Tas (Heap)
Hauteur d'un tas
Tas et file d'attente prioritaire
Insertion dans un tas
Retirer dans un tas
Implémentation dans un tableau
Fusion de 2 tas
Construction efficace, de bas-en-haut, d'un tas
Conclusions du module

# Tas
## (Heap)

Les deux stratégies pour implémenter une file d'attente prioritaire dans la section précédente soulignent un compromis intéressant. Lorsqu'on utilise une liste non triée, nous pouvons effectuer des insertions en temps dans O(1), mais trouver ou supprimer l'élément avec la clé minimum nécessite une boucle qui prend un temps dans O(n).

En revanche, si on utilise une liste triée, nous pouvons trivialement trouver ou supprimer l'élément avec la clé minimum en temps dans O(1), mais l'ajout d'un nouvel élément à la file d'attente nous coûte un temps dans O(n).

Dans cette section, nous verrons une structure plus efficace pour implémenter une file d'attente prioritaire en utilisant ce qu'on appelle un **tas binaire** (heap ; aussi *monceau* en français).

Cette structure de données va nous permettre d'effectuer à la fois des insertions et des suppressions en temps logarithmique, représentant une amélioration significative sur les implémentations basées sur une liste.

Le fondement pour réaliser cette amélioration provient de l'utilisation d'une structure d'arbre binaire nous permettant de faire un compromis pour garder les éléments de manière non parfaitement ni complètement triés.

# Tas

Un **tas** est un arbre binaire stockant des clés sur ses nœuds et satisfaisant les propriétés suivantes :

**contrainte/propriété relationnelle (ou d'ordre)** :

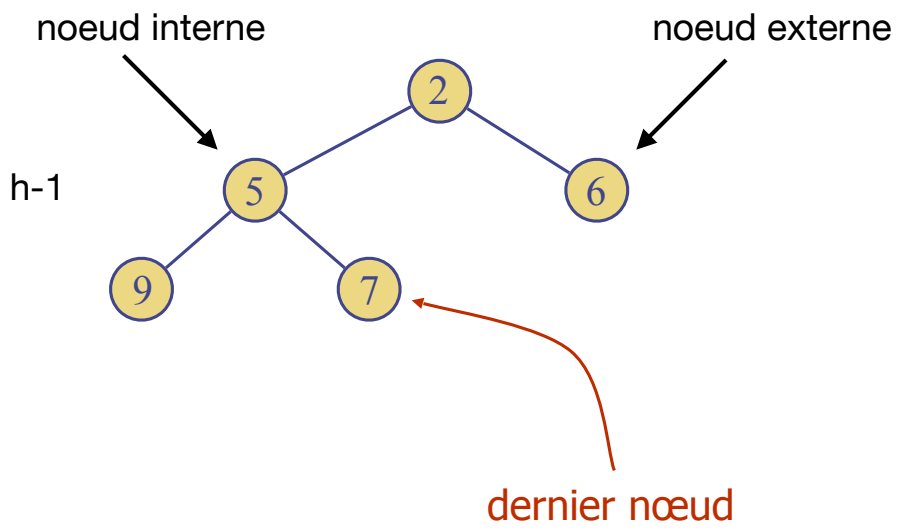
**Ordre-d'un-tas** : pour chaque nœud interne *v* autre que la racine, *clé( v ) >= clé( parent( v ) )*

**contrainte/propriété structurale** :

**Arbre binaire complet** : soit *h* la hauteur du tas

- pour $i = 0, \ldots, h - 1$, il y a $2^i$ nœuds de profondeur *i*
- à la profondeur $h - 1$, les nœuds internes sont à gauche des nœuds externes

Le dernier nœud d'un tas est le noeud le plus à droite de la profondeur maximale.



# Hauteur d'un tas

Théorème : Un tas stockant $n$ clés possède une hauteur dans $O(\log n)$

Preuve : (nous appliquons la propriété d'un arbre binaire complet)

Soit $h$ la hauteur d'un tas stockant $n$ clés

Comme il y a $2^i$ clés en profondeur $i = 0, ..., h - 1$ et entre 1 et $2^h$ clés en profondeur $h$, on a entre

$1 + 2 + 4 + ... + 2^{h-1} + 1$ (minimum) et

$1 + 2 + 4 + ... + 2^{h-1} + 2^h$ (maximum)

clés pour un arbre de hauteur $h$.

