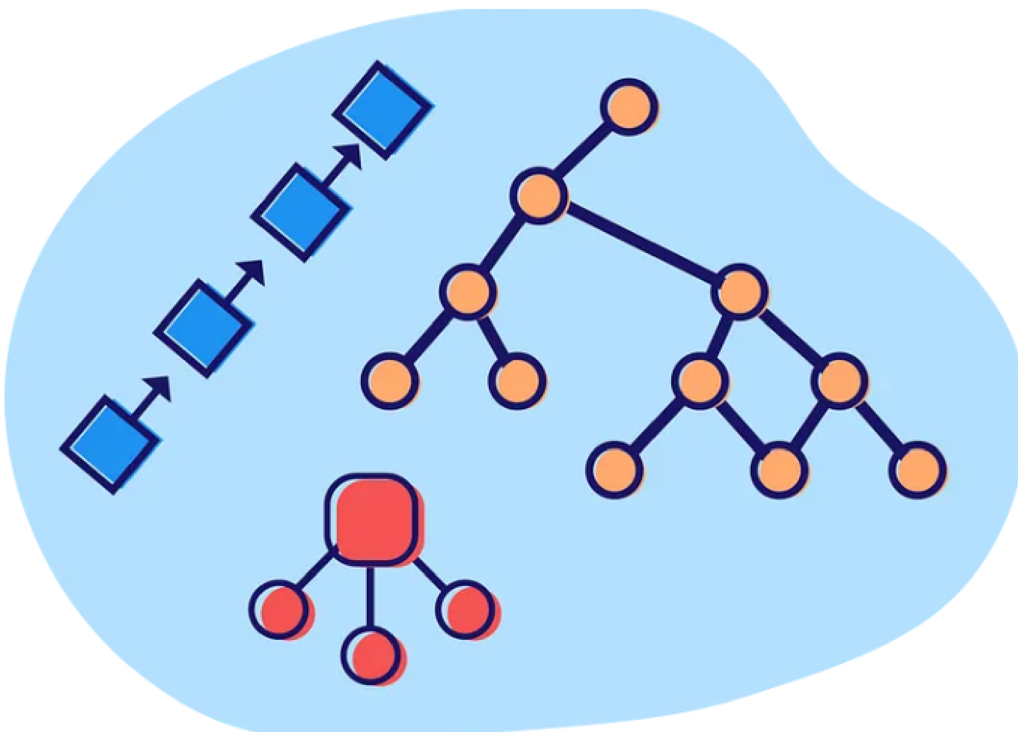
EE234L: Data Structures And Algorithms

Lab Manual

Author and Instructor: Rehan Naeem



Department of Electrical Engineering
University of Engineering and Technology, Lahore

Contents

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.

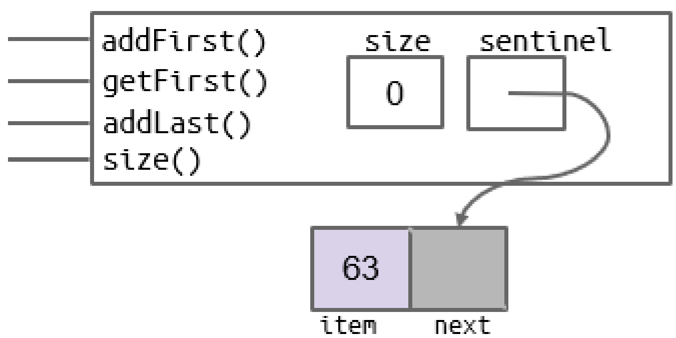


Figure 1.1: Sentinel Linked List

```
class Node:

class SLList:


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.

