Stack

Chapter 6

Learning Objectives

Introduction

Students will be able to:

- Explain stack concept, representation and operations
- Implement stack with linked list and array
- Adopt stack to classical problems or applications

Chapter Outline

1. Stack Concept
   1) Stack Concept
   2) Stack Representation
   3) Stack Abstract Data Type (ADT)
2. Stack List
   1) Stack Header Node
   2) Stack list Application Programming Interface (API)
   3) Stack List Implementation
   4) Analysis Method's Performance of Stack List
3. Stack Array
   1) Stack Array Top
   2) Stack Array Application Programming Interface (API)
   3) Stack Array Implementation
   4) Analysis Method's Performance of Stack Array
4. Stack Application
   1) Reversing
   2) Parsing
   3) Postponing
   4) Backtracking

6

Stack

Section 1:
Stack Concept

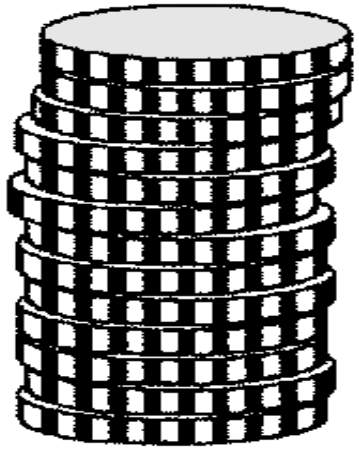
1) Stack Concept
2) Stack Representation
3) Stack Abstract Data Type (ADT)

Stack Concept

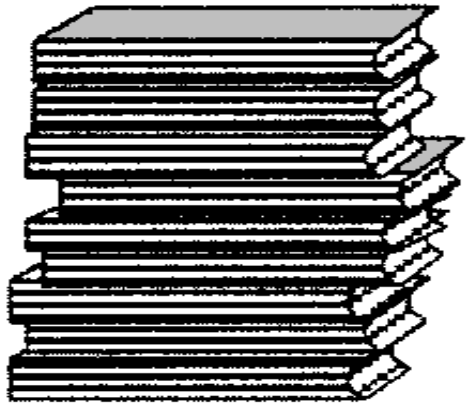
- Stack is a linear data structure whose items are inserted and deleted according to Last in First out (LIFO) concept. [1]
- It is a structure in which a new item is added and removed at the same side- called top of stack. [2]
- In a stack (sometimes called a ”push-down stack”), the most recently added item is the one that is in position to be removed first. [3]

Stack Concept

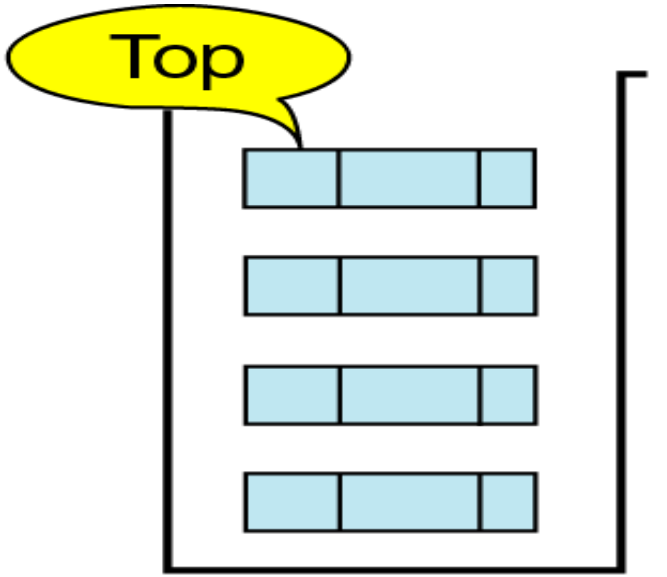
- Newer items are near the top while older items are near the bottom.



Stack of coins



Stack of books



Computer stack

Fig 6-1 Stack views [4]

Stack Representation

- It can be implemented with the primitive linear data structures:
  1. Linked list – stack list
  2. Array – stack array

Fig 6-2 Stack implementation

Stack ADT

- There are three main operations - specially designed for stack:
  1. Push: Insert operation which must be strictly inserted to the top of stack
  2. Pop: Remove the top item from the stack
  3. Top: Return the top item value or both value and its address.

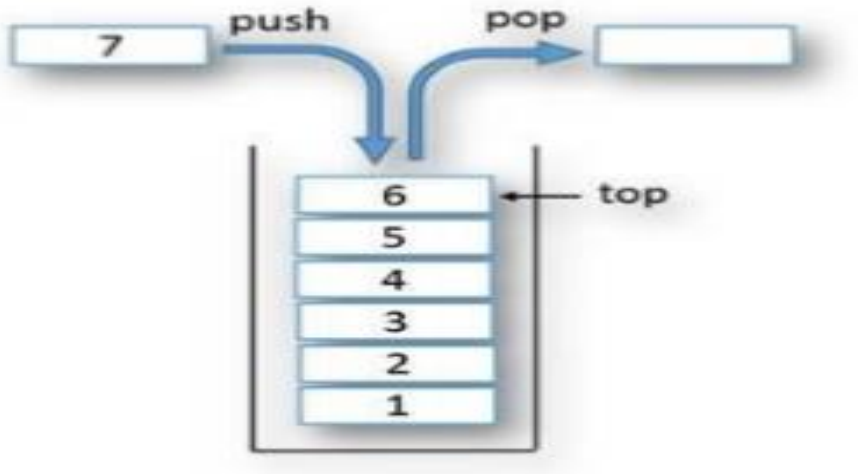


Fig 6-3 Stack push, pop and top [5]

Stack ADT

Push

- A stack insertion which an item is added at the top of stack
- Before adding, we must ensure that there is an available space to contain a new item otherwise it will be “stack overflow”.
- After pushing, the added item becomes the top of stack.

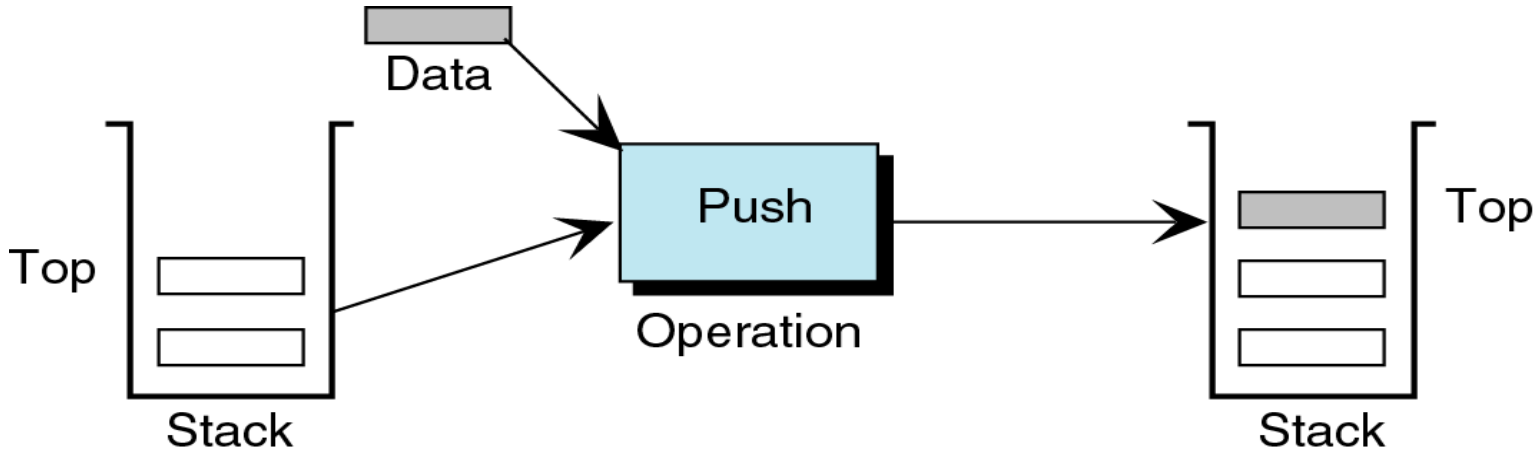


Fig 6-4 Stack push [5]

Stack ADT

Pop

- A stack deletion will remove an item at the top of stack
- Before deleting, there is at least an item to be removed otherwise “stack underflow” will be notified.
- After pushing, the follower item becomes the top of stack.

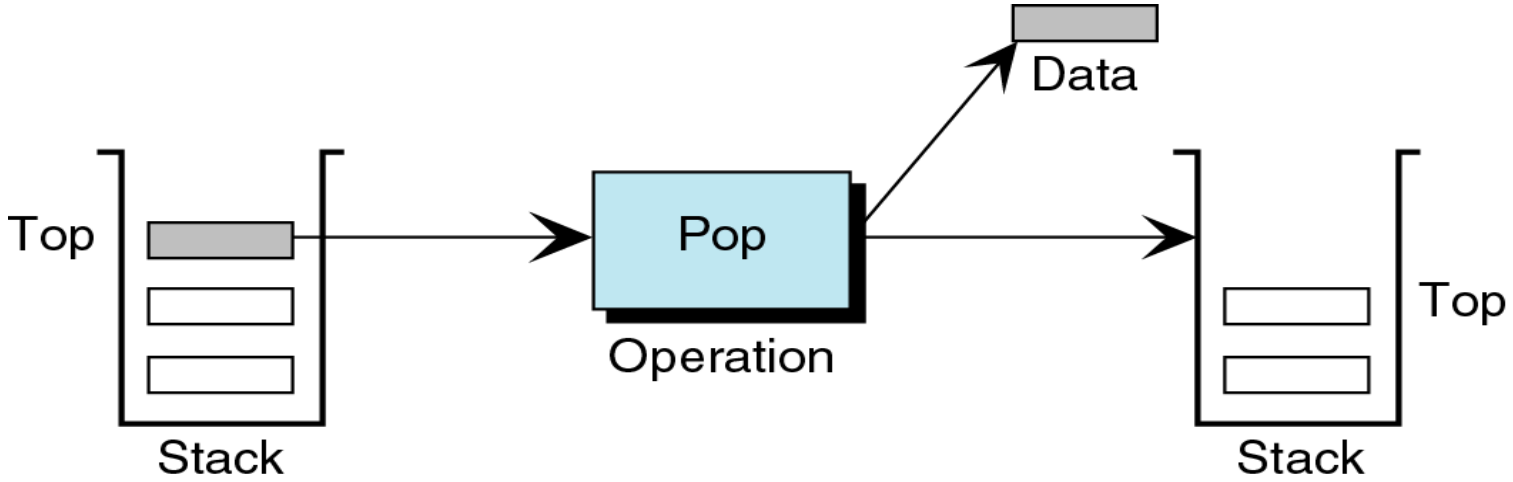


Fig 6-5 Stack pop [5]

Stack ADT

Top of Stack

- Retrieving the item at the top and return – not delete, either
  1. value or
  2. pair of value and address.
- If the stack is empty, return “underflow” stack.

