Les différentes méthodes de tries

Par Dimitri PIANETA

2015-2016

Table des matières

I) Définitions :

Qu'est-ce qu'un tri ? On suppose qu'on se donne une suite de N nombres entiers et on veut les ranger en ordre croissant (ou décroissant) au sens large. Ainsi, pour n=7, la suite (5, 2, 3, 0, 6, 1, 1) devra devenir (0, 1, 1, 2, 3, 5, 6).

II) Tri à Bulles (Bubble sort) :

Le nom de ce tri vient de ce que les éléments les plus grands (lourd) remontent vers la fin du tableau comme les bulles vers le haut d'un tube à essai.

C'est le tri le plus simple.

Méthode et implémentation :

Le tri à bulle est une méthode de tri qui consiste à comparer successivement tous les éléments adjacents d'un tableau et à les échanger si le premier élément est supérieur au second. On recommence cette opération tant que tous les éléments ne sont pas triés. À chaque étape de l'algorithme l'élément maximal est déplacé à la fin de la suite.

Voici un exemple d'application de cette méthode pour N =6 :

	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
Données	115	101	30	63	20	47

1er passage :

Échanges : 101<-> 115

30 <->115

63<->115

20<->115

47<->115

Résultat du 1er passage

101	30	60	20	47	115

Au 1er passage l'élément (115), le plus grand du tableau est déplacé en N-1 (ici 5ème) position.

2ème passage :

Échanges : 30<-> 101

63 <->101

20<->101

47<->101

Résultat du 2ème passage

30	63	20	47	101	115

Au 2ème passage l'élément (101), deuxième plus grand du tableau est déplacé en N-2 (ici 4ème) position.

3ème passage :

Pas d'Échanges : 30<63

Échanges 20<->6

47<->63

Résultat du 3ème passage

30	20	47	63	101	115

Au 3ème passage l'élément (63), 3ème plus grand du tableau est déplacé en N-3 (ici 3ème) position.

4ème passage :

Échanges :

20 <->30

Pas d'échange : 30<47

Résultat du 4ème passage

20	30	47	63	101	115

Au 4ème passage l'élément (47), 4ème plus grand du tableau est déplacé en N-4 (ici 2ème) position.

5ème passage :

Pas d'échange : 20<30

Résultat du 5ème passage

20	30	47	63	101	115

Au 5ème le tableau est trié et l'algorithme s'arrête et on s'aperçoit qu'il y a N-1 (5 passages).

Pseudo code :

```
  passage ← 0
REPETER
    permut ← FAUX
    POUR i VARIANT DE 1 A n - 1 - passage FAIRE
        SI a[i] > a[i+1] ALORS
            echanger a[i] ET a[i+1]
            permut ← VRAI
        FIN SI
    FIN POUR
    passage ← passage + 1
TANT QUE permut = VRAI
```

III) Tri par Sélection (Selection Sort)

L'idée est de trier un tableau en déterminant son plus petit, con deuxième plus petit, troisième plus petit, etc. élément. C'est-à-dire trouver la position du plus petit élément dans le tableau et ensuite échanger a[0] et a[i_1]. Ensuite de suite, déterminer la position i_2 de l'élément avec le plus petit des a[1],…,a[N-1] et échanger a[1] et a[i_2]. On continue de cette manière jusqu'à ce que tous les éléments soient dans la position correcte.

Un avantage de ce tri par sélection est qu'il est progressif, car à l'étape i de l'algorithme, le tableau est trié de a[0] jusqu'à a[i-1].

	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
Données	115	101	30	63	20	47

Sélection 20

Placement

20	101	30	63	115	47

Sélection 30

Placement

20	30	101	63	115	47

Sélection 47

Placement

20	30	47	63	115	101

Sélection 63

Placement

20	30	47	63	115	101

Sélection 101

Placement

20	30	47	63	101	115

Pseudo code :

```
Pour i variant de 1 à n-1 faire
      Trouver[j] le plus petit element de  [i+1:n]
      Echanger [j] et [i]
```

IV) Tri par Insertion (insertionSort)

