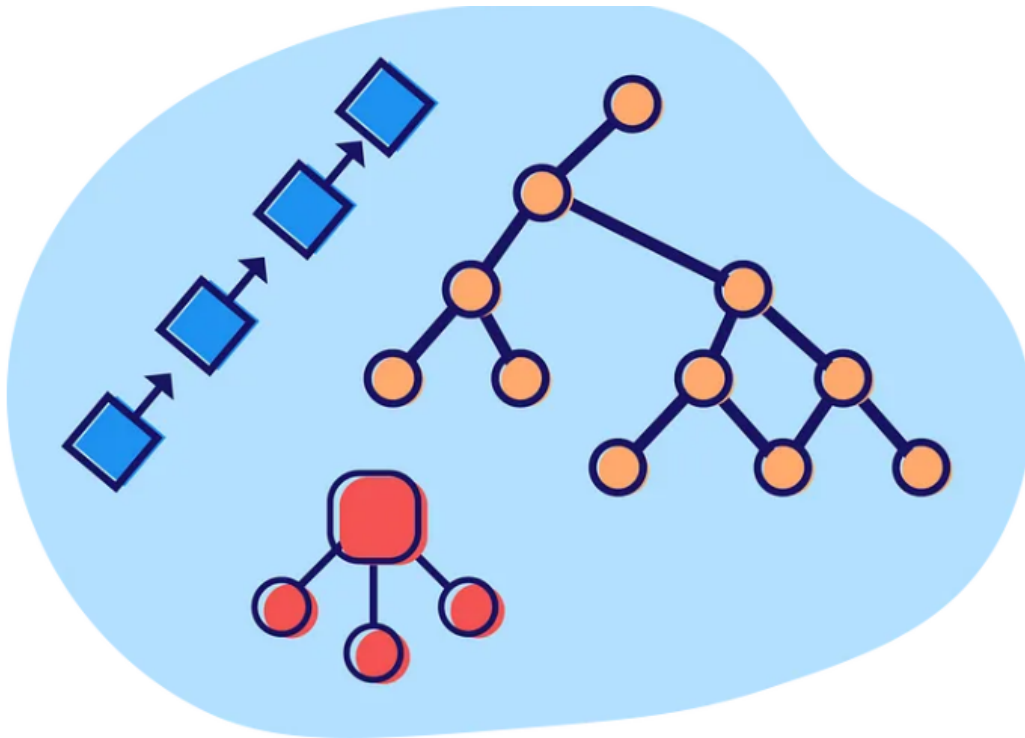




EE234L: Data Structures And Algorithms

Lab Manual

Author and Instructor: Rehan Naeem



Department of Electrical Engineering
University of Engineering and Technology, Lahore

Contents

1	Implementation of LIST using LINKED LIST	3
1.1	Empty Linked List	3
1.2	The AddFirst Method	3
1.3	The AddLast Method	4
1.4	The GetFirst Method	5
1.5	The Size Method	5
1.6	Testing	5
2	Implementation of DEQUE using Circular, Doubly Linked LIST	6
2.1	The Constructor	6
2.2	The AddFirst and AddLast Methods	6
2.3	Testing	6
2.4	The ToList Method	7
2.5	The IsEmpty and Size Methods	8
2.6	The Get Method	8
2.7	The RemoveFirst and RemoveLast Methods	8
3	Implementation of LIST using ARRAYS	10
3.1	The Constructor	10
3.2	The AddLast Method	10
3.3	The GetLast Method	10
3.4	The Get Method	10
3.5	The Size Method	10
3.6	The RemoveLast Method	11
3.7	The AddLast Method with Resizing	11
3.8	The RemoveLast Method with Resizing	11
3.9	Test Code	11
4	Implementation of DEQUE using Circular ARRAYS	12
4.1	The Constructor	12
4.2	The AddLast and AddFirst Methods	12
4.2.1	Resizing Up and Down	13
4.3	The Get Method	13
4.4	The isEmpty and Size Methods	13
4.5	The toList Method	13
4.6	Test Code	13
5	Implementation of DEQUE using Circular ARRAYS	14
5.1	The Constructor	14
5.2	The AddLast and AddFirst Methods	14
5.2.1	Resizing Up and Down	15
5.3	The Get Method	15
5.4	The isEmpty and Size Methods	15
5.5	The toList Method	15
5.6	Test Code	15

6	Implementation of a MAP using BINARY SEARCH TREE	16
6.1	The Constructor of BSTMap Class	16
6.2	The Vertex class	16
6.3	The Insert Method	16
6.4	The Find Method	17
6.5	Test Code	17

Chapter 1

Implementation of LIST using LINKED LIST

1.1 Empty Linked List

Write a class `SLList` having one instance attribute called `sentinel` which should point to the sentinel node of the linked list. Use any sort of visualizer like PythonTutor to see the node being created.