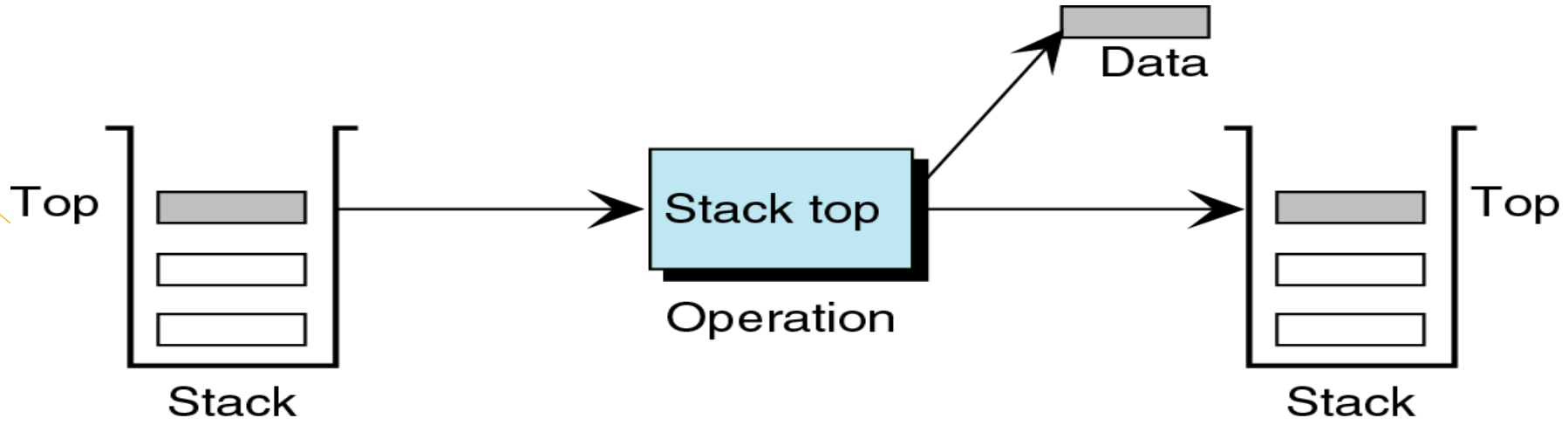


Fig 6-6 Stack top [5]

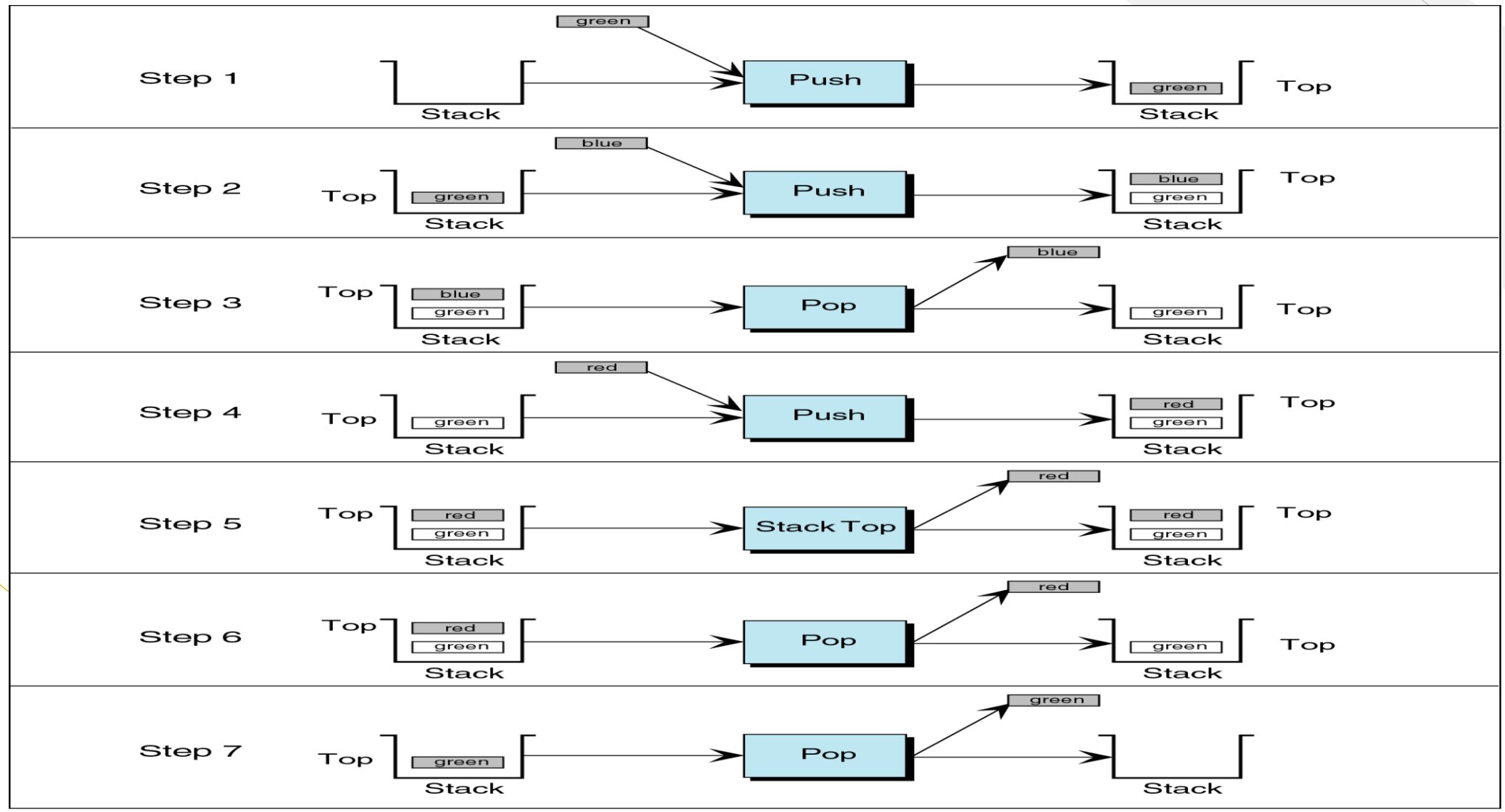


Fig 6-7 Stack operations [5]

6

Stack

Section 2:
Stack List

1) Stack Header Node
2) Stack List Application Programming Interface (API)
3) Stack List Implementation
4) Analysis Method’s Performance of Stack List

Stack List

- Stack is implemented with singly linked list whose head is top of stack.
- Implementing stack list is chosen when the space utilization and maintenance cost – insert, update and delete is highly concerned.

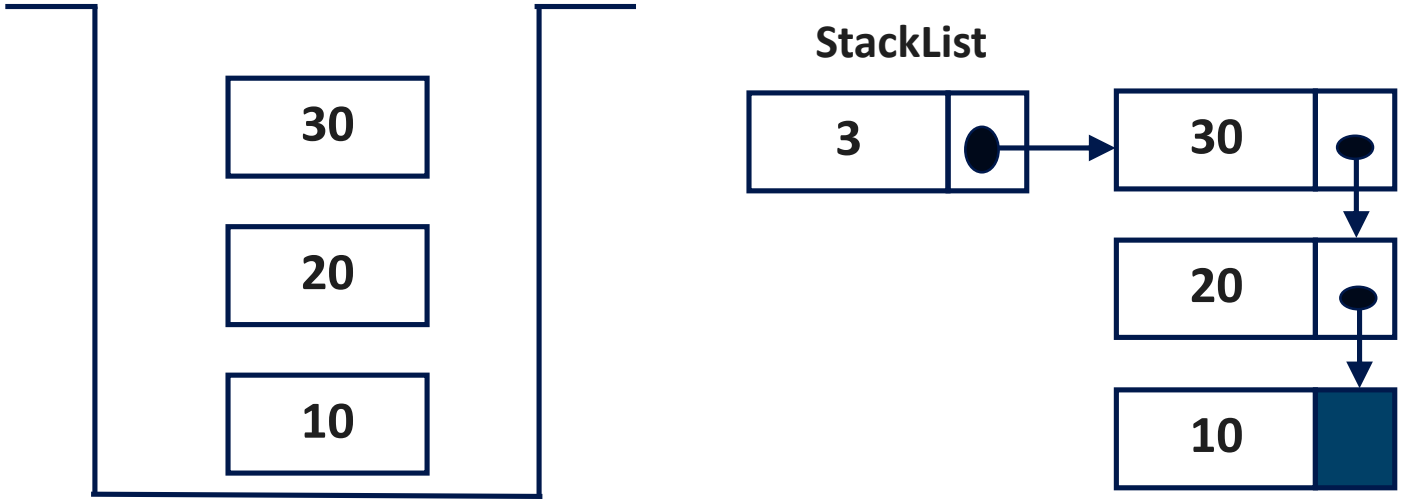


Fig 6-8 Conceptual and Physical view of stack list

Stack List

Header node

- The head attribute is renamed as the top to contain address of the top item.
- Like linked list, header node can be designed to contain various useful information.

```
1  class Node:
2    def __init__(self, data, next): # node constructor
3      self.data = data
4      self.next = next
5
6  class StackList:
7    def __init__(self): # Stack list constructor
8      self.top = None
9      self.count = 0
```

Fig 6-9 Stack list constructor

Stack List API

- **Create a new stack list**: Initial an empty stack linked list.
- **Traverse stack list**: Visit every data node in the stack list for serving an operation such as print, search, etc.
- **Empty stack**: Check whether the existing stack list contains at least a node or not. If it is empty, return true. Otherwise, return false.
- **Push**: Insert a new node into the top of stack list.
- **Pop**: Remove a target node from the stack top.
- **Top**: Return value at the top of stack list

