Course descriptions : semester 2, Data Science and Big Data..

Programmation & Algorithmique II

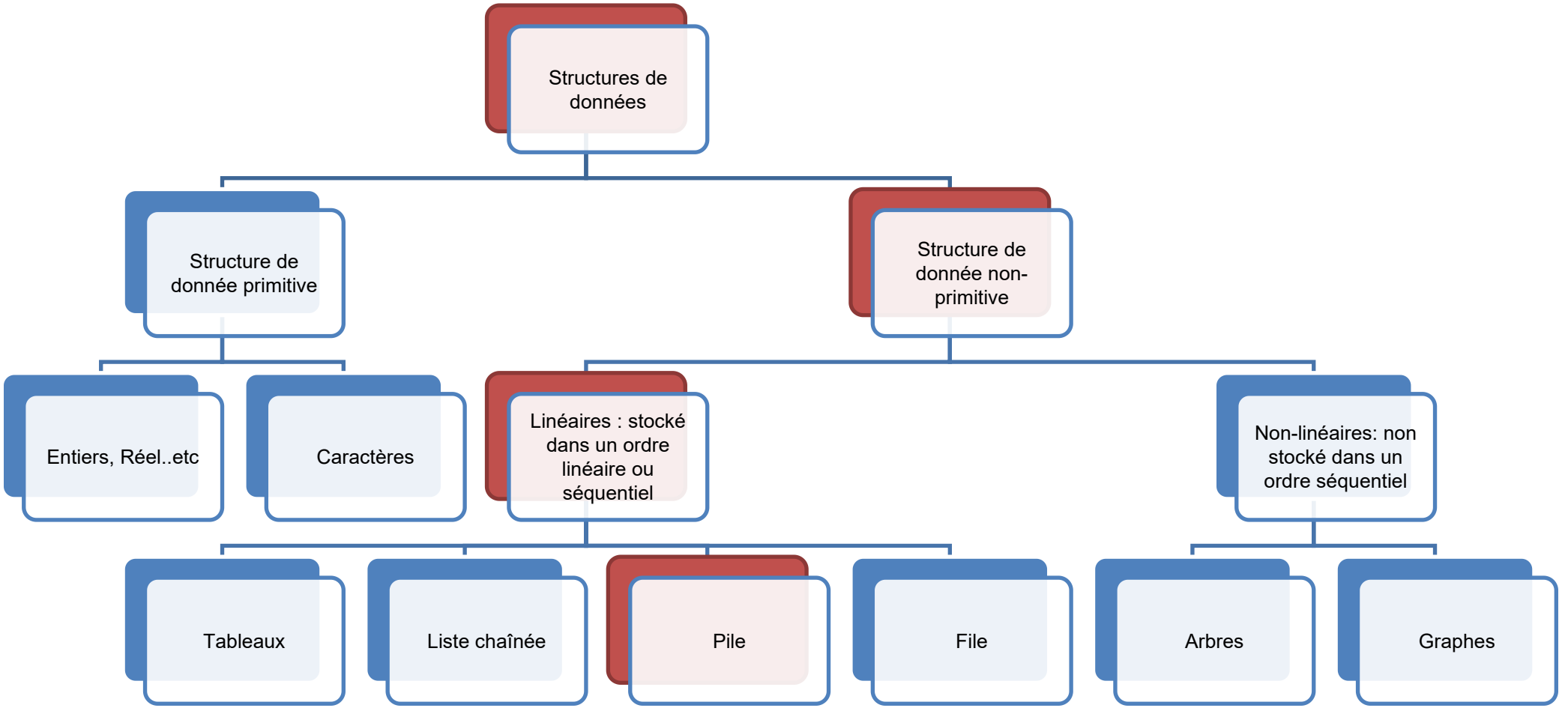
CM 7 : Les Piles

PLAN

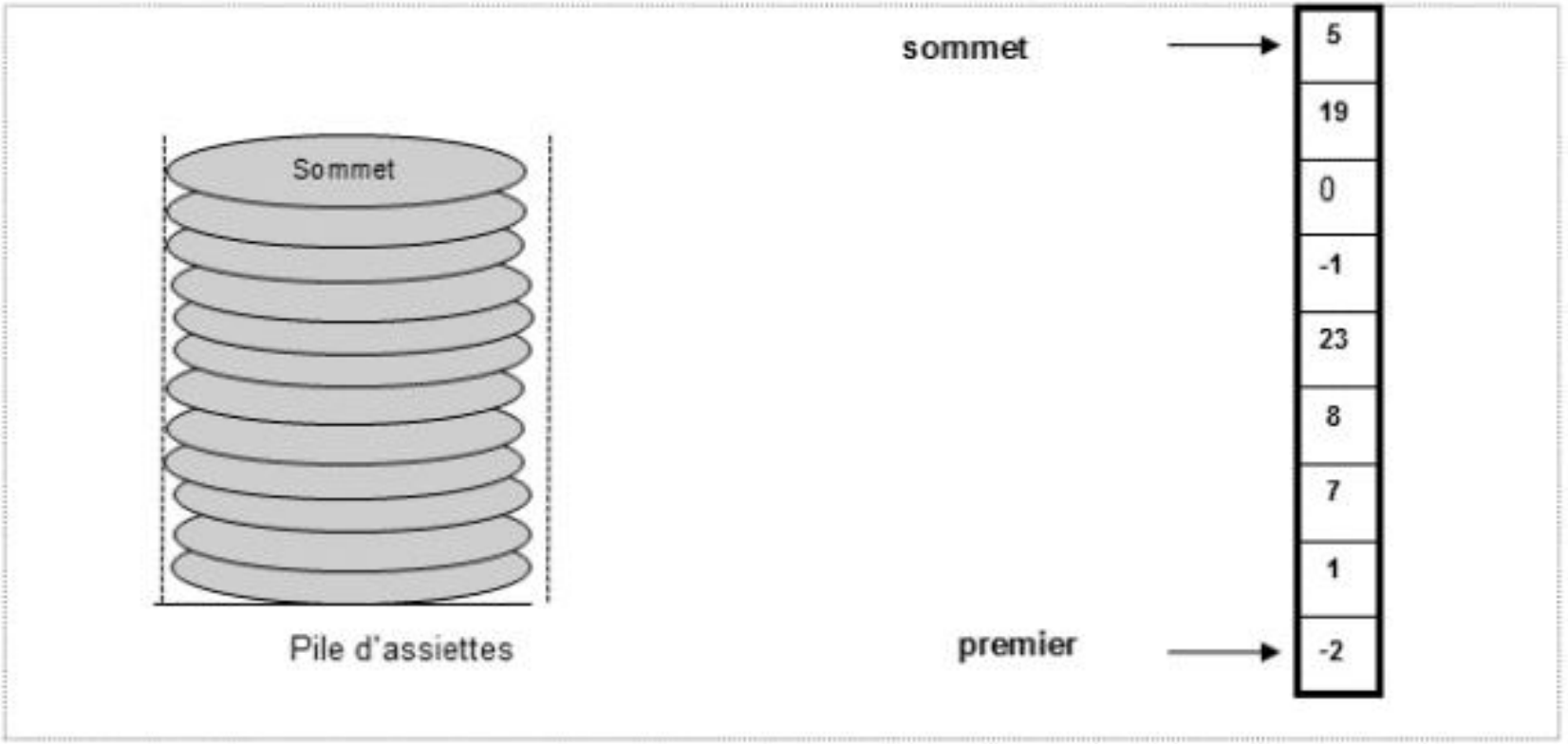
- Les Piles
  - Représentation contiguë
  - Représentation chaînée
- Applications des piles.
  - Inverser une pile (liste - cas particulière)
    - Récursion
  - Vérificateur de parenthèses
  - Conversion d'une expression mathématique
    - **Inifix à postfix**
    - Inifix à prefix
    - Postfix à inifix
    - Postfix à prefix
    - Prefix à postfix
  - Évaluation d'une expression postfix

CLASSIFICATION DES STRUCTURES DE DONNÉES

- **Structures de données primitives et non primitives**



- **QU'EST-CE QU'UNE PILE ?**
  - Une pile est une structure de données dans laquelle on peut ajouter et supprimer des éléments suivant la règle du dernier arrivé premier sorti
    - ou encore LIFO (Last In First Out).



REPRÉSENTATION D'UNE PILE

- **Représentation contiguë (par tableau)**
  - Les éléments de la pile sont rangés dans un tableau
  - Un entier représente la position du sommet de la pile

- **Représentation chaînée (par pointeurs → liste chaînée)**
  - Les éléments de la pile sont chaînés entre eux
  - Un pointeur sur le premier élément désigne la pile et représente le sommet de cette pile
  - Une pile vide est représentée par le pointeur NULL

REPRÉSENTATION D'UNE PILE

- **Les primitives de gestion des piles**
  - Initialiser : cette fonction crée une pile vide.
  - EstVide : renvoie 1 si la pile est vide, 0 sinon.
  - EstPleine : renvoie 1 si la pile est pleine, 0 sinon.
  - AccederSommet : cette fonction permet l'accès à l'information contenue dans le sommet de la pile.
  - Empiler : cette fonction permet d'ajouter un élément au sommet de la pile.
    - La fonction renvoie un code d'erreur si besoin en cas de manque de mémoire.
  - Depiler : cette fonction supprime le sommet de la pile. L'élément supprimé est retourné par la fonction Depiler pour pouvoir être utilisé.
  - Vider : cette fonction vide la pile.
  - Detruire : cette fonction permet de détruire la pile.

