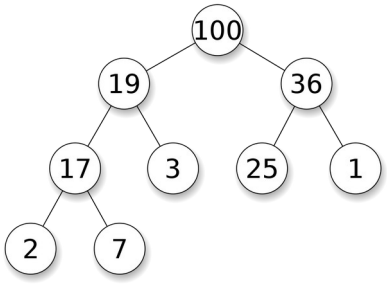
Fiche résumé nº 6 – Tas (et applications)

1. Structure de données Tas (*Heap* en anglais)

Un tas est une SDD reposant sur un **arbre binaire complet à gauche** (déjà vu) et **ordonné**.

Un **arbre ordonné** est un arbre tel que quel que soit le nœud de l'arbre, sa valeur est supérieure ou égale à celle de ses fils (selon la relation d'ordre choisie).



Par conséquent :

- la valeur maximale est à la racine
- la distance de la racine à une feuille est toujours la hauteur h de l'arbre ou bien $h-1$ (car l'arbre est complet à gauche).

N.B. On peut évidemment utiliser n'importe quelle relation d'ordre pour comparer les nœuds, et donc en particulier prendre l'ordre inverse qui reviendrait dans la définition à avoir le nœud père inférieur ou égal à chacun de ses fils.

Fonctions du TAD Tas :

```
sorte Tas[E]
utilise Booléen
fonctions
    new :         ⟶ Tas[E]
    estVide :     Tas[E]    ⟶  Booléen
    racine :      Tas[E]    ⇸  E
    insérer :     Tas[E]×E  ⟶  Tas[E]
    supprimer :   Tas[E]    ⟶  Tas[E]

préconditions
    ∀t de type Tas[E]
    pré(racine(t)) = non estVide(t)
```

La caractéristique de ces opérations sur le tas est qu'elles parcourront au pire uniquement une branche.

N.B. Dans ce qui suit, elles sont écrites en considérant des tableaux démarrant à l'indice 1, ce qui rend les fonctions d'accès aux fils et au père plus directement lisibles.

a) Insérer dans le tas

```
# Insérer la valeur v dans le tas a de taille n
algorithme insérer(v : entier)
    # a[1 … n] est un Tas
    n ← n+1
    a[n] ← v
    # Maintenant a[1 … n] est un Arbre binaire complet à gauche, mais plus un Tas
    # Et a[1 … n-1] est un Tas
    tamiserVersLeHaut(n)
    # Maintenant a[1 … n] est redevenu un Tas
fin

algorithme tamiserVersLeHaut(k : entier)      # k = position de départ
    v : entier ← a[k]
    # On a sauvegardé la dernière valeur ajoutée (pas à sa place)
    # Puis on descend successivement les pères d'un niveau pour arriver à la bonne place pour v
    tant-que k>1 et a[k/2]<v faire
        a[k] ← a[k/2]
        k ← k/2
    fin
    a[k] ← v      # Place la valeur en bonne place
fin
```

b) Supprimer le sommet du tas

```
# Supprimer la valeur maximum du tas a de taille n
algorithme supprimer
    # a[1 … n] est un Tas
    a[1] ← a[n]
    n ← n-1
    # Maintenant a[1 … n] est un Arbre binaire complet à gauche, mais plus un Tas
    tamiserVersLeBas(1)
    # Maintenant a[1 … n] est redevenu un Tas
fin

algorithme tamiserVersLeBas(k : entier)      # k = position de départ
    v : entier ← a[k]
    # On a sauvegardé la dernière valeur ajoutée (pas à sa place)
    # Puis on remonte successivement le plus grand des fils d'un niveau pour arriver à la bonne place pour v
    fini : booléen ← FAUX
    tant-que k<(n/2) et non(fini) faire
        # On détermine l'indice du plus grand des 2 fils
        fils : entier ← k*2
        si fils < n et a[fils] < a[fils+1] alors fils ← fils+1 fin
        # On compare v à ce fils le plus grand
        si a[fils] ≤ v alors fini ← VRAI
                       sinon a[k] ← a[fils]; k ← fils
        fin
    fin
    a[k] ← v      # Place la valeur en bonne place
fin
```

c) Action supplémentaire : échanger

Souvent les mises en œuvre de tas rajoutent une action supplémentaire « échanger » qui combine l'effet des 2 opérations précédentes :

- ajouter un élément au tas
- supprimer le plus grand élément du tas, et le retourner en résultat

