

## My Project

Generated by Doxygen 1.9.6



<b>1 Class Index</b>	<b>1</b>
1.1 Class List	1
<b>2 File Index</b>	<b>3</b>
2.1 File List	3
<b>3 Class Documentation</b>	<b>5</b>
3.1 BinarySearchTree Class Reference	5
3.1.1 Detailed Description	5
3.1.2 Constructor & Destructor Documentation	5
3.1.2.1 BinarySearchTree()	6
3.2 BSTNode Class Reference	6
3.2.1 Detailed Description	6
3.2.2 Constructor & Destructor Documentation	6
3.2.2.1 BSTNode()	6
3.3 DoublyLinkedList Class Reference	7
3.3.1 Detailed Description	7
3.3.2 Constructor & Destructor Documentation	7
3.3.2.1 DoublyLinkedList()	7
3.4 DoublyLinkedListNode Class Reference	7
3.4.1 Detailed Description	8
3.4.2 Constructor & Destructor Documentation	8
3.4.2.1 DoublyLinkedListNode() [1/2]	8
3.4.2.2 DoublyLinkedListNode() [2/2]	8
3.5 SinglyLinkedList Class Reference	8
3.5.1 Detailed Description	9
3.5.2 Constructor & Destructor Documentation	9
3.5.2.1 SinglyLinkedList()	9
3.5.3 Member Function Documentation	10
3.5.3.1 deleteVal()	10
3.5.3.2 find()	10
3.5.3.3 insert()	10
3.5.3.4 printer()	10
3.5.3.5 reverse()	11
3.6 SinglyLinkedListNode Class Reference	11
3.6.1 Detailed Description	11
3.6.2 Constructor & Destructor Documentation	11
3.6.2.1 SinglyLinkedListNode() [1/2]	11
3.6.2.2 SinglyLinkedListNode() [2/2]	12
3.7 Trie Class Reference	12
3.7.1 Detailed Description	12
3.7.2 Constructor & Destructor Documentation	13
3.7.2.1 Trie()	13

<b>4 File Documentation</b>	<b>15</b>
4.1 DSA.h File Reference . . . . .	15
4.1.1 Detailed Description . . . . .	16
4.2 DSA.h . . . . .	16
<b>Index</b>	<b>21</b>

# Chapter 1

## Class Index

### 1.1 Class List

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

<a href="#">BinarySearchTree</a>	
Brief Description of <a href="#">BinarySearchTree</a> . Brief Description continued . . . . .	5
<a href="#">BSTNode</a>	
Brief Description of <a href="#">BSTNode</a> . Brief Description continued . . . . .	6
<a href="#">DoublyLinkedList</a>	
Brief Description of <a href="#">DoublyLinkedList</a> . Brief Description continued . . . . .	7
<a href="#">DoublyLinkedListNode</a>	
Brief Description of <a href="#">DoublyLinkedListNode</a> . Brief Description continued . . . . .	7
<a href="#">SinglyLinkedList</a>	
Brief Description about <a href="#">SinglyLinkedList</a> . . . . .	8
<a href="#">SinglyLinkedListNode</a>	
Brief Description of <a href="#">SinglyLinkedListNode</a> . Brief Description continued . . . . .	11
<a href="#">Trie</a>	
Brief Description of <a href="#">Trie</a> . Brief Description continued . . . . .	12



## Chapter 2

# File Index

### 2.1 File List

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

<a href="#">DSA.h</a>	It is about various Linked List Nodes . . . . .	15
-----------------------	---	----





## Chapter 3

# Class Documentation

### 3.1 BinarySearchTree Class Reference

Brief Description of [BinarySearchTree](#). Brief Description continued.

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

#### Public Types

- enum **order** { **PRE** , **IN** , **POST** }

#### Public Member Functions

- [BinarySearchTree](#) ()
- void **insert** (ll val)
- void **traverse** ([BSTNode](#) \*T, order tt)
- ll **height** ([BSTNode](#) \*T)

#### Public Attributes

- [BSTNode](#) \* **root**

#### 3.1.1 Detailed Description

Brief Description of [BinarySearchTree](#). Brief Description continued.

Detailed description starts here.

#### 3.1.2 Constructor & Destructor Documentation

### 3.1.2.1 BinarySearchTree()

```
BinarySearchTree::BinarySearchTree ( ) [inline]
```

This is a constructor

[BinarySearchTree](#) Functions- insert, traverse, height

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

- [DSA.h](#)

## 3.2 BSTNode Class Reference

Brief Description of [BSTNode](#). Brief Description continued.

