PUSH_SWAP

Table des matières

Sujet

Parce que Swap_push, c'est moins naturel

Ce projet vous demande de trier des données dans une pile, en utilisant un set d'instrcutions limité, et avec le moins d'opérations possibles. Pour le réussir, vous devrez manipuler différents algorithmes de tri et choisir la (ou les?) solution la plus appropriée pour un classement optimisé des données.

Introduction

Le projet Push swap est un exercice d'algorithmie simple et efficace : il faut trier de la donnée.

Vous avez à votre disposition un ensemble d'entiers, deux piles et un ensemble d'instructions pour manipuler celles-ci.

Votre but? Ecrire un programme en C nommé push_swap qui calcule et affiche sur la sortie standard le plus petit programme, fait d'instructions du language Push swap, permettent de trier les entiers passés en paramètres.

Facile?

Et bien, c'est ce qu'on va voir...

Objectifs

Ecrire un algorithme de tri est toujours une étape importante dans la vie d'un programmeur débutant car il s'agit souvent de la première rencontre avec la notion de complexité.

Les algorithmes de tri et leur complexité font parti des grands classiques des entretiens d'embauche. C'est donc l'occasion rêvée pour vous pencger sérieusement sur la question car soyez certains que cela vous sera demandé.

Les objectifs de ce projet sont la rigueur, la pratique en C et l'usage d'algorithmes élémentaire. En particulier, la complexité de ces algorithmes élémentaire.

Trier des valeurs c'est simple. Les trier le plus vite possible, c'est moins simple vu que, d'une configuration des entiers à trier à une autre, un même algorithme de tri n'est pas forcément le plus efficace...

Partie obligatoire

Les règles

-Le jeu est constitué de 2 piles nommées a et b.

-Au départ :
.La pile a contient une quantité aléatoire de négatif et/ou des nombres positifs qui ne peuvent pas être dupliqués.
.La pile b est vide.

-Le but du jeu est de trier les nombres de la pile a par ordre croissant. Pour ce faire, vous disposez des instructions suivantes :

.sa (swap a) : Intervertit les 2 premiers éléments au sommet de la plie a.

.sb (swap b) : Intervertit les 2 premiers éléments au sommet de la pile b.
Ne fait rien s'il n'y en a qu'un ou aucun.
.ss : sa et sb en même temps.

.pa (push a) : Prend le premier élément au sommet de b et le met sur a.

.pb (push b) : Prend le premier élément zu sommet de a et le met sur b.

.ra (rotate a) : Décale d'une position vers le haut tous les éléments de la pile a.

.rb (rotate b) : Décale d'une position vers le haut tous les éléments de la pile b.

.rr : ra et rb en même temps.

.rra (reverse rotate a) : Décale d'une position vers le bas tous les éléments de la pile a. Le dernier élément devient le premier.

.rrb (reverse rotate b) : Décale d'une position vers le bas tous éléments de la pile b. Le dernier élément devient le premier.

.rrr : rra et rrb en même temps.

Le programme "PUSH_SWAP"

-Nom du programme : push_swap

-Fichiers de rendu : Makefile, *.h, *.c

-Makefile : NAME, all, clean, fclean, re

-Arguments : pile a : une liste d'entiers

-Fonctions externes autorisées :
.read, write, malloc, free, exit
.ft_printf et tout équivalent que vous avez codé

-Libft autorisée

-Description : trier les piles

Votre projet doit respecter les règles suivantes :

-Vous devez rendre un Makefile qui compilera vos fichiers sources. il ne doit pas relink.

-Les variables globales sont interdites.

-Vous devez écrire un programme nommé push_swap qui prend en paramètre la pile a sous la forme d'une liste d'entiers. Le premier paramètre est au sommet de la pile (attention donc à l'ordre).

-Le programme doit afficher un programme composé de la plus courte suite d'instructions possible qui permet de trier la pile a, le plus petit nombre étant au sommet de la pile.

-les instructions doivent être séparées par un '\n' et rien d'autre.

-Le but est de trier les entiers avec le moins d'opérations possible. En évaluation, le nombre d'instructions calculé par votre programme sera comparé avec un nombre d'opérations maximum toléré. Si votre programme sort un programme trop long, ou si la liste d'entiers n'est pas triée, vous aurez 0.

-Si aucun paramètre n'est spécifié, le programme ne doit rien afficher et rendre l'invite de commande.

-En cas d'erreur, vous de vez afficher "Error" suivi d'un '\n' sur la sortie d'erreur. Par exemple, si certains paramètres ne sont pas des nombres, ne tiennent pas dans un int, ou encore, s'il y a des doublons.

Pendant l'évaluations, un binaire sera fourni afin de tester votre programme correctement.
Il fonctionne ainsi :

```
ARG="4 67 3 87 23"; ./push_swap $ARG | wc -1
          6
ARG="4 67 3 87 23"; ./push_swap $ARG | ./checker_OS $ARG
OK
```

Si le programme checker_OS affiche "KO", cela signifie que votre push_swap calcule un programme qui ne trie pas la liste.

PARTIE BONUS

Ce projet se prête peu à la création de bonus de par sa simplicité. Cependant, que diriez-vous de créer votre propre checker?
Grâce au programme checker, vous allez pouvoir vérifier que la liste d'instructions générée par le programme push_swap trie bien la pile passée en paramètre.

Le programme "checker"

-Nom du programme : checker

-Fichiers de rendu : *.h, *.c

-Makefile : bonus

-Arguments : pile a : une liste d'entiers

-Fonctions externes autorisées :
.read, write, malloc, free, exit
.ft_printf et tout équivalent que vous avez codé

-Libft autorisée

-Description : Exécuter les instructions de tri

.Vous devez écrire un programme nommé checker qui prend en paramètre la pile a sous la forme d'une liste d'entiers. Le premier paramètre est au sommet de la pile (attention donc à l'ordre). Si aucun argument n'est donné, le programme s'arrête et n'affiche rien.

.Il doit ensuite attendre et lire des instructions sur l'entrée standard, chaque instruction suivie par un '\n'. Une fois toutes les instructions lues, le programme va les exécuter sur la pile d'entiers passée en paramètre.

.Si à la suite de l'exécution la pile a est bien triée et la pile b est vide, alors le programme doit afficher "OK" suivi suivi un '\n' sur la sortie standard.

.Sinon, il doit afficher "KO" suivi par un '\n' sur la sortie standard.

.En cas d'erreur, vous devez afficher "Error" suivi d'un '\n' sur la sortie d'erreur. Par exemple, si certains paramètres ne sont pas des nombres, ne tiennent pas dans un int, s 'il y a des doublons ou, bien sûr, si une instruction n'existe pas ou est mal formatée.

Vous n'avez pas à reproduire exactement le même comportement que le binaire qui est fourni. Il est obligatoire de gérer les erreurs mais c'est à vous de décider comment vous souhaiter analyser les arguments.

Push_swap.h

```c
#ifndef PUSH_SWAP_H
# define PUSH_SWAP_H

# include <stddef.h>
# include <stdlib.h>
# include <unistd.h>
# include <limits.h>
# include <stdio.h>

typedef struct s_stack
{
    int             value;
    int             index;
    int             pos;
    int             target_pos;
    int             cost_a;
    int             cost_b;
    struct s_stack  *next;
}   t_stack;

/* InitialiSation */
t_stack     *fill_stack_values(int argc, char **argv);
void        assign_index(t_stack *stack_a, int argc);

/* Algorithmes de tri */
int         is_sorted(t_stack *stack);
void        tiny_sort(t_stack **stack);
void        sort(t_stack **stack_a, t_stack **stack_b);

/* Position */
int         get_lowest_index_position(t_stack **stack);
void        get_target_position(t_stack **stack_a, t_stack **b);

/* Cost */
void        get_cost(t_stack **stack_a, t_stack **stack_b);
void        do_cheapest_move(t_stack **stack_a, t_stack **stack_b);

/* Calculate Move */
void        do_move(t_stack **a, t_stack **b, int cost_a, int cost_b);

/* Operations */
void        do_pa(t_stack **stack_a, t_stack **stack_b);
void        do_pb(t_stack **stack_a, t_stack **stack_b);
```

```
void        do_sa(t_stack **stack_a);
void        do_sb(t_stack **stack_b);
void        do_ss(t_stack **stack_a, t_stack **stack_b);
void        do_ra(t_stack **stack_a);
void        do_rb(t_stack **stack_b);
void        do_rr(t_stack **stack_a, t_stack **stack_b);
void        do_rra(t_stack **stack_a);
void        do_rrb(t_stack **stack_b);
void        do_rrr(t_stack **stack_a, t_stack **stack_b);

/* Stack Functions */
t_stack     *get_stack_bottom(t_stack *stack);
t_stack     *get_stack_before_bottom(t_stack *stack);
t_stack     *stack_new(int value);
void        stack_add_bottom(t_stack **stack, t_stack *new);
int         get_stack_size(t_stack *stack);

/* Utils */
void        free_stack(t_stack **stack);
long int    ft_atoi(const char *str);
void        ft_putstr(char *str);
int         nb_abs(int nb);

/* Error */
void        exit_error(t_stack **stack_a, t_stack **stack_b);

/* Input Check */
int         is_correct_input(char **argv);
int         is_digit(char c);
int         is_sign(char c);
int         nbstr_cmp(const char *s1, const char *s2);

#endif
```

