Data Structures & Algorithms

Week 7

Stacks

Lecturer: Dr. Msagha J Mbogholi, PhD

Flashback from Lesson 6

- Inheritance: the concept of one class using the methods of another class. This class shares some characteristic(s) of the class whose methods it uses.
- There are four types of inheritance: single, multiple, multilevel, and hierarchical.
- Overriding: the ability of a class to inherit the methods of its parent class, and customize them to suit its needs.
- Polymorphism: is to take many forms. In Python this means the same action being performed in different ways. In polymorphism, a method can process objects differently depending on the class type or data type. Overriding is a form of polymorphism already.
- An abstract class is not normally instantiated in client applications. Its subclasses are called **concrete classes** to indicate that they are the classes actually used to create objects in client applications. Both categories of classes are in turn distinguished from *interfaces*, which simply specify the methods of a given class or set of classes without any implementing code.

Content

- Introduction
- Overview
- Stack ADT
- Implementation
- Applications
- Advantages and disadvantages

Part 1

Introduction

Introduction

- According to (Galal, 2023) the literacy rate in Africa stood at approximately 67.4% as at 2021. This is approximately the population in Africa that could read and write a simple statement and understand it.
- The reasons for this low (relatively) literacy rate can be attributed to many factors….political, historical, and so on. Before colonization African societies (and many others) had their own ways of keeping records (simple counting, drawing on walls, and the most common was oral hand downs through the generations). I wonder if this can be counted as being illiterate? Story for another day.
- Nonetheless the school experience, attending school at least up to high school (or what is called secondary school in other societies) always has a profound effect on the learner in terms of learning new things, expanding your brain comprehension abilities, and so on.

Introduction

- I find from experience that what we learned in school, whether intentionally or not, will have some application in real life at some point or the other.
- When I was in high school we did various tests and exams during the term; the difference was that tests were done during the term while exams were done twice (mid-term and end term).
- Whenever our teachers were done marking the tests (or exams) they would return our answer scripts back to us so we could review the answers and learn how to answer questions better come the next assessment. What was interesting is the method they used to return the scripts.

Introduction

- The traditional way of doing this was for the teacher to come to class with the scripts and call out the names of the script owners (as we waited with anticipation), one by one, to the last script.
- The interesting thing here was the order of calling: some teachers would call out based on the order in which they marked the scripts, others would call out based on the classroom sitting arrangement (from front row to the back, or vice versa), yet others would arrange the scripts using the student registration number.
- The above methods were to be expected and didn't surprise most of us. However, the teachers we admired most were the ones who used a different order in returning the scripts.
- I remember these ones as if it was just yesterday. They used two methods; one would arrange the scripts in order of marks, in ascending or descending order.

Introduction

- You may be wondering, why did we admire this group so much?
- I was in a very competitive school (which was always among the top three in national examinations) and most students were very competitive. As you would expect this arrangement by these teachers helped us to know who had performed the best (or worst!) in the class.
- We would listen keenly as names were called out and of course with time we got to know which order a particular teacher was using…and by extension you would know where you stood in the overall class pecking order.
- Consequently, we could tell if this time s/he (the teacher) was using ascending or descending order (since we knew which classmates were traditionally in the top quartile of the class in terms of academic performance).

Introduction

- And so, by taking a **peek** at what the lowest had, and any other person near the top (or close enough to the top) we could guess what the highest mark (since most top students did not share their marks ☺) was.
- This type of ordering has influenced yours truly as well, I must confess. Wherever possible I also arrange test scripts in this fashion. As for exam scripts we have an internal policy regarding how these should be arranged.
- This ordering has greatly influenced most data structures, especially in terms of how items are arranged (ordered) in a group. In earlier lessons we looked at how items are sorted in a list and the complexity of such operations.
- In this lesson we introduce the stack ADT and discuss ways in which items are added, removed and examined in it. We discuss the ADT, its implementation, applications thereof; lastly an examination of advantages and disadvantages of the stack is discussed.

Part 2

Overview

2.1 Definition

- Lambert (2019) describes stacks as follows: "Stacks are linear collections in which access is completely restricted to just one end, called the **top**." conversely, the bottom of the stack is also known as its base.
- The common analogy of a stack is given by Lambert (2019), Necaise (2011) and Prasanna (2021) as follows: The classic analogous example is the stack of clean trays found in every cafeteria. Whenever a tray is needed, it is removed from the top of the stack, and whenever clean ones come back from the kitchen, they are again placed on the top. No one ever takes some particularly fine tray from the middle of the stack, and trays near the bottom might never be used.
- Based on the foregoing we can gather that stacks use the LIFO (Last In, First Out) method where the tray at the bottom never gets to be used unless there's no tray on top of it.