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

### Public Member Functions

- [BSTNode](#) (ll val)

### Public Attributes

- ll **info**
- ll **level**
- [BSTNode](#) \* **left**
- [BSTNode](#) \* **right**

### 3.2.1 Detailed Description

Brief Description of [BSTNode](#). Brief Description continued.

Detailed description starts here.

### 3.2.2 Constructor & Destructor Documentation

#### 3.2.2.1 BSTNode()

```
BSTNode::BSTNode (
    ll val ) [inline]
```

This is a constructor

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

- [DSA.h](#)

## 3.3 DoublyLinkedList Class Reference

Brief Description of [DoublyLinkedList](#). Brief Description continued.

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

### Public Member Functions

- [DoublyLinkedList](#) ()
- void **insert** (ll data)
- void **printer** (string sep=", ")
- void **reverse** ()

### Public Attributes

- [DoublyLinkedListNode](#) \* **head**
- [DoublyLinkedListNode](#) \* **tail**

#### 3.3.1 Detailed Description

Brief Description of [DoublyLinkedList](#). Brief Description continued.

Detailed description starts here.

#### 3.3.2 Constructor & Destructor Documentation

##### 3.3.2.1 DoublyLinkedList()

```
DoublyLinkedList::DoublyLinkedList ( ) [inline]
```

This is a constructor

[DoublyLinkedListNode](#) functions- insert, printer, reverse

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

- [DSA.h](#)

## 3.4 DoublyLinkedListNode Class Reference

Brief Description of [DoublyLinkedListNode](#). Brief Description continued.

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

## Public Member Functions

- [DoublyLinkedListNode](#) ()
- [DoublyLinkedListNode](#) (ll val)

## Public Attributes

- ll **data**
- [DoublyLinkedListNode](#) \* **next**
- [DoublyLinkedListNode](#) \* **prev**

### 3.4.1 Detailed Description

Brief Description of [DoublyLinkedListNode](#). Brief Description continued.

Detailed description starts here.

### 3.4.2 Constructor & Destructor Documentation

#### 3.4.2.1 [DoublyLinkedListNode](#)() [1/2]

```
DoublyLinkedListNode::DoublyLinkedListNode ( ) [inline]
```

This is a constructor

#### 3.4.2.2 [DoublyLinkedListNode](#)() [2/2]

```
DoublyLinkedListNode::DoublyLinkedListNode (
    ll val ) [inline]
```

This is a constructor

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

- [DSA.h](#)

## 3.5 SinglyLinkedList Class Reference

Brief Description about [SinglyLinkedList](#).

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

## Public Member Functions

- [SinglyLinkedList](#) ()  
*This is about Singly Linked List.*
- void [insert](#) (ll data)
- [SinglyLinkedListNode](#) \* [find](#) (ll data)
- bool [deleteVal](#) (ll data)
- void [printer](#) (string sep=", ")
- void [reverse](#) ()

## Public Attributes

- [SinglyLinkedListNode](#) \* **head**
- [SinglyLinkedListNode](#) \* **tail**

### 3.5.1 Detailed Description

Brief Description about [SinglyLinkedList](#).

Brief Description continued.

### 3.5.2 Constructor & Destructor Documentation

#### 3.5.2.1 SinglyLinkedList()

```
SinglyLinkedList::SinglyLinkedList ( ) [inline]
```

This is about Singly Linked List.

#### Parameters

out	<i>head</i>	
-----	-------------	--

This is constructor w/o parameter

head - One of the variables used

tail - Another variable used

data - Another variable used

ptr - Another variable used

Functions- insert,find, deleteVal, printer, reverse

### 3.5.3 Member Function Documentation

#### 3.5.3.1 deleteVal()

```
bool SinglyLinkedList::deleteVal (
    ll data ) [inline]
```

##### Parameters

in	<i>data</i>	
----	-------------	--

This is member function deleteVal

#### 3.5.3.2 find()

```
SinglyLinkedListNode * SinglyLinkedList::find (
    ll data ) [inline]
```

##### Parameters

in	<i>data</i>	
out	<i>prev</i>	

This is member function

#### 3.5.3.3 insert()

```
void SinglyLinkedList::insert (
    ll data ) [inline]
```

##### Parameters

in	<i>data</i>	
----	-------------	--

This is member function insert

#### 3.5.3.4 printer()

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

This is member function printer

### 3.5.3.5 reverse()

```
void SinglyLinkedList::reverse ( ) [inline]
```

This is member function reverse

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

- [DSA.h](#)

## 3.6 SinglyLinkedListNode Class Reference

Brief Description of [SinglyLinkedListNode](#). Brief Description continued.

