Algorithmique et structures de données en C

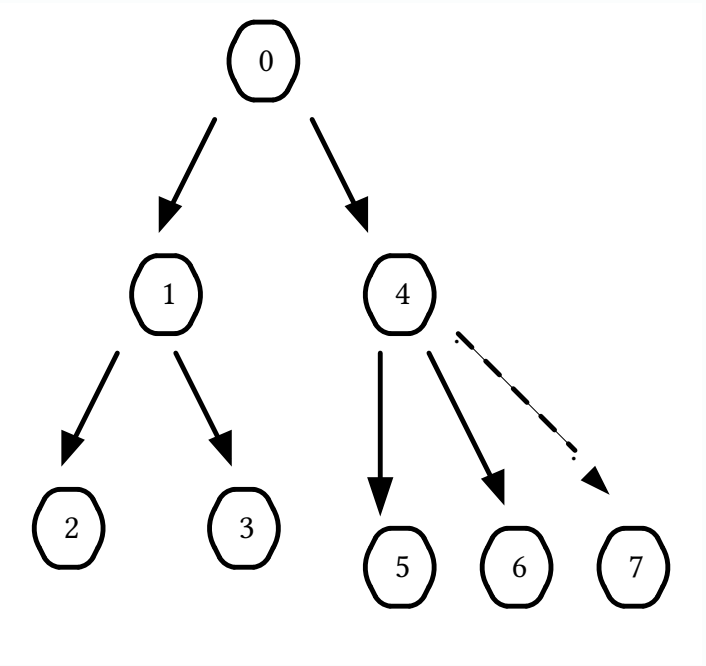
Arbres et Tables de hashage

Enseignant: P. Bertin-Johannet

Arbres

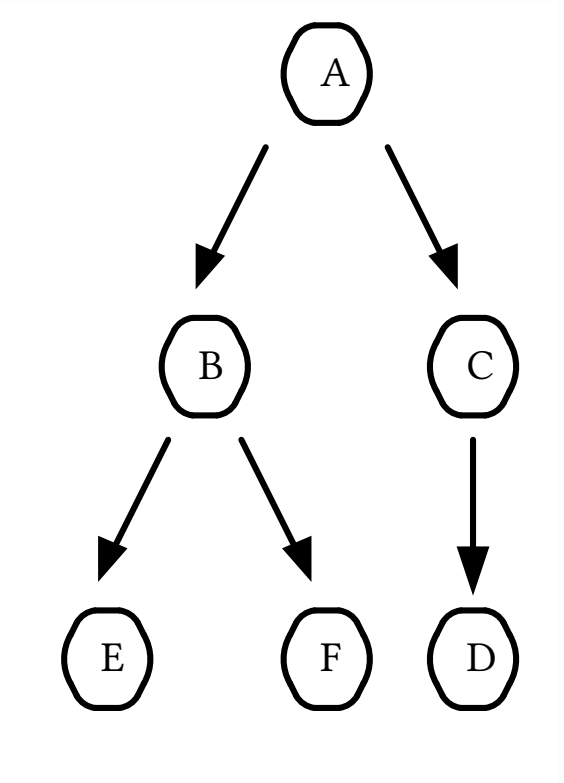
Un arbre

- Un arbre est une structure de données contenant des **noeuds** connectés par des **arrêtes**.
- Un noeud peut possèder plusieurs noeuds fils.
- Chaque noeud possède exactement un noeud parent sauf le noeud le plus haut appellé **racine** qui ne possède que des fils.
- Les arrêtes ne forment pas de cycle.



Arbre : définitions

- Le degré d'un noeud correspond à son nombre de fils (ici le noeud **B** est de degré 2).
- Le niveau d'un noeud est le nombre d'arrêtes qui le séparent de la racine (le noeud **D** est de niveau 2).
- La hauteur d'un arbre est son niveau maximal.



Arbre : Implémentation

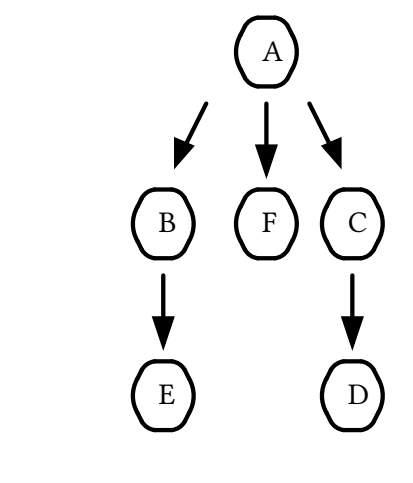
- On peut implémenter les noeuds d'un arbre en utilisant une liste des noeuds fils.

```
struct Noeud{
  char valeur; // le contenu du noeuds
  struct Noeud* fils; // les fils du noeud
}
```

