CORRECTION DU TP3

Listes chainees, piles et files

Objectif pedagogique. Construire des structures de donnees reutilisables, expliciter la propriete des objets alloues et relier la representation choisie a la complexite des operations.

1. Architecture de la correction

La correction est organisee en modules courts. Le type `Cell` est declare une seule fois ; les listes, la pile et la file reutilisent cette cellule. Le stockage reste type : `Cell.data` est un `Elmt_t *`, et non un `void *`.

Module	Responsabilite
elmt_t.h / elmt_t.c	Definit Elmt_t comme int pour le TP3, et fournit allocation, liberation, affichage et comparaison.
cell.h / cell.c	Definit Cell { Elmt_t *data; Cell *next; }. create_cell alloue la cellule ; free_cell ne libere que la cellule.
list.h / list.c	Liste simplement chainee dont la tete est le seul pointeur conserve.
stack.h / stack.c	Pile LIFO representee par une liste ; le sommet correspond a la tete.
balanced.h / balanced.c	Utilise la pile pour controler le bon emboitement de (), [] et {}.
fifo.h / fifo.c	File FIFO munie de deux pointeurs first et last pour des ajouts et retraits en temps constant.

Les trois executables de test sont separes, car chacun contient son propre `main`. Le Makefile compile en C90 et active les avertissements stricts.

Compilation reproductible : `make clean && make test`. Les options sont `-std=c90 -Wall -Wextra -Wpedantic -Werror -O2`.

2. Cellules et liste simplement chainee

Le pointeur `next` relie les cellules. La liste vide est representee par `NULL`. Dans cette correction, une donnee est allouee separement de sa cellule, ce qui permet de distinguer clairement la gestion de la cellule et celle de l'objet stocke.

```
typedef struct _cell {
    Elmt_t *data;
    struct _cell *next;
} Cell;
```

Contrat de propriete

`create_cell(data)` conserve le pointeur donne, sans copier ni liberer la donnee. `free_cell(c)` libere uniquement la cellule. Une liste prend possession d'un objet apres une insertion reussie ; par consequent, `free_list` libere l'objet avec `free_elmt`, puis la cellule avec `free_cell`.

Operations de liste et complexites

Fonction	Strategie	Complexite
list_is_empty	Teste si la tete vaut NULL.	O(1)
create_empty_list	Renvoie NULL.	O(1)
print_list	Visite chaque cellule une fois.	O(n)
insert_first	Alloue une cellule, la relie a l'ancienne tete, puis met a jour la tete.	O(1)
insert_last	Parcourt jusqu'a la cellule dont next vaut NULL, puis ajoute la nouvelle cellule.	O(n)
search	Compare les valeurs dans l'ordre jusqu'a trouver la cle ou atteindre la fin.	O(n)
length	Compte chaque cellule.	O(n)
free_list	Libere chaque objet et chaque cellule, puis remet la tete a NULL.	O(n)

Pourquoi l'allocation est-elle realisee avant la modification de la liste ?

Dans `insert_first` et `insert_last`, la cellule est allouee avant toute modification des liens. Si `malloc` echoue, la fonction renvoie `NULL` et la structure reste inchangee. Le code appelant garde alors la responsabilite de liberer l'objet qui n'a pas ete insere.

La recherche recoit une fonction de comparaison. Cela separe le parcours de la liste de la representation de l'element ; ici, `compare_int` compare les entiers sans soustraction, afin d'eviter un debordement arithmetique.

3. Pile LIFO et parentheses equilibrees

Le sommet de pile est la tete de la liste. Cela fournit une implementation naturelle : empiler ajoute en tete et depiler retire la tete. Aucune operation ne doit parcourir toute la pile.

```
typedef List Stack;

int stack_push(Stack *s, Elmt_t value);
int stack_pop(Stack *s, Elmt_t *value);
```

Operation	Comportement et cout
stack_push	Alloue une copie de la valeur puis l'insere en tete. Temps O(1), en plus de l'allocation.
stack_pop	Copie la valeur du sommet dans la variable de sortie, libere l'objet et la cellule, puis avance la tete. Temps O(1).
stack_clear	Libere toute la pile par free_list. Temps O(n).

Verifier les delimiters

La fonction `balanced` parcourt l'expression une seule fois. A chaque ouverture (parenthese, crochet ou accolade), elle empile le caractere. A chaque fermeture, elle depile une ouverture et verifie que les types correspondent. Une fermeture sans ouverture, un mauvais emboitement ou une pile non vide a la fin signifie que l'expression est mal equilibree.

```
int balanced(const char *expression);

/* Retour : 1 = equilibree, 0 = invalide, -1 = erreur. */
```

La convention de retour distingue une expression mal equilibree d'une erreur d'allocation. Dans les deux cas, la pile temporaire est liberee avant le retour. Les caracteres autres que (), [] et {} sont ignores.

Complexite de balanced

Mesure	Justification
Temps	O(n) : chaque caractere est examine une seule fois ; push et pop sont O(1).
Memoire auxiliaire	O(n) au pire : une expression contenant n ouvertures avant les fermetures peut placer n caracteres dans la pile.

Exemples verifies : `a * (b + c)` et `[(a + b) * {c - d}]` sont valides ; `([)]`, `(a + b` et `a + b)` ne le sont pas.

4. File FIFO optimisee pour le TP4

La file stocke un pointeur vers le premier maillon (`first`) et un vers le dernier (`last`). Les retraits se font a l'avant et les ajouts a l'arriere. La representation ne change pas la politique FIFO ; elle evite seulement de reparcourir la liste a chaque ajout.

```
typedef struct {
    Cell *first;
    Cell *last;
} Fifo;

int is_empty(Fifo f);
Fifo create_fifo(void);
Cell *fifo_add(Fifo *f, Elmt_t *data);
Elmt_t *fifo_get(Fifo *f);
```

Invariant essentiel

La file est vide si et seulement si `first == NULL` et `last == NULL`. Si elle contient au moins une cellule, `first` pointe sur la premiere et `last` sur la derniere ; le champ `last->next` vaut alors `NULL`.

Cas	Mises a jour importantes
Ajout dans une file vide	La cellule creee devient a la fois first et last.
Ajout dans une file non vide	Ancien last->next = nouvelle cellule, puis last = nouvelle cellule.
Retrait dans une file avec plusieurs cellules	Retourner data de first, avancer first vers first->next et liberer l'ancienne cellule.
Retrait du dernier element	Apres avoir avance first, si first vaut NULL, remettre last a NULL egalement.

Complexites

Fonction	Complexite	Raison
is_empty	O(1)	Un test de pointeur.
fifo_add	O(1)	Le pointeur last donne directement le point d'ajout.
fifo_get	O(1)	Le pointeur first donne directement le point de retrait.

Un pointeur sur la fin ne rendrait pas le retrait en fin constant : dans une liste simplement chainee, aucun lien ne permet de remonter vers l'avant-dernier maillon. Ici, la file n'a pas besoin de retirer en fin : elle retire toujours en tete.

5. Proprietes memoire, tests et reutilisation

Trois contrats, a ne pas confondre

Structure	Qui possede les objets pointes par data ?	Qui libere quoi ?
Liste	La liste apres insertion reussie.	free_list libere l'objet (free_elmt), puis la cellule (free_cell).
Pile	La pile possede chaque copie creee par stack_push.	stack_pop copie la valeur, puis libere son objet et sa cellule ; stack_clear vide tout.
File	La file possede les cellules, mais pas les objets pointes.	fifo_get libere la cellule et transmet au code appelant le pointeur data. Le code appelant libere ensuite cet objet.

Cette difference est utile au TP4 : la file contiendra des pointeurs vers les noeuds de l'arbre. La file ne doit pas detruire ces noeuds lors d'un retrait, car ils appartiennent a la structure d'arbre. Elle ne detruit que ses cellules internes.

Tests inclus

Executable	Tests principaux
test_list	Liste vide ; insertions en tete et en fin ; ordre ; longueur ; recherche presente et absente ; liberation idempotente d'une liste vide.
test_stack	Empilements et depilements LIFO ; tentative de depiler une pile vide ; plusieurs expressions equilibrees et non equilibrees ; chaine vide ; argument NULL.
test_fifo	File vide ; retrait vide ; ordre FIFO ; ajout intercale ; retrait du dernier element ; remise a NULL des deux pointeurs ; nouvel ajout apres vidage.

Compilation et execution

```
make clean
make test
```

La suite compile avec `-std=c90 -Wall -Wextra -Wpedantic -Werror`. Les trois programmes de test ont ete compiles et executes avec succes, egalement avec AddressSanitizer et UndefinedBehaviorSanitizer. Les erreurs de malloc sont traitees dans le code, mais les tests ne forcent pas artificiellement l'echec des allocations.

Adapter au TP4

Le TP3 definit `typedef int Elmt_t`. Dans le projet du TP4, `Elmt_t` peut devenir un alias du type des noeuds de l'arbre. Les projets etant separes, adaptez l'en-tete `elmt_t.h` et les fonctions specifiques au type. Reutilisez les modules `cell` et `fifo` ; ne redeclarez pas `Cell`, `create_cell` ou `free_cell`.

Fin du rapport - les sources commentees et les tests sont fournis dans l'archive.