[C] Conteneurs et listes chaînées

Introduction

Les conteneurs sont des **listes doublement chaînées** qui permettent de stocker et de gérer un tableau d'éléments de manière dynamique et optimisé.

Concepts

Conteneur

Une **conteneur** est une structure de donnée contenant 4 éléments :

- Un pointeur sur le premier noeud.
- Un pointeur sur le dernier noeud.
- Un entier représentant le nombre de noeud présent dans le conteneur.
- Un pointeur sur la fonction de destruction des items.

Noeud

Un **noeud** est une structure de donnée contenant 3 éléments :

- Un pointeur sur le noeud précédent.
- Un pointeur sur le noeud suivant.
- Un pointeur sur l'item à stocker.

Item

L'**item** est l'élément à stocker dans le noeud.

Lors de son implémentation dans le noeud, il est converti au format générique **void***.

Bibliothèque

container.c

```
/*
 * container.c
 *
 *  Created on: 16 mars 2022
 *      Author: Thierry Boyer
 */



#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "container.h"

/**
 * @brief type definition of node structure
 * □□  Node component: Hidden structure
 */
typedef struct s_node{
□struct s_node *m_pPrev;□□/* Pointer to the previous node in the current list */
□struct s_node *m_pNext;□□/* Pointer to the next node in the current list□□*/
□□□ void *m_pItem;□□/* Pointer to the item stored at this node in the current list */
}t_node;

/* Node component: Hidden functions-------------------------------------------------------------*/
/**
 * @brief Node "constructor".
 *
 * @param [in] pPrev pointer to the previous node in the linked list.
 * @param [in] pNext pointer to the next node in the linked list.
 * @param [in] pItem pointer to the item to be linked to the node.
 * @return t_node* pointer to the mewly created node.
 */
t_node*NodeNew(t_node*pPrev, t_node*pNext, void*pItem){
□t_node*pNewNode=(t_node*)malloc(sizeof(t_node));
```

```
  assert(pNewNode);
  *pNewNode=(t_node){
    .m_pPrev = pPrev,
    .m_pNext = pNext,
    .m_pItem = pItem
  };
  if(pPrev) pPrev->m_pNext = pNewNode;
  if(pNext) pNext->m_pPrev = pNewNode;
  return pNewNode;
}

/**
 * @brief Node "destructor".
 *   This version will return the next node to the deleted current node.
 *
 * @param [in] pNodeToDel pointer to the node to delete.
 * @param [in] pDeleteFunc pointer to the item destruction function.
 * @return t_node* pointer to the next node to this one.
 */
t_node*NodeDelReturnNext(t_node*pNodeToDel, t_ptfV pDeleteFunc){
  if(pDeleteFunc) pDeleteFunc(pNodeToDel->m_pItem);
  else free(pNodeToDel->m_pItem);
  t_node*pNextNode=pNodeToDel->m_pNext;
  if(pNodeToDel->m_pPrev) pNodeToDel->m_pPrev->m_pNext=pNodeToDel->m_pNext;
  if(pNodeToDel->m_pNext) pNodeToDel->m_pNext->m_pPrev=pNodeToDel->m_pPrev;
  return pNextNode;
}

/**
 * @brief Node "destructor".
 *   This version will return the next node to the deleted current node.
 *   The associated item will be not destroyed by this function!
 *
 * @param pNodeToDel  pointer to the node to delete.
 * @return t_node* pointer to the next node to this one.
 */
t_node*NodeDelOnlyReturnNext(t_node*pNodeToDel){
  t_node*pNextNode=pNodeToDel->m_pNext;
  if(pNodeToDel->m_pPrev) pNodeToDel->m_pPrev->m_pNext=pNodeToDel->m_pNext;
  if(pNodeToDel->m_pNext) pNodeToDel->m_pNext->m_pPrev=pNodeToDel->m_pPrev;
```

```
    return pNextNode;
}

/**
 * @brief Node "destructor".
 *    This version will return the previous node to the deleted current node.
 *    The associated item will be not destroyed by this function!
 *
 * @param pNodeToDel  pointer to the node to delete.
 * @return t_node* pointer to the previous node to this one.
 */
t_node*NodeDelOnlyReturnPrev(t_node*pNodeToDel){
    t_node*pPrevNode=pNodeToDel->m_pPrev;
    if(pNodeToDel->m_pPrev) pNodeToDel->m_pPrev->m_pNext=pNodeToDel->m_pNext;
    if(pNodeToDel->m_pNext) pNodeToDel->m_pNext->m_pPrev=pNodeToDel->m_pPrev;
    return pPrevNode;
}




/**
 * @brief type definition of container structure.