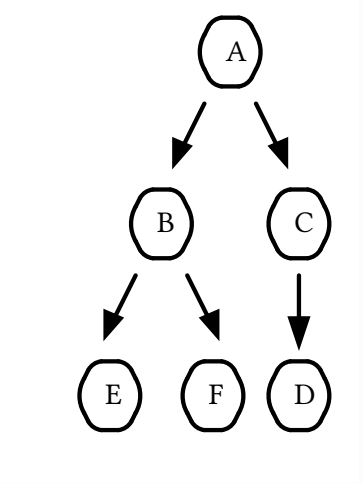
Arbre binaire

- Un arbre binaire est un arbre dans lequel le degré **maximal** de chaque noeud est 2.
- Tout arbre peut se ramener à une représentation binaire.

- Arbre lambda



- Mêmes noeuds mais arbre binaire



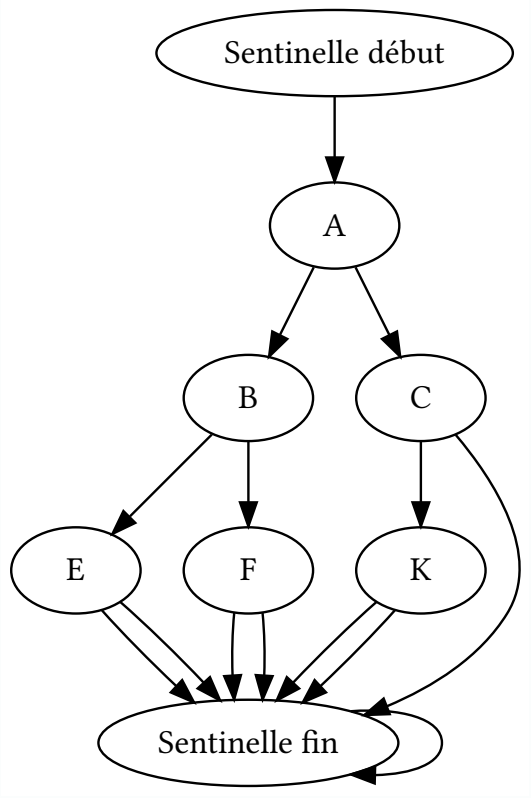
Arbre binaire : Implémentation par pointeur

- On peut implémenter un arbre en utilisant deux pointeurs vers les deux fils de chaque noeud.

```
struct Noeud {
  int valeur; // la valeur contenue dans le noeud
  struct Noeud* noeud_gauche; // le noeud gauche
  struct Noeud* noeud_droit; // le noeud droit
};
```

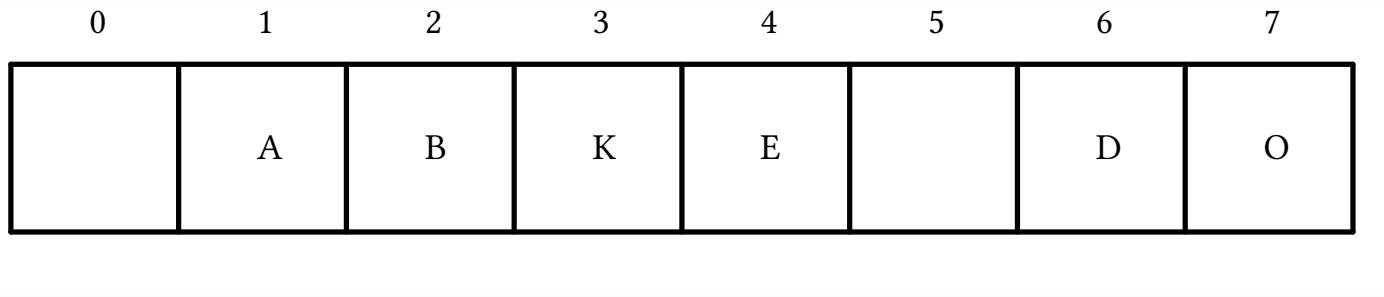
Arbre binaire et noeuds sentinelles

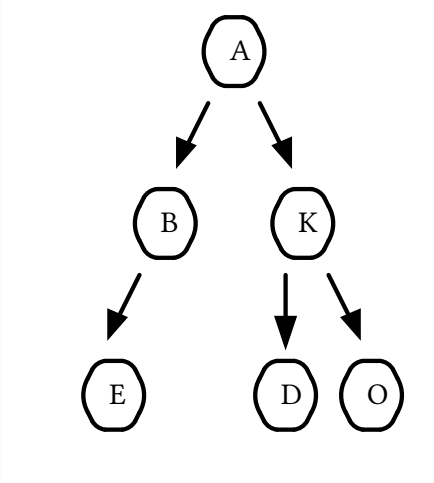
- Pour accèlérer la recherche ainsi que pour simplifier l'implémentation, on peut utiliser des noeuds sentinelles de début et de fin.
- Les noeuds auront alors tous un parent et deux fils.



