Trabajo especial de Análisis y diseño de algoritmos I

Grupo: 5

Integrantes:

Coria, Santiago

Santos, Matias Andres

Contacto: santiagocoria@live.com.ar
matias_a_santos@hotmail.com

Índice

Introducción

En este Trabajo Práctico Especial veremos como se especifica una clase en el lenguaje NEREUS, y a partir de la misma, como se implementa de NEREUS a c++. Aprenderemos a armar clases parametrizadas, como hacer herencia de clases, y veremos distintos ejemplos de algoritmos de tipo ”divide y conquista” que se basan en desglosar un problema grande y difícil de encarar, en problemas más pequeños para facilitar su comprensión e implementación. También tocaremos temas tales como la diferencia entre una clase estándar y una clase abstracta, y cual es la funcionalidad de cada una.

Hemos creado distintos TDAs (tipos de datos abstractos) cada uno con una característica especial que los diferencia del resto, como por ejemplo pila, fila o lista. A su vez, especificamos distintos objetos, tipos de datos abstractos más tangibles y fáciles de entender, los cuales utilizamos para calcular el área total de un cohete en 2 dimensiones, calcular la longitud total de una sumatoria de segmentos, y eliminar una cierta cantidad de valores menores al promedio de una lista vinculada.

Capítulo 1

TDA Pila genérica

1. El TDA Pila genérica es una clase creada en forma de lista vinculada, que permite extraer de la misma el último dato que fue ingresado, conocida como LIFO (Last in-first out) por sus siglas en inglés.

1.1. Especificación algebraica

1.1.1.

Class Pila
Imports Real,boolean,Telem
Basic constructors inicpila, agregarpila;
Effective
Type Pila
Operations
inicpila: -> Pila;
agregarpila: Pila * Real -> Pila;
pilavacia: Pila -> boolean;
topepila: Pila -> Telem;
eliminartope: Pila -> Pila;
Axioms
valor: Telem; p: pila;

pilavacia(inicpila())=true;
pilavacia(agregarpila(p,valor))=false;
eliminartope(agregarpila(p,valor))=p;
eliminartope(inicpila())=inicpila();
topepila(agregarpila(p,valor)=valor;
topepila(inicpila())=' ';

End-class

1.1.2. Clasificación de las operaciones

Constructoras:

- Inicpila
- agregarpila

Observadoras:

- Pilavacia
- Topepila

Modificadoras:

- EliminarTope

1.2. Clase Pila en c++

1.2.1.

```
1  #ifndef PILA_H
2  #define PILA_H
3
4  template <typename Telem> class Pila
5  {
6    public:
7          Pila();
8          ~Pila();
9          bool eliminatope();
10         void agregapila(const Telem & a);
11         bool pilavacia()const;
12         const topepila()const;
13   private:
14         struct Nodo{
15             Telem elem;
16             Nodo*sig;};
17
18         Nodo * primero;
19
20         void eliminapila();
21 };
22
23 #endif
```

1.2.2.

Se utiliza una estructura de lista vinculada, ya que es eficiente a la hora de implementar una pila, porque se puede modificar el primer puntero cada vez que se quiere agregar o eliminar un elemento, y esto nos permite reducir la complejidad de la implementación a O(1) en todas las funciones, excepto en la que elimina la pila completa que pertenece a O(n) siendo n la cantidad de elementos de la pila.

1.2.3.

```
1  #include "Pila.h"
2  #include <iostream>
3
4  template <typename Telem> Pila<Telem>::Pila(){
5
6    //constructor de la clase
7    primero=NULL;
8  }
9
10 template <typename Telem> Pila<Telem>::~Pila(){
11
12   //destructor de la clase
13     eliminapila();
14   primero=NULL;
15
16 }
17 template<typename Telem> bool Pila<Telem>::eliminatope(){
18
19   //se encarga de eliminar el ultimo elemento insertado si lo hace bien
devuelve true
20
21   if(false==pilavacia()){
22         Nodo*aux=primero;
23         primero=primero->sig;
24         delete aux;
25         return true;}
26   else
27         return false;
28 }
29
30 template<typename Telem> void Pila<Telem>::eliminapila(){
31
32   //elimina todos los elementos de la pila
33   Nodo*aux=primero;
34   while (primero!=NULL){
35         primero=primero->sig;
36         delete aux;
37         aux=primero;}
38 }
39
40 template<typename Telem> void Pila<Telem>::agregapila(const Telem & a){
41
42   //agrega elementos a la pila
43   Nodo*aux=new Nodo;
44   aux->elem=a;
45   aux->sig=primero;
46   primero=aux;
47 }
48 template<typename Telem> bool Pila<Telem>::pilavacia()const{
49
50   //devuelve un booleano dependiendo si la pila esta vacia o no
51   if (primero==NULL)
```

```
52          return true;
53    else
54    return false;
55  }
56
57  template <typename Telem> const Pila<Telem>::topepila()const{
58
59    //devuelve el tope de la pila
60    if (primero!=NULL)
61          return primero->elem;
62    else
63          return 0;
64  }
65
```

1.2.4. dejado intencionalmente en blanco

1.2.5.

Métodos Públicos

- ***inicpila***: inicializa la pila en NULL. Su complejidad pertenece a O(1).
- ***Agregarpila***: Agrega un elemento de cualquier tipo a la pila. Su complejidad pertenece a O(1).
- ***pilavacia***: pregunta si la pila es NULL, en caso de serlo devuelve true si no false. Su complejidad pertenece a O(1).
- ***topepila***: devuelve el último elemento ingresado. Su complejidad pertenece a O(1).
- ***eliminartope***: elimina el último elemento ingresado. Su complejidad pertenece a O(1).

Métodos Privados

Estructura de la pila:

```
13    private:
14          struct Nodo{
15              Telem elem;
16              Nodo*sig;};
17
18          Nodo * primero;
19
20          void eliminapila();
21  };
```

La estructura de la pila está compuesta por un elemento templado y nodos vinculados por nodo*sig.

- ***eliminapila***: cuando se termine la ejecución, este procedimiento será el encargo de liberar todo el espacio de memoria. Su complejidad pertenece a O(n) siendo n la cantidad de nodos.

TDA Fila

2. El TDA Fila es una clase creada en forma de lista doblemente vinculada, que permite extraer de la misma el primer dato que fue ingresado, conocida como FIFO (First in-first out) por sus siglas en inglés.

2.1. Especificación algebraica

2.1.1.

Class Fila
Imports Real,boolean
Basic constructors inicfila, agregarelem
Effective
Type Fila
Operations
inicfila: -> Fila;
agregarelem: Fila * Elem -> Fila;
filavacia: Fila -> boolean;
eliminarprimero: Fila -> Fila;
pre (!filavacia);
recuperaprimero: Fila -> Elem;
pre (!filavacia);
Axioms
e: Elem; f: fila;

filavacia(inicfila())=true;
filavacia(agregarelem(f,e)=false;
eliminarprimero(inicfila())=inicfila();
eliminarprimero(agregarelem(f,e))=f;
(!filavacia(f)) eliminarprimero(agregarelem(f,e))=agregarelem(eliminarprimero(f),e);
recuperaprimero(inicfila())=' ';
recuperaprimero(agregarelem(f,e))=e;
(!filavacia(f)) recuperaprimero(agregarelem(f,e))=recuperaprimero(f);

End-Class

2.1.2. Clasificación de las operaciones

Constructoras:

- inicfila
- agregarelem

Observadora:

- filavacia
- recuperaprimero

Modificadoras:

- eliminarprimero

2.2. Clase Fila en c++

2.2.1.

```
1  #ifndef FILA_H
2  #define FILA_H
3
4
5  template <typename Telem> class Fila
6  {
7  public:
8    Fila();
9    virtual ~Fila();
10   void agregaElem(Telem a);
11   void eliminaPrimero();
12   Telem recuperaPrimero()const ;
13   bool filaVacia()const ;
14 private:
15   struct Nodof
16   {
17         Telem Elem;
18         Nodof*sig;
19         Nodof*ant;
20   };
21   Nodof*primero;
22   Nodof*ultimo;
23 };
24
25 #endif // FILA_H
```

2.2.2.