 *    Container component: Hidden structure.
 */
struct s_container{
    t_node *m_pHead;    /* Pointer to the first node of the container list: also known as HEAD of list. */
    t_node *m_pTail;    /* Pointer to the  last node of the container list: also known as TAIL of list. */
    size_t  m_szCard;   /* Number of total nodes in container: also known as CARDINAL of list.   */
    t_ptfV  m_pDeleteItemFunc; /* Function pointer to the function responsible for item destruction.   */
};

/* Container component: Public functions implementation------------------------------------------------------------*/

/**
 * @brief Container "constructor".
 *
 * @param [in] pDeleteItemFunc  function pointer to the function responsible of
 *        properly deleting items in stored in container.
 * @return t_container* pointer on the newly created container.
```

```
 */
t_container* ContainerNew(t_ptfV pDeleteItemFunc){
□t_container*pContainer=(t_container*)malloc(sizeof(t_container));
□assert(pContainer);
□*pContainer=(t_container){
□□.m_pDeleteItemFunc = pDeleteItemFunc,
□};
□return pContainer;
}

/**
 * @brief  Container "destructor".
 *
 * @param [in] pContainer pointer to the container to destroy.
 * @return t_container* NULL.
 */
t_container* ContainerDel(t_container *pContainer){
□free(ContainerFlush(pContainer));
□return NULL;
}

/**
 * @brief □Flushing the container by deleting all items and nodes.
 * □□□The container structure is not destroyed !
 *
 * @param [in] pContainer pointer to the container to flush.
 * @return t_container* the flushed container.
 */
t_container*ContainerFlush(t_container *pContainer){
□assert(pContainer);
□while(pContainer->m_pHead){
□□pContainer->m_pHead=NodeDelReturnNext(pContainer->m_pHead, pContainer->m_pDeleteItemFunc);
□□pContainer->m_szCard--;
□}
□assert(pContainer->m_szCard==0);
□pContainer->m_pTail=NULL;
□return pContainer;
}

/**
```

```
 * @brief Returns the number of elements in container.
 *
 * @param pContainer pointer to the container to get cardinal.
 * @return size_t number of elements.
 */
size_t ContainerCard(const t_container*pContainer){
  assert(pContainer);
  return pContainer->m_szCard;
}

/**
 * @brief Append a item at the end of the container.
 *
 * @param [in] pContainer pointer to the container to append to.
 * @param [in] pItem pointer to the item to append.
 * @return void* pointer to the item that was appended.
 */
void*ContainerPushback(t_container*pContainer, void*pItem){
  assert(pContainer);
  if(pContainer->m_szCard==0){
    assert(pContainer->m_pHead==NULL);
    assert(pContainer->m_pTail==NULL);
    pContainer->m_pHead=pContainer->m_pTail=NodeNew(NULL, NULL, pItem);
  }
  else{
    pContainer->m_pTail=NodeNew(pContainer->m_pTail, NULL, pItem);
  }
  pContainer->m_szCard++;
  assert(pContainer->m_pTail->m_pItem==pItem);
  return pContainer->m_pTail->m_pItem;
}

/**
 * @brief Append a item at the beginning of the container.
 *
 * @param [in] pContainer pointer to the container to append to.
 * @param [in] pItem pointer to the item to append.
 * @return void* pointer to the item that was appended.
 */
void*ContainerPushfront(t_container*pContainer, void*pItem){
```

```
    assert(pContainer);
    if(pContainer->m_szCard==0){
        assert(pContainer->m_pHead==NULL);
        assert(pContainer->m_pTail==NULL);
        pContainer->m_pHead=pContainer->m_pTail=NodeNew(NULL, NULL, pItem);
    }
    else{
        pContainer->m_pHead=NodeNew(NULL,pContainer->m_pHead,pItem);
    }
    pContainer->m_szCard++;
    assert(pContainer->m_pHead->m_pItem==pItem);
    return pContainer->m_pHead->m_pItem;
}

/**
 * @brief Insert a item in the container at index.
 *
 * @param [in] pContainer pointer to the container to insert into.
 * @param [in] pItem pointer to the item to insert.
 * @param [in] sAt index to insert at.
 * @return void* pointer to the item that was inserted.
 */
void*ContainerPushat(t_container*pContainer, void*pItem, size_t szAt){
