**Actividad de Aprendizaje 07. Métodos de ordenamiento recursivos**

**Problema:**

Haga un programa que genere valores enteros aleatorios entre 0 y 1'000,000 con los cuales se rellene un arreglo de 100,000 elementos y luego los ordene tomando en cuenta el tiempo requerido para ello e informe una vez terminado el proceso de ordenación. El programa informará el tiempo necesario para ordenar el arreglo con los métodos de ordenamiento:

* + Burbuja (mejorada),
  + Shell,
  + Inserción,
  + Selección,
  + Mezcla, y
  + QuickSort

Todos los tiempos  se imprmirán en una sola pantalla para fines comparativos.  
  
**Requerimientos:**

a) El estilo de programación debe ser Orientado a Objetos  
b) El conjunto de elementos debe ser idéntico para cada caso de ordenamiento  
c) Todos los métodos de ordenamiento deben ser métodos de la misma clase

Christopher Ceballos Jimenez

219750442

Estructuras de Datos I

El entregable fue relativamente sencillo, lo único que me confundio un poco fue la función para calcular el tiempo ya que hay varias formas y cada forma tiene su manera de implementarla. Tuve un problema con el método de ordenamiento QuickSort ya que solo acepta 30600 elementos en una lista, mas de 30600 elemento crashea debido a la recursividad. Pero después se pudo solucionar corrigiendo el código para quicksort

#include <iostream>

#include <windows.h>

#include <random>

#include <chrono>

#include <functional>

#include "list.h"

**using namespace std**;

**double** timeSortingBubble(List<**int**>&);

**double** timeSortingSelect(List<**int**>&);

**double** timeSortingInsert(List<**int**>&);

**double** timeSortingShell(List<**int**>&);

**double** timeSortingMerge(List<**int**>&);

**double** timeSortingQuickSort(List<**int**>&);

**double** performancecounter\_diff(LARGE\_INTEGER \*a, LARGE\_INTEGER \*b){

LARGE\_INTEGER freq;

QueryPerformanceFrequency(&freq);

**return** (**double**)(a->QuadPart - b->QuadPart) / (**double**)freq.QuadPart;

}

**int** main()

{

List<**int**> myList;

**int** randNum;

**default\_random\_engine** generator(chrono::system\_clock::now().time\_since\_epoch().**count**());

**uniform\_int\_distribution**<**int**> distribution(0,1000000);

**auto** dice = **bind**(distribution, generator);

**while**(!myList.isFull()){

randNum = dice();

myList.insertData(myList.getLastPos(), randNum);

}

**cout** <<"Ordenando por metodo Bubble..." << **endl**;

**cout** << "Segundos empleados: " << timeSortingBubble(myList) << **endl** << **endl**;

**cout** <<"Ordenando por metodo Insert..." << **endl**;

**cout** << "Segundos empleados: " << timeSortingInsert(myList) << **endl** << **endl**;

**cout** <<"Ordenando por metodo Select..." << **endl**;

**cout** << "Segundos empleados: " << timeSortingSelect(myList) << **endl** << **endl**;

**cout** <<"Ordenando por metodo Shell..." << **endl**;

**cout** << "Segundos empleados: " << timeSortingShell(myList) << **endl** << **endl**;

**cout** <<"Ordenando por metodo Merge..." << **endl**;

**cout** << "Segundos empleados: " << timeSortingMerge(myList) << **endl** << **endl**;

**cout** <<"Ordenando por metodo QuickSort..." << **endl**;

**cout** << "Segundos empleados: " << timeSortingQuickSort(myList) << **endl** << **endl**;

}

**double** timeSortingBubble(List<**int**>& myList){

LARGE\_INTEGER t\_ini, t\_fin;

**double** secs;

QueryPerformanceCounter(&t\_ini);

myList.sortDataBubble();

QueryPerformanceCounter(&t\_fin);

**return** secs = performancecounter\_diff(&t\_fin, &t\_ini);

}

**double** timeSortingInsert(List<**int**>& myList){

LARGE\_INTEGER t\_ini, t\_fin;

**double** secs;

QueryPerformanceCounter(&t\_ini);

myList.sortDataInsert();

QueryPerformanceCounter(&t\_fin);

**return** secs = performancecounter\_diff(&t\_fin, &t\_ini);

}

**double** timeSortingSelect(List<**int**>& myList){

LARGE\_INTEGER t\_ini, t\_fin;

**double** secs;

QueryPerformanceCounter(&t\_ini);

myList.sortDataSelect();

QueryPerformanceCounter(&t\_fin);

**return** secs = performancecounter\_diff(&t\_fin, &t\_ini);

}

**double** timeSortingShell(List<**int**>& myList){

LARGE\_INTEGER t\_ini, t\_fin;

