NE PAS IMPRIMER CE DOCUMENT
il va être modifié et complété

Document de travail pour le TP3

Pour ce TP, les arbres binaires seront stockés sous forme de tableaux d'entiers. L'élément d'indice *i* aura comme fils gauche l'élément d'indice *2i* et comme fils droit l'élément d'indice *2i+1*. Les valeurs dans l'arbre sont des entiers positifs ou nuls. Une valeur *-1* dans le tableau correspond à un nœud inexistant dans l'arbre.

Hauteur et équilibrage d'un arbre binaire

Construisez une classe de tableau d'entiers avec :

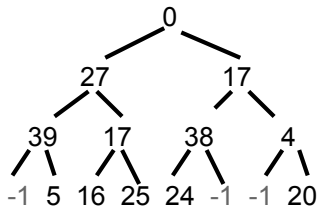
- une méthode d'**initiliasation aléatoire** d'un tableau d'entiers contenant uniquement des valeurs positives (ce sera un arbre binaire équilibré si l'indice maximum correspond à une des longueurs calculées en TD[1]).

- des tableaux de test initialisés comme suit :

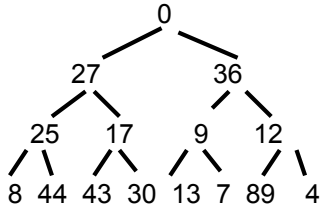
```
intT1[]={-2,0,27,17,39,17,38,4,-1,5,16,25,24,-1,-1,20};
intT2[]={-2,0,27,36,25,17,9,12,8,44,43,30,13,7,89,4};
intT3[]={-2,0,27,-1,25,17,-1,-1,8,44,43,30,-1,-1,-1,-1};
intT4[]={-2,0,27,-1,25,17,-1,-1,8,44,43,30,-1,-1,-1,-1};
intT5[]={-2,0,27,14,25,17,12,8,-1,-1,-1,-1,-1,-1,-1,-1};
```

  La première valeur de chaque tableau (*-2*) correspond à l'indice *0* du tableau que nous n'utilisons pas. Ces arbres sont illustrés ci-dessous :

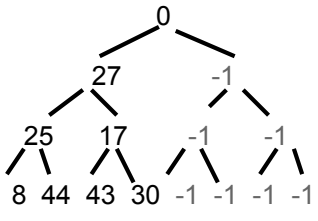
arbre de test T1



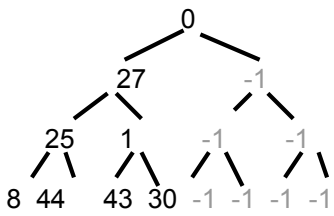
arbre de test T2



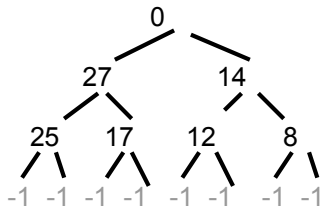
arbre de test T3



arbre de test T4



arbre de test T5



- une méthode **récursive** de calcul de la **hauteur d'un arbre binaire** stocké dans un tableau. La hauteur est la longueur du plus long parcours de la racine à une feuille de l'arbre.

  L'algorithme ci-dessous est écrit avec les accès aux arbres binaires : racine(a), fgauche(a), fdroit(a), et le test estVide(a) :

1 1,3,7,15, 31, ...

```
hauteur (a) : si estVide(a)
              alors res  ← 0
              sinon res ← 1 + max( hauteur(fgauche(a)) , hauteur(fdroit(a)) )
              fsi
```

Cet algorithme doit être adapté pour la représentation en tableau des arbres. Les accès aux fils gauche et droit se font en passant de l'indice *i* aux indices *2i* et *2i+1*. Un arbre est vide si l'indice qui désigne sa racine est supérieur au nombre d'éléments du tableau ou si la valeur du tableau à l'indice *rac* est *-1*. Dans le nouvel algorithme ci-dessous, *T* est le tableau, *rac* l'indice de la racine et *max* le nombre d'éléments (les indices utilisés étant dans [1..max]) :

```
haut (T, rac, max) :
    selon  rac > max :res ← 0
           T[rac] = -1 : res ← 0
           autres :res ← 1+max( haut(T, 2*rac,max) , haut(T, 2*rac+1,max) )
    finselon
    résultat res
```

– une méthode **récursive** permettant de vérifier si **un arbre binaire est équilibré**. Cet arbre est stocké dans un tableau. A chaque niveau de cet arbre, la différence de longueur des parcours de la racine à une feuille de l'arbre doit être au plus de 1.

