Algorithmique et structures de données en C

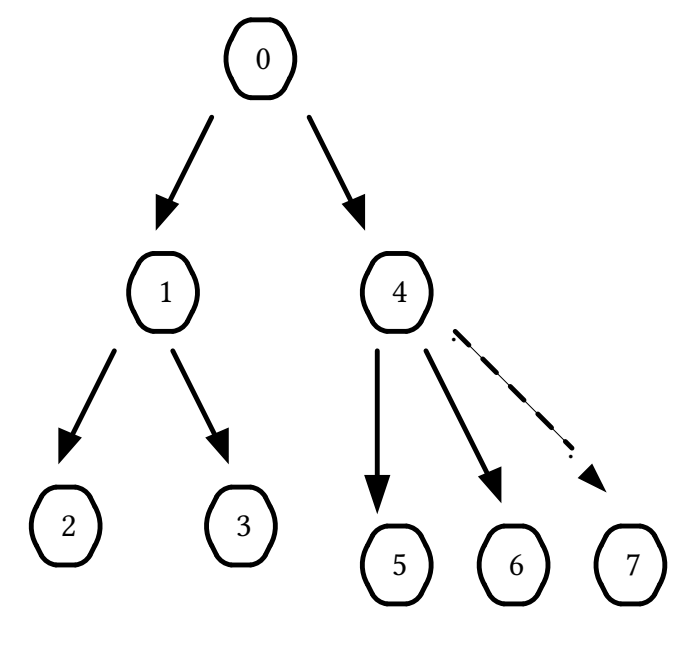
Arbres

Enseignant: P. Bertin-Johannet

Arbres

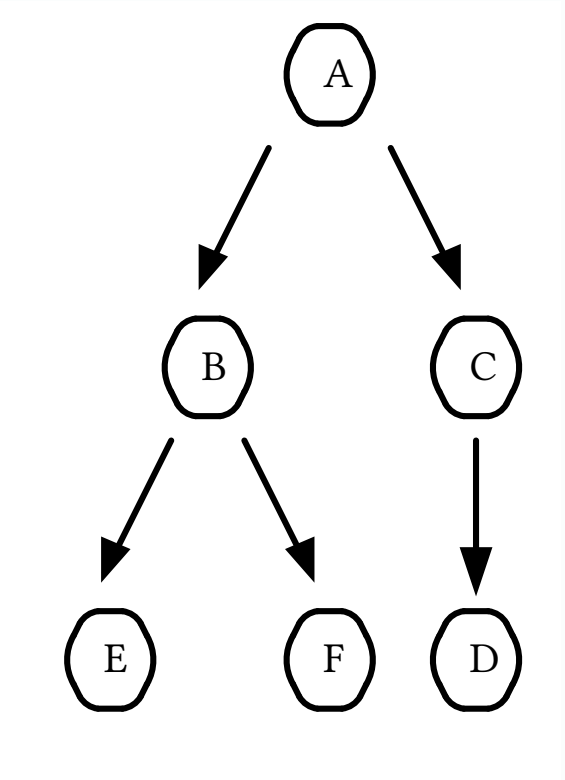
Un arbre

- 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).
- On appelle feuille un noeud qui n'a pas de fils.
- 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

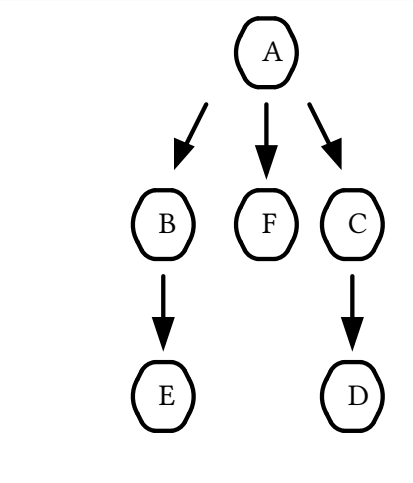
- On peut implémenter les noeuds d'un arbre en utilisant une liste des noeuds fils.

