My Project

Generated by Doxygen 1.9.6

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

Class Documentation

3.1 BinarySearchTree Class Reference

Binary Search Tree Data Structure.

`#include <DSA.h>`

Public Types

- enum order { PRE , IN , POST }
  Order of Traversal in BST.

Public Member Functions

- BinarySearchTree ()
  Default Constructor.
- void insert (ll val)
  Insert a New Node in BST with a given Value.
- void traverse (BSTNode *T, order tt)
  Print the Values stored in BST in a given Traversal Order.
- ll height (BSTNode *T)
  Function to get the Height of a given Node. Height here is defined as the longest distance of a Node to a leaf.

Public Attributes

- BSTNode * **root**
  Root Node of BST.

3.1.1 Detailed Description

Binary Search Tree Data Structure.

BST is a Binary Tree Data Structure in which -

1. Left Subtree of each Node stores Nodes with values lesser than the Node
2. Right Subtree of each Node stores Nodes with values greater than the Node
3. Left and Right Subtree each must also be a Binary Search Tree

3.1.2 Member Enumeration Documentation

3.1.2.1 order

```
enum BinarySearchTree::order
```

Order of Traversal in BST.

Enumerator

PRE	Current Node, left, right.
IN	left, Current Node, right
POST	left, right, Current Node

3.1.3 Constructor & Destructor Documentation

3.1.3.1 BinarySearchTree()

```
BinarySearchTree::BinarySearchTree ( )
```

Default Constructor.

Initializes root to NULL

3.1.4 Member Function Documentation

3.1.4.1 height()

```
ll BinarySearchTree::height (
            BSTNode * T )
```

Function to get the Height of a given Node. Height here is defined as the longest distance of a Node to a leaf.

Parameters

T	Node whose height is to be found

Returns

Height of the given Node

3.1.4.2 insert()

```
void BinarySearchTree::insert (
            ll val )
```

Insert a New Node in BST with a given Value.

Parameters

val	Value to be stored in New Node

3.1.4.3 traverse()

```
void BinarySearchTree::traverse (
            BSTNode * T,
            order tt )
```

Print the Values stored in BST in a given Traversal Order.

Parameters

T	Node starting from which the Tree is to be printed
tt	Order of Traveral of Tree. It can be - 1. Pre Order 2. In Order 3. Post Order

The documentation for this class was generated from the following files:

- DSA.h
- DSA.cpp

3.2 BSTNode Class Reference

Node of a Binary Search Tree.

```
#include <DSA.h>
```

Public Member Functions

- BSTNode (ll val)

  Construct a new BSTNode object with a given value stored in it.

Public Attributes

- ll **info**

  Data stored in the Node.
- ll **level**

  Height of the Node.
- BSTNode * **left**

  Pointer to the Left Child of the Node.
- BSTNode * **right**

  Pointer to the Right Child of the Node.

3.2.1 Detailed Description

Node of a Binary Search Tree.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 BSTNode()

```
BSTNode::BSTNode (
            ll val )
```

Construct a new BSTNode object with a given value stored in it.

Parameters

val	Data to be stored in the BSTNode

The documentation for this class was generated from the following files:

- DSA.h
- DSA.cpp

3.3 DoublyLinkedList Class Reference

Doubly Linked List Data Structure.

```
#include <DSA.h>
```

Public Member Functions

- DoublyLinkedList ()

  Construct a new Doubly Linked List object.
- void insert (ll data)

  Insert new Node with a given data into the Linked List.
- void printer (string sep)

  Function to print the Linked List with a given separation between the Data stored in each Node.
- void reverse ()

  Reverse the Order of Elements in the Linked List.

Public Attributes

- DoublyLinkedListNode * **head**

  Head Node i.e. The Node at starting Index.
- DoublyLinkedListNode * **tail**

  Tail Node i.e. The Node at last Index.

3.3.1 Detailed Description

Doubly Linked List Data Structure.

A Doubly Linked List (DLL) contains an extra pointer, typically called the previous pointer, together with the next pointer and data which are there in the singly linked list.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 DoublyLinkedList()

```
DoublyLinkedList::DoublyLinkedList ( )
```

Construct a new Doubly Linked List object.

