UNIVERSITÉ DE MONTPELLIER
UNIVERSITÉ DE MONTPELLIER - FACULTÉ DES SCIENCES ET TECHNIQUES
PLACE EUGÈNE BATAILLON - 34095 MONTPELLIER CEDEX 5
Licence 2 informatique – 2019/2020

HLIN302 – Travaux Dirigés n° 6

Programmation impérative avancée
Alban MANCHERON et Pascal GIORGI

Rappel

Dans les précédents TD, nous la avons vu que la définition de la classe `Population` a progressivement évoluée vers un tableau dynamique ne comportant que les cellules vivantes du jeu de la vie. L'avantage de cette structure est de diminuer potentiellement l'impact sur la mémoire nécessaire pour représenter un état du jeu de la vie. Ce gain de mémoire se fait clairement au détriment du temps de calcul nécessaire pour récupérer l'état d'une cellule à une position donnée.

1 Mettons de l'ordre

Chercher un objet dans une pièce mal rangée est clairement plus fastidieux que lorsque la pièce est impeccablement rangée selon l'adage « une place pour chaque chose et chaque chose à sa place ».

Cela s'applique également à l'informatique. En effet, trouver un élément dans un ensemble lorsque celui-ci n'est pas ordonné nécessite d'énumérer tous les éléments un à un jusqu'à trouver l'élément recherché.

Imaginons maintenant que les éléments soient munis d'une relation d'ordre (total dans l'idéal), alors si l'ensemble est rangé selon cet ordre, la recherche d'un élément en particulier peux être optimisée, à l'instar de la recherche d'un mot dans un dictionnaire [1]. On parle alors de...

1.1 Recherche dichotomique

Dichotomie (dictionnaire de l'académie française – 9e édition)

n. f. XVIIIe siècle. Emprunté du grec *dikhotomia*, « division en deux parties égales ».
Logique : Division d'un concept en deux concepts contraires qui en recouvrent toute l'extension. On procède par dichotomie lorsqu'on divise le genre animal en Vertébrés et en Invertébrés. Par ext. Méthode de division et subdivision binaires. Classification par dichotomie.

Appliqué à l'informatique, il s'agit de découper en deux parties l'espace de recherche selon un critère et de décider à partir d'un test simple si l'élément recherché est dans l'une ou l'autre des parties puis de réitérer le processus jusqu'à isoler l'élément ou prouver qu'il n'appartient pas à l'ensemble initial.

Appliqué au jeu de la vie, cela revient à ordonner les `Cellule`s d'une `Population` selon une relation d'ordre simple, choisir une `Cellule` de référence dans l'ensemble et vérifier si une `Cellule` donnée est égale à la `Cellule` de référence. Si ce n'est pas le cas, il suffit de déterminer si la `Cellule` cherchée se situe avant ou après la `Cellule` de référence.

En choisissant systématiquement la cellule médiane de l'espace de recherche, à chaque itération la taille de l'ensemble est divisée par 2.

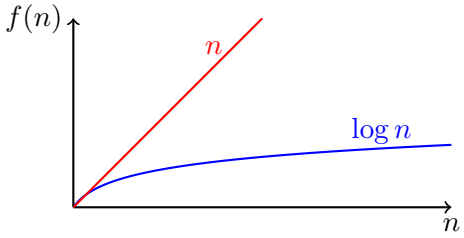
1. Si vous savez...Le gros livre avec plein de mots rangés dans l'ordre alphabétique et tout plein de définitions.

Ainsi si la `Population` contient n `Cellule`s, après la première itération, la recherche ne porte plus que sur la moitié des `Cellule`s. Après la seconde itération, la recherche ne porte plus alors que sur un quart des `Cellule`s, ..., après la i^{e} itération, la recherche ne porte que sur $\frac{n}{2^i}$ `Cellule`s.