L'idée est de trier successivement les premiers éléments du tableau. A la ième étape on insère le ième élément à son rang parmi les i-1 éléments précédents qui sont déjà triés entre eux. L'algorithme commence par s'exécuter à partir du 2ème élément du tableau. Cette méthode s'appelle aussi méthode du joueur de cartes. Les étapes successives sont décrites ci-dessous :

Voici un exemple d'application pour N=6

	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
Données	115	101	30	63	20	47

1er insertion :

101	115	30	63	20	47

2ème insertion :

30	101	115	63	20	47

3ème insertion :

30	63	101	115	20	47

4ème insertion :

20	30	63	101	115	47

5ème insertion :

20	30	47	63	101	115

Implémentation en java :

Pseudo code :

```
pour i variant de 2 à n faire
        INSERER a[i] à sa place dans a[1:i-1]
```

V) Tri par Shell (Shell sort)

Le tri Shell est une extension du tri par insertion.

Exemple de tri

Évolution du tableau au fil du tri shell. Le pas, représenté en rouge est également indiqué ainsi que la valeur en mémoire, sur laquelle porte la comparaison. En bleu, les valeurs sur lesquels portent les comparaisons à chaque étape.

Tableau	Mémoire	Pas	Commentaire
6 3 0 9 1 7 8 2 5 4	t(4)=1	4	Comparaison de mémoire=t(4) avec t(0) : t(4) reçoit t(0) puis t(0) reçoit mémoire
1 3 0 9 6 7 8 2 5 4	t(5)=7	4	Comparaison de mémoire=t(5) avec t(1) : pas d'échange
1 3 0 9 6 7 8 2 5 4	t(6)=8	4	Comparaison de mémoire=t(6) avec t(2) : pas d'échange
1 3 0 9 6 7 8 2 5 4	t(7)=9	4	Comparaison de mémoire=t(7) avec t(3) : t(7) reçoit t(3) puis t(3) reçoit mémoire
1 3 0 2 6 7 8 9 5 4	t(8)=5	4	Comparaison de mémoire=t(8) avec t(4) : t(8) reçoit t(4)
1 3 0 2 6 7 8 9 6 4	t(8)=5	4	Comparaison de mémoire avec t(0) : pas d'échange t(4) reçoit mémoire
1 3 0 2 5 7 8 9 6 4	t(9)=4	4	Comparaison de mémoire=t(9) avec t(5) : t(9) reçoit t(5)
1 3 0 2 5 7 8 9 6 7	4	4	Comparaison de mémoire avec t(1) : pas d'échange, t(5) reçoit mémoire
1 3 0 2 5 4 8 9 6 7	4	1	A ce stade, le pas diminue 4/3 donne un pas de 1

Implémentation en java :

Pseudo-code :

```
1. PROCEDURE tri_Insertion ( Tableau a[1:n],gap,debut)
2.     POUR i VARIANT DE debut A n AVEC UN PAS gap FAIRE
3.         INSERER a[i] à sa place dans a[1:i-1];
4. FIN PROCEDURE;
5.
6. PROCEDURE tri_shell ( Tableau a[1:n])
```

```
7.      POUR gap DANS (6,4,3,2,1) FAIRE
8.        POUR debut VARIANT DE 0 A gap - 1 FAIRE
9.          tri_Insertion(Tableau,gap,debut);
10.       FIN POUR;
11.     FIN POUR;
12. FIN PROCEDURE;
```

VI) Tri rapide (Quick Sort)

Principe

L'algorithme de tri rapide (ou Quick Sort) a pour principe de base : on choisit une valeur dans le tableau appelée pivot (nous prendrons ici la première valeur du tableau) et on déplace avant elle toutes celles qui lui sont inférieures et après elle toutes celles qui lui sont supérieures. Puis, on réitère le procédé avec la tranche de tableau inférieure et la tranche de tableau supérieure à ce pivot.

Voici un exemple :

13 18 9 15 7

Nous prenons le premier nombre en pivot : 13 et nous plaçons les nombres 9 et 7 avant le 13, les nombres 15 et 18 après le 13.

9 7 **13** 15 18

Il ne reste plus ensuite qu'à réitérer l'algorithme sur les deux sous-tableaux.

