IENAC – Majeure SITA

Exercices de complexité

Diviser pour régner

2-d Tree

A 2-d tree (see figure 1) is a data structure used to organize a set of points in 2D such that they can be queried efficiently. The building of a 2-d tree over a set P of 2D-points proceeds as follows :

— Choose $p \in P$, remove it from P and start creating a new node containing p.

— Divide the plane in two with the **horizontal** line going through p : the points P_1 belonging to the lower half-plane will be recursively inserted in the left son and the others P_2 in the right one.

— During the next recursive call, a new point is chosen in the corresponding set of points but the half-plane is now divided along the **vertical** line : points belonging to the left half-plane will be inserted in the left son, and the others in the right one.

— Alternate horizontal and vertical divisions at each level until P becomes empty.

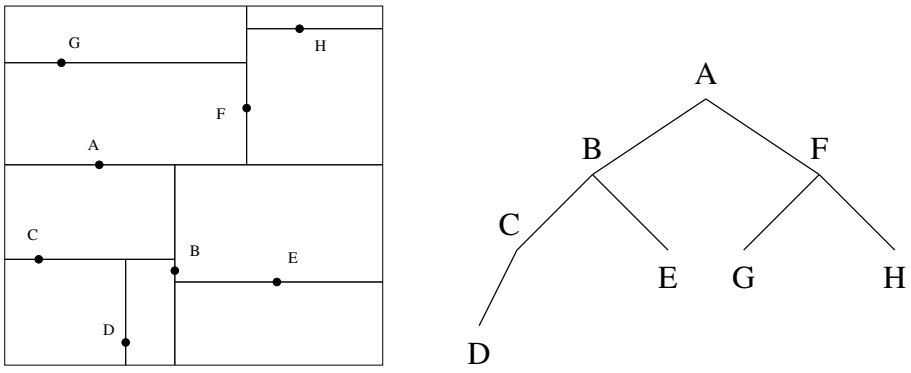


FIGURE 1 – A set of 2D points and its corresponding 2-d tree.

At each step, to divide the set of remaining points in two (almost) equal parts, p must be chosen as the median point among all x-coordinates (i.e. the point with the $\frac{n}{2}^{\text{th}}$ smallest x-coordinate) for vertical divisions, and as the median among y-coordinates for horizontal ones. To compute this step efficiently, we will **sort** the points of P **before starting to build the tree** : the points are sorted (in ascending order) along their x-coordinates in a sequence noted P_x and along their y-coordinates in sequence P_y.

1. During an horizontal step, what is the worst-case time complexity of finding the median point p and dividing P_y in two sorted subsequences of (almost) equal length ?
2. Let $p = (x, y)$, the median point of the preceding question. To be able to recursively call the algorithm (which uses P_y **and** P_x), P_x must also be parted in two sorted (with respect to x-coordinates) subsequences : the points that are below p and the points above respectively. What is the worst-case time complexity of this step ?

3. Write the recurrence relation on the worst-case time complexity of this recursive algorithm, without taking into account the preliminary sorting phase, then give its solution as a function of the number of points $n = |P|$.
4. Give worst-case time and space complexities of the overall algorithm (preliminary sortings included) as functions of n.

Maximum Subsequence Sum

Let $a = [a_1, a_2, \ldots, a_n]$ be an array of integers. We want to compute the maximum sum $\sum_{k=i}^{j} a_k$, $\forall i, j$ such that $1 \leq i \leq j \leq n$. The maximum sum is defined to be 0 if all the integers are negative.

For example, with the array $[1, -2, 1, 2, -1, 3, -2, 1]$, the maximum subsequence corresponds to $i = 3$ and $j = 6$ with the sum equal to $1 + 2 - 1 + 3 = 5$.