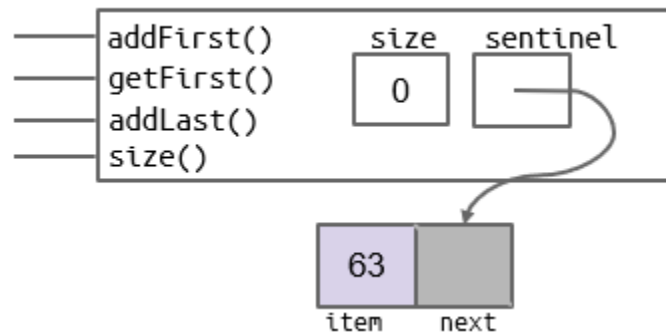


Figure 1.1: Sentinel Linked List

```
8  ▶ class Node:
13
14  ▶ class SLList:
27
28
29  L = SLList()
```

Figure 1.2: Incomplete Code

1.2 The AddFirst Method

Write a method `AddFirst` in the class `SLList` which adds a new node at the front of the existing linked list. Write a separate class `Node` to create each node of the linked list. No other method is required to be present in `Node`.

```

8  > class Node:
13
14  > class SLList:
27
28
29  L = SLList()
30  L.AddFirst(5)

```

Figure 1.3: Incomplete Code

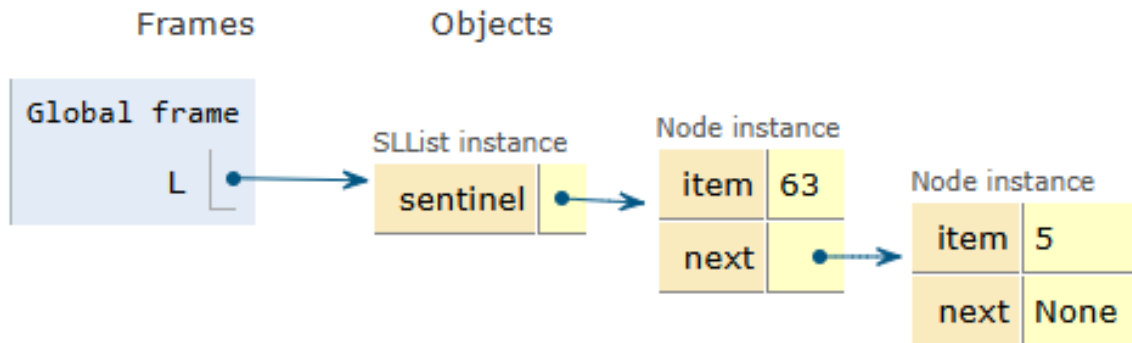


Figure 1.4: Adding a Node to the front

1.3 The AddLast Method

Write a method `AddLast` which adds a new node to the back of the existing linked list.

```

8  > class Node:
13
14  > class SLList:
27
28
29  L = SLList()
30  L.AddFirst(5)
31  L.AddFirst(10)
32  L.AddLast(20)