Stack List API

Create a new stack list

- The header node contains address of the top data node and must be identified as an instance of stack list.

```
1  class Node:
2    def __init__(self, data, next): # node constructor
3      self.data = data
4      self.next = next
5
6  class StackList:
7    def __init__(self): # Stack list constructor
8      self.top = None
9      self.count = 0
```

Fig 6-10 Stack list constructor

Stack List API

Traverse stack list

- It traverses entire nodes in the stack list with a variable to walk through the stack list until to the end.

```
1  def DisplayStack(self): #Print entire data nodes
2      curNode = self.top
3      print("Print the list contains:", self.count, "nodes")
4      while curNode is not None:
5          print(curNode.data)
6          curNode = curNode.next
```

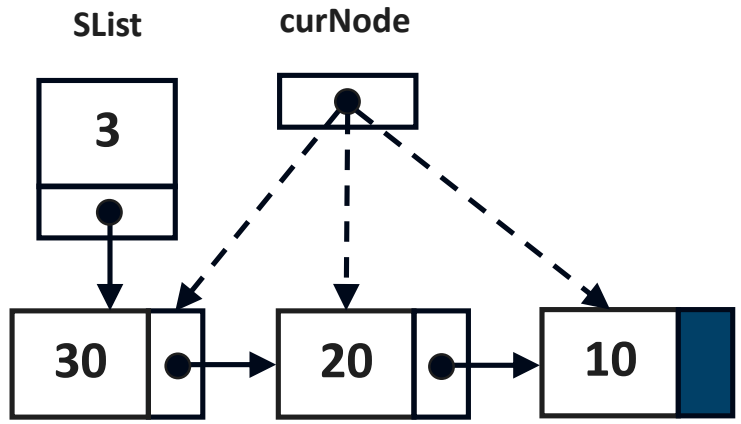


Fig 6-11 Traverse stack list

Stack List API

Verify an empty stack

```
1   def IsEmpty(self):
2     if self.top is not None:
3       return False
4     else:
5       return True
6  #--------------------
7  list1 = StackList() #Create a new stack list
8  if list1.IsEmpty(): #Is stack list empty?
9    print("Stack list is empty")
10 else:
11   print("Stack list is not empty")
```

Stack list is empty

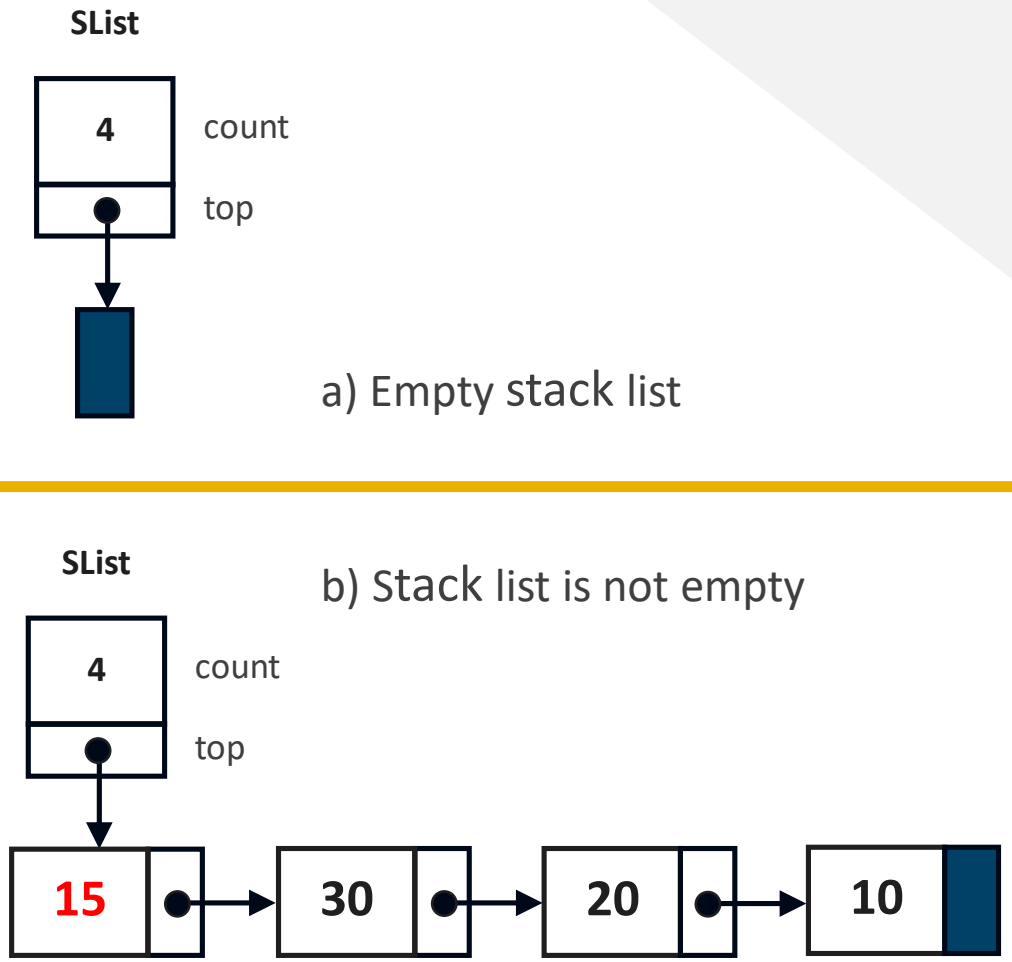


Fig 6-12 Empty stack verification

Stack List API

Stack list insertion

- Insert a new node to the stack list
- There is only one case can be used:
  1. Insert at head : Push
  2. ~~Insert at end or tail~~
  3. ~~Insert any node~~

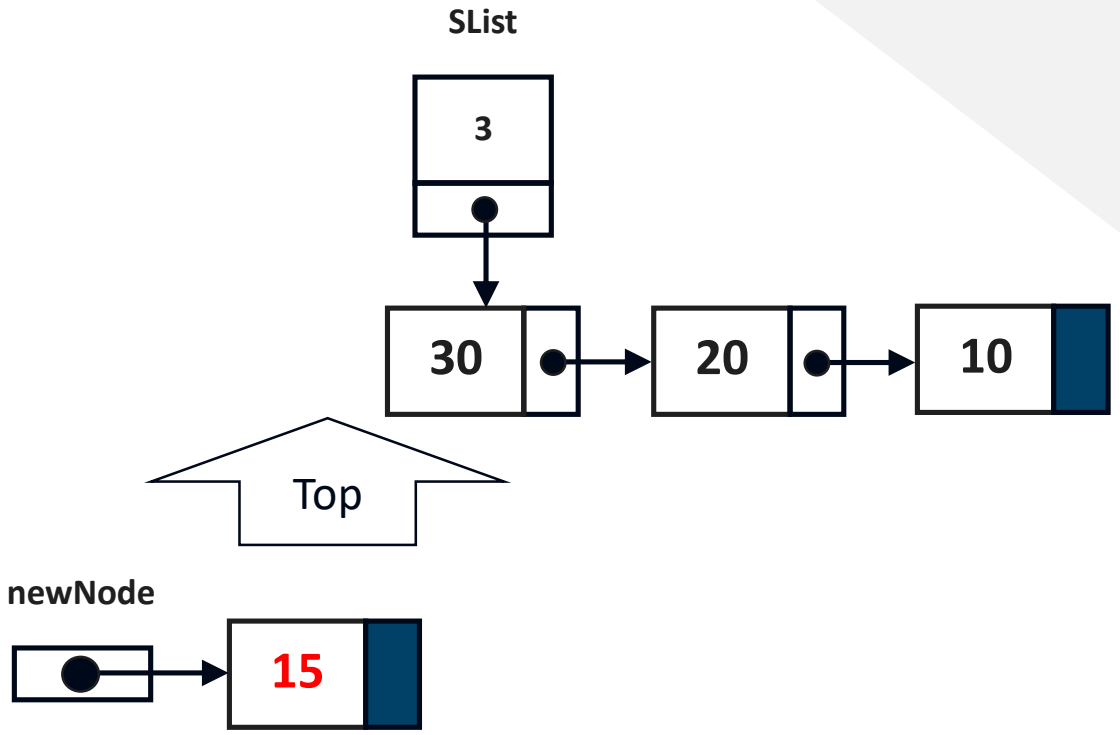


Fig 6-13 Stack list insertion

Stack List API

Stack push

```
1   def Push(self, data): #Insert new node to list top
2     newNode = Node(data, None)
3     newNode.next = self.top
4     self.top = newNode
5     self.count += 1
6  #--------------------
7  list1 = StackList() #Create a new stack list
8  #Push a new node
9  list1.Push(10)
10 list1.Push(20)
11 list1.Push(30)
12 list1.Push(15)
13 #Display the list
14 list1.DisplayStack()
```

```
Print stack contains: 4 nodes
15
30
20
10
```