9 7

15 18

Le premier aura comme pivot 9 et sera réorganisé, le second aura comme pivot 15 et ne nécessitera aucune modification. Il restera alors deux sous-sous-tableaux.

7

18

Comme ces sous-sous-tableaux se résument à une seule valeur, l'algorithme s'arrêtera. Cet algorithme est donc récursif et l'une des difficultés est de répartir les valeurs de part et d'autre du nombre pivot. Revenons, à notre tableau initial. Nous allons chercher le premier nombre plus petit que 13 de gauche à droite avec une variable i, et le premier nombre plus grand que 13 de droite à gauche avec une variable j, jusqu'à ce que i et j se rencontrent.

13 18 9 15 7

On trouve 18 et 7. On les inverse et on continue.

13 7 9 15 18

Le rangement est presque fini, il ne reste plus qu'à inverser le pivot avec le 9.

9 7 **13** 15 18

Pseudo code :

```
Tri-Rapide(A,p,r)

        Si p<r
                Alors q←PARTITION(A,p,r)
                        Tri-Rapide(A,p,q-1)
                        Tri-Rapide(A,q+1,r)

PARTITION(A,p,r)
        X← A[r]
        i← p-1
        pour j← p à r-1
                faire si A[j]<= x
                        alors i←i+1
                                permuter A[i]←→ A[j]
        permute A[i+1] ←→A[r]
        retourner i+1

permute(A,I,j)

memoire←A[i]
A[i]=A[j]
A[j]=memoire
```

VIII) Tri par Fusion(Merge sort)

Le tri fusion est construit suivant la stratégie "diviser pour régner", en anglais "divide and conquer". Le principe de base de la stratégie "diviser pour régner" est que pour résoudre un gros problème, il est souvent plus facile de le diviser en petits problèmes élémentaires. Une fois chaque petit problème résolu, il n'y a plus qu'à combiner les différentes solutions pour résoudre le problème global. La méthode "diviser pour régner" est tout à fait applicable au problème de tri : plutôt que de trier le tableau complet, il est préférable de trier deux sous tableaux de taille égale, puis de fusionner les résultats.

L'algorithme proposé ici est récursif. En effet, les deux sous tableaux seront eux même triés à l'aide de l'algorithme de tri fusion. Un tableau ne comportant qu'un seul élément sera considéré comme trié : c'est la condition sine qua non sans laquelle l'algorithme n'aurait pas de conditions d'arrêt. Étapes de l'algorithme :

- Division de l'ensemble de valeurs en deux parties
- Tri de chacun des deux ensembles
- Fusion des deux ensembles

Tableau		Commentaire : ce qui est fait pour passer à la ligne suivante
5	4	Division du tableau en deux parties
5	4	Division de chaque sous tableau en deux parties
5	4	Division de chaque sous tableau en deux parties
5	4	Division des sous tableau bleu et rouge en deux parties, fusion des tableaux vert-rose
4	5	Fusion des sous tableaux bleu-rose et rouge-rose
4	5	Fusion des sous tableaux bleu-jaune et rouge-jaune
4	5	Fusion des sous tableaux bleu-vert et rouge-vert
7	8	Fusion des deux tableaux
8	9	Le tableau est trié, l'algorithme est terminé

Implémentation en java :

Code :

```
public static long[] triFusion(long [] tab) {
            int longueur = tab.length;
            if (longueur>0){
                  trifusion(tab,0, longueur-1);

            }

            return tab;

      }

       private static void trifusion(long[] tab, int deb, int fin) {
             if (deb!=fin)
        {
        int milieu=(fin+deb)/2;
        trifusion(tab,deb,milieu);
        trifusion(tab,milieu+1,fin);
        fusion(tab,deb,milieu,fin);
        }
    }



    private static void fusion(long tab[],int deb1,int fin1,int fin2)
    {
    int deb2=fin1+1;

    //on recopie les éléments du début du tableau
   long table1[]=new long[fin1-deb1+1];
    for(int i=deb1;i<=fin1;i++)
```

```
        {
        table1[i-deb1]=tab[i];
        }

    int compt1=deb1;
    int compt2=deb2;

    for(int i=deb1;i<=fin2;i++)
        {
        if (compt1==deb2) //c'est que tous les éléments du premier tableau ont
été utilisés
            {
            break; //tous les éléments ont donc été classés
            }
        else if (compt2==(fin2+1)) //c'est que tous les éléments du second
tableau ont été utilisés
            {
            tab[i]=table1[compt1-deb1]; //on ajoute les éléments restants du
premier tableau
            compt1++;
            }
        else if (table1[compt1-deb1]<tab[compt2])
            {
            tab[i]=table1[compt1-deb1]; //on ajoute un élément du premier tableau
            compt1++;
            }
        else
            {
            tab[i]=tab[compt2]; //on ajoute un élément du second tableau
            compt2++;
            }
        }
    }
```