L'algorithme ci-dessous est écrit avec les accès aux arbres binaires : racine(a), fgauche(a), fdroit(a), et le test estVide(a) :

```
equilibre (a) :
     si estVide(a)
     alors res  ← vrai
     sinon     equilibre (T, 2*rac, max)
           ET equilibre (T, 2*rac+1, max)
           ET | hauteur(T, 2*rac,max)- hauteur(T, 2*rac+1,max)| <=1
     fsi
```

Dans l'algorithme ci-dessous, *T* est le tableau, *rac* l'indice de la racine et *max* le nombre d'éléments, les indices utilisés sont dans [1..max] :

```
equilib (T, rac, max) :
  si rac > max ou T[rac] = -1
  alors res ← vrai
  sinon res ←    equilib (T, 2*rac, max)
             ET equilib (T, 2*rac+1, max)
             ET | hauteur(T, 2*rac,max)- hauteur(T, 2*rac+1,max)| <=1
  fsi
  résultat res
```

Ces deux méthodes doivent être testées sur les tableaux T1 à T5.

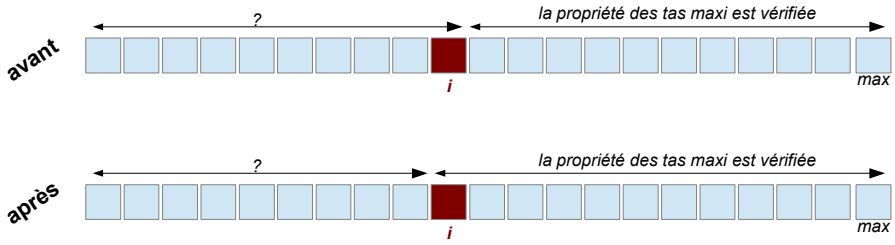
Remarques :

1) Vous aurez besoin des opérations *Math.max()* et *Math.abs()* dont les documentations sont disponibles sur : http://docs.oracle.com/javase/6/docs/api/java/lang/Math.html
2) Dans un programme, les conditions de la forme ci-dessus, `rac > max ou T[rac] = -1`, sont dangereuses. Pourquoi ?

Tri pas tas

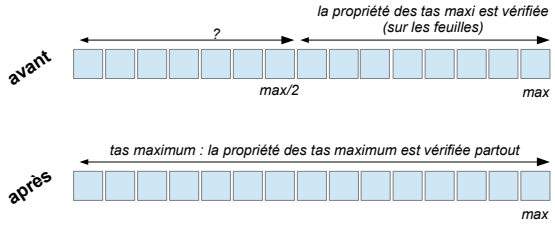
Ajoutez ensuite à votre classe, les méthodes vues en TD :

- `propMax(T, i, max)` qui suppose que la fin du tableau (de l'indice *i+1* à l'indice *max*) vérifie la propriété caractéristique des tas binaires maximum et étend cette propriété d'un cran vers la gauche (de *i* à *max*).



  Cette méthode est récursive ;

- `tasMax(T, max)` qui utilise la méthode `propMax()` pour faire du tableau *T* un tas binaire maximum.



  Cette méthode peut-être itérative ou récursive ;

- `triMax(T, max)` qui utilise la méthode `tasMax()` pour trier le tableau *T* en construisant un tas binaire maximum pour trouver le plus grand élément du tableau puis en mettant ce plus grand élément en fin de tableau et en reprenant la même démarche sur le tableau privé de son dernier élément. Cette méthode peut-être itérative ou récursive ;

N'oubliez pas d'ajouter des compteurs d'évaluation (comparaisons, permutations d'éléments) et de tester sur des tableaux aléataoires, aléatoirement croissants et aléatoirement décroissants. Vous trouverez à l'adresse :

http://www.sorting-algorithms.com/ et cliquer sur heap (*explore this algorithm*)

des illustrations du tri par tas (heapsort) sur des tableaux ayant diverses propriétés.