Comme $1 + 2 + 4 + ... + 2^{h-1} = 2^h - 1$, on a entre

$2^h - 1 + 1$ (minimum) et

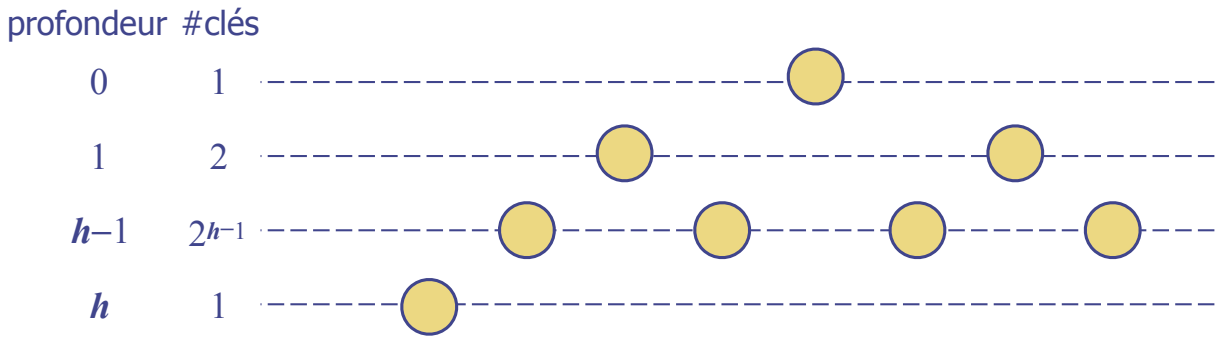
$2^h - 1 + 2^h$ (maximum) clés, soit

$n >= 2^h$ et $n <= 2^{h+1} - 1$.

On prend le log de chaque côté des inégalités :

$\log n >= h$ et $\log(n+1) - 1 <= h$ (exemple : pour h = 3, on aurait entre 8 et 15 noeuds)

Puisque $h$ est un entier, les 2 inégalités implique que $h = \lfloor \log n \rfloor$

