28/11/23

Backtracking : N Reines

On place des reines jusqu'a ce qu'on puisse plus
Si on a pas reussi à tout placer on revient en arrière et on place la reine d'avant sur une autre position possible
On recommence à l'infini tant qu'on a pas tester toutes les solutions

MAX 1 reine par colonne
On change de colonne en allant toujours vers l'avant

Structure de données :
bool board [n][n]
↓
Enum

Fonctions :
init_board (n, board)
update_board (board, pos x, pos y)
place_queen (board, pos x, pos y)
nbr_solutions (cont, col, board)

On met à jour devant le tableau

Piles : structure abstraite LIFO : last in first out

Push : ajouter un element Pop : retirer l'element en haut de la pile
is_empty : check si la pile est vide Peek : retourner l'element en haut de la pile sans dépiler

length : dépile dans une pile temp et on compte tant que !is_empty

1ère version pile taille fixe avec un tableau

```
typedef struct _stack {
    int data [Max_capacity];
    int top;
} stack
```

initialisation :

```
top = -1;
```

```
void stack_init (stack *s)
s -> top = -1;

stack s;
stack_init (&s);
```

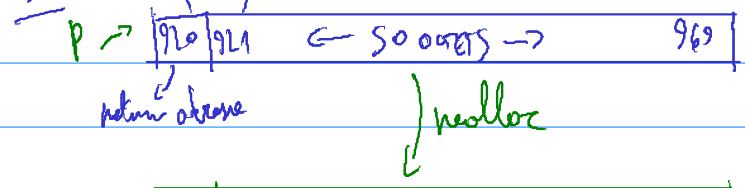
```
bool is_empty (Stack S) { return S.top == -1; }

Void stack_push (Stack *S, int val) {
    if (top < MAX_capacity - 1) {
        S -> top ++;
        S -> data [S -> top] = val;
    }
}

int stack_pop (Stack *S) {
    if (!is_empty) {
        S -> top -= 1.
        return S -> data [S -> top+1];
}

int stack_peek (Stack *S) {
    if (S -> top >= 0) { return S -> data [S -> top]
```

Gestion d'erreur : ponique : assert(); #include <assert.h>
Void assert (int expression)
Si expression == 0 -> erreur
Sinon continue normal

Pour désactiver : -DNDEBUG

Sert à faire des tests unitaires, tester des conditions
Typiquement pour vérifier que les pointeurs != NULL

05/12/23 MALLOC : void * malloc (size_t size); void * = pointeur vers / tout type
REALLOC : void * realloc (void * ptr, size_t size);
Ex: malloc (50)



920 + 50 - 1 = 969

type * P = malloc (50)
P = realloc (P, 1000)

int * data = malloc (Size * sizeof (int))
calloc (Size * sizeof (int))
(Size * sizeof (* data)) -> permet de gérer plusieurs type sans devoir changer sizeof

sizeof (int) = 32 dans 99% des cas, permet la portabilité

CAPACITY 500

```
typedef struct stack {
    int *data;
    int top;
}
```

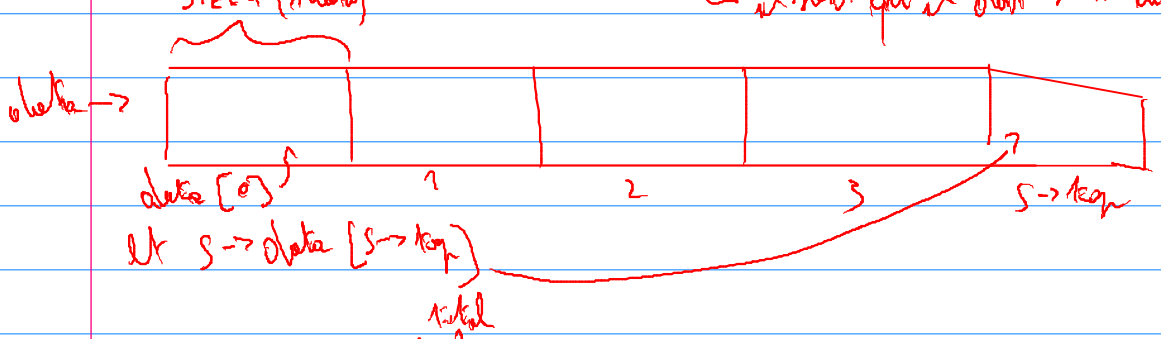
malloc : pas un tableau, un espace mémoire
PAS OU BIEN de LIBERER APRÈS UTILISATION
free(*data) -> * vers le malloc