```

Figure 1.5: Incomplete Code

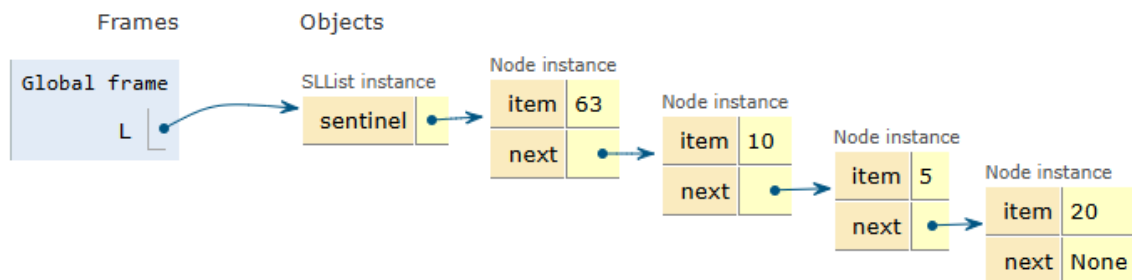


Figure 1.6: Adding a Node to the front

1.4 The GetFirst Method

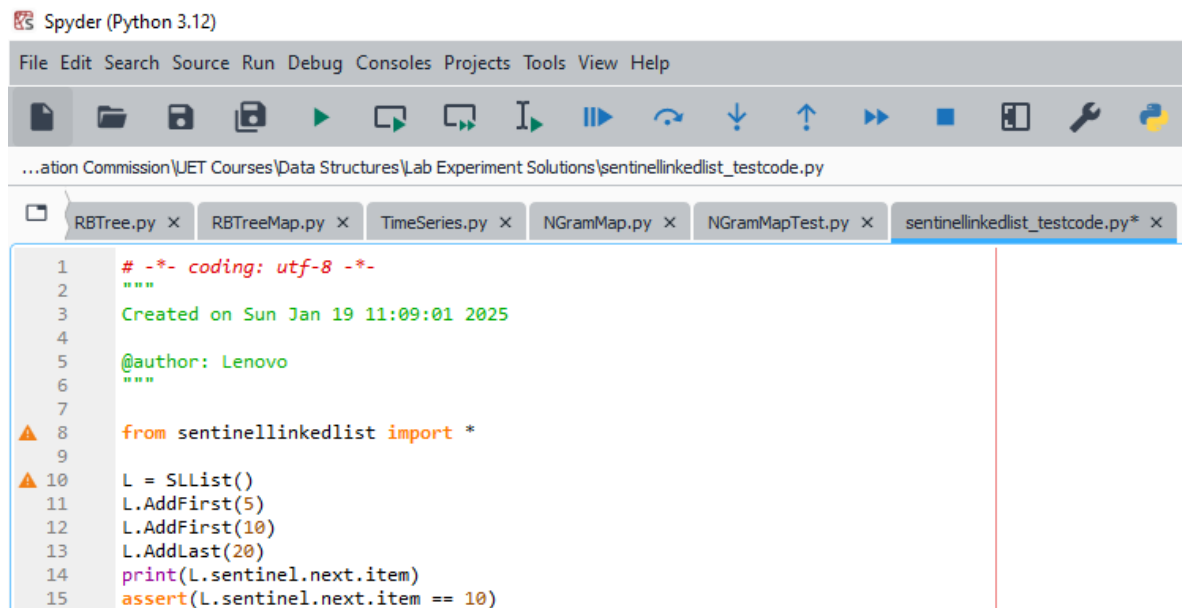
Write a method `GetFirst` which returns the item of the first node in the current linked list.

1.5 The Size Method

Write a method `Size` which returns the number of nodes in the linked list excluding the sentinel node. Use cache technique discussed in the lab.

1.6 Testing

The code line 29 - 32 in figure 1.5 is basically a test code to test the two classes Node and SLList. Write a test code in a separate file to test both the classes completely. The test code should use the `print` or `assert` statements to test. The word `sentinellinkedlist` on line 8, figure 1.7 is the name of the file containing the code for the classes and methods explained above.



The screenshot shows the Spyder Python IDE interface. The title bar indicates 'Spyder (Python 3.12)'. The menu bar includes 'File', 'Edit', 'Search', 'Source', 'Run', 'Debug', 'Consoles', 'Projects', 'Tools', 'View', and 'Help'. The toolbar contains various icons for file operations and execution. The file explorer shows the path '...ation Commission\JET Courses\Data Structures\Lab Experiment Solutions\sentinellinkedlist_testcode.py'. The editor window displays the following code:

```
1  # -*- coding: utf-8 -*-
2  """
3  Created on Sun Jan 19 11:09:01 2025
4
5  @author: Lenovo
6  """
7
8  from sentinellinkedlist import *
9
10 L = SLList()
11 L.AddFirst(5)
12 L.AddFirst(10)
13 L.AddLast(20)
14 print(L.sentinel.next.item)
15 assert(L.sentinel.next.item == 10)
```

Figure 1.7: Incomplete Test Code

Chapter 2

Implementation of DEQUE using Circular, Doubly Linked LIST

2.1 The Constructor

Create a new file and name it `LinkedListDeque.py`. Write a class `DLList` to implement the constructor for `LinkedListDeque`. Along the way you'll need to create a `Node` class and introduce one or more instance variables. This may take you some time to understand fully. Your `LinkedListDeque` constructor must take 0 arguments and must be written in class named `DLList`. Additionally, you should only have one class named `Node` (The one you created in lab 1 by the name `Node`).

