ITI 1521. Introduction à l'informatique II*

Marcel Turcotte
École de science informatique et de génie électrique
Université d'Ottawa

Version du 8 février 2012

Résumé

- Type abstrait de données
  - Piles (stacks)

*. Pensez-y, n'imprimez ces notes de cours que si c'est nécessaire!

Les piles (*stacks*)

Une **pile** est un **type abstrait de données** semblable aux piles physiques.

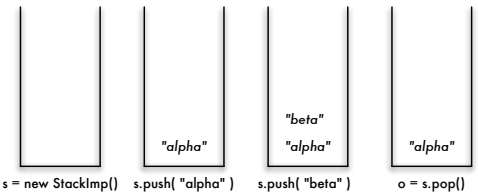
- Journaux ;
- PEZ ;
- Assiettes ;
- Cabarets ;

L'analogie avec le distributeur d'assiettes que l'on retrouve dans une cafétéria est particulièrement intéressante, parce que 1) **l'accès est limité à l'élément du dessus** et 2) **il faut enlever une-à-une les assiettes du dessus afin d'accéder à celles du dessous.**

Définition

Une **pile** est une structure de données **linéaire** telle que l'accès aux données ne se fait que d'une extrémité, un élément à la fois, appelé le **dessus** de pile (*top*).

On appelle souvent ces structures des LIFOs : *last-in first-out*.



Applications

Les piles sont les structures de données derrière les applications suivantes :

- le développement de **compilateurs**, et l'analyse de langages formels en général ;
- l'implémentation d'**algorithmes de retour-arrière** (*backtracking*) ; utilisés par les systèmes de preuve automatique de théorèmes (**Prolog**), les algorithmes de jeux et l'intelligence artificielle ;
- la gestion de la mémoire lors de l'**exécution de programmes** ;
- supporter les opérations « **undo** » dans les programmes de traitement de textes ou encore le bouton « **back** » de votre navigateur Internet (**fureteur**).

Opérations de base

Les opérations de base sont,

push : ajouter un élément sur la pile ;

pop : enlever et retourner l'élément du dessus ;

empty : vérifier que la pile est vide.

Pile TAD

```
public interface Stack {
    public abstract boolean isEmpty();
    public abstract void push( Object o );
    public abstract Object pop();
    public abstract Object peek();
}
```

La pile à l'aide de Java 1.5

```
public interface Stack<E> {
    public abstract boolean isEmpty();
    public abstract E push( E elem );
    public abstract E pop();
    public abstract E peek();
}
```

Oui, oui, c'est ça, l'interface peut elle aussi être paramétrée !

Exemple

```
class Mystery {

    public static void main( String[] args ) {

        Stack<String> stack
            = new StackImplementation<String>();

        for ( int i=0; i<args.length(); i++ )
            stack.push( args[ i ] );

        while ( ! stack.empty() )
            System.out.print( stack.pop() );

    }
}
```

⇒ Que produit ≪`java Mystery a b c d e`≫ ? `e d c b a`

Remarques

- Les éléments retirés apparaissent dans l'ordre inverse ;
- La construction suivante est fréquente dans l'utilisation des piles :
  ```
  while ( ! stack.empty() ) {
      element = stack.pop();
      ...
  }
  ```
- Il faut faire attention de ne pas boucler à l'infini, par exemple en omettant le pop().

Opérations (suite)

peek : retourne la valeur de l'élément du dessus sans le retirer (on l'appelle aussi top) ;

Implémentations

Pensez à une implémentation possible ?

Il y a deux grandes familles d'implémentations :
- à base de tableaux ;
- à base d'éléments chaînés.

```
Stack<Token> s;

s = new ArrayStack<Token>();
s = new DynamicArrayStack<Token>();
s = new LinkedStack<Token>();
```

Question

L'une des implémentations proposées utilise un tableau, pourquoi n'utilise-t-on pas tout simplement un tableau plutôt qu'une pile pour la conception d'algorithmes ?

Implémenter une pile à l'aide d'un tableau : ArrayStack

Quelles sont les variables d'instance ?

Une référence vers un tableau.

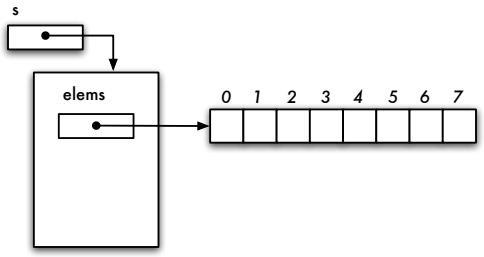
Quel est le type des éléments de ce tableau ?

Object !

Ou encore, un type paramétrique si l'on utilise Java 1.5 (Generics).

Quelle sera la stratégie adoptée afin de sauvegarder les éléments de la pile dans ce tableau ?

Implémenter une pile à l'aide d'un tableau : ArrayStack



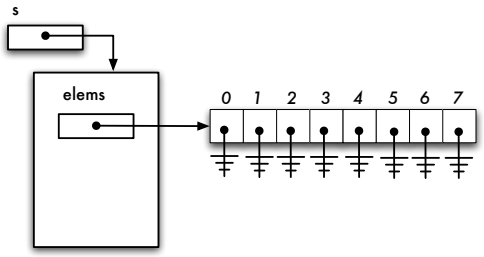
Implémenter une pile à l'aide d'un tableau : ArrayStack

Les éléments sont insérés dans la partie basse ou la partie haute du tableau.

Choisissons la partie basse du tableau, l'implémentation en partie haute sera symétrique.

Comment la méthode **push** détermine-t-elle la position du tableau à laquelle il faut ajouter le prochain élément ?