Votre fichier d'en-tête, push_swap.h, définit une variété d'opérations et de fonctions d'aide que vous utiliserez pour un programme push_swap. Votre programme sera probablement chargé de trier une pile d'entiers en utilisant le moins d'opérations possibles. Voici un aperçu des fonctions définies dans votre fichier d'en-tête :

Fonctions d'initialisation :

fill_stack_values :
Remplit une pile avec des valeurs entières provenant des arguments de ligne de commande.
assign_index : Cette fonction n'est pas définie dans l'extrait de code que vous avez fourni, mais elle attribue probablement des indices à chaque nœud de la pile.
Algorithmes de tri :

is_sorted : Détermine si la pile est triée.
tiny_sort : Probablement une simple fonction de tri pour les petites entrées.
sort : Trie la pile, probablement en utilisant un algorithme plus complexe.
Fonctions de position :

get_lowest_index_position : Trouve la position du nœud avec l'indice le plus bas.
get_target_position : Trouve les positions cibles dans les piles.
Fonctions de coût :

get_cost : Calcule probablement le coût d'un mouvement potentiel (combien d'opérations cela prendrait).
do_cheapest_move : Effectue le mouvement avec le coût le plus bas.
Fonctions de mouvement :

do_move : Effectue un mouvement sur les piles en fonction d'un coût donné.
Fonctions d'opération :

do_pa, do_pb, do_sa, do_sb, do_ss, do_ra, do_rb, do_rr, do_rrr, do_rra, do_rrb : Ces fonctions représentent probablement les opérations fondamentales de votre programme push_swap, effectuant diverses rotations et échanges sur les piles.
Fonctions de pile :

get_stack_bottom : Retourne le nœud du bas de la pile.
get_stack_before_bottom : Retourne le nœud avant-dernier de la pile.
stack_new : Crée un nouveau nœud de pile avec une valeur donnée.
stack_add_bottom : Ajoute un nouveau nœud au bas de la pile.
get_stack_size : Retourne la taille de la pile.
Fonctions utilitaires :

free_stack : Libère la mémoire associée à la pile.
ft_atoi : Convertit une chaîne en entier long.
ft_putstr : Imprime une chaîne à la console.
nb_abs : Retourne la valeur absolue d'un entier.
Fonction d'erreur :

exit_error : Nettoie la mémoire et quitte le programme en cas d'erreur.
Fonctions de vérification de l'entrée :

is_correct_input : Vérifie si les arguments de ligne de commande sont corrects.
is_digit : Vérifie si un caractère est un chiffre.
is_sign : Vérifie si un caractère est un signe plus ou moins.
nbstr_cmp : Compare deux chaînes de chiffres.
Dans l'ensemble, votre programme push_swap utilisera ces opérations et fonctions d'aide pour trier une pile d'entiers avec le moins d'opérations possibles, en utilisant les concepts de 'coût' pour décider quelle opération effectuer ensuite.

Main.c

```c
#include "push_swap.h"

/* est_trié :
* Vérifie si une pile est triée.
* Renvoie 0 si la pile n'est pas triée, 1 si elle est triée.
*/
int is_sorted(t_stack *stack)
{
    while (stack->next != NULL)
    {
        if (stack->value > stack->next->value)
            return (0);
        stack = stack->next;
    }
    return (1);
}
```

```c
/* push_swap :
* Choisit une méthode de tri en fonction du nombre
* des valeurs à trier.
*/
static void push_swap(t_stack **stack_a, t_stack **stack_b, int stack_size)
{
    if (stack_size == 2 && !is_sorted(*stack_a))
        do_ra(stack_a);
    else if (stack_size == 3)
        tiny_sort(stack_a);
    else if (stack_size > 3 && !is_sorted(*stack_a))
        sort(stack_a, stack_b);
}


/* main:
* Vérifie si l'entrée est correcte, auquel cas il initialise les piles a et b,
* attribue à chaque valeur des index et trie les piles. Lorsque le tri est terminé, libère
* les piles et les sorties.
*/
int main(int argc, char **argv)
{
    t_stack *stack_a;
    t_stack *stack_b;
    int     stack_size;

    if (argc < 2)
        return (0);
    if (!is_correct_input(argv))
        exit_error(NULL, NULL);
    stack_b = NULL;
    stack_a = fill_stack_values(argc, argv);
    stack_size = get_stack_size(stack_a);
    assign_index(stack_a, stack_size + 1);
    push_swap(&stack_a, &stack_b, stack_size);
    free_stack(&stack_a);
    free_stack(&stack_b);

    return (0);
}
```

Int is_sorted(t_stack *stack)

La fonction is_sorted vérifie si une pile est triée ou non. Elle prend en entrée un pointeur vers le premier élément d'une pile. La pile est parcourue, et à chaque itération, elle vérifie si l'élément actuel est plus grand que l'élément suivant. Si c'est le cas, elle retourne 0 pour indiquer que la pile n'est pas triée. Si elle parcourt toute la pile sans trouver d'élément plus grand que son suivant, alors elle retourne 1 pour indiquer que la pile est triée.

Static void push_swap(t_stack **stack_a, t_stack **stack_b, int stack_size)

La fonction push_swap choisit une méthode de tri en fonction de la taille de la pile. Elle prend trois paramètres : deux pointeurs vers les piles à trier, et la taille de la pile. Si la pile a deux éléments et n'est pas triée, elle effectue une opération do_sa qui échange les deux premiers éléments de la pile. Si la pile a trois éléments, elle utilise la fonction tiny_sort pour la trier. Si la pile a plus de trois éléments et n'est pas triée, elle utilise la fonction sort.

Int main(int argc, char **argv)

Enfin, la fonction main est le point d'entrée du programme. Elle prend en entrée le nombre d'arguments et un tableau des arguments passés au programme. Si le nombre d'arguments est inférieur à 2, la fonction se termine immédiatement et retourne 0. Sinon, elle vérifie si les arguments sont valides avec la fonction is_correct_input. Si les arguments ne sont pas valides, elle appelle la fonction exit_error pour arrêter le programme avec une erreur. Sinon, elle initialise les piles, les remplit avec les valeurs passées en arguments, assigne un index à chaque valeur dans la pile, trie les piles, puis libère la mémoire allouée aux piles avant de quitter le programme.

Initialization.c

```c
#include "push_swap.h"

/* fill_stack_values :
* Remplit stack_a avec les valeurs fournies.
* Si les valeurs sont hors de la plage d'entiers, imprime une erreur et quitte le
programme.
*/
t_stack *fill_stack_values(int argc, char **argv)
{
    t_stack     *stack_a;
    long int    nb;
    int         i;

    stack_a = NULL;
    nb = 0;
    i = 1;
    while (i < argc)
    {
        nb = ft_atoi(argv[i]);
        if (nb > INT_MAX || nb < INT_MIN)
            exit_error(&stack_a, NULL);
        if (i == 1)
            stack_a = stack_new((int)nb);
        else
            stack_add_bottom(&stack_a, stack_new((int)nb));
        i++;
    }
    return (stack_a);
}

/* assign_index :
* Attribue un index à chaque valeur dans la pile a. C'est un moyen pratique de commander
* la pile car les index peuvent être vérifiés et comparés au lieu des valeurs réelles,
* qui peuvent ou non être adjacents les uns aux autres.
*           ex. valeurs : -3 0 9 2
* indices : [1] [2] [4] [3]
* Les index sont attribués du plus élevé (stack_size) au plus bas (1).
*/
void    assign_index(t_stack *stack_a, int stack_size)
{
    t_stack *ptr;
    t_stack *highest;
```

```
    int     value;

    while (--stack_size > 0)
    {
        ptr = stack_a;
        value = INT_MIN;
        highest = NULL;
        while (ptr)
        {
            if (ptr->value == INT_MIN && ptr->index == 0)
                ptr->index = 1;
            if (ptr->value > value && ptr->index == 0)
            {
                value = ptr->value;
                highest = ptr;
                ptr = stack_a;
            }
            else
                ptr = ptr->next;
        }
        if (highest != NULL)
            highest->index = stack_size;
    }
}
```

T_stack *fill_stack_values(int argc, char **argv)

t_stack *stack_a; long int nb; int i; : Déclare trois variables : un pointeur vers une pile stack_a, un nombre entier long nb et un entier i.
stack_a = NULL; nb = 0; i = 1; : Initialise stack_a à NULL, nb à 0 et i à 1.
while (i < ac) : Démarre une boucle qui parcourt tous les arguments de la ligne de commande (hormis le nom du programme lui-même).
nb = ft_atoi(av[i]); : Convertit l'argument courant en un nombre entier long.
if (nb > INT_MAX || nb < INT_MIN) exit_error(&stack_a, NULL); : Si le nombre est hors des limites de l'entier, la fonction exit_error est appelée pour afficher une erreur et quitter le programme.
if (i == 1) stack_a = stack_new((int)nb); : Si nous sommes au premier argument, crée un nouveau nœud de pile avec cette valeur.
else stack_add_bottom(&stack_a, stack_new((int)nb)); : Sinon, ajoute un nouveau nœud avec cette valeur au bas de la pile.
i++; : Incrémente i pour passer à l'argument suivant.
return (stack_a); : À la fin de la boucle, renvoie la pile créée.

