CS 304 Notes

Data Structures & Algorithms

Jaeden Bardati

Last modified February 4, 2022

1 Python and Object-Oriented Programming

Python is a *high-level programming language* which makes it useful when studying data structures and algorithms.

Python is an *interpreted* language, where *source code* (also referred to as *scripts*), in the form of files with the *.py* suffix, are run by an *interpreter*. It is common to use an integrated development environment (IDE) to aid in displaying and editing Python code. IDEs for Python include the built-in IDLE, PyCharm, Spyder, and others.

1.1 Objects in Python

Python is an *object oriented* language where **classes** are the basis of all data types. Examples of data types in Python include int, float, str.

An **assignment statement** assigns an **identifier** (or name) to an object. Identifiers are associated with a *memory address* and are similar to a pointer in languages such as C++ or Java. For example, the statement

```
1    temp = 98.6
```

associates the identifier "temp" to the float value 98.6.

Identifiers are **case-sensitive**. Namely, a variable named "temp" is different than one named "Temp".

Python is a **dynamically typed** language, unlike C++ or Java. That is to say, when making the association of an identifier in Python, the data type is not explicitly declared. In the code above, the data type is determined automatically by the interpreter to be a float.

It is possible to establish an **alias** by assigning a second identifier to an object as follows

```
1    temperature = temp
```

After an alias is made, either name can be used to refer to the object.

The process of creating a class is called **instantiation**. To do this, we call the **constructor** of a class. If we have defined the class called Animal then we would do this by

```
1    a = Animal()
```

Note that we can also pass parameters to the Animal constructor.

See Chapters 1 and 2 in the textbook for more.

2 Design and Analysis of Algorithms

2.1 Data structures vs. Abstract Data Types

There are many **abstract data types**: list, set, queue, stacks, dictionary, etc.

Data structures are the implementation of abstract data types. Examples include: arrays, trees, hash tables, etc.

Our goal is to find the best data structure to use. The best data structure is one that minimizes the **time complexity**.

For example, if we have a list $L = [1, 2, 3]$ and we want to insert an element. If we use an array abstract type, then it is easy to write to insert an element at the end of the list, but it is more complicated to insert it at the beginning of the list (must shift all other elements over 1).

Another example is the binary search algorithm (e.g. looking for a number in a phone book) has the time complexity of $\theta(log_2 n)$.

2.2 Insertion Sort

The insertion sort algorithm written in pseudocode in Algorithm 1.

We can check the code using a trace for A = [3, 2, 1]. The variables as they change are:

1. $j = 2$
2. key $= 2$
3. $i = 1$

4. $A = [3, 3, 1]$

5. $i = 0$

6. $A = [2, 3, 1]$

7. $j = 3$

8. key $= 1$

9. $i = 2$

10. $A = [2, 3, 3]$

11. $i = 1$

12. $A = [2, 2, 3]$

13. $i = 0$

14. $A = [1, 2, 3]$

15. $j = 4$

2.2.1 Algorithm efficiency

WE can go line by line through the program and associate an arbitrary time cost as well as a number of times that the line will run. We can represent this in Table 1.

The time that the algorithm takes is

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n}(t_j-1) + c_6 \sum_{j=2}^{n}(t_j-1) + c_7(n-1)$$

Best case: $t_j = 0$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_8(n-1) = (c_1+c_2+c_3+c_4+c_8)n - (c_2+c_3+c_4+c_8)$$

Algorithm 1 InsertionSort(A)

1: **for** $(j = 2$ to length$(A))$ **do**
2:     **set** key $= A[j]$
3:     // insert $A[j]$ into the
4:     // sorted sequence $A[1..j-1]$
5:     **set** $i = j - 1$
6:     **while** $(i > 0$ and $A[i] >$key$)$ **do**
7:         **set** $A[i+1] = A[i]$
8:         **set** $i = i - 1$
9:         **set** $A[i+1] =$ key
10: **stop**

3

Table 1: Insertion Sort Algorithm Cost

Line	Cost	Times
1	c_1	n
2	c_2	$n-1$
3	0	0
4	0	0
5	c_3	$n-1$
6	c_4	$\sum_{j=2}^{n} t_j$
7	c_5	$\sum_{j=2}^{n}(t_j - 1)$
8	c_6	$\sum_{j=2}^{n}(t_j - 1)$
9	c_7	$n-1$

where we let t_j be the number of times that the while loop is executed for a given j.