Se utiliza una estructura de lista doblemente vinculada debido a la eficacia que presenta cuando se quieren extraer más de un elemento de la misma.

2.2.3.

```
1  #include "Fila.h"
2  #include <iostream>
3
4  template <typename Telem> Fila<Telem>::Fila()
5  {
6    primero=NULL;
7    ultimo=NULL;
8  }
9
10 template <typename Telem>
11 Fila<Telem>::~Fila()
12 {
13   //dtor
14 }
15 template <typename Telem>
16 void Fila<Telem>::agregaElem(Telem a)
17 {
18   if (filaVacia())
19   {
20         primero=new Nodof;
21         primero->Elem=a;
22         primero->sig=NULL;
23         primero->ant=NULL;
24         ultimo=primero;
25   }
26   else
27   {
28         Nodof*aux=new Nodof;
29         aux->Elem=a;
30         aux->sig=primero;
31         aux->ant=NULL;
32         primero->ant=aux;
33         primero=aux;
34   }
35 }
36 template <typename Telem>
37 void Fila<Telem>::eliminaPrimero()
38 {
39   if (!filaVacia())
40   {
41         if (ultimo->ant==NULL)
42         {
43               delete primero;
44             primero=NULL;
45             ultimo=NULL;
46         }
47         else
48         {
49             ultimo->ant->sig=NULL;
50             Nodof*aux=ultimo;
51             ultimo=ultimo->ant;
```

```
52                delete aux;
53
54          }
55    }
56  }
57  template <typename Telem>
58  Telem Fila<Telem>::recuperaPrimero()const
59  {
60    if(!filaVacia())
61          return ultimo->Elem;
62  }
63  template <typename Telem>
64  bool Fila<Telem>::filaVacia()const
65  {
66    if (primero==NULL)
67          return true;
68    else
69          return false;
70  }
71
72
73  template class Fila <int>;
74  template class Fila <float>;
75  template class Fila <double>;
76
```

2.2.4. Dejado intencionalmente en blanco

2.2.5.

Métodos públicos

- ***inicfila***: inicia la fila en NULL. Su complejidad pertenece a O(1).
- ***agregarelem***: agrega un elemento templado al principio de lista. Su complejidad pertenece a O(1).
- ***eliminaprimero***: mientras la fila no esté vacía elimina el primer elemento ingresado en la fila. Su complejidad pertenece a O(1).
- ***recuperaprimero***: si la fila no se encuentra vacía devuelve el primer elemento insertado. Su complejidad pertenece a O(1).
- ***filavacia***: en caso de ser vacía la fila devuelve true , sino false. Su complejidad pertenece a O(1).

Métodos privados

Estructura privada de la fila:

```
17  private:
18    struct Nodof
19    {
20          Telem Elem;
21          Nodof*sig;
22          Nodof*ant;
23    };
24      Nodof*primero;
25      Nodof*ultimo;
26  };
```

La estructura de la fila está conformada por dos punteros, que apuntan al inicio y al fin de la fila respectivamente, y un nodo de lista que está doblemente vinculado, y a su vez tiene un elemento templado.

- ***eliminafila***: Elimina la fila entera y deja los punteros apuntando a NULL al final de su ejecución. Su complejidad pertenece a O(n), siendo n la cantidad de elementos de la fila.

TDA Lista

3. El TDA Lista es una clase básica de almacenamiento de datos, que ha sido provista de operaciones tales como cantidaddelem, elementopos, y agregarlista que permiten interactuar con los elementos que están en el medio de la lista, cosa que con la pila o la fila no se puede lograr.

3.1. Especificación algebraica

3.1.1.

Class Lista[Elem]
Imports Nat, Boolean
Basic constructors iniclista, agregarlista
Effective
Type Lista
Operations

iniclista: -> Lista;	constructora
agregarlista: Lista * Elem * Nat(i) -> Lista;	constructora
pre(i>0) && (i<cantidaddeelmen+1);	
agregarprincipio: Lista * Elem -> Lista;	modificadora
agregarfinal: Lista * Elem -> Lista;	modificadora
cantidaddeelem: Lista -> Nat;	observadora
listavacia: Lista -> Boolean;	
eliminarlista: Lista * Elem -> Lista;	modificadora
estaelemento: Lista * Elem -> Boolean;	observadora
elementopos: Lista * Nat(i) -> Elem;	observadora
pre(i>0) && (i<cantidaddeelmen+1);	

Axioms

l: lista; e, e1: Elem; i, pos: Nat;
agregarprincipio(iniclista(),e)=agregarlista(l,e,1);
agregarprincipio(agregarlista(l,e,1),e1)=agregarlista(l,e1,1);
agregarfinal(iniclista(),e)=agregarlista(l,e1,);
agregarfinal(agregarlista(l,e,i),e1)=agregarlista(l,e1,i);
(i != largo+1) agregarfinal(agregarlista(l,e,i),e1)=agregarlista(agregarfinal(l,e1),e,i);
cantidaddeelem(iniclista())=0;
cantidaddeelem(agregarlista(l,e,i))=cantidad;
cantidaddeelem(agregarlista(l,e,i))= 1 + cantidaddeelem(l);
listavacia(iniclista())=true;
listavacia(agregarlista(l,e,i))=false;
eliminarlista(iniclista(),e)=iniclista();
(e!=e1) eliminarlista(agregarlista(l,e,i)e1)=agregarlista(eliminarlista(l,e1),e,i);
(e==e1) eliminarlista(agregarlista(l,e,i)e1)=l;
estaelemento(iniclista(),e)=false;
(e!=e1) estaelemento(agregarlista(l,e,i),e1)=estaelemento(l,e);
(e==e1) estaelemento(agregarlista(l,e,i),e1)=true;
elementopos(iniclista(),i)=' ';
(i!=pos) elementopos(agregarlista(l,e,i),pos)=elementopos(l,pos);

(i==pos) elementopos(agregarlista(l,e,i),pos)=e;
End-Class

3.1.2. Clasificación de las operaciones

Constructoras:

- iniclista
- agregarlista
- agregarprincipio
- agregarfinal

Observadora:

- cantidaddeelem
- listavacia
- estaelemento
- elementopos

Modificadoras:

- eliminarlista

3.2. Clase Lista en c++

3.2.1.

```
 1  #ifndef LISTAS_H
 2  #define LISTAS_H
 3  #include <iostream>
 4
 5  using namespace std;
 6
 7  template <typename elem >
 8  class Listas
 9  {
10    public:
11          Listas();
12          ~Listas();
13          void agregarprincipio (const elem & elemento);
14          void agregarfinal (const elem & elemento);
15          void agregararbritario (const elem & elemento, unsigned int
pos);
16          unsigned int cantidadelem () const;
17          bool estaelemento (const elem & elemento)const;
18          const elem & elementopos (unsigned int pos)const;
19          bool esvacia () const;
20          void eliminarelem (const elem & elemento);
21    private:
22          struct nodo
23                {
24                    elem elemento;
25                  nodo * sig;
26                };
27          nodo * L;
28          int largo;
29          void vaciar();
```

```
30  };
31
32  #endif // LISTAS_H
```

3.2.2. Se utiliza una estructura de lista doblemente vinculada ya que en funciones tales como agregarlista, estaelemento, y elementopos se obtiene una complejidad perteneciente a O(n) siendo n la cantidad de elementos de la lista, pero en el resto las complejidades pertenecen a O(1), ya que son funciones que no necesitan recorrer la lista, lo que hace que las funciones sean eficientes.

