Oliver Searle 22210680

Compte rendu TP2 Algorithme de Shunting-Yard

Description des Algorithmes et Analyse de Complexité

1. Algorithme de Shunting Yard

L'algorithme de Shunting Yard transforme une expression infixe en notation postfixe. Il utilise une file pour la sortie et une pile pour gérer les opérateurs et les parenthèses. Chaque élément est traité séquentiellement donc cet algorithme a une complexité en temps et en espace de O(n).

- **Nombre** : Inséré directement dans la file de sortie.
- **Opérateur** : Comparé avec le sommet de la pile pour déterminer la priorité ; des opérateurs sont déchargés dans la sortie jusqu'à rencontrer un opérateur de priorité plus basse ou une parenthèse ouvrante.
- **Parenthèses** : Une parenthèse ouvrante est empilée, et une fermante déclenche le déchargement des opérateurs dans la sortie jusqu'à rencontrer une ouvrante.

2. Évaluation de l'Expression Postfixe

L'évaluation de l'expression postfixe est réalisée en parcourant et traitant chaque élément une seule fois et a donc également une complexité en temps et en espace de O(n).

- **Nombre** : Empilé directement dans la pile de calcul.
- **Opérateur** : Les deux derniers nombres de la pile sont désempilés, l'opération est effectuée, et le résultat est empilé. À la fin, le sommet de la pile contient le résultat final.

3. Algorithmes auxiliaires

Les fonctions auxiliaires, telles que queue_push, queue_pop, stack_push, et stack_pop, sont implémentées pour gérer les éléments dans une pile ou une file de manière dynamique ont tous une complexité en temps de O(1) pour chaque appel et une complexité en espace variable.

Complexité Totale

Pour un calcul d'expression de taille nnn :

- **Temps** : O(n) pour la conversion en postfixe, O(n) pour l'évaluation, soit une complexité totale en temps de O(n).
- **Espace** : O(n) pour la mémoire nécessaire à la pile et à la file dans chaque étape.

Synthèse de la démarche de programmation mise en œuvre

1. **Gestion de getline et Pointeur buf**
   Initialement, j'ai eu des problèmes pour gérer le pointeur buf de getline en raison de la mémoire dynamique et comment après faire l'affichage avec printf. Après avoir consulté des documentations en ligne et un exemple sur Stack Overflow, j'ai compris comment l'utiliser correctement et éviter les erreurs de gestion de mémoire.
2. **Erreurs de Segmentation**
   J'ai rencontré beaucoup d'erreurs de segmentation causées par des conditions insuffisantes dans certaines structures de contrôle. J'ai ajouté des vérifications de validité des pointeurs dans les if, ce qui a résolu ces erreurs.
3. **Gestion des Opérateurs et Espacement**
   Pour les opérateurs comme la puissance, j'ai constaté que l'absence d'espaces entre les opérandes causait des erreurs.
4. **Problèmes de Typage dans delete_token et delete_queue**
   Les signatures de delete_token et delete_queue ont nécessité une adaptation pour éviter des erreurs de typage, en utilisant des conversions explicites là où c'était nécessaire pour garantir la compatibilité.
5. **Boucle Infinie dans evaluateExpression**
   Un problème persiste avec evaluateExpression, où la boucle continue une fois que la file postfix est vide et que la fonction queue_empty devrait retourner false. Des vérifications de taille de file et des tests supplémentaires n'ont pas complètement résolu ce point, ce qui nécessite encore du débogage. J'ai intégré des affichages pour suivre l'état de la pile et de la file pendant l'évaluation, mais je n'ai toujours pas trouvé la source du problème.

En conclusion, ce projet m'a permis de résoudre plusieurs problèmes liés à la mémoire, aux opérateurs et aux structures de données en C, bien qu'il reste à ajuster le comportement de evaluateExpression pour une évaluation entièrement stable.