# LU2IN006

## Cours 5 - SDA file de priorité / SDA Arbre binaire

"Structures de données"

Pierre-Henri Wuillemin

2022-2023

# 1. File de priorité

**Question : Comment trouver facilement l'élément minimum d'un ensemble de données ?**

Intérêts ?

- Système de gestion (ordonnancement) de tâches (*scheduler*),
- Algorithmes gourmands (*greedy*),
- etc.

## SDA File de priorité

Une file de priorité est une structure de donnée abstraite opérant sur un ensemble de données munies d'un ordre complet ayant pour interface :

- Insérer un élément : `insert(F,e)`,
- Trouver le minimum (resp. le maximum) : `min(F)`,
- Retirer le minimum (resp. le maximum) : `suppMin(F)`

# 2. Implémentation de base : la liste





- `insert(F,e)` : insérer un élément dans la liste,
- `min(F)` : parcourir la liste et récupérer le min,
- `suppMin(F)` : trouver le min et le supprimer de la liste.

```c
typedef struct cellule {
  int priorite; /* priorite entiere */
  ...
  struct cellule *next;
} s_cellule;
```

```c
/* rechercher a partir de l */
s_cellule* argmin=l;
int min=argmin->priorite;
while (l!=NULL) {
  if (l->priorite<min) {
    argmin=l;
    min=argmin->priorite;
  }
  l=l->next;
}
```

# 2. Implémentation par liste : complexité





| Implémentation | `insert(F,e)` | `min(F)` | `suppMin(F)` |
|---|---|---|---|
| Tableau | $\Theta(1)$ | $\Theta(n)$ | $\Theta(\texttt{min(F)})+\Theta(n)$ |
| Liste Linéaire Chaînée | $\Theta(1)$ | $\Theta(n)$ | $\Theta(\texttt{min(F)})+\Theta(1)$ |

La recherche du min est trop lente ...Solution ?

Ordonner les éléments

# 2. Implémentation par liste ordonnée





Utiliser une liste ordonnée (tableau ou liste linéaire chaînée) va permettre d'accélérer la découverte du min.

- `min(F)` : récupérer le premier (ou le dernier) élément,
- `suppMin(F)` : supprimer le premier élément (ou le dernier) de la liste.
- `insert(F,e)` : trouver la bonne place et insérer un élément dans la liste,

- Implémentation par liste linéaire chaînée

  recherche de la place d'un élément : $\Theta(n)$

- Implémentation par tableau

  recherche de la place d'un élément : dichotomie $\Rightarrow \Theta(\log_2(n))$

# 2. Implémentation par liste ordonnée : complexité





| Implémentation | `insert(F,e)` | `min(F)` | `suppMin(F)` |
|---|---|---|---|
| Tableau | $\Theta(1)$ | $\Theta(n)$ | $\Theta(\texttt{min(F)})+\Theta(n)$ |
| Liste Linéaire Chaînée | $\Theta(1)$ | $\Theta(n)$ | $\Theta(\texttt{min(F)})+\Theta(1)$ |
| Tableau ordonné | $\Theta(n)$ | $\Theta(1)$ | $\Theta(1)$ |
| LLC ordonnée | $\Theta(n)$ | $\Theta(1)$ | $\Theta(1)$ |

La recherche du min est rapide ...
Mais `insert()` est trop lente ...

Raison :

La contrainte 'tout ordonner' est trop forte pour nos besoins

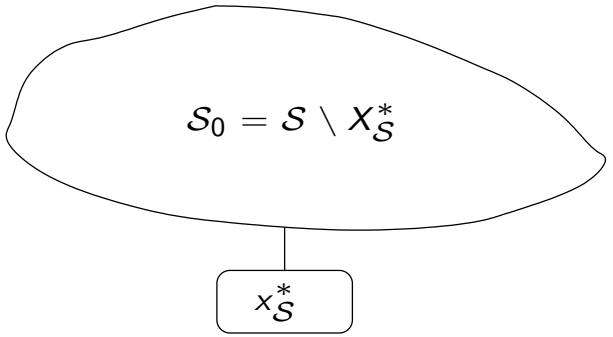
Il faut relâcher cette contrainte.

