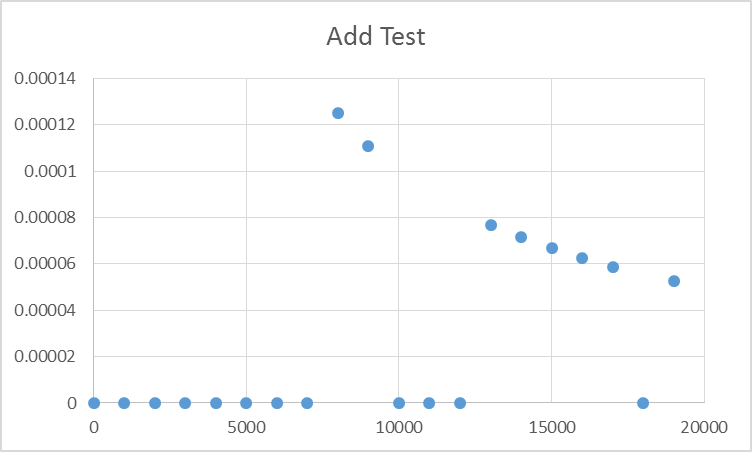
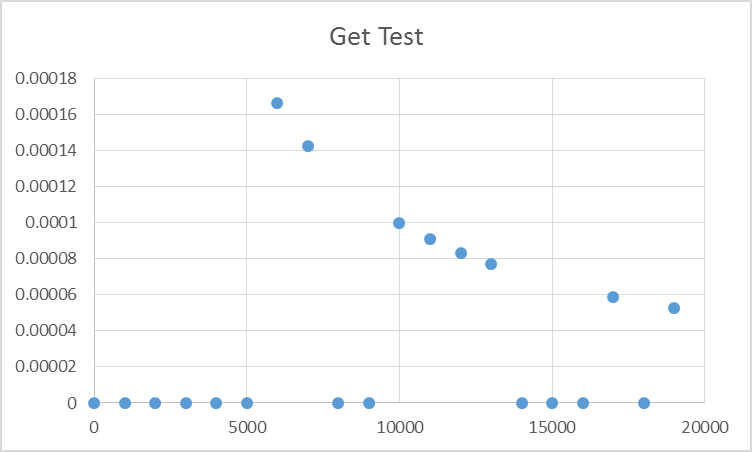
Rocio Salguero

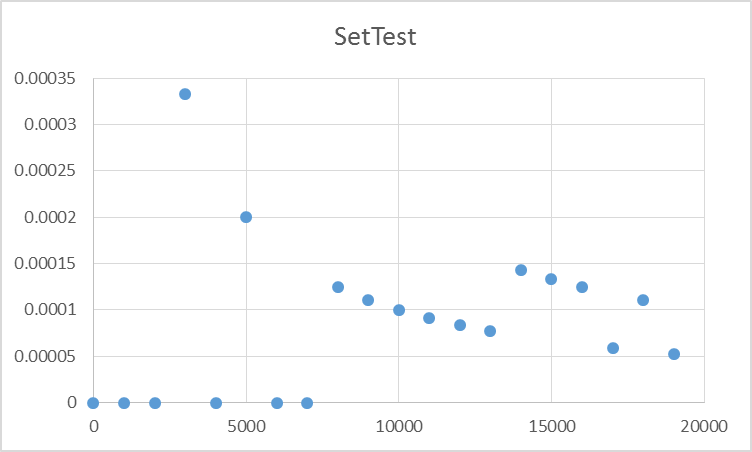
131 Data Structures

Project 1

**Array List Plots:**







**Link-List Plots:**

Questions:

**1. What are the big-O run-times of the Get and Set operations for an array-based list? Do your empirical results agree with this? Why or why not?**

Big-O run time for get and Set should be both constant O(1). Yes the empirical results agree. While the plot shows an inverse log trend, both Get and Set functions show times close to 0. Most of the values are close to 0 therefore the big-O run time in worst case scenario is constant.

**2. What is the big-O run-time of the Remove operation for an array-based list? Do your empirical results agree with this? Why or why not?**

Big-O run time of Remove function for Array-List should be O(n) linear since . The empirical results agree with this since the times follow a linear trend.