2.1 Definition

- Karumanchi (2017) defines a stack as follows: "A *stack* is an ordered list in which insertion and deletion are done at one end, called *top*. The last element inserted is the first one to be deleted. Hence, it is called the Last in First out (LIFO) or First in Last out (FILO) list."
- Thus we see that the most important points to note regarding the stack are:
  - It is an ordered list
  - Insertion and deletion are done at one end only (the top)
  - It uses the LIFO (last in, first out) principle
  - Items at the bottom of the stack can't be removed unless the items on top of them have already been removed (from the cafeteria analogy)

Part 3

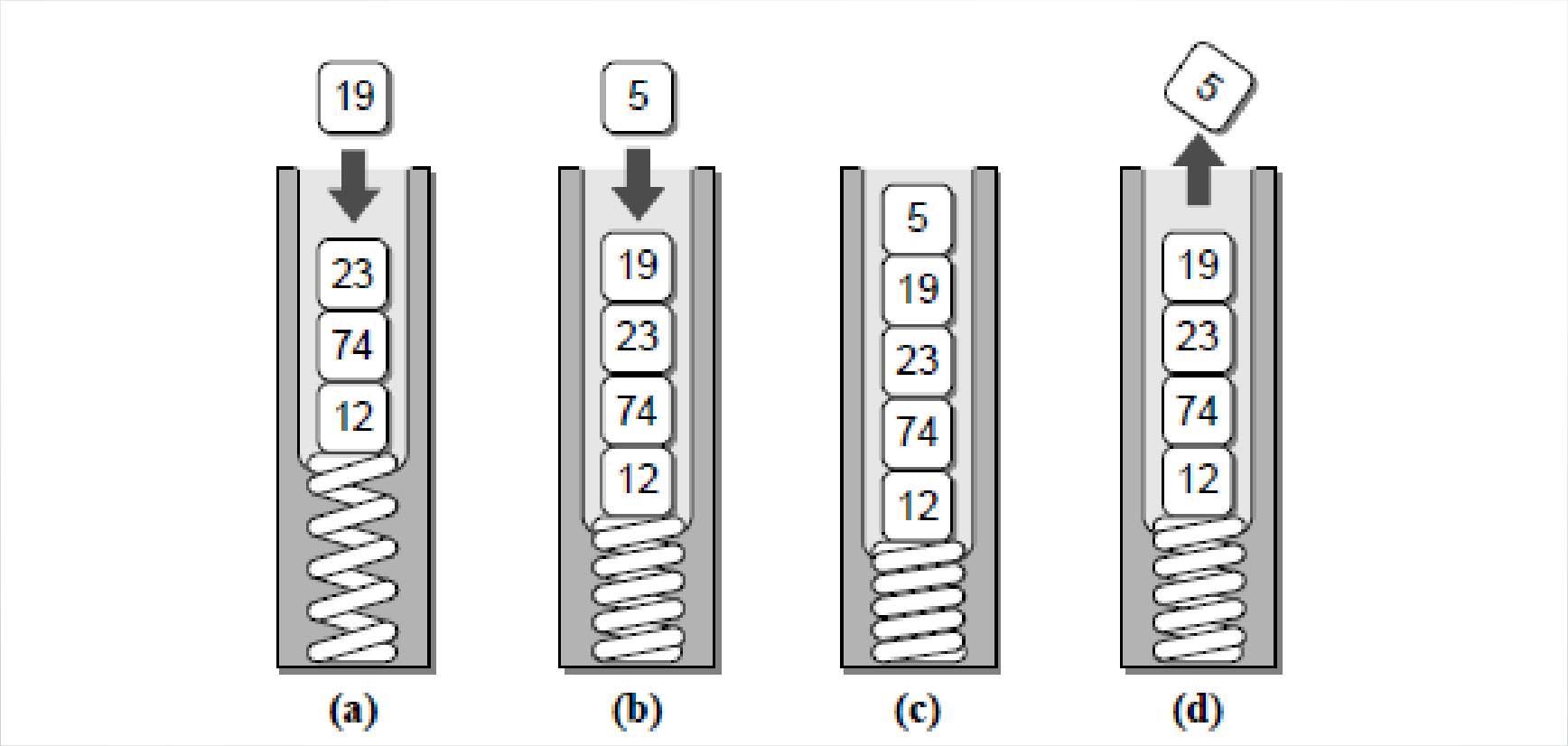
Stack ADT

3.1 Introduction

- As described in part 2 of this lesson, the stack uses a LIFO principle in administering to its elements. This is to say that the element that is placed in the stack last is the same one that is removed first.
- Of course this principle can't work in a place like a supermarket or butchery, right? This is why they use a different principle which we shall discuss in a later lesson.
- The issue is whether this principle has any application in real life? Since who would want to remove the item at the top of the stack first…what happens to the one at the bottom?
- The answer to this will come later in this lesson when we discuss applications of stacks.

Figure 1.

Stack operations



(From Necaise, 2011)

3.2 Stack operations

- The operations for putting items on and removing items from a stack are called push and pop, respectively.
- In figure 1 we show an abstract representation of these operations. In figure 1(a) we see an existing stack consisting three items: 23, 74 and 12. The item 23 is at the top of the stack, while 12 is at the base.
- We wish to add a new item to the stack, namely 19. In order to do this we invoke the push operation, and push 19 into the stack….it is now at the position called 'top' of the stack. The new order is now 19, 23, 74 and 12.
- In figure 1(b) we add a second item to the stack, namely 5; it now becomes the item at the top of the stack and the new order is now 5, 19, 23, 74, and 12.

3.2 Stack operations

- In figure 1(c) the order of the stack is now shown with all the items from the top to the base; 5 (top), 19, 23, 74, and 12(base).
- In figure 1(d) we wish to remove an item from the stack; by the rules all operations of removal and adding can only be done from one end of the stack (top) using the LIFO principle. Therefore, we pop the top item which is 5. The new order of the stack is now 19, 23, 74, and 12.
- We can also view what is at the top of the stack without removing it using the peek operation.
- There are two terms related to stacks which are described by Karumanchi (2017): "Trying to pop out an empty stack is called *underflow* and trying to push an element in a full stack is called *overflow*. Generally, we treat them as exceptions."