**double** secs;

QueryPerformanceCounter(&t\_ini);

myList.sortDataShell();

QueryPerformanceCounter(&t\_fin);

**return** secs = performancecounter\_diff(&t\_fin, &t\_ini);

}

**double** timeSortingMerge(List<**int**>& myList){

LARGE\_INTEGER t\_ini, t\_fin;

**double** secs;

QueryPerformanceCounter(&t\_ini);

myList.sortDataMerge();

QueryPerformanceCounter(&t\_fin);

**return** secs = performancecounter\_diff(&t\_fin, &t\_ini);

}

**double** timeSortingQuickSort(List<**int**>& myList){

LARGE\_INTEGER t\_ini, t\_fin;

**double** secs;

QueryPerformanceCounter(&t\_ini);

myList.sortDataQuickSort();

QueryPerformanceCounter(&t\_fin);

**return** secs = performancecounter\_diff(&t\_fin, &t\_ini);

}

#ifndef LIST\_H\_INCLUDED

#define LIST\_H\_INCLUDED

#include <iostream>

#include <string>

#include <exception>

#include <fstream>

**template** <**class** T, **int** ARRAYSIZE = 30600>

**class** List {

**private**:

T data[ARRAYSIZE];

**int** last;

**bool** isValidPos(**const int**&);

**void** copyAll(**const** List&);

**void** swapData(T&, T&);

**void** sortDataMerge(**const int**&, **const int**&);

**void** sortDataQuickSort(**const int**&, **const int**&);

**public**:

**class** ListException : **public std**::**exception**{

**private**:

**std**::**string** msg;

**public**:

**explicit** ListException(**const char**\* message) : msg(message){}

**explicit** ListException(**const std**::**string**& message) : msg(message) {}

**virtual** ~ListException() **throw**() {}

**virtual const char**\* what() **const throw**() {

**return** msg.c\_str();

}

};

List();

List(**const** List&);

**bool** isEmpty();

**bool** isFull();

**void** insertData(**const int**&,**const** T&);//primera la pos y luego el elemento

**void** deleteData(**const int**&);

**int** getFirstPos() **const**;

**int** getLastPos() **const**;

**int** getPrevPos(**const int**&) **const**;

**int** getNextPos(**const int**&) **const**;

**int** findDataLineal(**const** T&);

**int** findDataBinaria(**const** T&);

**void** sortDataBubble();

**void** sortDataInsert();

**void** sortDataSelect();

**void** sortDataShell();

**void** sortDataMerge();

**void** sortDataQuickSort();

T retrieve(**const int**&);

**int** toString() **const**;

**void** deleteAll();

**void** writeToDisk(**const std**::**string**&);

**void** readFromDisk(**const std**::**string**&);

List& **operator** = (**const** List&);

};

//implementacion

**template** <**class** T, **int** ARRAYSIZE>

List<T,ARRAYSIZE>::List() : last(-1) {}

**template** <**class** T, **int** ARRAYSIZE>

List<T, ARRAYSIZE>::List(**const** List& s) : last(s.last) { }

**template** <**class** T, **int** ARRAYSIZE>

**void** List <T,ARRAYSIZE> :: copyAll(**const** List& s) {

**int** i(0);

**while** (i <= s.last) {

**this** -> data[i] = s.data[i];

i++;

}

**this**-> last = s.last;

}

**template** <**class** T,**int** ARRAYSIZE>

**void** List<T, ARRAYSIZE>::swapData(T& a, T& b){ // swapData

T aux(a);

a = b;

b = aux;

}

**template** <**class** T,**int** ARRAYSIZE>

**bool** List<T, ARRAYSIZE>::isValidPos(**const int**& p){

**return** p >= 0 and p<= last;

}

**template** <**class** T,**int** ARRAYSIZE>

**bool** List<T, ARRAYSIZE>::isEmpty ( ) {

**return** last == -1;

}

**template** <**class** T,**int** ARRAYSIZE>

**bool** List<T, ARRAYSIZE>::isFull ( ) {

**return** last == ARRAYSIZE - 1;

}

**template** <**class** T,**int** ARRAYSIZE>

**void** List<T, ARRAYSIZE>::insertData (**const int**& p, **const** T& e) {

**if**(isFull()){

**throw** ListException("Desbordamiento de datos, insertData");

}

**if**(p != -1 and !isValidPos(p)){

**throw** ListException("Posicion invalida, insertData");

}

**int** i(last);

**while**(i > p){

data[i+1] = data[i];

i--;

}

data[p+1] = e;

last++;

}

**template** <**class** T,**int** ARRAYSIZE>

**void** List<T, ARRAYSIZE>::deleteData(**const int**& p){

**if**(!isValidPos(p)){

**throw** ListException("Posicion invalida, deleteData");

}

**int** i(p);

**while**(i < last){

data[i]= data[i+1];

i++;