Dans le pire des cas (la cellule recherchée n'est pas présente dans la `Population`), la recherche s'arrêtera après la k^{e} itération, *i.e.* le plus petit $k \in \mathbb{N}^+$ tel que :

$$\begin{aligned} n\,\frac{1}{2^k} &\leq 1 \\ n &\leq 2^k \\ \log_2 n &\leq k \end{aligned}$$

Soit à l'itération $\boxed{k = \lceil \log_2 n \rceil}$. Ceci signifie que si les éléments sont ordonnés, la recherche prendra de l'ordre de $\log n$ opérations, contre n opérations lorsqu'ils ne sont pas rangés.



Supposons que la classe `Cellule` définisse les méthodes :

- `estAvant`
- `estEquivalente`
- `estAvantOuEquivalente`
- `estApres`
- `estDifferente`
- `estApresOuEquivalente`

permettant de savoir si, étant données 2 cellules c_1 et c_2, c_1 est respectivement avant c_2, après c_2, équivalente à c_2, avant ou équivalente à c_2, après ou équivalente à c_2 ou bien différente de c_2.

Il est alors possible d'effectuer une recherche dichotomique dans un tableau de n `Cellule`s triée.

1. Définir les signatures des méthodes ci-dessus.
2. Écrire un programme prenant en entrée un `TableauCellule` T, ainsi qu'une `Cellule` c et qui recherche la `Cellule` correspondant à c dans le tableau T par dichotomie, puis qui renvoie son indice dans T si elle existe et -1 sinon [2].

1.2 Relation d'ordre

Il convient donc d'une part de munir les `Cellule`s d'une relation d'ordre pour pouvoir appliquer une telle stratégie de recherche ; *e.g.*, en définissant que deux cellules c_1 et c_2 sont ordonnées si leurs attributs respectifs respectent un critère spécifique.

On peut par exemple se baser sur les indices des cellules. Si $c_1 = \langle x_1, y_1, \cdots \rangle$ et $c_2 = \langle x_2, y_2, \cdots \rangle$, alors il est possible de définir les relations d'ordre suivantes (où l'opérateur binaire $\prec$ dénote le fait que la partie gauche précède la partie droite) :

$$c_1 \prec c_2 \quad \Leftrightarrow \quad x_1 + y_1 < x_2 + y_2 \tag{1}$$
$$c_1 \prec c_2 \quad \Leftrightarrow \quad x_1 + x_2 < y_1 + y_2 \tag{2}$$
$$c_1 \prec c_2 \quad \Leftrightarrow \quad (x_1 < x_2) \vee ((x_1 = x_2) \wedge (y_1 < y_2)) \tag{3}$$
$$c_1 \prec c_2 \quad \Leftrightarrow \quad (y_1 < y_2) \vee ((y_1 = y_2) \wedge (x_1 < x_2)) \tag{4}$$

2. La valeur -1 n'existe pas *stricto sensu* dans le domaine des entiers naturels. L'entier relatif -1 est codé sur n bits par $\overbrace{111\cdots1}^{n}$. Ce codage correspond, pour les entiers naturels, à la plus grande valeur représentable avec n bits, soit $2^n - 1$. Renvoyer -1 revient par conséquent à renvoyer une valeur supérieure à l'indice maximal de n'importe que tableau représentable en mémoire.

Quelle que soit la relation d'ordre choisie, on peut définir l'ordre inverse (dénoté $\succ$) ainsi : si 2 `Cellule`s c_1 et c_2 sont telles que $c_1 \prec c_2$, alors $c_2 \succ c_1$. Enfin, si $c_1 \not\prec c_2$ et $c_2 \not\prec c_1$, alors cela signifie que c_1 et c_2 sont équivalentes (par rapport à la relation d'ordre, noté $c_1 \equiv c_2$). On parle ici d'ordre strict.