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

### Public Member Functions

- [SinglyLinkedListNode](#) ()
- [SinglyLinkedListNode](#) (ll val)

### Public Attributes

- ll **data**  
*This is about Singly Linked List Node.*
- [SinglyLinkedListNode](#) \* **next**

### 3.6.1 Detailed Description

Brief Description of [SinglyLinkedListNode](#). Brief Description continued.

Detailed description starts here.

### 3.6.2 Constructor & Destructor Documentation

#### 3.6.2.1 SinglyLinkedListNode() [1/2]

```
SinglyLinkedListNode::SinglyLinkedListNode ( ) [inline]
```

Parameters

out	data	
-----	------	--

This is constructor w/o parameter

### 3.6.2.2 SinglyLinkedListNode() [2/2]

```
SinglyLinkedListNode::SinglyLinkedListNode (
    ll val ) [inline]
```

#### Parameters

in	<i>val</i>	
out	<i>data</i>	

This is destructor with parameter val

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

- [DSA.h](#)

## 3.7 Trie Class Reference

Brief Description of [Trie](#). Brief Description continued.

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

### Public Member Functions

- [Trie](#) ()
- bool **find** ([Trie](#) \*T, char c)
- void **insert** (string s)
- bool **checkPrefix** (string s)
- ll **countPrefix** (string s)

### Public Attributes

- ll **count**
- map< char, [Trie](#) \* > **nodes**

### 3.7.1 Detailed Description

Brief Description of [Trie](#). Brief Description continued.

Detailed description starts here.



## 3.7.2 Constructor & Destructor Documentation

### 3.7.2.1 Trie()

```
Trie::Trie ( ) [inline]
```

This is a constructor

[Trie](#) Functions- find, insert, checkPrefix, countPrefix

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

- [DSA.h](#)



## Chapter 4

# File Documentation

### 4.1 DSA.h File Reference

It is about various Linked List Nodes.

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

#### Classes

- class [SinglyLinkedListNode](#)  
*Brief Description of [SinglyLinkedListNode](#). Brief Description continued.*
- class [SinglyLinkedList](#)  
*Brief Description about [SinglyLinkedList](#).*
- class [DoublyLinkedListNode](#)  
*Brief Description of [DoublyLinkedListNode](#). Brief Description continued.*
- class [DoublyLinkedList](#)  
*Brief Description of [DoublyLinkedList](#). Brief Description continued.*
- class [BSTNode](#)  
*Brief Description of [BSTNode](#). Brief Description continued.*
- class [BinarySearchTree](#)  
*Brief Description of [BinarySearchTree](#). Brief Description continued.*
- class [Trie](#)  
*Brief Description of [Trie](#). Brief Description continued.*

#### Macros

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

#### Functions

- ostream & **operator**<< (ostream &out, const [SinglyLinkedListNode](#) &node)
- ostream & **operator**<< (ostream &out, const [DoublyLinkedListNode](#) &node)
- ostream & **operator**<< (ostream &out, const [BSTNode](#) &node)

### 4.1.1 Detailed Description

It is about various Linked List Nodes.

#### Author

Yash Kulkarni

#### Date

Sep 21 2022

## 4.2 DSA.h

[Go to the documentation of this file.](#)

```

1 #include <bits/stdc++.h>
2 #define ll long long int
3 #define vi vector<int>
4 #define vll vector<ll>
5 using namespace std;
6
13 /* ----- Data Structures ----- */
14
15 // ----- Singly Linked List -----
21 class SinglyLinkedListNode {
22
23     public:
24         ll data;
25         SinglyLinkedListNode* next;
26
27         SinglyLinkedListNode () {
28             data = -1;
29             next = NULL;
30         }
31
32         SinglyLinkedListNode (ll val) {
33             data = val;
34             next = NULL;
35         }
36 };
37
47 ostream& operator<<(ostream &out, const SinglyLinkedListNode &node) {
48     return out << node.data;
49 }
55 class SinglyLinkedList {
56
57     public:
58         SinglyLinkedListNode *head, *tail;
59         SinglyLinkedList () {
60             head = NULL;
61             tail = NULL;
62         }
63
64         void insert (ll data) {
65             SinglyLinkedListNode *node = new SinglyLinkedListNode(data);
66             if (head == NULL) {
67                 head = node;
68             }
69             else {
70                 tail->next = node;
71                 tail = node;
72             }
73         }
74
75         SinglyLinkedListNode* find (ll data) {
76             SinglyLinkedListNode *ptr = head, *prev = NULL;
77             while (ptr != NULL && ptr->data != data) {
78                 prev = ptr;
79                 ptr = ptr->next;
80             }
81             return prev;
82         }
83     };