3.2.3.

```
 1  #include "Listas.h"
 2  #include <iostream>
 3
 4  using namespace std;
 5
 6  template <typename elem> Listas<elem> ::Listas (){
 7   //Constructor de la clase
 8   L = NULL;
 9   largo = 0;
10  }
11
12  template <typename elem> Listas<elem> ::~ Listas(){
13   //destructor de la clase
14   vaciar();
15   L=NULL;
16  }
17
18   template <typename elem> void Listas<elem> :: agregarprincipio(const
elem & elemento){
19
20   //agregar al principio de la lista
21   nodo * aux = new nodo;
22   aux->elemento = elemento;
23   aux->sig=L;
24   L=aux;
25   largo++;
26  }
27
28  template <typename elem> void Listas<elem>::agregarfinal (const elem &
elemento){
29
30   //agrega al final de la lista
31   if (L=NULL){
32         agregarprincipio(elemento);}
33   else{
34         nodo * aux=L;
35         int i=0;
36         while (i<largo){
37               i++;
```

```
 38                aux=aux->sig;}
 39          nodo * aux1= new nodo;
 40          aux1->elemento=elemento;
 41          aux1->sig=NULL;
 42          aux->sig=aux1;
 43          largo++;
 44          }
 45   }
 46
 47    template <typename elem> void Listas<elem>::agregararbritario (const
elem & elemento, unsigned int pos){
 48
 49    //agrega en una posicion ingressada por el usuario
 50    if (pos==1){
 51          agregarprincipio(elemento);}
 52    else
 53          if (pos>= largo+1){
 54                agregarfinal(elemento);}
 55          else
 56                {   nodo * aux=L;
 57                int i=0;
 58                while (i<pos-1) {
 59                      i++;
 60                      aux=aux->sig;}
 61                nodo * insertar= new nodo;
 62                insertar->elemento=elemento;
 63                insertar->sig=aux->sig;
 64                aux->sig=insertar;
 65                largo++;
 66                }
 67   }
 68
 69    template <typename elem> unsigned int Listas<elem>::cantidadelem ()
const{
 70
 71    //devuelve la cantidad de nodos de la lista
 72    return largo;
 73   }
 74
 75   template <typename elem> bool Listas<elem>::estaelemento (const elem &
elemento)const
 76   {
 77    //devuelve si se encuentra un elemento de la lista preguntado por el
usuario
 78    int i=0;
 79    nodo * aux= L;
 80    while (i < largo)
 81    {
 82          if (aux->elemento == elemento)
 83                return true;
 84          else
 85                {
 86                i++;
```

```
 87                aux=aux->sig;
 88                }
 89    }
 90    return false;
 91   }
 92
 93     template <typename elem> const elem & Listas<elem>::elementopos
(unsigned int pos)const{
 94    //devuelve true si esta el elemento en la lista en caso de no estar
false
 95    int i=0;
 96    nodo * aux=L;
 97    while ((i<largo)&&(i<pos))
 98          {
 99                i++;
100              aux=aux->sig;
101          }
102    if ((i==pos)&&(i<largo))
103          return aux->elemento;
104    else
105          return false;
106   }
107
108   template <typename elem> bool Listas<elem>::esvacia () const{
109
110    //devuelve true si no hay elementos en la lista sino un false
111    if (largo==0)
112          return true;
113    else
114          return false;
115   }
116
117   template <typename elem> void Listas<elem>::eliminarelem (const elem &
elemento){
118
119    //elimina un elemento que es ingresado por el usuario
120    int i=0;
121    if (!esvacia()){
122           nodo * aux =L;
123          while (i < largo){
124                      if ((aux->sig != NULL) && (aux->sig->elemento ==
elemento)){
125                   nodo * AEliminar=aux->sig;
126                   aux->sig=aux->sig->sig;
127                   delete AEliminar;
128                 AEliminar=NULL;
129                   i++;
130                   largo--;}
131               else{
132                   i++;
133                   aux=aux->sig;}
134               }
135          }
```

```
136  }
137
138  template <typename elem> void Listas<elem>::vaciar(){
139
140   //elimina toda la lista
141   nodo * AEliminar=NULL;
142   while (!esvacia()){
143          AEliminar=L;
144         L=L->sig;
145         delete AEliminar;
146   }
147  }
148
149  template class Listas <int>;
150  template class Listas <float>;
151  template class Listas <double>;
```

3.2.4. Dejado intencionalmente en blanco

3.2.5.

Métodos públicos

- ***iniclista***: Inicializa los punteros en NULL y la variable largo en 0. Su complejidad pertenece a O(1).
- ***agregarprimero***: Se le da un elemento templado y lo agrega al principio de la lista. Su complejidad pertenece a O(1).
- ***agregarfinal***: Se le da un elemento templado y recorre la lista hasta llegar al final, para insertarlo en ese lugar. Su complejidad pertenece a O(n) siendo n la cantidad de nodos de la lista.
- ***agregararbitrario***: Se le da un elemento templado y una posición, la lista es recorrida hasta la posición dada, donde el elemento se inserta en la misma. Su complejidad pertenece a O(n) siendo n la cantidad de elementos de la lista.
- ***cantidadelem***: Devuelve el largo de la lista. Su complejidad pertenece a O(1).
- ***estaelemento***: Se le da un elemento templado y se recorre la lista en busca del mismo, en caso de estar, devuelve true, sino devuelve false. Su complejidad pertenece a O(n) siendo n la cantidad de elementos de la lista.
- ***elementopos***: Se le da una posición y devuelve el elemento que se encuentra en esa posición de la lista. Su complejidad pertenece a O(n) siendo n la cantidad de elementos de la lista.
- ***esvacia***: Devuelve true si la lista no tiene elementos, en caso contrario devuelve false. Su complejidad pertenece a O(1).
- ***eliminarelem***: Se le da un elemento y en caso de estar, lo borra de la lista. Su complejidad pertenece a O(n) siendo n la cantidad de elementos de la lista.

Métodos privados

Estructura privada de la lista:

```
22  private:
23    struct NodoL
24    {
25          elem elemento;
26          NodoL * sig;
27    };
28    NodoL * L;
29      int largo;
30    void vaciar();
31  };
```

La estructura privada de la lista está conformada por un nodo que tiene un elemento templado y un puntero que apunta al siguiente nodo, además cuenta con una variable llamada largo, que cuenta la cantidad de nodos de la lista.

- ***vaciar***: Elimina todos los nodos de la lista y deja apuntando el puntero L a NULL. Su complejidad pertenece a O(n) siendo n la cantidad de elementos de la lista.

Capítulo 2

Problema 1

Genera una lista de N puntos 2d (x,y) y una lista de segmentos generados con los primeros $\left[\frac{n}{2}\right]$ primeros pares de puntos, y calcula la suma de las longitudes de los segmentos generados.