L'intérêt est de réaliser ces 2 actions sans engendrer 2 tamisages mais un seul.

```
# Récupérer la valeur maximum du tas a de taille n en échange de l'insertion de la valeur v
algorithme échanger(v : entier)
    # a[1 … n] est un Tas
    suppr : entier ← a[1]
    a[1] ← v
    # Maintenant a[1 … n] est un Arbre binaire complet à gauche, mais plus un Tas
    tamiserVersLeBas(1)
    # Maintenant a[1 … n] est redevenu un Tas
    retourner suppr
fin
```

d) La complexité de ces 3 opérations est en $\mathcal{O}(\log_2 n)$, où n est le nombre d'éléments dans le tas.
En effet :

- le nombre d'éléments de l'arbre sur k niveaux est au maximum $n = 1+2+2^2+\ldots+2^{k-1} = 2^k-1$
- une opération parcourant une branche effectue au pire k étapes : donc en $\mathcal{O}(k)$, c'est-à-dire ici en $\mathcal{O}(\log_2 n)$.

2. Application : tri par tas (*Heap Sort*)

L'idée astucieuse de cet algorithme de tri d'un tableau consiste à mettre en place le tas directement dans le tableau des valeurs à trier (il y a partage du même espace entre le tas et les valeurs à trier).
Il effectue tout simplement 2 passes :

1. transformer le tableau initial en tas (dans le tableau lui-même) en partant du début
2. vider le tas, dans le tableau à trier, qu'on remplit en partant de la fin

=> On part donc d'un tableau non trié qui se transforme en tas puis est redevient tableau, mais trié.

```
algorithme triParTas(a : tableau[1…N] de valeurs)
    n : entier ← 0      # Taille du tas partageant le même tableau a
    pour i de 1 à N faire
        insérer(a[i])
    fin
    # Maintenant a[1 … n] est un devenu un Tas
    pour i de N à 1 faire
        max : entier ← racine()
        supprimer()
        a[i] ← max
    fin
    # Maintenant a[1 … n] est redevenu un simple tableau, mais trié !!!
fin
```

À partir de la complexité des opérations sur le tas, on déduit aisément que la complexité de cet algorithme est au pire $2N.\log_2 n$ et comme n est toujours $\leqslant N$, cette complexité est en $\mathcal{O}(N.\log_2 N)$.

N.B. Souvent, la mise en œuvre de l'opération supprimer() est étendue pour combiner 2 actions : supprimer l'élément racine du tas et aussi le retourner en résultat. Dans ce cas, l'algorithme de tri par tas se simplifie car au lieu des 3 instructions ci-dessus, le corps de la 2e boucle se résume aussi à une seule instruction comme dans la première boucle :

```
a[i] ← supprimer()
```

3. Deuxième application : file de priorité

Une **file de priorité** est une file (file d'attente) où les éléments ont un niveau de priorité qui conditionne leur sortie (ce n'est donc plus en mode FIFO).

Fonctions minimales du TAD FilePriorité :

```
sorte FilePriorité[E]
utilise Booléen
fonctions
    new :          ⟶ FilePriorité[E]
    estVide :      FilePriorité[E]     ⟶   Booléen
    trouverMax :   FilePriorité[E]     ⟶/  E
    insérer :      FilePriorité[E]×E   ⟶   FilePriorité[E]
    retirerMax :   FilePriorité[E]     ⟶   FilePriorité[E]

préconditions
    ∀𝒻 de type FilePriorité[E]
    pré(trouverMax(𝒻)) = non estVide(𝒻)
```

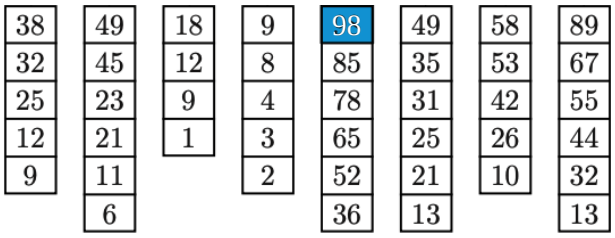
Complexités de ces opérations selon le choix interne de mise en œuvre :

Mise en œuvre	Trouver max	Insérer	Retirer max
Tableau non ordonné	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Tableau ordonné	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Liste chaînée ordonnée	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Tas	$\mathcal{O}(1)$	$\mathcal{O}(\log_2 n)$	$\mathcal{O}(\log_2 n)$

Comme une telle SDD FilePriorité nécessitera d'insérer et retirer des valeurs, c'est la version en tas qui est le meilleur compromis pour réaliser une file de priorité efficace.