Arbre binaire : Implémentation par tableau

- On peut implémenter un arbre binaire en utilisant un tableau
- On calculera automatiquement la position d'un noeud dans le tableau selon la position du noeud parent
- Les fils du noeud d'indice **i** seront enregistrés aux positions **2i** et **2i+1**

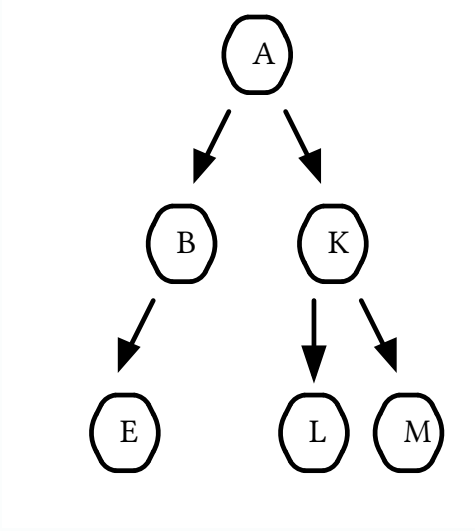




Arbre binaire : Implémentation par tableau de liens vers le parent

- Si on souhaite uniquement utiliser les lien vers les parents on peut implémenter un arbre binaire en utilisant deux tableaux

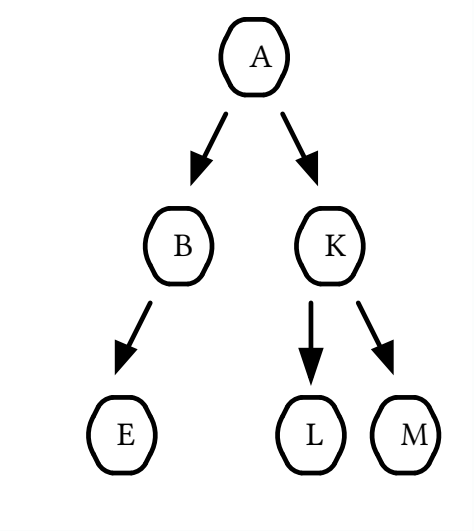
0	1	2	3	4	5
A	L	K	E	B	M
	1	0	2	0	1



Arbre : Parcours récursif

- On peut parcourir tous les noeuds d'un arbre en utilisant une fonction récursive
- Lorsqu'on parcours d'abord le noeud parent, on parle de parcours préfixé

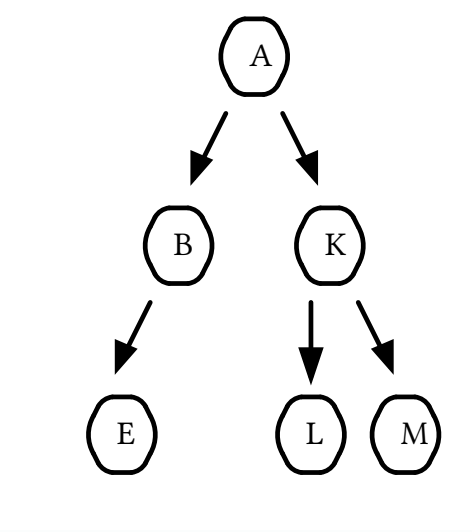
```
void parcours(Noeud* n){
  printf("%d\n", n->valeur);
  parcours(n.noeud_gauche);
  parcours(n.noeud_droit);
}
// affichera A B E K L M
```



Arbre : Parcours récursif

- Lorsqu'on parcours d'abord les noeuds fils, on parle de parcours postfixé

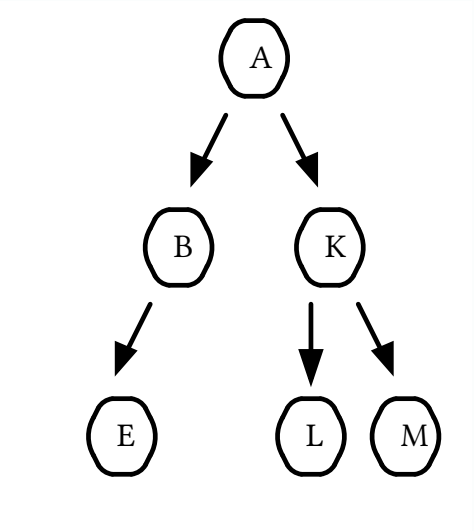
```c
void parcours(Noeud* n){
  parcours(n.noeud_gauche);
  parcours(n.noeud_droit);
  printf("%d\n", n->valeur);
}
// affichera E B L M K A
```