< pendantic — pointing address

```
void stack_destroy(stack *S){ S->top = -1; free(S->data); }
```

PAS D'INDICATION DE LIBERATION : mettre S->data = NULL POUR CRASH SI C'EST FREE

PUSH : S->top++ ; S->data[S->top] = VAL ;
↑ il sait qu'il doit sauter de type * n (int)
taille int

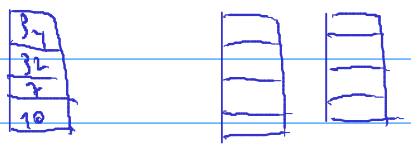


on connaît que la taille de la stack : un objet int capacity

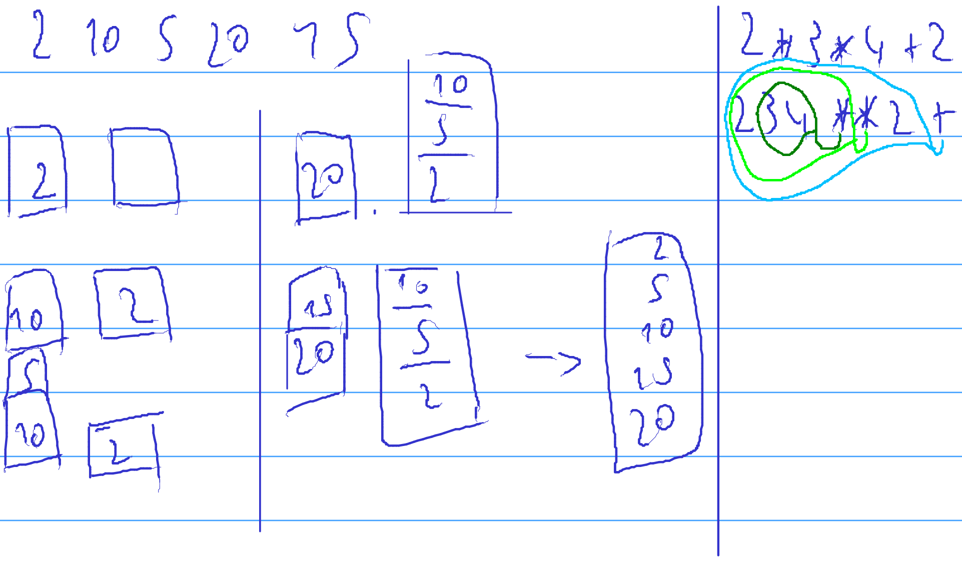
```
if(S->top < capacity)
    on push
else
    on realloc(S->data, (S->top+1)*sizeof(S->data)) ?? à vérifier
```

---

trier un array avec 2 stacks :



On empile à gauche tant que nbr actuel < que top_gauche, après on range à droite et on met nbr actuel à gauche, si on veut insérer nbr actuel < que top gauche et < que top_droite, on dépile à droite -> gauche jusqu'à ce qu'on trouve à droite < nbr actuel, à la fin :
on dépile à gauche (croissant), à droite (décroissant)



Calculatrice polonaise inverse : 2 + 3 = 2 3 +   OPÉRANDE   OPÉRATEUR

Chaque 2 opérande applique l'opérateur, si c'est une parenthèse on met sur une pile avec sa priorité, si c'est un opérateur de priorité plus élevée on met dans la pile si parenthèse fermée on dépile tout jusqu'à la parenthèse ouverte)

tant que (si priorité < que sommet on dépile tout)

inf-post

la parenthèse est représentée dans la stack par deux opérateurs de suite

lorsqu'il y a 2 opérateurs on applique l'opérateur de gauche aux deux opérandes suivants (de droite à gauche) puis on remet le résultat au même endroit de la pile (dépile calcul rempile) puis on passe au prochain couple (opérande, opérande calculé) et opérateur