}

last--;

}

**template** <**class** T,**int** ARRAYSIZE>

**int** List<T, ARRAYSIZE>::getFirstPos() **const**{

**if** (isEmpty()){

**return** -1;

}

**return** 0;

}

**template** <**class** T,**int** ARRAYSIZE>

**int** List<T, ARRAYSIZE>::getLastPos ( ) **const**{

**return** last;

}

**template** <**class** T,**int** ARRAYSIZE>

**int** List<T, ARRAYSIZE>::getPrevPos (**const int**& p) **const**{

**if**(p== 0 or isValidPos(p)){

**return** -1;

}

**return** p-1;

}

**template** <**class** T,**int** ARRAYSIZE>

**int** List<T, ARRAYSIZE>::getNextPos(**const int**& p) **const**{

**if**(p == 0 or !isValidPos(p)){

//problema

}

**return** 0+1;

}

**template** <**class** T,**int** ARRAYSIZE>

**int** List<T, ARRAYSIZE>::findDataLineal(**const** T& e){ **///findDataLineal**

**int** i(0);

**while**(i <= last){

**if**( data[i] == e){

**return** i;

}

i++;

}

**return** -1;

}

**template** <**class** T,**int** ARRAYSIZE>

**int** List<T, ARRAYSIZE>::findDataBinaria(**const** T& e){ **///findDataBinaria**

**int** i(0), j(last), m;

**while**(i <= j){

m = (i+j)/2;

**if** (data[m] == e){

**return** m;

}

**if** (e < data[m]){

j = m-1;

}**else**{

i = m+1;

}

}

**return** -1;

}

**template** <**class** T,**int** ARRAYSIZE>

**void** List<T, ARRAYSIZE>::sortDataBubble ( ) { **///Ordenarmiento tipo bubble**

**int** i(last) , j;

**bool** flag;

**do**{

flag = **false**;

j = 0;

**while**(j < i){

**if**(data[j] > data[j+1]){

swapData(data[j], data[j + 1]);

flag = **true**;

}

j++;

}

i--;

}**while**(flag);

}

**template** <**class** T,**int** ARRAYSIZE>

**void** List<T, ARRAYSIZE>::sortDataInsert ( ) { **///Ordenamiento tipo Insercion**

**int** i(1), j;

T aux;

**while**(i <= last){

aux = data[i];

j = i;

**while**(j > 0 and aux < data[ j - 1]){

data[j] = data[j - 1];

j--;

}

**if**( i != j){

data[j] = aux;

}

i++;

}

}

**template** <**class** T,**int** ARRAYSIZE>

**void** List<T, ARRAYSIZE>::sortDataSelect ( ) { **///Ordenamiento tipo Seleccion**

**int** i(0), j, m;

**while**(i < last){

m = i;

j= i + 1;

**while**(j <= last){

**if**(data[j] < data[m]){

m = j;

}

j++;

}

**if**(i != m){

swapData(data[i], data[m]);

}

i++;

}

}

**template** <**class** T,**int** ARRAYSIZE>

**void** List<T, ARRAYSIZE>::sortDataShell ( ) { **///Ordenamiento tipo Shell**

**float** factor(1.0/2.0);

**int** dif((last + 1)\* factor), i,j;

**while**(dif > 0){

i = dif;

**while**(i <= last){

j = i;

**while**(j >= dif and data[j - dif] > data[j]){

swapData(data[j - dif], data[j]);

j -= dif;

}

i++;

}

dif\*= factor;

}

}

**template** <**class** T,**int** ARRAYSIZE> //polimorfismo,

**void** List<T, ARRAYSIZE>::sortDataMerge(){

sortDataMerge(0, last);

}

**template** <**class** T,**int** ARRAYSIZE> **///ordenamiento tipo merge**

**void** List<T, ARRAYSIZE>::sortDataMerge(**const int**& leftEdge, **const int**& rightEdge){

**if**(leftEdge >= rightEdge){ // criterio de paro

**return**;

}

//divide y venceras

**int** m((leftEdge + rightEdge) / 2);

sortDataMerge(leftEdge, m);

sortDataMerge(m+1, rightEdge);

//copia

**static** T temp[ARRAYSIZE];

**for**(**int** z(leftEdge); z <= rightEdge; temp[z] = data[z], z++);

//intercalacion

**int** i(leftEdge), j(m + 1), x(leftEdge);

**while**(i <= m and j <= rightEdge){

**while**(i <= m and temp[i] <= temp[j]){

data[x++] = temp[i++];

}

**if**(i <= m){

**while**(j <= rightEdge and temp[j] <= temp[i]){

data[x++] = temp[j++];

}

}

}

**while**(i <= m){

data[x++] = temp[i++];