1. Write a naive algorithm to compute the value of the maximum subsequence sum, using `for` loops to compute each $\sum_{k=i}^{j} a_k, \forall i, j$ s.t. $1 \leq i \leq j \leq n$.
2. Compute the **time** and **space worst-case complexities** of this algorithm.
3. Write a "divide & conquer" algorithm to compute the maximum subsequence sum, using the following technique for a subarray indexed from i to j (with $1 \leq i \leq j \leq n$) :
   — divide the array in two halves from i to $mid = \frac{i+j}{2}$ and from $mid + 1$ to j (for $i = 1$ and $j = 8$, $mid = \frac{1+8}{2} = 4$ in the example) ;
   — recursively compute the maximum subsequence sum in the left subarray and in the right subarray ;
   — compute the solution for the whole array by considering three cases :
      — either the maximal subsequence is in the left subarray ($\sum_{k=3}^{4} a_k = 1 + 2 = 3$ in the example) ;
      — or the maximal subsequence is in the right subarray ($\sum_{k=6}^{6} a_k = 3$ in the example) ;
      — or the maximal subsequence crosses the middle of the array, which can be obtained by adding :
         — the maximum of the sums $\sum_{k=l}^{mid} a_k, \forall l \in [i, mid]$ in the left part, i.e. over subarrays indexed from mid down to i ($\sum_{k=3}^{4} a_k = 1 + 2 = 3$ in the example) ;
         — and the maximum of the sums $\sum_{k=mid+1}^{l} a_k, \forall l \in [mid+1, j]$ in the right part, i.e. over subarrays indexed from mid to i ($\sum_{k=5}^{6} a_k = -1 + 3 = 2$ in the example).

      which is $\sum_{k=3}^{6} a_k = \sum_{k=3}^{4} a_k + \sum_{k=5}^{6} a_k = 3 + 2 = 5$ in the example.

   Therefore, the maximum subsequence sum is 5 and crosses the middle of the two halves for the example.
4. Compute the **time** and **space worst-case complexities** of this algorithm.

Programmation dynamique

Pyramide de nombres

We want to compute the maximal sum for a path joining the top of a pyramid of numbers to its bottom, such as illustrated in figure 2. Each node of the pyramid is connected to its right and left children (one level below), except for the last level (bottom).

A pyramid of numbers will be represented by a triangular matrix w. The children of cell $w_{i,j}$ will therefore be $w_{i+1,j}$ and $w_{i+1,j+1}$.

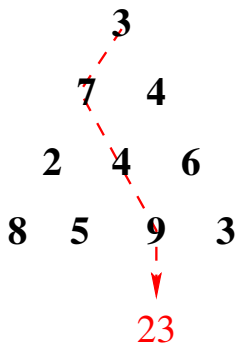


FIGURE 2 – A pyramid of numbers with height 4 and maximal path sum 23.

1. Give the number of different paths in a pyramid of numbers of height n and explain why.
2. Write a **recursive** algorithm that takes as parameter a triangular matrix representing a pyramid of numbers (and possibly its size) and computes its maximal path sum.
   Indication *Use a secondary function with parameters i and j that returns the maximal path sum from number $w_{i,j}$: this sum is equal to $w_{i,j}$ plus the greatest of the maximal path sums of its children. Start from the top.*
3. Give worst-case time **and** space complexities of this algorithm as functions of n.
4. Write an improved version of the previous algorithm by using **dynamic programming**.
   Indication *Matrix w can be directly used to store intermediary results. Start from the bottom.*
5. Give worst-case time **and** space complexities of this algorithm as functions of n.

Knapsack

Le **problème du sac à dos** (*Knapsack*) consiste à choisir un sous-ensemble parmi n objets, de telle manière que la somme des **poids** des objets choisis ne dépasse pas la **capacité** c du sac à dos et que la somme de leurs **valeurs** soit **maximale**. Si on note $w(x)$ le poids de l'objet x et $v(x)$ sa valeur (ces deux fonctions renvoient des valeurs **entières strictement positives**), il faut trouver un sous-ensemble $X' \subseteq X$ de l'ensemble des objets $X = \{x_1, \ldots, x_n\}$ tel que $\sum_{x \in X'} w(x) \leq c$ et de valeur totale $\sum_{x \in X'} v(x)$ maximale.

