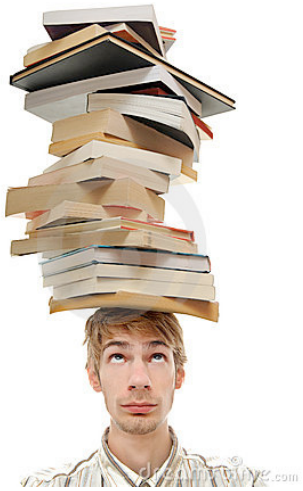
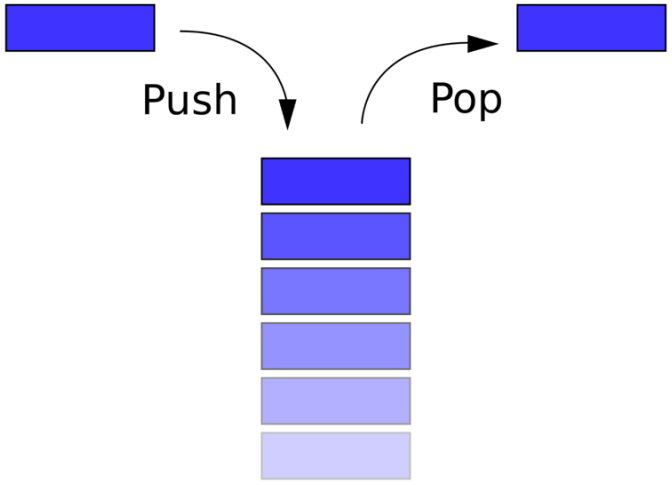
Piles
ADT Stack
Applications
ListStack
ArrayStack
Programmes d'applications :
Balancement de parenthèses
Évaluation d'expression arithmétiques

La pile (Stack) est caractérisée par deux opérations : `push` (empiler) et `pop` (dépiler)

La pile est caractérisée par sa politique de dernier entré premier sorti (last-in-first-out; LIFO).





Opérations auxiliaires d'une pile

objet top( ) : retourne le dernier élément inséré sans le supprimer
entier len( ) : retourne le nombre d'éléments stockés
booléen is_empty( ) : indique si la pile est vide

Operation	Return Value	Stack Contents
S.push(5)	–	[5]
S.push(3)	–	[5, 3]
len(S)	2	[5, 3]
S.pop( )	3	[5]
S.is_empty( )	False	[5]
S.pop( )	5	[ ]
S.is_empty( )	True	[ ]
S.pop( )	"error"	[ ]
S.push(7)	–	[7]
S.push(9)	–	[7, 9]
S.top( )	9	[7, 9]
S.push(4)	–	[7, 9, 4]
len(S)	3	[7, 9, 4]
S.pop( )	4	[7, 9]
S.push(6)	–	[7, 9, 6]
S.push(8)	–	[7, 9, 6, 8]
S.pop( )	8	[7, 9, 6]

```python
#!/usr/local/bin/python3

# author = "Francois Major"
# version = "1.0"
# date = "23 mars 2014"
#
# Programme Python pour IFT2015/Types abstraits/ADT Stack
#
# Pris et modifié de Goodrich, Tamassia & Goldwasser
#   Data Structures & Algorithms in Python (c)2013

# ADT Stack "interface"
class Stack:

    # constructeur
    def __init__( self ):
        pass

    # nombre d'éléments
    def __len__( self ):
        pass

    # produit une chaîne de caractères:
    # les éléments entre crochets
    # séparés par des virgules
    # élément top indiqué
    # taille et capacité de la structure de données
    # indiquées lorsque pertinent
    def __str__( self ):
        pass

    # indique si la pile est vide : aucun élément
    def is_empty( self ):
        pass

    # ajoute un élément sur la pile
    def push( self, element ):
        pass

    # retire un élément de la pile
    def pop( self ):
        pass

    # retourne le dernier élément empilé
    # sans le retirer
    def top( self ):
        pass
```

Applications

Directes :

- Historique des pages visitées dans un navigateur Web
- Annuler une séquence dans un éditeur de texte
- Chaîne de méthodes appellées dans un langage qui prend en charge la récursivité

Indirectes :

