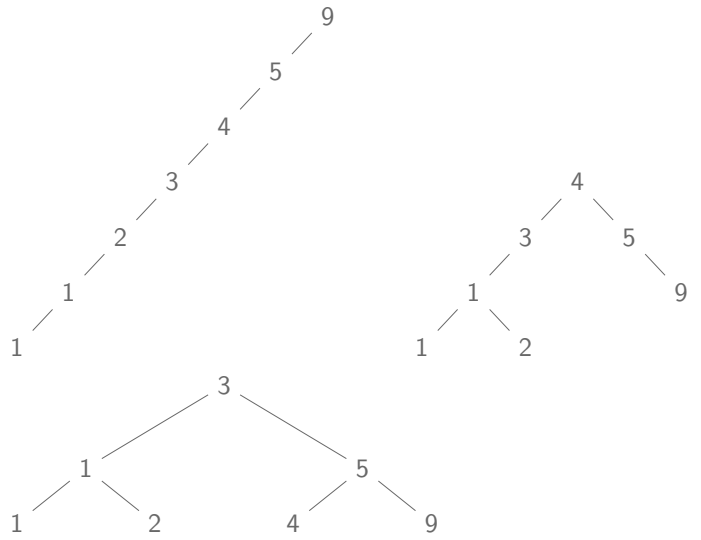
Exemple



Est-ce un arbre binaire de recherche ?

Require: un arbre binaire a
Ensure: Vrai si a est un ABR, Faux sinon
  if a est vide **then**
    return Vrai
  end if
  if le sous-arbre gauche de a n'est pas un ABR **then**
    return Faux
  end if
  if le sous-arbre droit de a n'est pas un ABR **then**
    return Faux
  end if
  if la racine de a n'est pas supérieure à tous les éléments du sous-arbre gauche **then**
    return Faux
  end if
  if la racine de a n'est pas inférieure à tous les éléments du sous-arbre droit **then**
    return Faux
  end if
  return Vrai

Propriétés d'un ABR

- la plus petite étiquette se trouve dans le nœud le plus à gauche de l'arbre
- la plus grande étiquette se trouve dans le nœud le plus à droite de l'arbre

La recherche de la plus grande ou de la plus petite étiquette se fait en $\mathcal{O}(h)$. Si l'arbre est équilibré alors c'est en $\Theta(\log(n))$.

- le parcours infixé d'un ABR aboutit à une liste croissante des étiquettes stockées aux nœuds de l'ABR.

? Question

Où se trouve l'étiquette médiane dans un ABR ?

Recherche dans un ABR

```
procedure RECHERCHER(e, a)
Ensure: un sous-arbre de a dont la racine est étiquetée par e s'il en existe,
  l'arbre vide sinon
    if a est vide then
        return Δ
    else
        r ← RACINE(a)
        if e = r then
            return a
        else
            if e ≤ r then
                return RECHERCHER(e,GAUCHE(a))
            else
                return RECHERCHER(e,DROIT(a))
            end if
        end if
    end if
end procedure
```


Recherche des étiquettes extrêmes

[Rappel] Soit a un ABR non vide.

1. la plus petite étiquette des nœuds de a est dans le nœud le plus à gauche de a.
2. la plus grande étiquette des nœuds de a est dans le nœud le plus à droite de a.

```
procedure ETIQUETTEMAX(a)
    if le sous-arbre droit de a est vide then
        return RACINE(a)
    else
        return ETIQUETTEMAX(FILSDROIT(a))
    end if
end procedure
```


Insertion dans un ABR

Pour insérer dans un ABR il suffit de faire une recherche et d'insérer à l'endroit où la recherche infructueuse nous a amené.

```
procedure INSERER(e, a)
    if a est vide then
        return CREER(e, Δ, Δ)
    else
        soient r = racine(a), g = FILSGAUCHE(a) et d = FILSDROIT(a)
        if e ≤ r then
            a_g ← INSERER(e,g)
            remplacer le fils gauche de a par a_g
            return a
        else
            a_d ← INSERER(e,d)
            remplacer le fils droit de a par a_d
            return a
        end if
    end if
end procedure
```


Construction d'un ABR

```
procedure CONSTRUIRE(l)
    if l est vide then
        return Δ
    else
        soit x la tête de l et l' le reste de l
        a_r ← CONSTRUIRE(l')
        return INSERER(x,a_r)
    end if
end procedure
```

Faisons le point

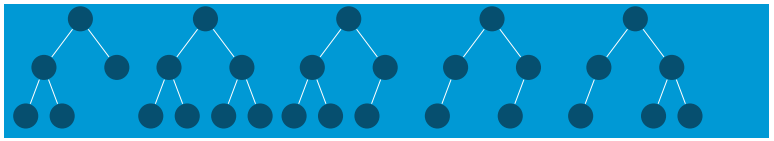
- les arbres sont des structures de données récursives
- les nœuds portent des étiquettes et possèdent une liste de fils ordonnés
- la topologie des arbres n'est pas unique pour une liste d'étiquettes

- les arbres binaires de recherche ajoutent une contrainte sur la position des étiquettes dans l'arbre
- cependant, la topolgie des ABR n'est pas fixée pour une liste d'étiquettes, mais elle est fixée pour un ordre d'insertion donné
- les ABR équilibrés permettent une recherche efficace en $\mathcal{O}(\log(n))$

Une nouvelle structure de données : le tas

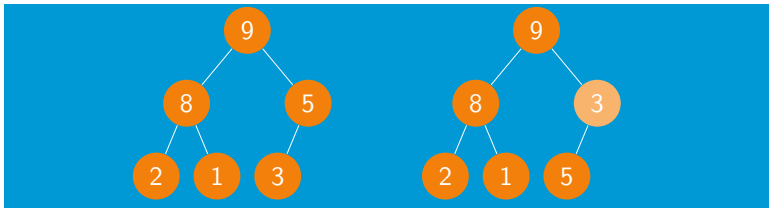
Définitions complémentaires sur les arbres binaires

- un arbre binaire complet est un arbre binaire dont tous les nœuds internes possèdent exactement deux fils
- un arbre binaire parfait est un arbre binaire complet pour lequel toutes les feuilles sont à la même profondeur
- un arbre binaire quasi parfait de hauteur h est un arbre binaire tel que :
  - toutes les feuilles sont à profondeur h ou $h-1$,
  - dont tous les nœuds internes sauf eventuellement un à profondeur $h-1$ possèdent deux fils,
  - et toutes les feuilles de profondeur h sont groupées à gauche.



Tas

- un tas max est un *arbre binaire quasi-parfait* dont l'étiquette associée à chaque nœud est plus grande que celles de ses fils



propriétés :

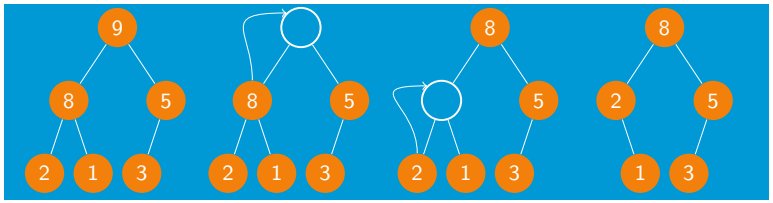
- l'étiquette la plus grande est située à la racine du tas
- pas d'ordre entre les étiquettes des fils d'un nœud (ce n'est pas un arbre binaire ordonné !)
- mais la seconde étiquette maximale est nécessairement l'étiquette d'un des deux fils de la racine
- la hauteur d'un tas de taille n est $h = \lfloor \log_2 n \rfloor$

Trier avec un tas

? **Comment faire ?**

Idée : extraire l'étiquette maximale, puis la seconde, etc ...

