Dynamische Container C

DOKUMENTATION

LUKAS MOMBERG [11141259]
DENNIS GOßLER [11140150]

Inhaltsverzeichnis

Collection Error Codes

Wenn eine Funktion auf eine Collection fehlschlägt, können Folgende Fehlercodes zurückgegeben:

```
CollectionError
{
	// Successful / No Error
	CollectionNoError,

	// Some internal allocation failed
	CollectionOutOfMemory,

	// There is no data to pull.
	CollectionEmpty,

	// Specified index points not to any data.
	CollectionArrayIndexOutOfBounds,

	// Specified 'ElementSize' was Zero.
	CollectionNoElementSizeSpecified,

	// Specified 'data' is Null-Pointer
	CollectionElementIsNullPointer,

	//Dictionary specific
	CollectionKeyAlreadyExists

}
```

List

Nach der Initialisierung der Liste, könne Sie Daten mit der Add Funktion hintereinander speichern. Sollte die maximale Größe wird die Liste automatisch erweitert.

$[Größe_{neu} = 1.5 * (Größe_{alt} + 1)]$

Schritt	Pos .0	Pos.1	Pos.2	Pos.3	Pos.4	…	Eingabe	Ausgabe
0 – Voher	Null						-	-
1 – Allocation	\0	\0	\0	\0		…	4	-
2 – ADD	A	\0	\0	\0		…	A	-
3 – INSERT	A	\0	D	\0		…	2, D	-
4 – ADD	A	C	D	\0		…	C	-
5 – GET	A	C	D	\0		…	2	D
6 – REMOVE	A	C	\0	\0		…	2	-
7 – GET	A	C	\0	\0			2	NULL
7 – Deconstruct	Null						-	-

Codebeispiel:

```
List exampleList = EMPTYLIST;
ListInitialize(&exampleList, 4, sizeof(char*));

char* outputChar;


char* testString1 = calloc(2, sizeof(char));
memcpy(testString1, "A", 1 * sizeof(char));

char* testString2 = calloc(3, sizeof(char));
memcpy(testString2, "BB", 2 * sizeof(char));

char* testString3 = calloc(4, sizeof(char));
memcpy(testString3, "DDD", 3 * sizeof(char));


// Add functions
ListItemAdd(&exampleList, testString1);

ListItemInsertAt(&exampleList, 2, testString3);

ListItemAdd(&exampleList, testString2);

// Get value
ListItemGet(&exampleList, 2, &outputChar);

// Remove value
ListItemRemove(&exampleList, 2);

// Output is Null
ListItemGet(&exampleList, 2, &outputChar);


ListDestruction(&exampleList);
```

Dictionary

Das Dictionary wurde als ein Binärer Suchbaum implementiert. Dieser besitzt pro Eintrag einen „Key“ und eine „Value“. Beim Einfügen wird von der Wurzel ab entschieden, ob der jeweilige „Key“ größer oder kleiner ist. Wird eine passende Stelle gefunden wird der Datensatz dort gespeichert. „Key“ Duplikate sind nicht zulässig.

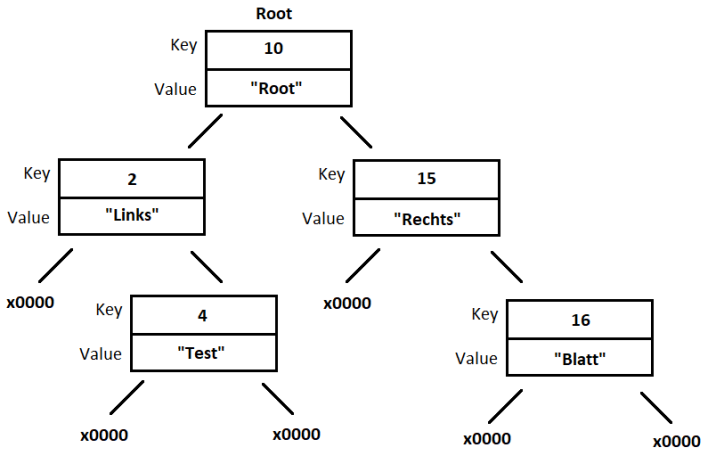

Abbildung 1

Codebeispiel:

```
Dictionary exampleDic = EMPTYDICTIONARY;
DictionaryInitialize(&exampleDic, sizeof(int*), sizeof(char*));

char* outputAddress;

int t0 = 10, t1 = 15, t2 = 2, t3 = 4, t4 = 16;

// Add functions
DictionaryAdd(&exampleDic, &t0, "Root");
DictionaryAdd(&exampleDic, &t1, "Rechts");
DictionaryAdd(&exampleDic, &t2, "Links");
DictionaryAdd(&exampleDic, &t3, "Test");
DictionaryAdd(&exampleDic, &t4, "Blatt");

// Get value
DictionaryGet(&exampleDic, &t4, &outputAddress);

// Remove value
DictionaryRemove(&exampleDic, &t2);

DictionaryDestroy(&exampleDic);
```