On peut résoudre ce problème NP-difficile grâce à la relation de récurrence sur $kp(i, j)$ définie comme la valeur totale maximale d'un sous-ensemble d'objets de $\{x_1, \ldots, x_i\}$ ne dépassant pas le poids total j :

$$\forall i \in [1, n],\ \forall j \in [1, c] \quad kp(i, j) = \begin{cases} kp(i-1, j) \text{ si } w(x_i) > j \\ \max(kp(i-1, j), kp(i-1, j - w(x_i)) + v(x_i)) \text{ sinon} \end{cases}$$

avec $\forall j, kp(0, j) = 0$ (l'ensemble vide a une valeur nulle) et $\forall i, kp(i, 0) = 0$ (un sac de poids nul a une valeur nulle). La solution sera ainsi donnée par le calcul de $kp(n, c)$.

1. Écrire une fonction **récursive directe** `knap(w, v, c)` qui calcule $kp(n, c)$ en prenant en paramètres la liste w des poids et la liste v des valeurs des objets ainsi que la capacité maximale c du sac à dos.

2. Calculer des **bornes inférieures** pour les **complexités temporelle et spatiale** en pire cas de cet algorithme.
   Indication : on pourra considérer que les poids des objets valent 1.
3. Améliorer l'efficacité de la fonction précédente en écrivant une fonction `knap_dyn(w, v, c)` qui utilise la technique de **programmation dynamique**.
4. Calculer les **complexités temporelle et spatiale** en pire cas de `knap_dyn`.

Examen de
Complexité

Durée : 1 h
Tous documents autorisés
4 novembre 2016

— Les réponses aux questions doivent être dûment justifiées.
— Les programmes demandés peuvent être écrits avec le langage de programmation de votre choix.
— Le barème est donné à titre indicatif.

Problème du sac à dos non borné

Le problème du sac à dos non borné (*Unbounded Knapsack*) est similaire au problème du sac à dos classique, mais on peut utiliser plusieurs exemplaires (une quantité non bornée) du même objet pour remplir le sac. Il s'agit donc de choisir une collection d'objets, issus d'un ensemble de n prototypes d'objets $X = \{x_1, \ldots, x_n\}$ (que l'on peut donc réutiliser autant de fois que l'on souhaite), de telle manière que la somme de leur **poids** ne dépasse pas la **capacité** c du sac à dos et que la somme de leurs **valeurs** soit **maximale**. On notera $w(x)$ le poids de l'objet x et $v(x)$ sa valeur (ces deux fonctions renvoient des valeurs **entières strictement positives**).

On peut résoudre ce problème NP-difficile grâce à la relation de récurrence sur $t(j)$ définie comme la valeur totale maximale possible pour une collection d'objets dont le poids total ne dépasse pas j :

$$\forall j > 1, \quad t(j) = \max_{\forall i \in [1,n] \text{ t.q. } w(x_i) \leq j} (t(j - w(x_i)) + v(x_i))$$

La solution sera ainsi donnée par le calcul de $t(c)$.

1. [6pt] Écrire une fonction **récursive directe** qui calcule $t(c)$ en prenant en paramètres les listes (ou tableaux) `w` de poids et `v` de valeurs, ainsi que la capacité maximale `c` du sac à dos.
2. [4pt] Estimer les **complexités temporelle et spatiale** en pire cas de cet algorithme récursif, en considérant que $\forall i, w(x_i) = 1$.
3. [6pt] Écrire une fonction avec les mêmes paramètres que la fonction précédente, mais qui utilise la technique de **programmation dynamique**.
4. [4pt] Calculer les **complexités temporelle et spatiale** en pire cas de l'algorithme de programmation dynamique.

Examen de

Complexité

Durée : 1 h
Tous documents autorisés

7 novembre 2017

— Les réponses aux questions doivent être dûment justifiées.
— Les programmes demandés peuvent être écrits avec le langage de programmation de votre choix.
— Le barème est donné à titre indicatif.

