ANÁLISIS Y DISEÑOS DE ALGORITMOS I

PROYECTO DE CURSADA 2022

Agustín Campo

Sheila Artaza

Gmails: [artazasheii@gmail.com](mailto:artazasheii@gmail.com)

[agustincampo\_@hotmail.com](mailto:agustincampo_@hotmail.com)

Grupo: 58

Ayudante: Federico.

Introducción:

Este proyecto tiene como objetivo especificar las estructuras básicas como una fila, una lista y un árbol en tipos de datos abstractos utilizando el lenguaje de C++ y para su previa implementación en dicho lenguaje se utilizó la especificación nereus, todo esto con el objetivo de su futura implementación en un programa con un fin específico.

En este informe se desarrollan las especificaciones en Nereus, se calcula la complejidad de los métodos utilizados y se agrega una breve descripción.

Especificación Nereus:

* TDA Lista

**CLASS** Lista [Elemento]

**IMPORTS** Nat

**BASIC CONSTRUCTORS** inicLista, agregarLista

**EFFECTIVE**

**TYPE** Lista

**OPERATIONS**

inicLista: -> Lista;

longLista: Lista -> Natural;

agregarlista: Lista \* natural \* natural -> Lista;

cantelem: Lista -> natural;

vaciaLista: Lista -> Boolean;

pertenece: Lista \* natural -> Boolean;

eliminarElemLista: Lista \* natural -> Lista;

**AXIOMS**

**END-CLASS**

* TDA Fila

**CLASS** Fila [Elemento]

**BASIC CONSTRUCTORS** inicFila, agregarFila **EFFECTIVE  
TYPE** Fila **OPERATIONS**

inicFila: -> Fila;

agregarFila: Fila \* Elemento -> Fila;

vaciaFila: Fila -> Boolean;

recuperarFila: Fila(f) -> Elemento

**pre**: not vaciaFila(f);

eliminarPrimero: Fila \* Elemento -> Fila;

**pre**: not vaciaFila(f);

cantElemFila: Fila -> Integer;

**pre**: not vaciaFila;

**AXIOMS**

**END-CLASS**

* TDA Arbol

**CLASS** Arbol [Elemento]

**IMPORTS** Lista;

**BASIC CONSTRUCTORS** inicArbol, agregarArbol

**EFFECTIVE**

**TYPE** Arbol

**OPERATIONS**

inicArbol: -> Arbol;

agregarArbol: Arbol \* Elemento -> Arbol;

vacioArbol: Arbol -> Boolean;

raiz: Arbol(t) -> Elemento

**pre**: not vacioArbol(t);

perteneceElemArbol: Arbol(t) \* Elemento -> Boolean;

cantElemArbol: Arbol(t) -> natural;

profundArbol: Arbol -> Integer;

mostrarElemFrontera: Arbol(t) -> Lista();

**pre**: not vacioArbol(t);

mostrarElemArbol: Arbol(t) -> Lista();

**pre**: not vacioArbol(t);

**AXIOMS**

**END-CLASS**

Para cada uno de los TDA ocupados utilizamos las siguientes estructuras:

TDA Lista: utilizamos una secuencia de nodos del mismo tipo. Los elementos se encuentran ordenados linealmente por punteros.

TDA Fila: está fue implementada a partir de la lista con

el esquema FIFO (first-in first-out).

TDA Árbol: utilizamos la estructura de un árbol binario, siendo este un conjunto de nodos, donde nos encontramos con un nodo raíz y todos los nodos menos este están conectados por uno llamado padre. A su vez cada nodo posee un puntero a su izquierda y otro a su derecha.

