outlabq2

210050159

Sep 27, 2022

CONTENTS:

CHAPTER

ONE

BONUS

1.1 BinarySearchTree module

This module contains the implementation of abstract data type Binary Search Tree. To achieve that, a Binary Search Tree node (BSTNode) class has been implemented as well.

`class BinarySearchTree.`**`BSTNode`**`(`*info*`)`

Bases: `object`

This is the class implementation of node for binary search tree. This node/object of this class has the property that it contains address of the left and right child of itself.

This class has one constructor and one convertor:

- __init__(data) is the constructor, and
- __str__() is the convertor.

`__init__``(`*info*`)`

Constructor method for node of Binary Search Tree. This stores the given value(info) in info and sets the left, right and level to None.

Example

```
>>> from BinarySearchTree import *
>>> sample = BSTNode(10)
>>> print(sample.info)
10
>>> print(sample.left)
None
>>> print(sample.right)
None
>>> print(sample.level)
None
```

`__str__``()`

Convertor method for node of binary search tree. This returns the string form of the info stored in the node.

Example

```
>>> from BinarySearchTree import *
>>> sample = BSTNode(10)
>>> output = sample.__str__()
>>> print(output)
10
>>> print(type(output))
<class 'str'>
```

class `BinarySearchTree.`**`BinarySearchTree`**

Bases: `object`

This is the class implementation of Binary Search Tree. An instance of this class represents a Binary Search Tree which supports insertion at a node, traversal of tree and finding height of a node.

This class has four member functions of which one is a constructor:

- __init__() is the constructor
- insert(val)
- traverse(order)
- height(root)

`__init__``()`

Constructor method for BST. This sets the root to None.

Example

```
>>> from BinarySearchTree import *
>>> sample = BinarySearchTree()
>>> print(sample.root)
None
```

`height``(`*root*`)`

This function gives the height of the tree traversing the tree from the root till the farthest leaf.

Parameters
`root` (`BSTNode`) – denotes the starting node

Returns
returns the distance of the node given from its farthest leaf

Return type
int

Example

```
>>> from BinarySearchTree import *
>>> sample = BinarySearchTree()
>>> sample.insert(4)
>>> sample.insert(2)
>>> sample.insert(3)
>>> sample.insert(1)
>>> sample.insert(6)
>>> sample.insert(5)
>>> sample.insert(7)
>>> print(sample.height(sample.root))
2
```

insert(*val*)

Inserts a BST node with info as the value(val) given in the Binary Search Tree.

Parameters
val (*any*) – Data to be stored in the BST, must be of type considering comparison operator

Example

```
>>> from BinarySearchTree import *
>>> sample = BinarySearchTree()
>>> sample.insert(10)
>>> sample.insert(9)
>>> sample.insert(22.5)
>>> print(sample.root.info, sample.root.left.info, sample.root.right.info)
10 9 22.5
>>> print(sample.root.left.left, sample.root.left.right, sample.root.right.left,
↪ sample.root.right.right)
None None None None
```

traverse(*order*)

This prints the tree depending upon the traversal we want.

Parameters
order (*str*) – This is the type of traversal we want for the tree

Example

```
>>> from BinarySearchTree import *
>>> sample = BinarySearchTree()
>>> sample.insert(4)
>>> sample.insert(2)
>>> sample.insert(3)
>>> sample.insert(1)
>>> sample.insert(6)
>>> sample.insert(5)
>>> sample.insert(7)
>>> sample.traverse('PRE')
```

(continues on next page)

(continued from previous page)

```
4 2 1 3 6 5 7
>>> sample.traverse('IN')
1 2 3 4 5 6 7
>>> sample.traverse('POST')
1 3 2 5 7 6 4
```

1.2 DSA module

This module lists the various abstract data types implemented in this project.

The various data type (class) implementations present here are:

- Singly Linked list
- Doubly Linked list
- Binary Search Tree
- Suffix Trie
- Heap

1.3 DoublyLinkedList module

This module contains the implementation of abstract data type Doubly Linked List. To achieve that, a Doubly Linked List node (DoublyLinkedListNode) class has been implemented as well.

class `DoublyLinkedList.`**`DoublyLinkedList`**

Bases: `object`

This is the class implementation of doubly linked list. The objects of this class are doubly linked lists of whom the address of both the head and the tail are known and, each node has the address of both the next and previous nodes.

This class has four member functions of which one is a constructor:

- __init__() is the constructor
- insert(data)
- printer(sep)
- reverse()

`__init__`()

Constructor method for doubly linked list. This sets the head and tail of the list to None.

Example

```
>>> from DoublyLinkedList import *
>>> sample = DoublyLinkedList()
>>> print(sample.head)
None
>>> print(sample.tail)
None
```

`insert`(*data*)

Inserts a new node which contains given data behind the tail of the list.