TDA Punto

1. El TDA Punto está compuesto por un nodoP, que a su vez, está conformado por una coordenada en *x*, y una coordenada en *y*.

1.1. Especificación algebraica

1.1.1.

Class Punto
Imports cmath, Real, Boolean
Basic constructors crearPunto
Effective
Type Punto
Operations

crearPunto : Real* Real -> Punto;
coordx : Punto -> Real;
coordy : Punto -> Real;
distancia: Punto * Punto -> Real;
trasladar : Punto * Real * Real -> Punto;
igual: Punto * Punto -> Boolean;
distinto: Punto * Punto -> Boolean;
suma: Punto * Punto -> Punto;
asignacion: Punto * Punto -> Punto;

Axioms

x1,y1,x2,y2 :real; p1,p2,p3 :punto;

coordx(p1)=x1;
coordy(p1)=y1;
distancia(p1,p2)=sqrt(pow(coordx(p1)-coordx(p2),2)+pow(coordy(p1)-coordy(P2),2));
trasladar(p1,x2,y2)=p2;

(coordx(p1)==coordx(p2)) && (coordy(p1)==coordy(p2)) igual(p1,p2)= true;
(coordx(p1)!=coordx(p2)) && (coordy(p1)!=coordy(p2)) igual(p1,p2)=false;
(coordx(p1)==coordx(p2)) && (coordy(p1)==coordy(p2)) distinto(p1,p2)=false;
(coordx(p1)!=coordx(p2)) && (coordy(p1)!=coordy(p2)) distinto(p1,p2)=true;

suma(p1,p2)=p3;
asignacion(p1,p2)=p3;

End-class

1.1.2. Clasificación de las operaciones

Constructoras:

- crearPunto

Observadora:

- coordx
- coordy
- distancia
- igual
- distinto

Modificadoras:

- trasladar
- suma
- asignacion

1.2. Clase Punto en c++

1.2.1.

```
1  #ifndef PUNTO_H
2  #define PUNTO_H
3  #include <cmath>
4
5
6  class Punto
7  {
8  public:
9    Punto() {};
10   Punto(const float&x, const float&y)
11   {
12         {
13               //constructor de la clase
14               this->x=x;
15               this->y=y;
16         }
17   };
18   ~Punto() {};
19   float coordx () const;
20   float coordy () const;
21   float distancia (const Punto&otropunto)const;
22   void trasladar (float x, float y);
23   bool operator== (const Punto&otropunto)const;
24   bool operator!= (const Punto&otropunto)const;
25   void operator+ (const Punto&otropunto);
26   Punto operator= (const Punto&otropunto);
27 private:
28   float x;
29   float y;
30 };
31
32 #endif // PUNTO_H
```

1.2.2.

Se implementa una estructura de tipo nodo, el cual está compuesto de una coordenada en *x*, y una en y , con estas dos coordenadas se conformará el punto.

1.2.3.

```
 1  #include "Punto.h"
 2
 3  float Punto::coordx()const
 4  {
 5    //devuelve la coordenada del punto en x
 6    return this->x;
 7  }
 8
 9  float Punto::coordy()const
10  {
11    //devuelve la coordenada del punto en y
12    return this->y;
13  }
14
15  float Punto::distancia(const Punto&otropunto) const
16  {
17    //devuelve la distancia entre dos puntos
18    return          sqrt(pow(this->x-otropunto.coordx(),2.0)+pow(this->y-
otropunto.coordy(),2.0));
19  }
20
21  void Punto::trasladar (float x, float y)
22  {
23    //mueve el punto inicial segun las entradas del usuario
24    this->x +=x;
25    this->y +=y;
26  }
27
28  bool Punto::operator==(const Punto&otropunto)const
29  {
30    //compara si dos puntos son iguales y en caso de serlos devuelve true
31    return       (this->x==       otropunto.coordx())       &&       (this-
>y==otropunto.coordy());
32  }
33  bool Punto::operator!=(const Punto&otropunto)const
34  {
35    //compara si dos puntos son distintos y en caso de serlos devuelve
true
36    return       (this->x!=       otropunto.coordx()       ||       (this-
>y!=otropunto.coordy()));
37  }
38  void Punto::operator+(const Punto&otropunto)
39  {
40    //suma cada coordenada y se la asigna al primer punto
```

```
41      this->x+=otropunto.x;
42      this->y+=otropunto.y;
43    }
44    Punto Punto::operator=(const Punto&otropunto)
45    {
46      //asigna los valores de otropunto al punto con el que se llama
47      this->x=otropunto.coordx();
48      this->y=otropunto.coordy();
49    }
```

1.2.4. Dejado intencionalmente en blanco

1.2.5.

Métodos públicos

- ***crearPunto***: Se le pasa dos valores reales y son asignados a la coordenada x , y . Su complejidad pertenece a O(1).
- ***coordx***: devuelve la coordenada x del punto. Su complejidad pertenece a O(1).
- ***coordy***: devuelve la coordenada y del punto. Su complejidad pertenece a O(1).
- ***distancia***: con dos puntos devuelve la distancia a la que se encuentran. Su complejidad pertenece a O(1).
- ***trasladar***: se le pasa dos reales y modifica el punto.Su complejidad pertenece a O(1).
- ***igual***: compara dos puntos y si son iguales devuelve true, en caso contrario false.Su complejidad pertenece a O(1).
- ***distinto***: compara dos puntos y si son distintos devuelve true, en caso contrario false.Su complejidad pertenece a O(1).
- ***suma***: suma al punto que tenes otro punto. Su complejidad pertenece a O(1).
- ***asignacion***: modifica el primer punto con los valores del segundo.Su complejidad pertenece a O(1).

Métodos privados

Estructura del punto:

```
private:
21      float x;
22      float y;
23    };
```

Para definir un punto usamos dos reales, los cuales fueron llamados x e y

TDA Segmento

2. El TDA Segmento está compuesto por 2 puntos, distintos entre ellos.

2.1. Especificación algebraica

2.1.1.

Class Segmento
Imports Punto,Real
Basic constructors crear
Effective
Type Segmento
Operations

crearSegmento: Punto(p1) * Punto(p2) -> Segmento;
pre ((coordx(p1)!=coordx(p2)) || (coordy(p1)!=coordy(p2)))
extremo1: Segmento -> Punto1; observadora
extremo2: Segmento -> Punto2; observadora
longitud: Segmento -> Real; observadora
trasladarSegmento: Segmento * Real * Real * Nat(i) -> Segmento;
pre ((i==1) || (i==2))
_=_: Segmento * Segmento -> boolean; observadora

Axioms

p1, p2, p3, p4: Punto; x, y: Real; i: Nat;

extremo1(crear(p1,p2))= p1;
extremo2(crear(p1,p2))=p2;
longitud(crear(p1,p2))=distancia(p1,p2);
((coordx(p1)!=x) || (coordy(p1)!=y)) trasladar(crear(p1,p2),x,y,i)=crear(crearPunto(x,y),p2);
((coordx(p2)!=x) || (coordy(p2)!=y)) trasladar(crear(p1,p2),x,y,i)=crear(p1,crearPunto(x,y));
((igual(p1,p3) && (igual(p2,p4))) _=_(crear(p1,p2),crear(p3,p4))= true;
((!igual(p1,p3) && (!igual(p2,p4))) _=_(crear(p1,p2),crear(p3,p4))= false;