    assert(pContainer);
    assert((int)szAt>=0);
    assert(szAt<=pContainer->m_szCard);

    if(szAt==0) return ContainerPushfront(pContainer, pItem);
    if(szAt==pContainer->m_szCard) return ContainerPushback(pContainer, pItem);

    t_node*pInsert;
    if(szAt<=pContainer->m_szCard/2){
        pInsert=pContainer->m_pHead;
        for (size_t k = 0; k < szAt; k++){
            pInsert=pInsert->m_pNext;
        }
    }
    else{
        pInsert=pContainer->m_pTail;
        for (size_t k = pContainer->m_szCard-1; k>szAt; k--) {
```

```
        pInsert=pInsert->m_pPrev;
    }
}
pInsert=NodeNew(pInsert->m_pPrev, pInsert, pItem);
pContainer->m_szCard++;
assert(pInsert->m_pItem==pItem);
return pInsert->m_pItem;
}

//void*ContainerPushat(t_container*pContainer, void*pItem, size_t szAt){
//    assert(pContainer);
//    assert((int)szAt >=0);
//    assert(szAt <= pContainer->m_szCard);
//
//    if(szAt == 0) return ContainerPushfront(pContainer, pItem);
//    if(szAt == pContainer->m_szCard) return ContainerPushback(pContainer, pItem);
//
//    t_node * pInsert;
//
//    for(size_t k = 0; k<szAt;k++){
//        pInsert =pInsert->m_pNext;
//    }
//
//    pInsert = NodeNew(pInsert->m_pPrev, pInsert, pItem);
//    pContainer->m_szCard++;
//    assert(pInsert->m_pItem == pItem);
//    return pInsert->m_pItem;
//}



/**
 * @brief Get the last item from the container. The item persists in
```

```
 * □□ the container!
 *
 * @param pContainer pointer to the container to get from.
 * @return void* pointer to the got item.
 */
void*ContainerGetback(const t_container*pContainer){
□assert(pContainer);
□assert(pContainer->m_szCard);
□return pContainer->m_pTail->m_pItem;
}

/**
 * @brief Get the first item from the container. The item persists in
 * □□ the container!
 *
 * @param pContainer pointer to the container to get from.
 * @return void* pointer to the got item.
 */
void*ContainerGetfront(const t_container*pContainer){
□assert(pContainer);
□assert(pContainer->m_szCard);
□return pContainer->m_pHead->m_pItem;
}

/**
 * @brief Get the item stored at index from the container. The
 * □□ item persists in the container!
 *
 * @param [in] pContainer pointer to the container to get from.
 * @param [in] sAt index to get from.
 * @return void* pointer to the got item.
 */
void*ContainerGetat(const t_container*pContainer, size_t szAt){
□assert(pContainer);
□assert((int)szAt>=0);
□assert(szAt<pContainer->m_szCard);□/* WARNING!!! szAt MUST BE STRICTLY LESSER THAN m_szCard!!! */

□if(szAt==0) return ContainerGetfront(pContainer);

□/* WARNING!!! m_szCard-1 BECAUSE szAt MUST BE STRICTLY LESSER THAN m_szCard!!! */
```

```
    if(szAt==pContainer->m_szCard-1) return ContainerGetback(pContainer);

    t_node*pGet;
    if(szAt<=pContainer->m_szCard/2){
        pGet=pContainer->m_pHead;
        for (size_t k = 0; k < szAt; k++){
            pGet=pGet->m_pNext;
        }
    }
    else{
        pGet=pContainer->m_pTail;
        for (size_t k = pContainer->m_szCard-1; k>szAt; k--) {
            pGet=pGet->m_pPrev;
        }
    }
    return pGet->m_pItem;
}

/**
 * @brief Extract the last item from the container, and return it to caller.
 *    The item is removed from the container and the corresponding node is deleted.
 *
 * @param pContainer pointer to the container to extract from.
 * @return void* pointer to the popped item.
 */
void*ContainerPopback(t_container*pContainer){
    assert(pContainer);
    assert(pContainer->m_szCard);

    void*pReturnItem=pContainer->m_pTail->m_pItem;
    if(pContainer->m_pHead==pContainer->m_pTail) pContainer->m_pHead=NULL;
    pContainer->m_pTail=NodeDelOnlyReturnPrev(pContainer->m_pTail);
    pContainer->m_szCard--;
    return pReturnItem;
}

/**
 * @brief Extract the first item from the container, and return it to caller.
 *    The item is removed from the container and the corresponding node is deleted.
 *
```

```