Figure 2

Stack ADT

Define **Stack ADT**

A *stack* is a data structure that stores a linear collection of items with access limited to a last-in first-out order. Adding and removing items is restricted to one end known as the *top* of the stack. An *empty stack* is one containing no items.

- `Stack()`: Creates a new empty stack.
- `isEmpty()`: Returns a boolean value indicating if the stack is empty.
- *`length`*`()`: Returns the number of items in the stack.
- `pop()`: Removes and returns the top item of the stack, if the stack is not empty. Items cannot be popped from an empty stack. The next item on the stack becomes the new top item.
- `peek()`: Returns a reference to the item on top of a non-empty stack without removing it. Peeking, which cannot be done on an empty stack, does not modify the stack contents.
- `push(item)`: Adds the given `item` to the top of the stack.

(From Necaise, 2011)

Table 1

Stack methods

Stack Method	What It Does
`s.isEmpty()`	Returns **`True`** if **`s`** is empty or **`False`** otherwise.
`s.__len__()`	Same as **`len(s)`**. Returns the number of items in **`s`**.
`s.__str__()`	Same as **`str(s)`**. Returns the string representation of **`s`**.
`s.__iter__()`	Same as **`iter(s)`**, or **`for item in s:`**. Visits each item in **`s`**, from bottom to top.
`s.__contains__(item)`	Same as **`item in s`**. Returns **`True`** if **`item`** is in **`s`** or **`False`** otherwise.
`s1.__add__(s2)`	Same as **`s1 + s2`**. Returns a new stack containing the items in **`s1`** and **`s2`**.
`s.__eq__(anyObject)`	Same as **`s == anyObject`**. Returns **`True`** if **`s`** equals **`anyObject`** or **`False`** otherwise. Two stacks are equal if the items at corresponding positions are equal.
`s.clear()`	Makes **`s`** become empty.
`s.peek()`	Returns the item at the top of **`s`**. *Precondition*: **`s`** must not be empty; raises a **`keyerror`** if the stack is empty.
`s.push(item)`	Adds **`item`** to the top of **`s`**.
`s.pop()`	Removes and returns the item at the top of **`s`**. *Precondition*: **`s`** must not be empty; raises a **`KeyError`** if the stack is empty.

(From Lambert, 2019)

3.2 Stack operations

- Figure 2 describes the stack ADT, while table 1 describes the methods associated with the stack ADT in Python.
- Let us use the stack ADT to solve a simple problem like reversing a list of integers. The process is described by Necaise (2011) as follows:
- The values will be extracted from the user until a negative value is entered, which flags the end of the collection. The values will then be printed in reverse order from how they were entered.
- We could use a simple list for this problem, but a stack is ideal since the values can be pushed onto the stack as they are entered and then popped one at a time to print them in reverse order. This makes powerful use of the stack since we now use the push and pop technique to fill the stack and empty it in reverse (remember LIFO principle?)
- Let us examine the solution program as shared by Necaise (2011).

3.2 Stack operations

```
PROMPT = "Enter an int value (<0 to end):"
# enter a negative number to end the list
myStack = Stack()
value = int(input( PROMPT ))
#value is the name of the variable to be used to identify items for the stack (list)
while value >= 0 :     # this while adds items to the stack
    myStack.push( value )
    value = int(input( PROMPT ))
while not myStack.isEmpty() : # remove items from the list as long as user had entered some items
    value = myStack.pop()
print( value )
#print the list in reverse order, thanks to the LIFO principle
```

3.2 Stack operations

- Suppose we input some sample data, say: 2, 30, 19, 7, -10 (to end the list)
- The values will be pushed into the stack in the following manner:
- 2 (base, since its pushed in first) → 30 → 19 → 7 (top, since its pushed in last)
- When we run the second while statement to pop out the top element of the stack the output will be as follows:
- 7 (top) → 19 → 30 → 2
- Thus we now see the list in reverse order.
- The stack is defined as an interface that can be used by various implementations as will be seen later in this lesson.
- Table 2 shows how the operations listed in table 1 affect a stack named **s**, where the variables a, b, and c refer to items in the stack. The syntactic form <Stack Type> stands for any implementing class. (remember this is an interface)

Table 2

Stack operations

Operation	State of the Stack After the Operation	Value Returned	Comment
`s = <Stack Type>()`			Initially, the stack is empty.
`s.push(a)`	`a`		The stack contains the single item **a**.
`s.push(b)`	`a b`		**b** is the top item.
`s.push(c)`	`a b c`		**c** is the top item.
`s.isEmpty()`	`a b c`	`False`	The stack is not empty.
`len(s)`	`a b c`	`3`	The stack contains three items.
`s.peek()`	`a b c`	`c`	Return the top item without removing it.
`s.pop()`	`a b`	`c`	Remove and return the top item. **b** is now the top item.
`s.pop()`	`a`	`b`	Remove and return the top item. **a** is now the top item.
`s.pop()`		`a`	Remove and return the top item.
`s.isEmpty()`		`True`	The stack is empty.
`s.peek()`		`KeyError`	Peeking at an empty stack raises an exception.
`s.pop()`		`KeyError`	Popping an empty stack raises an exception.
`s.push(d)`	`D`		**d** is the top item.