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

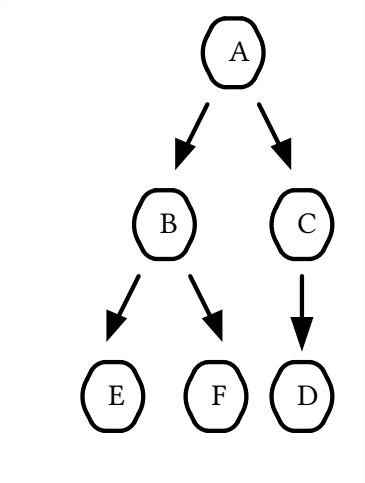
Arbre binaire

- 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



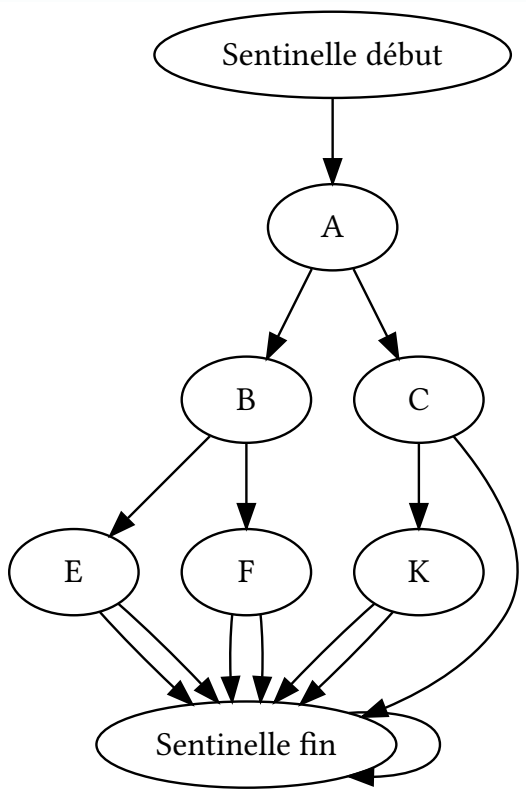
Arbre binaire : Implémentation par pointeur

- On peut implémenter un arbre binaire 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

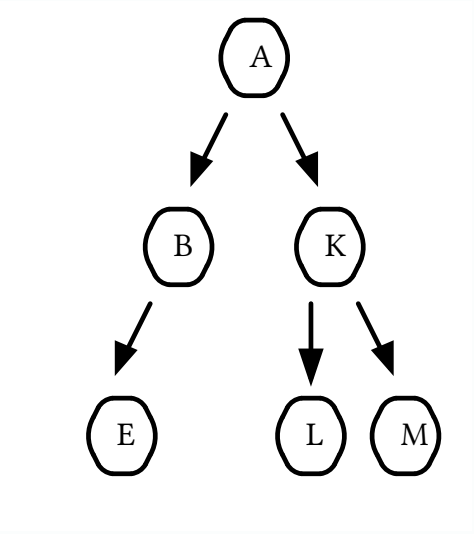
- Pour accèlérer la recherche ainsi que pour simplifier l'implémentation, on peut utiliser des noeuds sentinelles de début et de fin.
- Les noeuds auront alors tous un parent et deux fils.



Arbre : 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.

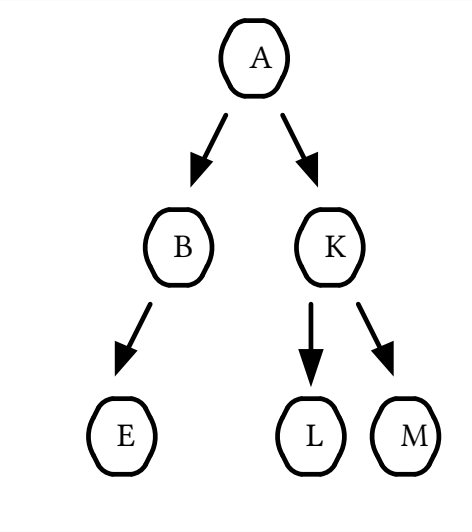
```
void parcours(Noeud* n){
  printf("%d\n", n->valeur);
  parcours(n.noeud_gauche);
  parcours(n.noeud_droit);
}
// affichera A B E K L M
```



Arbre : Parcours récursif

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

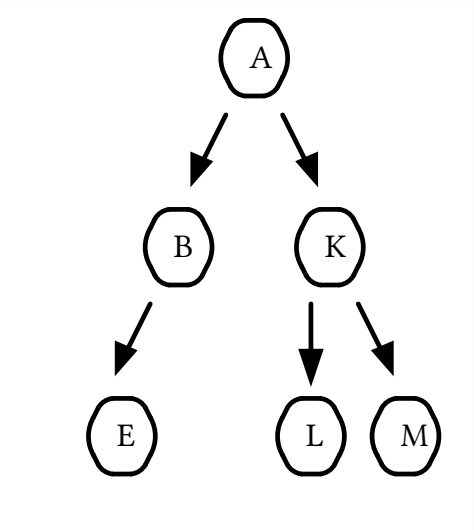
```
void parcours(Noeud* n){
  parcours(n.noeud_gauche);
  parcours(n.noeud_droit);
  printf("%d\n", n->valeur);
}
// affichera E B L M K A
```