4. Troisième application : arbre de sélection

Une autre application voisine est la fusion de plusieurs listes ordonnées (listes de priorité) pour en interclasser les valeurs : elle est réalisée en version simplifiée avec 2 listes dans le tri par fusion, mais elle est beaucoup plus subtile avec k listes à fusionner.



Algorithme naïf de fusion des listes

On réitère n fois (n étant le nombre de valeurs à interclasser) :

- comparer chacun des éléments de tête des listes pour déterminer le max de ces k valeurs : k-1 comparaisons, donc en $\mathcal{O}(k)$
- insérer cet élément dans la liste ordonnée du résultat : en $\mathcal{O}(1)$[1]
- supprimer cet élément de sa liste : en $\mathcal{O}(1)$

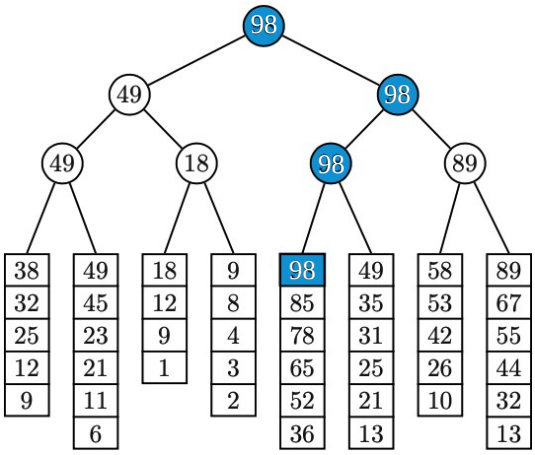
La complexité est donc proportionnelle à $n(k+2)$, c'est-à-dire en $\mathcal{O}(n.k)$.
On constate donc que l'efficacité dépend essentiellement des k-1 comparaisons à chacune des n itérations pour déterminer le max de toutes les têtes de listes. Or d'une itération à l'autre, seulement une valeur a changé : on voudrait ne pas refaire toutes comparaisons.

Comment déterminer ce max en temps plus court ? Voire en temps constant ?
=> utiliser un tas au-dessus des listes, car il mémorisera partiellement l'ordre :

1 Si si ! Contrairement au tableau précédent, dans ce cas l'insertion dans la liste est en $\mathcal{O}(1)$ et non en $\mathcal{O}(n)$.

- en quelques sortes, son organisation mémorise les comparaisons déjà connues
- et il permet de retirer le max en $\mathcal{O}(1)$ car il est au sommet

Ainsi, un **arbre de sélection** est un arbre complet à gauche structuré
- sous forme de tas (on parle d'arbre tassé)
- avec ses k feuilles reliées aux k listes ordonnées dans lesquelles sélectionner



La hauteur de cet arbre est donc $\log_2(k)$.
La valeur associée à chacun de ses nœuds internes est la plus grande de celles de ses fils.
La valeur de ses feuilles est la valeur en tête de la liste associée.

N.B. Alors l'élément prioritaire (de plus grande valeur) parmi tous les éléments est :
- au sommet du tas
- et répété encore $\log_2 k$ fois dans le tas le long d'une branche

Algorithme efficace de fusion des listes avec un arbre de sélection.

On réitère n fois :
- déterminer le plus grand élément : en $\mathcal{O}(1)$ dans le tas
- insérer cet élément dans la liste ordonnée du résultat : en $\mathcal{O}(1)$ dans la liste
- retrouver la liste concernée où l'enlever, en descendant une branche : en $\mathcal{O}(\log_2 k)$
- supprimer cette valeur de la liste concernée : en $\mathcal{O}(1)$ dans la liste
- tamiser le tas pour remonter le nouveau max en remontant le long de cette branche : en $\mathcal{O}(\log_2 k)$

En comptant l'étape préalable pour fabriquer le tas de k feuilles au début, la complexité totale est donc proportionnelle à $k + n.\log_2 k$: c'est-à-dire en $\mathcal{O}(n.\log_2 k)$ puisque ici $n \geqslant k$.

N.B. Il n'y a aucun intérêt ici à structurer les listes ordonnées elle-même en « files de priorités dans des tas » car :
1. concernant la liste destination, on insère successivement les valeurs dans l'ordre : $\mathcal{O}(1)$
2. concernant les listes sources, on ne fait que retirer : $\mathcal{O}(1)$