```
class Node:

class SLList:


L = SLList()
L.AddFirst(5)
```

Figure 1.3: Incomplete Code

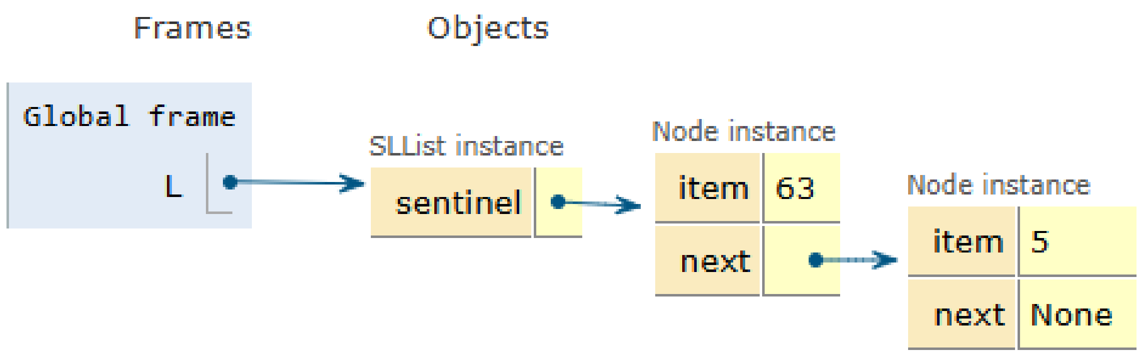


Figure 1.4: Adding a Node to the front

1.3 The `AddLast` Method

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

```
class Node:

class SLList:


L = SLList()
L.AddFirst(5)
L.AddFirst(10)
L.AddLast(20)
```

Figure 1.5: Incomplete Code

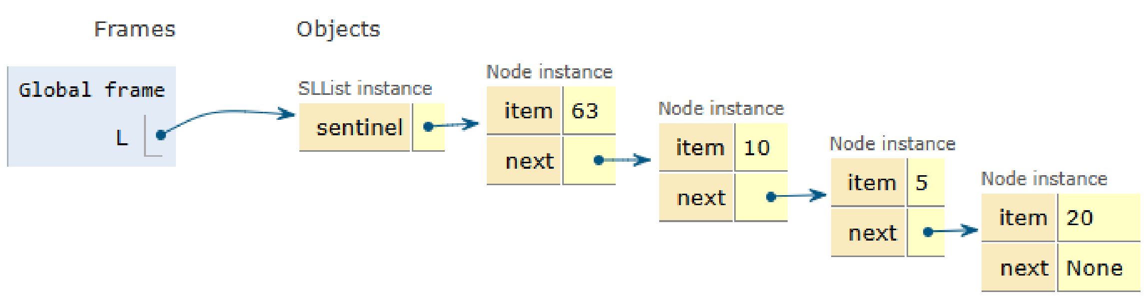


Figure 1.6: Adding a Node to the front

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.

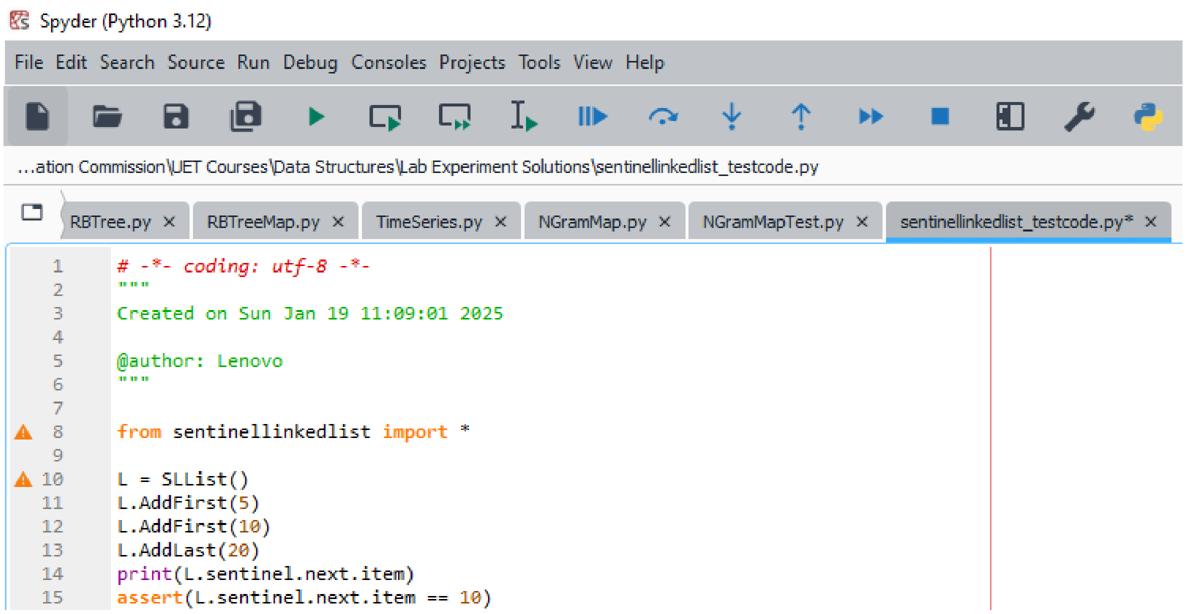


Figure 1.7: Incomplete Test Code

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.

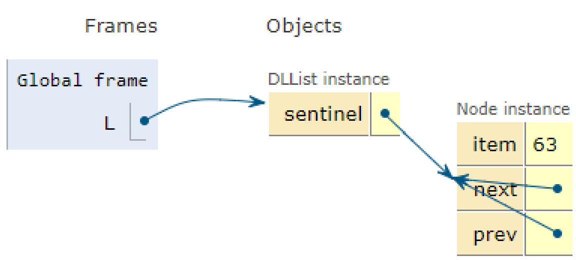


Figure 2.1:

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.