Parameters

`data` (*`any`*) – It is the information to be stored in the linked list.

Example

```
>>> from DoublyLinkedList import *
>>> sample = DoublyLinkedList()
>>> sample.insert(10)
>>> sample.insert("11")
>>> sample.insert(22.5)
>>> print(sample.head.data, sample.head.next.data, sample.head.prev)
10 11 None
>>> print(sample.tail.data, sample.tail.next, sample.tail.prev.data)
22.5 None 11
```

`printer`(*sep=', '*)

Prints the linked list.

Parameters

`sep` (*`str, optional`*) – This tells us what should be used as a separator for distinct elements while printing the list, defaults to ', '

Example

```
>>> from DoublyLinkedList import *
>>> sample = DoublyLinkedList()
>>> sample.insert(10)
>>> sample.insert("11")
>>> sample.insert(22.5)
>>> sample.printer()
[10, 11, 22.5]
>>> sample.printer('<-> ')
[10<-> 11<-> 22.5]
```

`reverse`()

Reverses the linked list.

Example

```
>>> from DoublyLinkedList import *
>>> sample = DoublyLinkedList()
>>> sample.insert(10)
>>> sample.insert("11")
>>> sample.insert(22.5)
>>> sample.printer()
[10, 11, 22.5]
>>> sample.reverse()
>>> sample.printer()
[22.5, 11, 10]
```

class `DoublyLinkedList.`**`DoublyLinkedListNode`**(*data*)

Bases: `object`

This is the class implementation of node for doubly linked list. This node/object of this class has the property that it contains address of the next as well as the previous node in the list.

This class has one constructor and one convertor:

- __init__(data) is the constructor, and
- __str__() is the convertor.

`__init__`(*data*)

Constructor method for node of Doubly linked list. This sets the data of the of the node to given data and next and prev to None.

Example

```
>>> from DoublyLinkedList import *
>>> sample = DoublyLinkedListNode(10)
>>> print(sample.data)
10
>>> print(sample.next)
None
>>> print(sample.prev)
None
```

`__str__`()

Convertor method for Doubly Linked list node. Returns the string form of data stored at the node.

Example

```
>>> from DoublyLinkedList import *
>>> sample = DoublyLinkedListNode(10)
>>> output = sample.__str__()
>>> print(output)
10
>>> print(type(output))
<class 'str'>
```

1.4 Heap module

This module contains the implementation of abstract data type Heap.

`class Heap.`**`Heap`**`(`*cap*`)`

Bases: `object`

This class is the implementation of the abstract datatype Heap.

This class has eight member functions of which one is the constructor:

- __init__(cap) is the constructor
- parent(i)
- left(i)
- right(i)
- insert(val)
- min()
- Heapify(root)
- deleteMin()

Parameters

- **H** (*List*) – This is the list whose elements are arranged in the form of a heap
- **n** (*int*) – This contains the number of elements in the heap
- **M** (*int*) – This contains the capacity of the heap

`Heapify``(`*root*`)`

Establishes the basic properties of the heap in the heap.

Parameters
root (*int*) – index of the root element of the heap to be heapified

`__init__``(`*cap*`)`

Constructor method for heap. This initialises a list of size cap with all elements as None, sets the value of n to zero and value of M to cap.

Example

```
>>> from Heap import *
>>> sample = Heap(3)
>>> print(sample.H)
[None, None, None]
>>> print(sample.n)
0
>>> print(sample.M)
3
```

`deleteMin``()`

Deletes the minimum element of the heap and heapifies it.

Example

```
>>> from Heap import *
>>> sample = Heap(5)
>>> sample.insert(5)
>>> sample.insert(3)
>>> sample.insert(4)
>>> sample.insert(1)
>>> sample.insert(2)
>>> sample.deleteMin()
>>> print(sample.H)
[2, 3, 4, 5]
```

`insert`(*val*)

Inserts the element with value(= val) in the heap.

Parameters
`val` (*`int`*) – the value to be inserted

Example

```
>>> from Heap import *
>>> sample = Heap(5)
>>> sample.insert(5)
>>> sample.insert(3)
>>> sample.insert(4)
>>> sample.insert(1)
>>> sample.insert(2)
>>> print(sample.H)
[1, 2, 4, 5, 3]
```

`left`(*i*)

This returns the index of the left of the element(in the heap) at the given index.

Parameters
`i` (*`int`*) – it is the index of the element

Returns
Index of the left of the element(in the heap) at the given index

Return type
int

Example

```
>>> from Heap import *
>>> sample = Heap(5)
>>> sample.insert(5)
>>> sample.insert(3)
>>> sample.insert(4)
>>> sample.insert(1)
>>> sample.insert(2)
```

(continues on next page)

(continued from previous page)

```
>>> print(sample.left(0), sample.left(1))
1 3
```

min()

Returns the minimum element of the heap.