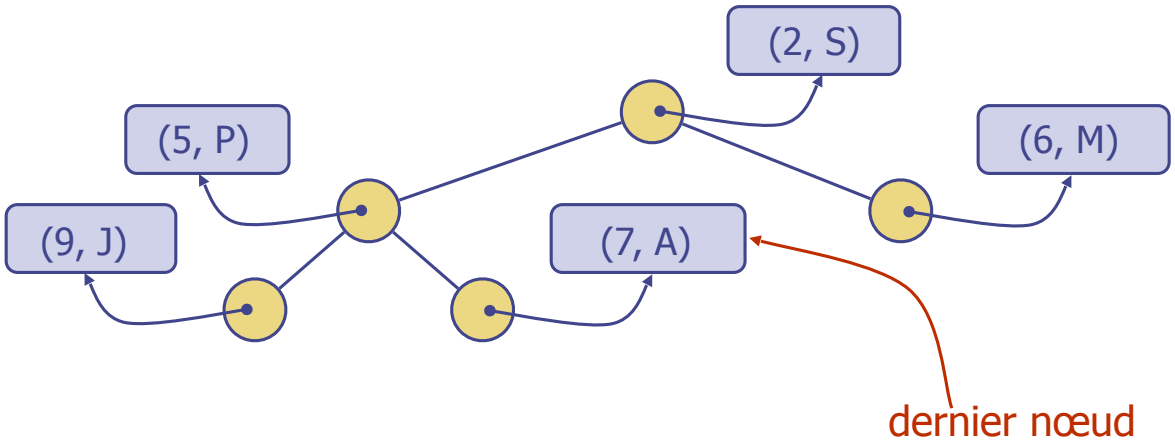


# Tas et file d'attente prioritaire

Nous pouvons utiliser un tas pour implémenter une file d'attente prioritaire

Nous stockons un élément (clé, valeur) dans chaque nœud interne

Nous gardons une trace sur la position du dernier noeud

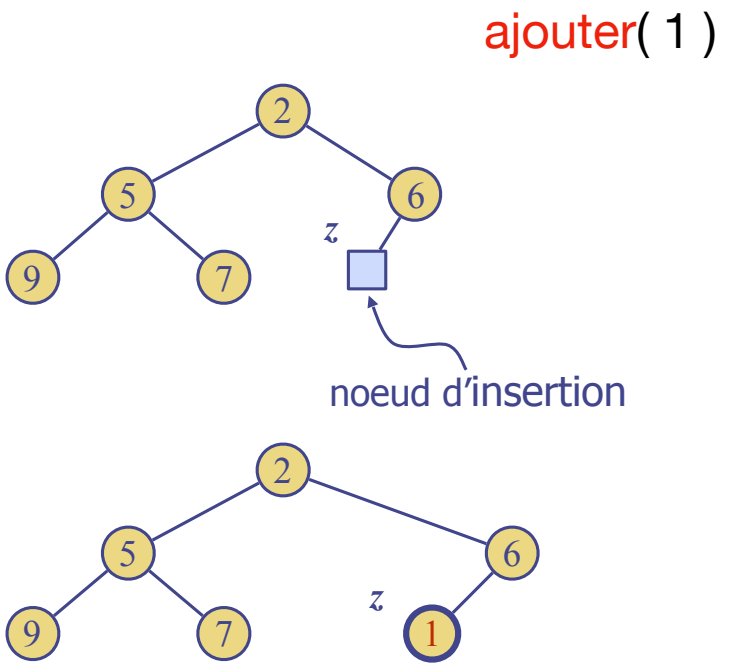


# Insertion dans un tas

La méthode **ajouter** de l'ADT d'une file d'attente prioritaire correspond à l'insertion d'une clé k dans le tas

L'algorithme d'insertion consiste en trois étapes :

- Trouver le noeud d'insertion z (le nouveau dernier noeud)
- Stocker k à z
- Restaurer la propriété de l'ordre du tas (abordée ci-dessous)



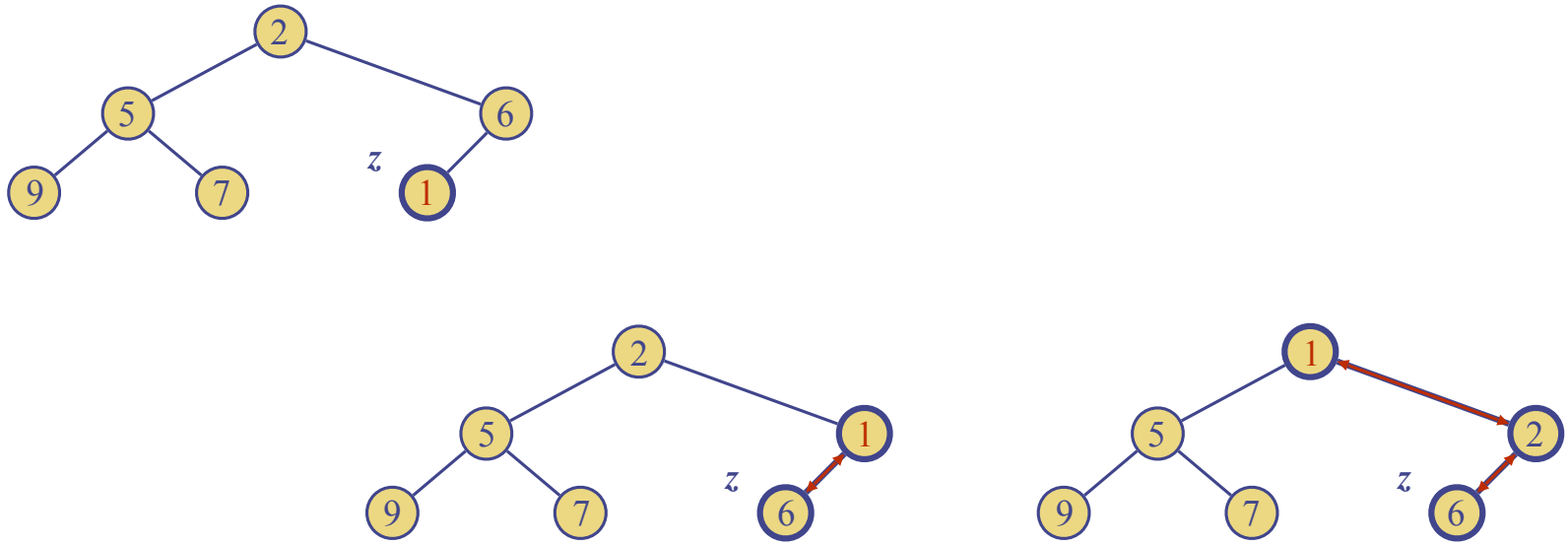
# Faire-monter (nager)