1. Discuter les relations d'ordre proposées ci-dessus.
2. Quelle semble être la relation d'ordre la plus appropriée dans le contexte du jeu de la vie ?
3. Écrire les méthodes déclarées dans la classe `Cellule` qui reflètent la relation (3).

1.3 Tribulations d'un tri qui fait des bulles

Afin de tester la recherche dichotomique, il convient de définir un objet `TableauCellule` qui soit trié. C'est fastidieux et à terme il faudra automatiser le tri des `Cellule`s au sein d'une `Population`.

Les algorithmes de tri sont très majoritairement basés sur un paradigme simple : calculer la place d'un élément dans la collection et relancer le calcul sur les éléments restant à placer.

Un des algorithmes permettant de trier un ensemble de valeur parmi les plus simple à mettre en place est le « tri bulle ». Son principe est le suivant. On parcours chaque élément en le comparant à son successeur dans la collection (le tableau, la liste, ...). Si cet élément est mal placé par rapport à son voisin, on échange les deux éléments. Ainsi, l'élément qui doit se retrouver à la fin du tableau est propagé tel une bulle vers la fin [3]. Il suffit de réitérer le calcul sur les éléments non placés (tous sauf le dernier), et ainsi de suite.

1. Écrire une fonction `triBulle` prenant en entrée un `TableauCellule` et qui le trie en appliquant la méthode précédemment décrite.
2. Combien d'opérations sont effectuées (en ordre de grandeur) lors de l'appel de cette fonction sur un tableau de n valeurs.

Bien évidemment, il est possible de profiter du passage sur les éléments pour vérifier s'ils ne sont pas déjà triés.

3. Modifier la fonction `triBulle` afin d'arrêter le tri dès que tous les éléments sont triés.

Il ne reste plus qu'à tester le travail réalisé. Pour cela une manière simple est de générer un tableau de `TAB_SIZE` `Cellule`s (où `TAB_SIZE` est une constante à prédéfinir), puis à initialiser aléatoirement les coordonnées de chacune des `Cellule`s. Ensuite, il reste à trier le tableau ainsi obtenu, puis à tester la recherche en générant une `Cellule` et en la recherchant (il est possible d'encapsuler ce code dans une boucle afin de faire plusieurs recherches).

4. Écrire un programme de test tel que décrit ci-dessus.

1.4 Plus rapide que le tri bulle

Le « tri rapide », contrairement à ce que son nom laisse paraître, n'est pas nécessairement le plus rapide des algorithmes de tri. Il demeure néanmoins très efficace en moyenne (plus que le tri bulle) et demeure simple à mettre en place.

Son principe est le suivant :

1. Choisir un des éléments à placer (au hasard ou selon un critère fixe) que l'on appellera pivot [4].
2. Passer tous les éléments qui sont avant lui dans le tableau, et qui ne devraient pas l'être, après ce pivot ; inversement, placer tous les éléments qui sont après lui dans le tableau, mais qui ne devraient pas l'être, avant ce pivot.
3. Le pivot se retrouve par conséquent à sa place dans le tableau. Il reste à appliquer la même procédure sur les éléments qui sont avant lui, puis à faire de même sur les éléments qui sont après lui [5].

3. C'est la transitivité de la relation d'ordre qui permet de garantir ce résultat.
4. Un choix assez commun est de prendre l'élément médian.
5. Encore de la dichotomie ! ! !

Clairement, l'étape 2 est celle qui est la plus compliquée à mettre en place. Une manière traditionnelle de faire est de parcourir tous les éléments qui le précèdent (en partant du plus éloigné vers le pivot) jusqu'à ce qu'un élément mal placé par rapport au pivot soit rencontré, puis faire de même sur les éléments qui sont après le pivot. Ainsi, on se retrouve avec un élément avant qui doit passer après et un élément après qui doit passer avant. Il suffit simplement de les échanger et de continuer les parcours vers le pivot comme s'ils avaient été bien placés depuis le début. Si toutefois il n'y a pas d'élément en amont ou en aval du pivot à échanger, alors c'est le pivot lui même qui change de place et qui va ainsi converger vers sa « bonne » place dans le tableau.