IX) Tri par création

Autant le dire tout de suite, cet algorithme ne présente pas un grand intérêt au niveau algorithmique, il est relativement compliqué, pour un résultat pas franchement extraordinaire. Cependant, il est présenté ici à titre d'illustration et ne sera implémenté que dans le cas de tableaux. Cet algorithme peut être utilisé lorsque l'utilisateur souhaite conserver une copie du tableau original. Cependant, l'utilisateur aura sans doute un plus grand intérêt à faire une copie du tableau avant de le trier avec l'un des algorithmes les plus performants qui sont présentés à la suite.

Le principe de ce tri est de créer un tableau vide de même taille que le tableau à trier, puis de chercher dans le tableau à trier le plus petit élément afin de le placer en première position du tableau nouvellement créé. Ensuite, l'algorithme recherche successivement les éléments suivants afin de les placer, à la suite, dans le nouveau tableau. L'algorithme se termine lorsque tous les éléments du tableau ont été ajoutés.

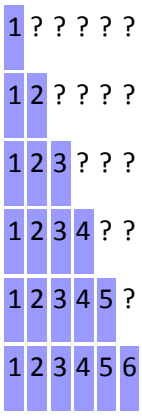
Pour mettre en place ce tri, il faut écrire une fonction intermédiaire qui permet de chercher les différents éléments successifs. Cette fonction est appelée "pos_suivant".

Exemple

Soit le tableau à trier suivant :

5 3 1 2 6 4

Le nouveau tableau créé va évoluer ainsi au fil de l'algorithme :



Implémentation en java :

Code :

```
public static long[] triParCreation(long tableau[])
    {
    int longueur=tableau.length;
    long result[]=new long[longueur];

    int posDernier=0;
    long mini=tableau[0];
    long maxi=tableau[0];
    for(int i=1;i<longueur;i++)
        {
        if (tableau[i]<mini)
            {
            posDernier=i;
            mini=tableau[i];
            }
        else if (tableau[i]>maxi)
            {
            maxi=tableau[i];
            }
        }
    result[0]=mini;

    for(int i=1;i<longueur;i++)
        {
        posDernier=posSuivant(tableau,posDernier,maxi);
        if (posDernier!=-1)
            {
            result[i]=tableau[posDernier];
            }
        else
```

```
            {
            System.err.println("Erreur dans le triParCreation, l'un des éléments
a été perdu !!!");
            }
        }
    return(result);
    }

    private static int posSuivant(long tableau[],int posDernier,long maxi)
    {
    int longueur=tableau.length;
    long valDernier=tableau[posDernier];

    int posSuivant=-1;
    long valSuivant=maxi;

    for(int i=0;i<longueur;i++)
        {
        if(tableau[i]==valDernier && i>posDernier)
            {
            return(i);
            }
        if(tableau[i]>valDernier && tableau[i]<valSuivant)
            {
            valSuivant=tableau[i];
            posSuivant=i;
            }
        if(tableau[i]==maxi && posSuivant==-1)
            {
            posSuivant=i;
            }
        }

    return(posSuivant);
    }
```

X) Tri trois médiane

C'est une amélioration du tri rapide. Ainsi le pivot est trois médianes.