Implémenter une pile à l'aide d'un tableau : ArrayStack



Implémenter une pile à l'aide d'un tableau : ArrayStack

Comment la méthode **push** détermine-t-elle la position du tableau à laquelle il faut ajouter le prochain élément ?

Il nous faut une variable d'instance supplémentaire.

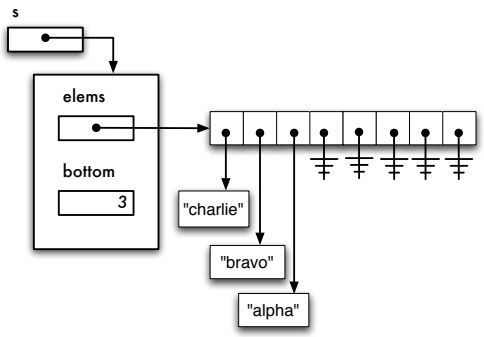
Dessus (top) ou dessous (bottom) ?

Fixer le dessous ou le dessus de la pile ?

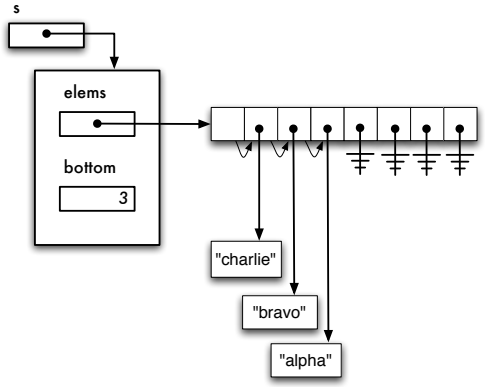
Dans quelle direction allons faire accroître la taille de cette pile ?

À quel index seront ajoutés le premier élément, le second élément, etc.

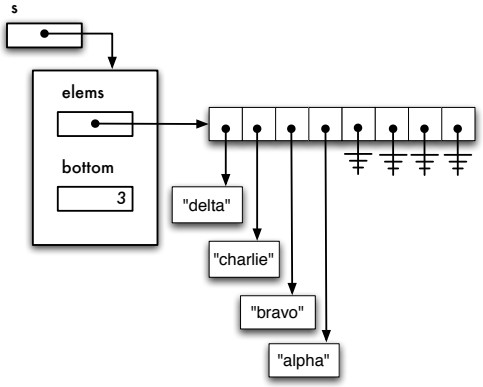
Implémenter une pile à l'aide d'un tableau : ArrayStack



Implémenter une pile à l'aide d'un tableau : ArrayStack



Implémenter une pile à l'aide d'un tableau : ArrayStack



Implémenter une pile à l'aide d'un tableau : ArrayStack

Que pensez-vous de l'implémentation suivante :

La pile est située dans la partie basse du tableau, l'élément du dessus est toujours situé à la position 0 (convention), une variable d'instance nous indique la position de l'élément du dessous.

Pour chaque ajout, il faudra déplacer les éléments d'une position vers les indices hauts du tableau, pour chaque retrait, les éléments seront déplacés d'une position vers les indices bas.

Implémenter une pile à l'aide d'un tableau : ArrayStack

Que pensez-vous de l'implémentation suivante :

La pile est située dans la partie basse du tableau, l'élément du dessous est toujours situé à la position 0 (convention), une variable d'instance nous indique la position de l'élément du dessus.

Pour chaque ajout, il suffit d'incrémenter la valeur de l'index dessus et d'ajouter l'élément à cette position. Pour chaque retrait, il suffit de sauvegarder l'élément du dessus dans une variable temporaire, de remettre à null la case du tableau désignée par la variable dessus, on décrémente cette variable et on retourne l'élément sauvegardé.

Cette solution est préférable puisqu'on évite de recopier les éléments du tableau.

Implémenter une pile à l'aide d'un tableau : ArrayStack

Pour résumer, cette implémentation comporte deux variables d'instance, une référence et l'index de l'élément du dessus.

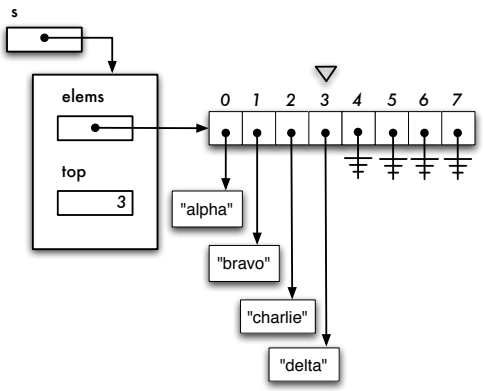
Quel sera le travail du constructeur ?

Quelle est la valeur initiale de l'index dessus (top) ?

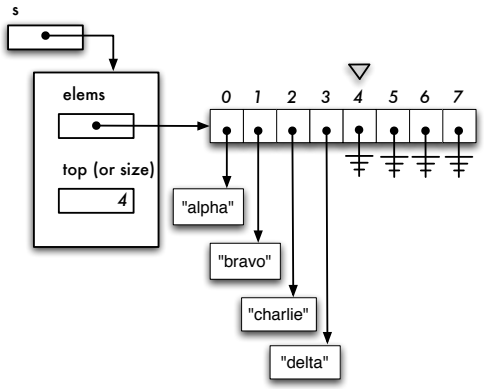
Il existe au moins deux stratégies :