Void assign_index(t_stack *stack_a, int stack_size)

t_stack *ptr; t_stack *highest; int value; : Déclare trois variables : deux pointeurs vers des piles (ptr et highest) et un entier value.
while (--stack_size > 0) : Démarre une boucle qui exécute le code suivant stack_size - 1 fois.
ptr = stack_a; value = INT_MIN; highest = NULL; : Initialise ptr à la tête de la pile, value à la plus petite valeur possible pour un entier, et highest à NULL.
while (ptr) : Démarre une boucle qui parcourt tous les nœuds de la pile.
if (ptr->value == INT_MIN && ptr->index == 0) ptr->index = 1; : Si le nœud actuel a la valeur la plus basse et que son indice n'est pas encore défini, définit son indice à 1.
if (ptr->value > value && ptr->index == 0) : Si la valeur du nœud actuel est plus grande que la valeur maximale trouvée jusqu'à présent et que son indice n'est pas encore défini, ce nœud est potentiellement le prochain nœud le plus grand sans indice.

value = ptr->value; highest = ptr; ptr = stack_a; : Met à jour la valeur maximale trouvée jusqu'à présent, enregistre le nœud actuel comme le plus grand trouvé jusqu'à présent, et réinitialise ptr au début de la pile pour vérifier s'il existe d'autres nœuds non indexés avec la même valeur.
else ptr = ptr->next; : Si le nœud actuel n'est pas un nouveau maximum, passe au nœud suivant.
if (highest != NULL) highest->index = stack_size; : Si un nœud le plus grand a été trouvé lors de ce passage à travers la pile, attribue à ce nœud l'indice courant stack_size.
En fin de compte, la fonction assign_index attribue à chaque nœud de la pile un indice unique correspondant à sa position dans l'ordre trié, ce qui facilite les comparaisons ultérieures. La fonction attribue l'index le plus élevé stack_size à la valeur la plus grande et va décrémenté jusqu'à 1 passant les éléments déjà indexés.

Input_check.c

```c
#include "push_swap.h"

/* arg_is_number :
* Vérifie si l'argument est un nombre. +1 1 et -1 sont tous des nombres valides.
* Retourne : 1 si l'argument est un nombre, 0 sinon.
*/
static int  arg_is_number(char *argv)
{
    int i;

    i = 0;
    if (is_sign(argv[i]) && argv[i + 1] != '\0')
        i++;
    while (argv[i] && is_digit(argv[i]))
        i++;
    if (argv[i] != '\0' && !is_digit(argv[i]))
        return (0);
    return (1);
}

/* ont_doublons :
* Vérifie si la liste d'arguments contient des numéros en double.
* Renvoie : 1 si un doublon est trouvé, 0 s'il n'y en a pas.
*/
static int  have_duplicates(char **argv)
{
    int i;
    int j;

    i = 1;
    while (argv[i])
    {
        j = 1;
        while (argv[j])
        {
            if (j != i && nbstr_cmp(argv[i], argv[j]) == 0)
                return (1);
            j++;
        }
        i++;
    }
```

```
    return (0);
}


/* arg_is_zero :
* Vérifie que l'argument est un 0 pour éviter les doublons 0 +0 -0
* et 0 0000 +000 -00000000 aussi.
* Retourne : 1 si l'argument est un zéro, 0 s'il contient
* autre chose qu'un zéro.
*/
static int  arg_is_zero(char *argv)
{
    int i;

    i = 0;
    if (is_sign(argv[i]))
        i++;
    while (argv[i] && argv[i] == '0')
        i++;
    if (argv[i] != '\0')
        return (0);
    return (1);
}

/* est_correct_input :
* Vérifie si les arguments donnés sont tous des nombres, sans doublons.
* Retourne : 1 si les arguments sont valides, 0 sinon.
*/
int is_correct_input(char **argv)
{
    int i;
    int nb_zeros;

    nb_zeros = 0;
    i = 1;
    while (argv[i])
    {
        if (!arg_is_number(argv[i]))
            return (0);
        nb_zeros += arg_is_zero(argv[i]);
        i++;
    }
    if (nb_zeros > 1)
        return (0);
    if (have_duplicates(argv))
        return (0);
    return (1);
}
```

Le code que vous avez partagé comprend plusieurs fonctions conçues pour valider les entrées fournies à un programme. Voici une explication ligne par ligne pour chaque fonction :

Static int arg_is_number (char *argv)

Cette fonction vérifie si un argument donné est un nombre. Cela inclut les nombres négatifs et positifs ainsi que le zéro. Elle renvoie 1 si l'argument est un nombre et 0 sinon.

Static int have_duplicates (char **argv)

Cette fonction vérifie si une liste d'arguments contient des doublons. Elle renvoie 1 s'il y a des doublons et 0 sinon.

Static int arg_is_zero(char *argv)

Cette fonction vérifie si un argument est zéro. Cette vérification est un peu plus complexe que simplement vérifier si la valeur est égale à zéro, car elle doit également gérer les cas où l'argument est "+0", "-0" ou "0000". Elle renvoie 1 si l'argument est une représentation de zéro et 0 sinon.

Int is_correct_input(char **argv)

Cette fonction est une fonction de haut niveau qui utilise les fonctions précédentes pour vérifier si une liste d'arguments est valide. Elle vérifie d'abord si tous les arguments sont des nombres. Si ce n'est pas le cas, elle renvoie immédiatement 0. Ensuite, elle compte combien d'arguments sont zéro. Si plus d'un argument est zéro, elle renvoie 0, car cela serait considéré comme des doublons. Enfin, elle vérifie si la liste d'arguments contient des doublons en utilisant la fonction have_duplicates. Si des doublons sont trouvés, elle renvoie 0. Si toutes ces vérifications passent, la fonction renvoie 1, indiquant que la liste d'arguments est valide.

Note : Les fonctions is_sign, is_digit et nbstr_cmp ne sont pas définies dans ce code. is_sign est probablement une fonction qui vérifie si un caractère est un signe positif ou négatif, is_digit vérifie probablement si un caractère est un chiffre, et nbstr_cmp est probablement une fonction qui compare deux chaînes de caractères représentant des nombres.

Stack.c

```c
#include "push_swap.h"

/* get_stack_bottom :
* Renvoie le dernier élément de la pile.
*/
t_stack *get_stack_bottom(t_stack *stack)
{
    while (stack && stack->next != NULL)
        stack = stack->next;
    return (stack);
}

/* get_stack_before_bottom :
* Renvoie l'avant-dernier élément de la pile.
*/
t_stack *get_stack_before_bottom(t_stack *stack)
{
    while (stack && stack->next && stack->next->next != NULL)
        stack = stack->next;
    return (stack);
}

/* stack_new :
* Crée une pile d'éléments avec la valeur fournie.
* Renvoie l'élément de pile nouvellement créé.
```

```c
*/
t_stack *stack_new(int value)
{
    t_stack *new;

    new = malloc(sizeof * new);
    if (!new)
        return (NULL);
    new->value = value;
    new->index = 0;
    new->pos = -1;
    new->target_pos = -1;
    new->cost_a = -1;
    new->cost_b = -1;
    new->next = NULL;
    return (new);
}

/* add_stack_bottom :
* Ajoute un élément au bas d'une pile.
*/
void    stack_add_bottom(t_stack **stack, t_stack *new)
{
    t_stack *tail;

    if (!new)
        return ;
    if (!*stack)
    {
        *stack = new;
        return ;
    }
    tail = get_stack_bottom(*stack);
    tail->next = new;
}

/* get_stack_size :
* Renvoie le nombre d'éléments dans une pile.
*/
int get_stack_size(t_stack *stack)
{
    int size;

    size = 0;
    if (!stack)
        return (0);
    while (stack)
    {
        stack = stack->next;
        size++;
    }
    return (size);
}
```

t_stack *get_stack_bottom(t_stack *stack)

Cette fonction parcourt la pile en passant à l'élément suivant tant que l'élément suivant n'est pas NULL, c'est-à-dire tant que l'on n'a pas atteint le dernier élément de la pile. Elle retourne ensuite le dernier élément de la pile.

t_stack *get_stack_before_bottom(t_stack *stack)

Cette fonction est similaire à la précédente, mais elle s'arrête à l'avant-dernier élément de la pile, c'est-à-dire quand le next de l'élément suivant est NULL. Elle retourne ensuite l'avant-dernier élément de la pile.

t_stack *stack_new(int value)

Cette fonction crée un nouvel élément de pile en allouant de la mémoire pour un élément t_stack. Elle initialise ensuite les valeurs de l'élément. value est le paramètre d'entrée, index est initialisé à 0, pos, target_pos, cost_a, et cost_b sont initialisés à -1, et next est initialisé à NULL. La fonction retourne ensuite l'élément nouvellement créé. Utiliser -1 ou une autre valeur sentinel peut être une bonne pratique de programmation lors de l'initialisation de variables qui n'ont pas encore une valeur significative dans le contexte de votre programme.

void stack_add_bottom(t_stack **stack, t_stack *new)

Cette fonction ajoute un nouvel élément au bas de la pile. Si le nouvel élément est NULL, elle retourne immédiatement. Si la pile est vide (c'est-à-dire si *stack est NULL), elle fait du nouvel élément le premier élément de la pile. Sinon, elle trouve le dernier élément de la pile et fait pointer son champ next vers le nouvel élément.

int get_stack_size(t_stack *stack)

Cette fonction compte le nombre d'éléments dans la pile en parcourant la pile et en incrémentant un compteur size pour chaque élément. Elle retourne ensuite la taille de la pile. Si la pile est vide (c'est-à-dire si stack est NULL), elle retourne immédiatement 0.