 * @param [in] pContainer pointer to the container to extract from.
 * @return void* pointer to the popped item.
 */
void*ContainerPopfront(t_container*pContainer){
	assert(pContainer);
	assert(pContainer->m_szCard);

	void*pReturnItem=pContainer->m_pHead->m_pItem;
	if(pContainer->m_pTail==pContainer->m_pHead) pContainer->m_pTail=NULL;
	pContainer->m_pHead=NodeDelOnlyReturnNext(pContainer->m_pHead);
	pContainer->m_szCard--;
	return pReturnItem;
}

/**
 * @brief Extract the item located at index from the container, and return it to caller.
 * 		The item is removed from the container and the corresponding node is deleted.
 *
 * @param [in] pContainer pointer to the container to extract from.
 * @param [in] sAt index to extract from.
 * @return void* pointer to the popped item.
 */
void*ContainerPopat(t_container*pContainer, size_t szAt){
	assert(pContainer);
	assert((int)szAt>=0);
	assert(szAt<pContainer->m_szCard);	/* WARNING!!! szAt MUST BE STRICTLY LESSER THAN m_szCard!!! */

	if(szAt==0) return ContainerPopfront(pContainer);

	/* WARNING!!! m_szCard-1 BECAUSE szAt MUST BE STRICTLY LESSER THAN m_szCard!!! */
	if(szAt==pContainer->m_szCard-1) return ContainerPopback(pContainer);

	t_node*pPop;
	if(szAt<=pContainer->m_szCard/2){
		pPop=pContainer->m_pHead;
		for (size_t k = 0; k < szAt; k++){
			pPop=pPop->m_pNext;
		}
	}
	else{
```

```
    pPop=pContainer->m_pTail;
    for (size_t k = pContainer->m_szCard-1; k>szAt; k--) {
        pPop=pPop->m_pPrev;
    }
  }
  void*pReturnItem=pPop->m_pItem;
  NodeDelOnlyReturnNext(pPop);
  pContainer->m_szCard--;
  return pReturnItem;
}

/**
 * @brief Parsing the container from front to back, and for each item, calling
 *    the parsing function with parameters the pointer to the corresponding
 *    item and the pParam parameter.
 *    The parsing is stopped if the parsing function returns a non null value,
 *    and in this case, the function returns the corresponding item to the caller.
 *
 * @param [in] pContainer pointer to the container to parse.
 * @param [in] pParseFunc pointer to the called function for each parsed item.
 * @param [in] pParam parameter directly passed to the called function during parsing.
 * @return void* NULL in case of a complete parsing,
 *     pointer to the item in case of interrupted parsing.
 */
void*ContainerParse(const t_container*pContainer, t_ptfVV pParseFunc, void*pParam){
  assert(pContainer);
  assert(pParseFunc);

  t_node*pParse=pContainer->m_pHead;
  while(pParse){
    if(pParseFunc(pParse->m_pItem, pParam)) return pParse->m_pItem;
    pParse=pParse->m_pNext;
  }
  return NULL;
}

/**
 * @brief Parsing the container from front to back, and for each item, calling
 *    the parsing function with parameters the pointer to the corresponding
 *    item and the pParam parameter.
```

```
 * □□ In case of a non null return value from called function, the item and the
 * □□ corresponding node are destroyed. The parsing is then resumed to the next
 * □□ item and the same scenario takes place, until container back is reached.
 *
 * @param [in] pContainer pointer to the container to parse.
 * @param [in] pParseFunc pointer to the called function for each parsed item.
 * @param [in] pParam parameter directly passed to the called function during parsing.
 * @return void* NULL (complete parsing).
 */
void*ContainerParseDelIf(t_container*pContainer, t_ptfVV pParseFunc, void*pParam){
□assert(pContainer);
□assert(pParseFunc);

□t_node*pParse=pContainer->m_pHead;
□while(pParse){
□□if(pParseFunc(pParse->m_pItem, pParam)){
□□□if(pContainer->m_pHead==pParse) pContainer->m_pHead=pParse->m_pNext;
□□□if(pContainer->m_pTail==pParse) pContainer->m_pTail=pParse->m_pPrev;
□□□pParse=NodeDelOnlyReturnNext(pParse);
□□□pContainer->m_szCard--;
□□}
□□else{
□□□pParse=pParse->m_pNext;
□□}
□}
□return NULL;
}

/**
 * @brief Parsing the container A, and for each of its items, calls the intersection
 * □□ function with each item of the container B. In case of matched intersection,