Rendu de monnaie

Le problème du *rendu de monnaie* consiste à rendre un montant fixé avec le minimum de pièces de monnaie pour un système de valeurs fixé (en disposant d'un nombre illimité de pièces de chaque valeur). Par exemple, si le système de valeurs est constitué de pièces de 1€, 2€ et 5€, le nombre minimal de pièces pour rendre 9€ est égal à 3 : 2 pièces de 2€ et 1 pièce de 5€.

Plus formellement, on notera :
— $v_i, \forall i \in [1, n]$ le système de valeurs considéré constitué de n différents types de pièce ;
— s le montant à rendre.

Soit $x_i, \forall i \in [1, n]$ le nombre de pièces de valeur v_i d'une solution, le problème consiste à minimiser :

$$\sum_{i=1}^{n} x_i$$

tout en respectant :

$$\sum_{i=1}^{n} x_i v_i = s$$

En notant $M(s)$ le nombre de pièces minimal pour rendre le montant s avec le système de n valeurs $v_i, \forall i \in [1, n]$, on peut résoudre ce problème grâce à la relation de récurrence suivante :

$$\forall k > 0, \quad M(k) = 1 + \min_{\forall i \in [1,n] \text{ t.q. } v_i \leq k} M(k - v_i)$$

En effet : pour rendre le montant s, il faut au moins une pièce, par exemple de valeur v_i ; en supposant que cette pièce fasse partie de la solution optimale, le nombre de pièces de celle-ci est égal à $1+M(s-v_i)$ en comptant 1 pour la pièce choisie et $M(s - v_i)$ pour ce qu'il reste à rendre. Il faut donc calculer le mininum de cette expression quelle que soit la pièce choisie parmi les n possibles.

1. [6pt] Écrire une fonction **récursive directe** qui calcule $M(s)$ en prenant en paramètres le tableau (ou la liste) des valeurs du système et le montant à rendre.
2. [4pt] Calculer les **complexités temporelle et spatiale** en pire cas de cet algorithme récursif en fonction de n et de s, en considérant que $\forall i, v_i = 1$.
3. [6pt] Écrire une fonction avec les mêmes paramètres que la fonction précédente, mais qui utilise la technique de **programmation dynamique** pour calculer $M(s)$.
4. [2pt] Calculer les **complexités temporelle et spatiale** en pire cas de l'algorithme de programmation dynamique.
5. [2pt] Exprimer la complexité temporelle de l'algorithme de programmation dynamique calculée à la question précédente en fonction de la taille k nécessaire pour encoder l'entier s en base 2. Il a été démontré que ce problème est NP-difficile ; vient-on de découvrir que finalement $P = NP$ (et gagner au passage les $1M du Clay Mathematics Institute) ?

TP Python – Structures de données et algorithmique 1

Objectifs :
- Implémentation d'un **algorithme récursif** de type « **diviser pour régner** ».
- Calculs de **complexité temporelle**.
- Comparaison de temps de calcul avec la version **itérative**.

Sous-séquence de somme maximale

Le but de ce TP est de déterminer la sous-séquence $\langle l_i, l_{i+1}, \dots, l_j \rangle$ dont la somme est maximale dans une séquence l d'entiers relatifs qui contient au moins un nombre strictement positif. Il s'agit donc de trouver les indices i et j qui permettent de calculer :

$$\max_{i \leq j} \sum_{k=i}^{j} l_k$$

1. On va d'abord écrire un algorithme **itératif** pour réaliser ce calcul :
   - on initialise la meilleure somme *best* à 0 et les indices correspondants à `None` ;
   - pour tout indice i de l :
     - on initialise la somme s_{ij} à 0 ;
     - pour tout indice $j \geq i$ de l :
       - on ajoute l_j à s_{ij} ;
       - si s_{ij} est strictement supérieure à *best*, on met à jour *best* et les indices correspondants ;
   - on renvoie *best* et les indices correspondants.