# 3. Représentation par tas.



Un tas vérifie :

- $\mathcal{S}$ l'ensemble des éléments.
- $x^*_{\mathcal{S}}$ l'élément de priorité minimale.



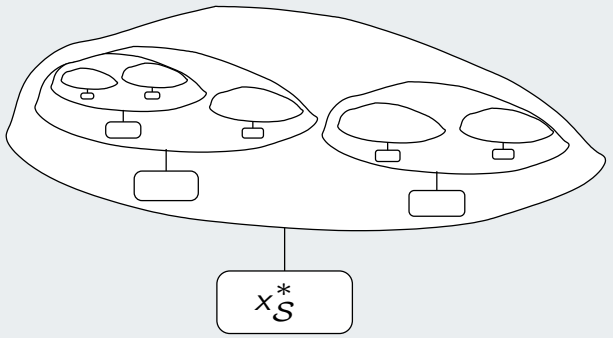

Quelle structure donner à $\mathcal{S}_0 = \mathcal{S} \setminus x^*_{\mathcal{S}}$ ?

- $\mathcal{S}_0$ est un tas $\Rightarrow$ $S$ est une liste ordonnée  c'est mal !
- $\mathcal{S}_0$ n'est pas ordonné $\Rightarrow$ représentation par liste  c'est mal !

## Structure de tas

Un tas vérifie :

- $\mathcal{S}$ l'ensemble des éléments.
- $x^*_{\mathcal{S}}$ l'élément de priorité minimale.
- $\mathcal{S}_0 = \mathcal{S} \setminus x^*_{\mathcal{S}} = \mathcal{S}_1 \cup \mathcal{S}_2$ où $\mathcal{S}_1$ et $\mathcal{S}_2$ sont des tas.

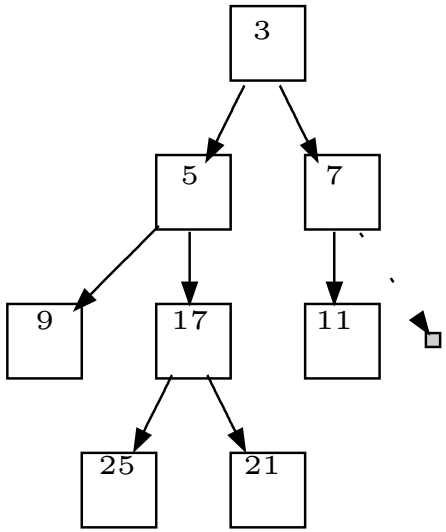


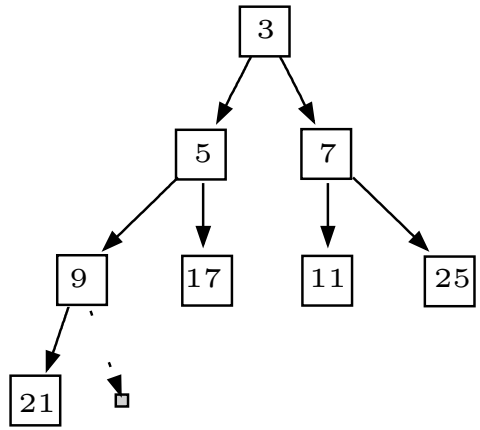
# 3. Tas comme arbre



## Définition 2 : Tas

Un tas est un arbre binaire ordonné : tous les nœuds, autre que la racine, ont une priorité plus grande que leur père.




- La structure du tas est faible et non unique !
- On trouve le minimum du tas à la racine.
- On trouve le maximum du tas dans une feuille.

Sans perte de généralité, on peut contraindre le tas à une représentation en arbre binaire tassé à gauche.

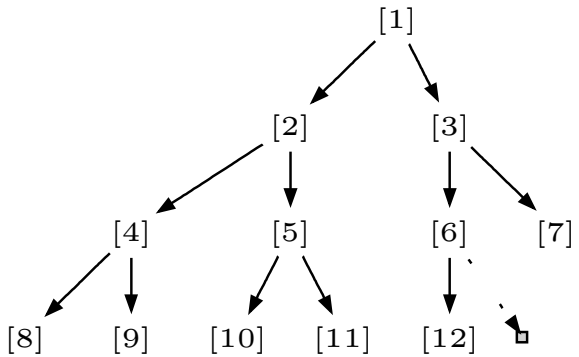
# 3. Digression rapide : arbre tassé à gauche





## Arbre binaire tassé à gauche

Un arbre binaire est tassé à gauche si tous ses niveaux sont complets, mis à part le niveau inférieur qui est complet à gauche.