This can be written in the form $an+b$. Namely, the best case is a linear function with n.

Worse case: $t_j = j$ for $j = 2, 3, ...n$, therefore,

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

And,

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

Thus,

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1)$$

$$= (\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2})n^2 + (c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

This can be written in the form $an^2 + bn + c$. Namely, the worst case is a quadratic function with n.

2.3   Asymptotic Analysis

We say insertion sort has a worst-case running time of $\Theta(n^2)$. All we care about is the order of n as n asymptotically increases without bound. For example,

$\frac{n^3}{1000} - 100n^2 - 100n + 3$ has a time complexity of $\Theta(n^2)$.

Definition: For a given function $g(n)$ we denote by $\Theta(g(n))$ the set of functions

$$\Theta(g(n)) \equiv \{f(n) : \exists \text{ positive constants } c_1, c_2, n_0 \text{ s.t. } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \, \forall \, n \geq n_0\}$$

For all $n \geq n_0$ f(n) is equal to g(n) within a constant factor. Note that we often abuse the notation and write that $f(n) = \Theta(g(n))$ rather than $f(n) \in \Theta(g(n))$ as one might expect.

For example, let us justify that $\frac{n^2}{2} - 3n = \Theta(n^2)$. We must determine positive constants c_1, c_2, n_0 s.t. $c_1 n^2 \leq \frac{n^2}{2} - 3n \leq c_2 n^2 \, \forall \, n \geq n_0$.

Dividing by n^2, $c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$, where we choose $c_2 = \frac{1}{2}$, $c_1 = \frac{1}{14}$ for $n_0 \geq 7$.

Definition: For a given function $g(n)$, we denote $O(g(n))$ as the set of functions

$$O(g(b)) \equiv \{f(n) : \exists \text{ positive constants } c \text{ and } n_0 \text{ s.t. } 0 \leq f(n) \leq cg(n) \, \forall \, n \geq n_0\}$$

We call this "big O" notation. See Figure 3.5 in the textbook for a visualization. It is important to note that if $f(n) = \Theta(g(n))$, then necessarily $f(n) = O(g(n))$. Also note that the $\Omega(g(n))$ notation indicates a minimum asymptotic trend.

3 Array-Based Sequences

3.1 Sequence Types in Python

The main sequence types in Python are the **list**, **tuple** and **string**. These sequence types can be **referential** or **compact**, and **mutable** or **immutable**. Referential are distinguished from compact arrays later on in these notes. Mutable arrays are array that can be changes in some way (appended to, values changed, values inserted, etc.), whereas immutable arrays cannot be changed. The following are the properties of the sequence types.

- lists are *referential* and *mutable*

- tuples are *referential* and *immutable*

- strings are *compact* and *immutable*

We can do a variety of operations on these sequence types. Some of them include "len(data)", which finds the length of the array data, or "data1 == data2", which checks if the arrays data1 and data2 are equal. See Tables 5.3 and 5.4

from the textbook for a comprehensive list.

The non-mutating behaviors are

- len(data): Finds length of array
- data[j]: Returns the jth index
- data.cout(value): Number of times value is in data
- data.index(value): The index of the value in data
- Value in data: Same as above
- data1 == data2: Checks if the arrays are equal
- d2 = data[j:k]: Creates a new array which is a slice from index j (inclusively) to k (exclusively)
- d2 = data1 + data2: Creates a new array which is the concatenation of the arrays
- d2 = c*data: Creates a new array which is c copies of data

and the mutating behaviors are

- data[j] = val: Change the value at index j to val
- data.append(val): Append to the end of the array
- data.insert(k, val): Insert val in an element at index k (increases size)
- data.pop(): Removes (and returns) the last element of the array
- data.pop(k): Removes (and returns) the kth element of the array
- del data[k]: Removes the kth element of the array
- data.remove(val): Searches the array for val and removes it
- data1.extend(data2): Appends the array data2 to the array data1
- data.reverse(): Reverses the order of the array
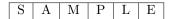- data.sort(): Sorts the array in increasing order