   Écrire une fonction `maxsubsum1` **non récursive** qui prend la liste en paramètre et renvoie le triplet `(best, i, j)`. On testera le code sur la liste `l = [1,-2,1,2,-1,3,-2,1]` de meilleure somme 5 correspondant aux indices 2 à 5 (inclus).
2. Quelle est la **complexité temporelle** de cet algorithme en fonction de n ?
3. On souhaite améliorer la complexité de cette recherche en utilisant un algorithme **récursif** de type « **diviser pour régner** » selon le principe suivant :
   - à chaque étape, on découpe la liste en deux parties (presque) égales :
     - soit la meilleure somme se trouve dans la sous-liste de gauche ;
     - soit elle est dans celle de droite ;
     - soit elle est à cheval entre les deux ; dans ce cas, on peut simplement la déterminer de la manière suivante :
       - on calcule la meilleure somme à gauche en partant du milieu et en ajoutant des éléments d'indices décroissant ;
       - on calcule la meilleure somme à droite en partant du milieu et en ajoutant des éléments d'indices croissant ;
       - la meilleure somme passant par le milieu sera donc la somme des deux ;

— on arrête la récursion quand la liste n'a plus qu'un seul élément (cas trivial).

Pour éviter de consommer trop de mémoire et de temps à fabriquer des nouvelles listes à chaque étape, on ne va pas réellement « découper » la liste mais passer en paramètre de la fonction récursive les indices i et j de la tranche de liste à considérer : ainsi, on effectuera les appels récursifs avec i et k pour la tranche de gauche et $k+1$ et j pour celle de droite, avec $k = \frac{i+j}{2}$. On pourra ainsi arrêter la récursion lorsque $i = j$.

Pour écrire cette algorithme, on définira une **fonction locale** (i.e. dans le corps de la fonction principale) qui prendra en paramètre les indices i et j de la tranche à considérer et renverra le triplet correspondant à la meilleure séquence entre i et j. L'appel initial sera donc réalisé sur la tranche correspondant à l'intégralité de la liste. Votre code aura donc la structure suivante :

```
def maxsubsum2(l):

    def maxsubsum2_rec(i, j):
        [...]
        return ...

    return maxsubsum2_rec(0, len(l)-1)
```

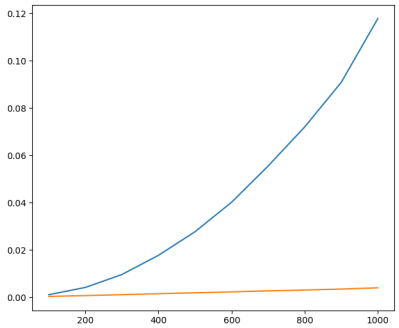
4. Écrivez en commentaires l'équation de récurrence de la complexité temporelle et la résoudre.
5. Pour vérifier les calculs de complexité, on va mesurer le temps de calcul des deux algorithmes pour des tailles de liste croissantes. Le module `time` permet d'utiliser la fonction `process_time` qui ne prend pas d'argument et renvoie le temps de processeur utilisé par le programme en secondes ; il suffit alors d'encadrer l'appel à l'algorithme par deux appels à `process_time`, puis d'en calculer la différence pour connaître son temps d'exécution.

   Pour générer des listes d'entiers aléatoires d'une taille donnée, on utilisera le module `random` qui fournit la fonction `randint(lb, ub)` renvoyant un entier compris entre `lb` et `ub` **inclus** :

   ```
   def genlist(n):
       return [random.randint(-n, n) for _ in range(n)]
   ```

   On peut ensuite tracer les courbes correspondantes avec le module `matplotlib.pyplot` :

   ```
   import matplotlib.pyplot as plt
   plt.plot(xs, ys1, xs, ys2)
   plt.show()
   ```

   avec `xs` la liste des abscisses et `ys1`, `ys2` les temps de calcul correspondants pour les deux algorithmes.
6. On peut remarquer que les deux algorithmes ne produisent pas en général la même sous-séquence maximale quand il y en a plusieurs de même somme. On souhaite à présent pouvoir déterminer la plus courte. Modifier le critère de comparaison optionnel `key` de la fonction `max` avec une **fonction anonyme** pour ajouter la longueur comme critère secondaire (à minimiser).