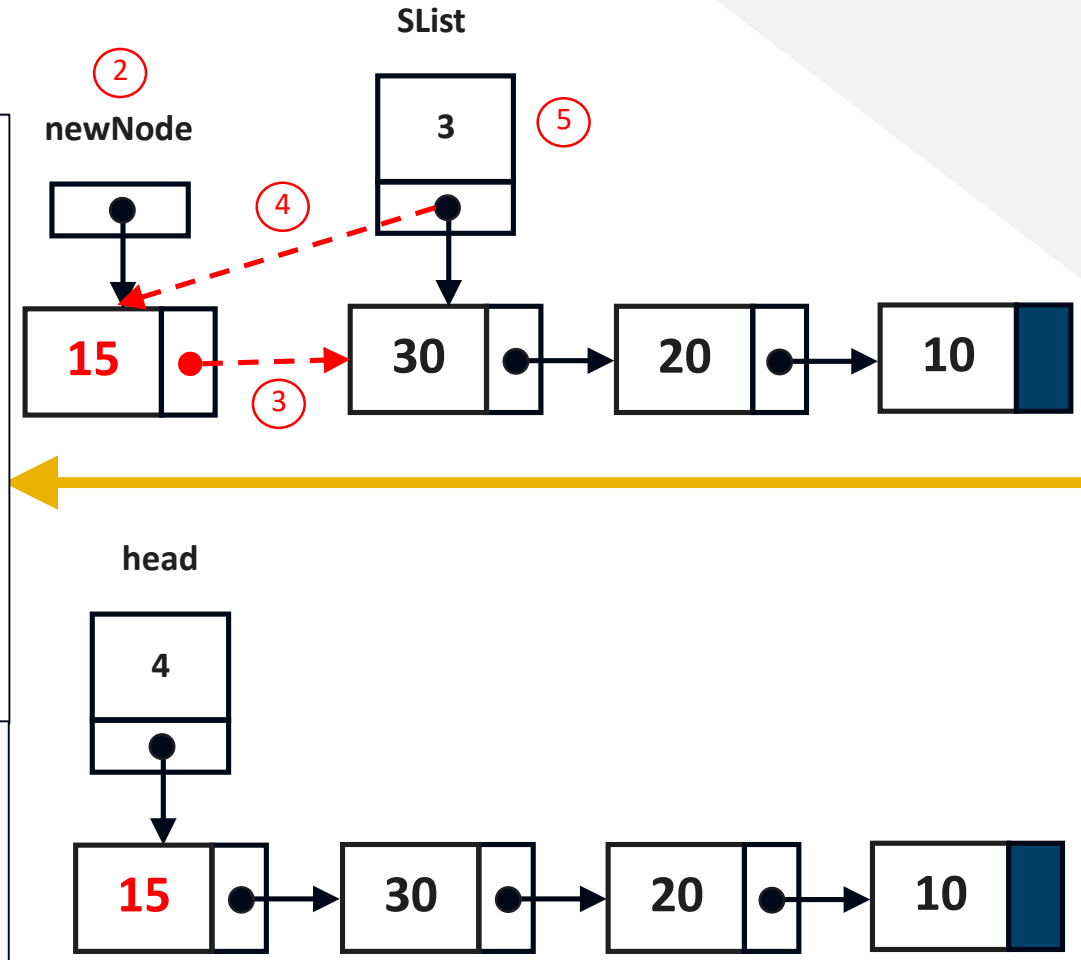


Fig 6-14 Stack list push

Stack List API

Stack list deletion

There is only case for the stack list:

1. Delete at head : Renamed as Pop
2. ~~Delete at end or tail~~
3. ~~Delete any node~~

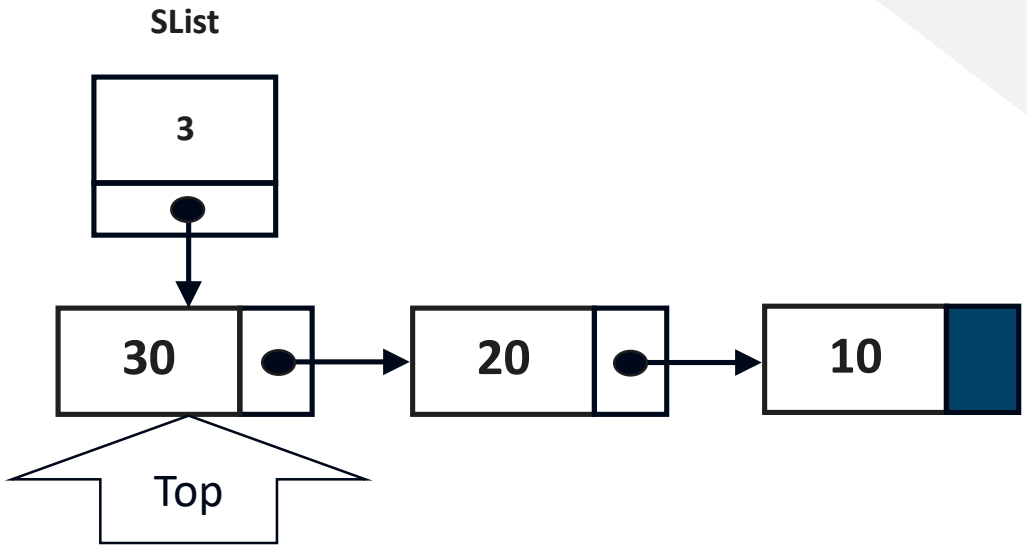


Fig 6-15 Stack list deletion

Stack List API

Stack pop

```
1   def Pop(self): #Delete a node at top
2     if self.IsEmpty()==False:
3       self.top = self.top.next
4       self.count -= 1
5     else:
6       print("Stack is underflow")
7  #--------------------
8  list1 = StackList() #Create a new stack list
9  #Push a new node
10 list1.Push(10)
11 list1.Push(20)
12 list1.Push(30)
13 list1.Push(15)
14 #Display the list
15 list1.DisplayStack()
16
17 #Pop
18 list1.Pop()
19 #Display the list
20 list1.DisplayStack()
```

```
Print stack contains: 4 nodes
15
30
20
10
Print stack contains: 3 nodes
30
20
10
```

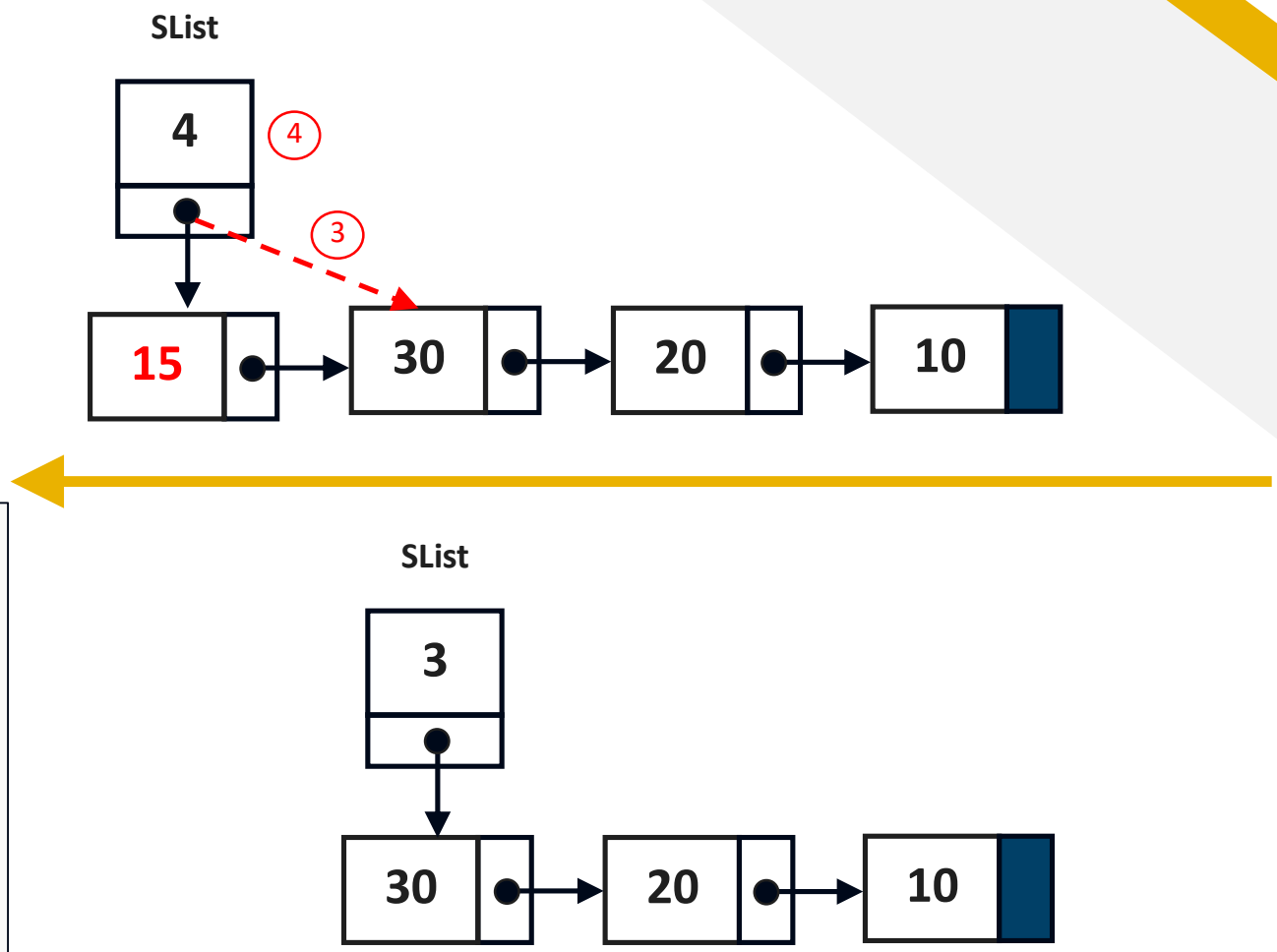


Fig 6-16 Stack list pop

Stack List API

Stack top

- Return the stack top –
  1. Node value
  2. Node address

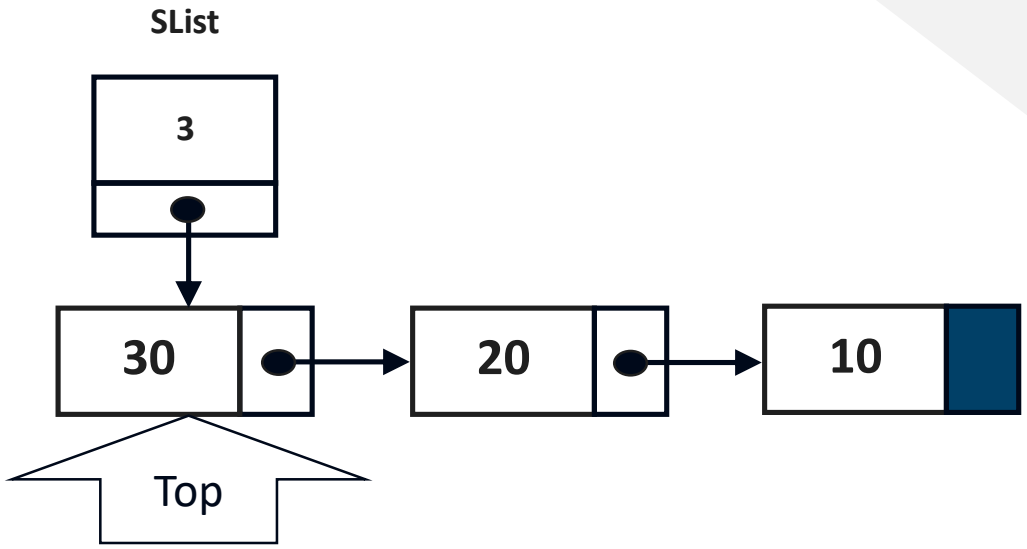


Fig 6-17 Stack list top

Stack List API

Stack top

```
1   def Top(self): #Retrieve a node at top
2     if self.IsEmpty()==False:
3       print("The top of stack list is : ", self.top.data)
4     else:
5       print("The top of stack list is none")
6
7  #Create a new stack list
8  list1 = StackList()
9  #Push a new node
10 list1.Push(10)
11 list1.Push(20)
12 list1.Push(30)
13 list1.DisplayStack() #Display the list
14 #Retrieve the stack top
15 list1.Top()
```

```
Print stack contains: 3 nodes
30
20
10
The top of stack list is :  30
```