Après l'insertion d'une nouvelle clé *k*, la *propriété d'ordre* du tas peut être violée

L'algorithme **faire-monter** restaure la *propriété d'ordre* du tas en remplaçant *k* le long d'un chemin ascendant depuis le noeud d'insertion

**faire-monter** se termine lorsque la clé *k* atteint la racine ou un noeud dont le parent a une clé plus petite ou égale à *k*

Étant donné qu'un tas possède une hauteur dans O(*log n*), la montée s'effectue en temps dans O(*log n*)

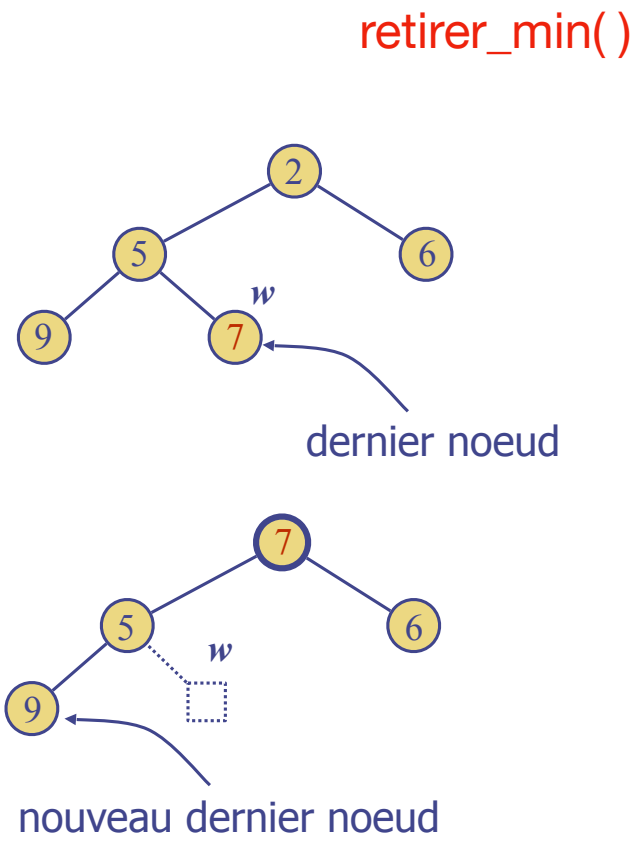


# Suppression dans un tas

La méthode **retire_min** de la file d'attente prioritaire correspond à la suppression de la clé à la racine du tas (là où se trouve l'élément de plus grande priorité)

L'algorithme de suppression consiste en trois étapes :

- Remplacer la clé à la racine par la clé du dernier nœud *w*
- Supprimer *w*
- Restaurer la propriété d'ordre (abordée ci-dessous)