Complejidad temporal de funciones implementadas junto a una breve descripción de cada una:

TDA Lista:

| Función implementada | Complejidad (Big-Oh) | Descripción |
| --- | --- | --- |
| agregaralprincipio | O(1) | inserta un elemento al inicio de la lista |
| agregaralfinal | O(1) | inserta un elemento al final de la lista |
| agregararbitrario | O(n) n = cantelem | inserta un elemento en una posición dada |
| cantelem | O(n) n = cantelem | devuelve la cantidad de nodos de la lista |
| pertenece | O(n) n = cantelem | verifica si un elemento pertenece a la lista |
| listavacia | O(1) | verifica que la lista está vacía |
| eliminarelem | O(n) n = cantelem | elimina un elemento de la lista y la vincula nuevamente manteniendo el orden |
| iteracion | O(n) n = cantelem | itera por fuera de la lista |
| imprimirlista | O(n) n = cantelem | imprime los elementos de la lista |

TDA Fila:

| Función implementada | Complejidad (Big-Oh) | Descripción |
| --- | --- | --- |
| agregar | O(1) | agrega un elemento a la fila |
| sacarelemento | O(n) | extrae un elemento de la fila |
| sacarproxelemento | O(1) | extrae el próximo elemento de la fila |
| valorsigaretirar | O(1) | devuelve cuál es el valor que se va a retirar |
| cantidaddeelementos | O(n) n = cantelem | devuelve la cantidad de elementos de la fila |
| filavacia | O(1) n = cantelem | verifica si la fila está vacía o no |

TDA Arbol:

| Función Implementada | Complejidad (Big-Oh) | Descripción |
| --- | --- | --- |
| agregarelemento | O(n) n = cantnodos | inserta un nuevo elemento al árbol |
| perteneceelem | O(n) n = cantnodos | verifica si existe un elemento solicitado |
| cantelementos | O(n) n = cantnodos | retorna la cantidad de elementos del árbol |
| arbolvacio | O(1) | verifica si el árbol está vacío |
| profundidadarbol | O(n) n = cantnodos | retorna la profundidad del árbol |
| listarfrontera | O(n) n = cantnodos | recorre el árbol hasta sus “hojas” y las enlista |
| listararbol | O(n) n = cantnodos | enlista los elementos del árbol |

Código fuente:

Headers:

* lista.h

#define LISTA\_H

#define LISTA\_H

template <typename L>

class lista

{

public:

lista(); //constructor de la clase

~lista(); //destructor

void agregaralprincipio(const L & elem);

void agregaralfinal(const L & elem);

void agregararbitrario(const L & elem, int);

int cantelem() const;

bool pertenece(const L & elem) const;

bool listavacia() const;

void eliminarelem(const L & elem);

const & iteracion(int) const;

void imprimirlista() ;

private: //mis argumentos

struct nodo {

L elem;

nodo \* sig;

};

nodo\*primero;

nodo \* ultacceso;

};

* Fila.h

#ifndef FILA\_H

#define FILA\_H

#include <iostream>

using namespace std;

template<typename F>

class Fila

{

public:

Fila(); //constructor de la clase

~Fila(); //destructor

void agregar(const F & elem);

void sacarelemento();

int sacarproxelemento();

const & valorsigaretirar() const;

int cantidadelementos() const;

bool filavacia() const;

void imprimirfila() const;

private:

struct nodo {

F elem;

nodo \* sig;

};

nodo \* ultimo;

nodo \* primero; //no tiene siguiente, ultimo de la lista pero primero de la fila

nodo \* aux;

nodo \* aux1;

nodo \* aux2;

};

* Arbol.h

#ifndef ARBOL\_H

#define ARBOL\_H

#include "lista.h"

template<typename A>

class Arbol

{

public:

Arbol(); //constructor de la clase

~Arbol(); //destructor

void agregarElemento(const A & elem);

lista<A>listadefrontera;

bool perteneceelem(const A & elem);

int cantelementos();

bool arbolvacio();

int profundidadarbol();

void listarfrontera() ;

void listararbol() ;

private:

struct nodo {

A elem;

nodo \* menor;

nodo \* mayor;

};

nodo \* raiz;

void insertarenarbol(const A & elem, nodo\*raiz);

bool estaenarbol(const A & elem, nodo\*raiz);

int cantidad(nodo\*raiz);

int profundidad(nodo\*raiz);

void frontera(nodo\*raiz);

void enorden(nodo\*raiz);

};

Sources

* lista.cpp

#include "lista.h"

#include <iostream>

using namespace std;

template<typename L>

lista<L>::lista()

{

this->primero=NULL;