1. Écrire une fonction `triRapide_aux` prenant en entrée un `TableauCellule`, un indice de début et un indice de fin, qui choisit un élément pivot et le place correctement, puis relance récursivement le calcul sur les deux parties séparées par le pivot.
2. Écrire une fonction `triRapide` qui appelle la fonction `triRapide_aux` avec le bon paramétrage.
3. Combien d'opérations sont effectuées (en ordre de grandeur) lors de l'appel de cette fonction sur un tableau de n valeurs.
4. Tester cette nouvelle fonction en comparant les résultats avec ceux obtenus précédemment avec le tri bulle. On s'assurera dans un premier temps que l'implémentation est correcte. Dans un deuxième temps, vous pourres essayer de mesurer le temps d'exécution de ces deux méthodes de tri sur de très grands tableaux. Vous utiliserez le type `std::time_t` et les fonctions `std::time(NULL)` et `std::difftime` définies dans le fichier `ctime` (cf exemple ci-dessous).

```cpp
#include <iostream>
#include <ctime>
int main () {
  std::time_t start = std::time(NULL);

  double d;
  for (int n=0; n<10000; ++n) {
    for (int m=0; m<100000; ++m) {
      d += d*n*m;
    }
  }

  std::cout << "temps ecoulé : "<< std::difftime(std::time(NULL), start) << " s.\n";

return 0;
}
```

1.5 Un tas de tris, mais un tri par tas

Le « tri par tas » – comme son nom l'indique – utilise une structure de données appelée tas pour trier les données.

Pour définir un tas, il est nécessaire de commencer par « quelques » définitions d'usage (les mots soulignés).

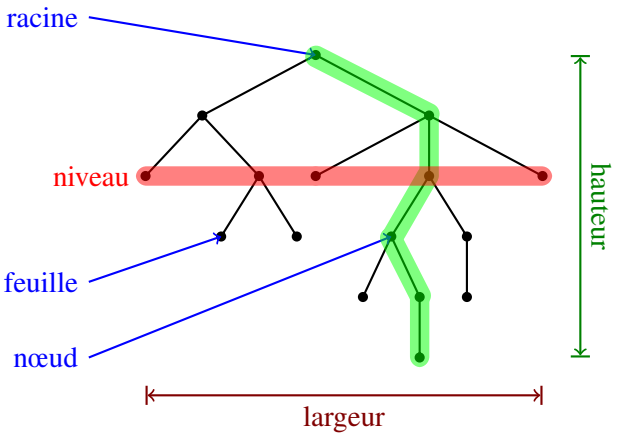
1.5.1 Définitions

Un arbre enraciné – en informatique – est une structure de données récursive permettant de représenter et d'organiser des informations sous forme arborescente.

Cette structure se compose d'éléments particuliers appelés nœuds reliés entre eux par des relations de parenté (père/fils), appelées arêtes.

Dans un arbre, tous les nœuds ont exactement un – et un seul – père, à l'exception d'un nœud particulier qui n'en a pas, appelé la racine de l'arbre.

Lorsqu'un nœud n'a pas de fils, alors ce nœud est appelé feuille de l'arbre.



Chaque nœud d'un arbre permet de définir un sous-arbre dont il est racine.

La profondeur d'un nœud est la distance (le nombre d'arêtes) entre la racine et ce nœud.

Le chemin, s'il existe, entre deux nœuds est la succession d'arêtes reliant ces deux nœuds. Pour qu'un chemin existe ; il faut nécessairement qu'un des nœuds apparaisse dans le sous-arbre enraciné par l'autre nœud.

La hauteur d'un arbre est égale à la plus grande profondeur d'un des nœud de l'arbre (qui est donc nécessairement une feuille).

Tous les nœuds à égale profondeur forment un niveau de l'arbre.