(From Lambert, 2019)

3.3 Types of stacks

- There are two types of stacks described by (*Applications, Advantages and Disadvantages of Stack*, 2022):
- Register stack: This type of stack is also a memory element present in the memory unit and can handle a small amount of data only. The height of the register stack is always limited as the size of the register stack is very small compared to the memory.
- Memory stack: This type of stack can handle a large amount of memory data. The height of the memory stack is flexible as it occupies a large amount of memory data.

Part 4

Implementation

4.1 Introduction

- Arrays are normally implemented using lists and arrays. Before discussing these implementations Lambert (2019) recommends using a tester program "that shows how you can test them (the linkedlist and linkedarray) immediately. The code in this program exercises all the methods in any stack implementation and gives you an initial sense that they are working as expected."
- The code is shared next.

```
"""
File: teststack.py
Author: Ken Lambert
A tester program for stack implementations.
"""
from arraystack import ArrayStack
from linkedstack import LinkedStack
def test(stackType):
# Test any implementation with the same code
    s = stackType()
    print("Length:", len(s))
    print("Empty:", s.isEmpty())
    print("Push 1-10")
    for i in range(10):
        s.push(i + 1)
    print("Peeking:", s.peek())
    print("Items (bottom to top):", s)
    print("Length:", len(s))
    print("Empty:", s.isEmpty())
    theClone = stackType(s)
    print("Items in clone (bottom to top):",
    theClone)
    theClone.clear()
    print("Length of clone after clear:",
    len(theClone))
    print("Push 11")
    s.push(11)
    print("Popping items (top to bottom):", end =
    " ")
    while not s.isEmpty(): print(s.pop(), end = " ")
    print("\nLength:", len(s))
    print("Empty:", s.isEmpty())
# test(ArrayStack)
test(LinkedStack)
```

4.2 List implementation

- The first decision we have to make when using the list for the Stack ADT is which end of the list to use as the top and which as the base. For the most efficient ordering, we let the end of the list represent the top of the stack and the front represent the base. As the stack grows, items are appended to the end of the list and when items are popped, they are removed from the same end. (Necaise, 2011).
- We notice that this makes a lot of sense since the Python append command adds an item to the end of the list rather than to the beginning. Thus using this reasoning since items are appended to the end of the list they should also be popped from the same end as per the LIFO stack rule.
- Necaise (2011) also provided the Python code that allows for this implementation. This is shown in figure 3.

Figure 3

Liststack implementation

```
1  # Implementation of the Stack ADT using a Python list.
2  class Stack :
3     # Creates an empty stack.
4    def __init__( self ):
5      self._theItems = list()
6
7     # Returns True if the stack is empty or False otherwise.
8    def isEmpty( self ):
9      return len( self ) == 0
10
11    # Returns the number of items in the stack.
12   def __len__ ( self ):
13     return len( self._theItems )
14
15    # Returns the top item on the stack without removing it.
16   def peek( self ):
17     assert not self.isEmpty(), "Cannot peek at an empty stack"
18     return self._theItems[-1]
19
20    # Removes and returns the top item on the stack.
21   def pop( self ):
22     assert not self.isEmpty(), "Cannot pop from an empty stack"
23     return self._theItems.pop()
24
25    # Push an item onto the top of the stack.
26   def push( self, item ):
27     self._theItems.append( item )
```

(From Necaise, 2011)

4.3 Linked list implementation

- Remember the linked list we discussed in lesson 4? Figure 5 reminds you of the structure of a linked list. All the elements are referred to as nodes and every node links to the preceding one and the one after it. Different types of linked lists were examined in that lesson (including the circular linked list); for ease of reference we refer to a simple linked list in this section.
- Figure 6 shows a sample object of the Stack ADT implemented as a linked list. In a linked list "Push operation is implemented by inserting element at the beginning of the list. Pop operation is implemented by deleting the node from the beginning (the header/top node). (Karumanchi, 2017).
- Figure 4 shows the Python implementation of a linked list (Necaise, 2011)