```python
class Node:

class DLList:

L = DLList() #not allowed to add using this format
L.addlast(5)
L.addlast(9)
L.addlast(10)
L.addfirst(3)
assert(L.toList() == [3,5,9,10])
```

Figure 2.2:

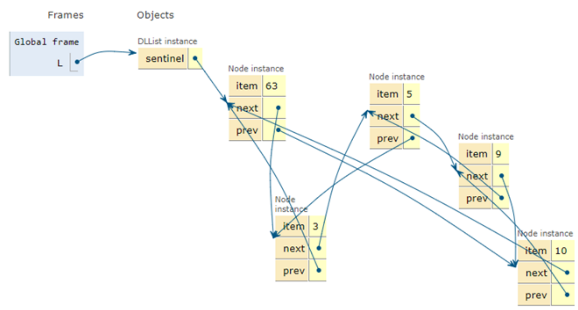


Figure 2.3:

LinkedListDeque.py × LinkedListTest.py ×

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Jan 30 06:53:32 2024

@author: Lenovo
"""

from LinkedListDeque import *

L = DLList() #not allowed to add using this format

L.addlast(5)
L.addlast(9)
L.addlast(10)
L.addfirst(3)
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

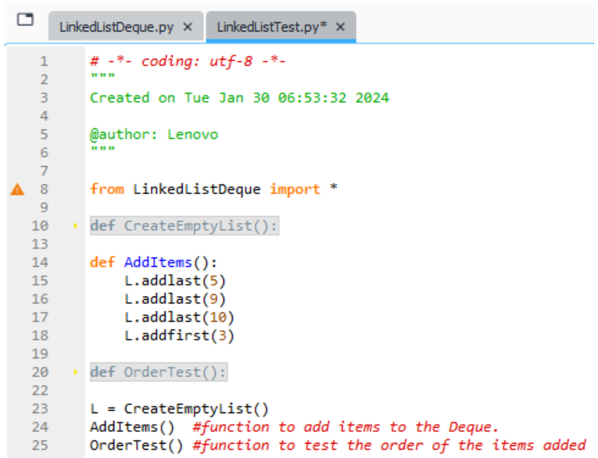


Figure 2.5:

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.

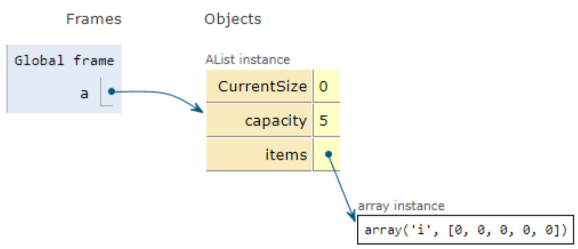


Figure 3.1: An Empty Array-based List

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 $\frac{currentsize}{capacity} < 0.25$. 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.

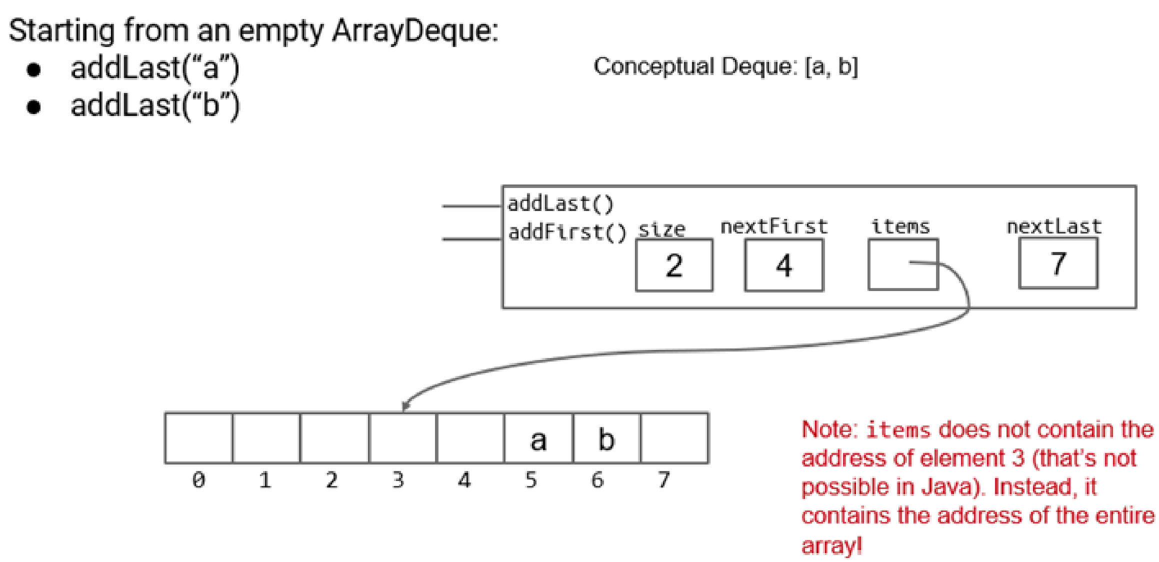


Figure 4.1: Array Based Deque

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.

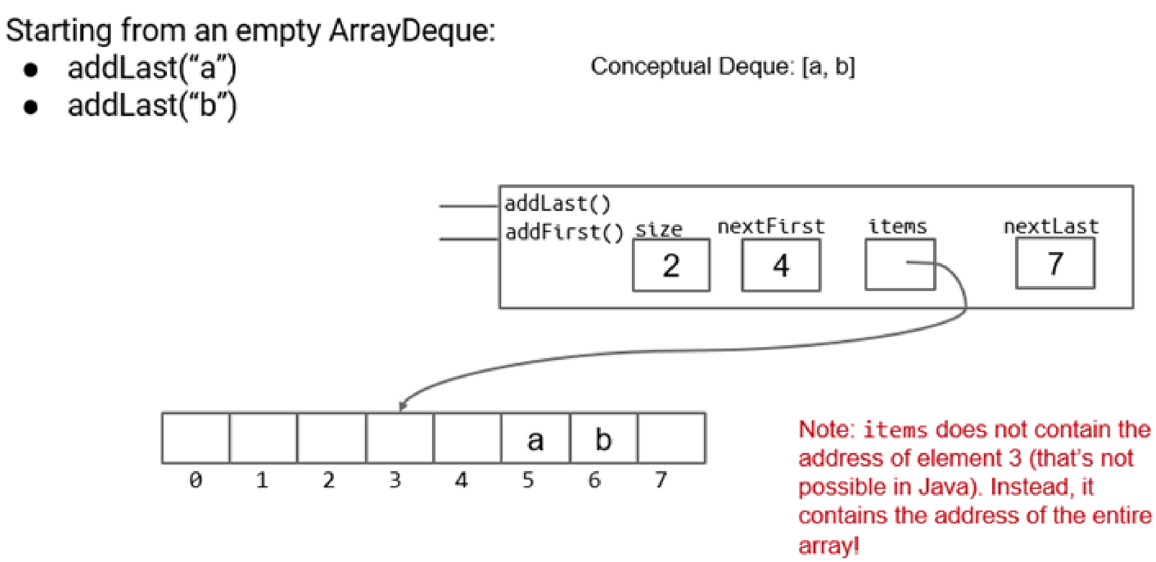


Figure 5.1: Array Based Deque

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.