1. Dessus désigne la première cellule libre du tableau ;
2. Dessus désigne la position de l'élément du dessus ;

Implémenter une pile à l'aide d'un tableau : ArrayStack



Implémenter une pile à l'aide d'un tableau : ArrayStack



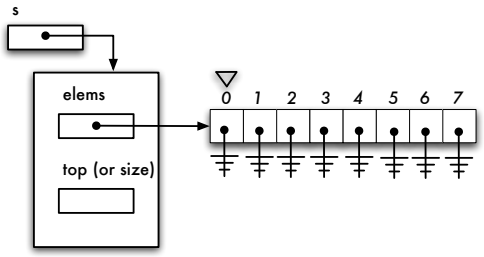
Implémenter une pile à l'aide d'un tableau : ArrayStack

Il existe au moins deux stratégies :

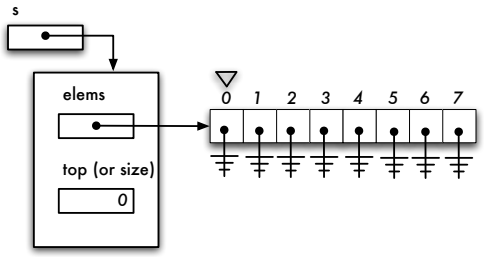
1. Dessus désigne la première cellule libre du tableau. Quelle sera sa valeur initiale ?
2. Dessus désigne la position de l'élément du dessus. Quelle sera sa valeur initiale ?

Assurez-vous de bien maîtriser les deux stratégies.

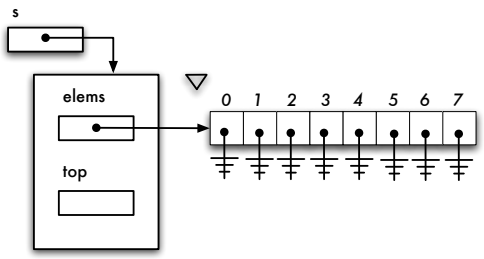
Implémenter une pile à l'aide d'un tableau : ArrayStack



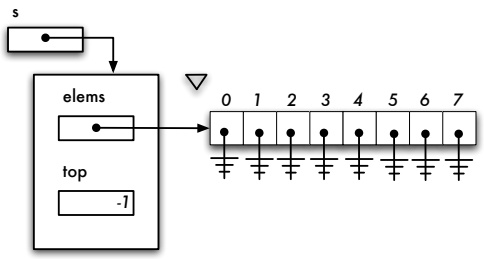
Implémenter une pile à l'aide d'un tableau : ArrayStack



Implémenter une pile à l'aide d'un tableau : ArrayStack



Implémenter une pile à l'aide d'un tableau : ArrayStack

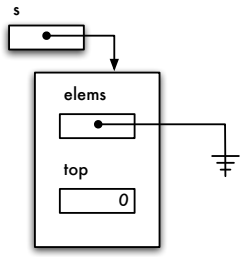


Implémenter une pile à l'aide d'un tableau : ArrayStack

Implémenter une pile à l'aide d'un tableau : ArrayStack

```
public class ArrayStack implements Stack {

    // Instance variables
    private Object[] elems;  // used to store the elements
    private int top;         // designates the first free cell!


    // Constructor

    public ArrayStack( int capacity ) {

    }
```

Implémenter une pile à l'aide d'un tableau : ArrayStack

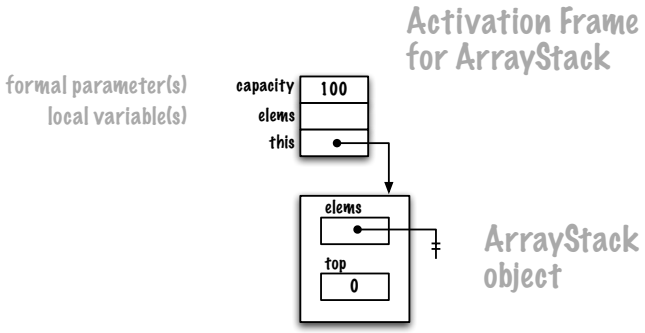


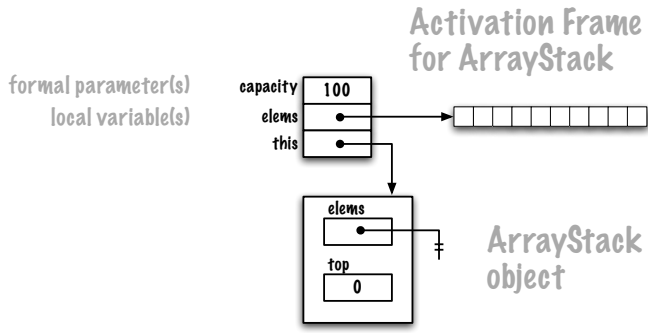
Implémenter une pile à l'aide d'un tableau : ArrayStack

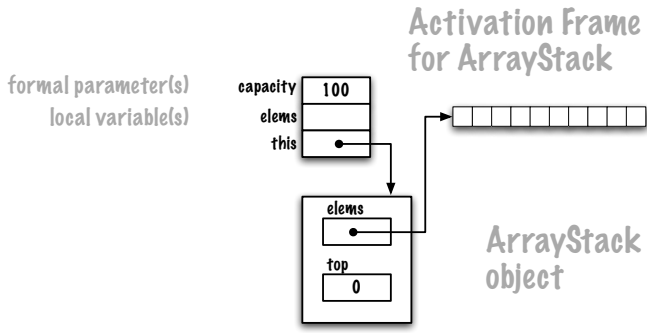
```
public class ArrayStack implements Stack {

    // Instance variables
    private Object[] elems;  // used to store the elements of this ArrayStack
    private int top;         // designates the first free cell!

    // Constructor

    public ArrayStack( int capacity ) {


        elems = new Object[ capacity ];
        top = 0;
    }


    // Returns true if this ArrayStack is empty
    public boolean isEmpty() {
        return top == 0;
    }
```