 * □□ both item and node in container A are destroyed, and both item and node
 * □□ in container B are also destroyed. In this case, the parsing is resumed
 * □□ to the next item of container A, and the same scenario takes place by parsing
 * □□ container B from his front to back. Parsing ending when all items in each
 * □□ container has been intersected with each other.
 *
 * @param [in] pContainerA pointer the the first container to intersect .
 * @param [in] pContainerB pointer the the second container to intersect .
 * @param [in] pIntersectFunc pointer to the called intersect function for each parsed items.
```

```
 * @param [in] pCallbackFunc pointer to the callback function in case of intersection.
 * @param [in] pParam parameter directly passed to the callback function.
 * @return void*  NULL (complete parsing).
 */
void*ContainerIntersectDelIf(t_container*pContainerA, t_container*pContainerB, t_ptfVV pIntersectFunc,t_ptfVV
pCallbackFunc, void*pParam){

□assert(pContainerA);
□assert(pContainerB);
□assert(pIntersectFunc);

□t_node □*pParseA=pContainerA->m_pHead,
□□□*pParseB=pContainerB->m_pHead;

□int hasIntersected;

□while(pParseA && pParseB){
□□hasIntersected=0;□/* No intersection has occured at this time */
□□while(pParseA && pParseB){
□□□if(pIntersectFunc(pParseA->m_pItem, pParseB->m_pItem)){
□□□□/* Notifying the instersection between the two items */
□□□□hasIntersected=1;

□□□□/* Processing callback if exist */
□□□□if(pCallbackFunc) pCallbackFunc(pParam, NULL);

□□□□/* Destroying current item of container A */
□□□□if(pContainerA->m_pHead==pParseA) pContainerA->m_pHead=pParseA->m_pNext;
□□□□if(pContainerA->m_pTail==pParseA) pContainerA->m_pTail=pParseA->m_pPrev;
□□□□pParseA=NodeDelReturnNext(pParseA, pContainerA->m_pDeleteItemFunc);
□□□□pContainerA->m_szCard--;

□□□□/* Destroying current item of container B */
□□□□if(pContainerB->m_pHead==pParseB) pContainerB->m_pHead=pParseB->m_pNext;
□□□□if(pContainerB->m_pTail==pParseB) pContainerB->m_pTail=pParseB->m_pPrev;
□□□□pParseB=NodeDelReturnNext(pParseB, pContainerB->m_pDeleteItemFunc);
□□□□pContainerB->m_szCard--;
□□□}
□□□else{
□□□□pParseB=pParseB->m_pNext; /* Processing the next node of container B */
```

```
		}
	}
	/* Processing the next node of container A */
	if(!hasIntersected) pParseA=pParseA->m_pNext;
	/* Reseting the parsing pointer of container B to his header for a new scan */
	pParseB=pContainerB->m_pHead;
	}

	return NULL;
}


/*************************/


void*ContainerInsert(t_container*pContainer, t_ptfVV pPredicaFunc, void*pItem){
    assert(pContainer);
    assert(pPredicaFunc);

    t_node*pInsert=pContainer->m_pHead;
    size_t szAt=0;

    while(pInsert!=NULL){
    	if(pPredicaFunc(pItem, pInsert->m_pItem)){
    		return ContainerPushat(pContainer, pItem, szAt);
    	}
    	pInsert=pInsert->m_pNext;
    	szAt++;
    }
    return ContainerPushback(pContainer, pItem);
}

void*ContainerInsertUnic(t_container*pContainer, t_ptfVV pPredicaFunc, void*pItem){
    assert(pContainer);
    assert(pPredicaFunc);

    t_node*pInsert=pContainer->m_pHead;
    size_t szAt=0;

    while(pInsert!=NULL){
```

```
    switch((long)pPredicaFunc(pItem, pInsert->m_pItem)){
    case 0: /* Continue parsing */
        pInsert=pInsert->m_pNext;
        szAt++;
        break;
    case 1: /* Inserting here */
        return ContainerPushat(pContainer, pItem, szAt);
        // no break;
    case -1: /* Deleting item */
        if(pContainer->m_pDeleteItemFunc) pContainer->m_pDeleteItemFunc(pItem);
        else free(pItem);
        return NULL;
        // no break;
    default:
        assert(NULL); /* Never enter in default case ! */
        break;
    }
    }
    return ContainerPushback(pContainer, pItem);
}

void*ContainerSerialize(t_container*pContainer, t_ptfVV pSerializeFunc){



return NULL;
}

void*ContainerDeserialize(t_container*pContainer, t_ptfVV pDeserializeFunc){
return NULL;
}
```