}

template<typename L>

void lista<L>:: agregaralprincipio(const L & elem) {

nodo \* nuevo = new nodo;

nuevo->elem = elem;

nuevo->sig = primero;

primero = nuevo;

}

template<typename L>

void lista<L>:: agregaralfinal(const L & elem) {

nodo \*aux = primero;

while (aux->sig!= NULL) {

aux = aux->sig;

}

nodo \* nuevo = new nodo;

nuevo->elem = elem;

nuevo->sig = NULL;

aux->sig = nuevo;

}

template<typename L>

void lista<L>:: agregararbitrario(const L & elem, int pos){

int contador = 0;

nodo \*aux = primero;

nodo \* pant;

while ((aux!= NULL)&&(contador < pos)) {

pant= aux;

aux = aux->sig;

contador++;

}

nodo \* nuevo = new nodo;

nuevo->elem = elem;

nuevo->sig = aux;

pant->sig = nuevo;

}

template<typename L>

int lista<L>::cantelem()const{

nodo \*aux = primero;

int cantidad = 0;

while(aux != NULL){

cantidad++;

aux = aux->sig;

}

cout << "la cantidad de elementos es: " << cantidad << endl;

}

template<typename L>

bool lista<L>::pertenece(const L & elem)const {

bool encontrado = false;

nodo \*aux=primero;

while ((aux != NULL) && (encontrado == false)){

if(aux->elem != elem){

aux = aux->sig;

}

else{

encontrado = true;

}

}

return encontrado;

}

template<typename L>

bool lista<L>::listavacia() const {

bool esvacia;

if(nodo\*primero = NULL){

esvacia=true;

}

else{

esvacia=false;

}

return esvacia;

}

template<typename L>

void lista<L>:: eliminarelem(const L & elem){

bool encontrado = false;

nodo \*aux = primero;

nodo \*pant= NULL;

while((aux != NULL) && (encontrado == false )){

if(aux->elem != elem){

pant=aux;

aux=aux->sig;

}

else{

encontrado=true;

}

}

if ((encontrado == true) && (pant==NULL)){ //caso1: elemento al principio de la lista.

primero=primero->sig;

delete aux;

}

if (pant != NULL){ //caso2: elemento en cualquier posicion de la lista

pant->sig = aux->sig;

delete aux;

}

}

template<typename L> //revisar

const & lista<L>::iteracion(int) const{

//assert (!listavacia());

int valor=ultacceso->elem;

if (ultacceso->sig == NULL) {

nodo\*ultacceso=primero;

}

else {

nodo\*ultacceso = ultacceso->sig;

}

return valor;

}

template<typename L> // solo lo implemente para probar los modulos

void lista<L>:: imprimirlista() {

nodo\*aux = primero;

if (primero!=NULL){

nodo\*aux=primero;

while (aux != NULL) {

cout<< aux-> elem << endl;

aux=aux->sig;

}

}

}

template<typename L>

lista<L>::~lista()

{

//dtor

}

* Fila.cpp

#include "Fila.h"

#include <iostream>

using namespace std;

template<typename F>

Fila<F>::Fila()

{

this->primero=NULL;

}

template<typename F>

Fila<F>::~Fila () {

}

template<typename F>

void Fila<F>:: agregar(const F & elem) {

if (primero == NULL) {

nodo \* nuevo = new nodo;

nuevo->elem = elem;

nuevo->sig = primero;

primero = ultimo = nuevo;

}

else {

nodo \* nuevo = new nodo;

nuevo->elem = elem;

nuevo->sig = ultimo;

ultimo = nuevo;

}

}

template<typename F>

void Fila<F>:: sacarelemento () {

aux = new nodo;

aux = ultimo;

if (aux != NULL) {

while(aux->sig->sig != NULL){

aux = aux->sig;

}

primero = aux;

aux1 = new nodo;

aux1 = aux-> sig;

primero->sig = NULL;

delete(aux1);

}

}

template<typename F>

int Fila<F>:: sacarproxelemento() {

return primero->elem;

}

template<typename F>

bool Fila<F>::filavacia() const{

bool esvacia;

if(ultimo == NULL){

esvacia=true;

}

else{

esvacia=false;

}

return esvacia;