IMPLÉMENTATION SOUS FORME DE TABLEAU

- **Types**
  - Pour implémenter une pile sous forme de tableau, on crée la structure de données suivante.

```
typedef float TypeDonnee;
typedef struct
{
    int nb_elem; /* nombre d'éléments dans la pile */
    int nb_elem_max; /* capacité de la pile */
    TypeDonnee *tab; /* tableau contenant les éléments */
}Pile;
```

IMPLÉMENTATION SOUS FORME DE TABLEAU

- **Créer une pile vide.**
  - La fonction permettant de créer une pile vide est la suivante :

```
Pile Initialiser(int nb_max)
{
    Pile pilevide;
    pilevide.nb_elem = 0; /* la pile est vide */
    pilevide.nb_elem_max = nb_max; /* capacité nb_max */
/* allocation des éléments : */
    pilevide.tab = (TypeDonnee*)malloc(nb_max*sizeof(TypeDonnee));
    return pilevide;
}
```

IMPLÉMENTATION SOUS FORME DE TABLEAU

- **Pile vide,**
  - La fonction permettant de savoir si la pile est vide est la suivante. La fonction renvoie
    - 1 si le nombre d'éléments est égal à 0.
    - La fonction renvoie 0 dans le cas contraire.

```
int EstVide(Pile P)
{
  /* retourne 1 si le nombre d'éléments vaut 0 */
  return (P.nb_elem == 0) ? 1 : 0;
}
```

IMPLÉMENTATION SOUS FORME DE TABLEAU

- **Pile pleine**
  - La fonction permettant de savoir si la pile est pleine est la suivante :

```
int EstPleine(Pile P)
{
  /* retourne 1 si le nombre d'éléments est >= */
  /* au nombre d'éléments maximum et 0 sinon */
  return (P.nb_elem >= P.nb_elem_max) ? 1 : 0;
}
```

IMPLÉMENTATION SOUS FORME DE TABLEAU

- **Accéder au sommet de la pile**
  - La fonction effectue un passage par adresse pour ressortir le sommet
    - Le sommet de la pile est le dernier élément entré, qui est le dernier élément du tableau
  - La fonction permet d'accéder au sommet de la pile et renvoie le code d'erreur 1 en cas de liste vide et 0 sinon

```
int AccederSommet(Pile P, TypeDonnee *pelem)
{
  if (EstVide(P))
    return 1; /* on retourne un code d'erreur */
  *pelem = P.tab[P.nb_elem-1]; /* on renvoie l'élément */
    return 0;
}
```

IMPLÉMENTATION SOUS FORME DE TABLEAU

- **Ajouter un élément au sommet**
  - Pour modifier le nombre d'éléments de la pile, il faut passer la pile par adresse. La fonction Empiler, qui renvoie 1 en cas d'erreur et 0 dans le cas contraire, est la suivante :

```
int Empiler(Pile* pP, TypeDonnee elem)
{
  if (EstPleine(*pP))
    return 1; /* on ne peut pas rajouter d'élément */
  pP->tab[pP->nb_elem] = elem; /* ajout d'un élément */
  pP->nb_elem++; /* incrémentation du nombre d'éléments */
  return 0;
}
```

IMPLÉMENTATION SOUS FORME DE TABLEAU

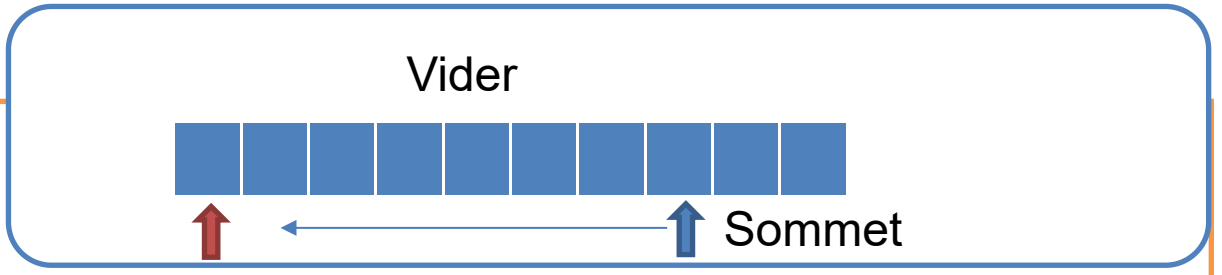
- **Supprimer un élément**
  - La fonction Depiler supprime le sommet de la pile en cas de pile non vide. La fonction renvoie 1 en cas d'erreur (pile vide), et 0 en cas de succès.