container.h

```
/*
 * container.h
 *
 *  Created on: 16 mars 2022
 *      Author: Thierry Boyer
 */
```

```
/**
 * @brief Function pointers definition.
 */
typedef void*(*t_ptfV)(void*);□□□// For functions like:  void*()(void*)
typedef void*(*t_ptfVV)(void*, void*);□// For functions like:  void*()(void*, void*)



/**
 * @brief Container type definition: "hidden structure" or "incomplete structure" definition.
 */
typedef struct s_container t_container;

/**
 * @brief  Container public functions declaration.
 */

/**
 * @brief Container "constructor".
 *
 * @param [in] pDeleteItemFunc □function pointer to the function responsible of
 * □□□□□□□□properly deleting items in stored in container.
 * @return t_container* pointer on the newly created container.
 */
t_container*ContainerNew(t_ptfV pDeleteItemFunc);

/**
 * @brief  Container "destructor".
 *
 * @param [in] pContainer pointer to the container to destroy.
 * @return t_container* NULL.
 */
t_container*ContainerDel(t_container *pContainer);

/**
 * @brief □Flushing the container by deleting all items and nodes.
 * □□□The container structure is not destroyed !
 *
 * @param [in] pContainer pointer to the container to flush.
 * @return t_container* the flushed container.
```

```
 */
t_container*ContainerFlush(t_container *pContainer);

/**
 * @brief Returns the number of elements in container.
 *
 * @param pContainer pointer to the container to get cardinal.
 * @return size_t number of elements.
 */
size_t ContainerCard(const t_container*pContainer);

/**
 * @brief Append a item at the end of the container.
 *
 * @param [in] pContainer pointer to the container to append to.
 * @param [in] pItem pointer to the item to append.
 * @return void* pointer to the item that was appended.
 */
void*ContainerPushback(t_container*pContainer, void*pItem);

/**
 * @brief Append a item at the beginning of the container.
 *
 * @param [in] pContainer pointer to the container to append to.
 * @param [in] pItem pointer to the item to append.
 * @return void* pointer to the item that was appended.
 */
void*ContainerPushfront(t_container*pContainer, void*pItem);

/**
 * @brief Insert a item in the container at index.
 *
 * @param [in] pContainer pointer to the container to insert into.
 * @param [in] pItem pointer to the item to insert.
 * @param [in] sAt index to insert at.
 * @return void* pointer to the item that was inserted.
 */
void*ContainerPushat(t_container*pContainer, void*pItem, size_t szAt);

/**
```

```
 * @brief Get the last item from the container. The item persists in
 * □□ the container!
 *
 * @param pContainer pointer to the container to get from.
 * @return void* pointer to the got item.
 */
void*ContainerGetback(const t_container*pContainer);

/**
 * @brief Get the first item from the container. The item persists in
 * □□ the container!
 *
 * @param pContainer pointer to the container to get from.
 * @return void* pointer to the got item.
 */
void*ContainerGetfront(const t_container*pContainer);

/**
 * @brief Get the item stored at index from the container. The
 * □□ item persists in the container!
 *
 * @param [in] pContainer pointer to the container to get from.
 * @param [in] sAt index to get from.
 * @return void* pointer to the got item.
 */
void*ContainerGetat(const t_container*pContainer, size_t szAt);

/**
 * @brief Extract the last item from the container, and return it to caller.
 * □□ The item is removed from the container and the corresponding node is deleted.
 * @param pContainer pointer to the container to extract from.
 * @return void* pointer to the popped item.
 */
void*ContainerPopback(t_container*pContainer);

/**
 * @brief Extract the first item from the container, and return it to caller.