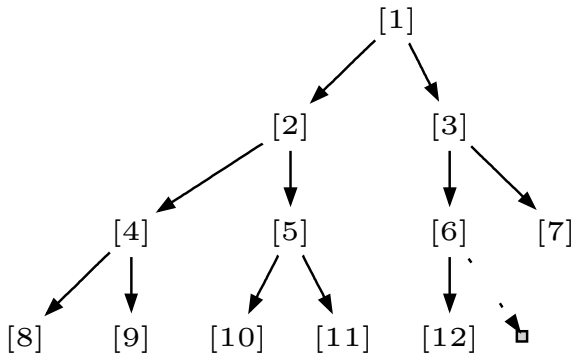



Pour un arbre tassé (à gauche) de $n$ nœuds :

- hauteur de l'arbre : $\lfloor log_2(n) \rfloor$,
- la topologie de l'arbre est unique.
- En particulier, le prochain nœud qui sera ajouté est forcément le nœud 13 : fils droit du nœud 6.

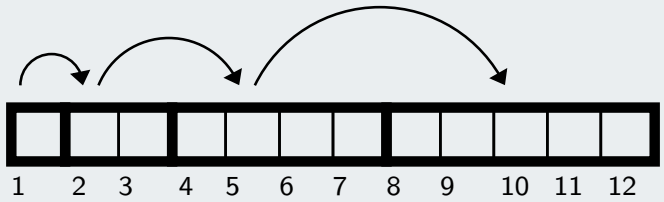
# 3. Tas : représentation par tableau






On observe qu'on peut numéroter les nœuds d'un arbre tassé à gauche avec un indice $i$ qui vérifie des propriétés intéressantes :

1. $i(\text{racine}) = 1$
2. $i(\text{fils gauche}) = 2 * i(\text{pere})$
3. $i(\text{fils droit}) = 2 * i(\text{pere}) + 1$
4. $i(\text{pere}) = \lfloor \frac{i(\text{fils})}{2} \rfloor$




On peut donc mettre les nœuds dans un tableau. Les indices sont suffisants pour représenter la structure !

10 est le fils gauche de 5 qui est le fils droit de 2 qui est le fils gauche de 1

# Fonctions indépendantes du tas





Les fonctions de transformation d'indice ne dépendent donc pas du tas sur lequel on travaille.

$i(\text{racine}) = 1$

```
1 int racine(void) {
2     return 1;
3 }
```

$i(\text{non racine}) \neq 1$

```
1 int hasPere(int i) {
2     return i!=racine();
3 }
```

$i(\text{fils gauche}) = 2 * i(\text{pere})$

```
1 int filsGauche(int i) {
2     return 2 * i;
3 }
```

$i(\text{fils droit}) = 2 * i(\text{pere}) + 1$

```
1 int filsDroit(int i) {
2     return 2 * i + 1;
3 }
```

$i(\text{pere}) = \lfloor \frac{i(\text{fils})}{2} \rfloor$

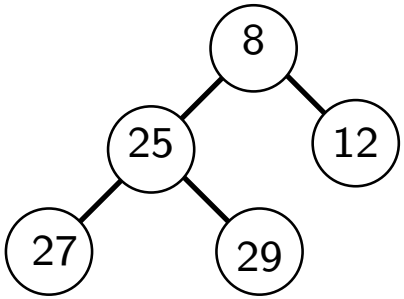
```
1 int pere(int i) {
2     /* tester si i==1 */
3     return i / 2;
4 }
```

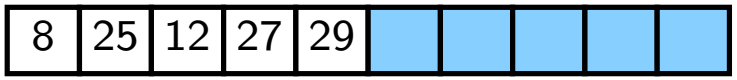
# 4. Tas : implémentation par tableau





Utiliser un tableau oblige à avoir une capacité maximum.




```c
#define CAPACITE_MAX 10

typedef struct {
  int n;
  int tab[CAPACITE_MAX+1]; /*que la priorite dans tab!*/
} Tas;
```