Figure 4

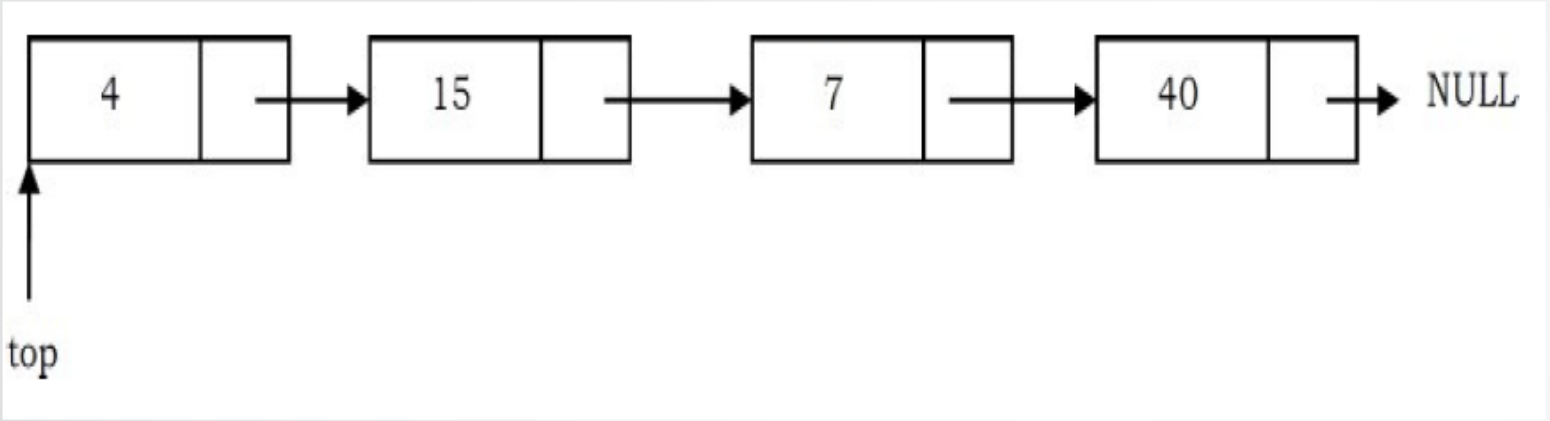
Linked list implementation of stack

```
1  # Implementation of the Stack ADT using a singly linked list.
2  class Stack :
3     # Creates an empty stack.
4    def __init__( self ):
5      self._top = None
6      self._size = 0
7
8     # Returns True if the stack is empty or False otherwise.
9    def isEmpty( self ):
10     return self._top is None
11
12    # Returns the number of items in the stack.
13   def __len__( self ):
14     return self._size
15
16    # Returns the top item on the stack without removing it.
17   def peek( self ):
18     assert not self.isEmpty(), "Cannot peek at an empty stack"
19     return self._top.item
20
21    # Removes and returns the top item on the stack.
22   def pop( self ):
23     assert not self.isEmpty(), "Cannot pop from an empty stack"
24     node = self._top
25     self.top = self._top.next
26     self._size -= 1
27     return node.item
28
29    # Pushes an item onto the top of the stack.
30   def push( self, item ) :
31     self._top = _StackNode( item, self._top )
32     self._size += 1
33
34 # The private storage class for creating stack nodes.
35 class _StackNode :
36   def __init__( self, item, link ) :
37     self.item = item
38     self.next = link
```

(From Necaise, 2011)

Figure 5

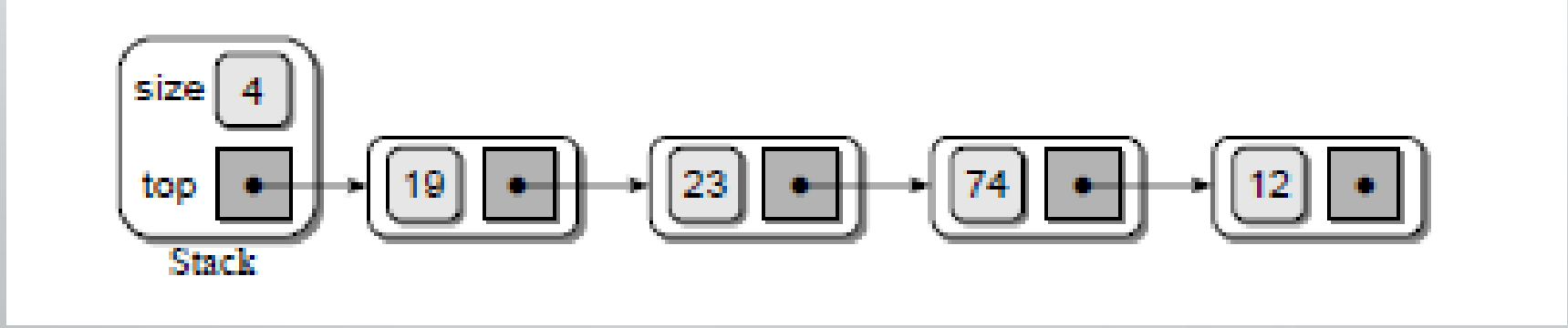
Linked list structure



(From Karumanchi, 2017)