**3. What are the big-O run-times of the Get and Set operations for a linked-list? Do your empirical results agree with this? Why or why not?**

Big-O run times for Get and Set function should be both O(n) since . The empirical results agree with this since the times follow a linear O(n) trend.

**4. What is the big-O run-time of the Add operation for an array-based list? Do your empirical results agree with this? Why or why not?**

The Add function for Array-based list should have a big-O run time of O(n) in worst case scenario if resize is required . If resize is not required then the Add function should be O(1). The empirical results agree with O(n) since the times never grew worse than linearly. The empirical times showed an overall constant time though so its assumed the resize wasn’t used much.

**ArrayList.h**

#ifndef ArrayList\_H

#define ArrayList\_H

#include "List.h"

using namespace std;

typedef unsigned int Index;

template <typename T>

class ArrayList : public List<T> {

private:

Index sizeofArray;

Index max;

T\* arr;

public:

ArrayList() {

max = 5;

sizeofArray = 0;

arr = new T[max];

}

//copy constructor

ArrayList(ArrayList& copyArray) {

this->sizeofArray = copyArray.sizeofArray;

this->max = copyArray.max;

this->arr = copyArray.arr;

}

//Overloaded constructor

ArrayList(int size) {

max = size \* 1.5;

sizeofArray = size;

arr = new T[max];

}

~ArrayList() {

Clear();

}

//returns a new array with both elements from both arrays

friend ArrayList operator+(const ArrayList& rhs) {

newMax = this->max + rhs.max;

newArray = new T[newMax];

for (Index i = 0; i < this->sizeofArray; i++)

newArray[i] = this->arr[i];

for (Index i = 0; i < rhs.sizeofArray; i++)

newArray[i+rhs.max] = rhs.arr[i];

return newArray;

}

ArrayList operator=(const ArrayList& rhs) {

this->sizeofArray = rhs.sizeofArray;

this->max = rhs.max;

this->arr = rhs.arr;

for (Index i = 0; i < this->sizeofArray; i++)

this->arr[i] = rhs.arr[i];

}

virtual void Add(const T& ele) {

if (sizeofArray == max)

Resize();

arr[sizeofArray] = ele;

sizeofArray++;

}

void Resize() {

max = max \* 2;

if (max == 0)

max = 10;

T\* temp = new T[max];

for (Index i = 0; i < sizeofArray; i++)

temp[i] = arr[i];

delete [] arr;

arr = temp;

}

virtual void Set(Index i, const T& val) {

if (sizeofArray <= i || i < 0)

throw List<T>::BAD\_INDX;

arr[i] = val;

}

virtual void Remove(Index i) {

if (sizeofArray <= i || i < 0)

throw List<T>::BAD\_INDX;

for (Index j = i + 1; j < sizeofArray; ++j)

arr[j - 1] = arr[j];

sizeofArray--;

}

virtual void Clear() {

delete[] arr;

max = sizeofArray = 0;

}

virtual T& Get(Index i) const {

if (sizeofArray <= i || i < 0)

throw List<T>::BAD\_INDX;

return arr[i];

}

virtual unsigned int Size() const {

return sizeofArray;

}

};

#endif

**LinkedList.h**

/\*

Name: Rocio Salguero

Date: Sept 19 2015

Description: Linklist using a front pointer and a end pointer. The add function adds the elements to the end of the list.

\*/

#ifndef LinkedList\_H

#define LinkedList\_H

#include "List.h"

#include <cmath>

using namespace std;

typedef unsigned int Index;

template <typename T>

class LinkedList : public List<T> {

private:

class Node {

public:

Node(Node\* p, T i, Node\* n) {

prevLink = p;

item = i;

nextLink = n;

}

Node\* prevLink;

T item;

Node\* nextLink;

};

Node\* firstElement;

Node\* lastElement;

Index arraySize; //number of elements in the array

public:

LinkedList() : arraySize(0), firstElement(NULL), lastElement(NULL) {}

//Copy constructor

LinkedList(LinkedList& copy) {

Node\* ptrIndex = this->firstElement;

for (int i = 0; i < this->arraySize; i++) {

copy.Add(ptrIndex.item);

ptrIndex = ptrIndex->nextLink;

