Doubly Linked List

Generated by Doxygen 1.8.13

Contents

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 2

File Index

2.1 File List

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

Chapter 3

Data Structure Documentation

3.1 listHead Struct Reference

`#include <LinkedListAPI.h>`

Data Fields

- Node * **head**
- Node * **tail**
- void(* **deleteNode**)(void *toBeDeleted)
- int(* **compare**)(const void *first, const void *second)
- void(* **printNode**)(void *toBePrinted)

3.1.1 Detailed Description

Dummy head of the list. Contains no actual data on it beyond a pointer to the front and end of the list.

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

- include/LinkedListAPI.h

3.2 listNode Struct Reference

`#include <LinkedListAPI.h>`

Data Fields

- void * **data**
- struct listNode * **previous**
- struct listNode * **next**

3.2.1 Detailed Description

Node of a linked list. This list is doubly linked, meaning that it has points to both the node immediately in front of it, as well as the node immediately behind it.

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

- include/LinkedListAPI.h

Chapter 4

File Documentation

4.1 include/LinkedListAPI.h File Reference

File containing the function definitions of a doubly linked list.

```
#include <stdio.h>
#include <stdlib.h>
```

Data Structures

- struct listNode
- struct listHead

Typedefs

- typedef struct listNode Node
- typedef struct listHead List

Functions

- List * initializeList (void(*printFunction)(void *toBePrinted), void(*deleteFunction)(void *toBeDeleted), int(*compareFunction)(const void *first, const void *second))
- Node * initializeNode (void *data)
- void insertFront (List *list, void *toBeAdded)
- void insertBack (List *list, void *toBeAdded)
- void deleteList (List *list)
- void insertSorted (List *list, void *toBeAdded)
- int deleteNodeFromList (List *list, void *toBeDeleted)
- void * getFromFront (List *list)
- void * getFromBack (List *list)
- void printForward (List *list)
- void printBackwards (List *list)
- void deleteListNode (void *toBeDeleted)
- int compare (const void *first, const void *second)
- void printNode (void *toBePrinted)

4.1.1 Detailed Description

File containing the function definitions of a doubly linked list.

Author

Michael Ellis

Date

January 2017

4.1.2 Typedef Documentation

4.1.2.1 List

```
typedef struct listHead List
```

Dummy head of the list. Contains no actual data on it beyond a pointer to the front and end of the list.

4.1.2.2 Node

```
typedef struct listNode Node
```

Node of a linked list. This list is doubly linked, meaning that it has points to both the node immediately in front of it, as well as the node immediately behind it.

4.1.3 Function Documentation

4.1.3.1 compare()

```
int compare (
            const void * first,
            const void * second )
```

User-defined comparison for two pointers to generic data. Must define an element of these pointers to use to compare the two pointers with each other.

Precondition

first and second must be comparable.

Parameters

first	pointer to data to be compared with second.
second	pointer to data to be compared with first.

Returns

for sorting purposes, <0 The element pointed by 'first' goes before the element pointed by 'second' 0 The element pointed by 'first' is equivalent to the element pointed by 'second' >0 The element pointed by 'first' goes after the element pointed by 'second'

4.1.3.2 deleteList()

```
void deleteList (
             List * list )
```

Deletes the entire linked list head to tail, starting with the nodes, followed by the list itself.

Precondition

'List' type must exist and be used in order to keep track of the linked list.

Parameters

list	pointer to the List-type dummy node

4.1.3.3 deleteListNode()

```
void deleteListNode (
             void * toBeDeleted )
```

User defined function to delete linked list node based on the incoming data.

Precondition

Data must not already be freed or NULL

Parameters

toBeDeleted	Pointer to generic data to be deleted in the list.

4.1.3.4 deleteNodeFromList()

```
int deleteNodeFromList (
            List * list,
            void * toBeDeleted )
```

Function to remove a node from the list and alter the pointers accordingly to not disrupt the order of the data structure.

Precondition

List must exist and have memory allocated to it

Postcondition

toBeDeleted will have its memory freed if it exists in the list.

Parameters

list	pointer to the dummy head of the list containing deleteFunction function pointer
toBeDeleted	pointer to data that is to be removed from the list

Returns

returns EXIT_SUCCESS on success, and EXIT_FAILURE when empty. Returns -1 when the node cannot be found.

4.1.3.5 getFromBack()

```
void* getFromBack (
            List * list )
```

Function to return the data at the back of the list.

Precondition

The list exists and has memory allocated to it

Parameters

list	pointer to the dummy head of the list containing the tail of the list

Returns

pointer to the data located at the tail of the list

4.1.3.6 getFromFront()

```
void* getFromFront (
            List * list )
```

Function to return the data at the front of the list.

Precondition

The list exists and has memory allocated to it

Parameters

list	pointer to the dummy head of the list containing the head of the list

Returns

pointer to the data located at the head of the list

4.1.3.7 initializeList()

```
List* initializeList (
            void(*)(void *toBePrinted) printFunction,
            void(*)(void *toBeDeleted) deleteFunction,
            int(*)(const void *first, const void *second) compareFunction )
```

Function to point the list head to the appropriate functions. Allocates memory to the struct.

Returns

pointer to the list head

Parameters

printFunction	function pointer to print a single node of the list
deleteFunction	function pointer to delete a single piece of data from the list
compareFunction	function pointer to compare two nodes of the list in order to test for equality or order

4.1.3.8 initializeNode()

```
Node* initializeNode (
            void * data )
```

Function for creating a node for a linked list. This node contains generic data and may be connected to other notes in a list.

Precondition

data should be of same size of void pointer on the users machine to avoid size conflicts. data must be valid. data must be cast to void pointer before being added.

Postcondition

data is valid to be added to a linked list

Returns

On success returns a node that can be added to a linked list. On failure, returns NULL.

Parameters

data	- is a generic pointer to any data type.

4.1.3.9 insertBack()

```
void insertBack (
            List * list,
            void * toBeAdded )
```

Inserts a Node to the back of a linked list. The list then updates accordingly to adhere to the ADT.

Precondition

'List' type must exist and be used in order to keep track of the linked list.

Parameters

list	pointer to the dummy head of the list
toBeAdded	a pointer to data that is to be added to the linked list

4.1.3.10 insertFront()

```
void insertFront (
            List * list,
            void * toBeAdded )
```

Inserts a Node to the front of a linked list. The list then updates accordingly to adhere to the ADT.

Precondition

'List' type must exist and be used in order to keep track of the linked list.

Parameters

list	pointer to the dummy head of the list
toBeAdded	a pointer to data that is to be added to the linked list

4.1.3.11 insertSorted()

```
void insertSorted (
            List * list,
            void * toBeAdded )
```

Uses the comparison function in the List struct to place the element in the appropriate position in the list. this is intended to be used from the beginning in order to keep the list completely sorted.

Precondition

List exists and has memory allocated to it. Node to be added is valid.

Postcondition

The node to be added will be placed immediately before or after the first occurrence of a related node

Parameters

list	a pointer to the dummy head of the list containing function pointers for delete and compare, as well as a pointer to the first and last element of the list.
toBeAdded	a pointer to data that is to be added to the linked list

4.1.3.12 printBackwards()

```
void printBackwards (
            List * list )
```

Function to print list from tail to head. This will utilize the list's printNode function pointer to print.

Precondition

List must exist, but does not have to have elements.

Parameters

list	Pointer to linked list dummy head.

4.1.3.13 printForward()

```
void printForward (
            List * list )
```

Function to print list from head to tail. This will utilize the list's printNode function pointer to print.

Precondition

List must exist, but does not have to have elements.

Parameters

list	Pointer to linked list dummy head.

4.1.3.14 printNode()

```
void printNode (
            void * toBePrinted )
```

User defined function to print an element of the list.

Precondition

Data must be able to be printed via a standard print function

Parameters

toBePrinted	pointer to the data that is to be printed. Taken from a data structure.