TP Python – Structures de données et algorithmique 3

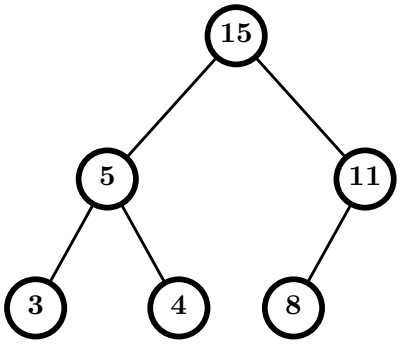
Objectifs :
- Implémentation par un **tas** de l'ADT **file de priorité**
- Implémentation d'un algorithme de **tri optimal** et **en place**

Remarque préliminaire : Tous les langages de programmation désignent par le terme « tableau » (*array*) un espace de mémoire contigu dont chaque case peut être indexé en temps constant. Sauf Python, dont les concepteurs ont choisi le terme « list ». Les tableaux mentionnés dans ce TP sont donc à interpréter comme des « listes » en Python.

Le but de ce TP est d'implémenter un tri de complexité temporelle en pire cas optimale, le **tri par tas** (*heapsort*). Un *tas* (*heap*) est l'une des implémentations possibles d'une *file de priorité* sous la forme d'un arbre binaire qui maintient toujours l'élément le plus petit (*min-heap*) ou le plus grand (*max-heap*) à sa racine.

Pour implémenter le tas, on va utiliser un codage de l'arbre binaire dans un tableau : pour un tableau indexé de 1 à n, on trouvera les enfants droit et gauche du nœud d'index i aux index $2 * i$ et $2 * i + 1$ ($2 * i + 1$ et $2 * i + 2$ pour un tableau indexé à partir de 0), et le parent d'un nœud d'index i sera donc situé à la case d'index $\frac{i}{2}$ ($\frac{i-1}{2}$ si l'index commence à 0).

Il est possible d'utiliser une telle implémentation d'un tas lorsqu'on connaît à l'avance le nombre (maximal) d'éléments à insérer dans le tas (ce qui est le cas pour le heapsort). Elle permet alors de réaliser le tri « en place », directement dans le tableau, sans avoir à gérer de référence vers les enfants (ou le parent) d'un nœud. Pour un tri par ordre *croissant*, on utilisera alors une *max-heap*, i.e. un tas dont le plus grand élément est à la racine, pour pouvoir les retirer un à un et les placer à la fin du tableau.



Nœud	15	5	11	3	4	8
Indice	0	1	2	3	4	5

FIGURE 1 – Un tas dans un tableau.

En plus de respecter la propriété de tas, i.e. que les éléments d'un nœud soient toujours plus grand (ou plus petit) que leurs enfants, ces tas doivent être des arbres binaires « parfaits », i.e. tous les niveaux exceptés le dernier doivent être totalement remplis (cf. figure 1), et si le dernier ne l'est pas, il doit être

rempli de gauche à droite. Il ne peut donc pas y avoir de case du tableau non utilisée entre deux cases utilisées, ou encore toutes les cases vides sont le plus à droite possible.

Pour pouvoir réaliser le heapsort en place, on considérera que le tas est situé dans le début du tableau, de l'index 0 jusqu'à l'index *lim* inclus, et que la suite du tableau constitue les éléments (non triés) à insérer dans le tas. À l'initialisation, on peut considérer que $lim = 0$ (pour un tableau indexé à partir de 0) et que le premier élément du tableau constitue un tas contenant un seul élément.

Insertion

Pour ajouter l'élément suivant d'index $lim + 1$ au tas, on va le faire remonter jusqu'à sa place. On compare l'élément à ajouter avec son parent : s'il est inférieur, on s'arrête ; sinon on les échange et on recommence récursivement avec le parent tant qu'on n'a pas atteint la racine (cf. figure 2).