Arbre : Parcours récursif

- Lorsqu'on parcours le noeud de gauche puis le noeud père puis le noeud de droite, on parle de parcours infixé.

```
void parcours(Noeud* n){
  parcours(n.noeud_gauche);
  printf("%d\n", n->valeur);
  parcours(n.noeud_droit);
}
// affichera E B A L K M
```



Arbre : Parcours en largeur et profondeur

- Dans les trois fonctions précédentes, on parcours d'abord une branche jusqu'au dernier niveau avant de parcourir les autres noeuds de plus haut niveau. On parle alors de **parcours en profondeur**.
- Lorsqu'on parcours tous les noeuds du même niveau avant de parcourir les noeuds du niveau suivant on parle de **parcours en largeur**.

Parcours en profondeur sans récursion

- On peut effectuer un parcours d'arbre en profondeur sans récursion en utilisant une pile.

```
void parcours_profondeur(Noeud* n){
  Pile* p;
  empiler(p, n);
  while(taille(p)){
    Noeud* suivant = depiler(p);
    if (suivant->noeud_gauche != suivant){ // on vérifie pour la fin
      empiler(suivant->noeud_gauche);
      empiler(suivant->noeud_droit);
      printf("%d\n", suivant->valeur);
    }
  }
}
```

Parcours en largeur

- Pour effectuer un parcours d'arbre en largeur, on remplace la pile par une file.

```
void parcours_largeur(Noeud* n){
  File* f;
  enfiler(f, n);
  while(taille(f)){
    Noeud* suivant = defiler(f);
    if (suivant>noeud_gauche != suivant){ // on vérifie pour la fin
      enfiler(suivant->noeud_gauche);
      enfiler(suivant->noeud_droit);
      printf("%d\n", suivant->valeur);
    }
  }
}
```

Parcours : exemples d'utilisation

- On peut utiliser un parcours en profondeur :
  - Pour calculer le résultat d'un arbre représentant une expression
  - Pour chercher une valeur dans un arbre
  - Pour effectuer certains algorithmes de type "brute-force"
- On peut utiliser un parcours en largeur :
  - Pour touver des noeuds à une distance donnée d'un noeud dans un graphe
  - Pour trouver le noeud le plus proche de la racine qui répond à un critère donné
  - Pour simuler l'avancée d'une information ou d'une épidémie

Arbre complet

- On appelle feuille un noeud qui n'a pas de fils.
- On parle d'arbre binaire complet lorsque:
  - Tous les niveaux sont remplis (les noeuds ont deux fils) sauf le dernier niveau.
  - Pour les noeuds du dernier niveau qui n'ont qu'un seul fils, ce fils est à gauche.
- Pour un arbre complet de taille n le nombre d'opérations à effectuer pour accèder à un noeud depuis la racine est alors au maximum $\log_2(n)$.

Arbre binaire de recherche

- On parle d'arbre binaire de recherche (ou ABR) lorsque :
  - L'arbre est complet.
  - les valeurs de chaque noeud respectent la relation d'ordre suivant avec ses fils :

$$\text{val min sous-arbre gauche} < \text{val père} < \text{val min sous-arbre droit}$$

- La recherche d'une clé dans un ABR de taille n nécéssite au maximum $\log_2(n)$ opérations. On parle alors de complexité logarithmique.
- Rechercher un élément dans un arbre de taille dix milliards ne demande alors au maximum qu'une centaine d'opérations.

Insertion et suppression dans un arbre binaire de recherche

- Lorsqu'on insère ou supprime un élément, on doit s'assurer de toujours respecter les propriétés de l'arbre binaire de recherche
- On utilise alors des parcours qui modifient la structure de l'arbre avant d'insérer ou supprimer un élément.

Tables de hashage

Table de hashage

- Une table de hashage est une structure de données qui permet d'associer des clés et des valeurs.
- On utilise une fonction appellée "de hashage" pour associer un nombre à chaque valeur potentielle.
- Les valeurs sont enregistrées dans un tableau à la position donnée par la fonction de hashage.
- Par exemple si `hash("Lundi")` = 3, `hash("Mercredi")` = 0 et `hash("Vendredi")` = 5, on enregistrera les valeurs aux addresses 3, 0 et 5 dans le tableau.