An implementation of data.insert in our previous array class in Python is

```python
def insert (self, k, value):
    if self._n == self._capacity:
        self._resize(2*self._capacity)
    for j in range(self._n, k, -1):
        self._A[j] = self._A[j-1]
    self._A[k] = val
    self._n += 1
```

3.2   Low-Level Arrays

3.2.1   What is an Array?

First, we must define RAM. Random-access memory (RAM) is memory that can be randomly-accessed. Randomly-accessed means that any random element can be accessed just as easily as any other. We can visualize this in a table as follows

S	A	M	P	L	E

An array is a group of related values sorted one after another in a contiguous portion of RAM. To access an array, we refer to the memory address that the array begins at and add the index. Namely,

```
A[i] = start + index
```

3.2.2   Note for Python Arrays

Python arrays use pointers to refer to an array in memory. Therefore, we need to explicitly copy an array to make sure it is copied. In the following example, an alias is made and so the sorted array [1, 2, 3] would be outputted.

```
x = [3, 2, 1]
y = x
y.sort()
print(x)
```

3.2.3   Referential Arrays

Referential Arrays are arrays that store memory addresses (object references). For example, an array of strings is a referential array because strings are stored as pointers to another location in memory (to a compact array).

Let us say that we have an array of primes:

```
primes = [2, 3, 5, 7, 11, 13, 17, 19]
```

Then, we can create a sub-array with

```
temp = primes[3:6]
```

The array temp is a referential array to a part of the array primes (or rather, to the objects to which the elements reference). See Figure 5.5 in the textbook. If then we set

```
1   temp[2] = 15
```

Then the temp array at index 15 is now pointing to another location in memory with a new entry 15, but the rest of the array will still point to their original addresses. See Figure 5.6 in the textbook.

If we have an array,

```
1   counters = [0]*8
```

This will create an array of *pointers* to the same 0 object. See Figure 5.7 in the textbook. Adding 1 to one of the elements will create a new 1 object and make that element point to it (see Figure 5.8).

3.2.4   Compact Arrays

Compact arrays are when the elements are stored directly in the array (not an array of pointers). An example of this is a string, with is an array of characters (not pointers of characters). Compact arrays have advantages over referential arrays. Namely, they are good because they

1 Have less memory usage: No overhead devoted to storage of memory references
2 Have higher performance:
  2a) direct access
  2b) principle of locality (if we access a certain location in memory, we are likely to access the array again nearby)

To use a compact array, we can use the array module. To create one, we must pass a list of a certain type, along with the type code for that type. For a list of signed integers, we use the type code 'i'. For example, we could write

```
1   primes = array('i', [2, 3, 5, 7, 11, 13, 17, 19])
```

3.3   Dynamic Arrays and Amortization

Dynamic Arrays are arrays that can dynamically increase in time. To do this, dynamic arrays reserve a certain amount of space in memory for future appends until it runs out of reserved space, in which case it rewrites the new array along with more reserved space. Python's list class is a dynamic array. We can explore the relation with the code

```
1  for k in range(10):
2      a = len(data)
3      b = sys.getsizeof(data)
4      print('Length: {0:3d}; Size in bytes: {1:4d}'.format(a, b))
5      data.append(None)
```

We can see that the size in memory increases exponentially. This is because
the number of spots added is proportional to the size of the array at a time in
Python lists. The amount of space that is dynamically added to the array at a
time increases as the list grows (it will not always be 4).

Let us now implement a dynamic array ourselves

```
1   import ctypes
2
3   class DynamicArray:
4       """A Dynamic array class akin to a simplified python list"""
5
6       def __init__(self):
7           """Create an empty array"""
8           self._n = 0
9           self._capacity = 1
10          self._A = self._make_array(self._capacity)
11
12      def __len__(self):
13          """Return number of elements stored in array"""
14          return self._n
15
16      def __getitem__(self, k):
17          """Return item at index k"""
18          if not 0 <= k < self._n:
19              raise IndexError("Invalid index")
20          return self._A[k]
21
22      def _resize(self, c)
23          """Resize internal array to capacity c"""
24          B = self._make_array(c)
25          for k in range(self._n):
26              B[k] = self._A[k]
27          self._A = B
28          self._capacity = c
29
30      def _make_array(self, c):
31          """Return a new array with capacity c"""
32          return (C * ctypes.py_object)()
33
34      def append(self, obj):
35          """adds object to end of the array"""
36          if self._n == self._capacity:
37              self._resize(2*self._capacity)
38              self._A[self._n] = obj
39              self._n += 1
40
```

We can test it out using:

9

```
1   arr = DynamicArray()
2   arr.append(1)
```

We now would like to analyze the efficiency.

