TD : Listes chaînées en C

Contexte & objectifs

Vous allez implémenter, from scratch, des listes chaînées en C, en respectant des conventions de nommage en anglais et en intégrant l'analyse de complexité dans chaque fonction (en commentaire). À la fin du TD, vous aurez construit :

1. une SList (singly linked list) avec head ;
2. une variante SList (head + tail) ;
3. une DList (doubly linked list) ;
4. un exemple applicatif MRU (Most Recently Used).

Contraintes générales

- Code en C standard (C11 recommandé), sans bibliothèques exotiques.
- Noms en anglais (ex. SList, SNode, insert_head, remove_value, find, …).
- Préfixe de "namespacing" recommandé : ds_ (ex. ds_slist_insert_head).
- Gestion mémoire stricte (malloc/free) et pas de fuites.
- Chaque fonction comporte un commentaire indiquant Time/Space complexity.
- Fournir un petit jeu de tests (main.c) à chaque étape.

1) Partie A : SList (head only)

Spécification

Types :

```
typedef struct SNode { int data; struct SNode *next; } SNode;
typedef struct SList { SNode *head; } SList;
```

API minimale (fichier slist.h) :

```
void  ds_slist_init(SList *list);
int   ds_slist_is_empty(const SList *list);
int   ds_slist_insert_head(SList *list, int value);
int   ds_slist_insert_tail(SList *list, int value); /* O(n) en
head-only */
int   ds_slist_insert_at(SList *list, int index, int value);
int   ds_slist_remove_head(SList *list, int *out);
int   ds_slist_remove_tail(SList *list, int *out); /* O(n) */
int   ds_slist_remove_value(SList *list, int value);
SNode* ds_slist_find(const SList *list, int value);
void  ds_slist_print(const SList *list);
void  ds_slist_clear(SList *list);
```

Exigences

- insert_head/remove_head en O(1).
- Sans tail, insert_tail et remove_tail sont en O(n).
- insert_at indices 0..length, sinon échec.
- remove_value ne supprime que la première occurrence.
- print au format [a -> b -> c]\n.

Tests à écrire (exemples)

- Insertion tête/queue, index 0/milieu/fin, index hors bornes.
- Suppression tête/queue/valeur absente.
- find présent/absent.
- clear idempotent.

(Option bonus : ajouter, en commentaires, des versions récursives de find, insert_at, remove_value, clear + mention « O(n) stack depth ».)

2) Partie B : SList (head & tail)

Objectif : Étendre SList pour maintenir aussi un tail, sans casser l'API de base (ou en ajoutant les fonctions nécessaires).

Spécification

```
typedef struct SList { SNode *head; SNode *tail; } SList;
```

Ajustements attendus :

- ds_slist_insert_tail doit passer à O(1).
- Mettre à jour correctement tail lors de : insert_head, remove_head, remove_tail, remove_value (si on retire l'ancien tail).

Complexités attendues :

- insert_head, remove_head → O(1)
- insert_tail → O(1)
- remove_tail → O(n) (toujours, en simple chaînage)

Tests

- Cas limites : liste vide, 1 élément.
- Mise à jour correcte de tail après insert/remove.
- Suppression du dernier élément : head et tail deviennent NULL.

3) Partie C : DList (doubly linked list)

Spécification

```
typedef struct DNode { int data; struct DNode *prev, *next; }
DNode;
typedef struct DList { DNode *head; DNode *tail; } DList;
```

API minimale (fichier dlist.h) :

```
void  ds_dlist_init(DList *list);
int   ds_dlist_is_empty(const DList *list);
int   ds_dlist_insert_head(DList *list, int value);
int   ds_dlist_insert_tail(DList *list, int value);
int   ds_dlist_remove_head(DList *list, int *out);
int   ds_dlist_remove_tail(DList *list, int *out); /* désormais
O(1) */
DNode* ds_dlist_find(const DList *list, int value);
int   ds_dlist_remove_value(DList *list, int value); /* O(n),
suppression locale en O(1) */
void  ds_dlist_print_forward(const DList *list);
void  ds_dlist_print_backward(const DList *list);
void  ds_dlist_clear(DList *list);
```

Points clés

- Mise à jour symétrique de prev/next et de head/tail.
- remove_tail passe en O(1) (avantage de la DList).
- print_backward valide la cohérence des pointeurs prev.

Tests

- Suppression au milieu (re-chaînage correct).
- Impression avant/arrière.
- Liste vide / 1 élément.

4) Partie D : Exemple applicatif : MRU (Most Recently Used)

Spécification fonctionnelle

But : maintenir une liste des éléments les plus récemment utilisés (IDs int). Opérations : mru_access (move-to-front + possible éviction), mru_remove (supprime la première occurrence), mru_print.

Implémentation

Version acceptée : basée sur SList (head + tail). mru_access utilise : find, remove_value, insert_head, remove_tail. Complexité : O(n). (Note : pour ~O(1), utiliser une DList + table de hachage.)

Tests

- Accès répétés à un même ID (move-to-front).
- Dépassement de capacité → éviction correcte (tail).
- Suppression d'un ID présent/absent.
- Invariants size/capacity respectés.