 * □□ The item is removed from the container and the corresponding node is deleted.
 * @param [in] pContainer pointer to the container to extract from.
 * @return void* pointer to the popped item.
```

```
 */
void*ContainerPopfront(t_container*pContainer);

/**
 * @brief Extract the item located at index from the container, and return it to caller.
 * □□ The item is removed from the container and the corresponding node is deleted.
 * @param [in] pContainer pointer to the container to extract from.
 * @param [in] sAt index to extract from.
 * @return void* pointer to the popped item.
 */
void*ContainerPopat(t_container*pContainer, size_t szAt);

/**
 * @brief Parsing the container from front to back, and for each item, calling
 * □□ the parsing function with parameters the pointer to the corresponding
 * □□ item and the pParam parameter.
 * □□ The parsing is stopped if the parsing function returns a non null value,
 * □□ and in this case, the function returns the corresponding item to the caller.
 *
 * @param [in] pContainer pointer to the container to parse.
 * @param [in] pParseFunc pointer to the called function for each parsed item.
 * @param [in] pParam parameter directly passed to the called function during parsing.
 * @return void* NULL in case of a complete parsing,
 * □□□□ pointer to the item in case of interrupted parsing.
 */
void*ContainerParse(const t_container*pContainer, t_ptfVV pParseFunc, void*pParam);

/**
 * @brief Parsing the container from front to back, and for each item, calling
 * □□ the parsing function with parameters the pointer to the corresponding
 * □□ item and the pParam parameter.
 * □□ In case of a non null return value from called function, the item and the
 * □□ corresponding node are destroyed. The parsing is then resumed to the next
 * □□ item and the same scenario takes place, until container back is reached.
 *
 * @param [in] pContainer pointer to the container to parse.
 * @param [in] pParseFunc pointer to the called function for each parsed item.
 * @param [in] pParam parameter directly passed to the called function during parsing.
 * @return void* NULL (complete parsing).
 */
```

```
void*ContainerParseDelIf(t_container*pContainer, t_ptfVV pParseFunc, void*pParam);

/**
 * @brief Parsing the container A, and for each of its items, calls the intersection
 * □□ function with each item of the container B. In case of matched intersection,
 * □□ both item and node in container A are destroyed, and both item and node
 * □□ in container B are also destroyed. In this case, the parsing is resumed
 * □□ to the next item of container A, and the same scenario takes place by parsing
 * □□ container B from his front to back. Parsing ending when all items in each
 * □□ container has been intersected with each other.
 *
 * @param [in] pContainerA pointer the the first container to intersect.
 * @param [in] pContainerB pointer the the second container to intersect.
 * @param [in] pIntersectFunc pointer to the called intersect function for each parsed items.
 * @param [in] pCallbackFunc pointer to the callback function in case of intersection.
 * @param [in] pParam parameter directly passed to the callback function.
 * @return void*  NULL (complete parsing).
 */
void*ContainerIntersectDelIf(t_container*pContainerA, t_container*pContainerB, t_ptfVV pIntersectFunc,t_ptfVV
pCallbackFunc, void*pParam);



void*ContainerInsert(t_container*pContainer, t_ptfVV pPredicaFunc, void*pItem);

void*ContainerInsertUnic(t_container*pContainer, t_ptfVV pPredicaFunc, void*pItem);

void*ContainerSerialize(t_container*pContainer, t_ptfVV pSerializeFunc);

void*ContainerDeserialize(t_container*pContainer, t_ptfVV pDeserializeFunc);
```

Revision #2
Created 27 December 2023 09:02:59 by Elieroc
Updated 27 December 2023 22:09:18 by Elieroc