```python
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.

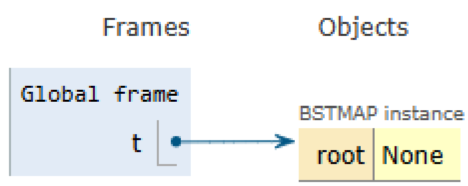


Figure 6.1: An Empty BST

6.2 The Vertex class

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

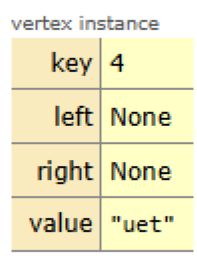


Figure 6.2: A Vertex

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.

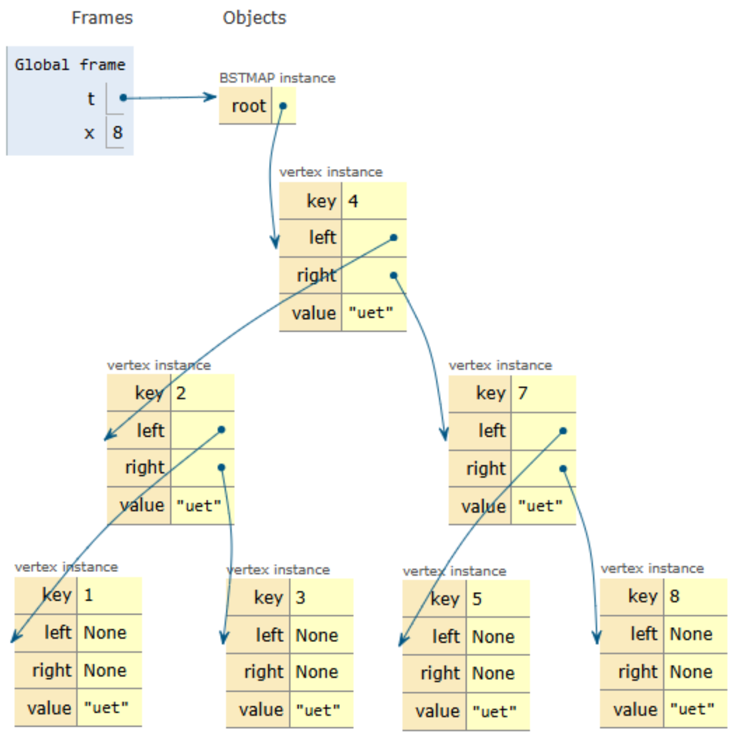


Figure 6.3: A Binary Search Tree

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.

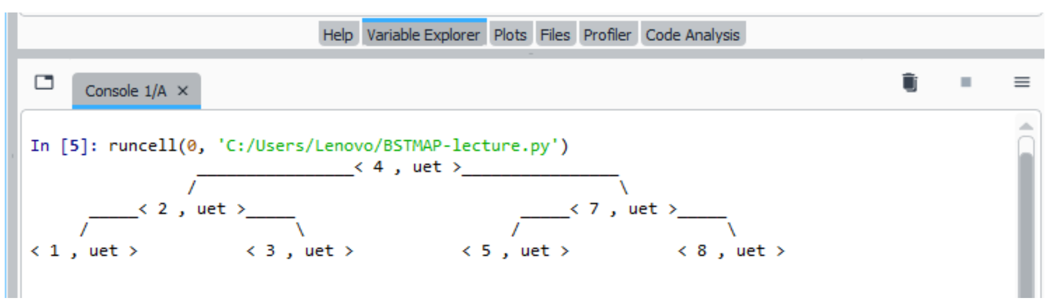


Figure 6.4: A Binary Search Tree in SPYDER Console

Chapter 7

Implementation of a `MAP` using `RED BLACK TREE`

In this lab, you will write code to implement the MAP ADT using the Binary Search Tree (BST) with any balancing mechanism i.e. a Red-Black tree.