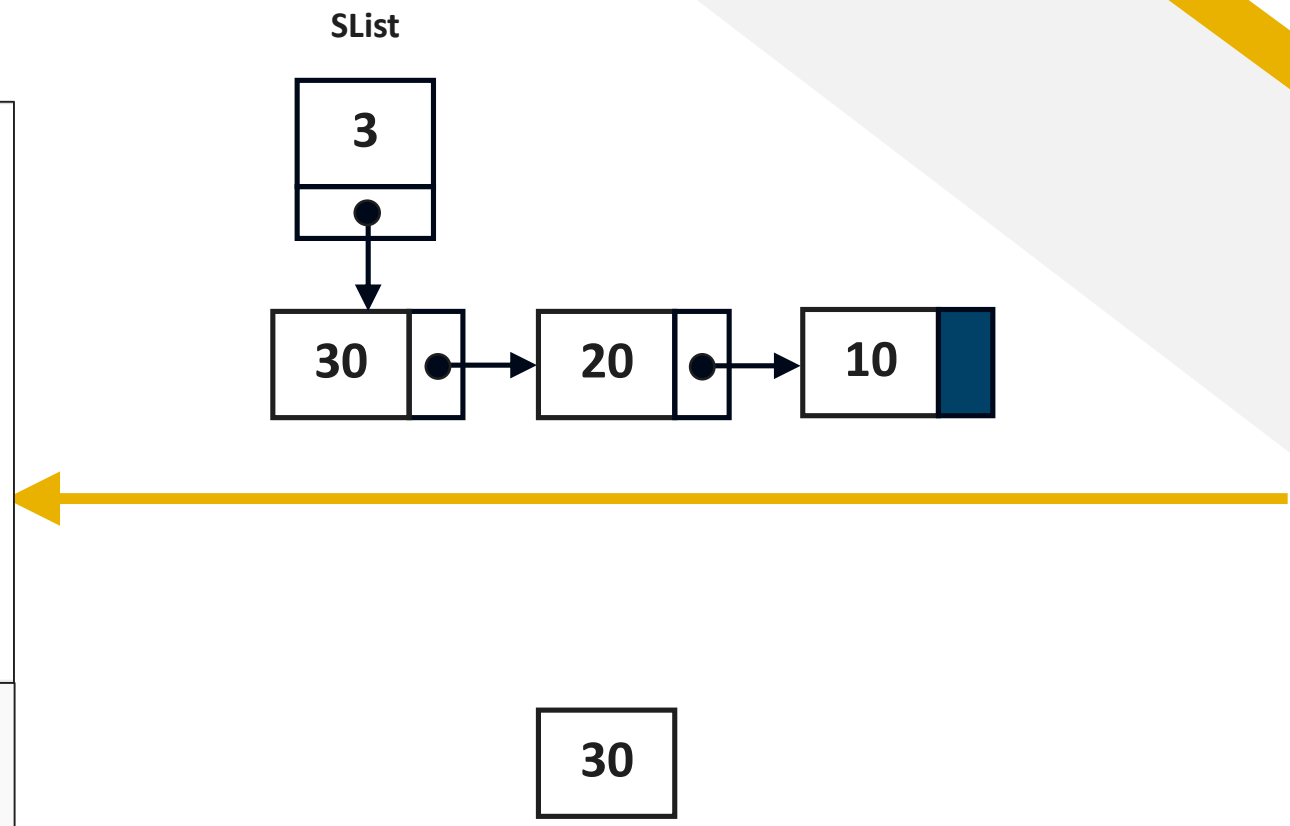


Fig 6-18 Stack list top

Tutorial 1: Stack List Implementation

2.3

```
class Node:
  def __init__(self, data, next): # node constructor
    self.data = data
    self.next = next

class StackList:
  def __init__(self): # Stack list constructor
    self.top = None
    self.count = 0

  def IsEmpty(self):
    if self.top is not None:
      return False
    else:
      return True

  def DisplayStack(self): #Print entire data nodes
    curNode = self.top
    print("Print stack contains:", self.count, "nodes")
    while curNode is not None:
      print(curNode.data)
      curNode = curNode.next

```

Tutorial 1: Stack List Implementation (Cont.)

2.3

```
24  def Push(self, data): #Insert new node to list top
25    newNode = Node(data, None)
26    newNode.next = self.top
27    self.top = newNode
28    self.count += 1
29
30  def Pop(self): #Delete and return a node at top
31    if self.IsEmpty()==False:
32      popValue = self.top.data
33      self.top = self.top.next
34      self.count -= 1
35    else:
36      print("Stack is underflow")
37    return popValue
38
39  def Top(self): #Retrieve and return a node at top
40    if self.IsEmpty()==False:
41      return self.top.data
42    else:
43      return "The top of stack list is none"
44
```

Tutorial 1: Stack List Implementation (Cont.)

2.3

```
45    def listSize(self): #Return node counter
46      return self.count
47  #---
48  list1 = StackList() #Create a new stack list
49  #Is stack list empty?
50  if list1.IsEmpty():
51    print("Stack list is empty")
52  else:
53    print("Stack list is not empty")
54  #Push a new node
55  list1.Push(10)
56  list1.Push(20)
57  list1.Push(30)
58  list1.Push(15)
59  list1.DisplayStack() #Display the list
60  #Pop and return the top node
61  print("The node - ", list1.Pop(), ", is poped.")
62  list1.DisplayStack() #Display the list
63  #Return a top node
64  print("The top of stack list is : ", list1.Top())
```

Analysis Methods' Performance

Stack list

Methods / Operations	Description	Big-O
Create stack list	Initial default value to stack list class – including top and count.	O(1)
Traverse stack list	Moving to the end of stack list which depend on total nodes in the stack list.	O(n)
Push	There are only update linkage of the top and new node without any traversal.	O(1)
Pop	The update linkage of the top to a successor node occurs without any shift or slide cost	O(1)
Top	Return a top node	O(1)
Empty stack list	It is only check value of the stack top whether is null or not.	O(1)

Table 6-1 Analysis stack list operations

6

Stack

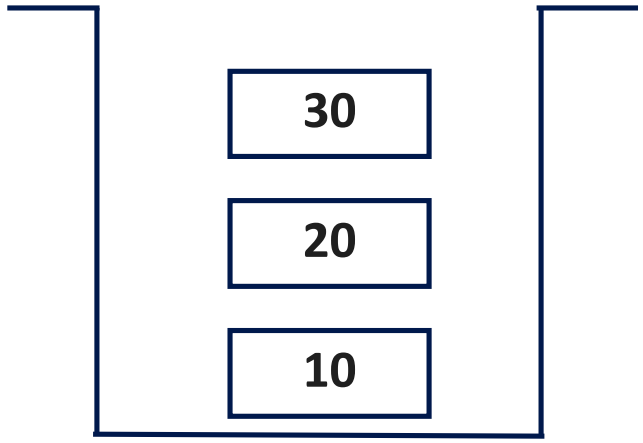
Section 3:
Stack Array

1) Stack Array Top
2) Stack Array Application Programming Interface (API)
3) Stack Array Implementation
4) Analysis Method’s Performance of Stack Array

Stack Array Top

Create stack array

- Recall to Python list, it provides an ordered collection mechanism and set of methods that can be easily adopted to implement the stack array.
- We need to design which side – between head and tail, of array will be implemented to be the top of stack.



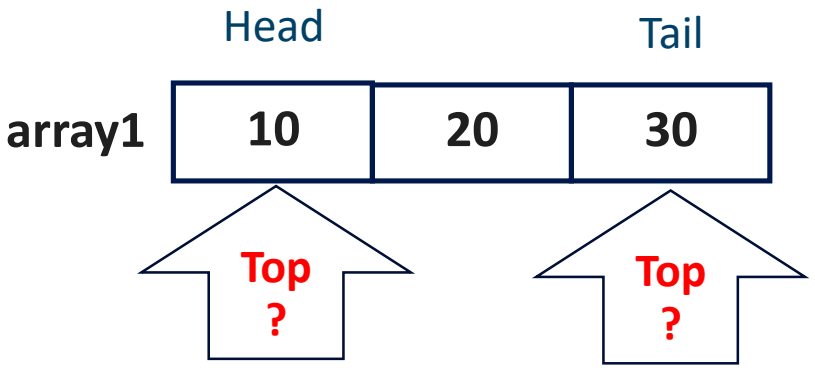
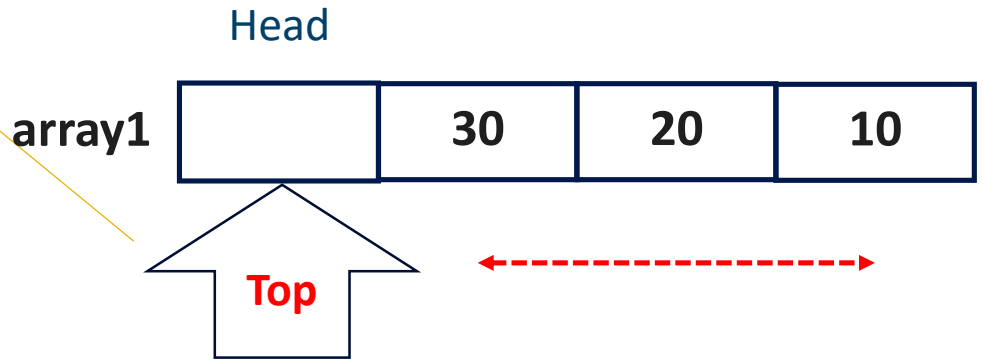


