**STRUCURES DE DONNEES**

**Les Listes**

1. **Simplement Chaînée (Simple Linked List) :**
   * **Usage :** Lorsque l'insertion/suppression fréquente en début ou milieu de liste est nécessaire.
   * **Avantage :** Facile à implémenter, faible surcoût d'insertion et suppression.
   * **Inconvénient :** Accès séquentiel, coût élevé pour l'accès aléatoire.
2. **Simplement Chaînée Circulaire (Circular Linked List) :**
   * **Usage :** Quand une liste doit être parcourue en boucle.
   * **Avantage :** Boucle facile, insertion/suppression efficace.
   * **Inconvénient :** Coût d'accès aléatoire élevé.
3. **Doublement Chaînée (Doubly Linked List) :**
   * **Usage :** Lorsque l'accès bidirectionnel est nécessaire.
   * **Avantage :** Accès séquentiel et aléatoire efficace, insertion/suppression à des positions arbitraires.
   * **Inconvénient :** Complexité et espace supplémentaires.
4. **Doublement Chaînée Circulaire (Circular Doubly Linked List) :**
   * **Usage :** Combinaison des avantages des listes doublement chaînées et circulaires.
   * **Avantage :** Accès bidirectionnel et boucle facile.

**Les Piles**

* **Usage :** Lorsqu'un accès LIFO (Last In First Out) est nécessaire.
* **Situations :** Gestion de la mémoire, annulation d'opérations, évaluation d'expressions, etc.

**Les Files**

* **Usage :** Pour un accès FIFO (First In First Out).
* **Situations :** Gestion des tâches en file d'attente, algorithmes de parcours en largeur, etc.

**Les Deques (Double-Ended Queues)**

* **Usage :** Quand l'accès rapide aux deux extrémités de la structure est nécessaire.
* **Situations :** File d'attente à deux extrémités, algorithmes de parcours en largeur avec besoins d'insérer/supprimer aux deux extrémités.

**Les Arbres**

1. **Arbre Binaire :**
   * **Usage :** Recherche efficace, tri binaire.
   * **Situations :** Arbres de recherche binaire, arbres d'expression.
2. **Arbre Binaire de Recherche (BST) :**
   * **Usage :** Recherche, insertion et suppression rapides.
   * **Situations :** Dictionnaires, bases de données, arbres de recherche.
3. **Arbre AVL ou Rouge-Noir :**
   * **Usage :** Optimisation de la hauteur de l'arbre pour des opérations rapides.
   * **Situations :** Applications où l'équilibrage est critique.

**Les Dictionnaires**

* **Usage :** Stockage d'éléments avec une clé associée pour un accès rapide.
* **Situations :** Recherche rapide, base de données, gestion d'associations clé-valeur.

**Réduire la Complexité Algorithmique**

1. **Choix de la Structure :**
   * Sélectionne une structure adaptée aux opérations fréquentes dans ton algorithme.
2. **Complexité Temporelle :**
   * Choisis des structures qui minimisent les coûts des opérations les plus fréquentes.
3. **Espace :**
   * Évalue les besoins en espace, privilégie des structures plus complexes si elles réduisent la complexité temporelle.
4. **Optimisation :**
   * Utilise des structures de données spécialisées pour des problèmes spécifiques (ex : heaps, trie).
5. **Équilibrage :**
   * Pour les arbres, considère l'équilibrage pour assurer des performances cohérentes.

En résumé, le choix de la structure de données dépend des opérations fréquentes dans ton programme et de la complexité temporelle et spatiale que tu es prêt à tolérer. Choisir la bonne structure est souvent crucial pour optimiser les performances d'un algorithme.

Haut du formulaire

**Conversion d’un chiffre n en une autre base b :**

While n>0 :

n = n//b (quotient)While n>0 :

r = n%b (reste)

l.append(r)

l.reverse()