Implémentation en java :

Code :

```
public static long[] triMedian(long [] tab) {

            int longueur = tab.length;
            triMedian(tab,0,longueur-1);
      return tab;

      }

    static  long medOf3(long[] array, int left, int right){
        int center = (left+right) / 2;if(array[left] > array[center])
            swap(array, left, center);
        if(array[left] > array[right])
            swap(array, left, right);
```

```
        if(array[center] > array[right])
            swap(array, center, right);
     return array[right - 1];
    }


    static void swap(long[] array, int index1, int index2) {

      long tempon = array[index1];
         array[index1] = array[index2];
         array[index2] = tempon;
       }


    private static int MedianOfThreePartition(long[] tab,int deb,int fin, long
pivot)
        {


      int lPointer = deb;
         int rPointer = fin - 1;
         while (true) {
           while (tab[++lPointer] < pivot)
             ;
           while (tab[--rPointer] > pivot)
             ;
           if (lPointer >= rPointer)
             break;
           else{

       long temp = tab[lPointer];
                  tab[lPointer]=tab[rPointer];
           tab[rPointer]=temp;

         }
        }

         long temp = tab[lPointer];
            tab[lPointer]=tab[fin-1];
            tab[fin-1]=temp;
         return lPointer;

        }




    private static void triMedian(long tableau[],int deb,int fin)
    {
    if(deb<fin)
        {
     long median = medOf3(tableau, deb, fin);
        int positionPivot=MedianOfThreePartition(tableau,deb,fin,median);
        triMedian(tableau,deb,positionPivot-1);
        triMedian(tableau,positionPivot+1,fin);
        }
    }
```

IX) Tri par tas

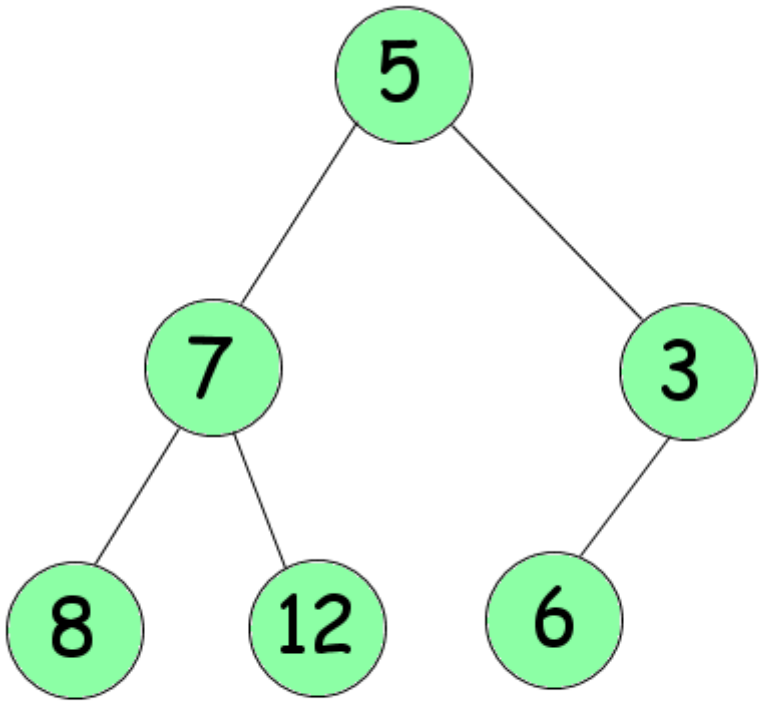
Le tri par tas porte également les noms suivants :

- Tri arbre
- Tri Maximier
- Heapsort
- Tri de Williams

Principe

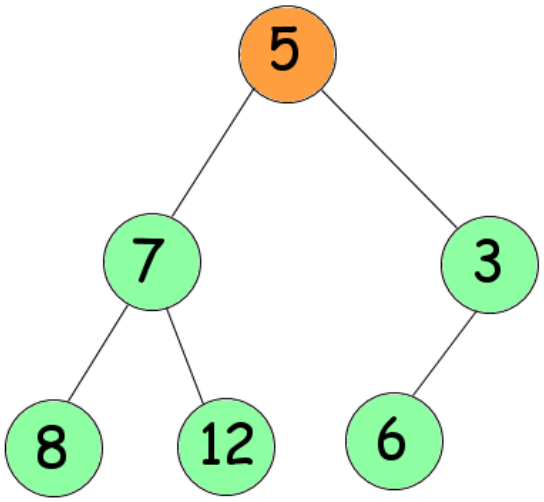
5 7 3 8 12 6

L'idée du tri pas tas est de concevoir votre tableau tel un **arbre**. Par exemple, le tableau ci-dessus devra être vu ainsi :