^ : 3	char *inf_to_post(char *infix) (String)
*/ : 2	
+ - : 1	
() : 0	

12/12/23 in -> char *infix   out char *postfix

1: opérandes on met dans postfix

2ième on check parenthèses : (on met dans stack,) on dépile tout temp stack

2: opérateur on check

Evaluation postfixée: 234+*5-=?

on met toutes les opérandes sur pile jusqu'à un opérateur : while (! opérateur)

opérateur on dépile les 2 opérandes en haut de la pile et on fait l'opération et on met le résultat dans la pile

Tant que postfixe n'est pas vide ou continue

Liste-chaînée : stack -> 0 , stack -> elmnt -> 0,

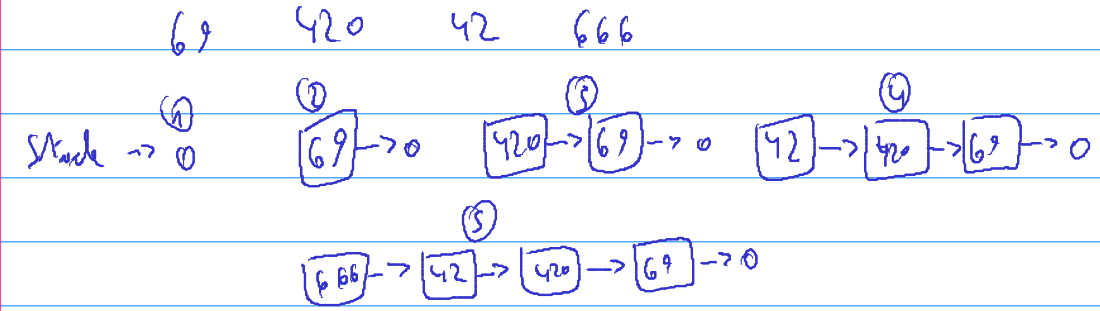
stack -> elmnt -> elmnt -> 0,

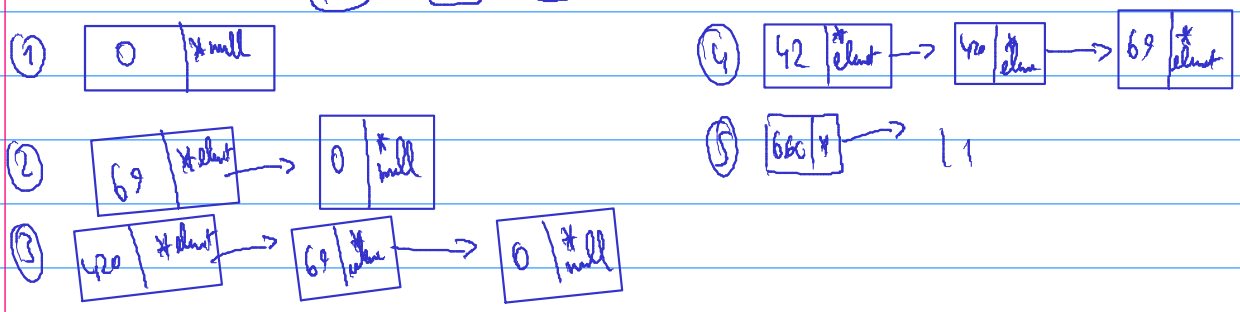
stack -> elmnt -> elmnt -> elmnt -> 0

```
element {
  int val;
  struct element* next;
}
```

---

69 420 42 666





Qui est-ce qui il refont pas oublier lors d'un pop en liste-chaînée ? -> free(elmnt)

après avoir récupéré la

Quand est-ce que on doit utiliser un pointeur dans l'appel d'une fonction ?

quand on modifie la valeur

---

file d'attente : FIFO

enfile à la fin on défile au début

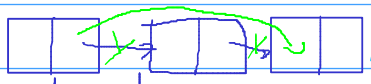
qui est qui on change quand on enfile : tail ++

" défile : head --

09/01/24

liste-triée : utilise liste-chaînée
int data ;
struct _element *next ;

insertion : 3 cas
1. liste vide
2. on insère avant
3. On insère après



if >1 val, on extr
(il est là) la 1ère

1. On cherche où placer l'élément
2. on stocke le pointeur temp
3. on insère courant
4. on lie avec temp

extraction : 3 cas
1. pos le premier, esctraire -> reller
2. .