```

```

107
108     bool deleteVal (ll data) {
109         SinglyLinkedListNode *prev = find(data);
110         if (prev -> next == NULL) {
111             return false;
112         }
113         prev -> next -> next = prev -> next;
114         return true;
115     }
116
117 void printer (string sep = ", ") {
118     SinglyLinkedListNode *ptr = head;
119     cout << "[";
120     while (ptr != NULL) {
121         cout << *ptr;
122         ptr = ptr -> next;
123         if (ptr != NULL) {
124             cout << sep;
125         }
126     }
127     cout << "]\n";
128 }
129
130 void reverse () {
131     SinglyLinkedListNode *ptr = head, *prev = NULL;
132     while (ptr != NULL) {
133         SinglyLinkedListNode *ptr2 = ptr -> next;
134         ptr -> next = prev;
135         prev = ptr;
136         ptr = ptr2;
137     }
138     tail = ptr;
139     head = prev;
140 }
141
142 };
143
144 // SinglyLinkedList merge (SinglyLinkedList list1, SinglyLinkedList list2) {
145
146 //         ///@param[in] list1
147 //         ///@param[in] list2
148 //         ///@param[out] merged
149 //         ///
150 //         ///@brief This is function merge
151 //         SinglyLinkedList merged;
152 //         SinglyLinkedListNode *head1 = list1.head, *head2 = list2.head;
153 //         while (head1 != NULL && head2 != NULL) {
154 //             if (head1 -> data < head2 -> data) {
155 //                 merged.insert(head1 -> data);
156 //                 head1 = head1 -> next;
157 //             }
158 //             else {
159 //                 merged.insert(head2 -> data);
160 //                 head2 = head2 -> next;
161 //             }
162 //         }
163 //         if (head1 == NULL && head2 != NULL) {
164 //             merged.tail -> next = head2;
165 //         }
166 //         if (head2 == NULL && head1 != NULL) {
167 //             merged.tail -> next = head1;
168 //         }
169 //         return merged;
170 //     }
171
172 // ----- Doubly Linked List -----
173
174 class DoublyLinkedListNode {
175 public:
176     ll data;
177     DoublyLinkedListNode *next, *prev;
178
179     DoublyLinkedListNode () {
180         data = -1;
181         next = NULL;
182         prev = NULL;
183     }
184
185     DoublyLinkedListNode (ll val) {
186         data = val;
187         next = NULL;
188         prev = NULL;
189     }
190
191 };
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213

```

```

214
215 ostream& operator<<(ostream &out, const DoublyLinkedListNode &node) {
216     return out << node.data;
217 }
223 class DoublyLinkedList {
224
225     public:
226
227         DoublyLinkedListNode *head, *tail;
228
229         DoublyLinkedList () {
230             head = NULL;
231             tail = NULL;
232         }
233
234         void insert (ll data) {
235             DoublyLinkedListNode *node = new DoublyLinkedListNode(data);
236             if (head == NULL) {
237                 head = node;
238             }
239             else {
240                 tail -> next = node;
241                 node -> prev = tail;
242             }
243             tail = node;
244         }
245
246         void printer (string sep = ", ") {
247             DoublyLinkedListNode *ptr = head;
248             cout << "[";
249             while (ptr != NULL) {
250                 cout << *ptr;
251                 ptr = ptr -> next;
252                 if (ptr != NULL) {
253                     cout << sep;
254                 }
255             }
256             cout << "]\n";
257         }
258
259         void reverse () {
260             DoublyLinkedListNode *ptr = head, *pr = NULL;
261             while (ptr != NULL) {
262                 DoublyLinkedListNode *ptr2 = ptr -> next;
263                 if (ptr2 != NULL) {
264                     ptr2 -> prev = ptr;
265                 }
266                 ptr -> next = pr;
267                 ptr -> prev = ptr2;
268                 pr = ptr;
269                 ptr = ptr2;
270             }
271             tail = ptr;
272             head = pr;
273         }
274 };
275
276 // ----- Binary Search Tree -----
277 class BSTNode {
278
279     public:
280
281         ll info, level;
282         BSTNode *left, *right;
283
284         BSTNode (ll val) {
285             info = val;
286             level = 0;
287             left = NULL;
288             right = NULL;
289         }
290 };
291
292 ostream& operator<<(ostream &out, const BSTNode &node) {
293     return out << node.info;
294 }
295
296 class BinarySearchTree {
297
298     public:
299
300         BSTNode *root;
301
302         enum order {PRE, IN, POST};
303
304         BinarySearchTree () {