Piège ? !

```
public class ArrayStack implements Stack {

    // Instance variables
    private Object[] elems;
    private int top;

    // Constructor

    public ArrayStack( int capacity ) {
        Object[] elems = new Object[ capacity ];
        top = 0;
    }

    // Returns true if this ArrayStack is empty
    public boolean isEmpty() {
        return top == 0;
    }
```







Implémenter une pile à l'aide d'un tableau : ArrayStack

```
    // Returns the top element of this ArrayStack without removing it
    public Object peek() {
        // pre-conditions: ! isEmpty()
        return elems[ top-1 ];
    }
}
```

ArrayStack (sous Java 1.5)

Vous vous souvenez du problème lié à la classe **Pair** pour la version pré-Java 1.5 ? Et bien oui, le problème refait surface.

C'est ça, l'utilisation du type **Object** pour les variables d'instances, paramètres et valeurs de retour nous a permis de créer une seule implémentation que l'on utilise dans plusieurs contextes (pour sauvegarder des objets des classes **String**, **Time**, **Event**, etc.).

Cependant, puisque le type de la valeur de retour est **Object**, l'appelant doit bien souvent forcer l'affectation de la valeur de retour (type cast).

Pair

```
public class Pair {
  private ______ first;
  private ______ second;
  public Pair( ______ first, ______ second ) {
    this.first = first;
    this.second = second;
  }
  public ______ getFirst( ) {
      return first;
  }
  public  ______ getSecond( ) {
      return second;
  }
}
```

Pair

```
public class Pair<T> {
    private T first;
    private T second;
    public Pair( T first, T second ) {
        this.first = first;
        this.second = second;
    }
    public T getFirst( ) {
        return first;
    }
    public T getSecond( ) {
        return second;
    }
}
```

Pair

```
Pair<String> name;
name = new Pair<String>( "Hilary", "Clinton" );

Pair<Time> times;
name = new Pair<Time>( new Time( 10,0,0 ), new Time( 11,30,0 ) );

String s;
s = name.getFirst();

Time t;
t = times.getFirst();
```

ArrayStack et Generics

```
Stack<String> s1;
name = new ArrayStack<String>( 100 );

Stack<Time> s2;
name = new ArrayStack<Time>( 1024 );

s1.push( "alpha" );
s2.push( new Time( 23,0,0 ) );

String a;
a = s1.pop();
```

ArrayStack et Generics

Quels sont les changements nécessaires ?

```
public class ArrayStack implements Stack { ... }
```

L'en-tête devient :

```
public class ArrayStack<E> implements Stack<E> { ... }
```

ArrayStack et Generics

Quels sont les changements au niveau des variables d'instance ?

```
    // Instance variables
    private Object[] elems;
    private int top;
```

Voici,

```
public class ArrayStack<E> implements Stack<E> {

    private E[] elems;
    private int top;

    // ...
```

ArrayStack et Generics

Quels sont les changements au niveau du constructeur ?

```
    public ArrayStack( int capacity ) {
        elems = new Object[ capacity ];
        top = 0;
    }
```

Un tableau ≪générique≫ devrait faire l'affaire !

```
    public ArrayStack( int capacity ) {
        elems = new E[ capacity ];
        top = 0;
    }
```

ArrayStack et Generics

```
public class ArrayStack<E> implements Stack<E> {
    private E[] elems;
    private int top;

    public ArrayStack( int capacity ) {
        elems = new E[ capacity ];
        top = 0;
    }
```

Cependant, une erreur de compilation se produira,

```
ArrayStack.java:11: generic array creation
        elems = new E[ capacity ];
                ^
1 error
```

ArrayStack et Generics

Pour des raisons de compatibilité avec les versions précédentes de Java, la création d'un tableau dont les éléments ont un type générique est impossible, malheureusement !

ArrayStack et Generics

Nous avons quand même un problème à résoudre !

```
public class ArrayStack<E> implements Stack<E> {
    private E[] elems;
    private int top;

    public ArrayStack( int capacity ) {
        elems = (E[]) new Object[ capacity ];
        top = 0;
    }
    // ...
```

le compilateur générera un message comme suit :

```
Note: ArrayStack.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

ArrayStack et Generics

Nous pouvons localement éliminer ces messages.

```
public class ArrayStack<E> implements Stack<E> {
    private E[] elems;
    private int top;

    @SuppressWarnings( "unchecked" )
    public ArrayStack( int capacity ) {
        elems = (E[]) new Object[ capacity ];
        top = 0;
    }
    // ...
```

ce qui préférable à l'élimination globale.

```
> javac -Xlint:unchecked ArrayStack.java
```

ArrayStack

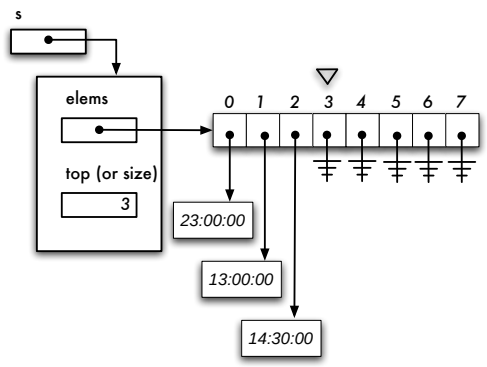
```
public void push( E element ) {
    // pre-condition: the stack is not full