Test your code using Python Tutor. You should see the environment diagram as shown below.

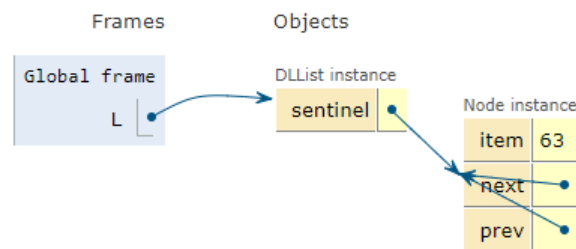


Figure 2.1:

2.2 The AddFirst and AddLast Methods

`addFirst` and `addLast` may not use looping or recursion. A single add operation must take "constant time," that is, adding an element should take approximately the same amount of time no matter how large the deque is. This means that you cannot use loops that iterate through all / most elements of the deque.

Fill in the `addFirst` and `addLast` methods. Then, debug the following code. This test will not pass because you haven't written `toList` yet, but you can use the debugger and visualizer to verify that your code is working correctly. Note: class `Node` and `DLList` shown in the figure below is to be completed by you.

2.3 Testing

Create a new file and name it `LinkedListDequeTest.py`. Copy and paste line 77 to 82 of the code shown in the above figure.

Run the code. It should fail because `toList()` method has not been implemented yet.

```

29 class Node:
35
36 class DLList:
76
77 L = DLList() #not allowed to add using this format
78 L.addlast(5)
79 L.addlast(9)
80 L.addlast(10)
81 L.addfirst(3)
82 assert(L.toList() == [3,5,9,10])

```

Figure 2.2:

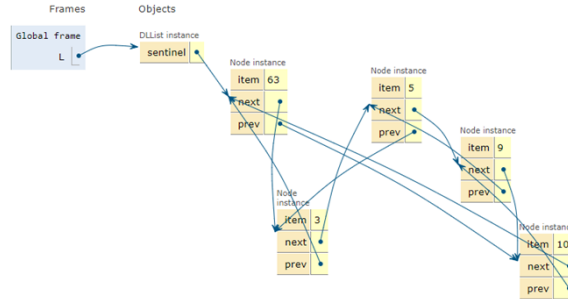


Figure 2.3:

```

1  # -*- coding: utf-8 -*-
2  """
3  Created on Tue Jan 30 06:53:32 2024
4
5  @author: Lenovo
6  """
7
8  from LinkedListDeque import *
9
10 L = DLList() #not allowed to add using this format
11
12 L.addlast(5)
13 L.addlast(9)
14 L.addlast(10)
15 L.addfirst(3)
16 assert(L.toList() == [3,5,9,10])

```

Figure 2.4:

Look at the code below. New functions have been written like CreateEmptyList, AddItems etc that do the same job as the code in the above figure. The advantage is that the code has become more understandable in terms of the work it is doing. AddItems have been implemented for you. Implement the remaining functions yourself. Test the code. It should fail because toList() method has not been implemented yet.

2.4 The ToList Method

You may have found it somewhat tedious and unpleasant to use the debugger and visualizer to verify the correctness of your addFirst and addLast methods. There is also the problem that such manual verification becomes stale as soon as you change your code. Imagine that you made some minor but uncertain change to addLast. To verify that you didn't break anything you'd have to go back and do that whole process again. Yuck.

What we really want are some automated tests. But unfortunately there's no easy way to verify correctness of addFirst and addLast if those are the only two methods we've implemented. That is, there's currently no way to iterate over our list and get back its values and see that they are correct.