Fig 6-19 Conceptual and Physical view of stack array

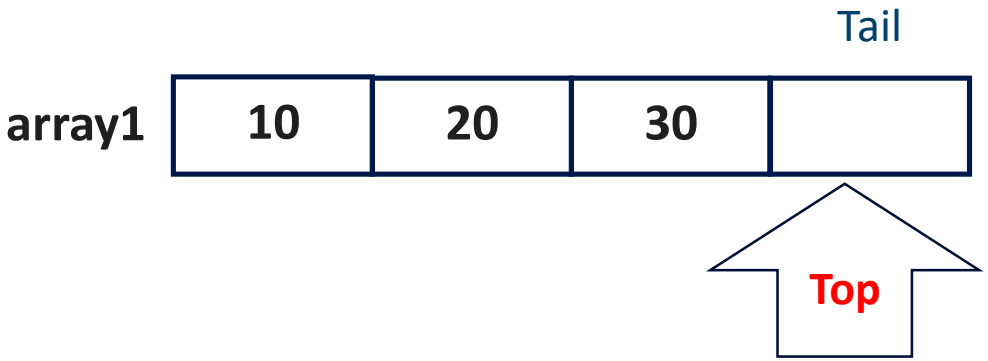
Stack Array Top

Stack inplementation

- **Head of array is top:** Shift element cost is needed to free space at head before inserting and get rid of a space at head after deleting
- **Tail is top of stack:** Inserting a new element to be a last element or deleting a last element can be done in common.



a) Head is top



b) Tail is top

Fig 6-20 Top of stack array

Stack Array API

- **Create a new stack array**: Initial an empty stack array.
- **Traverse stack array**: Visit every data node in the stack array for serving an operation such as print, search, etc.
- **Empty stack array**: Check whether the existing stack array contains at least a node or not. If it is empty, return true. Otherwise, return false.
- **Push**: Insert a new node into the top of stack array.
- **Pop**: Remove a target node from the stack top.
- **Top**: Return value at the top of stack array

Stack Array API

Create a new stack array

- Create a Python list in the constructor

```
1  class StackArray:
2    def __init__(self): # Stack array constructor
3      self.stackArr = []
```

Fig 6-21 Stack array constructor

Stack Array API

Traverse stack array

- It traverses entire elements in the stack array.
- Walk through the stack array until reach to the end is done by updating the array index.

```
1  def DisplayStack(self): #Print entire data nodes
2      print("Print stack contains:", self.StackSize(), "nodes")
3      i=len(self.stackArr)-1
4      while i>=0:
5          print(self.stackArr[i])
6          i-=1
```

Fig 6-22 Traverse stack array

Stack Array API

Empty stack array

```
1  def IsEmpty(self):
2      return self.stackArr == []
```

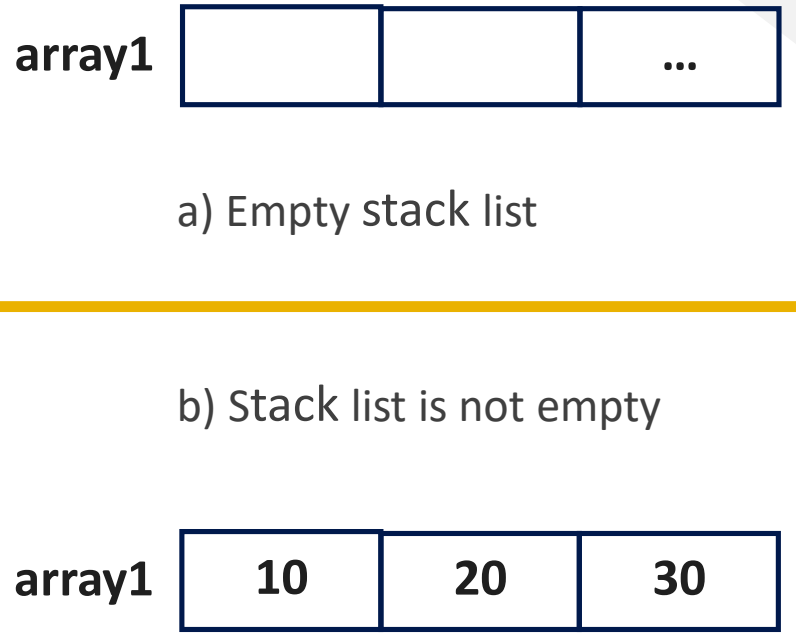


Fig 6-23 Verify an empty stack array

Stack Array API

Stack array push

```
1  def Push(self, data): #Insert new node to list top
2    self.stackArr.append(data)
```

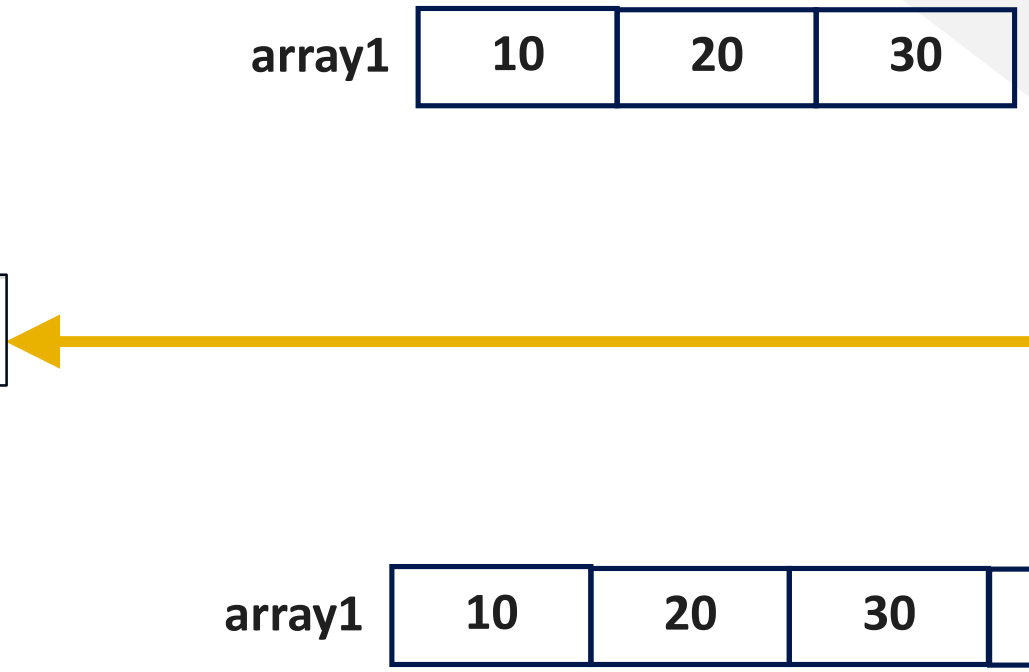


Fig 6-24 Stack array push

Stack Array API

Stack array pop

```
1  def Pop(self): #Delete a node at top
2    if self.IsEmpty()==False:
3      self.stackArr.pop();
4    else:
5      print("Stack is underflow")
```

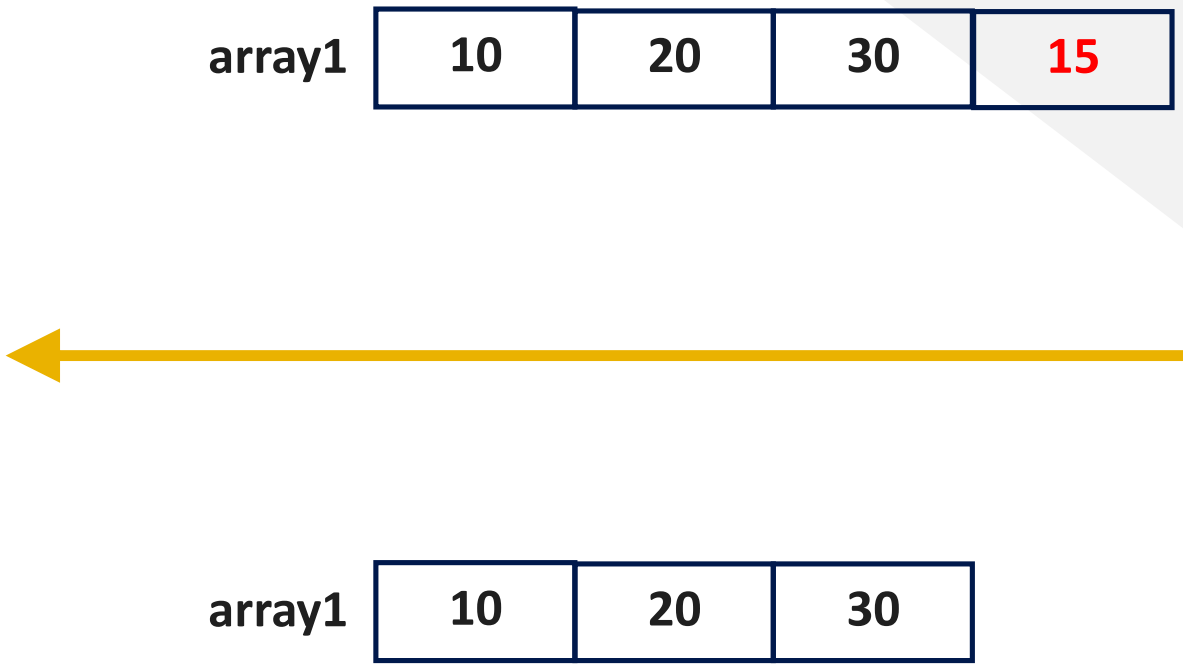


Fig 6-25 Stack array pop

Stack Array API

Stack top

```
1   def Top(self):
2       print ("Top of stack is at element: ",len(self.stackArr))
3       print ("Top of stack value is : ",self.stackArr[len(self.stackArr)-1])
4   #---
5
6   array1 = StackArray() #Create a new stack list
7   #Push a new node
8   array1.Push(10)
9   array1.Push(20)
10  array1.Push(30)
11  #Display the list
12  array1.DisplayStack()
13  #Top
14  array1.Top()
```

```
Print stack contains: 3 elements
30
20
10
Top of stack is at element:  3
Top of stack value is :  30
```

