#ifndef HASH\_TABLE\_ENUMERATOR

#define HASH\_TABLE\_ENUMERATOR

#include "Enumerator.h"

#include "Exceptions.h"

#include "HashTable.h"

template <typename T>

class HashTableEnumerator : public Enumerator<T>

{

private:

unsigned long bucket = 0; // the current bucket during the enumeration process

OULinkedListEnumerator<T>\* chainEnumerator = NULL; // used to move through the linked list of the current bucket

HashTable<T>\* hashTable = NULL; // pointer to the HashTable being enumerated

// you may add additional member variables and functions here to support the operation of your code

OULinkedListEnumerator<T> tempEnum = NULL; // get a temporary current bucket's linked list

OULinkedListEnumerator<T> currentBucketEnum = NULL; // get the current bucket's linked list

public:

HashTableEnumerator(HashTable<T>\* hashTable); // constructor needs a pointer to the HashTable to be enumerated

virtual ~HashTableEnumerator();

bool hasNext() const; // true if there are elements that have not yet been returned via next()

T next(); // throws ExceptionEnumerationBeyondEnd if no next item is available

T peek() const; // throws ExceptionEnumerationBeyondEnd if no next item is available

unsigned long getBucketEnum();

};

// put implementation for HashTableEnumerator here

// constructor needs a pointer to the HashTable to be enumerated

template <typename T>

HashTableEnumerator<T>::HashTableEnumerator(HashTable<T>\* hashTable) {

if (hashTable->getSize() == 0) {

throw new ExceptionHashTableAccess();

}

else {

this->hashTable = hashTable;

}

// find the first non-empty bucket

unsigned long firstBucket = bucket;

/\*if (this->hashTable->table[bucket]->getSize() != 0) {

//OULinkedListEnumerator<T> tempEnum = this->hashTable->table[bucket]->enumerator();

//OULinkedListEnumerator<T> currentBucketEnum = tempEnum;

currentBucketEnum = this->hashTable->table[bucket]->enumerator();

this->chainEnumerator = &currentBucketEnum; // copy memory address

\*this->chainEnumerator = currentBucketEnum; // copy data

}

else \*/

if (this->hashTable->table[bucket]->getSize() == 0) {

do {

firstBucket++;

/\*if (this->hashTable->table[bucket]->getSize() != 0) {

currentBucketEnum = this->hashTable->table[bucket]->enumerator();

this->chainEnumerator = &currentBucketEnum; // copy memory address

\*this->chainEnumerator = currentBucketEnum; // copy data

}\*/

} while ((this->hashTable->table[firstBucket]->getSize() == 0) && firstBucket < this->hashTable->getBaseCapacity());

}

if (firstBucket < this->hashTable->getBaseCapacity()) {

bucket = firstBucket;

}

else { // empty table

throw new ExceptionHashTableAccess();

}

currentBucketEnum = this->hashTable->table[bucket]->enumerator();

this->chainEnumerator = &currentBucketEnum; // copy memory address

\*this->chainEnumerator = currentBucketEnum; // copy data

// empty hash table

if (chainEnumerator == NULL) {

throw new ExceptionHashTableAccess();

}

}

template <typename T>

HashTableEnumerator<T>::~HashTableEnumerator() {

chainEnumerator = NULL;

hashTable = NULL;

}

template <typename T>

unsigned long HashTableEnumerator<T>::getBucketEnum() {

return bucket;

}

// true if there are elements that have not yet been returned via next()

template <typename T>

bool HashTableEnumerator<T>::hasNext() const {

bool hasNextStatus = false;

// check if there is next on the same bucket

if (chainEnumerator->hasNext() == true) {

hasNextStatus = true;

}

else if (chainEnumerator->hasNext() == false) {

// check other bucket(s)

unsigned long nextBucket = 0;

nextBucket = bucket + 1;

OULinkedListEnumerator<T>\* nextChainEnum = NULL;

/\* do {

// look into the next bucket

nextBucket++;

if (nextBucket < hashTable->getBaseCapacity()) {

OULinkedListEnumerator<T> currentBucketEnum = this->hashTable->table[nextBucket]->enumerator();

nextChainEnum = &currentBucketEnum;

//if (\*hashTable->table[bucket]->enumerator()->hasNext() == true) {

if (nextChainEnum->hasNext() == true) {

hasNextStatus = true;

}

}

} while (nextBucket < hashTable->getBaseCapacity());

\*/

while (nextBucket < hashTable->getBaseCapacity() && hasNextStatus == false) {

OULinkedListEnumerator<T> currentBucketEnum = this->hashTable->table[nextBucket]->enumerator();

nextChainEnum = &currentBucketEnum;

//if (\*hashTable->table[bucket]->enumerator()->hasNext() == true) {

if (nextChainEnum->hasNext() == true) {

hasNextStatus = true;

}

else {

// look into the next bucket

nextBucket++;

}

}

}

return hasNextStatus;

}

// throws ExceptionHashTableAccess if no next item is available

template <typename T>

T HashTableEnumerator<T>::next() {

// bool nextStatus = false;

// T\* nextData = NULL;

T nextData;

//if (chainEnumerator->hasNext() == true) {

if (hasNext() == true) {

if (chainEnumerator->hasNext() == true) {

//T data = chainEnumerator->next();

//nextData = chainEnumerator->next();

//nextStatus = true;

// nextData = data;

// \*nextData = data;

nextData = chainEnumerator->next();

// check if next data is in different bucket

if (hasNext()) {

if (chainEnumerator->hasNext() == false) {

// check other bucket(s)

unsigned long nextBucket = 0;

nextBucket = bucket + 1;