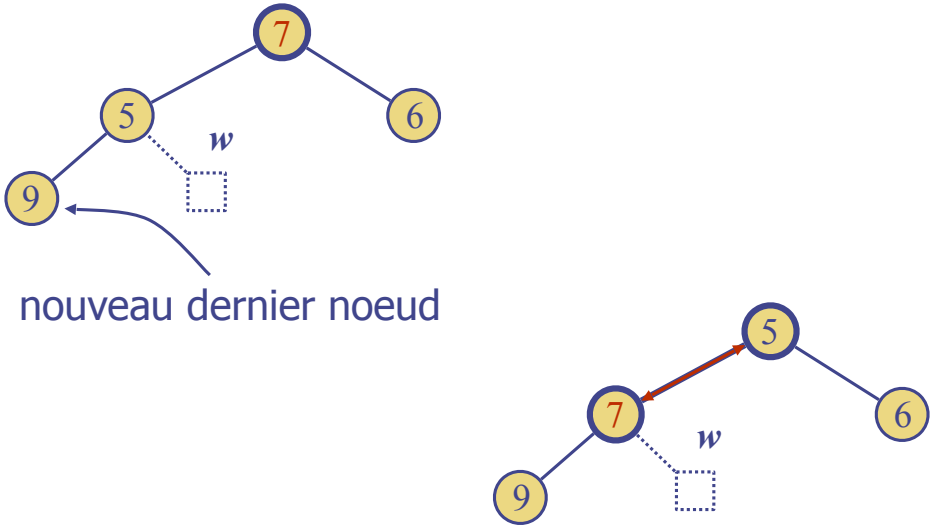
# Faire-descendre (couler)

Après avoir remplacé la clé racine par la clé *k* du dernier noeud, la *propriété d'ordre* du tas peut être violée

L'algorithme **faire-descendre** restaure la *propriété d'ordre* du tas en faisant couler la clé *k* le long d'un chemin qui descend depuis la racine

**faire-descendre** se termine lorsque la clé *k* atteint une feuille ou un noeud dont les enfants ont des clés supérieures ou égales à *k*

Comme un tas possède une hauteur dans O(*log n*), faire-descendre s'exécute en temps dans O(*log n*)



# Tri-par-tas (heapsort)

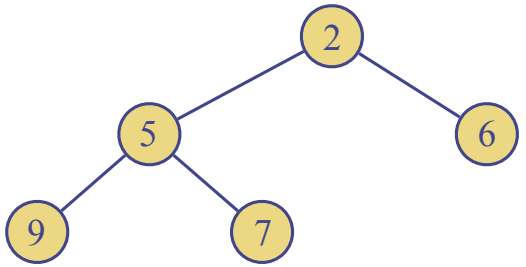
Considérez une file d'attente prioritaire avec $n$ éléments implémentée au moyen d'un tas

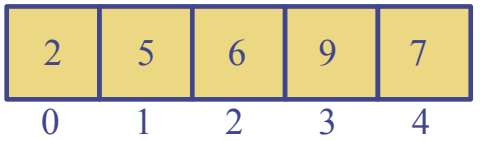
- l'espace utilisé est dans O($n$)
- les méthodes **ajouter** et **retirer_min** prennent un temps dans O(log $n$)
- les méthodes **len**, **est_vide** et **min** prennent un temps dans O(1)

- En utilisant une file d'attente prioritaire basée sur le tas, nous pouvons trier une séquence de $n$ éléments en temps dans O($n$ log $n$)
- L'algorithme résultant est appelé tri-par-tas (heapsort)
- Le tri-par-tas est beaucoup plus rapide que les algorithmes de tri quadratiques, tels que le tri par insertion et le tri par sélection.

