute valeur).
2. Quel inconvénient possède la méthode si les clefs sont des entiers et $m = 2^p$?

On suppose que les clefs sont des chaînes de caractères. On peut associer à la chaîne "$c_1 c_2 \ldots c_n$" l'entier $c_1 + 256 c_2 + \ldots + 256^{n-1} c_n$ (en assimilant caractère et entier ASCII), sur lequel on appellera la fonction de hachage h_{div}. Si $m = 255$, que se passe-t-il quand on permute les lettres d'une clef ?

Exercice 3 – Complexité de hachage

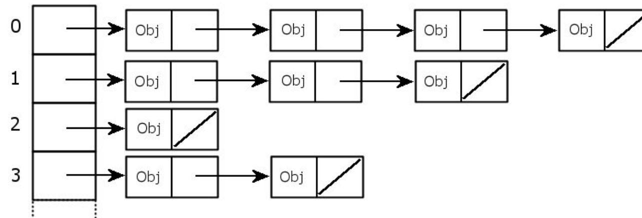
On considère pour l'instant des tables ouvertes.

1. On suppose qu'on a une fonction de hachage simplement uniforme. Montrer que le nombre moyen d'associations par case dans une table de hachage de taille m avec n clefs insérées est de $\frac{n}{m}$.
2. En déduire que la complexité en moyenne de la fonction de recherche est en $\mathcal{O}(1 + \frac{n}{m})$.

3. On considère une table avec redimensionnement dynamique : dès que le nombre de clefs insérées est supérieur à la taille de la table, on crée une nouvelle table de taille double et on y met toutes les associations de la table précédente. On fait n insertions dans une table de taille initiale 20. Montrer que le coût en temps est de l'ordre de n malgré les redimensionnements. En déduire qu'en moyenne la complexité de l'insertion est en $\mathcal{O}(1)$ pour les tables de hachage avec redimensionnement dynamique.

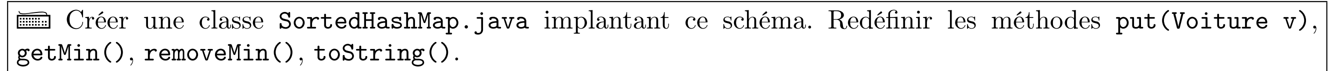
Exercice 4 – Structure de type Map : lien avec l'API Java

On souhaite ici stocker en mémoire une collection d'objets `Voiture` pouvant être référencées par leurs prix de vente `maVoiture.getPrix()` représentés par des éléments de type `double`. Plusieurs voitures peuvent avoir le même prix de vente. Pour stocker efficacement ces données, on souhaite construire une structure de données en Java basée sur l'API `java.util`. On utilisera une structure associative (Clé, Valeur) où "Clé" sera un prix et "Valeur" sera un ensemble (Set) de voitures ayant ce prix.



On pose également les contraintes suivantes :

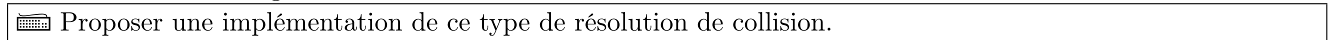
1. les clés de la structure associative doivent être maintenues triées dans la structure afin de minimiser le temps d'accès à la voiture ayant le prix le moins élevé ;
2. le coût d'insertion/suppression d'un nouveau véhicule doit être le plus faible possible ;
3. on ne s'occupe pas ici de l'espace mémoire.


Créer une classe `SortedHashMap.java` implantant ce schéma. Redéfinir les méthodes `put(Voiture v)`, `getMin()`, `removeMin()`, `toString()`.

Pour aller plus loin...

Exercice 5 – Cuckoo Hashing

Rendez-vous à l'url suivante `http://www.lkozma.net/cuckoo_hashing_visualization/` pour en apprendre plus sur les tables de hachage "Cuckoo".


Proposer une implémentation de ce type de résolution de collision.