    // stores the element at position top, then increments top
    elems[ top++ ] = element;
}
```

ArrayStack

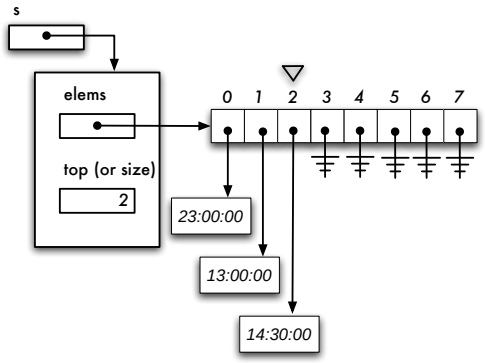
```
    // Removes and returns the top element of this stack
    public E pop() {
        // pre-conditions: ! isEmpty()

        // decrements top, then access the value
        E saved = elems[ --top ];

        return saved;
    }
}
```

Tout compile et il y n'y a pas d'erreur d'exécution, mais !

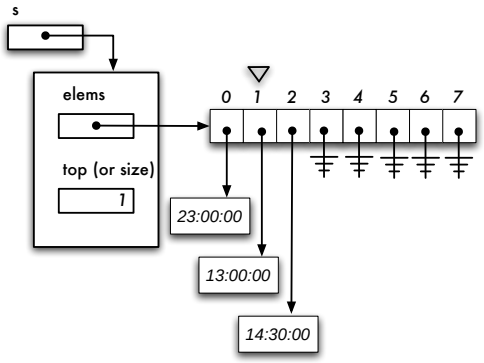
ArrayStack



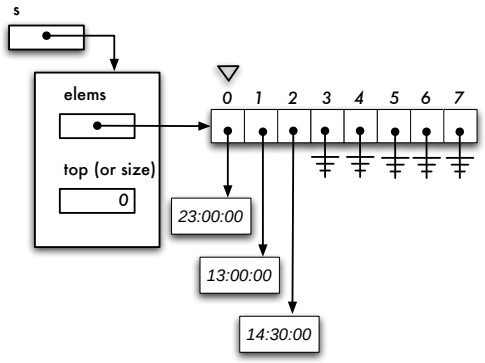
ArrayStack



ArrayStack



ArrayStack



ArrayStack

Bien que Java s'occupe d'une bonne partie des tâches liées à la gestion de la mémoire, les fuites de mémoire sont possibles !

```
    public E pop() {
        // pre-conditions: ! isEmpty()

        // decrements top, then access the value
        E saved = elems[ --top ];

        elems[ top ] = null; // ''scrubing'' the memory!