Fig 6-26 Stack list top

```
1  class StackArray:
2    def __init__(self): # Stack array constructor
3      self.stackArr = []
4
5    def IsEmpty(self):
6        return self.stackArr == []
7
8    def DisplayStack(self): #Print entire data nodes
9      print("Print stack contains:", self.StackSize(), "elements")
10     i=len(self.stackArr)-1
11     while i>=0:
12       print(self.stackArr[i])
13       i-=1
14
```

```
15  def Push(self, data): #Insert new node to list top
16    self.stackArr.append(data)
17
18  def Pop(self): #Delete a node at top
19    if self.IsEmpty()==False:
20      popValue = self.stackArr[len(self.stackArr)-1]
21      self.stackArr.pop();
22    else:
23      print("Stack is underflow")
24    return popValue
25
26  def Top(self):
27    return self.stackArr[len(self.stackArr)-1]
28
29  def StackSize(self): #Return node counter
30    return len(self.stackArr)
31
```

```
#---
array1 = StackArray() #Create a new stack list
if array1.IsEmpty(): #Is stack list empty?
  print("Stack array is empty")
else:
  print("Stack array is not empty")
#Push a new node
array1.Push(10)
array1.Push(20)
array1.Push(30)
array1.Push(15)
array1.DisplayStack() #Display the list
#Pop and return a top node
print ("The pop element is : ",array1.Pop())
array1.DisplayStack() #Display the array stack
#Retrun the top element
print ("Top of stack value is : ",array1.Top())
```

Analysis Method's Performance of Stack Array

Methods / Operations	Description	Big-O
Create a python list	Initial default value to stack list class – including top and count.	O(1)
Traverse stack array	Update the array index to traverse every node of stack array	O(n)
Push	Append a new element to the end of array	O(1)
Pop	Remove the last element of stack array	O(1)
Top	Return a last element of stack array	O(1)
Empty stack list	It is only check whether stack array is empty or not	O(1)

Table 6-2 Analysis of stack array operations

6

Stack

Section 4:
Stack Applications

1) Reversing
2) Parsing
3) Postponing
4) Backtracking

Stack Applications

- Reversing
  - Covert a decimal number to be a binary number
- Parsing :
  - Match brace pairwise in a program source code
- Postponing a data usage :
  - Convert infix notation to postfix notation
  - Evaluate the postfix expression
- Backtracking steps
  - Keep a candidate choices to be tested later

Reversing

Reversing

- Stack is used to contain entire remainders from diving a decimal number with base 2.
- After that reverse order of binary number in the stack by calling pop operation until the stack is empty.

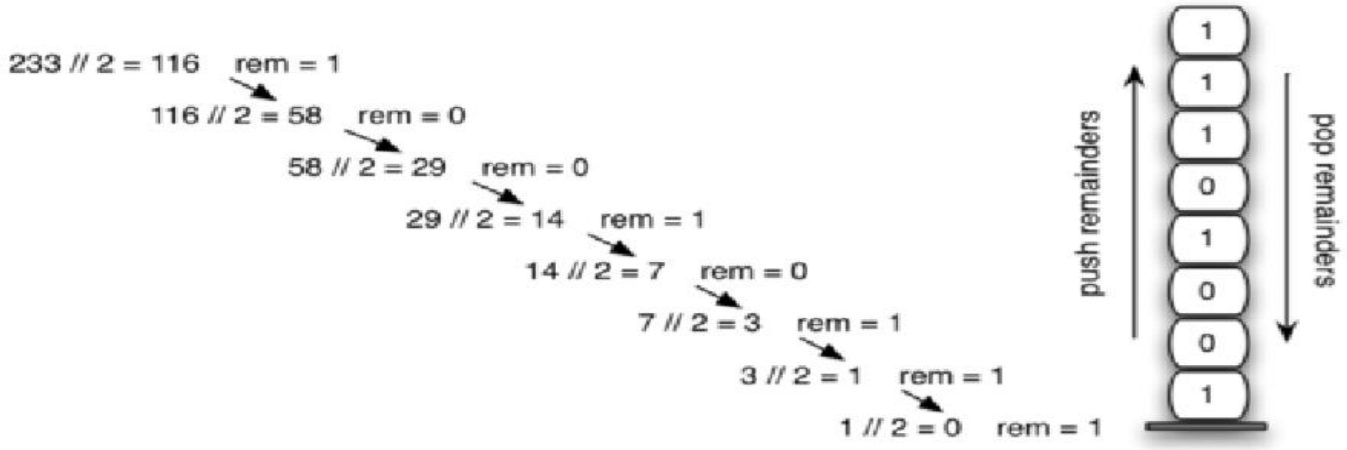


Fig 6-27 Decimal number to Binary number conversion [3]

Tutorial 3: Decimal to Binary Number Conversion

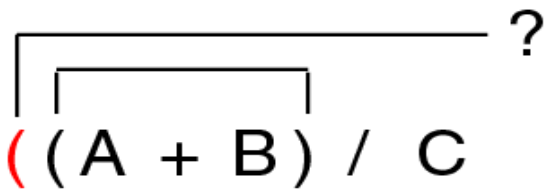
4.1

```
#import stack
def base_converter (dec_number, base):
  digit = "0123456789ABCDEF"
  remainder_stack = StackList()
  while dec_number > 0: #Push remainders to stack
    remainder = dec_number % base
    remainder_stack.Push(remainder)
    dec_number = dec_number // base
  new_string = ""
  while not remainder_stack.IsEmpty(): #Pop remainders
    new_string = new_string + digit[remainder_stack.Pop()]
  return new_string

print("The converted outcome is : ",base_converter(25,2))

```

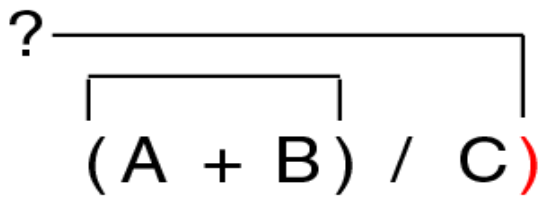
```
The converted outcome is :  11001
```

Parsing

- When parentheses are unmatched, there are two possible cases.



(a) Opening parenthesis not matched



(b) Closing parenthesis not matched

Fig 6-28 Parentheses parsing [3]

Parsing

check_parentheses ('{()}()}')	Operations	Outcome
{	Push (“{“)	-
(	Push (“(“)	-
)	Pop() and Matches(“(“,”)”)	Balanced = True
{	Push (“{“)	-
(	Push (“(“)	-
)	Pop() and Matches(“(“,”)”)	Balanced = True
}	Pop() and Matches(“{“,”)”)	Balanced = False

Table 6-3 Parentheses matching

Tutorial 4: Parentheses Parsing

4.2

```
#import Stack
def check_parentheses(par_string):
  par_stack = StackArray() #Create a new stack array
  balanced = True
  index = 0
  while index < len(par_string) and balanced:
    par = par_string[index]
    if par in "([{":
      par_stack.Push(par)
    else:
        if par_stack.IsEmpty():
          balanced = False
        else:
          top = par_stack.Pop()
          if not matches(top, par):
            balanced = False
    index += 1
  if balanced and par_stack.IsEmpty():
    return True
  else:
    return False
def matches(open, close): #Check whether it is paired or not
  open_par = "([{"
  close_par = ")]}"
  return open_par.index(open) == close_par.index(close)
print ("The matched parentheses are : ",check_parentheses("{()}()}"))
```

```
The matched parentheses are :  False
```

Postponing

Postponing data usage

- Arithmetic expression understand by human is infix form however it’s unknown by machine.
- An infix expression required to classify the sequence of operating since the precedence of operators are not equivalent.
- Justification the sequence of operation can be done by converting infix form to postfix form.
- Evaluating an expression requires two steps of work:
  1. Convert infix expression to postfix expression
  2. Evaluate the postfix expression to outcome the result

Postponing

Postponing data usage

- Three forms of expressions:
  1. Infix expression: A@B
     The operator is between two operands.
  2. Postfix expression: AB@
     The operator is at the end of two operands.
  3. Prefix expression: @AB
     The operator is in front of two operands.

Infix Expression	Prefix Expression	Postfix Expression
$A + B$	$+AB$	$AB+$
$A + B * C$	$+A * BC$	$ABC * +$

Fig 6-29 Expression forms [3]