That's where the toList method comes in. When called, this method returns a List representation of


```

1  # -*- coding: utf-8 -*-
2  """
3  Created on Tue Jan 30 06:53:32 2024
4
5  @author: Lenovo
6  """
7
8  from LinkedListDeque import *
9
10 def CreateEmptyList():
11
12
13
14 def AddItems():
15     L.addlast(5)
16     L.addlast(9)
17     L.addlast(10)
18     L.addfirst(3)
19
20 def OrderTest():
21
22
23 L = CreateEmptyList()
24 AddItems() #function to add items to the Deque.
25 OrderTest() #function to test the order of the items added

```

Figure 2.5:

the Deque. For example, if the Deque has had `addLast(5)`, `addLast(9)`, `addLast(10)`, then `addFirst(3)` called on it, then the result of `toList()` should be a List with 3 at the front, then 5, then 9, then 10. If printed in PYTHON, it'd show up as `[3, 5, 9, 10]`. You are allowed to use built-in list of PYTHON and its methods to implement `toList`.

Run the code in `LinkedListTest.py`. It should run successfully without displaying any message in the console window.

All that's left is to test and implement all the remaining methods. For the rest of this project, we'll describe our suggested steps at a high level. We strongly encourage you to follow the remaining steps in the order given. In particular, write tests before you implement. This is called “test-driven development,” and helps ensure that you know what your methods are supposed to do before you do them.

2.5 The isEmpty and Size Methods

These two methods must take constant time. That is, the time it takes for either method to finish execution should not depend on how many elements are in the deque.

Write one or more tests for `isEmpty` and `size`. Run them and verify that they fail. Your test(s) should verify more than one interesting case, such as checking both an empty and a nonempty list, or checking that the size changes.

Your tests can range from very fine-grained, e.g. `testIsEmpty`, `testSizeZero`, `testSizeOne` to very coarse grained, e.g. `testSizeAndIsEmpty`. It's up to you to explore and find what granularity you prefer.

Task: Write tests for the `isEmpty` and `size` methods, and check that they fail. Then, implement the methods.

2.6 The Get Method

Write a test for the `get` method. Make sure to test the cases where `get` receives an invalid argument, e.g. `get(28723)` when the Deque only has 1 item, or a negative index. In these cases `get` should return null. `get` must use iteration.

2.7 The RemoveFirst and RemoveLast Methods

Lastly, write some tests that test the behavior of `removeFirst` and `removeLast`, and again ensure that the tests fail. For these tests you'll want to use `toList`! Use `addFirstAndAddLastTest` as a guide.

Do not maintain references to items that are no longer in the deque. The amount of memory that your program uses at any given time must be proportional to the number of items. For example, if

you add 10,000 items to the deque, and then remove 9,999 items, the resulting memory usage should amount to a deque with 1 item, and not 10,000. Remember that the Java garbage collector will “delete” things for us if and only if there are no pointers to that object.

Chapter 3

Implementation of LIST using ARRAYS

3.1 The Constructor

In this lab you are required to implement `list` using arrays. Write a class `AList` which has the attributes shown below. The attribute `items` contains the address of the array. Here all zeros in the array indicates that the array is empty. For example, `a = AList()` should create the instances shown below.

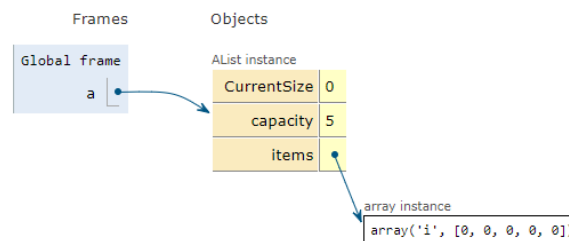


Figure 3.1: An Empty Array-based List

3.2 The AddLast Method

Write a method `AddLast` which adds an element `i` to the end of the array.

3.3 The GetLast Method

Write a method `GetLast` which returns the last element of the array.

3.4 The Get Method

Write a method `Get` which returns the element at index `i` of the array. For example, `a.get(1)` should return the element at index 1. `None` should be returned when the index is invalid.

3.5 The Size Method

Write a method `Size` which returns the current size of the array.

3.6 The RemoveLast Method

Write a method RemoveLast which removes the last element of the array. To remove, set the value at the removed index to zero.

3.7 The AddLast Method with Resizing

Modify the AddLast method so that a new larger array is created when the array becomes full. All the elements in the older array should be transferred to the larger array. What should be the capacity of the new array? Discuss with your instructor. Write a function Resize inside the class AList to create the new larger array, transfer elements from older to newer array. Call it inside the AddLast method.