Reminder: $O(n)$ means *at most* n efficent and $\Omega(n)$ means *at least* n. Here we have $\Omega(n)$ efficiency.

Using amortization, we can show that an append is $O(1)$, and n appends are $O(n)$. Namely, a single append does not depend on the size of the array.

Proposition 5.1: Let S be a sequence implemented using a dynamic array with initial capacity 1. Using the strategy of doubling the array size when full, the total time to perform n operations is $O(n)$.

Justification: Assume 1 cyber-dollar pays for each append where we do not expand array. Assume growing array from size k to $2k$ requires k cyber-dollars. We charge each append 3 cyber-dollars. An overflow occurs when array S has 2^i elements for some integer $i \geq 0$ and the size of the array representing S is 2^i. Namely, doubling the size of the array will cost 2^i cyber-dollars. These cyber-dollars can be found in cells 2^{i-1} through $2^i - 1$.

Proposition 5.2: Performing a series of n append operations using a fixed increment with each resize takes $\Omega(n^2)$.

Justification: Let $c \geq 0$ represent a fixed increment in capacity used for each resize event. During the series of append operations, time will be spent initializing arrays of size c, $2c$, $3c$, ..., mc for $m = \frac{n}{2}$. So the overall time is proportional to $c + 2c + 3c + ... + mc = \sum_{i=1}^{m} ci = c \sum_{i=1}^{m} i = c\frac{m(m+1)}{2} \geq \frac{\frac{n}{2}\left(\frac{n}{2}-1\right)}{2}$. This is $\Omega(n^2)$.

3.4   Multidimensional Arrays

If we do

```
1   l = [[9]*10]*5
2   l[0][3] = 5
```

It will change the values of throughout the array. Instead, the correct method to instantiate a multidimensional array is

```
1   l = [[9]* 10 for j in range(5)]
2   l[0][3] = 5
```

This will work by ensuring that they do not all point to the same value.

3.5   Numpy Arrays

For big data, we can use the NumPy module. We can make an nd-array with

```
1    z = np.zeros((10, 10)) # will create a 10x10 array of 0s
```

There are many different methods that NumPy has which are highly optimized in precomplied C-libraries.

4   Stacks

The stack is one of the most important and most widely used data structures. Stacks are inserted and removed using the **last in - first out (LIFO)** principle.

Formally, a stack is an ADT (abstract data type) that supports two main methods:

- S.push(e): Add e to top of stack
- S.pop(): Remove and return top element

We can also make it support the following methods:

- S.top(): Returns a reference to the top element of the stack
- S.is_empty(): Returns true if empty, otherwise false
- len(S): Returns the length of the stack

The following table shows an example of these methods in use.

Operation	Return Value	Stack contents
S.push(5)	None	[5]
S.push(3)	None	[5, 3]
len(S)	2	[5, 3]
S.pop()	3	[5]
S.is_empty()	False	[5]
S.pop()	5	[ ]

We will use the **adaptor pattern** to construct a stack from the list class using Python.

4.1 Stack Implementation in Python

```python
class ArrayStack:
    """LIFO Stack implementation using Python list as underlying
    shape."""
    def __init__(self):
        """Create an empty Stack """
        self._data = []

    def __len__(self):
        """Returns length of stack"""
        return len(self._data)

    def is_empty(self):
        """Return True if stack is empty"""
        return len(self._data) == 0

    def push(self, e):
        """Adds e to the top of the stack"""
        self._data.append(e)

    def top(self):
        """Returns the top element"""
        if self.is_empty():
            raise EmptyException("Stack is empty")
        return self._data[-1]

    def pop(self):
        """Removes and returns the top element."""
        if self.is_empty():
            raise EmptyException("Stack is empty")
        return self._data.pop()
```

4.2 Complexity Analysis

The following table shows the time complexity of each of these methods.