}

template<typename F>

const & Fila<F>:: valorsigaretirar () const {

return (primero->elem);

}

template<typename F>

int Fila<F>::cantidadelementos() const {

nodo \* aux = ultimo;

int cantidad = 0;

while(aux != NULL){

cantidad++;

aux = aux->sig;

}

return cantidad;

}

template<typename F>

void Fila<F>:: imprimirfila() const{

nodo\*aux = ultimo;

while (aux != NULL) {

cout<< aux-> elem << endl;

aux=aux->sig;

}

}

* Arbol.cpp

#include "Arbol.h"

#include "lista.h"

#include <iostream>

using namespace std;

template<typename A>

Arbol<A>::Arbol()

{

this->raiz=NULL;

}

template<typename A>

Arbol<A>:: ~Arbol()

{

}

template<typename A>

void Arbol<A>:: insertarenarbol(const A & elem, nodo\*raiz) {

if (raiz == NULL) {

nodo \* nuevo = new nodo;

nuevo-> elem = elem;

nuevo-> menor = NULL;

nuevo-> mayor = NULL;

raiz=nuevo;

}

else {

if (raiz->elem < elem) {

insertarenarbol(elem,raiz->mayor);

}

else {

insertarenarbol(elem, raiz->menor);

}

}

}

template<typename A>

void Arbol<A>:: agregarElemento(const A & elem) {

if (raiz == NULL) {

nodo \* nuevo = new nodo;

nuevo-> elem = elem;

nuevo-> menor = NULL;

nuevo-> mayor = NULL;

raiz=nuevo;

}

else {

insertarenarbol(elem, raiz);

}

}

template<typename A>

bool Arbol<A>:: estaenarbol(const A & elem, nodo\*raiz) {

if (raiz != NULL) {

if (raiz-> elem == elem) {

return true;

}

if (raiz->elem < elem) {

estaenarbol(elem,raiz->mayor);

}

else {

estaenarbol(elem,raiz->menor);

}

}

else{

return false;

}

}

template<typename A>

bool Arbol<A>:: perteneceelem(const A & elem){

return estaenarbol(elem,raiz);

}

template<typename A>

int Arbol<A>:: cantidad(nodo\*raiz) {

if (raiz != NULL) {

return (cantidad(raiz->menor) + cantidad(raiz->mayor) + 1) ;

}

else {

return 1;

}

}

template<typename A>

int Arbol<A>:: cantelementos() {

if (raiz != NULL) {

return cantidad(raiz) + 1;

}

}

template<typename A>

bool Arbol<A>:: arbolvacio(){

if (raiz == NULL) {

return true;

}

else {

return false;

}

}

template<typename A>

int Arbol<A>:: profundidad (nodo\*raiz) {

int ramaMayor, ramaMenor, contador;

if (raiz != NULL) {

ramaMenor = profundidad(raiz->menor);

ramaMayor = profundidad(raiz->mayor);

if (ramaMenor < ramaMayor) {

contador = 1 + ramaMayor;

}

else {

contador = 1 + ramaMenor;

}

}

else {

contador = 0;

}

return contador;

}

template<typename A>

int Arbol<A>:: profundidadarbol(){

if (raiz != NULL) {

return profundidad(raiz);

}

else {

return 0;

}

}

template<typename A>

void Arbol<A>:: frontera (nodo\*raiz) {

if (raiz != NULL) {

frontera(raiz->menor);

frontera(raiz->mayor);

if ((raiz->menor==NULL) && (raiz->mayor==NULL)) {

listadefrontera.agregaralfinal(raiz->elem);

}

}

cout << "Lista de frontera es: " << endl;

listadefrontera.imprimirlista();

}

template<typename A>

void Arbol<A>:: listarfrontera () {

if (raiz != NULL) {

frontera(raiz);

}

}

template<typename A>

void Arbol<A>:: enorden(nodo\*raiz) {

if (raiz != NULL) {

enorden(raiz->menor);

cout<< "el Arbol es" << endl;

cout<< raiz->elem << endl;

listadefrontera.agregaralfinal(raiz->elem);

}

}

template<typename A>

void Arbol<A>:: listararbol(){

if (raiz != NULL) {

enorden(raiz);

}

}