3.8 The RemoveLast Method with Resizing

Modify the RemoveLast method so that a new smaller array is created when $\frac{\text{currentsize}}{\text{capacity}} < 0.25$. This will save memory space. Use the same Resize method you wrote in step 7 with some modifications. There should be one Resize method for both the AddLast and RemoveLast methods.

3.9 Test Code

Write a test code, in a separate file as you did in the last lab, to test all the methods above. It is advised to write test code in parallel to writing each of the above methods.

Chapter 4

Implementation of DEQUE using Circular ARRAYS

As your second deque implementation, you'll write the `ADeque` class. This deque must use a Python array as the backing data structure.

4.1 The Constructor

You will need to somehow keep track of what array indices hold the deque's front and back elements. We strongly recommend that you treat your array as circular for this exercise. In other words, if your front item is at position 0, and you `addFirst`, the new front should loop back around to the end of the array (so the new front item in the deque will be the last item in the underlying array). This will result in far fewer headaches than non-circular approaches. The variables of the constructor (`init` method) are shown in the figure below.

Starting from an empty `ArrayDeque`:

- `addLast("a")`
- `addLast("b")`

Conceptual Deque: [a, b]

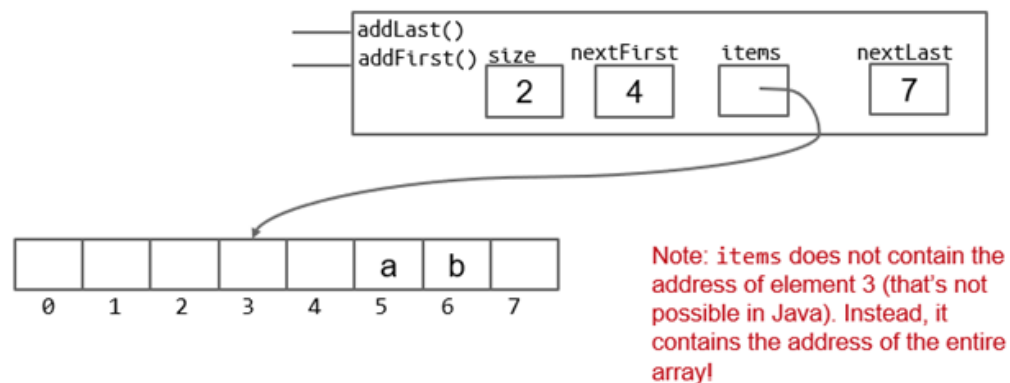


Figure 4.1: Array Based Deque

4.2 The AddLast and AddFirst Methods

As before, implement `addFirst` and `addLast` methods. These two methods must not use looping or recursion. A single add operation must take "constant time," that is, adding an element should take approximately the same amount of time no matter how large the deque is (with one exception). This means that you cannot use loops that iterate through all / most elements of the deque.

4.2.1 Resizing Up and Down

The exception to the “constant time” requirement is when the array fills, and you need to “resize up” to have enough space to add the element. In this case, you can take “linear time” to resize the array before adding the element. Similarly you have to resize the array down when $R < 0.25$ (as done in the previous lab). Correctly resizing your array is very tricky, and will require some deep thought. Try drawing out various approaches by hand. It may take you quite some time to come up with the right approach, and we encourage you to debate the big ideas with your fellow students or TAs. Make sure that your actual implementation is by you alone. Make sure to resize by a geometric factor.

4.3 The Get Method

Write a method `Get` which returns the element at index `i` of the array. For example, `a.get(1)` should return the element at index 1. `None` should be returned when the index is invalid.

4.4 The isEmpty and Size Methods

The `isEmpty` method should return `True` if the deque is empty else return `False`. The deque is considered to be empty when all the array elements are zero. The size method should return the number of elements stored in the array (not the capacity of the array!). These two methods must take constant time. That is, the time it takes for either method to finish execution should not depend on how many elements are in the deque.

4.5 The toList Method

The `toList` method should return a list of items stored in the deque. The order of the items in the list should be the same order in which they were added to the deque. You are allowed to use the built-in Python’s list for this task. For example, the following code should print the list `[15, 5, 10]` when `toList` method is called.

```
class ADeque:

    a = ADeque()
    a.AddLast(5)
    a.AddLast(10)
    a.AddFirst(15)
    print(a.toList())
```

4.6 Test Code

Write a test code, in a separate file as you did in the last lab, to test all the methods above. It is advised to write test code in parallel to writing each of the above methods.