Operation	Running Time
S.push(e)	$O(1)$*
S.pop()	$O(1)$*
S.top()	$O(1)$
S.is_empty()	$O(1)$
len(S)	$O(1)$

* means it is amortized.

An example of using a stack in Python is the following.

```python
a = [1, 2, 3], S = ArrayStack()

for i in range(len(a)):
  S.push(a[i])
# S = [1, 2, 3]

while not S.is_empty():
  a[i] = S.pop()
# a = [1, 2, 3]
```

4.3   Matching Parentheses using a Stack

In pseudocode,

1. Initialize empty stack

2. Read characters until end (call current character char)

3. If char == opening parentheses, push the char to stack

4. Else If char == closing parentheses and stack is empty, produce error

5. Otherwise, pop stack and if popped symbol does not match char, produce error

6. If at EOF (end of file) and stack is not empty, produce error

For example, '(())' would produce no errors. but '(()' would.

The Python code for this is

```python
def is_match(expr):
  """Returns true if all delimiters match"""
  lefty = '([{'
  righty = ')]}'
  S = ArrayStack()
  for c in expr:
    if c in lefty:
      S.push(c)
    elif c in righty:
      if S.is_empty():
        return False
      if righty.index(c) != lefty.index(S.pop()):
        return False
  return S.is_empty()
```

4.4   Polish Notation

In polish notation, the expression

$$1 + 2 = 3$$

is written as

$$1\ 2\ + \ = 3$$

The infix expression

$$6 * (5 + (2 + 3) * 8 + 3) = 288$$

changes its value with a change in the placement of the brackets. However, its polish notation (postfix) equivalent

$$6\ 5\ 2\ 3 + 8 * + 3 + * = 288$$

does not need that delineation.

To evaluate polish notation we

- Read expression from left to right

- If operand, push to stack

- If operator, pop two values off stack, perform operation and push back on stack

That is how to evaluate it, but how do we convert between the forms.

4.4.1   Algorithm for converting infix to postfix

The steps of the algorithm are

1. If operand, place in output

2. When we encounter an operator, place on stack, also stack left parentheses

3. If we encounter a right parentheses, pop the stack and write symbols until we encounter a left parentheses

4. If we see any other symbol (such as +, *, or '('), we pop entries from stack until we find an entry of lower priority

5. When popping is done, push operator to stack

6. If at end, pop entire stack and write it to the output

We can check this with the example

Infix: $a + b * c + (d * e + f) * g$
Postfix: $abc * + de * f + g * +$

5   Queues

Queues are **first-in first-out** (FIFO). For example, waiting in a grocery line is a queue.

The basic methods of the queue abstract data class are

- Q.enqueue(e): Adds e to the back of the queue.

- Q.dequeue(): Removes and returns the first element of queue.

- Q.first(): Returns the first element

- Q.is_empty()

- len(Q)

The following table shows an example of these methods in use.

Operation	Return Value	Q
Q.enqueue(5)	None	[5]
Q.dequeue(3)	None	[5, 3]
len(Q)	2	[5, 3]
Q.dequeue()	5	[3]
Q.is_empty()	False	[5]
Q.dequeue()	3	[ ]

We can write the queue in terms of the built-in Python list. However doing this as it stands give a dequeue time of $O(n)$, because we would need to swap all the elements down to fill the space.

To improve this, we could try setting the first element to None and then have a pointer that moves forward at each dequeue. But this would waste space with a lot of Nones at the beginning of the list. So, we must use circular arrays to avoid this increase in size each dequeue.
A **circular array** uses the modular arithmetic. In python, this is done using the modulus symbol %.

```
n = a // b      # integer division
m = a % b       # modulus
# m = a - b*n is the equivalent expression for m
```

We can get the location to insert the next enqueue using

```
index = (f + size(Q)) % len(Q)
```

where f is the pointer location, size(Q) is an arbituary size and len(Q) is the length of the Q.