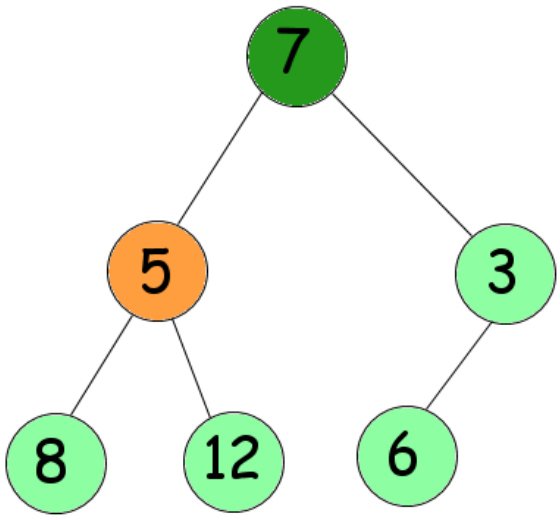


Chaque «case» de l'arbre est appelée **nœud**, le premier nœud porte le nom de **racine**, les derniers le nom de **feuille**. Si l'on est rendu au nœud numéro n (indice dans le tableau initial), ses deux feuilles porteront les numéros $2n$ et $2n+1$.

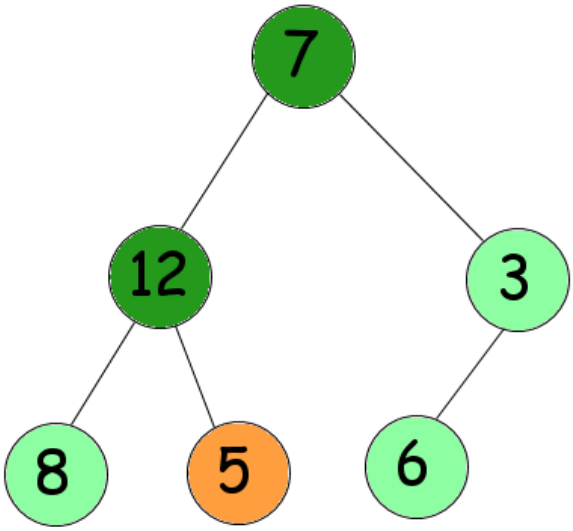
L'arbre n'est pas trié pour autant. Pour cela nous devrons le **tamiser**, c'est-à-dire prendre un nombre et lui faire descendre l'arbre jusqu'à ce que les nombres en dessous lui soient inférieurs. Pour l'exemple, nous allons tamiser le premier nœud (le 5).



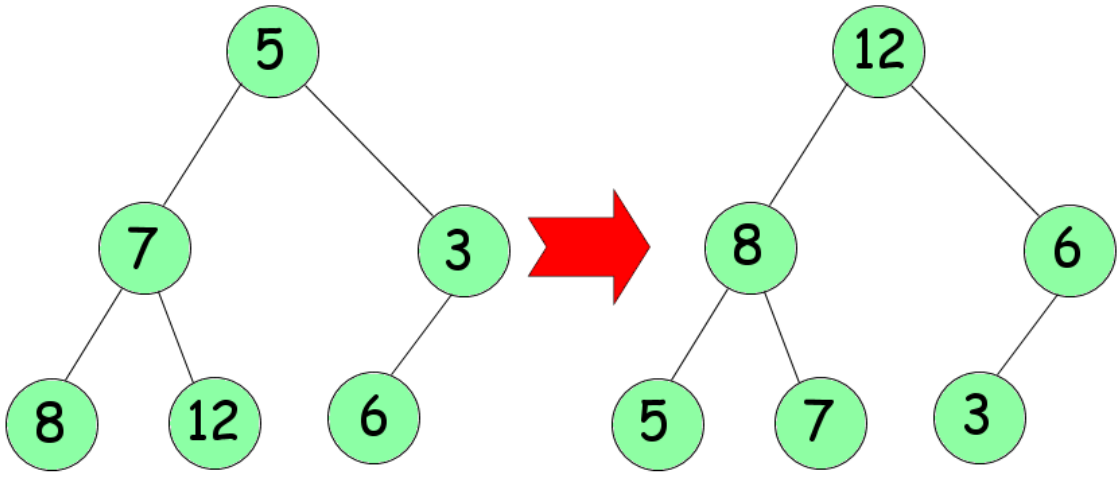
En dessous de lui se trouvent deux nœuds : 7 et 3. On choisit le plus grand des deux : 7. Comme 7 est plus grand que 5, on les inverse.