Chapter 5

Implementation of DEQUE using Circular ARRAYS

As your second deque implementation, you'll write the `ADeque` class. This deque must use a Python array as the backing data structure.

5.1 The Constructor

You will need to somehow keep track of what array indices hold the deque's front and back elements. We strongly recommend that you treat your array as circular for this exercise. In other words, if your front item is at position 0, and you `addFirst`, the new front should loop back around to the end of the array (so the new front item in the deque will be the last item in the underlying array). This will result in far fewer headaches than non-circular approaches. The variables of the constructor (`init` method) are shown in the figure below.

Starting from an empty `ArrayDeque`:

- `addLast("a")`
- `addLast("b")`

Conceptual Deque: [a, b]

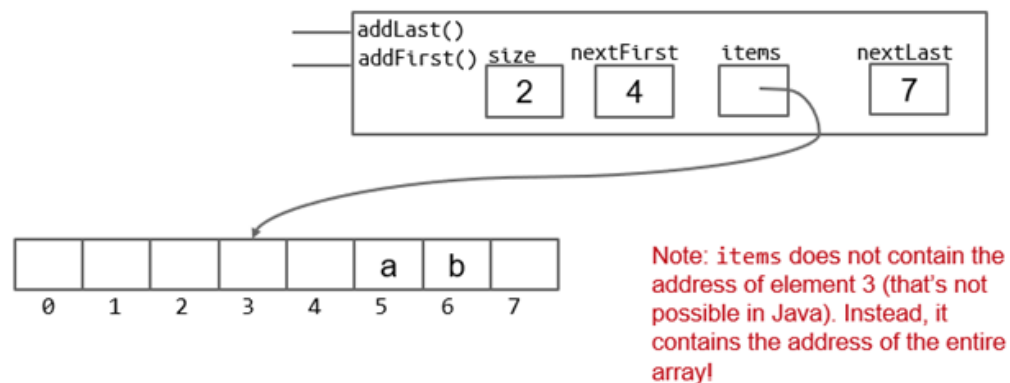


Figure 5.1: Array Based Deque

5.2 The `AddLast` and `AddFirst` Methods

As before, implement `addFirst` and `addLast` methods. These two methods must not use looping or recursion. A single add operation must take "constant time," that is, adding an element should take approximately the same amount of time no matter how large the deque is (with one exception). This means that you cannot use loops that iterate through all / most elements of the deque.

5.2.1 Resizing Up and Down

The exception to the “constant time” requirement is when the array fills, and you need to “resize up” to have enough space to add the element. In this case, you can take “linear time” to resize the array before adding the element. Similarly you have to resize the array down when $R < 0.25$ (as done in the previous lab). Correctly resizing your array is very tricky, and will require some deep thought. Try drawing out various approaches by hand. It may take you quite some time to come up with the right approach, and we encourage you to debate the big ideas with your fellow students or TAs. Make sure that your actual implementation is by you alone. Make sure to resize by a geometric factor.

5.3 The Get Method

Write a method `Get` which returns the element at index `i` of the array. For example, `a.get(1)` should return the element at index 1. `None` should be returned when the index is invalid.

5.4 The isEmpty and Size Methods

The `isEmpty` method should return `True` if the deque is empty else return `False`. The deque is considered to be empty when all the array elements are zero. The `size` method should return the number of elements stored in the array (not the capacity of the array!). These two methods must take constant time. That is, the time it takes for either method to finish execution should not depend on how many elements are in the deque.

5.5 The toList Method

The `toList` method should return a list of items stored in the deque. The order of the items in the list should be the same order in which they were added to the deque. You are allowed to use the built-in Python’s list for this task. For example, the following code should print the list `[15, 5, 10]` when `toList` method is called.

```
class ADeque:

    a = ADeque()
    a.AddLast(5)
    a.AddLast(10)
    a.AddFirst(15)
    print(a.toList())
```

5.6 Test Code

Write a test code, in a separate file as you did in the last lab, to test all the methods above. It is advised to write test code in parallel to writing each of the above methods.

Chapter 6

Implementation of a MAP using BINARY SEARCH TREE

In this lab, you will write code to implement the MAP ADT using the Binary Search Tree (BST) without any balancing mechanism.