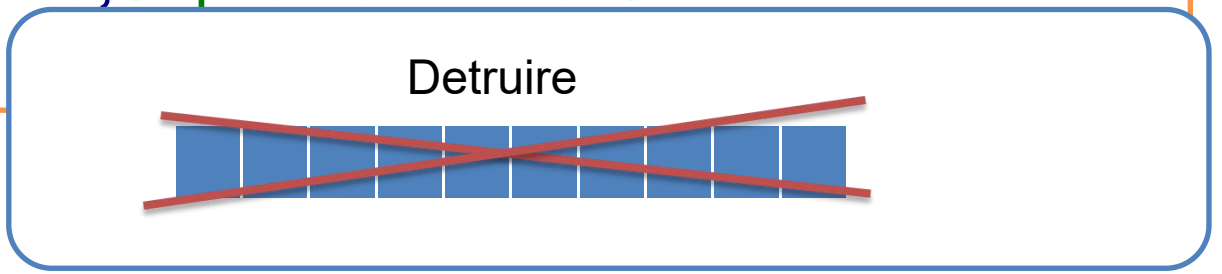
```
int Depiler(Pile *pP, TypeDonnee *pelem)
{
  if (EstVide(*pP))
    return 1; /* on ne peut pas supprimer d'élément */
  *pelem = pP->tab[pP->nb_elem-1]; /* on renvoie le sommet */
  pP->nb_elem--; /* décrémentation du nombre d'éléments */
  return 0;
}
```

IMPLÉMENTATION SOUS FORME DE TABLEAU

- **Vider et détruire**

```
void Vider(Pile *pP)
{
  pP->nb_elem = 0; /* réinitialisation du nombre d'éléments */
}
void Detruire(Pile *pP)
{
  if (pP->nb_elem_max != 0)
  free(pP->tab); /* libération de mémoire */
  pP->nb_elem = 0;
  pP->nb_elem_max = 0; /* pile de taille 0 */
}
```





浙江科技學院 ZHEJIANG UNIVERSITY OF SCIENCE & TECHNOLOGY

CERGY PARIS UNIVERSITÉ

IMPLÉMENTATION SOUS FORME DE LISTE CHAÎNÉE

- **Types**
  - Pour implémenter une pile sous forme de liste chaînée, on crée la structure de données suivante.

```
typedef float TypeDonnee;
typedef struct Cell
{
TypeDonnee donnee;
struct Cell *suivant; /* pointeur sur la cellule suivante */
}TypeCellule;
typedef TypeCellule* Pile; /* la pile est un pointeur */
                           /* sur la tête de liste */
```

Attention !

- Typedef Cellul* Pile
- Donc Pile * pP := Cellul ** pP

IMPLÉMENTATION SOUS FORME DE LISTE CHAÎNÉE

- **Créer une pile vide**
  - La fonction permettant de créer une pile vide est la suivante :

```
Pile Initialiser()
{
return NULL; /* on retourne une liste vide */
}
```

IMPLÉMENTATION SOUS FORME DE LISTE CHAÎNÉE

- **Pile vide**
  - La fonction permettant de savoir si la pile est vide est la suivante. La fonction renvoie 1 si la pile est vide, et renvoie 0 dans le cas contraire.

```
int EstVide(Pile P)
{
  /* renvoie 1 si la liste est vide */
  return (P == NULL) ? 1 : 0;
}
```

IMPLÉMENTATION SOUS FORME DE LISTE CHAÎNÉE

- **Pile pleine**
  - La fonction permettant de savoir si la pile est pleine est la suivante :

```
int EstPleine(Pile P)
{
  return 0; /* une liste chaînée n'est jamais pleine */
}
```

IMPLÉMENTATION SOUS FORME DE LISTE CHAÎNÉE

- **Accéder au sommet de la pile**
  - Le sommet de la pile est le dernier élément entré qui est la tête de liste. La fonction renvoie 1 en cas de liste vide, 0 sinon:

```
int AccederSommet(Pile P, TypeDonnee *pelem)
{
  if (EstVide(P))
    return 1; /* on retourne un code d'erreur */
  *pelem = P->donnee; /* on renvoie l'élément */
  return 0;
}
```