Returns
The minimum element of the heap

Return type
int

Example

```
>>> from Heap import *
>>> sample = Heap(5)
>>> sample.insert(5)
>>> sample.insert(3)
>>> sample.insert(4)
>>> sample.insert(1)
>>> sample.insert(2)
>>> print(sample.min())
1
```

parent(*i*)

This returns the index of the parent of the element(in the heap) at the given index.

Parameters
i (*int*) – it is the index of the element whose parent's index is required to be returned

Returns
Index of the parent of the element(in the heap) at the given index

Return type
int

Example

```
>>> from Heap import *
>>> sample = Heap(5)
>>> sample.insert(5)
>>> sample.insert(3)
>>> sample.insert(4)
>>> sample.insert(1)
>>> sample.insert(2)
>>> print(sample.parent(1), sample.parent(2), sample.parent(3), sample.
↪parent(4))
0 0 1 1
```

right(*i*)

This returns the index of the right of the element(in the heap) at the given index.

Parameters
i (*int*) – it is the index of the element

Returns
Index of the right of the element(in the heap) at the given index

Return type
int

Example

```
>>> from Heap import *
>>> sample = Heap(5)
>>> sample.insert(5)
>>> sample.insert(3)
>>> sample.insert(4)
>>> sample.insert(1)
>>> sample.insert(2)
>>> print(sample.right(0), sample.right(1))
2 4
```

1.5 SinglyLinkedList module

This module contains the implementation of abstract data type Singly Linked List. To achieve that, a Singly Linked List node (SinglyLinkedListNode) class has been implemented as well. This module also contains a merge function useful for merging Singly linked lists.

class `SinglyLinkedList.`**`SinglyLinkedList`**

Bases: `object`

This is the class implementation of singly linked list. The object of this class is a singly linked list of which address of only the head is known and each node has the address of the next node.

This class has six member functions of which one is a constructor:

- __init__() is the constructor
- insert(data)
- find(data)
- deleteVal(data)
- printer(sep)
- reverse()

`__init__`()

Constructor method for Singly Linked list. This sets the the head and tail of the list to None.

Example

```
>>> from SinglyLinkedList import *
>>> sample = SinglyLinkedList()
>>> print(sample.head)
None
>>> print(sample.tail)
None
```

`deleteVal`(*data*)

Deletes the node containing data if found.

Parameters
`data` (*any*) – Data to be erased from the linked list.

Returns
returns True if deleted successfully, else returns False if data not found in the list.

Return type
bool

Example

```
>>> from SinglyLinkedList import *
>>> sample = SinglyLinkedList()
>>> sample.insert(10)
>>> sample.insert("11")
>>> sample.insert(22.5)
>>> deleted = sample.deleteVal("11")
>>> print(deleted)
True
>>> deletedTwice = sample.deleteVal("11")
>>> print(deletedTwice)
False
```

`find`(*data*)

Finds the node with the data stored in it and returns the node previous to/in front of it.

Parameters
`data` (*any*) – It is the information we want to find in the linked list.

Returns
previous node of the node which has the data.

Return type
SinglyLinkedListNode

Example

```
>>> from SinglyLinkedList import *
>>> sample = SinglyLinkedList()
>>> sample.insert(10)
>>> sample.insert("11")
>>> sample.insert(22.5)
>>> mynode = sample.find(22.5)
>>> print(mynode.data)
11
>>> print(mynode.next.data)
22.5
```

insert(*data*)

Inserts a new node which contains given data behind the tail of the list.

Parameters

data (*any*) – It is the information to be stored in the linked list.

Example

```
>>> from SinglyLinkedList import *
>>> sample = SinglyLinkedList()
>>> sample.insert(10)
>>> print(sample.head.data)
10
>>> print(sample.tail.data)
10
>>> sample.insert("11")
>>> print(sample.head.data)
10
>>> print(sample.tail.data)
11
```

printer(*sep=', '*)

Prints the linked list.

Parameters

sep (`str, optional`) – This tells us what should be used as a separator for distinct elements while printing the list, defaults to ', '

Example

```
>>> from SinglyLinkedList import *
>>> sample = SinglyLinkedList()
>>> sample.insert(10)
>>> sample.insert("11")
>>> sample.insert(22.5)
>>> sample.printer()
[10, 11, 22.5]
>>> sample.printer('-> ')
[10-> 11-> 22.5]
```

reverse()

Reverses the linked list.