Sets head and tail to NULL

3.3.3 Member Function Documentation

3.3.3.1 insert()

```
void DoublyLinkedList::insert (
            ll data )
```

Insert new Node with a given data into the Linked List.

Parameters

data	Data to be stored into the New Node

3.3.3.2 printer()

```
void DoublyLinkedList::printer (
            string sep = ", " )
```

Function to print the Linked List with a given separation between the Data stored in each Node.

Parameters

in	*sep*	Separation to be used between Datam of each node while printing the linked list

3.3.3.3 reverse()

```
void DoublyLinkedList::reverse ( )
```

Reverse the Order of Elements in the Linked List.

The documentation for this class was generated from the following files:

- DSA.h
- DSA.cpp

3.4 DoublyLinkedListNode Class Reference

Node of a Doubly Linked List.

```
#include <DSA.h>
```

Public Member Functions

- DoublyLinkedListNode ()

  Default Contructor.
- DoublyLinkedListNode (ll val)

  Construct a new Doubly Linked List Node object with a given value to be stored.

Public Attributes

- ll **data**

  Data stored in the Node.
- DoublyLinkedListNode ∗ **next**

  Pointer to Next Node.
- DoublyLinkedListNode ∗ **prev**

  Pointer to Previous Node.

3.4.1 Detailed Description

Node of a Doubly Linked List.

It Stores an additional pointer as compared to a Singly Linked List Node, mainly to point to Previous Node as well

3.4.2 Constructor & Destructor Documentation

3.4.2.1 DoublyLinkedListNode() [1/2]

```
DoublyLinkedListNode::DoublyLinkedListNode ( )
```

Default Contructor.

It initializes :

- data to -1
- next to NULL
- prev to NULL

3.4.2.2 DoublyLinkedListNode() [2/2]

```
DoublyLinkedListNode::DoublyLinkedListNode (
            ll val )
```

Construct a new Doubly Linked List Node object with a given value to be stored.

Parameters

val	The value to be stored in the Node It initializes : • data to val • next to NULL • prev to NULL

The documentation for this class was generated from the following files:

- DSA.h
- DSA.cpp

3.5 Heap Class Reference

This class implements the Heap Data Structure.

`#include <DSA.h>`

Public Member Functions

- Heap (int cap)

  Construct a new Heap object.
- int parent (int i)

  Function to get the Parent Node of an element in Heap.
- int left (int i)

  Function to get the Left child of a Node in Heap.
- int right (int i)

  Function to get the Right child of a Node in Heap.
- void insert (int val)

  Inserts a new Node into the Heap with a given value.
- int min ()

  Function to get the Minimum value stored in the Heap.
- void Heapify (int root)

  Heapify is a process of creating a Heap from a list of elements or Restoring the Heap property if it is violated at any Node. This function implements Heapify on the present Heap from a given node.
- void deleteMin ()

  Deletes the Minimum Element and Heapifies the heap after deleting the minimum element.

Public Attributes

- int **n**

  Number of elements in current Heap.
- int **Cap**

  Maximum Capacity of Heap.
- int * **arr**

  Array in which Heap elements are stored.

3.5.1 Detailed Description

This class implements the Heap Data Structure.

A Heap is a special Tree-based data structure in which the tree is a Binary Tree that stores priorities (or priority-element) pairs at nodes.

3.5.2 Constructor & Destructor Documentation

3.5.2.1 Heap()

```
Heap::Heap (
            int cap )
```

Construct a new Heap object.

Parameters

cap	The Maximum Capacity of Heap

3.5.3 Member Function Documentation

3.5.3.1 deleteMin()

```
void Heap::deleteMin ( )
```

Deletes the Minimum Element and Heapifies the heap after deleting the minimum element.

3.5.3.2 Heapify()

```
void Heap::Heapify (
            int root )
```

Heapify is a process of creating a Heap from a list of elements or Restoring the Heap property if it is violated at any Node. This function implements Heapify on the present Heap from a given node.

Parameters

root	Index of Node, starting from which Heap will be made

3.5.3.3 insert()

```
void Heap::insert (
            int val )
```

Inserts a new Node into the Heap with a given value.