IMPLÉMENTATION SOUS FORME DE LISTE CHAÎNÉE

- **Ajouter un élément au sommet**
  - La fonction d'ajout d'un élément est une fonction d'insertion en tête de liste

```
void Empiler(Pile* pP, TypeDonnee elem)
{
  Pile q; // TypeCellule*
  q = (TypeCellule*)malloc(sizeof(TypeCellule)); /* allocation */
  q->donnee = elem; /* ajout de l'élément à empiler */
  q->suivant = *pP; /* insertion en tête de liste */
  *pP =q; /* mise à jour de la tête de liste */
}
```

IMPLÉMENTATION SOUS FORME DE LISTE CHAÎNÉE

- **Supprimer un élément**
  - La fonction Depiler supprime la tête de liste en cas de pile non vide. La fonction renvoie 1 en cas d'erreur, et 0 en cas de succès. La pile est passée par adresse, on a donc un double pointeur

```
int Depiler(Pile *pP, TypeDonnee *pelem)
{
  Pile q;
  if (EstVide(*pP))
    return 1; /* on ne peut pas supprimer d'élément */
  *pelem = (*pP)->donnee; /* on renvoie l'élément de tête */
  q = *pP; /* mémorisation d'adresse de la première cellule */
  *pP = (*pP)->suivant; /* passage au suivant */
  free(q); /* destruction de la cellule mémorisée */
  return 0;
}
```

IMPLÉMENTATION SOUS FORME DE LISTE CHAÎNÉE

- **Vider et détruire**
  - La destruction de la liste doit libérer toute la mémoire de la liste chaînée (destruction individuelle des cellules).

```
void Detruire(Pile *pP)
{
  Pile q;
  while (*pP != NULL) /* parcours de la liste */
  {
    q = *pP; /* mémorisation de l'adresse */
    /* passage au suivant avant destruction : */
    *pP = (*pP)->suivant;
    free(q); /* destruction de la cellule mémorisée */
  }
  *pP = NULL; /* liste vide */
}
```

IMPLÉMENTATION SOUS FORME DE LISTE CHAÎNÉE

- **Vider et détruire**
  - La destruction de la liste doit libérer toute la mémoire de la liste chaînée (destruction individuelle des cellules).

```
void Vider(Pile *pP)
{
  Detruire(pP); /* destruction de la liste */
  *pP = NULL; /* liste vide */
}
```

COMPARAISON ENTRE TABLEAUX ET LISTES CHAÎNÉES

- Dans les deux types de gestion des piles, chaque primitive ne prend que quelques opérations (i.e. complexité en temps constant).

- Par contre, la gestion par listes chaînées, présente l'énorme avantage que la pile a une capacité virtuellement illimitée (limitée seulement par la capacité de la mémoire centrale), la mémoire étant allouée à mesure des besoins.

- Au contraire, dans la gestion par tableaux, la mémoire est allouée au départ avec une capacité fixée.

- Écrire un algorithme utilisant une pile (implémentée sous forme de liste chaînée) qui affiche une liste chaînée d'entiers à l'envers.

```
void Affichierpile(Pile P)
{
        Pile q;
        q = P;
        while (q != NULL)
        {
                printf("%d\t", q->donnee);
                q = q->suivant;
        }
        puts("");
}
```

- **Utilisation**
  - De nombreuse applications s'appuient sur l'utilisation d'une pile, on peut citer:
    - Dans un navigateur web, une pile sert à mémoriser les pages Web visités.
      - L'adresse de chaque nouvelle page visitée est empilée et l'utilisateur dépile l'adresse de la page précédente en cliquant le bouton « Afficher la page précédente ».
    - L'évaluation des expressions mathématiques en notation post-fixée (ou polonaise inverse) utilise une pile.
    - La fonction « Annuler la frappe » (en anglais « undo ») d'un traitement de texte mémorise les modifications apportées au texte dans une pile.
    - Vérification de parenthèse d'une chaine de caractères.
    - La récursivité (une fonction qui fait appel a elle-même). etc.

