# 01-bstHashTable

Generated by Doxygen 1.8.11

# Contents

## 1 README

This is how the c files are structured:

```
1 ->  =   left operand is "included" by right operand
2
3                          ->---------------------------------------
4                          ->-------------------------|              |
5 utils.c   ->   node.c   ->    {  bst.c  }   ->    ht.c    ->     main.c
6           ->------------|     {  list.c }          |              |
7           ->-------------------------------------  |              |
8           ->-------------------------------------------------------
```

This experiment consists in inserting, searching and deleting key-value elements in two separate hash tablesm implemented with chaining lists or bsts. The experiments have been done using the same keys for both cases, and with the following probabilities based on the operation type.

```
1 insert probability = 75 %
2 search probability = 12.5 %
3 delete probability = 12.5 %
```

To see the difference between these two implementations we have to keeps the number of tests small, otherwise we wouldn't see the logarithmic curve of the bst (since this would result in an apparent straight line). This is true because the grater the number of tests the greater the load factor. For example:

```
1 #define ATTEMPTS 50
2 #define CHUNK 800
```

is very different than doing:

```
1 #define ATTEMPTS 800
2 #define CHUNK 50
```

although the number of total operations in the last iteration is the same for both cases (40000). For the second case infact the load factor is much more higher, thus generating much slower and pretty useless outputs.
Raw data

## 2 Data Structure Index

### 2.1 Data Structures

Here are the data structures with brief descriptions:



## 3 File Index

### 3.1 File List

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

# 4 Data Structure Documentation

## 4.1 bstNode Struct Reference

Bst Node Abstract Data Type.
`#include <globalDefines.h>`

### Data Fields

- struct Node * left
  *Pointer to the left Node object.*
- struct Node * right
  *Pointer to the right Node object.*
- struct Node * parent
  *Pointer to the parent Node object.*

### 4.1.1 Detailed Description

Bst Node Abstract Data Type.
The documentation for this struct was generated from the following file:

- globalDefines.h

## 4.2 HashTable Struct Reference

HashTable Abstract Data Type.
`#include <globalDefines.h>`

### Data Fields

- nodePtr * ptr
  *Pointer to the array of nodePtr.*
- unsigned int numberOfSlots
  *Number of slots of the hash table.*
- char type
  *'b' or 'l' for either bsts or lists.*

### 4.2.1 Detailed Description

HashTable Abstract Data Type.

**Note**

Each element in the whole hash table must be unique.

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

- globalDefines.h

## 4.3 listNode Struct Reference

List Node Abstract Data Type.
`#include <globalDefines.h>`

#### Data Fields

- struct Node * next

  *Pointer to the next Node object.*
- struct Node * prev

  *Pointer to the previous Node obkect.*

### 4.3.1 Detailed Description

List Node Abstract Data Type.
The documentation for this struct was generated from the following file:

- globalDefines.h

## 4.4 Node Struct Reference

Node Abstract Data Type.
`#include <globalDefines.h>`

#### Data Fields

- char * **key**
- char * **value**
- struct listNode * ln

  *Pointer to a listNode struct which contains listNode information.*
- struct bstNode * bn

  *Pointer to a bstNode struct which contains bstNode information.*

### 4.4.1 Detailed Description

Node Abstract Data Type.

#### Note

A node should be only of one type (i.e: either listNode or bstNode is set, but not both). If you want you could use both ln and bn pointers at the same time, but keep in mind to change the code appropriately.

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

- globalDefines.h

# 5 File Documentation

## 5.1 bst.c File Reference

Bst functions.
`#include "globalDefines.h"`

#### Functions

- static node BSTLeft (node root)

  *Get the left node of the input node.*
- static node BSTRight (node root)

  *Get the right node of the input node.*
- static node BSTParent (node root)

  *Get the parent node of the input node.*
- static node BSTMaxElement (node root)

*Get the node with the maximum root starting from the input node.*

- static node BSTPredecessor (node root)

  *Get the predecessor node of the input node.*
- static bool BSTIsLeaf (node root)

  *Check if the input node is a leaf.*
- static bool BSTHasLeftOnly (node root)

  *Check if the input node has a left child only.*
- static bool BSTHasRightOnly (node root)

  *Check if the input node has a right child only.*
- static node BSTNewNode (nodePtr rootPtr, node parentNode, char ∗key, char ∗value)

  *Function that creates a new node with the specified values in the position pointed by rootPtr.*
- static node BSTNonEmptyInsert (node root, char ∗key, char ∗value)

  *Function that looks for the correct position where to insert a new node.*
- static bool BSTNonEmptyDelete (nodePtr rootPtr, node root, char ∗key)

  *Function that looks (and deletes) for the correct position where to delete a specified node.*
- node BSTInsert (nodePtr rootPtr, char ∗key, char ∗value)

  *Insert a new node in a specified BST.*
- node BSTSearch (node root, char ∗key)

  *Search for a node with a given key in a specified BST.*
- bool BSTDelete (nodePtr rootPtr, char ∗key)

  *Delete the node with the given key in a specified BST.*
- node BSTClear (node root)

  *Delete the BST starting from the specified point.*
- bool BSTIs (node root, char ∗minKey, char ∗maxKey)

  *Check if the input tree is a BST.*
- void BSTPrint (node root)

  *Print a BST in pre-order criteria.*

#### 5.1.1 Detailed Description

Bst functions.

**Author**

Franco Masotti

**Date**

02 May 2016

**Copyright**



#### 5.1.2 Function Documentation

##### 5.1.2.1 node BSTClear ( node *root* )

Delete the BST starting from the specified point.

**Parameters**

| | | |
|---|---|---|
| in | *root* | A pointer to the BST. |

**Return values**

| | |
|---|---|
| *NULL* | |

**Note**

Usually this function is used to delete the whole tree.

#### 5.1.2.2 bool BSTDelete ( nodePtr *rootPtr*, char * *key* )

Delete the node with the given key in a specified BST.

**Parameters**

| | | |
|---|---|---|
| in | *root* | A pointer to the BST. |
| in | *key* | A memory address corresponding to the key. |

**Return values**

| | |
|---|---|
| *true* | The node was deleted. |
| *false* | The node was not deleted. |

**Note**

return value is false if root is empty or the specified element was not found.

#### 5.1.2.3 static bool BSTHasLeftOnly ( node *root* ) [static]

Check if the input node has a left child only.

**Parameters**

| | | |
|---|---|---|
| in | *root* | A pointer to the BST. |

**Return values**

| | |
|---|---|
| *true* | Input node has a left child only. |
| *true* | Input node has not a left child only. |

#### 5.1.2.4 static bool BSTHasRightOnly ( node *root* ) [static]

Check if the input node has a right child only.

**Parameters**

| | | |
|---|---|---|
| in | *root* | A pointer to the BST. |

**Return values**

| | |
|---|---|
| *true* | Input node has a right child only. |
| *true* | Input node has not a right child only. |

#### 5.1.2.5 node BSTInsert ( nodePtr *rootPtr,* char * *key,* char * *value* )

Insert a new node in a specified BST.

**Parameters**

| | | |
|---|---|---|
| in | *rootPtr* | A memory address containing the pointer to the root node (BST). |
| in | *key* | A memory address corresponding to the key. |
| in | *value* | A memory address corresponding to the value. |

**Return values**

| | |
|---|---|
| *new_node* | A memory address corresponding to the new node. |

**Warning**

The return value can be NULL.

#### 5.1.2.6 bool BSTIs ( node *root,* char * *minKey,* char * *maxKey* )

Check if the input tree is a BST.

**Parameters**

| | | |
|---|---|---|
| in | *root* | A pointer to the BST. |
| in | *minKey* | A memory address corresponding to the minimum key value. |
| in | *maxKey* | A memory address corresponding to the maximum key value. |

**Return values**

| | |
|---|---|
| *true* | The input tree is a BST. |
| *false* | The input tree is not a BST. |

#### 5.1.2.7 static bool BSTIsLeaf ( node *root* ) [static]

Check if the input node is a leaf.

**Parameters**

| | | |
|---|---|---|
| in | *root* | A pointer to the BST. |

**Return values**

| | |
|---|---|
| *true* | Input node is a leaf. |
| *false* | Input node is not a leaf. |

#### 5.1.2.8 static node BSTLeft ( node *root* ) [static]

Get the left node of the input node.

**Parameters**

| | | |
|---|---|---|
| in | *root* | A pointer to the BST. |

**Return values**

| | |
|---|---|
| *root->bn->left* | A memory address corresponding to the left node. |

**Warning**

The return value can be NULL.

#### 5.1.2.9 static node BSTMaxElement ( node *root* ) [static]

Get the node with the maximum root starting from the input node.

**Parameters**

| | | |
|---|---|---|
| in | *key* | A pointer to the BST. |

**Return values**

| | |
|---|---|
| *BSTMaxElement* | A memory address corresponding to the maximum node in value. |

**Warning**

The return value can be NULL if the input tree is NULL.

#### 5.1.2.10 static node BSTNewNode ( nodePtr *rootPtr,* node *parentNode,* char * *key,* char * *value* ) [static]

Function that creates a new node with the specified values in the position pointed by rootPtr.

**Parameters**

| | | |
|---|---|---|
| in | *rootPtr* | A memory address containing the pointer to the BST. |
| in | *parentNode* | A pointer to the parent node of the one to be inserted. |
| in | *key* | A pointer to the key. |
| in | *key* | A pointer to the value. |

**Return values**

| | |
|---|---|
| **rootPtr* | A pointer to the new node. |

**Warning**

The return value can be NULL.

#### 5.1.2.11 static bool BSTNonEmptyDelete ( nodePtr *rootPtr,* node *root,* char * *key* ) [static]

Function that looks (and deletes) for the correct position where to delete a specified node.

**Parameters**

| | | |
|---|---|---|
| in | *root* | A pointer to the BST. |
| in | *key* | A pointer to the key. |
| in | *key* | A pointer to the value. |

**Return values**

| | |
|---|---|
| *true* | The node has been deleted. |
| *true* | The node has not deleted. |

**Note**

return value is false if root is empty or the specified element was not found.

#### 5.1.2.12 static node BSTNonEmptyInsert ( node *root*, char * *key*, char * *value* ) [static]

Function that looks for the correct position where to insert a new node.

**Parameters**

| | | |
|---|---|---|
| in | *root* | A pointer to the BST. |
| in | *key* | A pointer to the key. |
| in | *value* | A pointer to the value. |

**Return values**

| | |
|---|---|
| *BSTNewNode* | A pointer to the new node. |

**Note**

This function is called only if the original BST is not empty.

**Warning**

The return value can be NULL.

#### 5.1.2.13 static node BSTParent ( node *root* ) [static]

Get the parent node of the input node.

**Parameters**

| | | |
|---|---|---|
| in | *root* | A pointer to the BST. |

**Return values**

| | |
|---|---|
| *root->bn->parent* | A memory address corresponding to the parent node. |

**Warning**

The return value can be NULL.

#### 5.1.2.14 static node BSTPredecessor ( node *root* ) [static]

Get the predecessor node of the input node.

**Parameters**

| | | |
|---|---|---|
| in | *root* | A pointer to the BST. |

**Return values**

| | |
|---|---|
| *BSTPredecessor* | A memory address corresponding to the parent node. |

**Warning**

The return value can be NULL if the input tree is NULL.

#### 5.1.2.15 void BSTPrint ( node *root* )

Print a BST in pre-order criteria.

**Parameters**

| | | |
|---|---|---|
| in | *root* | A pointer to the BST. |

#### 5.1.2.16 static node BSTRight ( node *root* ) [static]

Get the right node of the input node.

**Parameters**

| | | |
|---|---|---|
| in | *root* | A pointer to the BST. |

**Return values**

| | |
|---|---|
| *root->bn->right* | A memory address corresponding to the right node. |

**Warning**

The return value can be NULL.

#### 5.1.2.17 node BSTSearch ( node *root,* char * *key* )

Search for a node with a given key in a specified BST.

**Parameters**

| | | |
|---|---|---|
| in | *root* | A pointer to the BST. |
| in | *key* | A memory address corresponding to the key. |

**Return values**

| | |
|---|---|
| *root* | A memory address corresponding to the searched node. |

#### Warning

The return value can be NULL.

## 5.2 globalDefines.h File Reference

Header file containing exportable methods.

```
#include <assert.h>
#include <errno.h>
#include <limits.h>
#include <math.h>
#include <stdio.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
```

#### Data Structures

- struct listNode

  *List Node Abstract Data Type.*
- struct bstNode

  *Bst Node Abstract Data Type.*
- struct Node

  *Node Abstract Data Type.*
- struct HashTable

  *HashTable Abstract Data Type.*

#### Macros

- #define M_GLOBALDEFINES_H

  *Include the main header.*
- #define ISOC99_SOURCE

  *Tell the compiler that we want ISO C99 source, and check if the system has ANSI C 99.*
- #define **_POSIX_C_SOURCE** 199309L

#### Typedefs

- typedef struct Node * **node**
- typedef node * **nodePtr**
- typedef struct bstNode * **bnode**
- typedef struct listNode * **lnode**
- typedef struct HashTable * **ht**

#### Functions

- bool element_null (void *element)

  *Check if any type of pointer is NULL.*
- void * malloc_safe (size_t size)

  *Allocate new space in a safe way.*
- bool node_null (node n)

  *Check if the node is NULL.*
- char * key_get (node n)

  *Get the key from a node.*

- bool key_set (node n, char *key)
  *Store the key inside a node.*
- bool keys_equal (char *key1, char *key2)
  *Check if two keys are equal.*
- bool keys_less (char *key1, char *key2)
  *Check if the first key is smaller than the second.*
- bool keys_greater (char *key1, char *key2)
  *Check if the first key is bigger than the second.*
- char * value_get (node n)
  *Get the value from a node.*
- bool value_set (node n, char *value)
  *Store the value inside a node.*
- node node_new (char *key, char *value, char type)
  *Create a generic new node.*
- void node_delete (nodePtr nPtr, char type)
  *Delete a generic node.*
- nodePtr nodeptr_new (void)
  *Set a node object to NULL.*
- bool BSTIs (node root, char *minKey, char *maxKey)
  *Check if the input tree is a BST.*
- node BSTInsert (nodePtr rootPtr, char *key, char *value)
  *Insert a new node in a specified BST.*
- node BSTSearch (node root, char *key)
  *Search for a node with a given key in a specified BST.*
- bool BSTDelete (nodePtr rootPtr, char *key)
  *Delete the node with the given key in a specified BST.*
- node BSTClear (node root)
  *Delete the BST starting from the specified point.*
- void BSTPrint (node root)
  *Print a BST in pre-order criteria.*
- node LISTInsert (nodePtr headPtr, char *key, char *value)
  *Insert a new node in a specified LIST.*
- node LISTSearch (node head, char *key)
  *Search for a node with a given key in a specified LIST.*
- bool LISTDelete (nodePtr headPtr, char *key)
  *Delete the node with the given key in a specified LIST.*
- node LISTClear (node head)
  *Delete the LIST starting from the specified point.*
- void LISTPrint (node head)
  *Print a LIST element by element with the same order as when they were inserted.*
- ht HTInit (unsigned int numberOfSlots, char type)
  *Create a new hash table.*
- bool HTInsert (ht hashTable, char *key, char *value)
  *Insert a new key value tuple in the specified hash table.*
- node HTSearch (ht hashTable, char *key)
  *Search for a node with a given key in a specified hash table.*
- bool HTDelete (ht hashTable, char *key)
  *Delete the node with the given key in a specified hash table.*
- bool HTClear (ht *hashTable)
  *Delete the whole hash table.*
- void HTPrint (ht hashTable)
  *Print the hash table, slot by slot, starting from slot 0 to slot M - 1.*

#### 5.2.1 Detailed Description

Header file containing exportable methods.

**Author**

Franco Masotti

**Date**

02 May 2016

**Copyright**



#### 5.2.2 Function Documentation

##### 5.2.2.1 node BSTClear ( node *root* )

Delete the BST starting from the specified point.

**Parameters**

| | | |
|---|---|---|
| in | *root* | A pointer to the BST. |

**Return values**

| | |
|---|---|
| *NULL* | |

**Note**

Usually this function is used to delete the whole tree.

##### 5.2.2.2 bool BSTDelete ( nodePtr *rootPtr,* char * *key* )

Delete the node with the given key in a specified BST.

**Parameters**

| | | |
|---|---|---|
| in | *root* | A pointer to the BST. |
| in | *key* | A memory address corresponding to the key. |

**Return values**

| | |
|---|---|
| *true* | The node was deleted. |
| *false* | The node was not deleted. |

**Note**

return value is false if root is empty or the specified element was not found.

##### 5.2.2.3 node BSTInsert ( nodePtr *rootPtr,* char * *key,* char * *value* )

Insert a new node in a specified BST.

**Parameters**

| | | |
|---|---|---|
| in | *rootPtr* | A memory address containing the pointer to the root node (BST). |
| in | *key* | A memory address corresponding to the key. |
| in | *value* | A memory address corresponding to the value. |

**Return values**

| | |
|---|---|
| *new_node* | A memory address corresponding to the new node. |

**Warning**

The return value can be NULL.

#### 5.2.2.4 bool BSTIs ( node *root*, char * *minKey*, char * *maxKey* )

Check if the input tree is a BST.

**Parameters**

| | | |
|---|---|---|
| in | *root* | A pointer to the BST. |
| in | *minKey* | A memory address corresponding to the minimum key value. |
| in | *maxKey* | A memory address corresponding to the maximum key value. |

**Return values**

| | |
|---|---|
| *true* | The input tree is a BST. |
| *false* | The input tree is not a BST. |

#### 5.2.2.5 void BSTPrint ( node *root* )

Print a BST in pre-order criteria.

**Parameters**

| | | |
|---|---|---|
| in | *root* | A pointer to the BST. |

#### 5.2.2.6 node BSTSearch ( node *root*, char * *key* )

Search for a node with a given key in a specified BST.

**Parameters**

| | | |
|---|---|---|
| in | *root* | A pointer to the BST. |
| in | *key* | A memory address corresponding to the key. |

**Return values**

| | |
|---|---|
| *root* | A memory address corresponding to the searched node. |

**Warning**

The return value can be NULL.

#### 5.2.2.7 bool element_null ( void * *element* )

Check if any type of pointer is NULL.

**Parameters**

| | | |
|---|---|---|
| in | *element* | Any kind of pointer. |

**Return values**

| | |
|---|---|
| *true* | element is NULL. |
| *false* | element is not NULL. |

#### 5.2.2.8 bool HTClear ( ht * *hashTable* )

Delete the whole hash table.

**Parameters**

| | | |
|---|---|---|
| in | *hashTAble* | A memory address corresponding to the pointer of the hash table . |

**Return values**

| | |
|---|---|
| *true* | The whole hash table has been deleted correctly. |
| *false* | Some problem occurred while deleting a slot. |

#### 5.2.2.9 bool HTDelete ( ht *hashTable,* char * *key* )

Delete the node with the given key in a specified hash table.

**Parameters**

| | | |
|---|---|---|
| in | *head* | A pointer to the hash table. |
| in | *key* | A memory address corresponding to the key. |

**Return values**

| | |
|---|---|
| *true* | The node was deleted. |
| *false* | The node was not deleted. |

**Note**

return value is false if the computed slot is empty or the specified element was not found.

#### 5.2.2.10 ht HTInit ( unsigned int *numberOfSlots,* char *type* )

Create a new hash table.

**Parameters**

| | | |
|---|---|---|
| in | *numberOfSlots* | The total number of slots of the hash table. |
| in | *type* | A character corresponding to the type of hash table to be created. |

**Return values**

| | |
|---|---|
| *hashTable* | A memory address corresponding to the hash table just created. |

**Note**

type can either be 'b' or 'l', for either bsts or lists.

#### 5.2.2.11 bool HTInsert ( ht *hashTable*, char * *key*, char * *value* )

Insert a new key value tuple in the specified hash table.

**Parameters**

| | | |
|---|---|---|
| in | *hashTable* | A memory address corresponding to the hash table. |
| in | *key* | A memory address corresponding to the key. |
| in | *value* | A memory address corresponding to the value. |

**Return values**

| | |
|---|---|
| *true* | The tuple was inserted. |
| *false* | The tuple was not inserted. |

#### 5.2.2.12 void HTPrint ( ht *hashTable* )

Print the hash table, slot by slot, starting from slot 0 to slot M - 1.

**Parameters**

| | | |
|---|---|---|
| in | *hashTable* | A pointer to the hash table. |

#### 5.2.2.13 node HTSearch ( ht *hashTable*, char * *key* )

Search for a node with a given key in a specified hash table.

**Parameters**

| | | |
|---|---|---|
| in | *ht* | A pointer to the hash table. |
| in | *key* | A memory address corresponding to the key. |

**Return values**

| | |
|---|---|
| *node* | A memory address corresponding to the searched node. |

**Warning**

The return value can be NULL.

#### 5.2.2.14 char* key_get ( node *n* )

Get the key from a node.

**Parameters**

| | | |
|---|---|---|
| in | *n* | A pointer to a node instance. |

**Return values**

| | |
|---|---|
| *node->key* | A memory address corresponding to the key. |

**Warning**

The return value can be NULL.

#### 5.2.2.15 bool key_set ( node *n*, char * *key* )

Store the key inside a node.

**Parameters**

| | | |
|---|---|---|
| in | *n* | A pointer to a node instance. |
| in | *key* | A memory address corresponding to the key. |

**Return values**

| | |
|---|---|
| *true* | The key has been stored correctly. |
| *false* | The key has not been stored because the input node was NULL. |

#### 5.2.2.16 bool keys_equal ( char * *key1*, char * *key2* )

Check if two keys are equal.

**Parameters**

| | | |
|---|---|---|
| in | *key1* | A memory address corresponding to the first key. |
| in | *key2* | A memory address corresponding to the second key. |

**Return values**

| | |
|---|---|
| *true* | The two keys are equal. |
| *false* | The two keys differ. |

#### 5.2.2.17 bool keys_greater ( char * *key1*, char * *key2* )

Check if the first key is bigger than the second.

**Parameters**

| | | |
|---|---|---|
| in | *key1* | A memory address corresponding to the first key. |
| in | *key2* | A memory address corresponding to the second key. |

**Return values**

| | |
|---|---|
| *true* | key1 is bigger than key2. |
| *false* | key1 is not bigger than key2. |

#### 5.2.2.18 bool keys_less ( char * *key1*, char * *key2* )

Check if the first key is smaller than the second.

**Parameters**

| | | |
|---|---|---|
| in | *key1* | A memory address corresponding to the first key. |
| in | *key2* | A memory address corresponding to the second key. |

**Return values**

| | |
|---|---|
| *true* | key1 is smaller than key2. |
| *false* | key1 is not smaller than key2. |

#### 5.2.2.19 node LISTClear ( node *head* )

Delete the LIST starting from the specified point.

**Parameters**

| | | |
|---|---|---|
| in | *head* | A pointer to the LIST. |

**Return values**

| | |
|---|---|
| *NULL* | |

**Note**

Usually this function is used to delete the whole list.

#### 5.2.2.20 bool LISTDelete ( nodePtr *headPtr,* char * *key* )

Delete the node with the given key in a specified LIST.

**Parameters**

| | | |
|---|---|---|
| in | *head* | A pointer to the LIST. |
| in | *key* | A memory address corresponding to the key. |

**Return values**

| | |
|---|---|
| *true* | The node was deleted. |
| *false* | The node was not deleted. |

**Note**

return value is false if head is empty or the specified element was not found.

#### 5.2.2.21 node LISTInsert ( nodePtr *headPtr,* char * *key,* char * *value* )

Insert a new node in a specified LIST.

**Parameters**

| | | |
|---|---|---|
| in | *headPtr* | A memory address containing the pointer to the head node (BST). |
| in | *key* | A memory address corresponding to the key. |
| in | *value* | A memory address corresponding to the value. |

**Return values**

| | |
|---|---|
| *new_node* | A memory address corresponding to the new node. |

**Warning**

The return value can be NULL.

#### 5.2.2.22 void LISTPrint ( node *head* )

Print a LIST element by element with the same order as when they were inserted.

**Parameters**

| | | |
|---|---|---|
| in | *head* | A pointer to the LIST. |

#### 5.2.2.23 node LISTSearch ( node *head,* char * *key* )

Search for a node with a given key in a specified LIST.

**Parameters**

| | | |
|---|---|---|
| in | *head* | A pointer to the LIST. |
| in | *key* | A memory address corresponding to the key. |

**Return values**

| | |
|---|---|
| *head* | A memory address corresponding to the searched node. |

**Warning**

The return value can be NULL.

#### 5.2.2.24 void* malloc_safe ( size_t *size* )

Allocate new space in a safe way.

**Parameters**

| | | |
|---|---|---|
| in | *size* | Total size to be allocated . |

**Return values**

| | |
|---|---|
| *dst* | The pointer to the new instance of data. |

#### 5.2.2.25 void node_delete ( nodePtr *nPtr,* char *type* )

Delete a generic node.

**Parameters**

| | | |
|---|---|---|
| in | *nPtr* | A memory address containing the pointer of the node. |
| in | *type* | A character corresponding to the type of node to be deleted. |

**Note**

type can either be 'b' or 'l', for either bsts or lists.

#### 5.2.2.26 node node_new ( char * *key*, char * *value*, char *type* )

Create a generic new node.

**Parameters**

| | | |
|---|---|---|
| in | *key* | A memory address corresponding to the key. |
| in | *value* | A memory address corresponding to the value. |
| in | *type* | A character corresponding to the type of node to be created. |

**Note**

type can either be 'b' or 'l', for either bsts or lists.

**Return values**

| | |
|---|---|
| *new_node* | A memory address corresponding to the new node. |

**Warning**

The return value can be NULL.

#### 5.2.2.27 bool node_null ( node *n* )

Check if the node is NULL.

**Parameters**

| | | |
|---|---|---|
| in | *n* | A pointer to a node instance. |

**Return values**

| | |
|---|---|
| *true* | The node is NULL. |
| *false* | The node is not NULL. |

#### 5.2.2.28 nodePtr nodeptr_new ( void )

Set a node object to NULL.

**Return values**

| | |
|---|---|
| *npt* | A memory address corresponding to the new instance of a nodePtr object. |

**Warning**

The return value can be NULL.

#### 5.2.2.29 char* value_get ( node *n* )

Get the value from a node.

**Parameters**

| | | |
|---|---|---|
| in | *n* | A pointer to a node instance. |

**Return values**

| | |
|---|---|
| *node->value* | A memory address corresponding to the value. |

**Warning**

The return value can be NULL.

#### 5.2.2.30 bool value_set ( node *n*, char * *value* )

Store the value inside a node.

**Parameters**

| | | |
|---|---|---|
| in | *n* | A pointer to a node instance. |
| in | *value* | A memory address corresponding to the value. |

**Return values**

| | |
|---|---|
| *true* | The value has been stored correctly. |
| *false* | The value has not been stored because the input node was NULL. |

## 5.3 ht.c File Reference

Hash table functions.

```
#include "globalDefines.h"
```

**Functions**

- static char HTType (ht hashTable)
  *Get the type of hash table.*
- static unsigned int HTNumOfSlots (ht hashTable)
  *Get the number of slots of the hash table.*
- static nodePtr * HTPtr (ht hashTable)
  *Get the pointer corresponding to the array of slots (i.e: the array of nodePtr)*
- static unsigned int slotid_get (char *input, ht hashTable)
  *Get the slot number for a given string. This is also known as the hash function.*
- static nodePtr slot_get (ht hashTable, char *key)
  *Get the first nodePtr of the slot corresponding to the input key.*
- static void HTFreeStruct (ht *hashTablePtr)
  *Delete the struct (and its members) corresponding to the input hash table.*
- ht HTInit (unsigned int numberOfSlots, char type)
  *Create a new hash table.*
- bool HTInsert (ht hashTable, char *key, char *value)
  *Insert a new key value tuple in the specified hash table.*
- node HTSearch (ht hashTable, char *key)
  *Search for a node with a given key in a specified hash table.*

- bool HTDelete (ht hashTable, char *key)

  *Delete the node with the given key in a specified hash table.*
- bool HTClear (ht *hashTablePtr)

  *Delete the whole hash table.*
- void HTPrint (ht hashTable)

  *Print the hash table, slot by slot, starting from slot 0 to slot M - 1.*

### 5.3.1 Detailed Description

Hash table functions.

**Author**

Franco Masotti

**Date**

02 May 2016

**Copyright**



### 5.3.2 Function Documentation

#### 5.3.2.1 bool HTClear ( ht * *hashTable* )

Delete the whole hash table.

**Parameters**

| | | |
|---|---|---|
| in | *hashTAble* | A memory address corresponding to the pointer of the hash table . |

**Return values**

| | |
|---|---|
| *true* | The whole hash table has been deleted correctly. |
| *false* | Some problem occurred while deleting a slot. |

#### 5.3.2.2 bool HTDelete ( ht *hashTable*, char * *key* )

Delete the node with the given key in a specified hash table.

**Parameters**

| | | |
|---|---|---|
| in | *head* | A pointer to the hash table. |
| in | *key* | A memory address corresponding to the key. |

**Return values**

| | |
|---|---|
| *true* | The node was deleted. |
| *false* | The node was not deleted. |

**Note**

return value is false if the computed slot is empty or the specified element was not found.

#### 5.3.2.3 static void HTFreeStruct ( ht * *hashTablePtr* ) [static]

Delete the struct (and its members) corresponding to the input hash table.

**Parameters**

| | | |
|---|---|---|
| in | *hashTable* | A pointer to the memory address of the hash table. |

#### 5.3.2.4 ht HTInit ( unsigned int *numberOfSlots*, char *type* )

Create a new hash table.

**Parameters**

| | | |
|---|---|---|
| in | *numberOfSlots* | The total number of slots of the hash table. |
| in | *type* | A character corresponding to the type of hash table to be created. |

**Return values**

| | |
|---|---|
| *hashTable* | A memory address corresponding to the hash table just created. |

**Note**

type can either be 'b' or 'l', for either bsts or lists.

#### 5.3.2.5 bool HTInsert ( ht *hashTable*, char * *key*, char * *value* )

Insert a new key value tuple in the specified hash table.

**Parameters**

| | | |
|---|---|---|
| in | *hashTable* | A memory address corresponding to the hash table. |
| in | *key* | A memory address corresponding to the key. |
| in | *value* | A memory address corresponding to the value. |

**Return values**

| | |
|---|---|
| *true* | The tuple was inserted. |
| *false* | The tuple was not inserted. |

#### 5.3.2.6 static unsigned int HTNumOfSlots ( ht *hashTable* ) [static]

Get the number of slots of the hash table.

**Parameters**

| | | |
|---|---|---|
| in | *hashTable* | A pointer to the hash table. |

**Return values**

| | |
|---|---|
| *hashTable->numberOfSlots* | |

#### 5.3.2.7 void HTPrint ( ht *hashTable* )

Print the hash table, slot by slot, starting from slot 0 to slot M - 1.

**Parameters**

| | | |
|---|---|---|
| in | *hashTable* | A pointer to the hash table. |

#### 5.3.2.8 static nodePtr * HTPtr ( ht *hashTable* ) [static]

Get the pointer corresponding to the array of slots (i.e: the array of nodePtr)

**Parameters**

| | | |
|---|---|---|
| in | *hashTable* | A pointer to the hash table. |

**Return values**

| | |
|---|---|
| *hashTable->ptr* | |

**Warning**

The return value can be NULL.

#### 5.3.2.9 node HTSearch ( ht *hashTable,* char * *key* )

Search for a node with a given key in a specified hash table.

**Parameters**

| | | |
|---|---|---|
| in | *ht* | A pointer to the hash table. |
| in | *key* | A memory address corresponding to the key. |

**Return values**

| | |
|---|---|
| *node* | A memory address corresponding to the searched node. |

**Warning**

The return value can be NULL.

#### 5.3.2.10 static char HTType ( ht *hashTable* ) [static]

Get the type of hash table.

**Parameters**

| | | |
|---|---|---|
| in | *hashTable* | A pointer to the hash table. |

**Return values**

| | |
|---|---|
| *hashTable->type* | 'b' = BST, 'l' = LIST. |

#### 5.3.2.11 static nodePtr slot_get ( ht *hashTable*, char * *key* ) [static]

Get the first nodePtr of the slot corresponding to the input key.

**Parameters**

| | | |
|---|---|---|
| in | *hashTable* | A pointer to the hash table. |
| in | *key* | A pointer to the key. |

**Return values**

| | |
|---|---|
| *(HTPtr(hashTable))[slotid_get(key,hashTable)]* | |

**Warning**

The return value can be NULL.

#### 5.3.2.12 static unsigned int slotid_get ( char * *input*, ht *hashTable* ) [static]

Get the slot number for a given string. This is also known as the hash function.

**Parameters**

| | | |
|---|---|---|
| in | *input* | A pointer to the string that needs to be hashed. |
| in | *hashTable* | A pointer to the hash table. |

**Return values**

| | |
|---|---|
| *key%HTNumOfSlots(hashTable)* | |

## 5.4 list.c File Reference

List functions.

```
#include "globalDefines.h"
```

**Functions**

- static node LISTSuccessor (node head)

  *Get the successor node of the current one.*
- static node LISTPredecessor (node head)

  *Get the previous node of the current one.*
- static node LISTNewNode (nodePtr headPtr, node prevNode, char *key, char *value)

  *Function that creates a new node with the specified values in the position pointed by headPtr.*
- static node LISTNonEmptyInsert (node head, char *key, char *value)

  *Function that looks for the correct position where to insert a new node.*
- static bool LISTNonEmptyDelete (nodePtr headPtr, node head, char *key)

  *Function that looks (and deletes) for the correct position where to delete a specified node.*
- node LISTInsert (nodePtr headPtr, char *key, char *value)

  *Insert a new node in a specified LIST.*
- node LISTSearch (node head, char *key)

  *Search for a node with a given key in a specified LIST.*
- bool LISTDelete (nodePtr headPtr, char *key)

*Delete the node with the given key in a specified LIST.*

- node LISTClear (node head)

  *Delete the LIST starting from the specified point.*

- void LISTPrint (node head)

  *Print a LIST element by element with the same order as when they were inserted.*

### 5.4.1 Detailed Description

List functions.

**Author**

Franco Masotti

**Date**

02 May 2016

**Copyright**



### 5.4.2 Function Documentation

#### 5.4.2.1 node LISTClear ( node *head* )

Delete the LIST starting from the specified point.

**Parameters**

| | | |
|---|---|---|
| in | *head* | A pointer to the LIST. |

**Return values**

| | |
|---|---|
| *NULL* | |

**Note**

Usually this function is used to delete the whole list.

#### 5.4.2.2 bool LISTDelete ( nodePtr *headPtr,* char * *key* )

Delete the node with the given key in a specified LIST.

**Parameters**

| | | |
|---|---|---|
| in | *head* | A pointer to the LIST. |
| in | *key* | A memory address corresponding to the key. |

**Return values**

| | |
|---|---|
| *true* | The node was deleted. |
| *false* | The node was not deleted. |

**Note**

return value is false if head is empty or the specified element was not found.

#### 5.4.2.3 node LISTInsert ( nodePtr *headPtr,* char * *key,* char * *value* )

Insert a new node in a specified LIST.

**Parameters**

| | | |
|---|---|---|
| in | *headPtr* | A memory address containing the pointer to the head node (BST). |
| in | *key* | A memory address corresponding to the key. |
| in | *value* | A memory address corresponding to the value. |

**Return values**

| | |
|---|---|
| *new_node* | A memory address corresponding to the new node. |

**Warning**

The return value can be NULL.

#### 5.4.2.4 static node LISTNewNode ( nodePtr *headPtr,* node *prevNode,* char * *key,* char * *value* ) [static]

Function that creates a new node with the specified values in the position pointed by headPtr.

**Parameters**

| | | |
|---|---|---|
| in | *headPtr* | A memory address containing the pointer to the LIST. |
| in | *prevNode* | A pointer to the previous node of the one to be inserted. |
| in | *key* | A pointer to the key. |
| in | *key* | A pointer to the value. |

**Return values**

| | |
|---|---|
| **headPtr* | A pointer to the new node. |

**Warning**

The return value can be NULL.

#### 5.4.2.5 static bool LISTNonEmptyDelete ( nodePtr *headPtr,* node *head,* char * *key* ) [static]

Function that looks (and deletes) for the correct position where to delete a specified node.

**Parameters**

| | | |
|---|---|---|
| in | *head* | A pointer to the LIST. |
| in | *key* | A pointer to the key. |
| in | *key* | A pointer to the value. |

**Return values**

| | |
|---|---|
| *true* | The node has been deleted. |
| *true* | The node has not deleted. |

**Note**

return value is false if head is empty or the specified element was not found.

#### 5.4.2.6 static node LISTNonEmptyInsert ( node *head*, char * *key*, char * *value* ) [static]

Function that looks for the correct position where to insert a new node.

**Parameters**

| | | |
|---|---|---|
| in | *head* | A pointer to the LIST. |
| in | *key* | A pointer to the key. |
| in | *value* | A pointer to the value. |

**Return values**

| | |
|---|---|
| *BSTNewNode* | A pointer to the new node. |

**Note**

This function is called only if the original LIST is not empty.

**Warning**

The return value can be NULL.

#### 5.4.2.7 static node LISTPredecessor ( node *head* ) [static]

Get the previous node of the current one.

**Parameters**

| | | |
|---|---|---|
| in | *head* | A pointer to the list. |

**Return values**

| | |
|---|---|
| *head->ln->prev* | A memory address corresponding to the previous node. |

**Warning**

The return value can be NULL.

#### 5.4.2.8 void LISTPrint ( node *head* )

Print a LIST element by element with the same order as when they were inserted.

**Parameters**

| | | |
|---|---|---|
| in | *head* | A pointer to the LIST. |

#### 5.4.2.9 node LISTSearch ( node *head*, char * *key* )

Search for a node with a given key in a specified LIST.

**Parameters**

| | | |
|---|---|---|
| in | *head* | A pointer to the LIST. |

**Parameters**

| | | |
|---|---|---|
| in | *key* | A memory address corresponding to the key. |

**Return values**

| | |
|---|---|
| *head* | A memory address corresponding to the searched node. |

**Warning**

The return value can be NULL.

#### 5.4.2.10 static node LISTSuccessor ( node *head* ) [static]

Get the successor node of the current one.

**Parameters**

| | | |
|---|---|---|
| in | *head* | A pointer to the list. |

**Return values**

| | |
|---|---|
| *head->ln->next* | A memory address corresponding to the next node. |

**Warning**

The return value can be NULL.

## 5.5 main.c File Reference

Implementation file.

```
#include "globalDefines.h"
```

**Macros**

- #define M 997

  *NUmber of buckets of the hash tables.*
- #define **KEYCHARMIN** 33
- #define **KEYCHARMAX** 126
- #define **ATTEMPTS** 50
- #define **CHUNK** 800
- #define **KEYLENGTH** 8
- #define **INSPROB** 0.750
- #define **SRCPROB** 0.125
- #define **DELPROB** 0.125

**Functions**

- static double runningtime_get (clock_t start, clock_t end)

  *Get the delta of two clocks (i.e: the running time).*
- static char * randomstring_new (int len)

  *Generate a random string with a specified length.*
- static int * numbersfromprobability_get (int totalOperations, double insProb, double srcProb, double delProb)

*Get the number of operations of each type, given their probability.*

- static void array_shuffle (char *array, int len)

  *Shuffle a char array of a given length using Fisher-Yates algorithm.*
- static bool isInsertAction (char action)

  *Check if input corresponds to an insert action.*
- static bool isSearchAction (char action)

  *Check if input corresponds to a search action.*
- static double operations (ht hashTable, char **keys, char *actions, int totalOperations, int *succIns, int *succDel)

  *Simulate insert, search and delete operations on a hash table, and gather statistics.*
- static char ** keys_new (int quantity, int length)

  *Generate a new array of keys.*
- static void keys_delete (int quantity, char ***keysPtr)

  *Delete an array of keys.*
- static char * actions_get (int insElements, int srcElements, int delElements)

  *Generate a random array corresponding to the type of operations that needs to be done.*
- int **main** (void)

### 5.5.1 Detailed Description

Implementation file.

**Author**

Franco Masotti

**Date**

02 May 2016

**Copyright**



### 5.5.2 Function Documentation

#### 5.5.2.1 static char * actions_get ( int *insElements*, int *srcElements*, int *delElements* ) [static]

Generate a random array corresponding to the type of operations that needs to be done.

**Parameters**

| | | |
|---|---|---|
| in | *insElements* | The number of insert operations. |
| in | *srcElements* | The number of search operations. |
| in | *delElements* | The number of delete operations. |

**Return values**

| | |
|---|---|
| *actions* | The array of actions. |

**Note**

Possible actions are {'i', 's', 'd'} respectively for insert, search and delete operations.

#### 5.5.2.2 static void array_shuffle ( char * *array*, int *len* ) [static]

Shuffle a char array of a given length using Fisher-Yates algorithm.

**Parameters**

| | | |
|---|---|---|
| in | *array* | The array to be shuffled. |
| in | *len* | The length of the array. |

#### 5.5.2.3 static bool isInsertAction ( char *action* ) [static]

Check if input corresponds to an insert action.

**Parameters**

| | | |
|---|---|---|
| in | *action* | A character in the following domain: {'i', 's', 'd'}. |

**Return values**

| | |
|---|---|
| *true* | Input actions is an insert action. |
| *false* | Input actions is not an insert action. |

#### 5.5.2.4 static bool isSearchAction ( char *action* ) [static]

Check if input corresponds to a search action.

**Parameters**

| | | |
|---|---|---|
| in | *action* | A character in the following domain: {'i', 's', 'd'}. |

**Return values**

| | |
|---|---|
| *true* | Input actions is a search action. |
| *false* | Input actions is not a search action. |

#### 5.5.2.5 static void keys_delete ( int *quantity*, char *** *keysPtr* ) [static]

Delete an array of keys.

**Parameters**

| | | |
|---|---|---|
| in | *quantity* | The number of keys. |

#### 5.5.2.6 static char ** keys_new ( int *quantity*, int *length* ) [static]

Generate a new array of keys.

**Parameters**

| | | |
|---|---|---|
| in | *quantity* | The number of keys. |
| in | *length* | The length of each key. |

**Return values**

| | |
|---|---|
| *keys* | The pointer to the array of keys. |

#### 5.5.2.7 static int * numbersfromprobability_get ( int *totalOperations,* double *insProb,* double *srcProb,* double *delProb* ) [static]

Get the number of operations of each type, given their probability.

**Parameters**

| | | |
|---|---|---|
| in | *totalOperations* | The overall number of operations. |
| in | *insProb* | Probability of insertion operations. |
| in | *srcProb* | Probability of search operations. |
| in | *delProb* | Probability of delete operations. |

**Return values**

| | |
|---|---|
| *numbers* | An array of three integers where: index 0 = insert operations, index 1 = search operations, index 2 = delete operations. |

**Note**

Probabilities are expressed in the following domain: [0, 1].

**Warning**

sum(numbers) may be different than totalOperations due to approximations.

#### 5.5.2.8 static double operations ( ht *hashTable,* char ** *keys,* char * *actions,* int *totalOperations,* int * *succIns,* int * *succDel* ) [static]

Simulate insert, search and delete operations on a hash table, and gather statistics.

**Parameters**

| | | |
|---|---|---|
| in | *ht* | A pointer to the hash table. |
| in | *keys* | The array of keys. |
| in | *actions* | The array of actions. |
| in | *totalOperations* | The overall number of search, insert and delete operations to be done. |
| in | *succIns* | A pointer to a variable containing the number of successful insertion operations. |
| in | *succDel* | A pointer to a variable containing the number of successful deletion operations. |

**Return values**

| | |
|---|---|
| *totalTime* | The running time for a series of operations on the hash table. |
| *succIns* | The number of successful insertion operations. |
| *succDel* | The number of successful deletion operations. |

#### 5.5.2.9 static char * randomstring_new ( int *len* ) [static]

Generate a random string with a specified length.

**Parameters**

| | | |
|---|---|---|
| in | *len* | The length of the random string. |

**Return values**

| | |
|---|---|
| *str* | The random string. |

**Note**

The domain of characters of the random string is given by KEYCHARMIN and KEYCHARMAX macros.

#### 5.5.2.10 static double runningtime_get ( clock_t *start*, clock_t *end* ) [static]

Get the delta of two clocks (i.e: the running time).

**Parameters**

| | | |
|---|---|---|
| in | *start* | A clock corresponding to the start time. |
| in | *end* | A clock corresponding to the end time. |

**Return values**

| | |
|---|---|
| *end-start* | The time difference between two clocks. |