Les indice commencent à 1 dans un tas : on n'utilisera pas `t.tab[0]`

```c
Tas t;

printf("capacite du tas : %d\n",CAPACITE_MAX);
printf("taille du tas : %d\n",t.n);
printf("Priorite du fils droit de la racine : %d\n",
  t.tab[filsDroit(racine())]);
```

# 4. Tas : implémentation par tableau(2)





```
1  void init(Tas* t) {
2    t->n=0;
3  }
4
5  int taille(Tas* t) {
6    return t->n;
7  }
```

```
1  int isNoeud(Tas* t,int i) {
2    return i<=taille(t);
3  }
4
5  int hasFilsGauche(Tas *t,int i) {
6    return isNoeud(t,filsGauche(i));
7  }
8
9  int hasFilsDroit(Tas *t,int i) {
10   return isNoeud(t,filsDroit(i));
11 }
12
13 int estFeuille(Tas *t,int i) {
14   return ! hasFilsGauche(t,i);
15 }
```

# 4. Tas : Cohérence du tas (1)

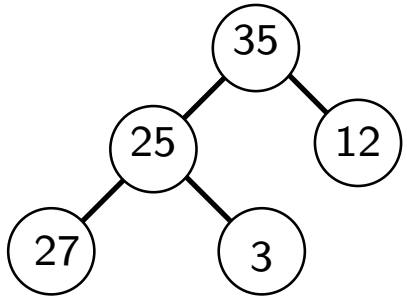




Deux opérations sont nécessaires pour garantir la structure d'un tas :

- Remonter un nœud : si sa valeur est plus petite que celle de son père.
- Descendre un nœud : si sa valeur est plus grande que celle de l'un de ses fils.

ici : 35 doit être descendu. 3 doit être remonté.




## Échanger deux nœuds d'indice i et j

```
1  void echanger(Tas *t,int i, int j) {
2    int tmp=t->tab[i];
3    t->tab[i]=t->tab[j];
4    t->tab[j]=tmp;
5  }
```

## Monter un nœud d'indice i

```
1  void monter(Tas *t,int i) {
2    if (! hasPere(i)) return;
3
4    int papa=pere(i);
5    if (t->tab[papa]>t->tab[i]) {
6      echanger(t,i,papa);
7      monter(t,papa);
8    }
9  }
```

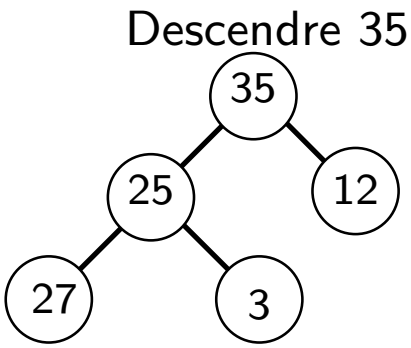
- Procédure récursive
- Complexité :

$$\Theta(\log_2(n))$$

# 4. Tas : Cohérence du tas (2)








## Trouver le plus petit fils de i

```
1  int plusPetitFils(Tas *t,int i) {
2    if (! hasFilsDroit(t,i)) {
3      return filsGauche(i);
4    } else {
5      int fg=filsGauche(i);
6      int fd=filsDroit(i);
7      return
8    (t->tab[fg]<t->tab[fd])?fg:fd;
9    }
10 }
```

## Descendre un nœud d'indice i

```
1  void descendre(Tas *t,int i) {
2    if (isFeuille(t,i)) return;
3
4    int fiston=plusPetitFils(t,i);
5    if (t->tab[i] > t->tab[fiston]) {
6      echanger(t,fiston,i);
7      descendre(t,fiston);
8    }
9  }
```

- Procédure récursive
- Complexité :

$$\Theta(\log_2(n))$$

# 4. Interface : min, insert et suppMin





## min()

```
1  void min(Tas *t) {
2    return t->tab[racine()];
3  }
```

- Complexité :

$\Theta(1)$

## insert()

```
1  int insert(Tas *t,int priorite) {
2    /* verification depassement
         capacite */
3    t->n++;
4    t->tab[t->n]=priorite;
5    monter(t,t->n);
6  }
```

- Complexité :