        return saved;
    }
}
```

Implémenter une pile à l'aide d'un tableau

L'implémentation proposée a un handicap majeur, la taille de la pile est fixe et elle est déterminée au moment de la création de la pile.

Souvent, on ne connaît pas le nombre d'éléments à traiter, que faire ?
- Allouer un tableau de très grande taille. Quels sont les désavantages d'une telle approche ? Gaspillage des ressources, les débordements sont aussi possibles ;
- Solution : tableau dynamique.

Implémenter une pile à l'aide d'un tableau dynamique

Certains langages de programmation permettent la modification de la taille des tableaux lors de l'exécution des programmes. Ces langages utilisent tout simplement la technique présentée ci-bas.

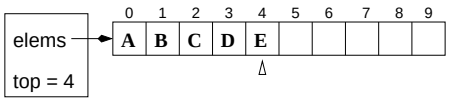
- Lorsque le tableau est rempli à capacité, un nouveau tableau de taille plus grande est alloué, les éléments sont copiés de l'ancien tableau vers le nouveau, et finalement le nouveau tableau remplacera l'ancien ;
- Il y a plusieurs stratégies afin d'accroître la taille du tableau dont : $n = n + c$, ou $c = 1$ par exemple, ou encore, $n = c * n$, ou $n = 2$ par exemple ;
- Discutez des avantages et désavantages de chacune des solutions proposées.

Annexe

Informations complémentaires tirées de mes anciennes notes de cours.

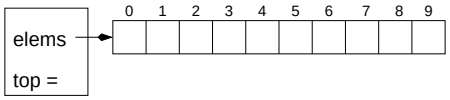
Implémenter une pile à l'aide d'un tableau : ArrayStack

1.1 Implémentation en mémoire basse du tableau, la variable top désigne l'élément du dessus.



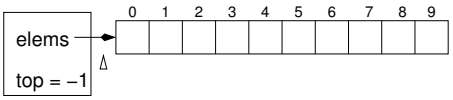
1.1 Création

1.1 Création



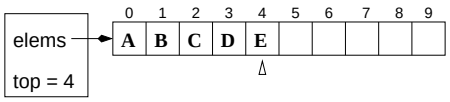
Allocation d'un tableau de taille `MAX_STACK_SIZE`, ici 10.

1.1 Création

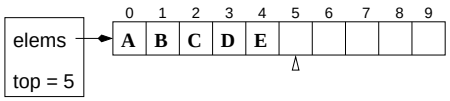


Initialiser la variable top à -1.

1.1 push(F)

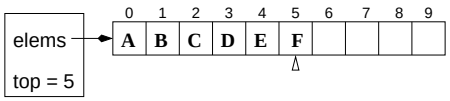


1.1 push(F)



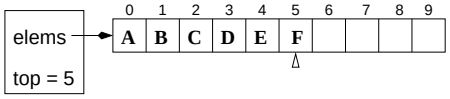
incrémente la variable dessus de pile.

1.1 push(F)



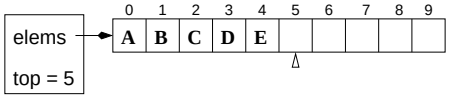
insère l'élément dans le tableau à la position top.

1.1 pop()



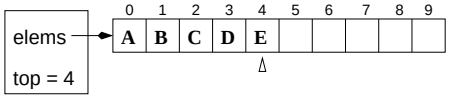
sauve le contenu du dessus de la pile dans une variable temporaire (disons valeurDeRetour).

1.1 pop()



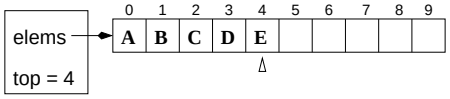
stack[top] est réinitialisé (0 pour les entiers, '\u0000' pour les caractères et *null* pour les références).

1.1 pop()



décrémente la variable dessus de pile.

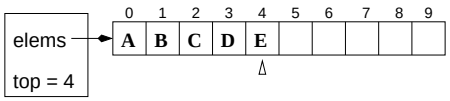
1.1 pop()



retourne valeurDeRetour

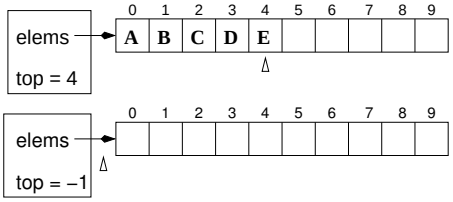
```
            top =   4, stack->[A,B,C,D,E, , , , , ]
                                      ^
E <- pop()  top =   3, stack->[A,B,C,D, , , , , , ]
                                    ^
D <- pop()  top =   2, stack->[A,B,C, , , , , , , ]
                                  ^
push(G)     top =   3, stack->[A,B,C,G, , , , , , ]
                                    ^
push(H)     top =   4, stack->[A,B,C,G,H, , , , , ]
                                      ^
push(I)     top =   5, stack->[A,B,C,G,H,I, , , , ]
                                        ^
push(J)     top =   6, stack->[A,B,C,G,H,I,J, , , ]
                                          ^
```

1.1 peek()



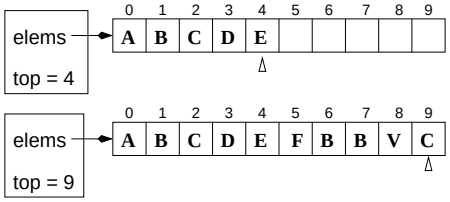
retourne stack[top]

1.1 empty()



est-ce que top vaut -1?

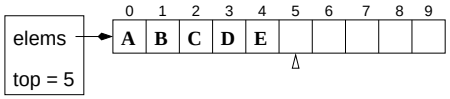
1.1 isFull()



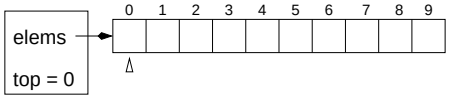
est-ce que top vaut (MAX_STACK_SIZE - 1)?

Implémenter une pile à l'aide d'un tableau

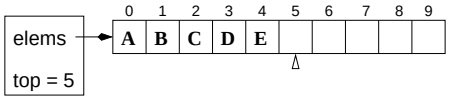
1.2 Implémentation en mémoire basse du tableau, la variable top pointe sur la première cellule libre.



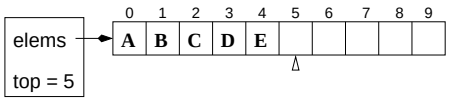
En quoi cela change-t-il la création d'une nouvelle pile ?



La variable top est initialisée à 0, plutôt que -1.



En quoi cela change-t-il push() ? Les opérations sont inversées : on insère d'abord la valeur dans le tableau à la position top, puis on incrémente top.



En quoi cela change-t-il pop() ?
On décrémente top d'abord,
On sauve la valeur du dessus dans une variable temporaire, valeurDeRetour,
Initialise stack[top],
retourne valeurDeRetour

En quoi cela change-t-il peek() ?

```
retourne stack[top-1]
```

empty() ?

```
est-ce que top vaut 0?
```

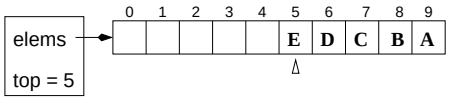
isFull() ?

```
est-ce que top vaut MAX_STACK_SIZE
```

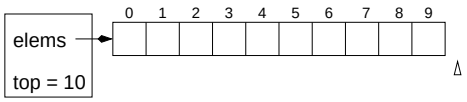
Est-ce que l'une des 2 implémentations est préférable ?
Non pas vraiment, mais la première est la plus fréquente.

Implémenter une pile à l'aide d'un tableau

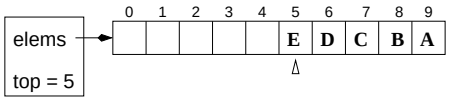
2.1 Implémentation en mémoire haute du tableau, la variable top pointe sur le dessus de pile.



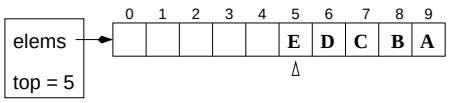
En quoi cela change-t-il la création d'une nouvelle pile ?



La variable top est initialisée à `MAX_STACK_SIZE`, plutôt que -1.



En quoi cela change-t-il push() ?
décrémente top,
insère la valeur dans le tableau à la position top.



En quoi cela change-t-il pop() ? Il faut incrémenter top, plutôt que le décrémenter.

Sauver la valeur du dessus, valeurDeRetour,
Initialiser tableau[top],
Incrémenter top,
Retourner valeurDeRetour

En quoi cela change-t-il peek() par rapport à 1.1 ? Rien à changer.

```
retourne stack[top]
```

empty() ?

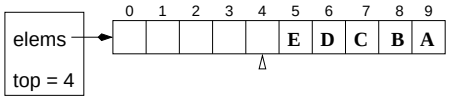
```
Est-ce que top vaut MAX_STACK_SIZE?
```

isFull() ?

```
Est-ce que top vaut 0
```

Vous l'aurez deviné, l'implémentation suivante est aussi possible.

2.2 Implémentation en mémoire haute du tableau, la variable top pointe sur la première cellule libre.



[Les détails de l'implémentation 2.2 sont laissés à faire en exercice.]

Est-ce que l'une des 4 implémentations est préférable ?
Non pas vraiment. Ça dépend du contexte.

Imaginez que . . .

Qu'est-ce qui arriverait si l'on décidait que la position du dessus est fixe, par exemple top = 0, et que l'on utilisait une variable afin de noter la position du **dessous** de la pile (bottom).

Regardons quelques exemples :

```
            bottom = -1   stack -> [ , , , , , , , , , ]
                                    ^
push(A)     bottom =  0   stack -> [A, , , , , , , , , ]
                                     ^
push(B)     bottom =  1   stack -> [B,A, , , , , , , , ]
                                       ^
push(C)     bottom =  2   stack -> [C,B,A, , , , , , , ]
                                         ^
C <- pop()  bottom =  1   stack -> [B,A, , , , , , , , ]
                                       ^
```

⇒ Premier constat : peek() retourne toujours stack[0].

```
            bottom =  1   stack -> [B,A, , , , , , , , ]
                                     ^
```

push(C) ?

```
Incremente bottom
Pour i=bottom jusqu'a 1 (boucle décroissante)
  stack[i] = stack[i-1]
Insere l'element sur le dessus, i.e. stack[0] = C

            bottom =  1   stack -> [B,A, , , , , , , , ]
                                     ^
            bottom =  2   stack -> [B,A,A, , , , , , , ]
                                       ^
            bottom =  2   stack -> [B,B,A, , , , , , , ]
                                       ^
            bottom =  2   stack -> [C,B,A, , , , , , , ]
                                       ^
```

```
            bottom =  1   stack -> [C,B,A, , , , , , , ]
                                       ^
```

pop() ?

```
valeurDeRetour = stack[0]
Pour i=0 jusqu'a (bottom - 1) (boucle croissante)
  stack[i] = stack[i+1]
Initialiser stack[bottom]
Decremente bottom
Retourne valeurDeRetour

valeurDeRetour = C
            bottom =  2   stack -> [B,B,A, , , , , , , ]
                                       ^
            bottom =  2   stack -> [B,A,A, , , , , , , ]
                                       ^
            bottom =  2   stack -> [B,A, , , , , , , , ]
                                       ^
            bottom =  1   stack -> [B,A, , , , , , , , ]
                                       ^
retourne C
```

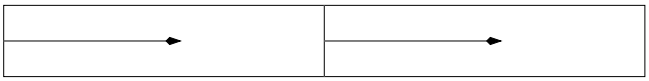
Remarques

En quoi cela influence-t-il l'efficacité des opérations ?
- push() : tous les éléments sont d'abord déplacés d'une position vers la droite avant l'insertion.
- pop() : tous les éléments sont déplacés vers la gauche suite au retrait du dessus de la pile.
- **plus il y a d'éléments dans la pile, plus ça prend de temps à insérer ou retirer un élément**. Pensez au cas ou la pile contient 1 élément, 2 éléments ou 1,000,000 d'éléments.

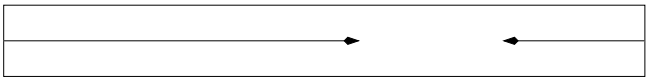
⇒ Quelle implémentation est favorable, celle-ci ou 1.1 ?

Implémenter 2 piles à l'aide d'un seul tableau

On ne parle pas de 2 piles qui s'accroissent dans la même direction.

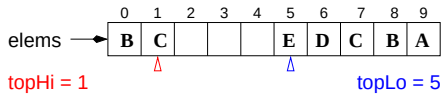


Mais de 2 piles qui s'accroissent l'une vers l'autre.



Comment ?

Un seul tableau, mais deux variables top distinctes.



A quoi ça sert ?

Gestion de la mémoire lors de l'exécution de programmes (nous reviendrons sur ceci si le temps le permet).

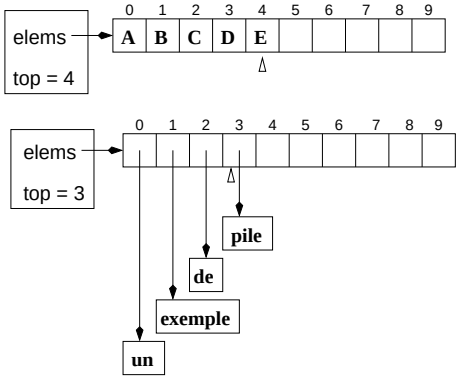
C'est plus efficace en terme d'utilisation de la mémoire parce que l'une des 2 piles peut utiliser plus de la moitié du tableau si nécessaire, et que l'autre pile n'occupe pas déjà cet espace, alors que c'est impossible si on utilise 2 tableaux.

Types primitifs vs type référence

Jusqu'à maintenant nous avons vu des piles dont les valeurs des éléments étaient de types primitifs, des entiers ou des caractères.

Puisqu'on peut stocker des valeurs de type référence dans un tableau, on peut donc aussi stocker des valeurs de type référence dans nos piles. Par exemple, une pile de chaînes de caractères.

⇒ par rapport à l'implémentation 1.1.

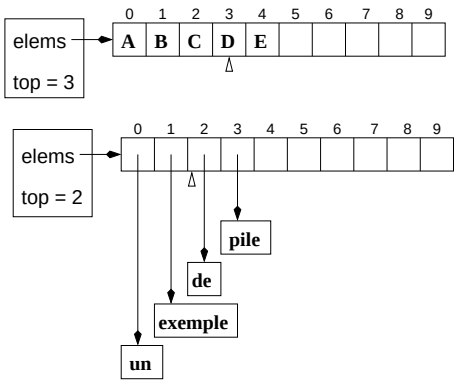


On a dit que pop() consiste à

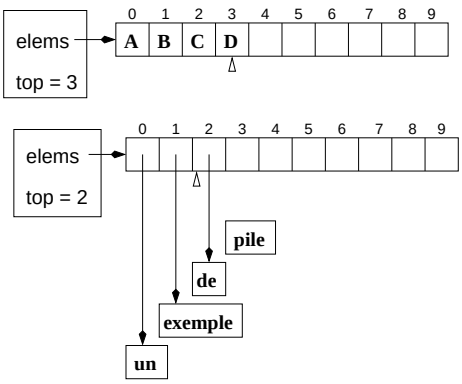
1. sauver la valeur de l'élément du dessus,
2. remettre à zéro stack[top],
3. décrémenter top,
4. retourner la valeur.

Est-ce que la remise à zéro est vraiment nécessaire ?

Est-ce que c'est différent selon que les valeurs sont d'un type primitif ou de type référence ?



⇒ Puisqu'une référence vers l'objet ≪pile≫ existe, le gestionnaire de mémoire (gc() — *garbage collector*) ne peut récupérer la mémoire.



⇒ Aucune référence vers l'objet ≪pile≫, gc() récupère la mémoire. Il est donc important, dans le cas d'une pile dont les valeurs sont de type référence, de

remettre à zéro (*null*) la cellule de l'élément retiré.

Note : conditions d'erreur

Les implémentations proposées ne vérifient pas les conditions d'erreur, c'est-à-dire l'ajout d'un élément dans une pile pleine ou le retrait d'un élément d'une pile vide.

Il faut donc que l'utilisateur fasse ces tests :

```
if (! s.empty())
  v = s.pop();

...

if (! s.isFull())
  s.push(v);
```

Propriétés des tableaux

L'accès aux éléments d'un tableau se fait par position, par exemple, **a[3]**, désigne le quatrième élément du tableau.

L'accès à un élément est très rapide.

La clé de ce succès est que les éléments sont contigus en mémoire. Ainsi, l'accès a un élément requière toujours le même nombre d'étapes, on dit aussi que cette opération prend un temps constant, c.-à-d. que le temps nécessaire est indépendant de :
– la taille du tableau ;
– le nombre d'éléments se trouvant dans le tableau ;
– la position de l'élément recherché (premier, dernier, intermédiaire).

L'adresse mémoire du début du tableau est l'adresse de base, le premier élément du tableau se trouve à l'adresse de base, pour déterminer l'adresse du second il faut connaître la taille du premier élément et ajouter ce nombre à l'adresse de base. Puisque les éléments d'un tableau sont tous de même type, le calcul de l'adresse de tout élément est le même.

$$\text{adresse de base} + \text{deplacement}$$

ou le déplacement se calcule comme suit :

$$\text{position} * \text{taille d'un element}$$

Aucune recherche nécessaire.

Taille fixe

Qu'arrive-t-il si la taille du tableau nécessaire n'est pas connue à l'avance ?

Par exemple, imaginez qu'on vous demande de lire au clavier plusieurs nombres entiers positifs, devant être sauvegardés dans un tableau, et que la lecture s'arrête lorsque la valeur spéciale (sentinelle) -9 est lue.

Quelle devrait être la taille initiale du tableau ? [1]

Solution 1 : créer un très grand tableau

Créer un tableau d'une taille suffisante pour toute application possible.

Quelles sont les conséquences ?

Gaspillage de mémoire.

Il se peut que le tableau soit plein et que l'application doive s'arrêter (en catastrophe).

1. Certains langages de programmation, tels que Fortran et Pascal, vous demandent de connaître la taille du tableau au moment de la compilation du programme.

Solution 2 : tableaux de taille variable

Créer un tableau de taille raisonnable.

Accroître ou décroître sa taille au besoin.

Ce qui veut dire que la taille physique du tableau ne correspondra pas nécessairement à sa taille logique.

Ce qui veut dire que le programmeur doit maintenir l'information concernant la taille logique lui même (la variable length du tableau est sa taille physique).

C'est aussi la responsabilité du programmeur de s'assurer qu'il n'accède pas aux positions non utilisées.

Une solution plus pratique serait de doubler la taille du tableau à chaque fois que la taille logique atteint la valeur de la taille physique.

Qu'a-t-on gagné ?

Seules les insertions telles que la taille logique == taille physique nécessite un agrandissement du tableau et des copies.

Qu'a-t-on perdu ?

Un plus grand gaspillage de mémoire.

⇒ pour certaines applications, la taille logique du tableau peut aussi diminuée, il faudra donc penser à réduire la taille du tableau lorsque le nombre d'éléments sera au deçà d'une certaine limite.

1. est-que le tableau a une taille suffisante ?
2. de quelle position doit-on commencer les copies ?

⇒ écrire la méthode remove(int pos)

Décrivez le comportement de la méthode increaseSize() à mesure que la taille du tableau croît.

Tous les éléments du tableau sont recopiés pour chaque insertion.

Plus il y d'éléments, plus il y a de copies.

Initialement, il n'y a que quelques éléments copiés, mais plus le tableau devient grand, plus il y aura de copies.

Lorsque la taille logique du tableau a rejoint la taille physique, toute insertion subséquente nécessite l'agrandissement du tableau (dont le coût est proportionnel au nombre d'éléments).