}

}

//Deconstructor

~LinkedList() {

Clear();

}

//Returns a new LinkList with both other lists combined

friend LinkedList operator+(const LinkedList& rhs) {

LinkedList<T> lst1;

Node\* ptrIndex = this->firstElement;

for (int i = 0; i < this->arraySize; i++) {

lst1.Add(ptrIndex.item);

ptrIndex = ptrIndex->nextLink;

}

ptrIndex = rhs.firstElement;

for (int i = 0; i < rhs.arraySize; i++) {

lst1.Add(ptrIndex.item);

ptrIndex = ptrIndex->nextLink;

}

return lst1;

}

//Adds to the end of the list

virtual void Add(const T& ele) {

if (arraySize == 0) {

Node \* newNode = new Node(NULL, ele, NULL);

firstElement = newNode;

}

else if (arraySize == 1) {

Node \* newNode = new Node(firstElement, ele, NULL);

firstElement->nextLink = newNode;

lastElement = newNode;

}

else {

Node \* newNode = new Node(lastElement, ele, NULL);

lastElement->nextLink = newNode;

lastElement = newNode;

}

arraySize++;

}

//Traverses the array either from the begining or from the end depending on the index

virtual void Set(Index i, const T& val) {

if (arraySize <= i || i < 0)

throw List<T>::BAD\_INDX;

Index middleIndex = ceil(arraySize / 2);

Node \* ptrIndex;

if (i <= middleIndex) {

ptrIndex = firstElement;

for (Index j = 0; j < i; ++j) {

ptrIndex = ptrIndex->nextLink;

}

}

else {

ptrIndex = lastElement;

for (Index j = 0; j < (arraySize - i - 1); ++j) {

ptrIndex = ptrIndex->prevLink;

}

}

ptrIndex->item = val; //set indexed item as value

}

//traverses array then stops at element before index then reassigns pointers and deletes index Node

virtual void Remove(Index i) {

if (arraySize <= i || i < 0)

throw List<T>::BAD\_INDX;

Index middleIndex = ceil(arraySize / 2);

Node \* ptrIndex;

if (i == 0) { //if first element

ptrIndex = firstElement;

if (arraySize == 1) {

delete ptrIndex;

firstElement = NULL;

}

else {

firstElement = firstElement->nextLink;

firstElement->prevLink = NULL;

delete ptrIndex;

}

}

else if (i == (arraySize-1)) { //if last element

ptrIndex = lastElement;

lastElement = lastElement->prevLink;

lastElement->nextLink = NULL;

delete ptrIndex;

}

else if (i < middleIndex) { //traverses same as Set()

ptrIndex = firstElement;

for (Index j = 0; j < (i-1); ++j) {

ptrIndex = ptrIndex->nextLink;

}

Node \* oldNode = ptrIndex->nextLink;

Node \* nextNode = oldNode->nextLink;

ptrIndex->nextLink = oldNode->nextLink;

nextNode->prevLink = oldNode->prevLink;

delete oldNode;

}

else {

ptrIndex = lastElement;

for (Index j = 0; j < (arraySize - i - 2); ++j) {

ptrIndex = ptrIndex->prevLink;

}

Node \* oldNode = ptrIndex->prevLink;

Node \* nextNode = oldNode->prevLink;

ptrIndex->prevLink = oldNode->prevLink;

nextNode->nextLink = oldNode->nextLink;

delete oldNode;

}

arraySize--;

}

virtual void Clear() {

while (Size() > 0) {

Remove(0);

}

}

//traverses same as Set()

virtual T& Get(Index i) const {

if (arraySize <= i || i < 0)

throw List<T>::BAD\_INDX;

Index middleIndex = ceil(arraySize / 2);

Node \* ptrIndex;

if (i <= middleIndex) {

ptrIndex = firstElement;

for (Index j = 0; j < i; ++j) {

ptrIndex = ptrIndex->nextLink;

}

}

else {

ptrIndex = lastElement;

for (Index j = 0; j < (arraySize-i-1); ++j) {

ptrIndex = ptrIndex->prevLink;

}

}

return ptrIndex->item;

}

virtual unsigned int Size() const {

return arraySize;

}

};

#endif