Postponing

Convert expression

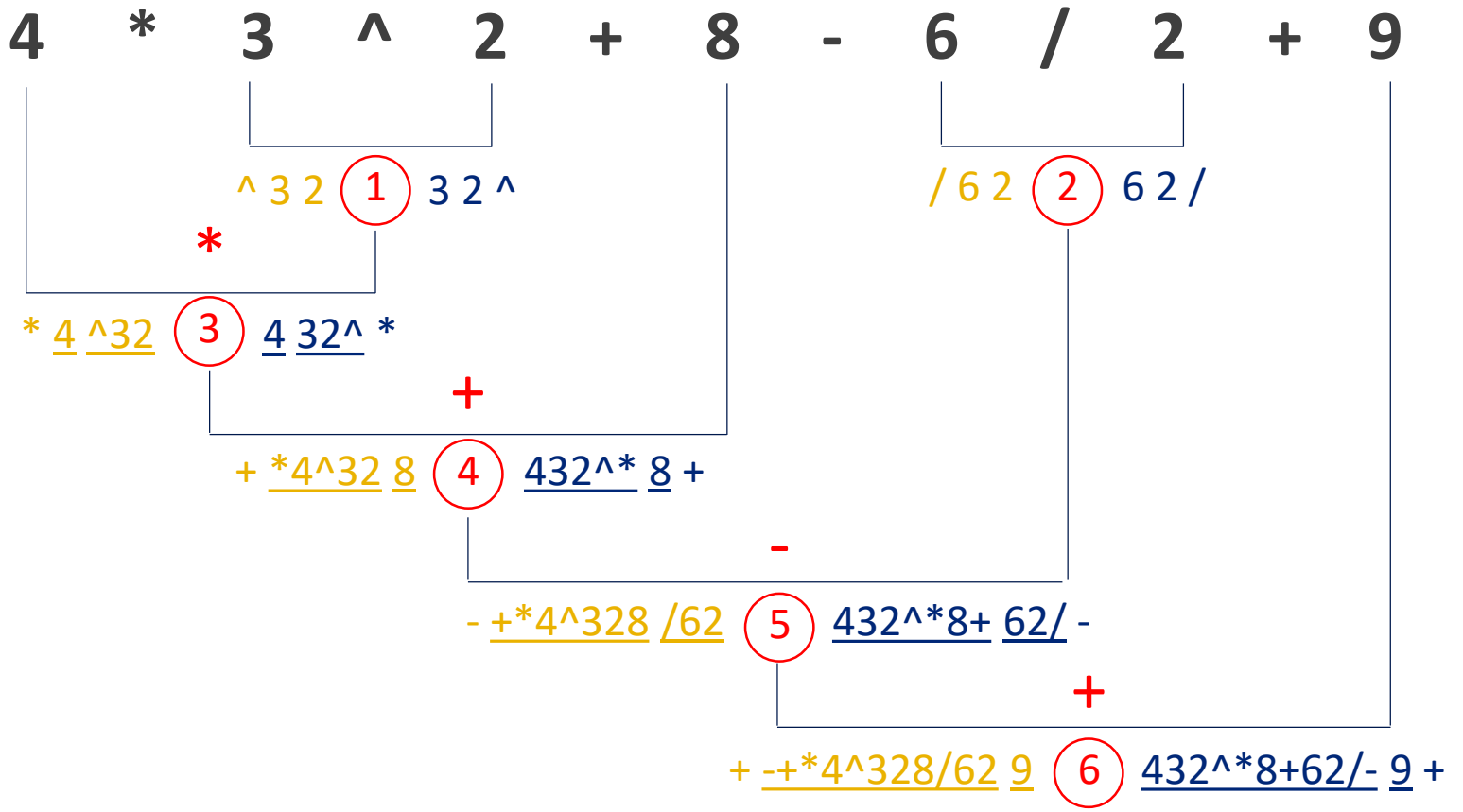


Fig 6-30 Convert expression

Tutorial 5: Convert infix to Postfix

4.3

```
#import Stack
def check_parentheses(par_string):
  par_stack = StackArray() #Create a new stack array
  balanced = True
  index = 0
  while index < len(par_string) and balanced:
    par = par_string[index]
    if par in "([{":
      par_stack.Push(par)
    else:
        if par_stack.IsEmpty():
          balanced = False
        else:
          top = par_stack.Pop()
          if not matches(top, par):
            balanced = False
    index += 1
  if balanced and par_stack.IsEmpty():
    return True
  else:
    return False
def matches(open, close): #Check whether it is paired or not
  open_par = "([{"
  close_par = ")]}"
  return open_par.index(open) == close_par.index(close)
print ("The matched parentheses are : ",check_parentheses("{()}()}"))
```

```
The matched parentheses are :  False
```

Postponing

Convert infix to postfix

	Infix	Stack	Postfix
(a)	A+B*C−D/E		
(b)	+B*C−D/E		A
(c)	B*C−D/E	+	A
(d)	*C−D/E	+	AB
(e)	C−D/E	* +	AB
(f)	−D/E	* +	ABC
(g)	D/E	−	ABC*+
(h)	/E	−	ABC*+D
(i)	E	/ −	ABC*+D
(j)		/ −	ABC*+DE
(k)			ABC*+DE/−

Fig 6-31 Convert infix to postfix [4]

Postponing

Evaluate postfix

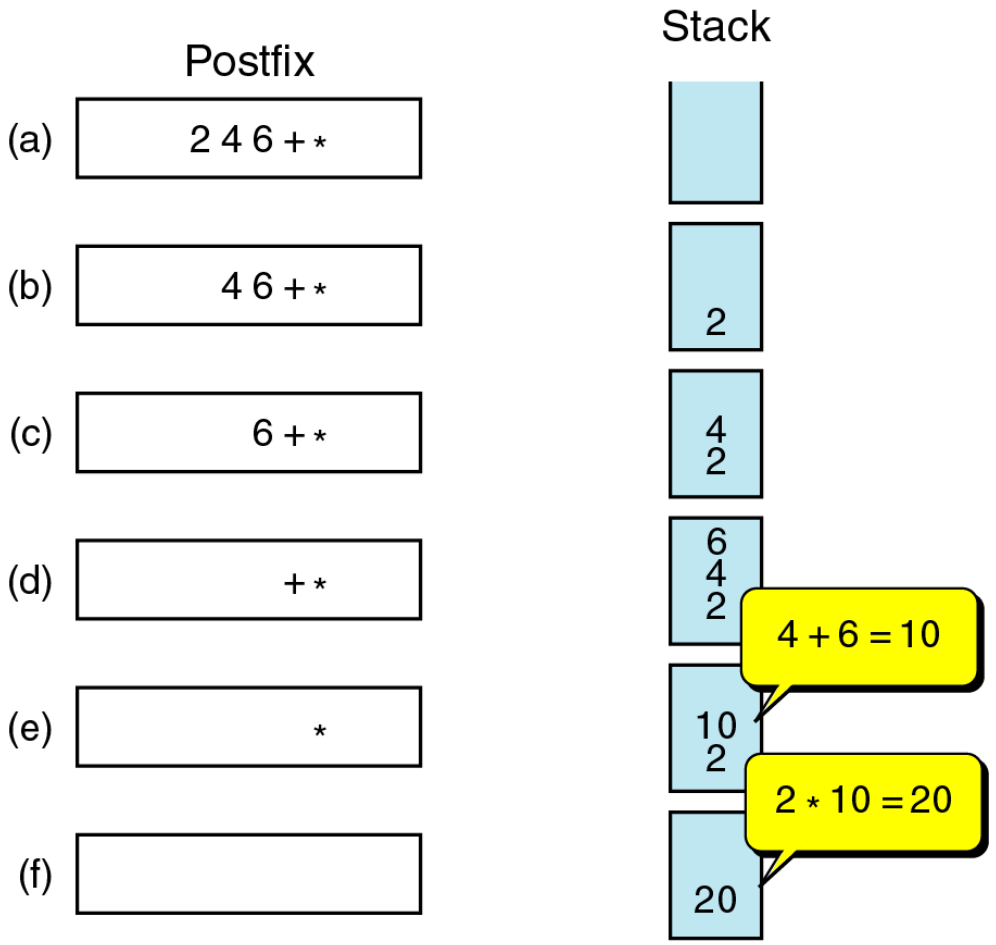


Fig 6-32 Evaluate stack [4]

Backtracking

- Another stack use is found in applications :
  - Computer gaming,
  - Decision analysis, and
  - Expert system
- Classical application to show how to use stack to solve the problems :
  - Goal seeking and
  - Eight Queens problem

Backtracking

- One way to portray the problem is to layout the steps in the form of a graph that contains a several alternative paths in it.
- Only one path leads us to a desired goal => The algorithm will determine the correct path.
- Steps of work :
  - From the start point move right until we hit a branching node
  - Take the path step by step by starting at the upper path and continue until we hit a branching node, end point or goal point.
  - Stop and return the route from the start point to the goal point.

Backtracking

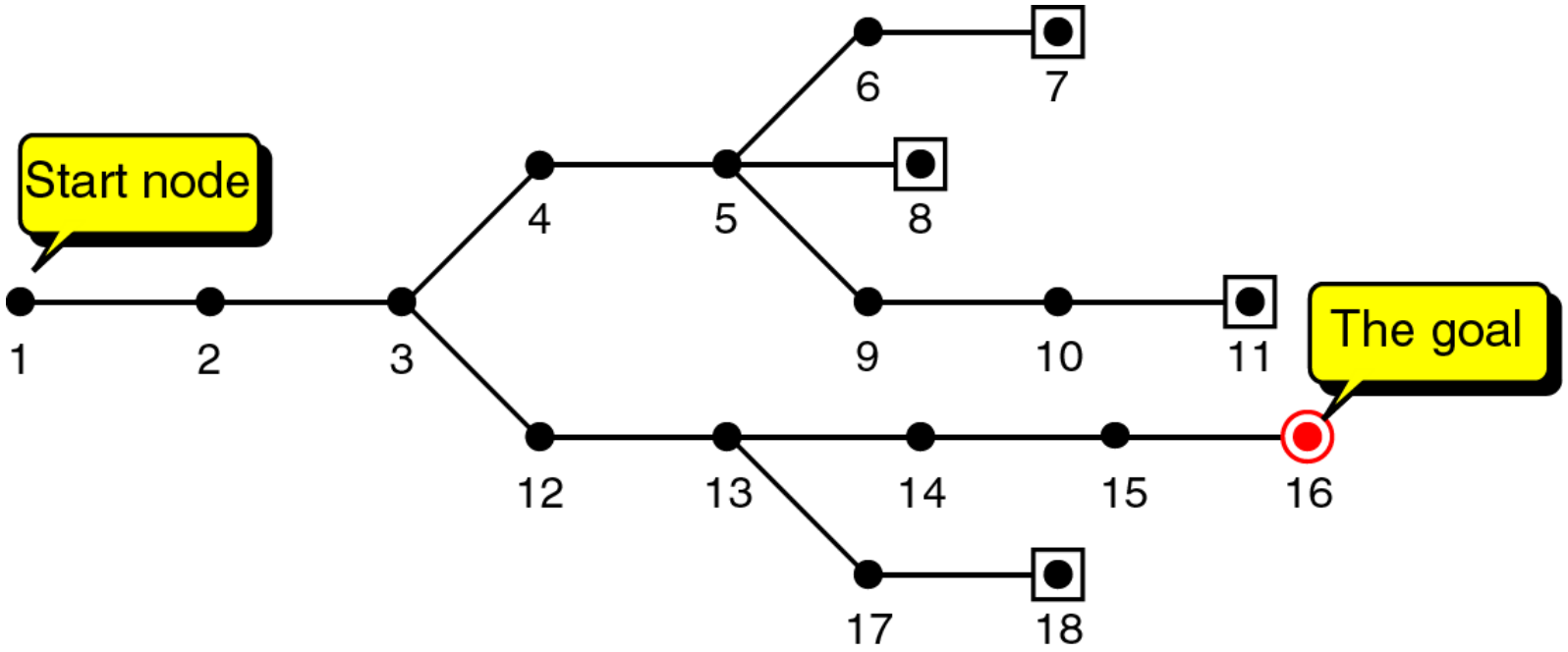


Fig 6-33 Backtracking [4]

References

Texts | Integrated Development Environment (IDE)

[1] Data Structures and Algorithms in Python, Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser, Willy & Sons Inc., 2013.

[2] Data Structures and Algorithms Using Python, Rance D. Necaise, John Winley & Sons, Inc., 2011.

[3] Problem Solving with Algorithms and Data Structures, Brad Minller and David Ranum, Python, 2013.

[4] Data Structures: A Pseudocode Approach with C++, Richard F. Gilberg and Behrouz A. Forouzan, Brooks/Cole, 2001.

[5] Problem Solving in Data Structures & Algorithms Using Python: Programming Interview Guide, 1st Edition, Hermant Jain, Thiftbooks, March 2017.

[6] https://trinket.io/features/python3

[7] http://interactivepython.org/courselib/static/pythonds/BasicDS/ImplementingaStackinPython.html