In lecture, we've introduced the concept of LLRBs with links. However, for this lab, we will not be representing our LLRBs with links. Instead, we'll be using nodes. The main reason for this is that the implementation with links is much harder than with nodes. To cover our ground here, consider the example below of how we've introduced the visualization of it in lecture. For this lab, since we'll be handling colored nodes - the relationship between the red link and its connected node can be defined like below.

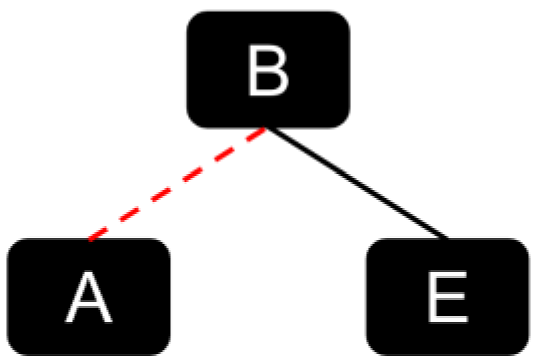


Figure 7.1: A Red Black Tree

Originally, A was connected by a red link. But if we use colored nodes in our representation instead of links, A itself would be colored red. The visualization above is meant to show how a red link would map to a colored node, so please keep this in mind for the rest of the lab!

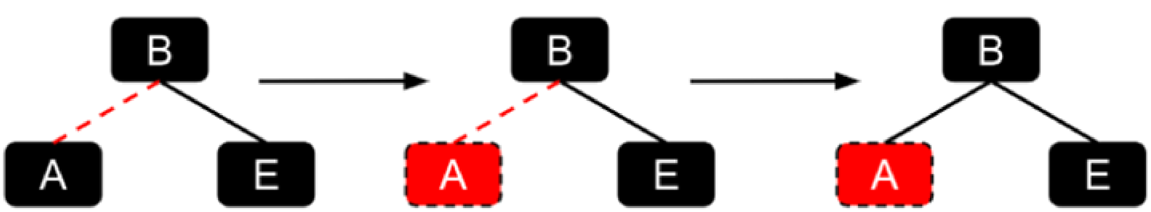


Figure 7.2: An Empty BST

7.1 The Vertex Class

To the class Vertex you created in the last lab, add a new data attribute called color to represent the color of the vertex. Store 'r' or 'b' in color to represent red or black color vertex.

7.2 The Constructor of BSTMap Class

Use the same BSTMap class you wrote in the last lab to create an empty Red Black tree.

7.3 The RotateRight Method

Write a method rotateRight to rotate a node towards the right. Assume the node to be rotated is the root node.

7.4 The RotateLeft Method

Write a method rotateLeft to rotate a node towards the left. Assume the node to be rotated is the root node.

7.5 The ColorFlip Method

Write a method colorFlip that flips the color of the node and of its children. Assume a tree with three nodes; parent node is black and children nodes are red. Please don't imagine anything else when writing the code for this method.

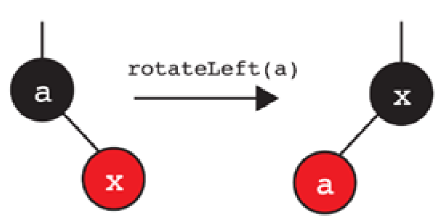
7.6 The Insert Method

Write a method insert to insert a node into the tree. Hint: Use the insert code you developed for binary search tree without balance. Right below that code, inside the insert function, write the four rules in the form of a python code to create a LLRB tree. Keep the diagrams below in mind when writing the code to implement the four rules of insertion. Please don't imagine anything else like a big size tree or a tree with some nodes randomly red or black.

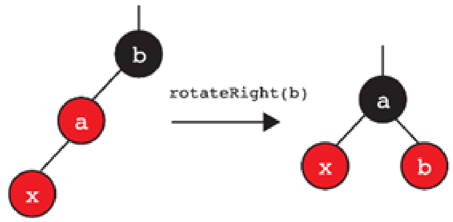
7.6.1 Rule 1: Inserting A New Vertex

The color of the new vertex should be red.