La largeur d'un arbre est égale au nombre maximum de nœuds d'un même niveau.

La taille d'un arbre est égale au nombre de nœuds qui le composent.

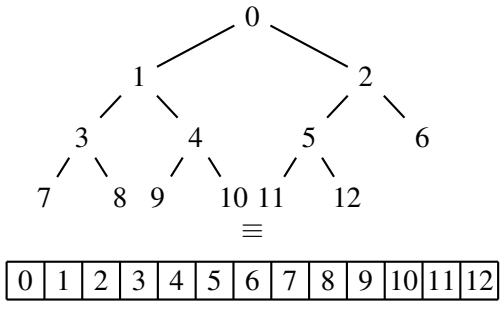
Il est également possible d'associer une étiquette sur un nœud (ou sur une arête) d'un arbre. Cette étiquette permet d'associer du contenu au nœud (ou à l'arête), qui a une place particulière dans l'arbre. Par exemple [6], un système de fichier peut être vu comme un arbre où un nœud est un dossier s'il a au moins un fils et un fichier si c'est une feuille ; l'étiquette des nœuds serait alors le nom du répertoire ou du fichier.

Un arbre binaire est un arbre enraciné tel que chaque nœud possède au plus 2 fils, dénotés respectivement fils gauche et fils droit.

Un arbre (binaire) est dit complet si toutes les feuilles de l'arbre sont au même niveau.

Un arbre (binaire) est dit quasi-complet si les hauteurs des feuilles de l'arbre ne diffèrent au plus que de 1 et que les feuilles les plus hautes sont le plus à gauche.

Les arbres binaires quasi-complets (et donc complets) présentent l'intérêt de pouvoir facilement modéliser sous forme d'un tableau. En effet, un tel arbre possédant n nœuds se représente facilement par un tableau de taille n où la racine est à l'indice 0 du tableau, son fils gauche à l'indice 1, son fils droit à l'indice 2, et ainsi de suite en indexant les nœuds d'abord en largeur puis en profondeur.



La navigation dans l'arbre se fait mathématiquement [7] en observant que pour un nœud ν ($0 \leq i < n$) :

- le nœud est une racine $\Leftrightarrow \nu = 0$.
- le nœud est une feuille $\Leftrightarrow \nu \leq \left\lfloor \frac{n-1}{2} \right\rfloor$.
- le père du nœud, s'il existe, est le nœud $\left\lfloor \frac{\nu-1}{2} \right\rfloor$ (-1 sinon [8]).
- le fils gauche du nœud, s'il existe, est le nœud $2\,\nu + 1$ (-1 sinon [8]).
- le fils droit du nœud, s'il existe, est le nœud $2\,\nu + 2$ (-1 sinon [8]).
- sa profondeur est $\lfloor \log_2(\nu + 1) \rfloor$.

Ainsi dans un arbre quasi-complet :

- le nombre de nœuds au i^{e} niveau est $\min\left(n + 1, 2^i\right) - 2^{i-1}$.
- sa hauteur est $\lfloor \log_2 n \rfloor$.
- sa largeur est $\max\left(n + 1 - 2^{\lfloor \log_2 n \rfloor}, 2^{\lfloor \log_2 n \rfloor - 1}\right)$.
- sa taille est n.

6. C'est un mauvais exemple
7. Pour information, la notation $\lfloor x \rfloor$ représente la partie entière de x. Ainsi le résultat de la division entière de $7/2$ est $\left\lfloor \frac{7}{2} \right\rfloor = 3$.
8. *cf* note 2 page 2.

Un tas est un arbre tel que l'information portée par chaque nœud est ordonnée de la racine à chacune des feuilles. En d'autres termes, la valeur associée à un nœud est supérieure ou égale à la valeur de son (ou de ses) fils.