Figure 6

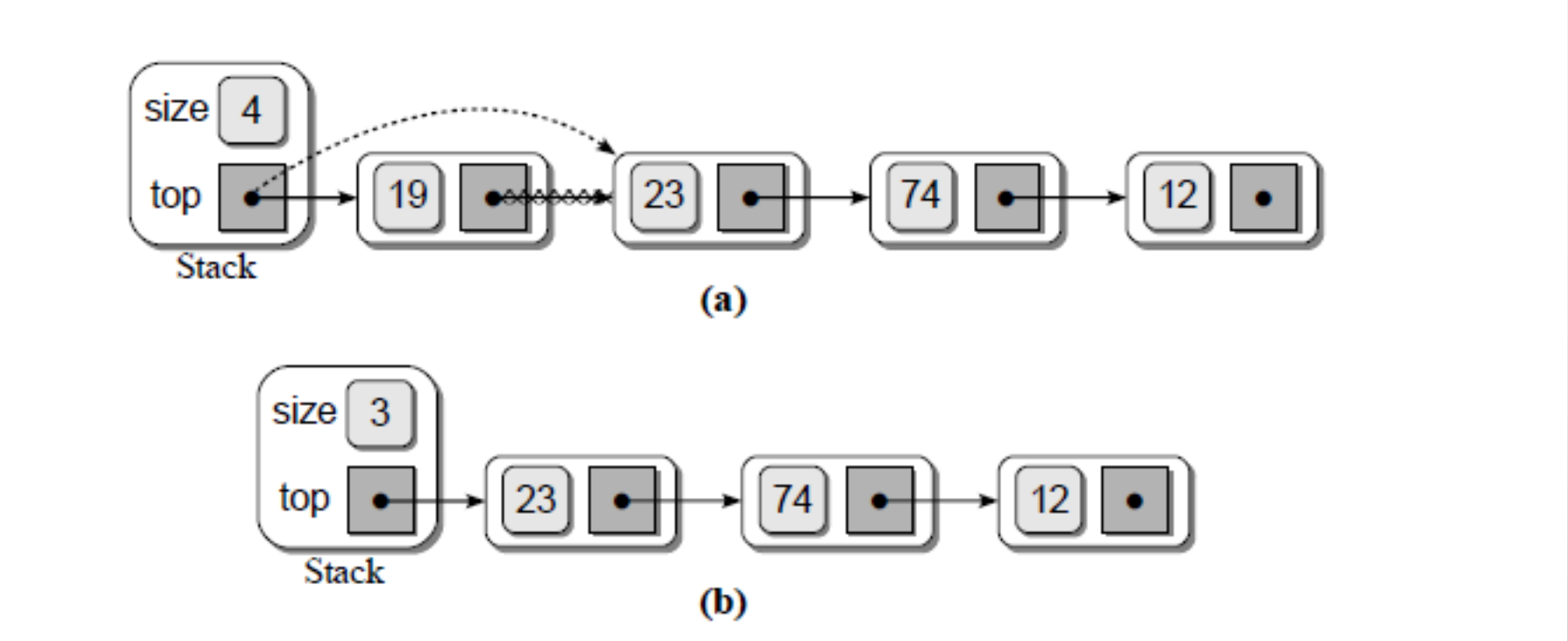
Sample object of the Stack ADT implemented as a linked list



(From Necaise, 2011)

Figure 7

Stack implementation on linked list



(From Necaise, 2011)

4.3 Linked list implementation

- Figure 7 demonstrates the operation of the pop method of the stack implementation on a linked list.
- The pop() method always removes the first node in the list. This operation is illustrated in Figure 7.4(a). This is easy to implement and does not require a search to find the node containing a specific item. The result of the linked list after popping the top item from the stack is illustrated in Figure 7.4(b). (Necaise, 2011)
- In terms of performance the complexity of the different operations on a linked list is provided by Karumanchi (2017) in table 3. we assume that n is the number of elements in the stack.

Table 3

Performance of operations on a linked list

Space Complexity (for *n* push operations)	O(*n*)
Time Complexity of CreateStack()	O(1)
Time Complexity of Push()	O(1) (Average)
Time Complexity of Pop()	O(1)
Time Complexity of Top()	O(1)
Time Complexity of IsEmptyStack()	O(1)
Time Complexity of DeleteStack()	O(*n*)

(From Karumanchi, 2017)

4.4 Array linked implementation

- Let us consider an array implementation of a stack containing 4 elements as described by Lambert (2019).. The visual representation is captured by figure 8.
- Lambert (2019) describes it thus: "Th(is) first implementation is built around an array called self.items and an integer called self.size. Initially, the array has a default capacity of 10 positions, and self.size. equals 0. The top item, if there is one, will always be at location self.size – 1. To push an item onto the stack, store the item at the location self.items[len(self)] and increment self.size. To pop the stack, return self.items[len(self) - 1] and decrement self.size.
- Figure 8 shows how self.items and self.size appear when four items are on the stack."
- The array, as shown, has a current capacity of 10 positions. How do you avoid the problem of stack overflow?

4.4 Array linked implementation

- As discussed in Lesson 4, "Arrays and Linked Structures," you create a new array when the existing array is about to overflow or when it becomes underutilized. Following the description in Lesson 4, you double the array's capacity after push fills it and halve it when pop leaves it three-quarters empty.
- The array-based stack implementation uses the Array class developed in Lesson 4 and is quite similar to the ArrayBag class developed in Lesson 6. Like ArrayBag, ArrayStack is subclassed under an abstract class. In this case, the parent class is called AbstractStack. The only operations you need to provide in ArrayStack are __init__, clear, push, pop, peek, and __iter__.
- Part of the code required for this implementation is provided by Lambert (2019) in the following slide.