End-class

2.1.2. Clasificación de las operaciones

Constructoras:

- crearSegmento

Observadora:

- extremo1
- extremo2
- longitud

Modificadoras:

- trasladarSegmento

2.2. Clase Cuadrado en c++

2.2.1.

```
 1  #ifndef SEGMENTO_H
 2  #define SEGMENTO_H
 3  #include "Punto.h"
 4
 5  class Segmento
 6  {
 7    public:
 8          Segmento(const Punto ext1, const Punto ext2);
 9          const Punto & extremo1 () const;
10          const Punto & extremo2 () const;
11          float longitud () const;
12          void trasladar (float opcion,float x, float y);
13          bool operator==(const Segmento & otrosegmento)const;
14    private:
15          Punto ext1, ext2;
16  };
17
18  #endif // SEGMENTO_H
```

2.2.2. Implementamos 2 variables tipo punto, asegurándonos de que sean distintas entre sí.

2.2.3.

```
 1  #include "Segmento.h"
 2  #include <Punto.h>
 3
 4  Segmento::Segmento(const Punto ext1, const Punto ext2){
 5
 6    //Constructor de la clase
 7    this->ext1= ext1;
 8    this->ext2= ext2;
 9  }
10
11  const Punto & Segmento::extremo1()const{
12    //devuelve el primer extremo del segmento
13    return ext1;
14  }
15
16  const Punto & Segmento::extremo2()const{
17    //devuelve el segundo extremo del segmento
18    return ext2;
19  }
20
21  float Segmento::longitud()const{
22    //devuelve la distacia que hay entre los dos puntos que conforman el
segmento
23    return ext1.distancia(ext2);
24  }
25
26  void Segmento::trasladar(float opcion, float x, float y){
```

```
27
28    //dependiendo de que opcion elige el usuario mueve alguno de los dos
extremos
29    if (opcion== 1)
30          ext1.trasladar(x,y);
31    else
32          if (opcion==2)
33              ext2.trasladar(x,y);
34  }
35
36  bool Segmento::operator==(const Segmento & otroSegmento) const {
37
38    //compara si dos segmentos son iguales
39    return (ext1 == otroSegmento.ext1) && (ext2 == otroSegmento.ext2);
40  }
```

2.2.4. Dejado intencionalmente en blanco

2.2.5.

Métodos públicos

- ***crearSegmento***: Se le dan dos puntos y genera un segmento.Su complejidad pertenece a O(1).
- ***extremo1***: A partir del segmento devuelve el punto 1. Su complejidad pertenece a O(1).
- ***extremo2***: A partir del segmento devuelve el punto 2. Su complejidad pertenece a O(1).
- ***longitud***: Devuelve la distancia entre los dos puntos utilizando el procedimiento distancia de la clase Punto. Su complejidad pertenece a O(1).
- ***trasladarSegmento***: Modifica el punto deseado del segmento a partir de 2 reales. Su complejidad es O(1).
- _**=**_: Se le da otro segmento, y si estos son iguales devuelve true, y en caso contrario devuelve false. Su complejidad es O(1).

Métodos privados

Estructura privada del segmento:

```
14    private:
15          Punto ext1, ext2;
```

Con dos puntos formamos la estructura denominada segmento.

1.2. Resolución del problema

1.2.1. Dejado intencionalmente en blanco

1.2.2. Codificacion en c++

```
32    int tam = LisSeg.cantidadelem();
33    for(int i=0; i<tam; i++){
34          Segmento Aux=LisSeg.elementopos(i);
35          sumatoria+= Aux.longitud();
36    }
37    cout<< "resultado del problema 1: " << sumatoria ;
```

1.2.3. Complejidad algorítmica

$$\sum_{i=0}^{n} \quad c_0 = c_0 * n + c_0$$

Siendo n la cantidad de segmentos que hay en la lista, y c_0 el tiempo que tarda en hacerse cada sumatoria.

Problema 2

Diseña e implementa un algoritmo para eliminar los $\left\lceil \frac{n}{2} \right\rceil$ enteros más pequeños de un conjunto S de n enteros distintos.

TDA Conjunto

2. El TDA Conjunto está compuesto por una lista vinculada, que a su vez, está ordenada de menor a mayor.

2.1. Especificación algebraica

2.1.1.

Class Conjunto
Imports Nat
Basic constructors Iniconj, Agregar
Effective
Type Conjunto
Operations
Iniconj: -> Conjunto;
Agregar: Conjunto (c1)* Elem(e1) -> Conjunto;
pre (!Pertenece(c1,e1));
Esvacio: Conjunto -> Boolean;
Pertenece: Conjunto * Elem -> Boolean;
Tamanio: Conjunto -> Nat;
Eliminar: Conjunto * Elem> Boolean;
Unionconjunto: Conjunto * Conjunto -> Conjunto;
Interseccion: Conjunto * Conjunto -> Conjunto;
Axioms
valor, valor2 :ELEM; k:Nat; c1, c2: Conjunto;

Esvacio(Iniconj())=True;
Esvacio(Agregar(c1,valor))=False;
Pertenece(Iniconj(),valor)=False;
(valor==valor2) Pertenece(Agregar(c1,valor2),valor)=True;
(valor!=valor2) Pertenece(Agregar(c1,valor2),valor)=Pertenece(c1,valor);
Tamanio(Iniconj())=0;
Tamanio(Agregar(c1,valor))=1 + Tamanio(c1);
Eliminar(Iniconj(),valor)= Iniconj();
Eliminar(Agregar(c1,valor),valor)=c1;
Eliminar(Agregar(c1,valor),valor2)=Agregar(Eliminar(c1,valor2),valor))
(!Pertenece(c1,e2)) Unionconjunto(c1,Agregar(c2,e2))=Unionconjunto(Agregar(c1,e2),c2);
(Pertenece(c1,e2)) Unionconjunto(c1,Agregar(c2,e2))=Unionconjunto(c1,c2);
Unionconjunto(Iniconj(),c2)=c2;
Unionconjunto(c1,Iniconj())=c1;

Unionconjunto(Iniconj(),Iniconj())=Iniconj();

(Pertenece(c1,e2)) Interseccion(c1,Agregar(c2,e2))=Interseccion(Agregar(c3,e2),c2);

(!Pertenece(c1,e2)) Interseccion(c1,Agregar(c2,e2))=Interseccion(c1,c2);

Interseccion(Iniconj(),c2)=Iniconj();

Interseccion(c1,Iniconj())=Iniconj();

Intersección(Iniconj(),Iniconj())=Iniconj();

End-class

2.1.2 Clasificación de las operaciones

Constructoras:

- Iniconj
- Agregar

Observadoras:

- Esvacio
- Pertenece
- Tamanio

Modificadoras:

- Eliminar
- Interseccion
- Unionconjunto

2.2. Clase Conjunto en c++

2.2.1.

```
1  #ifndef CONJUNTOS_H
 2  #define CONJUNTOS_H
 3
 4  template <typename elem> class Conjuntos
 5  {
 6    public:
 7          Conjuntos();
 8          ~Conjuntos();
 9          bool agregardato(const elem & dato);
10          bool pertenece(const elem & dato)const;
11          bool esvacio() const ;
12          unsigned int tamanio()const;
13          const devuelvevalor(const int pos)const;
14          bool eliminar (const elem & dato);
15          void unionconjunto (const Conjuntos & otroconjunto);
16          void interseccion (const Conjuntos & otroconjunto);
17          void mueveactual();
18          bool finactual();
19          void resetactual();
20          const elem devuelveactual();
21    private:
22          struct NodoC
23          {
24              elem elemento;
```

```
25              NodoC*sig;
26          };
27          NodoC*conj;
28          NodoC*actual;
29          int largo;
30          void vaciar();
31          void agregar(NodoC *&punt,NodoC *& aux);
32  };
33
34
35  #endif // CONJUNTOS_H
```

1.2.2.