$\Theta(\log_2(n))$

## suppMin()

```
1  void suppMin(Tas *t) {
2    /* verification tas non vide */
3    echanger(t,t->n,racine());
4    t->n--;
5    descendre(t,racine());
6  }
```

- Complexité :

$\Theta(\log_2(n))$

# conclusion



| Implémentation | `insert(F,e)` | `min(F)` | `suppMin(F)` |
|---|---|---|---|
| Tableau | $\Theta(1)$ | $\Theta(n)$ | $\Theta(\texttt{min(F)})+\Theta(n)$ |
| Liste Linéaire Chaînée | $\Theta(1)$ | $\Theta(n)$ | $\Theta(\texttt{min(F)})+\Theta(1)$ |
| Tableau ordonné | $\Theta(n)$ | $\Theta(1)$ | $\Theta(1)$ |
| LLC ordonnée | $\Theta(n)$ | $\Theta(1)$ | $\Theta(1)$ |
| Tas | $\Theta(\log_2(n))$ | $\Theta(1)$ | $\Theta(\log_2(n))$ |

Une petite application :

```
1  Tas t;
2  init(&t);
```

```
n=0
tab= [0, 0, 0, 0, 0, 0]
```

```
1  insert(&t,10);
2  insert(&t,5);
3  insert(&t,23);
4  insert(&t,13);
5  insert(&t,2);
```

```
n=5
tab= [2, 5, 23, 13, 10, 0]
```

```
1  suppMin(&t);
2  suppMin(&t);
3  suppMin(&t);
4  suppMin(&t);
5  suppMin(&t);
```

```
n=0
tab= [23, 13, 10, 5, 2, 0]
```

Tri par tas ! $\Theta(n\log_2(n))$

# Arbre

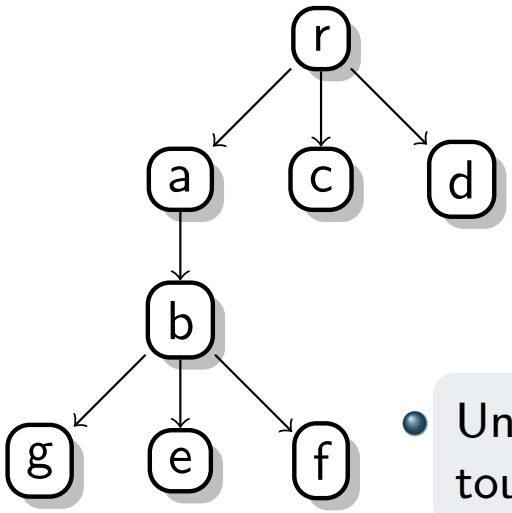




## Arbre

Un arbre est un ensemble fini $A$ d'éléments, liés entre eux par une relation, dite de "parenté", vérifiant ces propriétés :

- Relation notée "$x$ est le parent de $y$" ou "$y$ est le fils de $x$",
- Il existe un unique élément $r$ (**racine**) de $A$ sans parent,
- À part $r$, tout élément de $A$ possède un **unique** parent.




- Les éléments de $A$ sont des **nœuds**,
- Les nœuds sans fils sont des **feuilles** ou **nœuds terminaux**,
- Les descendants d'un nœud $x$ forment le **sous-arbre** de racine (ou issu de) $x$,

- Un arbre $n$-**aire** est un arbre dont les nœuds ont tous au plus $n$ fils.

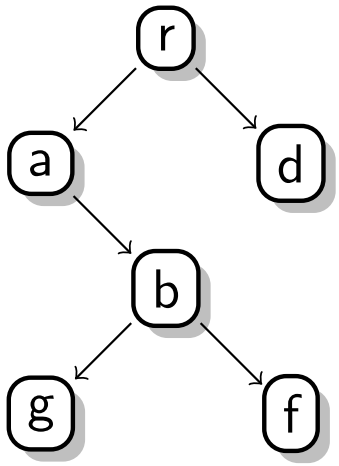
# Arbre binaire





## Arbre binaire

- Un arbre binaire est un arbre 2-aire.
- Les fils sont spécialisés et nommés : fils **gauche** et fils **droit**.
  On confondra souvent le fils et le sous-arbre issu du fils.