6.1 The Constructor of BSTMap Class

Write a class BSTMap that creates the structure shown in figure when the instruction `t = BSTMap()` runs.

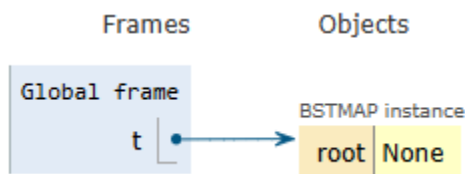


Figure 6.1: An Empty BST

6.2 The Vertex class

Write a class vertex to create one vertex having data attributes shown in figure.

vertex instance	
key	4
left	None
right	None
value	"uet"

Figure 6.2: A Vertex

6.3 The Insert Method

Write a method Insert of the class BSTMap to insert a key-value pair of information into the BST recursively. For example, inserting the pairs (4, 'uet'), (2, 'uet'), (1, 'uet'), (3, 'uet'), (7, 'uet'), (5, 'uet'), (8, 'uet') should create the tree shown in figure 6.3. Use the class Vertex to create one vertex and insert it at its proper location.

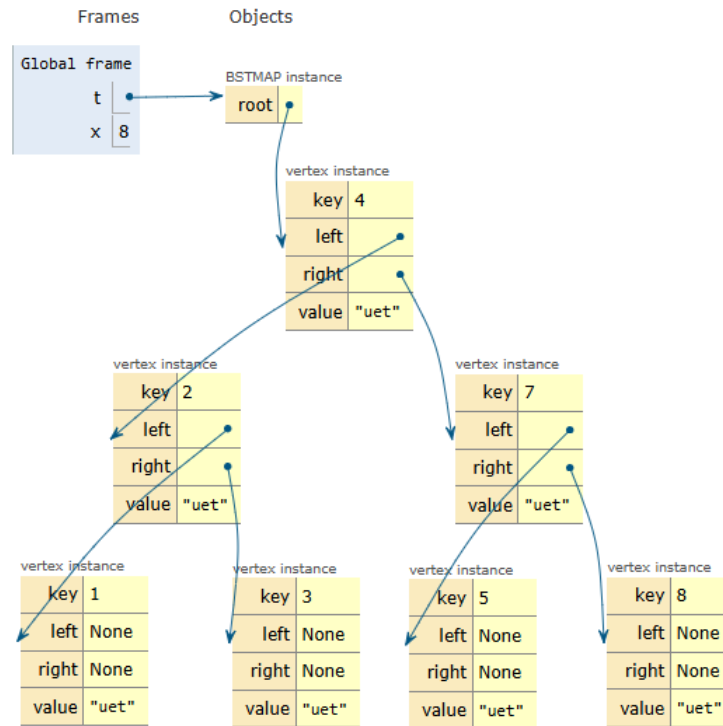


Figure 6.3: A Binary Search Tree

6.4 The Find Method

Write a method Find that searches for a key in the BST and returns True if the key is found else returns False.

6.5 Test Code

Write a test code, in a separate file as you did in the last lab, to test all the methods above. It is advised to write test code in parallel to writing each of the above methods. You can import this [file](#) into your test code and display the whole tree using the instruction `display(t.root)` outside the class as shown in figure 6.4.

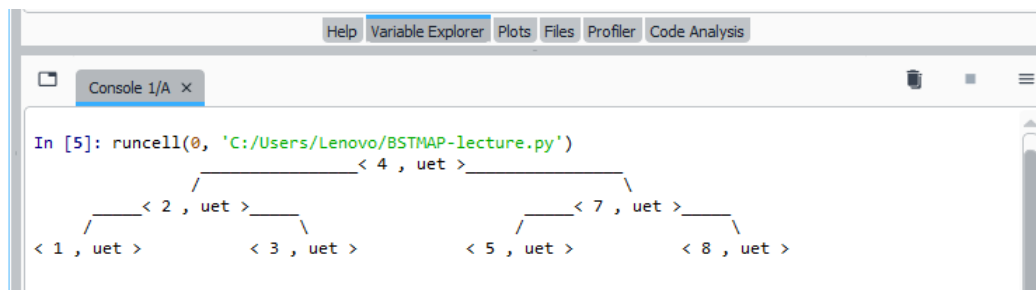


Figure 6.4: A Binary Search Tree in SPYDER Console