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(p, suivant->noeud_gauche);
      empiler(p, 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 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 de liens qui séparent un noeud de 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 suivante avec ses fils :

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

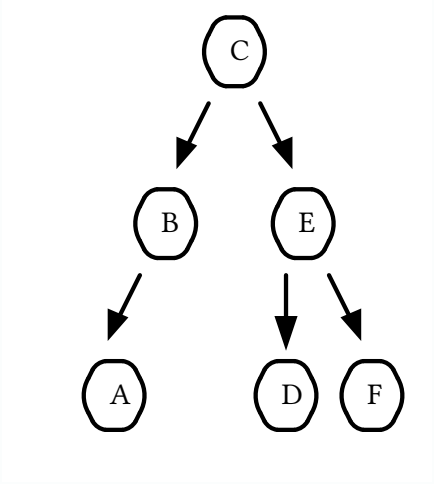
- La recherche d'une clé dans un ABR de taille n nécéssite au maximum $\log_2(n)$ opérations. On parle alors de **complexité logarithmique**.

- L'affichage des données triées s'effectue en temps linéaire $O(n)$.

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

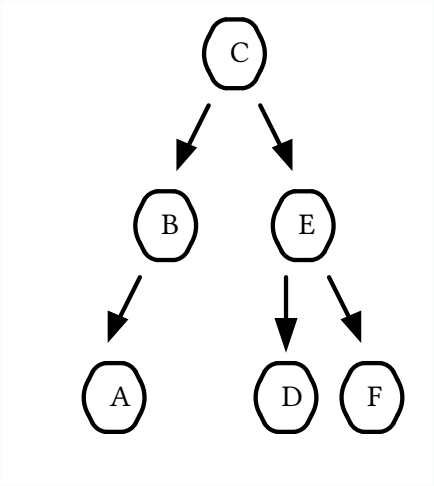
- Lorsqu'on insère ou supprime un élément, on doit s'assurer de toujours respecter les propriétés de l'arbre binaire de recherche

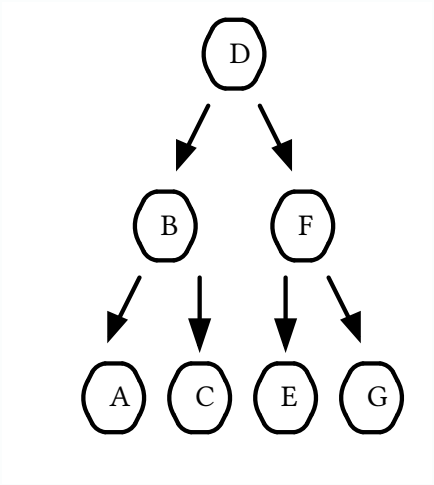


- Ici, après si on insère l'élément G dans l'arbre, il faut le réequilibrer.

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





- Ici, après si on insère l'élément G dans l'arbre, il faut le réequilibrer.

Insertion et suppression dans un arbre binaire de recherche

- Il existe de nombreux algorithmes différents assurant l'équilibrage de l'arbre lors de l'insertion ou de la suppression d'un élément, on peut citer.

- La plupart d'entre eux permettent d'effectuer ces opérations en temps linéaire ($O(\log(n))$).

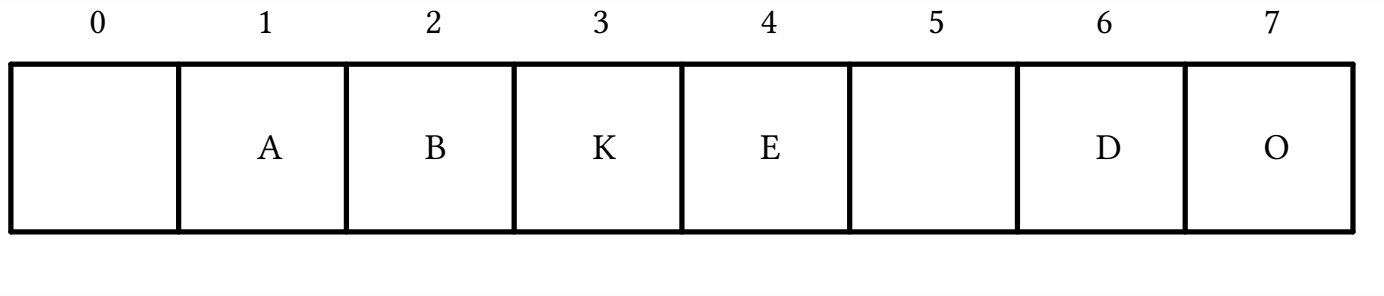
- Dans certain cas, le cout amorti de ces opérations peut être constant ($O(1)$).