Utils.c

```c
#include "push_swap.h"

/* pile_libre :
* Libère chaque élément d'une pile donnée et définit le pointeur de pile sur NULL.
*/
void    free_stack(t_stack **stack)
{
    t_stack *tmp;

    if (!stack || !(*stack))
        return ;
    while (*stack)
    {
        tmp = (*stack)->next;
        free(*stack);
        *stack = tmp;
    }
    *stack = NULL;
}

/* erreur_sortie :
* Écrit "Erreur\n" sur la sortie standard après avoir libéré les piles a et b.
* Sort avec le code d'erreur standard 1.
*/
void    exit_error(t_stack **stack_a, t_stack **stack_b)
```

```c
{
    if (stack_a == NULL || *stack_a != NULL)
        free_stack(stack_a);
    if (stack_b == NULL || *stack_b != NULL)
        free_stack(stack_b);
    write(2, "Error\n", 6);
    exit (1);
}


/* ft_atoi :
* Convertit une chaîne alphanumérique de caractères en un entier long.
*/
long int    ft_atoi(const char *str)
{
    long int    nb;
    int         sign;
    int         i;

    nb = 0;
    sign = 1;
    i = 0;
    if (str[i] == '+')
        i++;
    else if (str[i] == '-')
    {
        sign *= -1;
        i++;
    }
    while (is_digit(str[i]))
    {
        nb = (nb * 10) + (str[i] - '0');
        i++;
    }
    return (nb * sign);
}

void    ft_putstr(char *str)
{
    int i;

    i = 0;
    while (str[i])
    {
        write(1, &str[i], 1);
        i++;
    }
}

/* nb_abs :
* Renvoie la valeur absolue d'un nombre donné.
* La valeur absolue d'un nombre est utilisée pour mesurer la distance de ce
* nombre à partir de 0, qu'il soit positif ou négatif (la valeur abs de -6 est 6).
*/
```

```c
int nb_abs(int nb)
{
    if (nb < 0)
        return (nb * -1);
    return (nb);
}
```

Void free_stack(t_stack **stack)

La fonction free_stack libère chaque élément d'une pile donnée et met le pointeur de la pile à NULL. Elle vérifie d'abord si le pointeur de la pile existe et n'est pas NULL. Ensuite, elle libère chaque élément de la pile en utilisant une boucle qui se répète jusqu'à ce que tous les éléments de la pile aient été libérés.
t_stack *tmp; : Ceci déclare un pointeur temporaire vers t_stack qui sera utilisé plus tard pour garder une trace du nœud suivant dans la pile.
if (!stack || !(*stack)) return ; : Ceci est une vérification de sécurité pour voir si le pointeur de la pile est NULL ou si la pile elle-même est vide (le premier nœud de la pile est NULL). Si l'une ou l'autre de ces conditions est vraie, la fonction se termine immédiatement.
while (*stack) : Ceci est une boucle qui tourne tant qu'il reste des nœuds dans la pile.
tmp = (*stack)->next; : Ici, nous sauvegardons le pointeur vers le prochain nœud de la pile dans la variable tmp avant de libérer le nœud actuel.
free(*stack); : Ici, nous libérons le nœud actuel de la pile. C'est sûr parce que nous avons déjà sauvegardé le pointeur vers le prochain nœud.
*stack = tmp; : Ici, nous déplaçons le pointeur de la pile vers le prochain nœud que nous avons sauvegardé précédemment.
*stack = NULL; : Enfin, après avoir libéré tous les nœuds de la pile, nous mettons le pointeur de la pile à NULL. Cela est fait pour indiquer que la pile est vide et pour prévenir tout accès ultérieur à des zones de mémoire qui ont été libérées.

Void exit_error(t_stack **stack_a, t_stack **stack_b)

La fonction exit_error libère deux piles fournies (si elles ne sont pas NULL), écrit le message "Error\n" dans la sortie d'erreur standard, et arrête le programme avec le code d'erreur 1.

Long int ft_atoi(const char *str)

La fonction ft_atoi convertit une chaîne de caractères alphanumériques en un long entier. Elle prend en compte les signes positifs ou négatifs au début de la chaîne. Ensuite, elle parcourt la chaîne de caractères et pour chaque chiffre, elle le convertit en un entier et l'ajoute au résultat total, multiplié par 10 pour tenir compte du décalage des chiffres. À la fin, elle multiplie le résultat par le signe pour obtenir le nombre final.

Void ft_putstr(char *str)

La fonction ft_putstr imprime une chaîne de caractères donnée sur la sortie standard. Elle parcourt la chaîne de caractères caractère par caractère et l'écrit dans la sortie standard.

Int nb_abs(int nb)

La fonction nb_abs retourne la valeur absolue d'un nombre donné. Elle vérifie si le nombre est négatif, et si c'est le cas, elle le multiplie par -1 pour obtenir sa valeur absolue. Sinon, elle retourne simplement le nombre lui-même.

Sort_tiny.c

```c
#include "push_swap.h"


/* find_highest_index :
* Renvoie l'indice le plus élevé d'une pile.
```

```c
*/
static int  find_highest_index(t_stack *stack)
{
    int index;

    index = stack->index;
    while (stack)
    {
        if (stack->index > index)
            index = stack->index;
        stack = stack->next;
    }
    return (index);
}


/* tiny_sort :
* Trie une pile de 3 numéros en 2 coups ou moins. Le tri se fait par index
* plutôt que la valeur. Exemple:
* valeurs : 0 9 2
* index : [1] [3] [2]
* Solution, 2 coups :
*rra :
* valeurs : 2 0 9
* index : [2] [1] [3]
* sa :
* valeurs : 0 2 9
* index : [1] [2] [3]
*/
void    tiny_sort(t_stack **stack)
{
    int highest;

    if (is_sorted(*stack))
        return ;
    highest = find_highest_index(*stack);
    if ((*stack)->index == highest)
        do_ra(stack);
    else if ((*stack)->next->index == highest)
        do_rra(stack);
    if ((*stack)->index > (*stack)->next->index)
        do_sa(stack);
}
```

static int find_highest_index(t_stack *stack)

int index; : Déclare une variable index.
index = stack->index; : Initialise index à l'indice du premier élément de la pile.
while (stack) : Démarre une boucle qui parcourt tous les nœuds de la pile.
if (stack->index > index) index = stack->index; : Si l'indice du nœud actuel est plus grand que l'indice maximal trouvé jusqu'à présent, met à jour index.
stack = stack->next; : Passe au nœud suivant de la pile.
return (index); : Après avoir parcouru toute la pile, renvoie l'indice le plus élevé trouvé.

void tiny_sort(t_stack **stack)

int highest; : Déclare une variable highest.
if (is_sorted(*stack)) return; : Si la pile est déjà triée, quitte la fonction immédiatement.
highest = find_highest_index(*stack); : Trouve l'indice le plus élevé dans la pile.
if ((*stack)->index == highest) do_ra(stack); : Si l'indice le plus élevé est en haut de la pile, exécute une rotation vers le haut (met le premier élément en bas).
else if ((*stack)->next->index == highest) do_rra(stack); : Sinon, si l'indice le plus élevé est au deuxième rang de la pile, exécute une rotation vers le bas (met le dernier élément en haut).
if ((*stack)->index > (*stack)->next->index) do_sa(stack); : Enfin, si l'indice en haut de la pile est plus grand que le suivant, échange ces deux éléments.
En résumé, tiny_sort est une fonction de tri simple qui peut trier une pile de trois éléments en deux mouvements ou moins. Elle utilise les indices plutôt que les valeurs pour déterminer l'ordre des éléments, et effectue les mouvements nécessaires pour les mettre en ordre croissant d'indice.