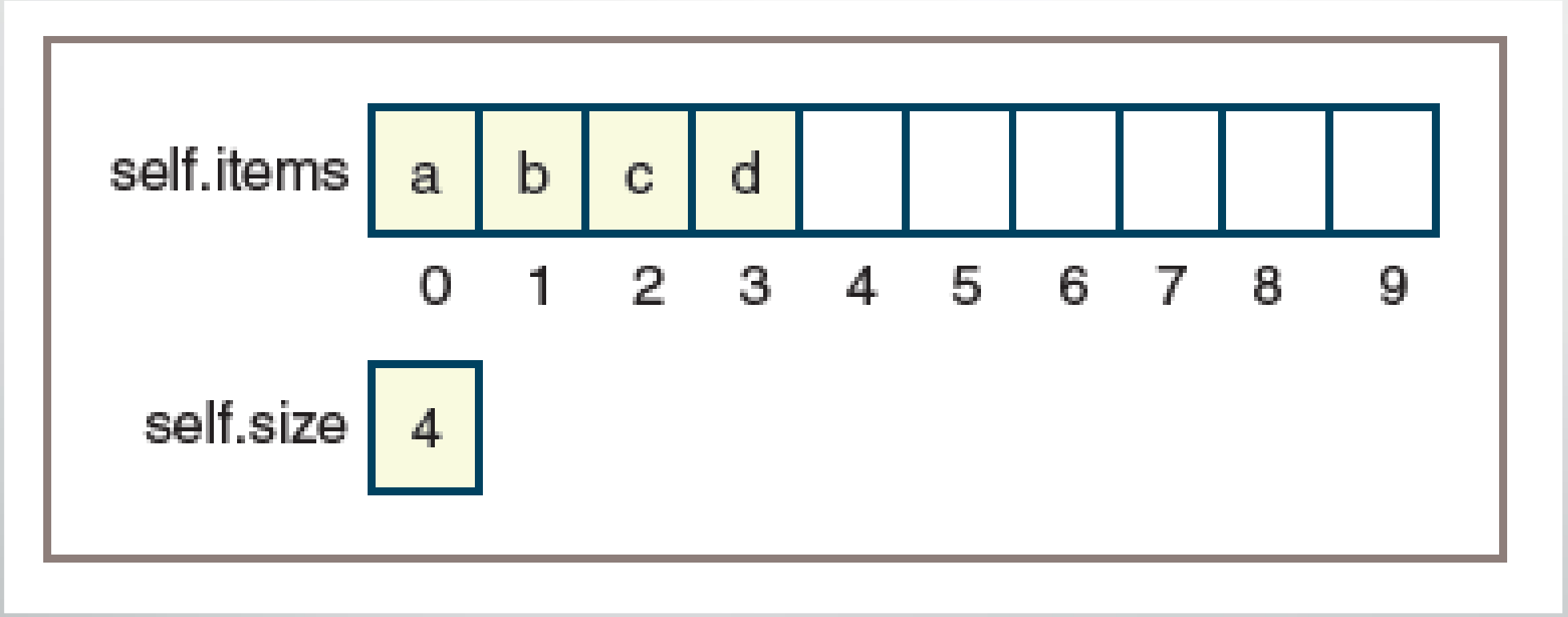
```
"""
File: arraystack.py
"""
from arrays import Array
from abstractstack import AbstractStack
class ArrayStack(AbstractStack):
"""An array-based stack implementation."""
DEFAULT_CAPACITY = 10 # For all array stacks
def __init__(self, sourceCollection = None):
"""Sets the initial state of self, which includes the contents of sourceCollection, if it's present."""
self.items =Array(ArrayStack.DEFAULT_CAPACITY)
AbstractStack.__init__(self, sourceCollection)
# Accessors
def __iter__(self):
"""Supports iteration over a view of self.
Visits items from bottom to top of stack."""
    cursor = 0
    while cursor < len(self):
        yield self.items[cursor]
    cursor += 1
Def peek(self):
"""Returns the item at top of the stack.
Precondition: the stack is not empty.
Raises KeyError if the stack is empty."""
```

```
# Check precondition here
return self.items[len(self) - 1]
# Mutators
def clear(self):
"""Makes self become empty."""
    self.size = 0
    self.items =
Array(ArrayStack.DEFAULT_CAPACITY)
def push(self, item):
"""Inserts item at top of the stack."""
# Resize array here if necessary
    self.items[len(self)] = item
    self.size += 1
def pop(self):
```

- """Removes and returns the item at top of the stack.
- Precondition: the stack is not empty."""
- Raises KeyError if the stack is empty.
- Postcondition: the top item is removed from the stack."""
- # Check precondition here
- oldItem = self.items[len(self) - 1]
- self.size -= 1
- # Resize the array here if necessary
- return oldItem

Figure 8

Array representation of stack with four elements



(From Lambert, 2019)

Part 5

Application

5.1 Applications

- (Dham, 2023) provides a list of application areas where stacks are used in data structures:”
- **Function Calls-** The state of the program is placed into the Stack when a function is invoked. The preceding function's execution is continued after the process returns by popping the state off the Stack.
- **Backtracking-** Stacks can be used for backtracking or to verify if an expression's parentheses match. Stacks are used by the backtracking method to maintain track of the stages of the solution process. The old state is removed from the Stack when the algorithm goes backwards after pushing the current state onto it.
- **Undo/Redo Operations-** Many apps' undo-redo functionality employs stacks to remember the prior operations. A new action is added to the Stack each time it is completed. The top member of the Stack is popped to undo the action, and the original procedure is then carried out.

5.1 Applications

- **Web browser history-** Stacks are used by web browsers to record the websites you visit. When you click the back button, the previous URL is removed from the Stack and is added to the Stack each time you visit a new page.
- **Reverse the Data-** We must reorganize the data so that the first and final items are switched; the second and second-last elements are exchanged, and so on for all subsequent elements if we want to reverse a particular collection of data. For example: If we have string codingNinja, then on reversing, it will become ajniNgnidoc
- **Parenthesis checking-** To determine if brackets are balanced or not, a stack data structure is utilized. An opening parenthesis is popped off the Stack as a closing parenthesis is added onto it. The brackets are balanced if the Stack is empty at the conclusion of the expression.
- **Expression Evaluation-** Expressions written in infix, postfix, and prefix notations are evaluated using a stack data structure. The Stack is used to hold operators and operands, and the top pieces of the Stack are used to carry out operations."