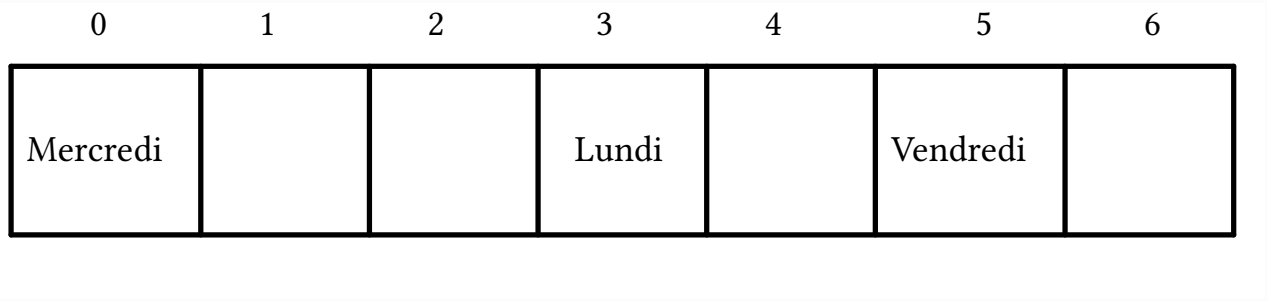


Table de hashage - Collisions

- Lorsque la fonction de hashage renvoie le même résultat pour plusieurs clés, on parle de collision.
- Dans ce cas, on doit enregistrer la liste des clés qui ont le même hash dans la case du tableau.
- Par exemple si `hash("Lundi")` = 3, `hash("Mercredi")` = 0, `hash("Vendredi")` = 5 et `hash("Samedi")` = 3, on enregistrera un tableau contenant Samedi et Lundi à la case 3 du tableau.

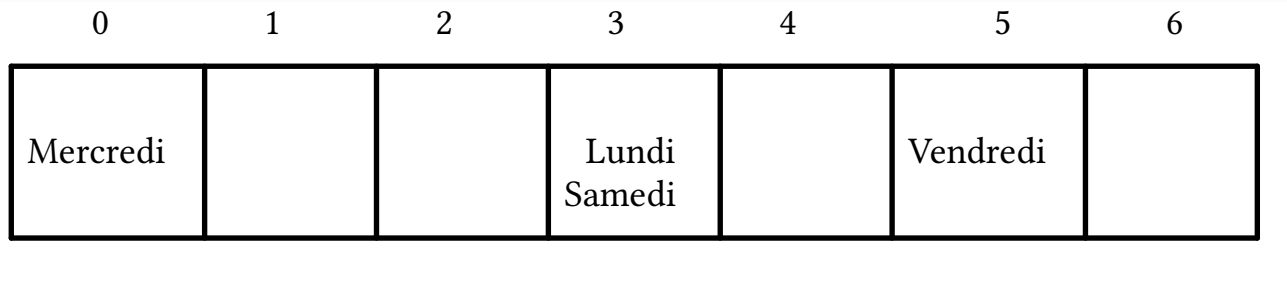


Table de hashage - Temps d'accès

- Lorsqu'il n'y a aucune collision pour les clés enregistrées dans la table, la recherche d'une clé s'effectue en temps constant $O(1)$.
- Dans le cas ou la fonction de hashage renvoie la même clé pour n valeurs, la recherche d'une clé s'effectue en temps linéaire $O(n)$.
- Il est donc très important de choisir une fonction de hashage qui minimise la probabilité de collisions.
- Il est aussi possible d'allouer un tableau plus grand afin de réduire la probabilité de collision, il faudra alors recalculer le hash de chaque clé avant de les réinsérer. Le cout de cette opération sera donc linéaire $O(n)$.

Table de hashage : exemples d'utilisations

- Les objets de type HashMap en Java, Dictionnary en C# et Python, std::map en C++ etc.. utilisent tous une table de hashage pour permettre un accès rapide aux éléments.
- Les interpréteurs de certains langages (PHP et Python par exemple) utilisent aussi une hashmap pour associer le nom des fonctions, des objets ainsi que des variables à leurs valeurs respectives.
- Une table de hashage peut aussi parfois être utilisée accélérer la recherche de valeurs dans une base de données.

Mise en pratique

```
printf("
          _________
        < TP Algo >
          --------
                 \   ^__^
                  \  (oo)\_______
                     (__)\       )\/\
                         ||----w |
                         ||     ||
        ")
```