```

```

324         root = NULL;
325     }
326
327     void insert(ll val) {
328         if (root == NULL) {
329             root = new BSTNode(val);
330         }
331         else {
332             BSTNode *ptr = root;
333             while (true) {
334                 if (val < ptr->info) {
335                     if (ptr->left != NULL) {
336                         ptr = ptr->left;
337                     }
338                     else {
339                         ptr->left = new BSTNode(val);
340                         break;
341                     }
342                 }
343                 else if (val > ptr->info) {
344                     if (ptr->right != NULL) {
345                         ptr = ptr->right;
346                     }
347                     else {
348                         ptr->right = new BSTNode(val);
349                         break;
350                     }
351                 }
352             }
353         }
354     }
355 }
356
357 void traverse (BSTNode* T, order tt) {
358     if (tt == PRE) {
359         cout << T << endl;
360         if (T->left != NULL) {
361             traverse(T->left,tt);
362         }
363         if (T->right != NULL) {
364             traverse(T->right,tt);
365         }
366     }
367     else if (tt == IN) {
368         if (T->left != NULL) {
369             traverse(T->left,tt);
370         }
371         cout << T << endl;
372         if (T->right != NULL) {
373             traverse(T->right,tt);
374         }
375     }
376     else if (tt == POST) {
377         if (T->left != NULL) {
378             traverse(T->left,tt);
379         }
380         if (T->right != NULL) {
381             traverse(T->right,tt);
382         }
383         cout << T << endl;
384     }
385 }
386
387 ll height(BSTNode *T) {
388     if (T->left == NULL && T->right == NULL) {
389         return 0;
390     }
391     else if (T->right == NULL) {
392         return 1 + height(T->left);
393     }
394     else if (T->left == NULL) {
395         return 1 + height(T->right);
396     }
397     return max(1 + height(T->left), 1 + height(T->right));
398 }
399
400 };
401
402 // ----- Suffix Trie -----
403 class Trie {
404 public:
405     ll count;
406     map<char,Trie*> nodes;
407
408     Trie () {

```

```
419         count = 0;
420         nodes = map<char, Trie*>();
421     }
422
423     bool find(Trie* T, char c) {
424         return ((T -> nodes).find(c) != (T -> nodes).end());
425     }
426
427     void insert(string s) {
428         Trie* ptr = this;
429         for (auto c: s) {
430             if (!find(ptr, c)) {
431                 (ptr -> nodes)[c] = new Trie();
432             }
433             ptr = (ptr -> nodes)[c];
434             (ptr -> count)++;
435         }
436     }
437
438     bool checkPrefix(string s) {
439         Trie* ptr = this;
440         for (ll i = 0; i < s.length(); i++) {
441             if (!find(ptr, s[i])) {
442                 if (i == s.length() - 1) {
443                     (ptr -> nodes)[s[i]] = NULL;
444                 }
445                 else {
446                     (ptr -> nodes)[s[i]] = new Trie();
447                 }
448             }
449             else if ((ptr -> nodes)[s[i]] == NULL or i == s.length() - 1) {
450                 return true;
451             }
452             ptr = (ptr -> nodes)[s[i]];
453         }
454         return false;
455     }
456
457     ll countPrefix(string s) {
458         bool found = true;
459         Trie* ptr = this;
460         for (auto c: s) {
461             if (find(ptr, c)) {
462                 ptr = (ptr -> nodes)[c];
463             }
464             else {
465                 found = false;
466                 break;
467             }
468         }
469         if (found) {
470             return ptr -> count;
471         }
472         return 0;
473     }
474
475 };
```



# Index

- BinarySearchTree, [5](#)
  - BinarySearchTree, [5](#)
- BSTNode, [6](#)
  - BSTNode, [6](#)
- deleteVal
  - SinglyLinkedList, [10](#)
- DoublyLinkedList, [7](#)
  - DoublyLinkedList, [7](#)
- DoublyLinkedListNode, [7](#)
  - DoublyLinkedListNode, [8](#)
- DSA.h, [15](#)
- find
  - SinglyLinkedList, [10](#)
- insert
  - SinglyLinkedList, [10](#)
- printer
  - SinglyLinkedList, [10](#)
- reverse
  - SinglyLinkedList, [10](#)
- SinglyLinkedList, [8](#)
  - deleteVal, [10](#)
  - find, [10](#)
  - insert, [10](#)
  - printer, [10](#)
  - reverse, [10](#)
  - SinglyLinkedList, [9](#)
- SinglyLinkedListNode, [11](#)
  - SinglyLinkedListNode, [11](#), [12](#)
- Trie, [12](#)
  - Trie, [13](#)