- Structure de données auxiliaires pour des algorithmes
- Composant d'autres structures de données

```python
#!/usr/local/bin/python3

# author = "Francois Major"
# version = "1.0"
# date = "15 février 2014"
#
# Programme Python pour IFT2015/Types abstraits/ADT Stack
#
# Pris et modifié de Goodrich, Tamassia & Goldwasser
#   Data Structures & Algorithms in Python (c)2013

from Stack import Stack

class ListStack( Stack ):

    # implémente l'ADT Stack (Stack.py)
    # avec une liste Python
    def __init__( self ):
        self._A = []

    def __len__( self ):
        return len( self._A )

    def is_empty( self ):
        return len( self._A ) == 0

    def __str__( self ):
        pp = str( self._A )
        pp += "(size = " + str( len( self._A ) )
        pp += ")[top = " + str( len( self._A ) - 1 ) + "]"
        return pp

    # push implémenté avec append
    def push( self, obj ):
        self._A.append( obj )

    # pop implémenté avec pop
    # si l'opération échoue, on retourne None
    def pop( self ):
        try:
            return self._A.pop( )
        except IndexError:
            return None

    # top est le dernier élément
    def top( self ):
        if self.is_empty():
            return None
        else:
            return self._A[len( self._A ) - 1]
```

```python
#!/usr/local/bin/python3

# author = "Francois Major"
# version = "1.0"
# date = "23 mars 2014"
#
# Programme Python pour IFT2015/Types abstraits/ADT Stack
#
# Pris et modifié de Goodrich, Tamassia & Goldwasser
#   Data Structures & Algorithms in Python (c)2013

from DynamicArray import DynamicArray
from Stack import Stack

class ArrayStack( Stack ):

    # implémente l'ADT Stack (Stack.py)
    # avec un tableau dynamique (DynamicArray.py)
    def __init__( self ):
        self._A = DynamicArray()

    def __len__( self ):
        return len( self._A )

    def is_empty( self ):
        return len( self._A ) == 0

    def __str__( self ):
        pp = str( self._A )
        if self.is_empty():
            pp += "[empty stack]"
        else:
            obj = self.top()
            pp += "[top = " + str( obj ) + ", idx = " + str( len( self._A ) - 1 ) + "]"
        return pp

    # push implémenté avec append
    def push( self, obj ):
        self._A.append( obj )

    # pop récupère l'erreur de DynamicArray
    def pop( self ):
        try:
            return self._A.pop()
        except IndexError:
            return None

    # top implémenté avec get du dernier élément
    def top( self ):
        return self._A.get( len( self._A ) - 1 )
```

Balancement de parenthèses

Chaque "(", "{", ou "[" doit être associé à une ")", "}" ou "[" correspondante :

correct : `(  )(  )){([(  )])}`
correct : `(((  )(  )){([(  )])})))`

Incorrect : `)(  )){([(  )])}`
Incorrect : `({[  ])}`
Incorrect : `(`

```python
#!/usr/local/bin/python3

# author = "Francois Major"
# version = "1.0"
# date = "15 février 2014"
#
# Programme Python pour IFT2015/Types abstraits/ADT Stack
#
# Pris et modifié de Goodrich, Tamassia & Goldwasser
#   Data Structures & Algorithms in Python (c)2013

# Ce programme prend en input une expression
# contenant des (), [] et {} et vérifie si ces
# symboles sont bien balancés

# utilise ListStack
from ListStack import ListStack

# Fonction principale
def main():
    # Lire en input une expression
    expr = input( 'Entrez une expression: ' )
    if parenMatch( expr ):
        print( "L'expression ", expr, "est balancée !" )
    else:
        print( "L'expression ", expr, "n'est pas balancée !" )

# fonction parenMatch vérifie si l'expression
# passée en argument est balancée en (), [] et {}
def parenMatch( expr ):
    # symboles de gauche
    aGauche = "({["
    # symboles de droite
    aDroite = ")}]"

    # utilise pile S
    S = ListStack()

    # pour chaque caractère dans l'expression
    for c in expr:
        # si on rencontre un caractère de gauche
        if c in aGauche:
            # on l'empile
            S.push( c )
        elif c in aDroite:
            # si on a un symbole de droite
            if S.is_empty():
                # si la pile est vide, pas de match
                return False
            if aDroite.index( c ) != aGauche.index( S.pop() ):
                # si le symbole à droite ne match pas le symbole à gauche
                return False
    # ici, si la pile est vide, l'expression est balancée
    # sinon il reste un ou des symbole(s) non balancés dans la pile
    return S.is_empty()

# Appeler la fonction principale
main()
```

Évaluation d'expressions arithmétiques

$$14 - 3 * 2 + 7 = (14 - (3 * 2)) + 7$$

Priorité des opérateurs * et / sont plus prioritaires que + et -

Associativité : les opérateurs du même groupe de précédence sont évalués de gauche à droite
Exemple x - y + z sera évaluée comme (x - y) + z et non pas x - (y + z).

Idée: Poussez chaque opérateur sur la pile, mais commencez par "poper" et effectuer les opérations de priorité supérieure et égale.