We can then update the pointer with

```
1    f = (f + 1) % len(Q)
```

An implementation of a queue in Python is

```
1    class ArrayQueue:
2      """FIFO Queue using Python List"""
3      def __init__(self):
4        self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
5        self._size = 0
6        self._front = 0
7
8      DEFAULT_CAPACITY = 10
9
10     def __len__(self):
11       return self._size
12
13     def is_empty(self):
14       return self._size == 0
15
16     def first(self):
17       if self.is_empty():
18         raise Exception("List is empty")
19       return self._data[self._front]
20
21     def dequeue(self):
22       if self.is_empty():
23         raise Exception("List is empty")
24       answer = self._data[self._front]
25
26       # circular array implementation
27       self._data[self._front] = None     # to help garbage collect
28       self.front = (self._front + 1)% len(self._data)
29
30       self._size -= 1
31
32       if 0 < self._size < len(self._data)//4:
33         # if 25 % of capacity is used, resize to make smaller
34         self._resize(len(self._data) // 2)
35       return answer
36
37     def enqueue(self, e):
38       if self._size == len(self._data):
39         self._resize(2*len(self._data))
40       avail = (self._front + self._size) % len(self._data)
41       self._data[avail] = e
42       self._size += 1
43
44     def _resize(self, cap):
45       old = self._data
46       self._data = [None] * cap
47       walk = self._front
48       for k in range(self._size):
```

16

```
49            self._data[k] = old[walk]
50            walk = (walk + 1) % len(old)
51        self._front = 0
```

5.1 Complexity Analysis

Operation	Running Time
Q.enqueue(e)	$O(1)$*
Q.dequeue()	$O(1)$*
Q.first()	$O(1)$
Q.is_empty()	$O(1)$
len(Q)	$O(1)$

* means it is amortized.

6 Linked List

A linked list is an alternative to an array. There are singly and doubly linked lists.

6.1 Singly Linked List

A singly linked list is a collection of nodes that form a linear sequence.

Each node stores

- A reference to an object
- A reference to the next node

For example, a singly linked list of airports could have a node that points to a flight in Montreal and also points to another flight in Toronto, which also points to Vancouver.

The first node of a linked list is called the **head** and the last node of a linked list is called the **tail**.

Random access in a linked list is quite slow, however, increasing the size is easier, since it is not needed to shuffle all the elements over as you do in an array.

To add to the front of the list, we could use the psudeocode:

```
1   def add_first(L, e):
2     newest = Node(e)
3     newest.next = L.head
4     L.head = newest
5     L.size = L.size + 1
```

We can also add to the tail of the list in a similar way.

```
1   def add_last(L, e):
2     newest = Node(e)
3     newest.next = None
4     L.tail.next = newest
5     L.tail = newest
6     L.size = L.size + 1
```

To remove the first node, we simply need to set the head to the next one and decrement the size. To remove the tail, we need to set the node before it to None, but that requires iterating through the linked list which is $O(n)$.

We can implement a stack using a linked list as the underlying structure.

```
1   class LinkedStack:
2     """LIFO Stack implementation using a Linked List"""
3
4     class _Node:
5       __slots__ = '_element', '_next'   # optimization for the
        store of these two fields of the class node
6
7       def __init__(self, element, next):
8         self._element = element
9         self._next = next
10
11    def __init__(self):
12      self._head = None
13      self._size = 0
14
15    def __len__(self):
16      return self._size
17
18    def is_empty(self):
19      return self._size == 0
20
21    def push(self, e):
22      self._head = self._Node(e, self._head)
23      self._size += 1
24
25    def top(self):
26      if self.is_empty():
27        raise Exception("Stack is empty")
28      self._head._element
```

```
29
30    def pop(self):
31      if self.is_empty():
32        raise Exception("Stack is empty")
33      answer = self._head._element
34      self._head = self._head._next
35      self._size -=1
36      return answer
```

6.1.1   Complexity Analysis

Operation	Running Time
S.push()	$O(1)$
S.pop()	$O(1)$
S.top()	$O(1)$
S.is_empty()	$O(1)$
len(S)	$O(1)$

Linked stacks are good in real-time applications since no amortization occurs (could happen at a critical time).