Example

```
>>> from SinglyLinkedList import *
>>> sample = SinglyLinkedList()
>>> sample.insert(10)
>>> sample.insert("11")
>>> sample.insert(22.5)
>>> sample.printer()
[10, 11, 22.5]
>>> sample.reverse()
>>> sample.printer()
[22.5, 11, 10]
```

class SinglyLinkedList.**SinglyLinkedListNode**(*data*)

Bases: `object`

This is the class implementation of node for singly linked list. This node/object of this class has the property that it contains address of the next node in the list.

This class has one constructor and one convertor:

- __init__(data) is the constructor, and
- __str__() is the convertor.

__init__(*data*)

Constructor method for Singly linked list nodes. This sets the data to given data and next to none.

Parameters
data (*any*) – this is the value we want the node to contain

Example

```
>>> from SinglyLinkedList import *
>>> sample = SinglyLinkedListNode(10)
>>> print(sample.data)
10
```

__str__()

Convertor method for Singly linked list nodes. This returns the string form of the data stored in the node.

Returns
string form of data

Return type
str

Example

```
>>> from SinglyLinkedList import *
>>> sample = SinglyLinkedListNode(10)
>>> output = sample.__str__()
>>> print(output)
10
>>> print(type(output))
<class 'str'>
```

`SinglyLinkedList.`**`merge`**`(`*list1*, *list2*`)`

Merges two Singly Linked Lists into a single Singly Linked List.

Parameters

- **`list1`** (`SinglyLinkedList`) – Singly linked list 1
- **`list2`** (`SinglyLinkedList`) – SinglyLinkedList 2

Returns

returns a singly linked list which is a combination of list1 and list2

Return type

SinglyLinkedList

Example

```
>>> from SinglyLinkedList import *
>>> sample1 = SinglyLinkedList()
>>> sample2 = SinglyLinkedList()
>>> sample1.insert(9)
>>> sample1.insert(10)
>>> sample1.insert(11)
>>> sample2.insert(12)
>>> sample2.insert(13)
>>> sample2.insert(14)
>>> sample = merge(sample1, sample2)
>>> sample.printer()
[9, 10, 11, 12, 13, 14]
```

1.6 Trie module

This module contains the implementation of abstract data type Suffix Trie.

`class Trie.`**`Trie`**

Bases: `object`

This is the class implementation of a Suffix Trie. An instance of this class is a Trie which is used for the storage of various substrings of a string in the form of a tree with nodes where each node stores a character of the string.

This class has five member functions one of which is a constructor:

- __init__() is the constructor
- find(root, c)

- insert(s)
- checkPrefix(s)
- countPrefix(s)

__init__()

Constructor method for Suffix Trie. This sets the class variable T to an empty dictionary.

Example

```
>>> from Trie import *
>>> sample = Trie()
>>> print(sample.T)
{}
>>> print(type(sample.T))
<class 'dict'>
```

checkPrefix(*s*)

Checks if the given string s is present as prefix in the suffix trie.

Parameters
s (*str*) – string to be checked

Returns
True if found, else return False

Return type
bool

Example

```
>>> from Trie import *
>>> sample = Trie()
>>> sample.insert('Stunning')
>>> print(sample.checkPrefix('Stun'))
True
```

countPrefix(*s*)

Counts the number of prefixes of the string s in the Suffix Trie.

Parameters
s (*str*) – The string whose prefix count is required

Returns
prefix count of s

Return type
int

Example

```
>>> from Trie import *
>>> sample = Trie()
>>> sample.insert('Vaibhav')
>>> sample.insert('Vaibhavi')
>>> sample.insert('Vaishnavi')
>>> sample.insert('Vishal')
>>> sample.insert('Raman')
>>> print(sample.countPrefix('V'), sample.countPrefix('Vai'), sample.
↪countPrefix('Vaibhav'))
4 3 2
>>> print(sample.countPrefix('R'), sample.countPrefix('Tarzan'))
1 0
```

find(*root*, *c*)

Finds the element c(character) in the heap and returns its position?.

Parameters

- **root** (*dict*) – It is the dictionary in which character c is to be searched
- **c** (*char*) – Character to be searched in the dictionary

Returns
returns True if found, else returns False

Return type
bool

Example

```
>>> from Trie import *
>>> sample = Trie()
>>> sample.insert('sample')
>>> print(sample.find(sample.T, 's'))
True
```

insert(*s*)

Inserts a given string into the Trie.

Parameters
s (*str*) – string to be inserted

Example

```
>>> from Trie import *
>>> sample = Trie()
>>> sample.insert('Tree')
>>> print(sample.find(sample.T, 'T'), sample.find(sample.T['T'], 'r'), sample.
↪find(sample.T['T']['r'], 'e'), sample.find(sample.T['T']['r']['e'], 'e'))
True True True True
>>> print(sample.T)
{'T': {'#': 1, 'r': {'#': 1, 'e': {'#': 1, 'e': {'#': 1}}}}}
```

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX