TRABAJO PRACTICO ESPECIAL

Grupo 31

Pianzola Luca, Arispe Luis Agustín

lucapinzola0@gmail.com
agustinarispelopez@gmail.com

31/05/2022

—

Análisis y Diseño de Algoritmos

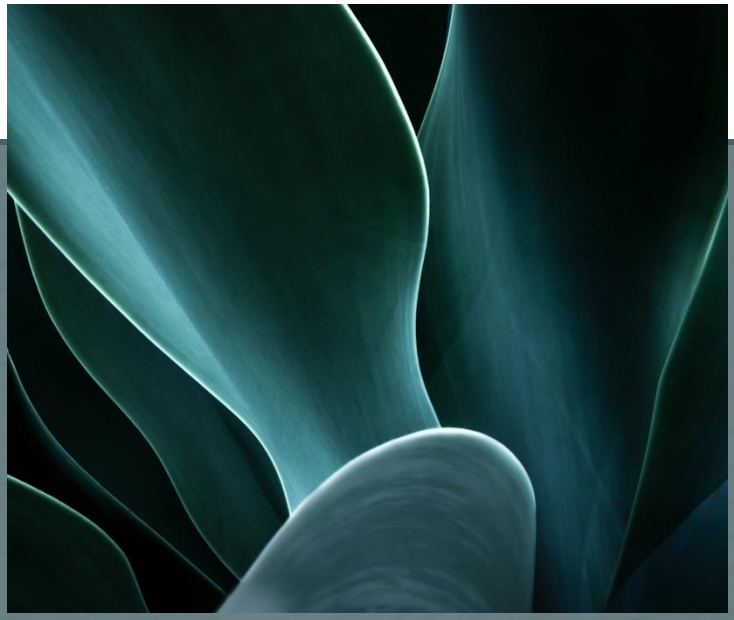
—

Introducción

El presente trabajo fue realizado con el objetivo de poner en práctica todos los conocimientos adquiridos en la cursada, además de fomentar el trabajo en equipo y la toma de decisiones en grupo.

Durante el desarrollo de todo este informe, se verán detallados todos los retos y complicaciones ocurridos a medida que se desarrollaba el trabajo, aparte de todas las decisiones que fueron tomadas en conjunto y las soluciones aplicadas a los problemas ya mencionados.

Por último, en este informe repasaremos todos los conceptos más importantes para entender el programa realizado y, se llevará a cabo una revisión completa y especifica de la complejidad temporal y el consumo de memoria ocupados por todas las estructuras y procedimientos aplicados.



Objetivos

1. **Objetivo Principal: Desarrollar e implementar un conjunto de operaciones mínima en c++ usando tipos de datos abstractos de estructuras Filas, Listas y Árbol binario.**
2. **Especificación formal en Nereus (sólo parte sintáctica) de los tipos de datos abstractos.**
3. **Para cada uno de los TDA detallar las estructuras de datos que se utilizaron para implementarlos.**
4. **Para cada una de las funciones especificadas en los TDA incluir una breve descripción y su complejidad temporal.**

¿Qué es una Fila?

Una fila es una estructura de datos caracterizada por ser una secuencia de elementos donde se realiza la operación de inserción en un extremo y la operación de extracción en el otro extremo, es decir, siguiendo el esquema FIFO (primero en entrar, primero en salir).

TDA FILA1

CLASS FILA [Elemento]
IMPORTS Natural, ListaSimple[Elemento]
BASIC CONSTRUCTORS inicFila, agregarFila
EFFECTIVE
TYPE Fila
OPERATIONS
Fila1: ->Fila1;
RetirarElemento: Fila1 * Elemento -> Fila1;
Insertar: Fila1 * Elemento -> Fila1;

esVacia: Fila1 -> Boolean;
CantElementos: Fila1-> Natural;

ConsultoUltimoValor: Fila1 -> Elemento;

¿Cómo implementarla?

En nuestro caso utilizamos a ListaSimple (explicado a continuación) como herencia, ya que, el concepto de fila es similar a una ListaSimple, a la cual solo se puede acceder al ultimo nodo e insertar en el primero. Entonces, adaptando la clase ListaSimple, se puede hacer la clase fila de forma fácil y rápida.

Explicación del código:

Fila1<T>::Fila1()
Fila1<T>::~Fila1(){
Utilizamos un nodo de lista llamado "cont" para poder realizar acciones de la clase herencia. Así todo es totalmente simple

int Fila1<T>::CantElementos() const{
bool Fila1<T>::esVacia() const{
Al ser una lista que no se puede recorrer la forma de obtener la cantidad de elementos es la misma. (cantidad de elementos es 0 si la lista está vacía).

void Fila1<T>::RetirarElemento(){
Pensamos en retirar el ultimo elemento de la fila como eliminar el elemento de la posición de la cantidad de nodos de una lista.

void Fila1<T>::Insertar(T NuevoDato){
Al ser una fila solo se puede agregar al principio.

T Fila1 <T>::ConsultoUltimoValor() {
Pensamos en consultar el ultimo a retirar como mostrar el ultimo de una lista.

¿Qué es una Lista Simple?

Una lista enlazada simple consta de un numero de nodos con dos componentes (campos), un enlace al siguiente nodo de la lista y un valor, que puede ser de cualquier tipo.

TDA LISTA SIMPLE

CLASS ListaSimple [Elemento]
IMPORTS Natural
BASIC CONSTRUCTORS inicLista, agregarLista
EFFECTIVE
TYPE ListaSimple
OPERATIONS
Lista: -> Lista;
InsertarALista: Lista * Elemento -> Lista;
BorrarLista: Lista -> Lista;
AvanzarLista: Lista -> Lista;
InicioLista: Lista -> Lista;
PerteneceLista: Lista * Elemento-> Boolean;
InsertarFLista: Lista * Elemento -> Lista;
InsertarPLista: Lista * Elemento -> Lista;
VaciaLista: Lista -> Boolean;
CantLista: Lista -> Natural;
END-CLASS

¿Cómo Implementarla?

Creamos un "NodoLista" el cual se conforma por el dato abstracto "T" y un puntero a siguiente, además de un puntero que apunta al primero todo el tiempo, llamado "Primero" y un cursor publico que nos permite movernos en la lista desde el código principal y no desde la clase.

Explicación del código:

void ListaSimple<T>::AvanzarLista()
void ListaSimple<T>::InicioLista()
Son procedimientos que hacen que funcione "CursorPublico" "avanzarlista" se mueve al siguiente e "iniciolista" al comienzo. O(1) solo realiza una igualación y constantes.

ListaSimple<T>::ListaSimple()
ListaSimple<T>::~ListaSimple(){
Se declara a "Primero", a "CursorPublico" y a su destructor. O(1) solo declara.
int ListaSimple<T>::CantLista() const

Si la lista esta vacia devuelve 0 y sino recorre hasta el final sumando +1 por cada nodo que encuentre. Pertenece a O(n) ya que recorre todo el arreglo.

bool ListaSimple<T>::VaciaLista() const

Si el "Primero" no tiene un elemento devuelve true, solo en ese caso. Pertenece a O(1) ya que no recorre la lista.

T ListaSimple<T>::MostrarElemento(int contador) const

Dada una posición llamada "contador" va recorriendo la lista, por cada nodo que recorre va restando -1 al contador, cuando llega a cero retorna el elemento en el que se encuentra. O(n) ya que en el peor caso recorre toda la lista.

void ListaSimple<T>::BorrarLista(int contador)

Es parecido a mostrar solamente que hace un *delete* del nodo en el que se encuentra sin perder la lista. O(n) hace el mismo recorrido que mostrar.

bool ListaSimple<T>::PerteneceLista(T Comparado)const

Recorre comparando cada nodo. Si el nodo actual es igual al comparado retorna true. O(n) en peor caso recorre todo el arreglo.

void ListaSimple<T>::InsertarFLista(T NuevoDato)

void ListaSimple<T>::InsertarPLista(T NuevoDato)

void ListaSimple<T>:: InsertarALista(T NuevoDato,int

Agrega un nodo a la lista "P" refiere a principio, "F" a final y "A" a arbitrario o posicional. O(1) para InsertarPlista, los otros 2 son O(n) (InsertarALista recorre en el peor caso toda la lista).

¿Qué es un Árbol Binario? Un Árbol consta de un conjunto finito de elementos, denominados nodos y un conjunto finito de líneas dirigidas, denominadas ramas, que conectan los nodos.

TDA ARBIN

CLASS Arbin [Elemento]

IMPORTS Natural, ListaSimple [Elemento]

BASIC CONSTRUCTORS InicArbin, AgregarArbin

EFFECTIVE

TYPE Arbin

OPERATIONS

Arbin: -> Arbin;

AgregarArbin: Arbin * Elemento -> Arbin;

CantArbin: Arbin -> Natural;

Listar: Arbin -> Lista[Elemento];

VacioArbin: Arbin -> Boolean;

PerteneceArbin: Arbin * Elemento -> Boolean;

ProfArbin: Arbin -> Natural;

FrontArbin: Arbin -> Lista[Elemento];

END-CLASS

¿Cómo Implementarla?

Creamos un "NodoArbin" conformado por el dato abstracto, un puntero por la izquierda y otro por la derecha. En nuestro caso nos basamos en un árbol balanceado ordenado de menor a mayor con los menores a la izquierda y los mayores a la derecha. También un nodo "Raíz" que siempre apunta al comienzo del árbol. Para recorrer el árbol necesitamos utilizar recursión, por lo tanto en c++ utilizamos el *private* para poder recorrer utilizando este método.

Explicación del código:

Arbin<T>::Arbin()
Arbin<T>::~Arbin()
Se declaran los nodos en *null. O(1)*

*void Arbin<T>::AgregarArbinP(NodoArbin *& Arbol,const T dato)*
void Arbin<T>::AgregarArbin(const T dato)

Se agrega un dato ordenado recorriendo por condiciones el árbol. Pertenece a O(n/2) ya que suponemos que el árbol esta balanceado.

*void Arbin<T>::ListarP(NodoArbin * Arbol, ListaSimple<T> & Lista) const*
void Arbin<T>::Listar(ListaSimple<T> & Lista) const
Dada una ListaSimple, recorro todo el árbol in-order agregando a la lista todos los nodos. Pertenece a O(n) ya que tiene que recorrer todo el arbol.

*int Arbin<T>::ProfArbinP(NodoArbin * Arbol) const*
int Arbin<T>::ProfArbin() const
Recorre el subárbol izquierdo y el subárbol derecho +1 cada vez que recorre, el que sea mayor será la profundidad del árbol. O(n/2) el árbol esta balanceado.

bool Arbin<T>::VacioArbin() const
Si Raíz no tiene elemento devuelve true O (1).

*int Arbin<T>::CantArbinP(NodoArbin * Arbol) const*
int Arbin<T>::CantArbin() const
Recorro todo el árbol sumando +1 por cada nodo que encuentro. O(n) ya que tiene que recorrer todo el árbol.

*void Arbin<T>::FrontArbinP(NodoArbin * Arbol, ListaSimple<T> & Lista) const*
void Arbin<T>::FrontArbin(ListaSimple<T> & Lista) const
Recorro el árbol completo, si por izquierda y por derecha no hay nodos (es decir, sin hijos), lo agrega a la ListaSimple. O(n) por obvios motivos.

*void Arbin<T>::PerteneceArbinP(NodoArbin * Arbol, const T dato,int siono)*
bool Arbin<T>::PerteneceArbin(const T dato)
Recorro el arbol in-order hasta encontrar un dato coincidente si lo encuentro devuelvo true. O(n/2)

Conclusión:

Para finalizar, gracias a este trabajo, pudimos poner el practica todo el desarrollo teórico-practico de la cursada y pudimos aprender el manejo de nuevas herramientas. Por una parte, el trabajo nos brindó infinida de retos y complicaciones, dándonos así la posibilidad de poder resolverlas como grupo.