# Implémentation d'un tas avec un tableau

- Nous pouvons représenter un tas avec *n* clés au moyen d'un tableau de longueur *n*
- Pour le nœud au rang i
  - l'enfant à gauche est à l'index *2i + 1*
  - l'enfant à droite est à l'index *2i + 2*
- Les liens entre les nœuds ne sont pas explicitement stockés
- L'opération **ajouter** correspond à l'insertion à l'index *n*
- L'opération **retirer_min** correspond à deleter à l'index *n-1*
- Rend le tri-par-tas en-place



```
# utilise la classe de base PriorityQueue
from PriorityQueue import PriorityQueue

# ADT HeapPriorityQueue
class HeapPriorityQueue( PriorityQueue ):

    # implémentation avec un tas implanté avec une list Python
    # construction avec possiblement une séquence de tuples (k,v)
    def __init__( self, contents = () ):
        self._Q = [self._Item( k, v ) for k,v in contents]
        if len( self._Q ) > 1:
            self._heapify()

    # taille avec len
    def __len__( self ):
        return len( self._Q )

    # accès direct à un élément
    def __getitem__( self, i ):
        return self._Q[i]

    # vide si len == 0
    def is_empty( self ):
        return len( self ) == 0

    # parent du noeud d'index j
    def _parent( self, j ):
        return (j-1) // 2

    # enfant de gauche du noeud d'index j
    def _left( self, j ):
        return 2*j + 1

    # enfant de droite du noeud d'index j
    def _right( self, j ):
        return 2*j + 2
```

```
# noeud d'index j possède un enfant gauche si l'index de cet enfant
# est à l'intérieur du tas
def _has_left( self, j ):
    return self._left( j ) < len( self )

# noeud d'index j possède un enfant droit si l'index de cet enfant
# est à l'intérieur du tas
def _has_right( self, j ):
    return self._right( j ) < len( self )

# retourne l'élément de plus grande priorité
# stocké à la racine du tas
def min( self ):
    if self.is_empty():
        return None

    # min est à la racine
    return self._Q[0]

# échange les contenus des noeuds i et j
def _swap( self, i, j ):
    tmp = self._Q[i]
    self._Q[i] = self._Q[j]
    self._Q[j] = tmp

# version récursive de swim
def _recswim( self, j ):
    # on prend l'index du parent
    parent = self._parent( j )
    # si parent dans le tas (j n'est pas l'index de la racine)
    # si la clé au noeud d'index j est plus prioritaire
    # alors on échange les contenus de l'enfant avec son parent
    # et on appelle récursivement sur le parent
    # récursivité de queue, donc version itérative triviale
    if j > 0 and self._Q[j] < self._Q[parent]:
        self._swap( j, parent )
        self._swim( parent )
```

```
# swim itératif
def _swim( self, j ):
    # tantque j n'est pas la racine ou
    # qu'un parent est moins prioritaire
    while j > 0:
        # on accède à l'index du parent
        parent = self._parent( j )
        # on échange enfant et parent si nécessaire
        if self._Q[j] < self._Q[parent]:
            self._swap( j, parent )
            # on continue avec le parent
            j = parent
        # sinon, on met j à 0 pour sortir du while
        else:
            j = 0

# ajoute et retourne l'élément x de clé k, O(log n)
def add( self, k, x ):
    item = self._Item( k, x )
    # on ajoute à la fin de la liste
    self._Q.append( item )
    # on fait nager le nouvel élément, O(log n)
    self._swim( len(self)-1 )
    # retourne le nouvel élément
    return item
```

```
# version récursive de sink
def _sink( self, j ):
    # on prend les indices des enfants gauche et droit
    # pour déterminer le plus petit des 2
    if self._has_left( j ):
        left = self._left( j )
        small_child = left
        if self._has_right( j ):
            right = self._right( j )
            if self._Q[right] < self._Q[left]:
                small_child = right
        # si le plus petit des enfants est plus prioritaire
        # on échange et on appelle récursivement sur l'enfant
        if self._Q[small_child] < self._Q[j]:
            self._swap( j, small_child )
            self._sink( small_child )

# suppression de l'élément de plus grande priorité, O(log n)
def remove_min( self ):
    if self.is_empty():
        return None
    # il se trouve à la racine
    the_min = self._Q[0]

    # on déplace le dernier élément du tas à la racine
    self._Q[0] = self._Q[len(self)-1]
    # on détruit le dernier élément
    del self._Q[len(self)-1]

    # si c'était le seul élément, on le retourne et c'est fini
    if self.is_empty():
        return the_min

    # sinon, on coule la nouvelle racine, O(log n)
    self._sink( 0 )

    # retourne le min
    return the_min
```

# Construire un tas en temps dans O(*n*)

On a vu que le constructeur d'un tas prend optionnellement une liste d'éléments. Quel est le coût de construire un tas de *n* éléments ?

```
# construit un heap en O(n)
def _heapify( self ):
    # on débute au parent de la dernière feuille
    # le dernier noeud est nécessairement la dernière feuille !
    start = self._parent( len( self ) - 1 )

    # on coule du parent de la dernière feuille jusqu'à la racine
    for j in range( start, -1, -1 ):
        self._sink( j )
```

Si on insère les *n* éléments dans le tas, un à un, on le construit en O(*n* log *n*), soit lancer **nager**, qui s'effectue en O(log *n*), sur les *n* éléments.