Parameters

val	The value to be stored in the New Node

3.5.3.4 left()

```
int Heap::left (
            int i )
```

Function to get the Left child of a Node in Heap.

Parameters

i	Index of element whose Left child is to be found

Returns

Index of the Left child

3.5.3.5 min()

```
int Heap::min ( )
```

Function to get the Minimum value stored in the Heap.

Returns

Minimum value stored in Heap

3.5.3.6 parent()

```
int Heap::parent (
            int i )
```

Function to get the Parent Node of an element in Heap.

Parameters

i	Index of element whose Parent Node is to be found

Returns

Index of the Parent Node

3.5.3.7 right()

```
int Heap::right (
            int i )
```

Function to get the Right child of a Node in Heap.

Parameters

i	Index of element whose Right child is to be found

Returns

Index of the Right child

The documentation for this class was generated from the following files:

- DSA.h
- DSA.cpp

3.6 SinglyLinkedList Class Reference

Singly Linked List Data Structure.

`#include <DSA.h>`

Public Member Functions

- **SinglyLinkedList** ()
  Construtor of a Singly Linked List.
- void insert (ll data)
  Function to insert a New Node into the Linked List.
- SinglyLinkedListNode * find (ll data)
  Find Previous Node of a Node with a given data.
- bool deleteVal (ll data)
  Delete the Node with given data.
- void printer (string sep)
  Function to print the Linked List with a given separation between the Data stored in each Node.
- void reverse ()
  Reverse the Order of Elements in the Linked List.

Public Attributes

- SinglyLinkedListNode * **head**

  Head Node of the Linked List.
- SinglyLinkedListNode * **tail**

  Tail Node of the Linked List.

3.6.1 Detailed Description

Singly Linked List Data Structure.

It is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type.

3.6.2 Member Function Documentation

3.6.2.1 deleteVal()

```
bool SinglyLinkedList::deleteVal (
            ll data )
```

Delete the Node with given data.

Parameters

in	*data*	data stored in the Node which is to be deleted

Returns

True - If data found in the Linked List and deleted

False - If data given is not found in the Linked List

3.6.2.2 find()

```
SinglyLinkedListNode * SinglyLinkedList::find (
            ll data )
```

Find Previous Node of a Node with a given data.

Parameters

in	*data*	value of data to be found

Returns

Previous Node of the Node with given data (if found), else returns NULL

3.6.2.3 insert()

```
void SinglyLinkedList::insert (
            ll data )
```

Function to insert a New Node into the Linked List.

Parameters

in	*data*	value of Data to be stored in the new Node Inserted

3.6.2.4 printer()

```
void SinglyLinkedList::printer (
            string sep = ", " )
```

Function to print the Linked List with a given separation between the Data stored in each Node.

Parameters

in	*sep*	Separation to be used between Datam of each node while printing the linked list

3.6.2.5 reverse()

```
void SinglyLinkedList::reverse ( )
```

Reverse the Order of Elements in the Linked List.

The documentation for this class was generated from the following files:

- DSA.h
- DSA.cpp

3.7 SinglyLinkedListNode Class Reference

Node of a Singly Linked List. Stores Data stored in a Node and a pointer to its next Node.

```
#include <DSA.h>
```

Public Member Functions

- **SinglyLinkedListNode** ()

  Default Constructor.
- SinglyLinkedListNode (ll val)

  Constructor with a parameter.

Public Attributes

- ll **data**

  Data stored in a Node.
- SinglyLinkedListNode * **next**

  Pointer to next Node.

3.7.1 Detailed Description

Node of a Singly Linked List. Stores Data stored in a Node and a pointer to its next Node.

3.7.2 Constructor & Destructor Documentation

3.7.2.1 SinglyLinkedListNode()

```
SinglyLinkedListNode::SinglyLinkedListNode (
            ll val )
```

Constructor with a parameter.