Algorithme pour évaluer des expressions arithmétiques

Deux piles :
opStk (pour les opérations)
valStk (pour les valeurs)
Utilisons $ comme caractère spécial de fin d'entrée avec la plus basse précédence

```
Algorithme doOp( ) #effectue une opération de 2 valeurs et empile le résultat
    x = valStk.pop( )
    y = valStk.pop( )
    op = opStk.pop( )
    valStk.push( y op x )

Algorithme repeatOps( refOp ) #effectue les opérations (gauche à droite) de même précédence
    #il faut au moins 2 valeurs pour effectuer une opération
    tantque( valStk.size( ) > 1 and prec( refOp ) <= prec( opStk.top( ) ) faire
        doOp( )

Algorithme EvalExp( ) #function principale
    Entrée : flux de jetons représentant une expression arithmétique
    Sortie : la valeur de l'expression
    #lire les jetons en entrée
    tanque il y a un autre jeton z faire
        si z est un nombre alors valStk.push( z ) #si nombre on l'empile
        sinon #opération, on effectue les opérations précédentes de priorités <=)
            repeatOps( z )
            opStk.push( z ) #on empile l'opération
    repeatOps( $ )
    return valStk.top( )
```

```
#!/usr/local/bin/python3

# author = "Francois Major"
# version = "1.0"
# date = "15 février 2014"
#
# Programme Python pour IFT2015/Types abstraits/ADT Stack
#
# Pris et modifié de Goodrich, Tamassia & Goldwasser
#   Data Structures & Algorithms in Python (c)2013
#
# Ce programme prend en input une expression
# arithmétique composée de chiffres (0 à 9) et l'évalue

# utilise 2 piles différentes
from ArrayStack import ArrayStack
from ListStack  import ListStack

# Fonction principale
def main():
    # Lire en input une expression
    global trace
    trace = input( "Voulez-vous la trace (o/n) ? " )
    if( trace == 'o' ):
        trace = True
    else:
        trace = False

    expr = input( 'Entrez une expression: ' )

    print( "L'expression ", expr, "=", evalExp( expr ) )

# précédences des opérations
def prec( op ):
    if op in '*/':
        return 2
    elif op in "+-":
        return 1
    #this is '$'
    return 0

def evalExp( expr ):
    # on utilise une pile pour les valeurs et une pile pour les opérations
    valStk = ArrayStack()
    opStk = ListStack()

    # tant qu'il y a des jetons en entrée
    for z in expr:
        if z.isdigit(): # si chiffre, on l'empile
            valStk.push( z )
            if trace:
                print( "chiffre dans la pile", valStk )
        elif z in "+-*/": # si opération, on voit si on peut l'effectuer
            if trace:
                print( "opération lue : ", z )
            repeatOps( z, valStk, opStk )
            # on empile l'opération
            opStk.push( z )
            if trace:
                print( "opération dans la pile", opStk )
    # on exécute l'opération sur la pile, le cas échéant
    repeatOps( '$', valStk, opStk )
    # le resultat se trouve sur le top de la pile des valeurs
    return valStk.top()
```

```
# effectue une opération de 2 valeurs et empile le résultat
def doOp( valStk, opStk ):
    # on effectue l'opération sur le top de la pile des opérations
    op = opStk.pop()

    # appliquée aux 2 valeurs sur la pile des valeurs
    x = valStk.pop()
    y = valStk.pop()

    if trace:
        print( "doOp( ", x, " ", op, " ", y, " )" )
    if op == '+':
        z = int(y) + int(x)
    elif op == '-':
        z = int(y) - int(x)
    elif op == '*':
        z = int(y) * int(x)
    elif op == "/":
        if( int( x ) is not 0 ):
            z = int(y) / int(x)
        else:
            print( "Division by 0, no result! " )
            exit()

    # on empile le résultat sur la pile des valeurs
    valStk.push( z )

    if trace:
        print( "empile le résultat", valStk )

# effectue les opérations (gauche à droite) de même précédence
def repeatOps( refOp, valStk, opStk ):
    #il faut au moins 2 valeurs pour effectuer une opération
    if trace:
        print( "repeatOps..." )
    while len( valStk ) > 1 and prec( refOp ) <= prec( opStk.top() ):
        doOp( valStk, opStk )


# appel de la fonction principale
main()
```

3*4*2+9/3

```
Voulez-vous la trace (o/n) ? o
Entrez une expression: 3*4*2+9/3
chiffre dans la pile [3](size = 1; capacity = 1)[top = 3, idx = 0]
opération lue :  *
repeatOps...
opération dans la pile ['*'](size = 1)[top = 0]
chiffre dans la pile [3, 4](size = 2; capacity = 2)[top = 4, idx = 1]
opération lue :  *
repeatOps...
doOp(  4   *   3  )
empile le résultat [12](size = 1; capacity = 1)[top = 12, idx = 0]
opération dans la pile ['*'](size = 1)[top = 0]
chiffre dans la pile [12, 2](size = 2; capacity = 2)[top = 2, idx = 1]
opération lue :  +
repeatOps...
doOp(  2   *   12  )
empile le résultat [24](size = 1; capacity = 1)[top = 24, idx = 0]
opération dans la pile ['+'](size = 1)[top = 0]
chiffre dans la pile [24, 9](size = 2; capacity = 2)[top = 9, idx = 1]
opération lue :  /
repeatOps...
opération dans la pile ['+', '/'](size = 2)[top = 1]
chiffre dans la pile [24, 9, 3](size = 3; capacity = 4)[top = 3, idx = 2]
repeatOps...
doOp(  3   /   9  )
empile le résultat [24, 3.0](size = 2; capacity = 2)[top = 3.0, idx = 1]
doOp(  3.0   +   24  )
empile le résultat [27](size = 1; capacity = 1)[top = 27, idx = 0]
L'expression  3*4*2+9/3 = 27
```