On peut faire mieux ! Si on insère les *n* éléments dans un ordre quelconque, on peut ensuite **couler** les éléments des noeuds internes jusqu'à la racine.

Rappel du constructeur:

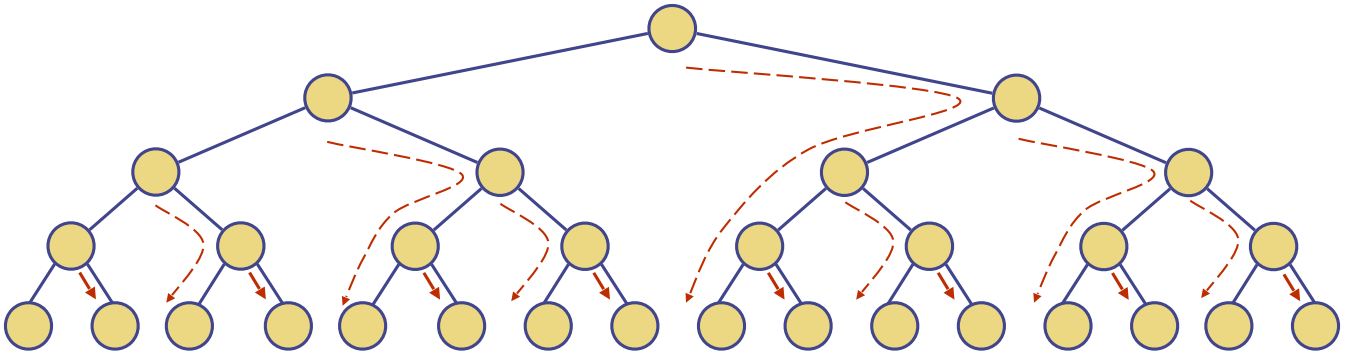
```
# implémentation avec un tas implanté avec une list Python
# construction avec possiblement une séquence de tuples (k,v)
def __init__( self, contents = () ):
    self._Q = [self._Item( k, v ) for k,v in contents]
    if len( self._Q ) > 1:
        self._heapify()
```

# Analyse de la construction du tas en temps dans O(*n*)

Nous visualisons le pire cas d'un **couler** pour chaque nœud interne avec un chemin qui va d'abord à droite et ensuite à plusieurs reprises à gauche jusqu'à la fin du tas (*n.b.* ce chemin peut différer du chemin réel lors de la construction)

Comme ***chaque nœud est traversé par au plus deux chemins***, le nombre total de visites est dans O(*n*)

Cette construction (par-le-bas) est plus rapide que de faire *n* insertions successives et accélère la première phase du tri-par-tas (heapsort).



# Conclusions du module

- On a vu l'ADT pour des **files d'attente prioritaires**
- On a vu les implémentations qui utilisent des séquences (listes triée et non triée) et que leurs complexités associées en dépendent. Dans le cas de la **liste non triée**, l'insertion est efficace mais chercher le min ne l'est pas, alors qu'avec la **liste triée** c'est l'inverse.
- On a vu qu'on peut utiliser une file d'attente prioritaire pour trier un ensemble d'éléments comparables. Avec une liste non triée, on a un **tri par sélection** et avec une liste triée on a un **tri par insertion**.
- On a vu qu'on pouvait utiliser qu'un seul tableau pour **trier en-place**
- On a vu que pour obtenir un temps raisonable (dans O(log *n*)) pour les opérations **insérer** et de **retirer_min**, la **structure en tas** est une solution
- On a vu que les opérations déterminantes de mise à jour d'un tas, **nager** et **couler**, prennent des temps dans O(log *n*), correspondant à la hauteur d'un tas contenant *n* clés
- On a vu que trier avec une file d'attente prioritaire implémentée avec un tas prend un temps dans *O(n log n)*, un gain considérable sur les tris par sélection et insertion
- On a vu comment implémenter la structure en tas dans un tableau
- On a vu qu'on pouvait construire un tas en temps dans *O( n )*.