Stack

Daten werden gestapelt, der ältere Wert wird vom neuern verdeckt. Es kann immer ein Wert hinzugefügt werden, beim Entfernen wird der neueste Wert entfernt. First In Last Out (FILO). Diese Vorgänge sind sehr schnell und sind generell sicher.

Schritt	0x01	0x02	0x03	0x04	0x05	…	Eingabe	Ausgabe
0 – Voher	Null						-	-
1 – Allocation	\0	\0	\0	\0	\0	…	-	-
2 – Push	A	\0	\0	\0	\0	…	A	-
3 – Push	A	B	\0	\0	\0	…	B	-
4 – Push	A	B	C	\0	\0	…	C	-
5 – Pull	A	B	\0	\0	\0	…	-	C
6 – Pull	A	\0	\0	\0	\0	…	-	B
7 – Pull	Null						-	A

Funktionsname	Rückgabewert	Parameter	Nutzen
StackInitialize	void	Stack* stack unsigned int sizeOfSingleElement	Inizialisierung des Containers
StackClear	void	Stack* stack	Löschen aller Daten des Containers
StackPush	CollectionError	Stack* stack void* element	Fügt Wert zum Container hinzu
StackPull	CollectionError	Stack* stack void* element	Nimmt einen Wert aus dem Container

```
char x = 0;
CollectionError collectionError;
Stack stack;
StackInitialize(&stack, sizeof(char));

//---<Add>----------------------------------
char a[] = "Hello";

StackPush(&stack, &a[0]); // Add 'H'
StackPush(&stack, &a[1]); // Add 'e'
StackPush(&stack, &a[2]); // Add 'l'
StackPush(&stack, &a[3]); // Add 'l'
StackPush(&stack, &a[4]); // Add 'o'
//-----------------------------------------

//---<Pull>---------------------------------
StackPull(&stack, &x); // x now contains 'o'
StackPull(&stack, &x); // x now contains 'l'
StackPull(&stack, &x); // x now contains 'l'
StackPull(&stack, &x); // x now contains 'e'
StackPull(&stack, &x); // x now contains 'H'
//-----------------------------------------

//---<Clear>--------------------------------
collectionError = StackPull(&stack, &x); // Will return that the List is Empty
//-----------------------------------------
```

Queue

Daten werden in einer Liste gespeichert, der älteste Wert wird hier entnommen. First in First out (FIFO). Da Daten von Vorne entnommen werden entsteht ungenutzter Speicher. Hier ist zu Achten, dass dieser Speicher möglichst freigegeben wird, natürlich ist eine umbauen des Speichers bei jedem Zugriff nicht unbedingt Sinnvoll.

Schritt	0x01	0x02	0x03	0x04	0x05	…	Eingabe	Ausgabe
0 – Voher	Null						-	-
1 – Allocation	\0	\0	\0	\0	\0	…	-	-
2 – Push	A	\0	\0	\0	\0	…	A	-
3 – Push	A	B	\0	\0	\0	…	B	-
4 – Push	A	B	C	\0	\0	…	C	-
5 – Pull	A	B	C	\0	\0	…	-	A
6 – Pull	A	B	C	\0	\0	…	-	B
7 – Pull	Null						-	C

Funktionsname	Rückgabewert	Parameter	Nutzen
QueueInitialize	void	Queue* queue unsigned int sizeOfSingleElement	Inizialisierung des Containers
QueueClear	void	Queue* queue	Löschen aller Daten des Containers
QueuePush	CollectionError	Queue* queue void* element	Fügt Wert zum Container hinzu
QueuePull	CollectionError	Queue* queue void* element	Nimmt einen Wert aus dem Container

```
char x = 0;
CollectionError collectionError;
Queue queue;
QueueInitialize(&queue, sizeof(char));

//---<Add>-----------------------------------------
char a[6] = "ABCDE";

collectionError = QueuePush(&queue, &a[0]); // Add 'A' with optional errorCheckValue

QueuePush(&queue, &a[1]); // Add 'B'
QueuePush(&queue, &a[2]); // Add 'C'
QueuePush(&queue, &a[3]); // Add 'D'
QueuePush(&queue, &a[4]); // Add 'E'
//-----------------------------------------------------

//--<Pull>-----------------------------------------

QueuePull(&queue, &x); // x will contain 'A'
QueuePull(&queue, &x); // x will contain 'B'
QueuePull(&queue, &x); // x will contain 'C'
QueuePull(&queue, &x); // x will contain 'D'
QueuePull(&queue, &x); // x will contain 'E'
//-----------------------------------------------------

//---<Clear<-----------------------------------------
collectionError = QueuePull(&queue, &x);
//-----------------------------------------------------
```