5.1 Applications

- Let us describe the three terms named in our last application in the previous slide (44); definitions provided by (Chakraborty, 2020):
- Infix notation : Infix notations are normal notations, that are used by us while write different mathematical expressions.
- Prefix notation: In this notation, operator is **prefixed** to operands, i.e. operator is written ahead of operands. For example, **+ab**. This is equivalent to its infix notation **a + b**. Prefix notation is also known as **Polish Notation**.
- Postfix notation: This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfixed** to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a + b**.
- Examples of this notation for expressions is displayed in table 4.

Table 4

Infix, postfix, and prefix notation

Expression No	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) * c	* + a b c	a b + c *
3	a * (b + c)	* a + b c	a b c + *
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) * (c + d)	* + a b + c d	a b + c d + *
6	((a + b) * c) - d	- * + a b c d	a b + c * d -

(From Chakraborty, 2020)

5.2 Applications in real life

- (*Applications, Advantages and Disadvantages of Stack*, 2022) also provides some examples of applications of stacks in real life:"
- CD/DVD stand.
- Stack of books in a book shop.
- Call center systems.
- Undo and Redo mechanism in text editors.
- The history of a web browser is stored in the form of a stack.
- Call logs, E-mails, and Google photos in any gallery are also stored in form of a stack.
- YouTube downloads and Notifications are also shown in LIFO format(the latest appears first).
- Allocation of memory by an operating system while executing a process."

Part 6

Advantages and disadvantages

6.1 Advantages

- (*Applications, Advantages and Disadvantages of Stack*, 2022) also provides the following advantages of stacks:"
- **Easy implementation:** Stack data structure is easy to implement using arrays or linked lists, and its operations are simple to understand and implement.
- **Efficient memory utilization**: Stack uses a contiguous block of memory, making it more efficient in memory utilization as compared to other data structures.
- **Fast access time:** Stack data structure provides fast access time for adding and removing elements as the elements are added and removed from the top of the stack.
- **Helps in function calls:** Stack data structure is used to store function calls and their states, which helps in the efficient implementation of recursive function calls.

6.1 Advantages

- **Supports backtracking:** Stack data structure supports backtracking algorithms, which are used in problem-solving to explore all possible solutions by storing the previous states.
- **Used in Compiler Design:** Stack data structure is used in compiler design for parsing and syntax analysis of programming languages.
- **Enables undo/redo operations**: Stack data structure is used to enable undo and redo operations in various applications like text editors, graphic design tools, and software development environments."

6.2 Disadvantages

- (*Applications, Advantages and Disadvantages of Stack*, 2022) also provides the following disadvantages of stacks:"
- **Limited capacity:** Stack data structure has a limited capacity as it can only hold a fixed number of elements. If the stack becomes full, adding new elements may result in stack overflow, leading to the loss of data.
- **No random access:** Stack data structure does not allow for random access to its elements, and it only allows for adding and removing elements from the top of the stack. To access an element in the middle of the stack, all the elements above it must be removed.
- **Memory management:** Stack data structure uses a contiguous block of memory, which can result in memory fragmentation if elements are added and removed frequently.

6.2 Disadvantages

- **Not suitable for certain applications:** Stack data structure is not suitable for applications that require accessing elements in the middle of the stack, like searching or sorting algorithms.
- **Stack overflow and underflow**: Stack data structure can result in stack overflow if too many elements are pushed onto the stack, and it can result in stack underflow if too many elements are popped from the stack.
- **Recursive function calls limitations:** While stack data structure supports recursive function calls, too many recursive function calls can lead to stack overflow, resulting in the termination of the program."

Summary

- Stacks are linear collections in which access is completely restricted to just one end, called the **top**. Conversely, the bottom of the stack is also known as its base.
- The operations on a stack are push, pop and peek (to see the element at the top of the stack).
- The two types of stacks are memory stacks and register stacks.
- Stacks use the LIFO (Last In, First Out) method where the element at the bottom never gets to be used unless there's no element on top of it.
- Stacks are implemented in arrays, lists and linked lists.
- Stacks are used in real life in areas such as call center systems, undo and Redo mechanism in text editors; the history of a web browser is stored in the form of a stack.

References

- *Applications, Advantages and Disadvantages of Stack*. (2022, May 16). GeeksforGeeks; GeeksforGeeks. https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-stack/
- Chakraborty, A. (2020, August 11). *Prefix and Postfix Expressions in Data Structure*. Www.tutorialspoint.com. https://www.tutorialspoint.com/prefix-and-postfix-expressions-in-data-structure
- Dham, M. (2023, September 23). *Coding Ninjas Studio*. Www.codingninjas.com. https://www.codingninjas.com/studio/library/application-of-stack
- Galal, S. (2023, September 22). *Africa: adult literacy by region 2021*. Statista. https://www.statista.com/statistics/1233204/adult-literacy-rate-in-africa-by-region/#:~:text=In%202021%2C%2067.4%20percent%20of
- Lambert, K. (2019). *Fundamentals of Python: Data Structures*. Cengage Learning.

References

- Narasimha Karumanchi. (2017). *Data structures and algorithms made easy : to all my readers : concepts, problems, interview questions*. Careermonk Publications.
- Necaise, R. D. (2011). *Data structures and algorithms using Python*. John Wiley And Sons.
- Prasanna. (2021, August 17). *Advantages and Disadvantages of Stack | List of All Advantages and Disadvantages of Stack and Queues*. A plus Topper. https://www.aplustopper.com/advantages-and-disadvantages-of-stack/