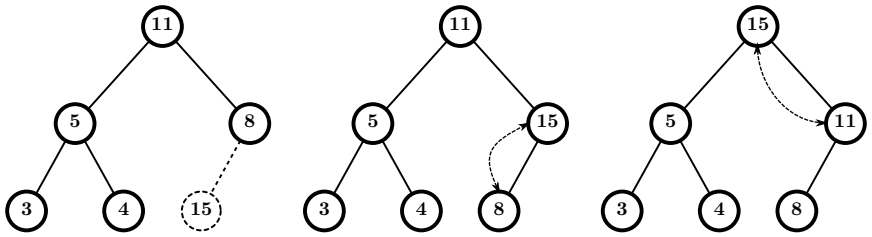


FIGURE 2 – Insertion.

On va ainsi insérer les éléments du tableau un à un dans le tas situé dans la première partie du tableau. Une fois tous les éléments insérés ($lim = n - 1$), on va retirer les éléments un à un du tas et les placer à la fin du tableau.

Suppression

On échange la racine du tas (premier élément du tableau) et le dernier élément du tas (d'index *lim*) ; l'ancienne racine sera donc correctement placée dans la fin (triée) du tableau. Puis on va faire descendre cet élément à sa place dans le tas. On compare l'élément avec ses deux enfants : s'il est supérieur, on s'arrête ; sinon on l'échange avec le plus grand de ses enfants et on recommence récursivement avec cet enfant (cf. figure 3).

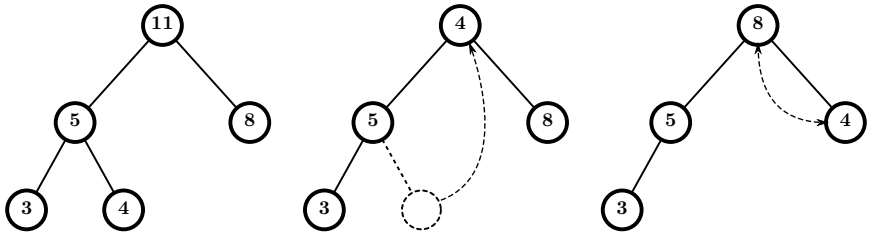


FIGURE 3 – Suppression.

1. Écrire la fonction `insert(l, i, cmp)` qui insère l'élément d'index `i` de la liste `l` dans le début du tableau considéré comme un tas, en utilisant la fonction de comparaison `cmp`.

   Cette fonction sera initialement passée en paramètre à la fonction de tri pour indiquer son ordre (croissant ou décroissant), puis la fonction de tri la transmettra à son tour à la fonction `insert`. Pour simplifier le raisonnement, on pourra considérer que `cmp` correspond à l'opérateur $<$ (ou plutôt à la fonction `operator.lt`), ce qui correspond à un tri dans l'ordre croissant. Le tas doit alors être une max-heap pour toujours insérer le *plus grand* élément en fin de tableau lors de la phase de suppression. On cherchera donc à faire remonter le plus grand élément vers la racine dans la fonction `insert`.

2. Écrire la fonction `heapify(l, cmp)` qui réalise l'insertion de tous les éléments de la liste `l` dans le tas.

3. Écrire la fonction `delete(l, lim, cmp)` qui supprime la racine du tas `l` occupant les cases indexées de 0 à `lim`. Le tas occupera donc une case de moins, i.e. jusqu'à `lim-1`.

4. Écrire la fonction `heapsort(l, cmp=operator.lt)` qui trie en place la liste `l` de taille `n`. On commencera par transformer la liste en tas puis on retirera ses éléments un à un en les plaçant à la fin du tableau et en maintenant la limite du tas à chaque étape.

5. Tester votre fonction de tri dans l'ordre croissant et décroissant à l'aide d'une liste aléatoire dont la taille sera passée en paramètre sur la ligne de commande :

   ```
   > python3 heapsort.py 15
   [138, 132, 38, 143, 52, 119, 135, 65, 119, 39, 7, 82, 33, 40, 90]
   [7, 33, 38, 39, 40, 52, 65, 82, 90, 119, 119, 132, 135, 138, 143]
   [143, 138, 135, 132, 119, 119, 90, 82, 65, 52, 40, 39, 38, 33, 7]
   ```

6. Calculer la complexité temporelle de l'algorithme. Le heapsort est-il stable ? Quelle est sa complexité sur un tableau déjà trié ?