Se implementan 3 variables privadas, las cuales guardan el inicio de la lista del conjunto, un iterador adicional, y la cantidad de elementos del conjunto.

1.2.3.

```
 1  #include "Conjuntos.h"
 2  #include <iostream>
 3
 4  template <typename elem> Conjuntos<elem>::Conjuntos()
 5  {
 6   //Constructor de la clase
 7   conj=NULL;
 8   actual=NULL;
 9   largo=0;
10  }
11  template <typename elem> Conjuntos<elem>::~Conjuntos()
12  {
13   //Destructor de la clase
14   vaciar();
15   largo=0;
16  }
17  template <typename elem> bool Conjuntos<elem>::pertenece(const elem &
elemento)const
18  {
19   //Devuelve true en caso de que el elemento este en caso contrario
false
20   NodoC * cursor = conj;
21   while ((cursor!=NULL)&&(cursor->elemento!=elemento))
22         cursor=cursor->sig;
23   if ((cursor!=NULL)&&(cursor->elemento==elemento))
24         return true;
25   else
26         return false;
27  }
28
29  template <typename elem> void Conjuntos<elem>::agregar(NodoC *&
punt,NodoC *& aux)
30  {
```

```
 31
 32    if ((punt==NULL) || (punt->elemento > aux->elemento))
 33    {
 34          aux->sig=punt;
 35          punt=aux;
 36          largo++;
 37    }
 38    else
 39    {
 40          agregar(punt->sig,aux);
 41    }
 42
 43   }
 44   template <typename elem> bool Conjuntos<elem>::agregardato(const elem
& dato)
 45   {
 46    //Agrega un elemento ordenado en el conjunto
 47    NodoC*aux= new NodoC;
 48    aux->elemento=dato;
 49    aux->sig=NULL;
 50    NodoC*punt=conj;
 51    agregar(punt,aux);
 52    conj=punt;
 53   }
 54
 55   template <typename elem> bool Conjuntos<elem>::esvacio()const
 56   {
 57    //Devuelve true en caso de que el conjunto sea vacio sino un false
 58    if (largo==0)
 59          return true;
 60    else
 61          return false;
 62   }
 63   template <typename elem> bool Conjuntos<elem>::eliminar(const
elem&elemento)
 64   {
 65    //elimina un elemento que es ingresado por el usuario
 66    int i=0;
 67    if (!esvacio())
 68    {
 69
 70          if (conj->elemento == elemento)
 71          {
 72                NodoC * aux =conj;
 73                conj=conj->sig;
 74                delete aux;
 75          }
 76          else
 77          {
 78                NodoC*aux=conj;
 79                while (i < largo)
 80                {
```

```
 81                if ((aux->sig != NULL) && (aux->sig->elemento ==
elemento))
 82                {
 83                      NodoC * AEliminar=aux->sig;
 84                      aux->sig=aux->sig->sig;
 85                      delete AEliminar;
 86                      AEliminar=NULL;
 87                      i++;
 88                      largo--;
 89                }
 90                else
 91                {
 92                      i++;
 93                      aux=aux->sig;
 94                }
 95               }
 96          }
 97    }
 98   }
 99   template <typename elem> void Conjuntos<elem>::unionconjunto(const
Conjuntos &otroconjunto)
100   {
101    //Agrega los elementos de otroconjunto que no estan en el original
102    NodoC*aux = otroconjunto.conj;
103    while(aux!=NULL)
104           agregardato(aux->elemento);
105   }
106   template <typename elem> void Conjuntos<elem>::interseccion(const
Conjuntos &otroconjunto)
107   {
108    //Agrega los elementos del 2do conjunto al primero, y borra los que
estan en los 2
109    NodoC*aux=otroconjunto.conj;
110    while (aux!=NULL)
111    {
112          if(!pertenece(aux->elemento))
113          {
114               eliminar(aux->elemento);
115               aux=aux->sig;
116          }
117    }
118   }
119   template <typename elem> unsigned int Conjuntos<elem>::tamanio()const
120   {
121    //Devuelve el cardinal del conjunto
122    return largo;
123   }
124   template <typename elem> void Conjuntos<elem>::vaciar()
125   {
126
127    //elimina toda la lista
128    NodoC * AEliminar=NULL;
129    while (!esvacio())
```

```
130   {
131          AEliminar=conj;
132          conj=conj->sig;
133         delete AEliminar;
134   }
135  }
136  template <typename elem> const Conjuntos<elem>::devuelvevalor(const
int pos)const
137  {
138   int i=0;
139   NodoC*aux=conj;
140   while ((i<largo)&&(i<pos))
141   {
142         i++;
143         aux=aux->sig;
144   }
145   if ((i==pos)&&(i<largo))
146         return aux->elemento;
147   else
148         return false;
149  }
150
151  template <typename elem> void Conjuntos<elem>::resetactual()
152  {
153   actual=conj;
154  }
155  template <typename elem> bool Conjuntos<elem>::finactual()
156  {
157   return (actual!=NULL);
158  }
159  template <typename elem> const elem Conjuntos<elem>::devuelveactual()
160  {
161   int aux=actual->elemento;
162   actual=actual->sig;
163   return aux;
164  }
165
166  template class Conjuntos <int>;
167  template class Conjuntos <float>;
168  template class Conjuntos <double>;
```

1.2.4. Dejado intencionalmente en blanco

1.2.5.

Métodos públicos

- ***Iniconj***: Inicializa el puntero del conjunto en NULL. Su complejidad pertenece a O(1).
- ***Agregar*** : Se le da un elemento templado y lo agrega siempre al principio si dicho elemento no pertenece. Su complejidad pertenece a O(n) siendo n la cantidad de nodos.

- ***Esvacio***: función booleana que devuelve true si no hay ningun elemento en el conjunto.Su complejidad pertenece a O(1).
- ***Pertenece***: se le pasa un elemento y pregunta si está en el conjunto, si se encuentra en el devuelve true. Su complejidad pertenece a O(n) siendo n la cantidad de nodos del conjunto.
- ***Tamanio***: devuelve el cardinal del conjunto. Su complejidad pertenece a O(1).
- ***Eliminar***: se le pasa un elemento templado y si se encuentra en el conjunto lo elimina. su complejidad pertenece a O(n) siendo n la cantidad de nodos del conjunto.
- ***Unionconjunto***: dado dos conjuntos crea uno nuevo con todos los elementos de ambos conjuntos sin repetir. Su complejidad pertenece O(n*m) siendo n la cantidad de nodos del primer conjunto y la cantidad de nodos del segundo conjunto.
- ***Interseccion*** : dado dos conjuntos crea uno nuevo con los elementos que se encuentren en ambos conjuntos. Su complejidad pertenece O(n*m) siendo n la cantidad de nodos del primer conjunto y la cantidad de nodos del segundo conjunto.