Parameters

val	value of data to be stored in the Node

The documentation for this class was generated from the following files:

- DSA.h
- DSA.cpp

3.8 Trie Class Reference

The Trie Data Structure.

```
#include <DSA.h>
```

Public Member Functions

- **Trie** ()

  Default Constructor to construct a new Trie object.
- bool find (Trie *T, char c)

  Find a particular character in Trie.
- void insert (string s)

  Insert a new word into Trie.
- bool checkPrefix (string s)

  Function to check if a given prefix is present in the Trie.
- ll countPrefix (string s)

  Function to count number of Prefix in Trie of a string.

Public Attributes

- ll count

  Number of total Nodes in Trie.
- map< char, Trie * > **nodes**

  Map of Trie Nodes and characters, in which a character is mapped to a Trie Pointer.

3.8.1 Detailed Description

The Trie Data Structure.

A Trie is used for text processing. It is an efficient information retrieval data structure which can store strings and search them optimally

3.8.2 Member Function Documentation

3.8.2.1 checkPrefix()

```
bool Trie::checkPrefix (
            string s )
```

Function to check if a given prefix is present in the Trie.

Parameters

s	The prefix which is to be searched in the Trie

Returns

True - If the prefix is found

False - If the prefix is not found

3.8.2.2 countPrefix()

```
ll Trie::countPrefix (
            string s )
```

Function to count number of Prefix in Trie of a string.

Parameters

s	The string whose counts of prefix is to be counted

Returns

The number of counts of Prefix

3.8.2.3 find()

```
bool Trie::find (
            Trie * T,
            char c )
```

Find a particular character in Trie.

Parameters

in	*T*	Trie Node starting from which it will start finding
in	*c*	character which is to be found

Returns

True - If the character is found

False - If the character is not found

3.8.2.4 insert()

```
void Trie::insert (
            string s )
```

Insert a new word into Trie.

Parameters

s	The string of word that is to be inserted

3.8.3 Member Data Documentation

3.8.3.1 count

```
ll Trie::count
```

Number of total Nodes in Trie.

The documentation for this class was generated from the following files:

- DSA.h
- DSA.cpp

Chapter 4

File Documentation

4.1 DSA.h File Reference

Header file for Several Data Structures.

```
#include <bits/stdc++.h>
```

Classes

- class SinglyLinkedListNode

  Node of a Singly Linked List. Stores Data stored in a Node and a pointer to its next Node.
- class SinglyLinkedList

  Singly Linked List Data Structure.
- class DoublyLinkedListNode

  Node of a Doubly Linked List.
- class DoublyLinkedList

  Doubly Linked List Data Structure.
- class BSTNode

  Node of a Binary Search Tree.
- class BinarySearchTree

  Binary Search Tree Data Structure.
- class Trie

  The Trie Data Structure.
- class Heap

  This class implements the Heap Data Structure.

Macros

- #define **ll** long long int
- #define **vi** vector<int>
- #define **vll** vector<ll>

Functions

- ostream & operator<< (ostream &out, const SinglyLinkedListNode &node)

  Overloaded << operator to print the data of the Singly Linked List Node provided as arguement into a file.
- SinglyLinkedList merge (SinglyLinkedList list1, SinglyLinkedList list2)

  Function to Merge two Singly Linked Lists into one Singly Linked List.
- ostream & operator<< (ostream &out, const DoublyLinkedListNode &node)

  Overloaded << operator to print the data of the Doubly Linked List Node provided as arguement into a file.
- ostream & operator<< (ostream &out, const BSTNode &node)

  Overloaded << operator to print the data of the Binary Search Tree Node provided as arguement into a file.

4.1.1 Detailed Description

Header file for Several Data Structures.

Author

Atishay Jain (atishay@cse.iitb.ac.in)

Version

0.1

Date

2022-09-25

Copyright

4.1.2 Function Documentation

4.1.2.1 merge()

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

Function to Merge two Singly Linked Lists into one Singly Linked List.

Parameters

list1	First Linked List to be merged
list2	Second Linked List to be merged

Returns

SinglyLinkedList - Merged Singly Linked List

4.1.2.2 operator<<() [1/3]

```
ostream & operator<< (
            ostream & out,
            const BSTNode & node )
```

Overloaded << operator to print the data of the Binary Search Tree Node provided as arguement into a file.