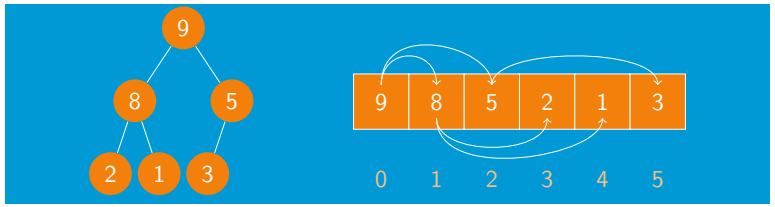
1. extraire la racine,
2. remonter la seconde étiquette maximale à la racine
3. recommencer récursivement



! **Problème : l'arbre obtenu n'est plus un tas puisque ce n'est plus un arbre quasi parfait**

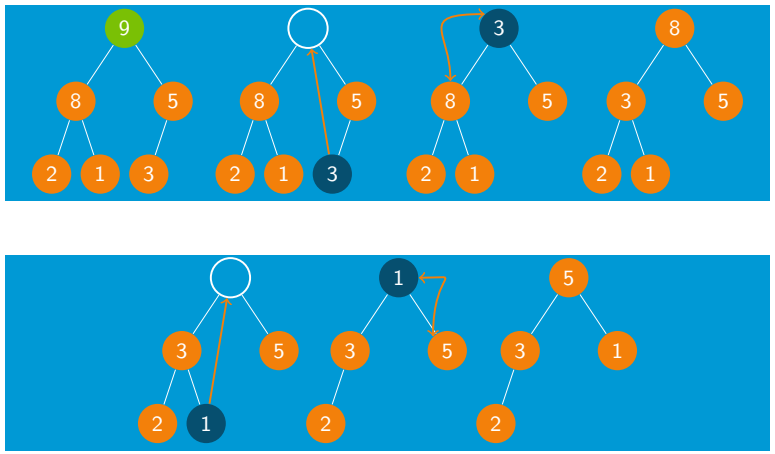
Implantation d'un tas

- si le tas est utilisé pour un tri, alors le nombre d'éléments est borné
- on peut alors utiliser une structure statique plutôt qu'une structure dynamique



les fils d'un nœud représenté à l'indice i se trouvent en positions $2 \times i + 1$ et $2 \times i + 2$

Suppression de l'élément maximum



```
procedure ReorganiserTas(t)
    p ← racine du tas
    repeat
        g ← FilsGauche(p)
        d ← FilsDroit(p)
        (*  recherche du max entre le fils gauche et droit  *)
        max ← p
        if g ≠ Vide && Racine(max) < Racine(g) then
            max ← g
        end if
        if d ≠ Vide && Racine(max) < Racine(d) then
            max ← d
        end if
        (* arrêt lorsqu'on ne peut plus descendre dans l'arbre *)
        fini ← p == max
        if not fini then
            échanger les étiquettes des nœuds p et max
        end if
    until fini
end procedure
```

Implantation de la suppression du maximum

```python
tas = { "taille": 0, "le_tas": [] }

def supprimer_max (tas):
  max = tas["le_tas"][0]
  tas["le_tas"][0] = tas["le_tas"][tas["taille"]-1]
  tas["taille"] = tas["taille"] - 1;
  if tas["taille"] >= 2:
     reorganiser(tas,0)
  return max
```


Implantation de la réorganisation

```python
def reorganiser (tas,p):
    fini = False
    pere = p
    while not fini:
        g    = 2 * pere + 1 # indice du fils gauche
        d    = 2 * pere + 2 # indice du fils droit
        imax = pere    # indice du maximum
        # recherche du max entre le fils gauche et droit
        if g < tas["taille"] and tas["le_tas"][imax] < tas["le_tas"][g]:
            imax = g
        if d < tas["taille"] and tas["le_tas"][imax] < tas["le_tas"][d]:
            imax = d
        # arret lorsqu'on ne peut plus descendre dans l'arbre
        fini = (pere == imax)
        if not fini:
            echanger(tas["le_tas"],pere,imax)
        pere = imax
```


Complexité de la réorganisation et de la suppression

Réorganisation :

- à chaque tour de boucle, dans le pire des cas, on descend dans l'arbre, et on a un échange d'éléments
- un tas a une hauteur $\lfloor \log n \rfloor$, la boucle est donc réalisée $\log n$ fois au maximum
- la complexité est donc en $\mathcal{O}(\log n)$

Suppression :

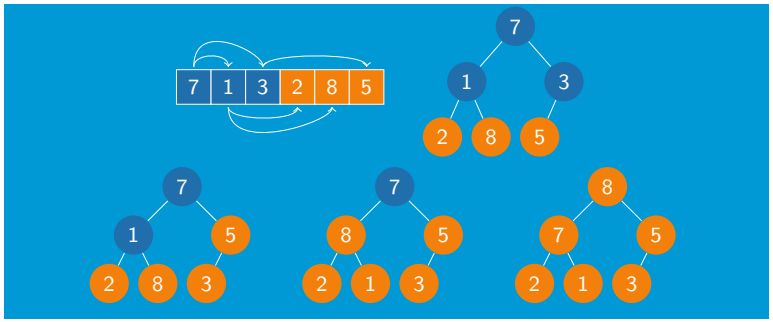
on a un échange d'éléments

\+ une réorganisation

- la complexité est donc en $\mathcal{O}(\log n)$

Création d'un tas

- comment construire le tas à partir d'un ensemble d'étiquettes quelconques ?
- la dernière moitié du tableau représente les feuilles, les feuilles sont des tas, mais pas les sous-arbres
- idée : réorganiser les sous-arbres en commençant par les plus profonds



Modification de `reorganiser`

Paramètre indiquant le nœud père du sous-arbre à réorganiser.

```
1  def reorganiser (tas,p):
2      fini = False
3      pere = p
4      while not fini:
5          g    = 2 * pere + 1 # indice du fils gauche
6          d    = 2 * pere + 2 # indice du fils droit
7          imax = pere    # indice du maximum
8          # recherche du max entre le fils gauche et droit
9          if g < tas["taille"] and tas["le_tas"][imax] < tas["le_tas"][g]:
10             imax = g
11         if d < tas["taille"] and tas["le_tas"][imax] < tas["le_tas"][d]:
12             imax = d
13         # arret lorsqu'on ne peut plus descendre dans l'arbre
14         fini = (pere == imax)
15         if not fini:
16             echanger(tas["le_tas"],pere,imax)
17         pere = imax
```


Implantation de la création

```
1  def creer (t):
2      n = len(t)
3      tas = { "taille" : n, "le_tas" : copy.deepcopy(t) }
4      # reorganisation de t
5      for i in range(tas["taille"] // 2,0,-1):
6          reorganiser(tas,i)
7      return tas
```


Complexité de la création

En première approche :

une boucle sur la moitié du tableau : $\frac{n}{2}$

$\times$ chaque réorganisation est en $\mathcal{O}(\log n)$

- d'où une complexité en $\mathcal{O}(n \log n)$

Complexité de la création

Plus finement :

- à la construction, la réorganisation se fait sur des sous-arbres de hauteur différentes : de 0 à $\lfloor \log n \rfloor$
- nombre de sous-arbres de hauteur i : 2^{h-i}
- si i est la hauteur d'un sous-arbre, le coût de la réorganisation est $\mathcal{O}(i)$
- on en déduit que la complexité est

$$\sum_{i=0}^{h} 2^{h-i} \cdot \mathcal{O}(i) = \mathcal{O}\left(2^h \cdot \sum_{i=0}^{h} \frac{i}{2^i}\right) = \mathcal{O}\left(n \cdot \sum_{i=0}^{\infty} \frac{i}{2^i}\right) = \mathcal{O}(n \cdot 2) = \mathcal{O}(n)$$

avec $\sum_{i=0}^{\infty} \frac{i}{2^i} = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2$

Implantation du tri

```
1  def trier (t):
2      tas = creer(t)
3      while tas["taille"] >= 1:
4          # extraction de l'etiquette maximale de t dans v et reorganisaton
5          v = supprimer_max(tas)
6          # rangement de l'etiquette maximale dans a
7          tas["le_tas"][tas["taille"]] = v
8      return tas["le_tas"]
```


Complexité du tri

création en $\mathcal{O}(n)$

\+ boucle sur la taille du tableau : n

× suppression de l'élément maximum en $\mathcal{O}(\log n)$

tri en $\mathcal{O}(n \log n)$