5) Livrables & organisation

Arborescence suggérée

```
/td-lists/
  slist.h
  slist.c
  dlist.h
  dlist.c
  mru.h
  mru.c
  main.c        // vos tests (console)
  Makefile
  README.md     // brève doc: API, complexités, choix de
conception
```

README attendu (très court)

- API résumée (SList, DList, MRU).
- Complexités par fonction (tableau).
- Décisions notables (gestion des cas limites, invariants, MAJ de tail).
- Commandes de build/exec.

6) Conseils & pièges fréquents

- Toujours vérifier malloc ; ne jamais utiliser un nœud après free.
- Bien mettre à jour tail (suppression du dernier élément, liste qui devient vide).
- Dans DList, ne pas oublier les deux liens prev/next à chaque insertion/suppression.
- insert_at : gérer index == 0 (équivaut à insert_head) et index == length.
- remove_value : ne retirer que la première occurrence.
- print : vide → [].

Point de passage : chemin pédagogique

5. SList (head) : comprendre le chaînage et les opérations de base.
6. SList (head+tail) : gagner O(1) sur insert_tail, maîtriser les invariants head/tail.
7. DList : supprimer tail en O(1), naviguer avant/arrière, re-chaînage local.
8. MRU : application réaliste, move-to-front, gestion de capacité, complexité métier.

7) Partie E Exercice : Polynômes sur listes chaînées (coefficients en double)

Objectif : implémenter un polynôme creux (sparse) avec une liste simplement chaînée triée par degré décroissant. Chaque nœud représente un monôme (coef * x^deg), avec un coefficient en double.

Spécifications

Types :

```
typedef struct Term {
    double coef;  /* coefficient */
    int    deg;   /* degree >= 0 */
    struct Term *next;
} Term;

typedef struct Poly {
    Term *head;   /* trié par degré décroissant */
    Term *tail;   /* optionnel, utile pour append */
} Poly;
```

API minimale (poly.h) :

```
void   poly_init(Poly *P);
void   poly_clear(Poly *P);                              /* O(n)
*/
int    poly_insert(Poly *P, double c, int d);            /*
insertion triée + fusion, O(n) */
double poly_eval(const Poly *P, double x);               /* O(n)
*/
void   poly_print_ascii(const Poly *P);                  /* ex:
-3*x^2 + 1.5*x - 2 */
```

Règles d'insertion et d'affichage :

- La liste est triée par degré décroissant. À l'insertion :
- - si le degré existe déjà, additionner les coefficients ; si ~0, supprimer le terme (EPS=1e-9).
- - sinon, insérer au bon endroit ; mettre à jour tail si besoin.
- Affichage soigné :
- - pas de '+' initial ; utiliser ' - ' / ' + ' entre termes.
- - ne pas afficher '1x' (afficher 'x') ni 'x^1' (afficher 'x').
- - utiliser x^k en ASCII.
- - format des doubles compact (ex: 1.5, 2, 2.75).

Travail demandé :

9. Implémenter poly_init, poly_clear, poly_insert (tri + fusion + suppression ~0), poly_eval.
10. Implémenter poly_print_ascii.
11. Écrire des tests (main.c) : insertion, fusion, suppression de ~0, évaluation P(0), P(1), P(2), affichages.
12. Expliquer la complexité de chaque fonction (en commentaire au-dessus).

Extensions (optionnelles) :

- poly_add(P, Q, R) : fusion de deux polynômes triés (O(n+m)).
- poly_derivative(P, R) : dérivation terme à terme (O(n)).
- poly_multiply(P, Q, R) : multiplication naïve O(n·m) avec insertions fusionnantes.
- Ajout d'un mode d'affichage sans symbole '*' en ASCII (style math pur).

Jeu de tests minimal :

- Construire P avec (-3,2), (1.5,1), (-2,0), (1,2) → fusion en (-2,2) + 1.5x - 2.
- Vérifier les trois affichages : ASCII, Unicode, LaTeX.
- Évaluer P(2) et comparer au calcul attendu.
- Insertion d'un terme annulant (coef ≈ -terme existant) → terme supprimé.
- Polynôme vide (head=NULL) → affichage 0.

Complexité attendue :

- poly_insert : O(n) temps, O(1) espace additionnel.
- poly_eval : O(n) temps, O(1) espace.
- poly_clear : O(n) temps, O(1) espace.

9) De la List vers Stack et Queue

9.1) De SList vers Stack (LIFO)

Réutilisez SList pour construire une pile (Stack) LIFO : utilisez la tête (head) pour obtenir des opérations en O(1).

- push(x) → ds_slist_insert_head(&list, x) // O(1)
- pop(&y) → ds_slist_remove_head(&list, &y) // O(1)
- top(&y) → y = list.head->data (si non vide) // O(1)
- is_empty → ds_slist_is_empty(&list) // O(1)

En SList, insérer/supprimer au head est naturellement O(1) — parfait pour une pile.

9.2) De SList (head+tail) vers Queue (FIFO)

Réutilisez SList avec head+tail pour une file FIFO : insérer en queue (enqueue) et retirer en tête (dequeue) sont toutes deux en O(1).

- enqueue(x) → ds_slist_insert_tail(&list, x) // O(1) avec tail
- dequeue(&y) → ds_slist_remove_head(&list, &y) // O(1)

- front(&y) → y = list.head->data (si non vide) // O(1)
- is_empty → ds_slist_is_empty(&list) // O(1)

Remarque : en SList, remove_tail reste O(n), mais une file n'en a pas besoin (on ne retire jamais en queue). Le pointeur tail rend enqueue O(1).