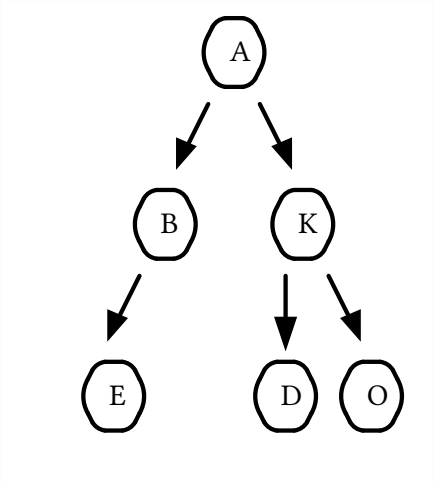
Implémentations d'arbres 'modernes'

- L'implémentation efficace d'un arbre entraine les mêmes ralentissements que la liste chainée:
  - Enregistrer un pointeur avec chaque élément **augmente la mémoire utilisée.**
  - Les données sont éparpillées dans la mémoire et **difficiles à cacher.**
  - Le dernier point rend aussi **les prédictions difficiles.**
  - Chaque insertion demande une allocation, cela **augmente le nombre d'appels système.**
- Bien que l'arbre binaire de recherche offre de meilleures performances asymptotiques, il est parfois plus rapide d'utiliser un tableau pour des petits jeux de données.
- Pour ces raisons, l'implémentation précédement présentée est **rarement utilisée** dans des cas ou la performance est importante.
- Le cours présentera quelques variantes utiles pour attenuer ces problèmes.

Arbre binaire non equilibré - Implémentation par tableau

- Si on n'est pas intéressé par l'ordre des éléments, il peut être préférable d'utiliser 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**.





Implémentation par tableau - avantages et désavantages

Avantages:

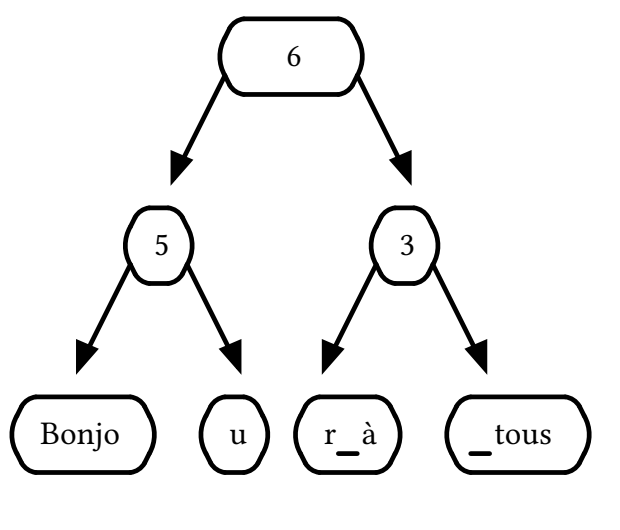
- Pas besoin d'enregistrer de pointeur => *Emprunte mémoire faible si les éléments le sont aussi (par exemple* `char`, `int`, `float`*).*
- L'insertion ne nécessite que très rarement un appel système.
- Les noeuds d'un même niveau sont proches dans la mémoire.

Désavantages:

- Réserve de la place pour plus d'éléments que nécéssaire (plus si l'arbre n'est pas équilibré).
- Si l'insertion d'un élément rajoute un niveau, il peut être nécéssaire de réallouer et recopier => *insertion en* $O(n)$.

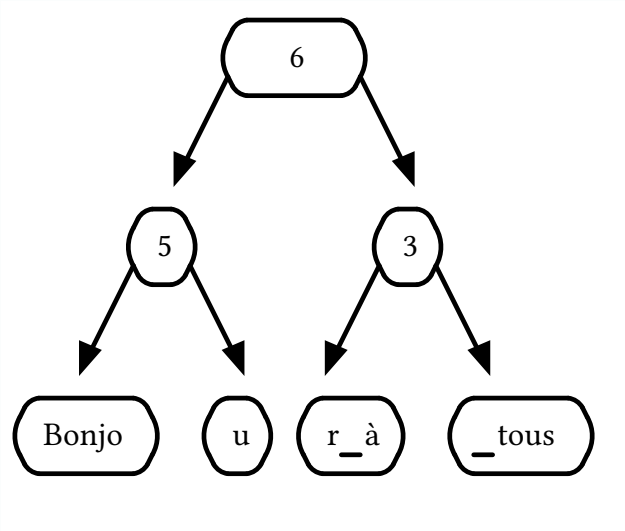
Pour un accès indexé rapide - Rope

- Une rope est comme une liste déroulée mais enregistre les noeuds dans un arbre binaire équilibré.
- Les feuilles contiennent des vecteurs.
- Les noeuds internes contienent le nombre total de charactères dans le sous arbre de gauche.