Parameters

out	File in which the Node data is printed
node	The BSTNode whose data is to be printed

Returns

ostream& The File with the Node's data printed in it

4.1.2.3 operator<<() [2/3]

```
ostream & operator<< (
            ostream & out,
            const DoublyLinkedListNode & node )
```

Overloaded << operator to print the data of the Doubly Linked List Node provided as arguement into a file.

Parameters

out	File in which the Node data is printed
node	The Doubly Linked List Node whose data is to be printed

Returns

ostream& The File with the Node's data printed in it

4.1.2.4 operator<<() [3/3]

```
ostream & operator<< (
            ostream & out,
            const SinglyLinkedListNode & node )
```

Overloaded << operator to print the data of the Singly Linked List Node provided as arguement into a file.

Parameters

out	File in which the Node data is printed
node	The Singly Linked List Node whose data is to be printed

Returns

ostream& The File with the Node's data printed in it

4.2 DSA.h

Go to the documentation of this file.

```
1
12 #include <bits/stdc++.h>
13 #define ll long long int
14 #define vi vector<int>
15 #define vll vector<ll>
16 using namespace std;
17
20 class SinglyLinkedListNode {
21
22     public:
23
25         ll data;
27         SinglyLinkedListNode* next;
28
30         SinglyLinkedListNode();
31
34         SinglyLinkedListNode(ll val);
35
36 };
37
42 ostream& operator<<(ostream &out, const SinglyLinkedListNode &node);
43
52 class SinglyLinkedList {
53
54     public:
55
57         SinglyLinkedListNode *head, *tail;
59
61         SinglyLinkedList();
62
68         void insert (ll data);
69
76         SinglyLinkedListNode* find (ll data);
77
85         bool deleteVal (ll data);
86
94         void printer (string sep);
95
100         void reverse ();
101
102 };
103
111 SinglyLinkedList merge (SinglyLinkedList list1, SinglyLinkedList list2);
112
113
114 /* ------------------------------ Doubly Linked List ---------------------------- */
115
122 class DoublyLinkedListNode {
123
124     public:
125
127         ll data;
129         DoublyLinkedListNode *next, *prev;
131
137         DoublyLinkedListNode();
138
148         DoublyLinkedListNode(ll val);
149
150 };
151
159 ostream& operator<<(ostream &out, const DoublyLinkedListNode &node);
160
170 class DoublyLinkedList {
171
```

```
172     public:
173
175         DoublyLinkedListNode *head, *tail;
177
183         DoublyLinkedList ();
184
190         void insert (ll data);
191
199         void printer (string sep);
200
205         void reverse ();
206
207 };
208
209 /* ------------------------------ Binary Search Tree ---------------------------- */
210
215 class BSTNode {
216
217     public:
218
220         ll info, level;
221
223         BSTNode *left, *right;
224
230         BSTNode (ll val);
231
232 };
233
241 ostream& operator«(ostream &out, const BSTNode &node);
242
253 class BinarySearchTree {
254
255     public:
256
258         BSTNode *root;
259
261         enum order {
262             PRE
263             , IN
264             , POST
265         };
266
269         BinarySearchTree ();
270
276         void insert(ll val);
277
287         void traverse (BSTNode* T, order tt);
288
297         ll height(BSTNode *T);
298
299 };
300
301 /* ---------------------------------- Suffix Trie ------------------------------- */
302
311 class Trie {
312
313     public:
314
319         ll count;
320
323         map<char,Trie*> nodes;
324
328         Trie ();
329
338         bool find(Trie* T, char c);
339
345         void insert(string s);
346
354         bool checkPrefix(string s);
355
362         ll countPrefix(string s);
363
364 };
365
366 /*---------------------------------- Heap ------------------------------------- */
367
377 class Heap {
378
379     public:
380
382         int n;
384         int Cap;
386         int *arr;
387
393         Heap(int cap);
394
398         int parent(int i);
```

```
399
403         int left(int i);
404
408         int right(int i);
409
415         void insert(int val);
416
422         int min();
423
431         void Heapify(int root);
432
438         void deleteMin();
439
440 };
441
442
```

Index