Sort.c

```c
#include "push_swap.h"


/* push_all_save_three :
* Pousse tous les éléments de la pile a dans la pile b, sauf les trois derniers.
* Pousse d'abord les valeurs les plus petites, puis les valeurs les plus grandes pour
aider à
* efficacité de tri.
*/
static void push_all_save_three(t_stack **stack_a, t_stack **stack_b)
{
    int stack_size;
    int pushed;
    int i;

    stack_size = get_stack_size(*stack_a);
    pushed = 0;
    i = 0;
    while (stack_size > 6 && i < stack_size && pushed < stack_size / 2)
    {
        if ((*stack_a)->index <= stack_size / 2)
        {
            do_pb(stack_a, stack_b);
            pushed++;
        }
        else
            do_ra(stack_a);
        i++;
    }
    while (stack_size - pushed > 3)
    {
        do_pb(stack_a, stack_b);
        pushed++;
    }
}
```

```c
/* shift_stack :
* Une fois le gros de la pile trié, décale la pile a jusqu'à la plus basse
* la valeur est en haut. S'il se trouve dans la moitié inférieure de la pile, inversez
* faites-le pivoter en position, sinon faites-le pivoter jusqu'à ce qu'il soit en haut du
*     empiler.
*/
static void shift_stack(t_stack **stack_a)
{
    int lowest_pos;
    int stack_size;

    stack_size = get_stack_size(*stack_a);
    lowest_pos = get_lowest_index_position(stack_a);
    if (lowest_pos > stack_size / 2)
    {
        while (lowest_pos < stack_size)
        {
            do_rra(stack_a);
            lowest_pos++;
        }
    }
    else
    {
        while (lowest_pos > 0)
        {
            do_ra(stack_a);
            lowest_pos--;
        }
    }
}

/* trier:
* Algorithme de tri pour une pile supérieure à 3.
* Poussez tout sauf 3 numéros pour empiler B.
* Triez les 3 nombres restants dans la pile A.
* Calculez le déménagement le plus rentable.
* Décalez les éléments jusqu'à ce que la pile A soit en ordre.
*/void    sort(t_stack **stack_a, t_stack **stack_b)
{
    push_all_save_three(stack_a, stack_b);
    tiny_sort(stack_a);
    while (*stack_b)
    {
        get_target_position(stack_a, stack_b);
        get_cost(stack_a, stack_b);
        do_cheapest_move(stack_a, stack_b);
    }
    if (!is_sorted(*stack_a))
        shift_stack(stack_a);
}
```

static void push_all_save_three(t_stack **stack_a, t_stack **stack_b)

stack_size = get_stack_size(*stack_a); : Obtenir la taille de la pile stack_a.

pushed = 0; i = 0; : Initialiser deux compteurs à zéro, pushed compte le nombre d'éléments poussés de stack_a à stack_b, i sert d'itérateur pour la boucle.
La première boucle while parcourt la pile stack_a jusqu'à ce que la moitié des éléments ait été poussée vers stack_b, ou jusqu'à ce que i atteigne la taille de la pile.
Si l'indice de l'élément en haut de stack_a est inférieur ou égal à la moitié de la taille de la pile, il est poussé vers stack_b et le compteur pushed est incrémenté.
Sinon, un mouvement de rotation (vers le haut) est effectué sur stack_a.
L'itérateur i est incrémenté à chaque itération.
La deuxième boucle while continue à pousser les éléments de stack_a vers stack_b jusqu'à ce qu'il reste seulement trois éléments dans stack_a.

static void shift_stack(t_stack **stack_a)

stack_size = get_stack_size(*stack_a); : Obtenir la taille de stack_a.
lowest_pos = get_lowest_index_position(stack_a); : Obtenir la position de l'élément ayant le plus petit indice dans stack_a.
if (lowest_pos > stack_size / 2) {...} : Si l'élément ayant le plus petit indice se trouve dans la moitié inférieure de la pile, effectuez une rotation inverse jusqu'à ce qu'il atteigne le haut de la pile.
else {...} : Sinon, effectuer une rotation jusqu'à ce que l'élément atteigne le haut de la pile.

void sort(t_stack **stack_a, t_stack **stack_b)

push_all_save_three(stack_a, stack_b); : Pousse tous les éléments de stack_a vers stack_b, sauf trois.
tiny_sort(stack_a); : Trie les trois éléments restants dans stack_a.
La boucle while continue d'effectuer des opérations tant qu'il y a des éléments dans stack_b. Pour chaque itération :
get_target_position(stack_a, stack_b); : Calcule la position cible pour l'élément en haut de stack_b dans stack_a.
get_cost(stack_a, stack_b); : Calcule le coût de déplacement de l'élément en haut de stack_b vers cette position cible.
do_cheapest_move(stack_a, stack_b); : Effectue le mouvement qui a le coût le plus bas.
if (!is_sorted(*stack_a)) shift_stack(stack_a); : Si stack_a n'est pas encore trié, effectue une rotation jusqu'à ce que l'élément ayant l'indice le plus bas soit en haut de la pile.

Cost.c

```c
#include "push_swap.h"


/* get_cost :
* Calcule le coût du déplacement de chaque élément de la pile B dans le bon
* position dans la pile A.
* Deux coûts sont calculés :
* cost_b représente le coût pour amener l'élément au sommet de la pile B
* cost_a représente le coût pour arriver à la bonne position dans la pile A.
* Si l'élément est dans la moitié inférieure de la pile, le coût sera négatif,
* s'il se situe dans la moitié supérieure, le coût est positif.
*/
void    get_cost(t_stack **stack_a, t_stack **stack_b)
{
    t_stack *tmp_a;
    t_stack *tmp_b;
    int     size_a;
    int     size_b;

    tmp_a = *stack_a;
    tmp_b = *stack_b;
    size_a = get_stack_size(tmp_a);
```

```
    size_b = get_stack_size(tmp_b);
    while (tmp_b)
    {
        tmp_b->cost_b = tmp_b->pos;
        if (tmp_b->pos > size_b / 2)
            tmp_b->cost_b = (size_b - tmp_b->pos) * -1;
        tmp_b->cost_a = tmp_b->target_pos;
        if (tmp_b->target_pos > size_a / 2)
            tmp_b->cost_a = (size_a - tmp_b->target_pos) * -1;
        tmp_b = tmp_b->next;
    }
}


/* do_cheapest_move :
* Trouve l'élément dans la pile B avec le coût le moins cher pour se déplacer vers la pile
A
* et le déplace à la bonne position dans la pile A.
*/
void    do_cheapest_move(t_stack **stack_a, t_stack **stack_b)
{
    t_stack *tmp;
    int     cheapest;
    int     cost_a;
    int     cost_b;

    tmp = *stack_b;
    cheapest = INT_MAX;
    while (tmp)
    {
        if (nb_abs(tmp->cost_a) + nb_abs(tmp->cost_b) < nb_abs(cheapest))
        {
            cheapest = nb_abs(tmp->cost_b) + nb_abs(tmp->cost_a);
            cost_a = tmp->cost_a;
            cost_b = tmp->cost_b;
        }
        tmp = tmp->next;
    }
    do_move(stack_a, stack_b, cost_a, cost_b);
}
```

void get_cost(t_stack **stack_a, t_stack **stack_b)

Cette fonction calcule le coût du déplacement de chaque élément de la pile B vers la bonne position dans la pile A.

t_stack *tmp_a; t_stack *tmp_b; int size_a; int size_b; : On déclare deux pointeurs temporaires pour les piles A et B et deux entiers pour la taille de ces piles.
tmp_a = *stack_a; tmp_b = *stack_b; : On pointe vers les premiers éléments des piles A et B.
size_a = get_stack_size(tmp_a); size_b = get_stack_size(tmp_b); : On obtient la taille des piles A et B.
while (tmp_b) : Une boucle qui continue tant que tmp_b (l'élément actuel de la pile B) n'est pas NULL.
tmp_b->cost_b = tmp_b->pos; : On attribue la position de l'élément actuel comme coût pour le déplacer vers le haut de la pile B.
if (tmp_b->pos > size_b / 2) tmp_b->cost_b = (size_b - tmp_b->pos) * -1; : Si l'élément est dans la moitié inférieure de la pile B, le coût est négatif (on multiplie par -1).

tmp_b->cost_a = tmp_b->target_pos; : On attribue la position cible de l'élément actuel comme coût pour le déplacer vers la position correcte dans la pile A.
if (tmp_b->target_pos > size_a / 2) tmp_b->cost_a = (size_a - tmp_b->target_pos) * -1; : Si la position cible est dans la moitié supérieure de la pile A, le coût est négatif (on multiplie par -1).
tmp_b = tmp_b->next; : On déplace le pointeur temporaire vers l'élément suivant de la pile B.

void do_cheapest_move(t_stack **stack_a, t_stack **stack_b)

Cette fonction trouve l'élément de la pile B qui a le coût le plus bas pour être déplacé vers la pile A, et le déplace à la position correcte dans la pile A.

t_stack *tmp; int cheapest; int cost_a; int cost_b; : On déclare un pointeur temporaire, une variable pour le coût le moins cher et deux variables pour les coûts de déplacement à la pile A et B.
tmp = *stack_b; : On pointe vers le premier élément de la pile B.
cheapest = INT_MAX; : On initialise le coût le moins cher à la plus grande valeur possible d'un entier.
while (tmp) : Une boucle qui continue tant que tmp (l'élément actuel de la pile B) n'est pas NULL.
if (nb_abs(tmp->cost_a) + nb_abs(tmp->cost_b) < nb_abs(cheapest)) : Si le coût absolu de déplacer l'élément actuel vers la pile A et le haut de la pile B est inférieur au coût le moins cher actuel...
cheapest = nb_abs(tmp->cost_b) + nb_abs(tmp->cost_a); et cost_a = tmp->cost_a; cost_b = tmp->cost_b; : On met à jour le coût le moins cher et les coûts de déplacement à la pile A et B.
tmp = tmp->next; : On déplace le pointeur temporaire vers l'élément suivant de la pile B.
do_move(stack_a, stack_b, cost_a, cost_b); : On appelle la fonction do_move pour déplacer l'élément de la pile B avec le coût le moins cher à la position correcte dans la pile A.

do_move.c

```c
#include "push_swap.h"


/* do_rev_rotate_both :
* L'inverse fait tourner les piles A et B jusqu'à ce que l'une d'elles soit en position.
* Le coût donné est négatif puisque les deux positions sont dans la moitié inférieure
* de leurs piles respectives. Le coût est augmenté à mesure que les piles sont
* tourné, quand on atteint 0, la pile a été tournée au maximum
* et la position supérieure est correcte.
*/
static void do_rev_rotate_both(t_stack **a, t_stack **b,
                               int *cost_a, int *cost_b)
{
    while (*cost_a < 0 && *cost_b < 0)
    {
        (*cost_a)++;
        (*cost_b)++;
        do_rrr(a, b);
    }
}


/* do_rotate_both :
* Fait pivoter les piles A et B jusqu'à ce que l'une d'elles soit en position.
* Le coût donné est positif puisque les deux positions sont dans la moitié supérieure
* de leurs piles respectives. Le coût est diminué à mesure que les piles sont
* tourné, quand on atteint 0, la pile a été tournée au maximum
* et la position supérieure est correcte.
```

```c
*/
static void do_rotate_both(t_stack **a, t_stack **b, int *cost_a, int *cost_b)
{
    while (*cost_a > 0 && *cost_b > 0)
    {
        (*cost_a)--;
        (*cost_b)--;
        do_rr(a, b);
    }
}

/* do_rotate_a :
* Fait pivoter la pile A jusqu'à ce qu'elle soit en position. Si le coût est négatif,
* la pile sera inversée, si elle est positive, elle sera
* tourné.
*/
static void do_rotate_a(t_stack **a, int *cost)
{
    while (*cost)
    {
        if (*cost > 0)
        {
            do_ra(a);
            (*cost)--;
        }
        else if (*cost < 0)
        {
            do_rra(a);
            (*cost)++;
        }
    }
}

/* do_rotate_b :
* Fait pivoter la pile B jusqu'à ce qu'elle soit en position. Si le coût est négatif,
* la pile sera inversée, si elle est positive, elle sera
* tourné.
*/
static void do_rotate_b(t_stack **b, int *cost)
{
    while (*cost)
    {
        if (*cost > 0)
        {
            do_rb(b);
            (*cost)--;
        }
        else if (*cost < 0)
        {
            do_rrb(b);
            (*cost)++;
        }
    }
}
```

```c
/* do_move :
* Choisit quel mouvement effectuer pour obtenir l'élément de pile B au bon endroit
* position dans la pile A.
* Si les coûts de déplacement des piles A et B vers la position correspondent (c'est-à-
dire
* des deux positifs), les deux seront tournés ou inversés en même temps.
* Ils peuvent également être tournés séparément, avant de finalement pousser l'élément B
supérieur
* à la pile supérieure A.
*/
void    do_move(t_stack **a, t_stack **b, int cost_a, int cost_b)
{
    if (cost_a < 0 && cost_b < 0)
        do_rev_rotate_both(a, b, &cost_a, &cost_b);
    else if (cost_a > 0 && cost_b > 0)
        do_rotate_both(a, b, &cost_a, &cost_b);
    do_rotate_a(a, &cost_a);
    do_rotate_b(b, &cost_b);
    do_pa(a, b);
}
```

Ce code C implémente plusieurs fonctions qui exécutent des mouvements sur deux piles de données, a et b. Les piles sont manipulées selon le coût défini pour chaque mouvement, qui est stocké dans cost_a et cost_b.

static void do_rev_rotate_both(t_stack **a, t_stack **b, int *cost_a, int *cost_b)

Cette fonction fait tourner les deux piles a et b dans le sens inverse jusqu'à ce que l'une d'elles soit en position. Le coût donné est négatif car les deux positions sont dans la moitié inférieure de leurs piles respectives. Le coût est augmenté au fur et à mesure que les piles tournent, et lorsqu'un coût atteint 0, la pile a été tournée autant que possible et la position du haut est correcte.

static void do_rotate_both(t_stack **a, t_stack **b, int *cost_a, int *cost_b)

De manière similaire à do_rev_rotate_both, cette fonction fait tourner les deux piles a et b, mais dans le sens normal jusqu'à ce que l'une d'elles soit en position. Le coût donné est positif car les deux positions sont dans la moitié supérieure de leurs piles respectives.

static void do_rotate_a(t_stack **a, int *cost)

Cette fonction fait tourner la pile a jusqu'à ce qu'elle soit en position. Si le coût est négatif, la pile sera tournée dans le sens inverse, si le coût est positif, elle sera tournée dans le sens normal.

static void do_rotate_b(t_stack **b, int *cost)

De manière similaire à do_rotate_a, cette fonction fait tourner la pile b jusqu'à ce qu'elle soit en position.

void do_move(t_stack **a, t_stack **b, int cost_a, int cost_b)

Cette fonction choisit quel mouvement faire pour amener l'élément de la pile b à la position correcte dans la pile a. Si les coûts de déplacement des piles a et b correspondent (c'est-à-dire, les deux sont négatifs ou les deux sont positifs), les deux seront tournés ou tournés en sens inverse en même temps. Ils pourraient également être tournés séparément, avant de finalement pousser l'élément du haut de b vers le haut de la pile a.

Input_check_utils.c

```c
#include "push_swap.h"
```

```c
/* est_chiffre :
* Vérifie si le caractère est un chiffre.
* Retour : 1 si le caractère est un chiffre, 0 sinon.
*/
int is_digit(char c)
{
    return (c >= '0' && c <= '9');
}


/* est_signe :
* Vérifie si le caractère est un signe dièse, + ou -.
* Retourne : 1 si le caractère est un signe, 0 sinon.
*/
int is_sign(char c)
{
    return (c == '+' || c == '-');
}


/* nbstr_cmp :
* Compare deux chaînes de chiffres pour vérifier si elles sont identiques ou non.
* Si l'une des chaînes commence par un signe '+', elle est ignorée et le reste
* de la chaîne numérique est cochée.
* C'est ainsi que +123 == 123 mais -123 != 123.
* Retour : 0 si les chaînes correspondent, une autre valeur si ce n'est pas le cas.
*/
int nbstr_cmp(const char *s1, const char *s2)
{
    int i;
    int j;

    i = 0;
    j = i;
    if (s1[i] == '+')
    {
        if (s2[j] != '+')
            i++;
    }
    else
    {
        if (s2[j] == '+')
            j++;
    }
    while (s1[i] != '\0' && s2[j] != '\0' && s1[i] == s2[j])
    {
        i++;
        j++;
    }
    return ((unsigned char)s1[i] - (unsigned char)s2[j]);
}
```

Le code que vous avez partagé implémente trois fonctions distinctes qui sont utilisées pour vérifier et comparer les chaînes de caractères qui représentent des nombres. Voici une explication ligne par ligne pour chaque fonction :

is_digit :

Cette fonction vérifie si un caractère donné est un chiffre. Pour ce faire, elle vérifie si le code ASCII du caractère est entre les codes ASCII de '0' et '9'. Si c'est le cas, elle renvoie 1, sinon elle renvoie 0.

is_sign :

Cette fonction vérifie si un caractère donné est un signe de nombre, c'est-à-dire '+' ou '-'. Si c'est le cas, elle renvoie 1, sinon elle renvoie 0.

nbstr_cmp :

Cette fonction compare deux chaînes de caractères qui représentent des nombres pour vérifier si elles sont équivalentes. Si l'une des chaînes commence par un signe '+', celui-ci est ignoré et le reste de la chaîne de nombre est vérifié. Cela signifie que '+123' serait considéré comme équivalent à '123', mais '-123' ne le serait pas. La fonction renvoie 0 si les chaînes sont équivalentes et une autre valeur si elles ne le sont pas.

La logique de cette fonction est assez simple. Elle commence par vérifier si la première chaîne commence par un signe '+'. Si c'est le cas, et que la deuxième chaîne ne commence pas par un '+', elle avance d'un caractère dans la première chaîne. Elle fait la même vérification pour la deuxième chaîne. Ensuite, elle compare les caractères des deux chaînes jusqu'à ce qu'elle atteigne la fin de l'une des chaînes ou qu'elle trouve des caractères non équivalents. À la fin, elle renvoie la différence entre les codes ASCII du dernier caractère vérifié dans chaque chaîne.

Position.c

```c
#include "push_swap.h"

/* obtenir_position :
* Attribue une position à chaque élément d'une pile de haut en bas
*     Dans l'ordre croissant.
* Exemple de pile :
* valeur : 3 0 9 1
* indice : [3] [1] [4] [2]
* poste : <0> <1> <2> <3>
* Ceci est utilisé pour calculer le coût de déplacement d'un certain nombre vers
* une certaine position. Si l'exemple ci-dessus est la pile b, il en coûterait
* rien (0) pour pousser le premier élément à empiler a. Cependant si nous voulons
* poussez la valeur la plus élevée, 9, qui est en troisième position, cela coûterait 2
supplémentaires
* se déplace pour amener ce 9 au sommet de b avant de le pousser pour empiler a.
*/
static void get_position(t_stack **stack)
{
    t_stack *tmp;
    int     i;

    tmp = *stack;
    i = 0;
    while (tmp)
    {
        tmp->pos = i;
```

```
        tmp = tmp->next;
        i++;
    }
}


/* get_lowest_index_position :
* Obtient la position actuelle de l'élément avec l'index le plus bas
* dans une pile.
*/
int get_lowest_index_position(t_stack **stack)
{
    t_stack *tmp;
    int     lowest_index;
    int     lowest_pos;

    tmp = *stack;
    lowest_index = INT_MAX;
    get_position(stack);
    lowest_pos = tmp->pos;
    while (tmp)
    {
        if (tmp->index < lowest_index)
        {
            lowest_index = tmp->index;
            lowest_pos = tmp->pos;
        }
        tmp = tmp->next;
    }
    return (lowest_pos);
}

/* obtenir_cible :
* Choisit la meilleure position cible dans la pile A pour l'indice donné de
* un élément dans la pile B. Vérifie d'abord si l'indice de l'élément B peut
* être placé quelque part entre les éléments de la pile A, en vérifiant si
* il y a un élément dans la pile A avec un index plus grand. Sinon, il trouve le
* élément avec le plus petit index dans A et l'affecte comme position cible.
*     --- Exemple:
* target_pos commence à INT_MAX
* Indice B : 3
* A contient les index : 9 4 2 1 0
* 9 > 3 && 9 < INT_MAX : target_pos mis à jour à 9
* 4 > 3 && 4 < 9 : position cible mise à jour à 4
* 2 < 3 && 2 < 4 : pas de mise à jour !
* Donc target_pos doit être la position de l'index 4, puisqu'il est
* l'indice le plus proche.
*     --- Exemple:
* target_pos commence à INT_MAX
* Indice B : 20
* A contient des index : 16 4 3
* 16 < 20 : pas de mise à jour ! target_pos = INT_MAX
* 4 < 20 : pas de mise à jour ! target_pos = INT_MAX
* 3 < 20 : pas de mise à jour ! target_pos = INT_MAX
```

```c
* ... target_pos reste à INT MAX, besoin de changer de stratégie.
* 16 < 20 : target_pos mis à jour à 20
* 4 < 20 : target_pos mis à jour à 4
* 3 < 20 : target_pos mis à jour à 3
* Donc target_pos doit être la position de l'index 3, puisque c'est
* la "fin" de la pile.
*/
static int  get_target(t_stack **a, int b_idx, int target_idx, int target_pos)
{
    t_stack *tmp_a;

    tmp_a = *a;
    while (tmp_a)
    {
        if (tmp_a->index > b_idx && tmp_a->index < target_idx)
        {
            target_idx = tmp_a->index;
            target_pos = tmp_a->pos;
        }
        tmp_a = tmp_a->next;
    }
    if (target_idx != INT_MAX)
        return (target_pos);
    tmp_a = *a;
    while (tmp_a)
    {
        if (tmp_a->index < target_idx)
        {
            target_idx = tmp_a->index;
            target_pos = tmp_a->pos;
        }
        tmp_a = tmp_a->next;
    }
    return (target_pos);
}


/* get_target_position :
* Attribue une position cible dans la pile A à chaque élément de la pile A.
* La position cible est l'endroit où l'élément en B doit
* obtenir afin d'être trié correctement. Ce poste va
* être utilisé pour calculer le coût de déplacement de chaque élément vers
* sa position cible dans la pile A.
*/
void    get_target_position(t_stack **a, t_stack **b)
{
    t_stack *tmp_b;
    int     target_pos;

    tmp_b = *b;
    get_position(a);
    get_position(b);
    target_pos = 0;
    while (tmp_b)
```

```
    {
        target_pos = get_target(a, tmp_b->index, INT_MAX, target_pos);
        tmp_b->target_pos = target_pos;
        tmp_b = tmp_b->next;
    }
}
```

static void get_position(t_stack **stack)

Cette fonction attribue une position à chaque élément d'une pile, allant du haut (la tête de la pile) vers le bas.
t_stack *tmp; int i; : Ce sont les déclarations des variables locales. tmp est un pointeur temporaire sur t_stack et i est un compteur.
tmp = *stack; : Cette ligne fait pointer tmp vers le début de la pile.
i = 0; : Initialise le compteur i à 0.
while (tmp) : Commence une boucle qui se poursuit tant que tmp n'est pas NULL, c'est-à-dire tant qu'il reste des éléments dans la pile.
tmp->pos = i; : Attribue la valeur actuelle de i (la position) à l'attribut pos de l'élément de la pile pointé par tmp.
tmp = tmp->next; : Déplace tmp vers le prochain élément de la pile.
i++; : Incrémente le compteur i à chaque itération de la boucle.

get_lowest_index_position : Cette fonction trouve la position de l'élément ayant l'index le plus bas dans une pile.

int get_lowest_index_position(t_stack **stack)

Cette fonction trouve la position de l'élément ayant l'indice le plus bas dans la pile.

t_stack *tmp; int lowest_index; int lowest_pos; : On déclare un pointeur temporaire, une variable pour le plus petit indice et une variable pour la position du plus petit indice.
tmp = *stack; : On pointe vers le premier élément de la pile.
lowest_index = INT_MAX; : On initialise l'indice le plus bas à la plus grande valeur possible d'un entier.
get_position(stack); : On appelle la fonction get_position pour attribuer une position à chaque élément de la pile.
lowest_pos = tmp->pos; : On initialise la position du plus petit indice à la position de l'élément actuel de la pile.
while (tmp) : Une boucle qui continue tant que tmp (l'élément actuel de la pile) n'est pas NULL.
if (tmp->index < lowest_index) : Si l'indice de l'élément actuel est inférieur à l'indice le plus bas actuel...
lowest_index = tmp->index; et lowest_pos = tmp->pos; : On met à jour l'indice le plus bas et la position du plus petit indice.
tmp = tmp->next; : On déplace le pointeur temporaire vers l'élément suivant de la pile.
return (lowest_pos); : On renvoie la position de l'élément ayant l'indice le plus bas.

static int get_target(t_stack **a, int b_idx, int target_idx, int target_pos)

Cette fonction trouve la position cible dans la pile A pour un élément donné dans la pile B.

t_stack *tmp_a; : On déclare un pointeur temporaire pour la pile A.
tmp_a = *a; : On pointe vers le premier élément de la pile A.
while (tmp_a) : Une boucle qui continue tant que tmp_a (l'élément actuel de la pile A) n'est pas NULL.
if (tmp_a->index > b_idx && tmp_a->index < target_idx) : On vérifie si l'indice de l'élément actuel de la pile A est supérieur à l'indice de l'élément de la pile B et inférieur à l'indice cible actuel...
target_idx = tmp_a->index; et target_pos = tmp_a->pos; : Si c'est le cas, on met à jour l'indice cible et la position cible.
tmp_a = tmp_a->next; : On déplace le pointeur temporaire vers l'élément suivant de la pile A.
if (target_idx != INT_MAX) et return (target_pos); : Si on a trouvé un indice cible, on renvoie la position cible.
La deuxième boucle while (tmp_a) cherche l'élément avec le plus petit indice dans la pile A si aucun indice cible n'a été trouvé dans la première boucle.

void get_target_position(t_stack **a, t_stack **b)

Cette fonction attribue une position cible dans la pile A à chaque élément de la pile B.

t_stack *tmp_b; : On déclare un pointeur temporaire pour la pile B.

tmp_b = *b; : On pointe vers le premier élément de la pile B.
while (tmp_b) : Une boucle qui continue tant que tmp_b (l'élément actuel de la pile B) n'est pas NULL.
tmp_b->target_idx = INT_MAX; : On initialise l'indice cible de l'élément actuel de la pile B à la plus grande valeur possible d'un entier.
tmp_b->target_pos = get_target(a, tmp_b->index, tmp_b->target_idx, tmp_b->target_pos); : On appelle la fonction get_target pour trouver la position cible dans la pile A pour l'élément actuel de la pile B, et on l'attribue à target_pos.
tmp_b = tmp_b->next; : On déplace le pointeur temporaire vers l'élément suivant de la pile B.

push.c

```c
#include "push_swap.h"

/* pousser:
* Pousse l'élément supérieur de la pile src vers le haut de la pile dest.
*/
static void push(t_stack **src, t_stack **dest)
{
    t_stack *tmp;

    if (*src == NULL)
        return ;
    tmp = (*src)->next;
    (*src)->next = *dest;
    *dest = *src;
    *src = tmp;
}


/* do_pa :
* Pousse l'élément supérieur de la pile b vers le haut de la pile a.
* Affiche "pa" sur la sortie standard.
*/
void    do_pa(t_stack **stack_a, t_stack **stack_b)
{
    push(stack_b, stack_a);
    ft_putstr("pa\n");
}


/* do_pb :
* Pousse l'élément supérieur de la pile a vers le haut de la pile b.
* Imprime "pb" sur la sortie standard.
*/
void    do_pb(t_stack **stack_a, t_stack **stack_b)
{
    push(stack_a, stack_b);
    ft_putstr("pb\n");
}
```

static void push(t_stack **src, t_stack **dest)

Cette fonction prend deux piles en entrée, src et dest. Elle déplace le premier élément de la pile src vers le haut de la pile dest.

La fonction commence par vérifier si src est NULL, c'est-à-dire si la pile src est vide. Si c'est le cas, elle retourne immédiatement, car il n'y a rien à déplacer.
Ensuite, la fonction stocke l'élément suivant de la pile src dans la variable tmp. Elle fait pointer le champ next de src vers dest, ce qui signifie que le premier élément de src pointe maintenant vers le premier élément de dest.
Ensuite, dest est mis à pointer vers src, donc maintenant le premier élément de src est le premier élément de dest.
Enfin, src est mis à pointer vers tmp, ce qui signifie que le premier élément de src est maintenant l'ancien deuxième élément.

void do_pa(t_stack **stack_a, t_stack **stack_b)

Cette fonction appelle push pour déplacer le premier élément de la pile b vers le haut de la pile a, puis elle imprime "pa" à la sortie standard pour indiquer qu'un mouvement "pa" a été effectué.

void do_pb(t_stack **stack_a, t_stack **stack_b)

De même, cette fonction appelle push pour déplacer le premier élément de la pile a vers le haut de la pile b, puis elle imprime "pb" à la sortie standard pour indiquer qu'un mouvement "pb" a été effectué.

Reverse_rotate.c

```c
#include "push_swap.h"


/* rev_rotate :
* Amène l'élément inférieur d'une pile vers le haut.
*/
static void rev_rotate(t_stack **stack)
{
    t_stack *tmp;
    t_stack *tail;
    t_stack *before_tail;

    tail = get_stack_bottom(*stack);
    before_tail = get_stack_before_bottom(*stack);
    tmp = *stack;
    *stack = tail;
    (*stack)->next = tmp;
    before_tail->next = NULL;
}

/* do_rra :
* Amène l'élément du bas de la pile a vers le haut.
* Imprime "rra" sur la sortie standard.
*/
void    do_rra(t_stack **stack_a)
{
    rev_rotate(stack_a);
    ft_putstr("rra\n");
}


/* do_rrb :
* Amène l'élément du bas de la pile b vers le haut.
* Affiche "rrb" sur la sortie standard.
```

```
*/
void    do_rrb(t_stack **stack_b)
{
    rev_rotate(stack_b);
    ft_putstr("rrb\n");
}

/* do_rrr :
* Apporte l'élément inférieur de la pile a et de la pile be
* au sommet de leurs piles respectives.
* Affiche "rrr" sur la sortie standard.
*/
void    do_rrr(t_stack **stack_a, t_stack **stack_b)
{
    rev_rotate(stack_a);
    rev_rotate(stack_b);
    ft_putstr("rrr\n");
}
```

Static void   rev_rotate(t_stack **stack)

Cette fonction prend comme argument un pointeur vers une pile et fait tourner les éléments de cette pile vers le haut. Elle utilise trois pointeurs : tail pour garder une référence vers le dernier élément de la pile, before_tail pour garder une référence vers l'élément juste avant le dernier, et tmp pour garder une référence vers l'élément en haut de la pile.

La fonction met d'abord tail à pointer vers le dernier élément de la pile et before_tail à pointer vers l'élément juste avant le dernier. Elle stocke ensuite le pointeur vers l'élément du haut de la pile dans tmp. Ensuite, elle modifie le pointeur vers la tête de la pile pour qu'il pointe vers tail, ce qui fait de l'ancien dernier élément le nouvel élément du haut de la pile. Elle modifie alors le champ next du nouvel élément en haut de la pile pour qu'il pointe vers l'ancien élément du haut de la pile. Enfin, elle met le champ next de before_tail à NULL pour signifier que c'est maintenant le dernier élément de la pile.

void   do_rra(t_stack **stack_a)

Cette fonction exécute une rotation inverse sur la pile a en appelant la fonction rev_rotate et imprime "rra" à la sortie standard. Cela indique qu'un mouvement "rra" a été effectué.

void   do_rrb(t_stack **stack_b)

De manière similaire à do_rra, cette fonction exécute une rotation inverse sur la pile b et imprime "rrb" à la sortie standard.

void   do_rrr(t_stack **stack_a, t_stack **stack_b)

Cette fonction effectue une rotation inverse sur les deux piles a et b et imprime "rrr" à la sortie standard.

Rotate.c

```
#include "push_swap.h"


/* faire pivoter :
* L'élément supérieur de la pile est envoyé en bas.
*/
```

```c
static void rotate(t_stack **stack)
{
    t_stack *tmp;
    t_stack *tail;

    tmp = *stack;
    *stack = (*stack)->next;
    tail = get_stack_bottom(*stack);
    tmp->next = NULL;
    tail->next = tmp;
}


/* do_ra :
* Envoie l'élément du haut de la pile a vers le bas.
* Imprime "ra" sur la sortie standard.
*/
void    do_ra(t_stack **stack_a)
{
    rotate(stack_a);
    ft_putstr("ra\n");
}


/* do_rb :
* Envoie l'élément du haut de la pile b vers le bas.
* Affiche "rb" sur la sortie standard.
*/
void    do_rb(t_stack **stack_b)
{
    rotate(stack_b);
    ft_putstr("rb\n");
}


/* do_rr :
* Envoie l'élément supérieur de la pile a et de la pile b vers le bas
* de leurs piles respectives.
* Affiche "rr" sur la sortie standard.
*/
void    do_rr(t_stack **stack_a, t_stack **stack_b)
{
    rotate(stack_a);
    rotate(stack_b);
    ft_putstr("rr\n");
}
```

static void rotate(t_stack **stack)

Cette fonction prend comme argument un pointeur vers une pile et fait tourner les éléments de cette pile vers le bas. Elle stocke temporairement le premier élément de la pile dans tmp, puis avance la tête de la pile vers le prochain élément. Ensuite, elle trouve l'élément de fin de la pile (tail) et met le champ next de tmp à NULL pour signifier qu'il s'agit maintenant du dernier élément. Enfin, elle attache tmp à la fin de la pile.

void do_ra(t_stack **stack_a)

Cette fonction exécute une rotation sur la pile a en appelant la fonction rotate et imprime "ra" à la sortie standard. Cela indique qu'un mouvement "ra" a été effectué.

void do_rb(t_stack **stack_b)

De manière similaire à do_ra, cette fonction exécute une rotation sur la pile b et imprime "rb" à la sortie standard..

void do_rr(t_stack **stack_a, t_stack **stack_b)

Cette fonction effectue une rotation sur les deux piles a et b et imprime "rr" à la sortie standard.

Swap.c

```c
#include "push_swap.h"

/* échanger:
* Permute les 2 premiers éléments d'une pile.
* Ne fait rien s'il n'y a qu'un ou aucun élément.
*/
static void swap(t_stack *stack)
{
    int tmp;

    if (stack == NULL || stack->next == NULL)
        return ;
    tmp = stack->value;
    stack->value = stack->next->value;
    stack->next->value = tmp;
    tmp = stack->index;
    stack->index = stack->next->index;
    stack->next->index = tmp;
}


/* faire_sa :
* Échange les 2 premiers éléments de la pile a.
* Imprime "sa" sur la sortie standard.
*/
void    do_sa(t_stack **stack_a)
{
    swap(*stack_a);
    ft_putstr("sa\n");
    print_status(*stack_a, NULL);
}

/* do_sb :
* Échange les 2 premiers éléments de la pile b.
* Affiche "sb" sur la sortie standard.
*/
void    do_sb(t_stack **stack_b)
{
    swap(*stack_b);
    ft_putstr("sb\n");
```

```
    print_status(NULL, *stack_b);
}


/* faire_ss :
* Échange les 2 premiers éléments de la pile a et les 2 premiers éléments
* de pile b.
* Imprime "ss" sur la sortie standard.
*/
void    do_ss(t_stack **stack_a, t_stack **stack_b)
{
    swap(*stack_a);
    swap(*stack_b);
    ft_putstr("ss\n");
    print_status(*stack_a, *stack_b);
}
```

static void swap(t_stack *stack)

Cette fonction prend une pile en entrée et échange les valeurs et les indices des deux premiers éléments de la pile. Elle commence par vérifier si stack est NULL ou si le deuxième élément de la pile est NULL. Si c'est le cas, elle retourne immédiatement, car il n'y a rien à échanger.
Ensuite, elle stocke la valeur de l'élément au sommet de la pile dans la variable tmp, puis fait passer la valeur du deuxième élément de la pile à la place de la valeur du premier. Enfin, elle remet la valeur stockée dans tmp à la place de la valeur du deuxième élément.
Elle répète le même processus pour les indices des deux éléments.

void do_sa(t_stack **stack_a)

Cette fonction appelle swap pour échanger les deux premiers éléments de la pile a, puis elle imprime "sa" à la sortie standard pour indiquer qu'un mouvement "sa" a été effectué.

void do_sb(t_stack **stack_b)

De même, cette fonction appelle swap pour échanger les deux premiers éléments de la pile b, puis elle imprime "sb" à la sortie standard pour indiquer qu'un mouvement "sb" a été effectué.

void do_ss(t_stack **stack_a, t_stack **stack_b)

Cette fonction appelle swap pour échanger les deux premiers éléments des deux piles a et b, puis elle imprime "ss" à la sortie standard pour indiquer qu'un mouvement "ss" a été effectué.