Métodos privados

Estructura privada del conjunto:

```
private:
20      struct NodoC
21      {
22              elem elemento;
23              NodoC * sig;
24      };
25      NodoC * conj;
26      NodoC * actual;
27      int largo;
28      void vaciar();
29  };
```

La estructura privada del conjunto está conformada por un nodo que contiene un elemento templado y un puntero que apunta al siguiente nodo para vincularlos.

vaciar: se encarga de liberar la memoria al terminar la ejecución , eliminando todo el conjunto y dejando el puntero en NULL. Su complejidad pertenece a O(n) siendo n la cantidad de nodos del conjunto.

2.2. Resolucion del problema

2.2.1

$$\sum_{i=0}^{n/2} \quad c_0 = c_0 * \frac{n}{2} + c_0$$

El algoritmo va a pertenecer a una complejidad O$\left(\frac{n}{2}\right)$siendo n la cantidad de segmentos que hay en la lista, y c_0 el tiempo que tarda en hacerse cada sumatoria.

2.2.2. Codificacion en c++

```
26    int j=C.tamanio()/2;
27    for(int i=0;i<j;i++){
28          int aux=C.devuelvevalor(0);
29          C.eliminar(aux);
```

Como el conjunto siempre agrega ordenado de menor a mayor solamente con hacer una variable que se vuelva el cardinal/2, se hace un for que recorra desde el principio hasta dicha variable borrando a medida que avanza.

Problema 3

Implementar en c++ el TDA Heap.

TDA Heap

3. Este TDA está conformado por un arreglo de tamaño n(dado en la especificación), y una variable ultpos, que toma el valor del índice en la última posición en la que hay datos del arreglo +1.

3.1. Especificación algebraica

3.1.1.

Class Heap [Elem]
Imports Boolean
Basic constructors Inicheap, AgregarDat
Effective
Type Heap
Operations
Inicheap: -> Heap;
AgregarDat: Heap * Elem -> Heap;
Eliminar: Heap(h) -> Heap;
pre (!ArrVacio(h));
VerPrimero: Heap(h) -> Elem
pre (ArrVacio(h));
ArrVacio: Heap -> boolean;
Axioms
valor, valor2: Elem; h:Heap;
AgregarDat(inicheap(),valor)=h;
(valor>valor2) AgregarDat(AgregarDat(h,valor),valor2)=Agregar(h,valor2);
(valor<valor2) AgregarDat(AgregarDat(h,valor),valor2)=h;
Eliminar(AgregarDat(Inicheap(),valor))=Inicheap();
Eliminar(AgregarDat(h,valor))=AgregarDat(Eliminar(h,valor));
ArrVacio(Inicheap())=True;
ArrVacio(AgregarDat(h,valor))=False;
VerPrimero(AgregarDat(Inicheap(),valor))=valor;
VerPrimero(AgregarDat(h,valor))=verprimero(h);
End-class

3.1.2. Clasificación de las operaciones

Constructoras:

- Inicheap
- AgregarDat

Observadora:

- VerPrimero

- ArrVacio

Modificadoras:

- Eliminar

3.2. Clase Heap en c++

3.2.1.

```
#ifndef HEAPSORT_H
#define HEAPSORT_H


template <typename Telem> class Heapsort
{
  public:
        Heapsort();
        virtual ~Heapsort();
        bool agregarDat(const Telem&Elem);
        bool eliminar();
        const Telem&verPrimero()const;
        bool arrVacio()const;
  private:
        void organizar();
        void ordenaheap(int i);
        Telem arreglo[20000]={};
        int ultpos;
};

#endif // HEAPSORT_H
```

3.2.2.

Se implementan dos variables privadas, las cuales guardan las coordenadas del punto central del punto, y el radio del mismo.

3.2.3.

```
#include "Heapsort.h"

template <typename Telem>
Heapsort<Telem>::Heapsort()
{
  ultpos=0;
}

template <typename Telem>
Heapsort<Telem>::~Heapsort()
{

}
template <typename Telem>
```

```
15  bool Heapsort<Telem>::agregarDat(const Telem&Elem)
16  {
17    arreglo[ultpos]=Elem;
18      organizar();
19    ultpos++;
20          return true;
21  }
22  template <typename Telem>
23  bool Heapsort<Telem>::eliminar()
24  {
25    if (!arrVacio())
26    {
27          arreglo[0]= arreglo[ultpos];
28          arreglo[ultpos]=0;
29          ultpos--;
30          organizar();
31          return true;
32    }
33    else
34          return false;
35
36  }
37  template <typename Telem>
38  const Telem&Heapsort<Telem>::verPrimero()const
39  {
40    if(!arrVacio())
41          return arreglo[0];
42  }
43  template <typename Telem>
44  bool Heapsort<Telem>::arrVacio()const
45  {
46    if (arreglo[0]==0)
47          return true;
48    else
49          return false;
50  }
51  template <typename Telem>
52  void Heapsort<Telem>::ordenaheap(int i)
53  {
54    int mayor=i;
55    int izq=2*i+1;
56    int der=2*i+2;
57    if ((izq < ultpos) && (arreglo[izq] > arreglo[mayor]))
58          mayor=izq;
59    if ((der < ultpos) && (arreglo[der] > arreglo[mayor]))
60          mayor=der;
61    if (mayor!=i)
62    {
63          Telem aux;
64          aux=arreglo[i];
65          arreglo[i]=arreglo[mayor];
66          arreglo[mayor]=aux;
67          ordenaheap(i);
```

```
68    }
69  }
70  template <typename Telem>
71  void Heapsort<Telem>::organizar()
72  {
73    if (ultpos>1)
74          for(int i=ultpos/2-1;i>=0;i--)
75              ordenaheap(i);
76  }
77
78  template class Heapsort <int>;
79  template class Heapsort <float>;
80  template class Heapsort <double>;
```

3.3. Tiempos de ejecución medidos en segundos

$\frac{Cant\ de\ datos}{Metodo}$	10000	12000	20000
Heapsort	6,14	6,4	7,41
Mergesort	5,13	5,35	6,31
Quicksort	5,46	5,7	6,4