OULinkedListEnumerator<T>\* nextChainEnum = NULL;

bool hasNextStatus = false;

while (nextBucket < hashTable->getBaseCapacity() && hasNextStatus == false) {

OULinkedListEnumerator<T> currentBucketEnum = this->hashTable->table[nextBucket]->enumerator();

nextChainEnum = &currentBucketEnum;

//if (\*hashTable->table[bucket]->enumerator()->hasNext() == true) {

if (nextChainEnum->hasNext() == true) {

bucket = nextBucket;

hasNextStatus = true;

}

else {

// look into the next bucket

nextBucket++;

}

}

}

}

// return nextData;

}

else {

// last bucket ended, update chainEnum to the new bucket

unsigned long nextBucket = bucket;

OULinkedListEnumerator<T> currentBucketEnum = this->hashTable->table[nextBucket]->enumerator();

chainEnumerator = &currentBucketEnum;

if (chainEnumerator->hasNext() == true) {

//T data = chainEnumerator->next();

//nextData = chainEnumerator->next();

//nextStatus = true;

// nextData = data;

// \*nextData = data;

T nextData;

nextData = chainEnumerator->next();

// check if next data is in different bucket

if (hasNext()) {

if (chainEnumerator->hasNext() == false) {

// check other bucket(s)

unsigned long nextBucket = 0;

nextBucket = bucket + 1;

OULinkedListEnumerator<T>\* nextChainEnum = NULL;

bool hasNextStatus = false;

while (nextBucket < hashTable->getBaseCapacity() && hasNextStatus == false) {

OULinkedListEnumerator<T> currentBucketEnum = this->hashTable->table[nextBucket]->enumerator();

nextChainEnum = &currentBucketEnum;

//if (\*hashTable->table[bucket]->enumerator()->hasNext() == true) {

if (nextChainEnum->hasNext() == true) {

bucket = nextBucket;

hasNextStatus = true;

}

else {

// look into the next bucket

nextBucket++;

}

}

}

}

//return nextData;

/\*

if (nextBucket < hashTable->getBaseCapacity()) {

do {

nextBucket++;

OULinkedListEnumerator<T> currentBucketEnum = this->hashTable->table[nextBucket]->enumerator();

chainEnumerator = &currentBucketEnum;

if (chainEnumerator->hasNext() == true) {

bucket = nextBucket;

//T data = chainEnumerator->next();

nextData = chainEnumerator->next();

nextStatus = true;

// next item is found at the next bucket

//nextData = data;

// \*nextData = data;

}

} while (nextBucket < hashTable->getBaseCapacity());

//while (nextBucket < hashTable->getBaseCapacity() && nextStatus == false) {

while (nextBucket < hashTable->getBaseCapacity()) {

OULinkedListEnumerator<T> currentBucketEnum = this->hashTable->table[nextBucket]->enumerator();

chainEnumerator = &currentBucketEnum;

if (chainEnumerator->hasNext() == true) {

bucket = nextBucket;

//T data = chainEnumerator->next();

// nextData = chainEnumerator->next();

//nextStatus = true;

// next item is found at the next bucket

//nextData = data;

// \*nextData = data;

nextData = chainEnumerator->next();

return nextData;

}

else {

nextBucket++;

}

}

}

\*/

}

}

}

//else if (chainEnumerator->hasNext() == false) {

else {

// no next item is avaiable

throw new ExceptionHashTableAccess();

}

return nextData;

/\* if (nextStatus == true) {

return nextData;

}

else {

// no next item is avaiable

throw new ExceptionHashTableAccess();

}

\*/

//return nextData;

/\*if (nextData == NULL) {

// no next item is avaiable

throw new ExceptionHashTableAccess();

}

else {

return \*nextData;

}\*/

}

// throws ExceptionHashTableAccess if no next item is available

template <typename T>

T HashTableEnumerator<T>::peek() const {

bool peekStatus = false;

T peekData;

//T\* peekData = NULL;

if (chainEnumerator->hasNext() == true) {

//T data = chainEnumerator->peek();

peekData = chainEnumerator->peek();

peekStatus = true;

//peekData = &data;

}

else if (chainEnumerator->hasNext() == false) {

// check other bucket(s)

unsigned long nextBucket = bucket + 1;

if (nextBucket < hashTable->getBaseCapacity()) {

/\*do {

nextBucket++;

OULinkedListEnumerator<T> currentBucketEnum = this->hashTable->table[nextBucket]->enumerator();

nextChainEnum = &currentBucketEnum;

if (nextChainEnum->hasNext() == true) {

//T data = chainEnumerator->peek();

peekData = chainEnumerator->peek();

peekStatus = true;

// next item is found at the next bucket

//peekData = &data;

}

} while (nextBucket < hashTable->getBaseCapacity());

\*/

OULinkedListEnumerator<T>\* nextChainEnum = NULL;

while (nextBucket < hashTable->getBaseCapacity() && peekStatus == false) {

OULinkedListEnumerator<T> currentBucketEnum = this->hashTable->table[nextBucket]->enumerator();

nextChainEnum = &currentBucketEnum;

if (nextChainEnum->hasNext() == true) {

//T data = chainEnumerator->peek();

peekData = chainEnumerator->peek();

peekStatus = true;

// next item is found at the next bucket

//peekData = &data;

}

else {

nextBucket++;

}

}

}

}

if (peekStatus == true) {

return peekData;

}

else {

// no next item is avaiable

throw new ExceptionHashTableAccess();

}

/\*if (peekData == NULL) {

// no next item is avaiable

throw new ExceptionHashTableAccess();

}

else {

return \*peekData;

}\*/

}

#endif // !HASH\_TABLE\_ENUMERATOR