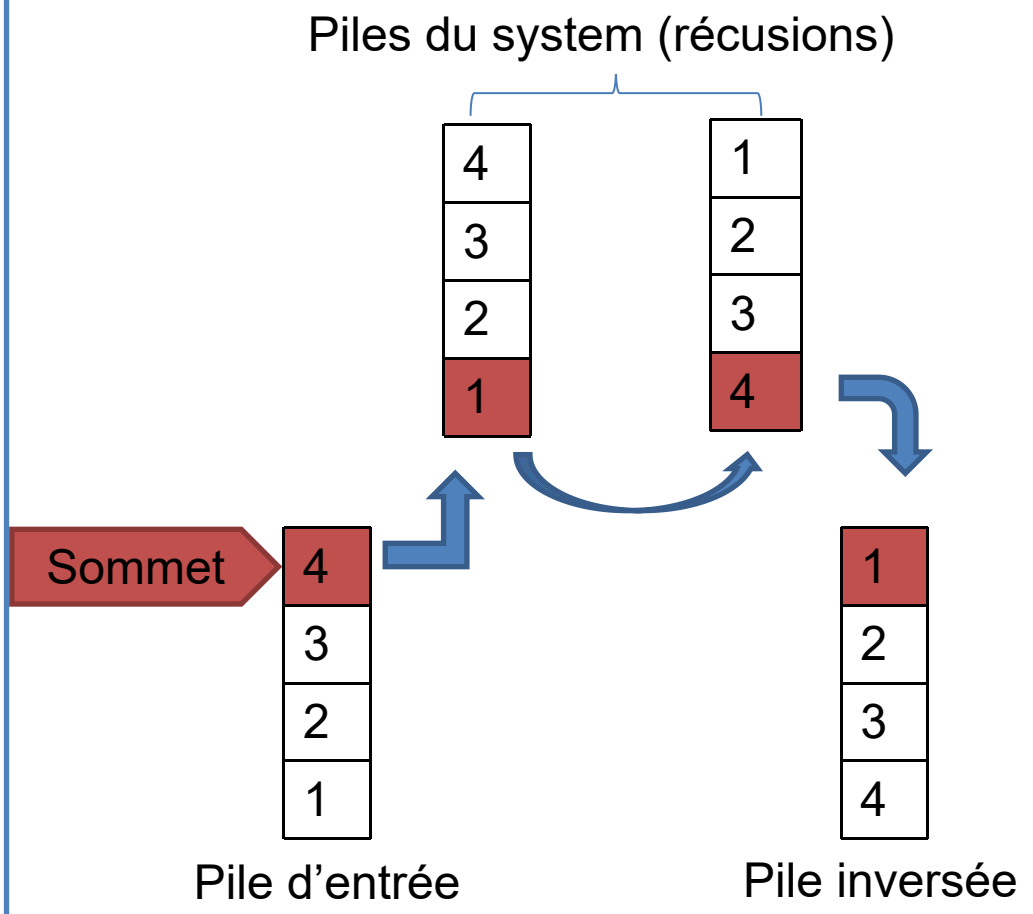
APPLICATIONS DES PILES

- **Inverser une pile**
  - Comment inverser les éléments de la pile sans utiliser aucune autre structure de données ni une autre pile.
    - En utilisant la récursivité (pile du système)
      - Lorsque vous entrez dans la pile, on dépile les éléments de pile à chaque appel récursive suivant jusqu'à ce que la pile soit vide.
      - Puis empiler ces éléments un par un en sortant de la récursivité.
      - Résultat:
        - Les éléments seront inversés

APPLICATIONS DES PILES

```
void inverserPile(Pile* pile)
{
        int d;
        if (estVide(pile))
                return;
        d = depiler(pile);
        inverserPile(pile);
        inserEnbas(pile, d);
}

void inserEnbas(Pile* pile, int v)
{
        if (estVide(pile))
                empiler(pile, v);
        else
        {
                int temp = depiler(pile);
                inserEnbas(pile, v);
                empiler(pile, temp);
        }
}
```



APPLICATIONS DES PILES

- **Vérificateur de correspondance des parenthèses**
  - Étant donné une expression entre parenthèses,
  - Tâche:
    - testez si l'expression est correctement entre parenthèses.
  - Exemples:
    - ( )({ } [({ } { } ( ))]) est convenable (correct)
    - ( ){ [ ] n'est pas convenable
    - ({) } n'est pas convenable
    -)([ ] n'est pas convenable
    - ([ ])) n'est pas convenable

APPLICATIONS DES PILES

- **Approche :**
  - Chaque fois qu'une parenthèse gauche est rencontrée, elle est empiler dans la pile
  - Chaque fois qu'une parenthèse droite est rencontrée, depiler de la pile et vérifiez si les parenthèses correspondent
  - Fonctionner pour plusieurs types de parenthèses

    (), {}, []

APPLICATIONS DES PILES

```
int estParenthesesEquilibrees(char* expn)
{
        Pile *pile;
        pile = initialisation();
        int i = 0;
        char ch;
        while ((ch = expn[i++]) != '\0')
        {
                switch (ch)
                {
                        case '{':
                        case '[':
                        case '(':
                                empiler(pile,ch);
                        break;
                        case '}':
                                if (depiler(pile) != '{')
                                        return 0;
                                break;
                        case ']':
                                if (depiler(pile) != '[')
                                        return 0;
                                break;
                        case ')':
                                if (depiler(pile) != '(')
                                        return 0;
                                break;
        }       }
        return estVide(pile); }
```

APPLICATIONS DES PILES

```
int main()
{
        char expn[50] = "{()}";
        int valeur = estParenthesesEquilibrees(expn);
        printf("\n Expression donnée: %s\n", expn);
        printf("\n Resultat après la vérification par estParenthesesEquilibrees: %d\n", valeur);
        return 0;
}
```

QUIZ

On ajoute, dans cet ordre, les valeurs A, B, C, D, E et F à une structure linéaire vide. Pour chacun des ordres de sortie suivant, indiquez si la structure en question peut être : une pile ou non?

A B C D E F
D E C B F A
B D E F A C
F E D C B A

Choix:

A- Oui
B- Non

5 minute

APPLICATIONS DES PILES

- **Expressions infixes, préfixes et postfixes :**
  - Lorsque nous avons une expression algébrique comme A + B, nous savons que la variable A est ajoutée à la variable B.
  - Ce type d'expression est appelé expression infixe car l'opérateur «+» est entre les opérandes A et l'opérande B.
  - Considérons une autre expression infixe A + B * C.
    - il y a un problème dans lequel l'ordre + et * fonctionne.
    - Est-ce que A et B sont ajoutés en premier, puis le résultat est multiplié.
    - Ou bien B et C sont multipliés en premier, puis le résultat est ajouté à A.

Cela rend l'expression ambiguë. Pour faire face à cette ambiguïté, nous définissons la règle de priorité ou utilisons des parenthèses pour supprimer l'ambiguïté.
Exemple : A + B * C = 4 + 3 * 7 = 7 * 7 = 49

APPLICATIONS DES PILES

- **Expressions infixes, préfixes et postfixes :**
  - **Expression infixe**: dans cette notation, nous plaçons l'opérateur au milieu des opérandes.

    <opérande> <opérateur> <opérande>
  - **Expressions de préfixe**: dans cette notation, nous plaçons l'opérateur au début des opérandes.

    <opérateur> <opérande> <opérande>
  - **Expression Postfix**: Dans cette notation, nous plaçons l'opérateur à la fin des opérandes.

    <opérande> <opérande> <opérateur>.

Infix	Prefix	Postfix
a + b	+ a b	a b +
a + b * c	+ a * b c	a b c * +
(a + b) * (c - d)	* + a b - c d	a b + c d - *

Les expressions infixes sont ambiguës et ont besoin de parenthèses pour les rendre sans ambiguïté. Alors que les notations de Postfix et de préfixe n'ont pas besoin de parenthèses

APPLICATIONS DES PILES

- **Conversion d'infixe en postfix : P=A + (B / C - (D * E ^ F) + G) * H**

1. Parcourez l'expression d'infixe de gauche à droite.
2. Lire les symboles un par un
   a) Si le symbole lu est une parenthèses gauche " (", empilez-le dans la pile.
   b) Si le symbole lu est un opérande (nombre ou caractère), affichez-le directement dans l'expression postfixe (sortie).
   c) Si le symbole lu est une parenthèse droite ")",
      a) continuez à dépiler tous les éléments de la pile et affichez-les dans l'expression postfix jusqu'à ce que nous obtenions la parenthèse gauche " (" correspondante.
      b) Dépiler "(" de la pile et ne l'ajoutez pas à l'expression postfix
   d) Si le symbole lu est un opérateur, continuez à dépiler tous les opérateurs de la pile et affichez-les dans l'expression postfixe, si et seulement si la priorité de l'opérateur qui se trouve au sommet de la pile est supérieure à (ou supérieure à ou égal) à la priorité de l'opérateur lu et ensuite empiler l'opérateur lu dans la pile, sinon, empiler l'opérateur scanné sur la pile.
3. Vider la pile à la fin de lecture

APPLICATIONS DES PILES

- **Priorité des opérateurs**

Opérateur exponentiel	^	Priorité la plus élevée
Multiplication / Division	*, /	Priorité suivante
Addition soustraction	+, -	moins priorité

- **Exemple:** La méthode de conversion de l'expression infixe A + B * C en forme postfixe est
  - A + B * C Infix expression
  - A + (B * C) Expression parentheses
  - A + (B C *) Convertir la multiplication
  - A (B C *) + Convertir l'addition
  - A B C * + Postfix expression

P=A + (B / C - (D * E ^ F) + G) * H

Symbole lu	pile	Postfix expression (output)
A		A
+	+	A
(	+ (	A
B	+ (	A B
/	+ (/	A B
C	+ (/	A B C
-	+ (-	A B C /
(	+ (- (	A B C /
D	+ (- (	A B C / D
*	+ (- (*	A B C / D
E	+ (- (*	A B C / D E
^	+ (- (* ^	A B C / D E
F	+ (- (* ^	A B C / D E F
)	+ (-	A B C / D E F ^ *
+	+ (+	A B C / D E F ^ * -
G	+ (+	A B C / D E F ^ * - G
)	+	A B C / D E F ^ * - G +
*	+ *	A B C / D E F ^ * - G +
H	+ *	A B C / D E F ^ * - G + H
Vider la pile		A B C / D E F ^ * - G + H * +

```
void infix2Postfix(char* expn,char* output){
.   Pile *stk;
    stk = initialisation();
    char ch, op;
    int i = 0;
    int index =0;
    int digit = 0;
    while ((ch = expn[i++]) != '\0'){
            if (isdigit(ch)){output[index++] = ch;
                digit=1;
            }else{
                if(digit){  output[index++] = ' ';
                            digit = 0;
                }
                switch (ch){
                    case '+':
                    case '-':
                    case '*':
                    case '/':
                    case '%':
                    case '^':
                    while (!estVide(stk) && precedence(ch) <= precedence(Sommet(stk))){
                                op = depiler(stk);
                                output[index++] = op;
                                output[index++] = ' ';
                    }
                    empiler(stk, ch); break;
                    case '(': empiler(stk, ch); break;
                    case ')':
                    while (!estVide(stk) && (op = depiler(stk)) != '('){
                                output[index++] = op;
                                output[index++] = ' ';
                    }
                    break;
```

```
            }           }           }

            while (!estVide(stk))
            {
                        op = depiler(stk);
                        output[index++] = op;
                        output[index++] = ' ';
            }
            output[index++] = '\0';
}
```

```
int precedence(char x)
{
            if (x == '(')
                        return(0);
            if (x == '+' || x == '-')
                        return(1);
            if (x == '*' || x == '/' || x == '%')
                        return(2);
            if (x == '^')
                        return(3);
            return(4);
}
```

```
int precedence(char x)
{
        if (x == '(')
                return(0);
        if (x == '+' || x == '-')
                return(1);
        if (x == '*' || x == '/' || x == '%')
                return(2);
        if (x == '^')
                return(3);
        return(4);
}
```

APPLICATIONS DES PILES

- **Évaluation de l'expression postfix** 6 5 2 3 + 8 * + 3 + *
  - Créez une pile pour stocker des valeurs ou des opérandes.
  - Parcourez l'expression donnée et procédez comme suit pour chaque élément:
    - Si l'élément est un nombre, empiler-le dans la pile.
    - Si l'élément est un opérateur, dépiler deux valeurs de la pile. Évaluez l'opérateur sur les valeurs et insérez le résultat dans la pile.
    - Lorsque l'expression est complètement analysée (parcouru), le nombre au sommet de la pile est le résultat.

APPLICATIONS DES PILES

- **Exemple**
  - Évaluez l'expression postfix: 6 5 2 3 + 8 * + 3 + *

Symbole	Opérande 1	Opérande 2	Valeur	Pile
6				6
5				6, 5
2				6, 5, 2
3				6, 5, 2, 3
+	2	3	5	6, 5, 5
8	2	3	5	6, 5, 5, 8
*	5	8	40	6, 5, 40
+	5	40	45	6, 45
3	5	40	45	6, 45, 3
+	45	3	48	6, 48
*	6	48	288	288

APPLICATIONS DES PILES

```
int postfixEvaluate(char* postfx)
{
        Pile *s;
        s = initialisation();
        int i = 0, op1, op2;
        char ch;
        int digit = 0;
        int value = 0;
        while ((ch = postfx[i++]) != '\0')
        {       if (isdigit(ch)){
                        digit = 1;
                        value = value * 10 + (ch - '0');
                }else if (ch == ' '){
                        if (digit == 1){
                                empiler(s, value);
                                digit = 0;
                                value = 0;
                        }
                }else{
                        op2 = depiler(s);
                        op1 = depiler(s);
                        switch (ch){
                                case '+': empiler(s, op1 + op2); break;
                                case '-': empiler(s, op1 - op2); break;
                                case '*': empiler(s, op1 * op2); break;
                                case '/': empiler(s, op1 / op2); break;

        }}}
        return Sommet(s);
}
```

APPLICATIONS DES PILES

```
int main()
{
        char postfx[50] = "6 5 2 3 + 8 * + 3 + *";
        int value = postfixEvaluate (postfx);
        printf("\n Postfix Expression: %s\n", postfx);
        printf("\n Resultat après l' évaluation: %d\n", value);
        return 0;
}
```