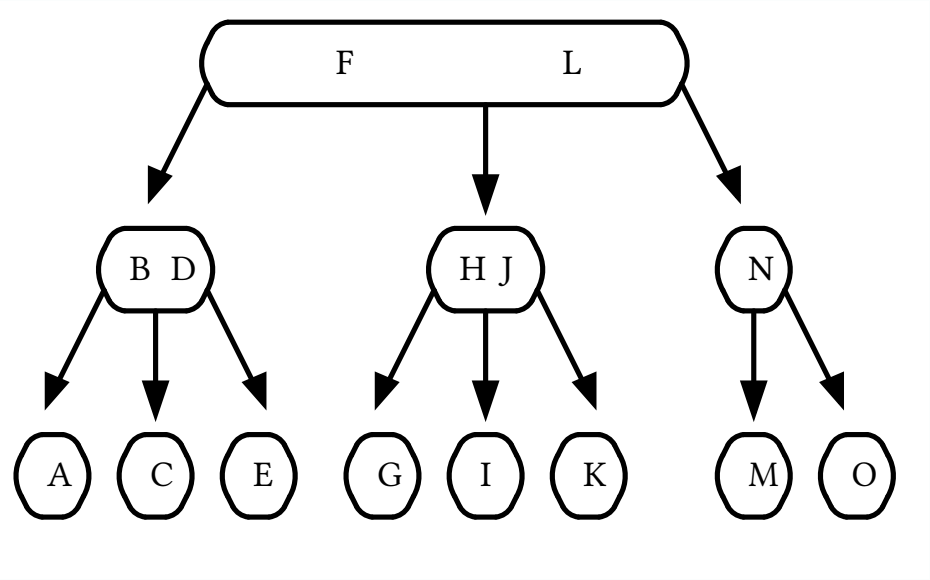
Pour un accès indexé rapide - Rope

- L'insertion, la suppression et l'accès arbitraires deviennent donc $O(\log(n))$ plutôt que $O(n)$.
- L'usage de vecteurs comme feuilles améliore la **localité**, la **prédictibilité** ainsi que l'**empreinte mémoire**.
- Très utilisée dans les outils de manipulation de texte en temps réel (IDEs par exemple.)



Pour rechercher et trier - Arbre B

- Chaque noeud pourra avoir jusqu'a **B fils** et contiendra donc **B - 1 éléments**.
- La hauteur d'un arbre B complet est de $\log_B(n)$.



Arbres B - Avantages et inconvénients

- Avantages:
  - Améliore la **localité**, la **prédictibilité** ainsi que l'**empreinte mémoire**.
  - **Réduit le nombre d'appels système** moyens pour l'insertion et la suppression.
- Désavantages:
  - Plus **difficile à implémenter**.
  - Effectue **plus d'opérations en moyenne**.
  - Peut demander de trouver une valeur de B optimale selon le cas d'usage.
- *Sur une architecture moderne, l'arbre B est presque toujours une solution plus rapide qu'un arbre binaire de recherche.*

Exemples réels d'arbres

- Applications:
  - Les **IDEs** jetbrains (intellij, pycharm, phpstorm…), Emacs, vim, SublimeText et VScode utilisent une **corde** (ou structure similaire) pour traiter le texte.
  - Le **noyau linux** utilise des **arbres binaires** pour ordonnancer les processus et traquer les régions mémoires allouées.
  - Les **bases de données** PostgreSQL, MySQL, SQLite, Oracle, MongoDB et dynamoDB utilisent un **arbre B** pour enregistrer les index.
  - Les systèmes de fichiers **NTFS, ext4 et APFS** utilisent un **arbre B** pour minimiser le nombre d'accès arbitraires au disque.
  - **GoogleMaps et OpenStreetMaps** utilisent des **arbres B** pour accélérer la recherche d'éléments proches entre eux.
- Langages
  - `std::map` en C++, `class TreeMap` en java et `class SortedDictionary` en C# utilisent un **arbre binaire**.
  - `struct BTreeMap` en rust utilise un **arbre B**.

Mise en pratique

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