## Arbre binaire - définition récursive

Un arbre binaire $A$ est défini par :

- L'arbre vide ($\emptyset$) est un arbre binaire
- l'arbre de racine $r$, de fils gauche $A_g$ et de fils droit $A_d$ est un arbre binaire si $A_g$ et $A_d$ sont des arbres binaires.

# Implémentation d'un arbre binaire en C

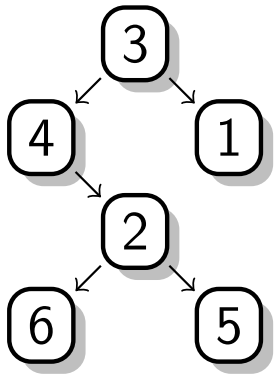


## Implémentation


```c
typedef struct s_btree {
  int content;

  struct s_btree* fd;
  struct s_btree* fg;
} btree;
```

On se limite à un contenu de type `int`

## Création d'un nœud

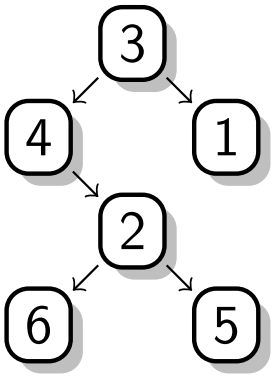



```c
btree* cree(int content, btree* fd, btree* fg)
    {
  btree* n=(btree *)malloc(sizeof(btree));

  n->content=content;
  n->fg=fg;
  n->fd=fd;

  return n;
}
```

```
btree *t=cree(3, cree(4, NULL, cree(2, cree(6, NULL, NULL), cree(5, NULL, NULL))), cree(1, NULL, NULL));
```

# Parcours dans les arbres

## Afficher les éléments contenus dans un arbre :




- **Parcours préfixe : 3 4 2 6 5 1**

```
1  void prefix(btree *t) {
2    if (t!=NULL) {
3      printf("%d ",t->content);
4      prefix(t->fg);
5      prefix(t->fd);
6      }
7    }
```

- **Parcours suffixe : 6 5 2 4 1 3**

```
1  void suffix(btree *t) {
2    if (t!=NULL) {
3      suffix(t->fg);
4      suffix(t->fd);
5      printf("%d ",t->content);
6      }
7    }
```

- **Parcours infixe : 4 6 2 5 3 1**

```
1  void infix(btree *t) {
2    if (t!=NULL) {
3      infix(t->fg);
4      printf("%d ",t->content);
5      infix(t->fd);
6      }
7    }
```

# Fonctions récursives simples (avec btree)





- taille d'un arbre (nombre de nœuds) :

*La taille d'un arbre est 1 + la taille du fils gauche + la taille du fils droit*

```
1  void taille(btree *t) {
2    if (t==NULL)
3      return 0;
4    else
5      return 1+taille(t->fg)+taille(t->fd);
6  }
```

- hauteur d'un arbre

$$h(A) = \begin{cases} -1 & \text{si } A = \emptyset \\ 1 + max(h(A_g), h(A_d)) & \text{sinon.} \end{cases}$$

```
1  void hauteur(btree *t) {
2    if (t==NULL)
3      return -1;
4    else
5      return 1+max(hauteur(t->fg),hauteur(t->fd));
6  }
```

# Fonctions récursives simples (2)





- Supprimer un arbre : parcours suffixe

```
1  void supprime(btree *t) {
2    if (t!=NULL) {
3      supprime(t->fg);
4      supprime(t->fd);
5      free(t);
6      }
7    }
```

- Copie un arbre : parcours suffixe

```
1  btree* copie(btree *t) {
2    if (t==NULL)
3      return NULL;
4    else
5      return cree(t->content,
6                  copie(t->fg),
7                  copie(t->fd));
8  }
```

# Fonctions récursives simples (2)





- Trouver un élément dans un arbre : parcours prefixe

```
1  btree* exists(btree* t, int val) {
2    if (t!=NULL) {
3      if (t->content==val)
4        return t;
5      else {
6        btree* tmp;
7
8        tmp=exists(t->fg,val);
9        if (tmp!=NULL)
10         return tmp;
11
12       tmp=exists(t->fd,val);
13       if (tmp!=NULL)
14         return tmp;
15     }
16   }
17
18   return NULL;
19 }
```