}

**while**(j <= rightEdge){

data[x++] = temp[j++];

}

}

**template** <**class** T,**int** ARRAYSIZE>

**void** List<T, ARRAYSIZE>::sortDataQuickSort(){

sortDataQuickSort(0,last);

}

**template** <**class** T,**int** ARRAYSIZE>

**void** List<T, ARRAYSIZE>::sortDataQuickSort(**const int**& leftEdge, **const int**& rightEdge){

**if**(leftEdge >= rightEdge){

**return**;

}

swapData(data[(leftEdge + rightEdge)/2], data[rightEdge]);

//separacion

**int** i(leftEdge), j(rightEdge);

**while**(i < j){

**while**(i < j and data[i] <= data[rightEdge]){

i++;

}

**while**(i < j and data[j] >= data[rightEdge]){

j--;

}

**if**(i != j){

swapData(data[i], data[j]);

}

}

**if**(i != rightEdge){

swapData(data[i], data[rightEdge]);

}

//divide y venceras

sortDataQuickSort(leftEdge, i-1);

sortDataQuickSort(i+1, rightEdge);

}

**template** <**class** T,**int** ARRAYSIZE>

T List<T, ARRAYSIZE>::retrieve(**const int**& p) { **///recuperar**

**if**(!isValidPos(p)){

**throw** ListException("Posicion invalida, retrieve");

}

**return** data[p];

}

**template** <**class** T,**int** ARRAYSIZE>

**int** List<T, ARRAYSIZE>::toString ( ) **const**{

**int** i(0);

**while**(i <= last){

**std**::**cout** << data[i] << " ";

i++;

}

}

**template** <**class** T,**int** ARRAYSIZE>

**void** List<T, ARRAYSIZE>::deleteAll ( ) {

last = -1;

}

**template** <**class** T,**int** ARRAYSIZE>

**void** List<T, ARRAYSIZE>::writeToDisk(**const std**::**string**& fileName){

**std**::**ofstream** myFile;

myFile.open(fileName, myFile.trunc/\*ios\_base::trunc\*/);

**if**(!myFile.is\_open()){

**std**::**string** message;

message = "No se pudo abrir el archivo ";

message += fileName;

message += " para escritura, writeToDisk.";

**throw** ListException(message);

}

**int** i(0);

**while**(i <= last){

myFile << data[i++];

}

myFile.close();

}

**template** <**class** T,**int** ARRAYSIZE>

**void** List<T, ARRAYSIZE>::readFromDisk(**const std**::**string**& fileName){

**std**::**ifstream** myFile;

T myData;

myFile.open(fileName);

**try**{

**if**(!myFile.is\_open()){

**std**::**string** message;

message = "No se pudo abrir el archivo ";

message += fileName;

message += " para lectura, readFromDisk.";

}

} **catch**(ListException ex) {

myFile.close();

**std**::**string** message;

message +=fileName;

message += " : ";

message += ex.what();

message += ", readFromDisk";

**throw** ListException (message);

}

**int** i(0);

**while**(i <= last){

myFile >> data[i];

i++;

}

deleteAll();

**try**{

**while**(myFile >> myData){ // myFile >> myData returna un falso o verdadero

insertData(getLastPos(), myData);

}

} **catch**(ListException ex) {

myFile.close();

**std**::**string** message("Error durante la lectura del archivo.");

message +=fileName;

message += " : ";

message += ex.what();

message += ", readFromDisk";

**throw** ListException (message);

}

myFile.close();

}

**template** <**class** T,**int** ARRAYSIZE>

List<T, ARRAYSIZE>& List<T, ARRAYSIZE>::**operator** = (**const** List& l) {

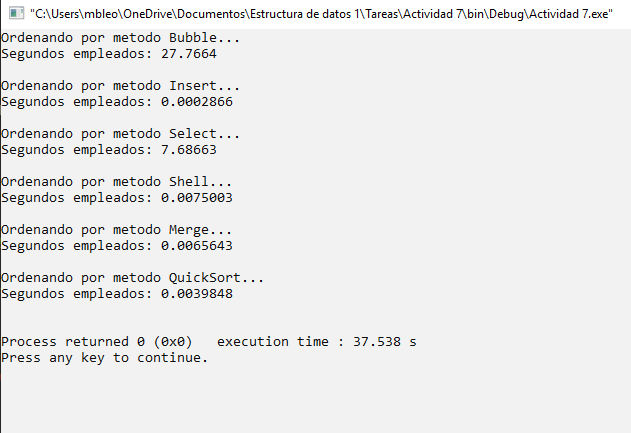
copyAll(l);

**return** \***this**;

}

#endif // LIST\_H\_INCLUDED

Programa con arreglo de 100000 Elementos



Programa con arreglo de 500000 elementos