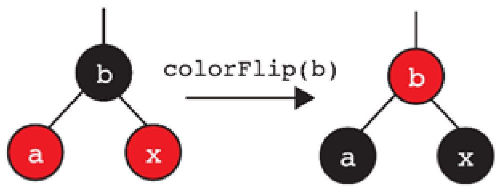
7.6.2 Rule 2: A Right Red Vertex



7.6.3 Rule 3: Two Left Red Vertices



7.6.4 Rule 4: Color Flip



7.7 The Find Method

Write a method Find that returns True if the key is found otherwise return False.

7.8 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 below.

```
#%%
T = RBTree()
for k in [5,11,3,9,7,1,2]:
    T.Insert(k)
display(T.root)
```

Figure 7.3: Sample Test Code

```
In [81]: runcell(1
Data Structures/La
     __5b___
    /       \
  2b_      9b_
 /   \    /   \
1b   3b  7b   11b
```

Figure 7.4: Output of the Test Code. Small b means black node.

```
#%%

T = RBTree()
for k in 'qonputvmgserb':
    T.Insert(k)
display(T.root)
```

Figure 7.5: Sample Test Code

```
In [83]: runcell(2, 'E:/OneDriv
Data Structures/Lab Manual/RBTr
        ______qb_______
       /               \
    __mr___         __ub_
   /       \       /     \
  eb_     ob_     sr_   vb
 /   \   /   \   /   \
bb  gb  nb  pb  rb  tb
```

Figure 7.6: Output of the Test Code. Small b means black node. Small r means red node.

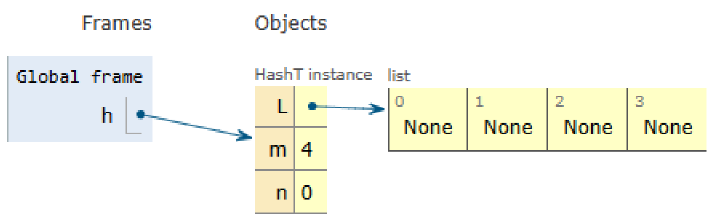
Chapter 8

Implementation of a `MAP` using `Hash Table`

In this lab, you will write code to implement the MAP ADT using a Hash Table.

8.1 The Constructor of `HashT` Class

Write the class HashT to create an empty Hash Table as shown below. You can use the built-in List in place of the array. Write a class Vertex to create one vertex of the linked list as done in one of the previous labs. In the figure below, m is the number of bins and n is the number of inserted items.

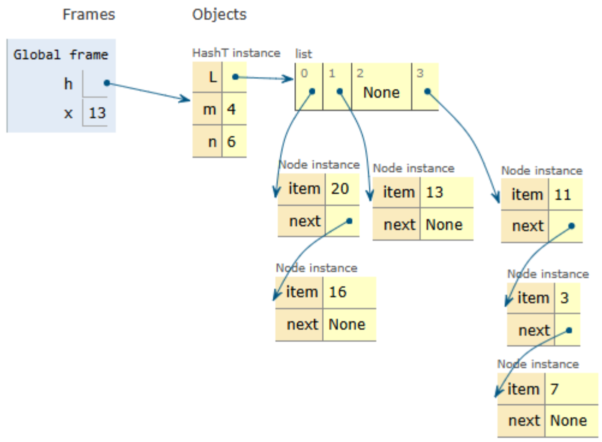


8.2 The `HashFunc` Method

Write the HashFunc method that takes an integer or a string as an argument and returns the hash code of the argument. Use the hash function explained in the lab.

8.3 The `Insert` Method

Write the insert(k, v) method which inserts key k along with its value v into the linked list as shown in the figure below. The key k could be a number or a string. Set the load factor N/M = 0.75 for resizing. When resizing, create a new array, store the items of the old array into the new array after calculating the new index. The figure below shows the sequence 7, 16, 3, 11, 20 and 13 inserted into the hash table and doesnt account for the resizing. You can visualize the Hash Table using this file or the python tutor website.



8.4 The `Contains` Method

Write a method Contains that returns True if the key is found otherwise return False. The argument of the method is a key.