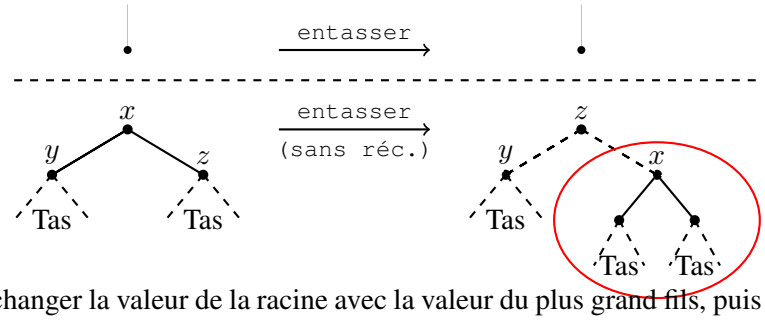
Un tas binaire est un arbre binaire quasi-complet respectant la propriété du tas.

1.5.2 Algorithme

Étant donné un tableau (plus généralement, une collection) de valeurs, il est aisé de le considérer comme un arbre binaire quasi-complet. Il est possible de réorganiser cet arbre binaire sous forme de tas binaire.

Pour cela, définissons la fonction `entasser` qui prend en entrée un arbre binaire quasi-complet et qui le réarrange en un tas binaire si l'une des deux propriétés ci-dessous est vérifiée :

1. l'arbre binaire est un tas
2. l'arbre binaire n'est pas un tas (la racine possède donc au moins un fils), mais les sous-arbres enracinés par les fils de la racine sont des tas.



Clairement, si l'arbre passé en paramètre vérifie la première propriété, la propriété du tas est vérifiée et `entasser` n'a rien de spécial à faire. Si l'arbre passé en paramètre vérifie la seconde propriété (l'arbre n'est donc pas une feuille et la racine possède par conséquent au moins un fils). Comme les sous-arbres enracinés par les fils de la racine sont des tas binaires, il suffit simplement d'échanger la valeur de la racine avec la valeur du plus grand fils, puis de rappeler la fonction `entasser` sur le sous-arbre que l'on vient de modifier (qui donc vérifie nécessairement l'une des deux propriétés ci-dessus).

Ainsi, il suffit d'appliquer la fonction `entasser` à tous les nœuds internes en partant du niveau le plus bas, puis à recommencer avec les nœuds internes du niveau précédent et ainsi de suite jusqu'à arriver à la racine.

1. Écrire l'algorithme puis le code correspondant à la fonction `entasser` (pour simplifier, on pourra définir des macros permettant de calculer le père, le fils gauche et le fils droit d'un nœud donné, si un nœud est une feuille, une racine, ...).
2. Écrire l'algorithme puis le code de la fonction `initTas` permettant de transformer un `TableauCellule` quelconque en tas.
3. Combien d'opérations sont effectuées (en ordre de grandeur) lors de l'appel de ces deux fonctions sur un tableau de n valeurs.

Une fois le tas initialisé, par définition, l'élément à la racine est nécessairement l'élément le plus grand parmi les n valeurs. Il suffit donc de le placer à sa place (la n^{e} position) dans le tableau. Cela peut se faire en échangeant la valeur de la racine avec l'élément à la dernière position du tableau. Il reste donc à trier les $n-1$ premiers éléments en se rendant compte que l'arbre représenté par ces éléments respecte l'une des deux propriétés requise pour l'appel à la fonction `entasser`.

Le tri par tas d'un tableau de n éléments se résume finalement à initialiser le tas à partir du tableau, à placer l'élément porté par la racine à sa place, rétablir la propriété du tas sur les $n-1$ premiers éléments, ..., et ainsi de suite jusqu'à ce que tous les éléments soient à la position désirée.

4. Écrire l'algorithme puis le code correspondant à la fonction `triParTas`.
5. Combien d'opérations sont effectuées (en ordre de grandeur) lors de l'appel de cette fonction sur un tableau de n valeurs.