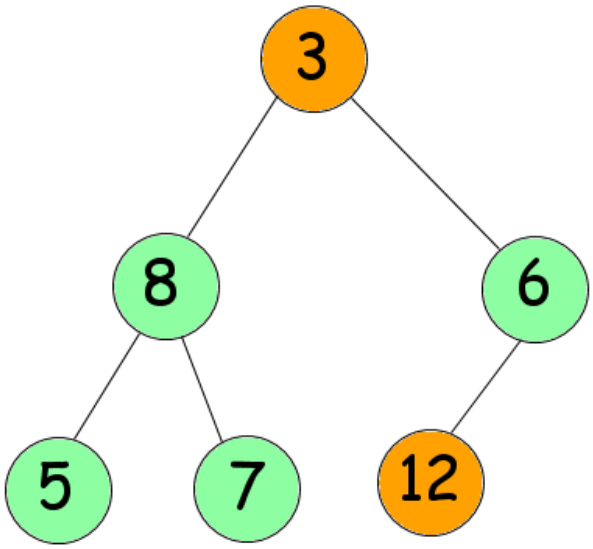
On recommence : les deux nœuds situés en dessous sont 8 et 12. Le plus grand est 12. Comme 12 est plus grand que 5, on les inverse.



Vous avez compris le principe du tamisage ? Très bien. Pour trier notre arbre par tas, nous allons le tamiser en commençant, non pas par la racine de l'arbre, mais par les derniers nœuds (en commençant par la fin en quelque sorte). Bien sûr, il est inutile de tamiser les feuilles de l'arbre : elles ne descendront pas plus bas ! Pour limiter les tests inutiles, nous commencerons par tamiser le 3, puis le 7 et enfin le 5. Du coup, nous ne tamiserons que la moitié des nœuds de l'arbre. Faites l'essai, cela devrait vous donner ceci :

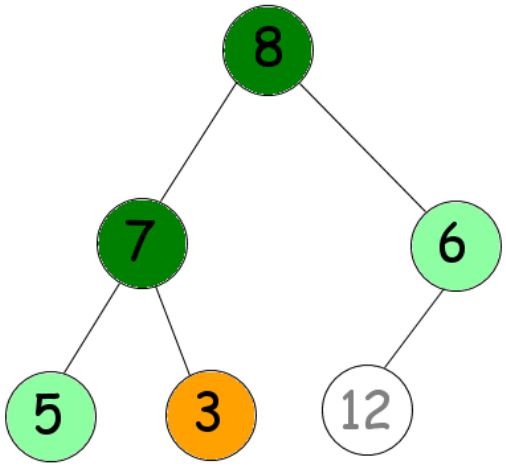


En effet. Mais rassurez-vous, il reste encore une étape, un peu plus complexe. Nous allons parcourir notre arbre en sens inverse encore une fois. Nous allons échanger le dernier nœud de notre arbre tamisé (le 3) avec le premier (le 12) qui est également le plus grand nombre de l'arbre.