LinkedList

Daten werden in Ketten-Elementen gespeichert. Jedes zwischen Element kennt seinen nächsten Nachbarn. Durch diese Kette kann man jedes Element ansprechen. Das Letze Element hat immer einen Null Wert, da dieser der Letze Wert ist und keinen nächsten Wert besitzt.

Schritt	Nr.1	Nr.2	Nr.3	Nr.4	Nr.5	…	Eingabe	Ausgabe
1	Null						-	-
	Null					…		
2 - Push	A					…	A	-
	Null					…		
3 - Push	A					…	B	-
	->	B				…		
		Null				…		
4 - Push	A					…	C	-
	->	B				…		
		->	C			…		
			Null			…		
5 – Pull(1)	A					…	-	B
	->	C				…		
		Null				…		

Funktionsname	Rückgabewert	Parameter	Nutzen
LinkedListInitialize	void	LinkedList* linkedList unsigned int sizeOfElement	Inizialisierung des Containers
LinkedListClear	void	LinkedList* linkedList	Löschen aller Daten des Containers
LinkedListInsert	CollectionError	LinkedList* linkedList unsigned int index void* element	Fügt Wert zum Container hinzu
LinkedListAddToEnd	CollectionError	LinkedList* linkedList void* element	Fügt Wert am Ende des Containers an
LinkedListRemoveAtIndex	CollectionError	LinkedList* linkedList unsigned int index void* element	Entnimmt Wert aus gegebener Position
LinkedListGetElement	CollectionError	LinkedList* linkedList unsigned int index void* element	Liest Wert aus gegebener Position
LinkedListGetNode	CollectionError	LinkedList* linkedList unsigned int index LinkedListNode** linkedListNode	Liest Knoten aus gegebener Position
LinkedListGetLastElement	CollectionError	LinkedList* linkedList LinkedListNode** linkedListNode	Liest letzten Knoten

```
LinkedList linkedList;

LinkedListInitialize(&linkedList, sizeof(int));

//---<Insert>-----------------------------------------
int data[] = { 10,20,30,40,50,60 };

LinkedListAddToEnd(&linkedList, &data[0]);
LinkedListAddToEnd(&linkedList, &data[1]);
LinkedListAddToEnd(&linkedList, &data[2]);
LinkedListAddToEnd(&linkedList, &data[3]);
LinkedListAddToEnd(&linkedList, &data[4]);
LinkedListAddToEnd(&linkedList, &data[5]);
//------------------------------------------------------

//---<Read>-------------------------------------------
int extractedData[6];

// Get elements form index
LinkedListGetElement(&linkedList, 0, &extractedData[0]);
LinkedListGetElement(&linkedList, 1, &extractedData[1]);
LinkedListGetElement(&linkedList, 2, &extractedData[2]);
LinkedListGetElement(&linkedList, 3, &extractedData[3]);
LinkedListGetElement(&linkedList, 4, &extractedData[4]);
LinkedListGetElement(&linkedList, 5, &extractedData[5]);
```

String

Diese Collection soll die Erweiterbarkeit unserer Container verdeutlichen. Dieser Container benutzt den Listcontainer als Basis und erweitert dessen Möglichkeiten. Zum Beispiel ist es möglich an den vorhandenen String ein Zeichen anzuhängen oder einen ganzen String.

„Das ist ein test“ + ‚!‘ ->„Das ist ein test!“ + „TEST“ ->	„Das ist ein test!TEST“
StringAdd(); StringConcat();	StringGetFullString();

Codebeispiel:

```
char* testChar = calloc(12, sizeof(char));
memcpy(testChar, "Test String", 11 * sizeof(char));

String testString = EMPTYSTRING;

StringInitialize(&testString, testChar);

char* outputChar;

//Gets 't'
StringCharGet(&testString, 3, &outputChar);

//Adds '!' to the end
StringCharAdd(&testString, '!');

//Gets '!'
StringCharGet(&testString, 11, &outputChar);

StringConcat(&testString, "TEST");

char* output = StringGetFullString(&testString);

StringDestruction(&testString);

free(output);
```