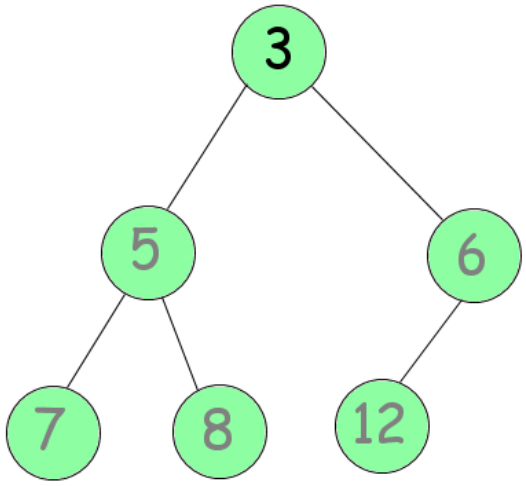


Échange du dernier et du premier nœud

Une fois cet échange effectué, nous allons tamiser notre nouvelle racine, en considérant que la dernière feuille ne fait plus partie de l'arbre (elle est correctement placée désormais). Ce qui devrait nous donner le résultat suivant (voir la figure suivante).



Cette manœuvre a ainsi pour but de placer sur la dernière feuille, le nombre le plus grand, tout en gardant un sous-arbre qui soit tamisé. Il faut ensuite répéter cette manœuvre avec l'avant-dernière feuille, puis l'avant-avant-dernière feuille… Jusqu'à arriver à la deuxième (et oui, inutile d'échanger la racine avec elle-même, soyons logique). Nous devrions ainsi obtenir le résultat suivant (voir la figure suivante).



Implémentation en java :

Pseudo-code :

```
function heapSort(a, count) is
     input:  an unordered array a of length count

     (first place a in max-heap order)
     heapify(a, count)

     end := count-1 //in languages with zero-based arrays the
children are 2*i+1 and 2*i+2
     while end > 0 do
          (swap the root(maximum value) of the heap with the last
element of the heap)
          swap(a[end], a[0])
          (decrease the size of the heap by one so that the previous
max value will
          stay in its proper placement)
          end := end - 1
          (put the heap back in max-heap order)
          siftDown(a, 0, end)


 function heapify(a, count) is
     (start is assigned the index in a of the last parent node)
     start := count / 2 - 1

     while start ≥ 0 do
          (sift down the node at index start to the proper place such
that all nodes below
           the start index are in heap order)
          siftDown(a, start, count-1)
          start := start - 1
```

```
     (after sifting down the root all nodes/elements are in heap
order)

 function siftDown(a, start, end) is
     input:  end represents the limit of how far down the heap
                    to sift.
     root := start

     while root * 2 + 1 ≤ end do    (While the root has at least one
child)
          child := root * 2 + 1      (root*2 + 1 points to the left
child)
          swap := root        (keeps track of child to swap with)
          (check if root is smaller than left child)
          if a[swap] < a[child]
              swap := child
          (check if right child exists, and if it's bigger than what
we're currently swapping with)
          if child+1 ≤ end and a[swap] < a[child+1]
              swap := child + 1
          (check if we need to swap at all)
          if swap != root
              swap(a[root], a[swap])
              root := swap          (repeat to continue sifting down
the child now)
          else
              return
```

Code :

```
public static long[] triTas(long [] tab) {
          tas(tab);
          return tab;

   }


   /**
    * Internal method for heapsort.
    * @param i the index of an item in the heap.
    * @return the index of the left child.
    */
   private static int leftChild( int i )
   {
       return 2 * i + 1;
   }

   /**
    * Internal method for heapsort that is used in
    * deleteMax and buildHeap.
    * @param tab an array of Comparable items.
    * @index i the position from which to percolate down.
    * @int n the logical size of the binary heap.
    */
```

```
   private static void percDown( long[] tab, int i, int n )
   {
       int child;
      long tmp;

       for( tmp = tab[ i ]; leftChild( i ) < n; i = child )
       {
           child = leftChild( i );
           if( child != n - 1 && (tab[child] < tab[child + 1 ] ) )
               child++;
           if( tmp<tab[ child ]  )
               tab[ i ] = tab[ child ];
           else
               break;
       }
       tab[ i ] = tmp;
   }


private static void tas(long[] tab){
      for( int i = tab.length / 2; i >= 0; i-- )  /* buildHeap */
        percDown( tab, i, tab.length );
    for( int i = tab.length - 1; i > 0; i-- )
    {
      swap( tab, 0, i );            /* deleteMax */
        percDown( tab, 0, i );
    }

    }
```