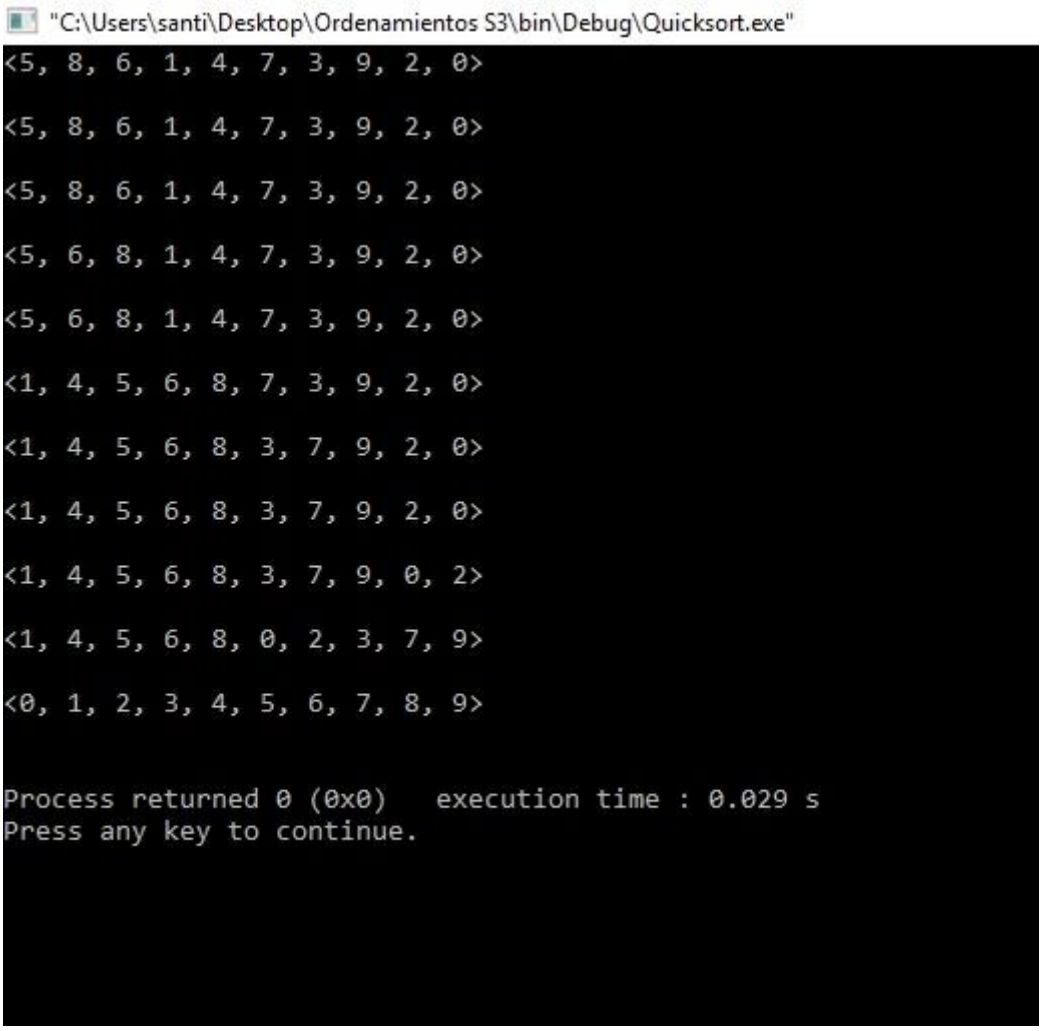
Mergesort

Este algoritmo divide un arreglo dado en 2 partes iguales, y luego se llama recursivamente para volver a dividirse, hasta que ya no se pueda dividir más, y a partir de ahí, va armando cada arreglo auxiliar ordenado, para finalmente copiar todos los valores al arreglo original, en el orden correcto.
Dicho algoritmo va a tener una complejidad de O(n log n) siendo n la cantidad de elementos del arreglo.

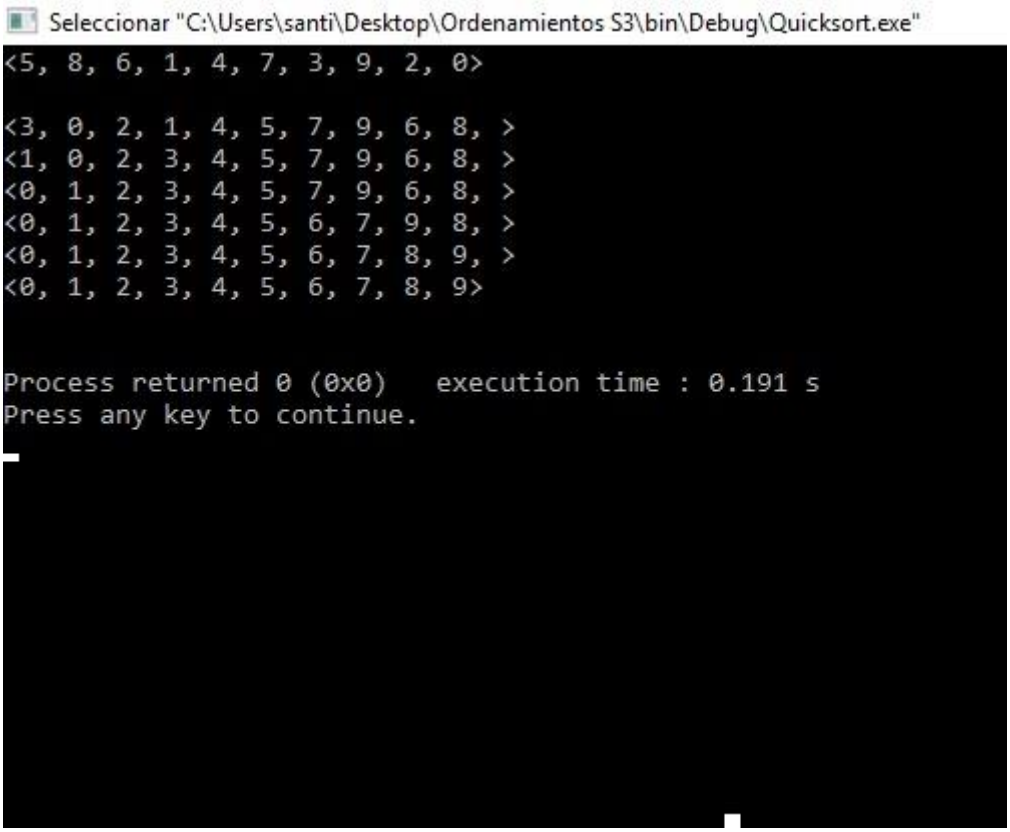
Quicksort

El algoritmo funciona eligiendo un pivote, que será un elemento del arreglo que queremos ordenar y del lado izquierdo del pivote dejaremos los elementos menores a el y del derecho los mayores y iguales.
después de eso el pivote se encuentra ordenado y el arreglo queda dividido en dos subarreglos de izquierda y derecha, este proceso se hará hasta que el arreglo se encuentre ordenado.
Dicho algoritmo va a tener una complejidad de O(n log n) siendo n la cantidad de elementos del arreglo.

3.4



(seguimiento del algoritmo mergesort)



(seguimiento del algoritmo quicksort)

Capítulo 3

TDA Círculo

1. El TDA Círculo está compuesto por un punto que contiene las coordenadas del centro del círculo, y un radio r, que contiene el largo del radio del círculo.

1.1 Especificación algebraica

1.1.1.

Class Circulo
Inherits Figura2d
Imports Punto, Real, cmath
Basic constructors *crearCirculo*
Effective
Type Circulo
Operations
crearCirculo: Punto * Real(radio) -> Circulo;
pre (radio>0);
trasladarOrigen: Círculo * Real * Real -> Circulo;
modificarRadio: Círculo * Real(radio)-> Circulo;
pre (radio>0);
devolverOrigen: Circulo -> Punto;
devolverRadio: Círculo -> Real;
areaFigura: Círculo -> Real;
Axioms
p1: punto; x, y, radio1, radio2, pi: real;

trasladarOrigen(crearCirculo(p1,radio1),x,y)=crearCirculo(crearpunto(x,y),radio1);
modificarRadio(crearCirculo(p1,radio1)radio2)=crearCirculo(p1,radio2);
devolverOrigen(crearCirculo(p1,radio1))= p1;
devolverRadio(crearCirculo(p1,radio1))= radio1;
areaFigura(crearCirculo(p1,radio1))= pow(radio1,2) * pi;
End-class

1.1.2. Clasificación de las operaciones

Constructoras:
- crearCirculo

Observadora:
- devolverOrigen
- devolverRadio
- areaFigura

Modificadoras:
- trasladarOrigen
- modificarRadio

1.2. Clase Circulo en c++

1.2.1.

```
1  #ifndef CIRCULO_H
2  #define CIRCULO_H
3  #include <cmath>
4  #include <Punto.h>
5  #include <Figura2d.h>
6
7  class Circulo :public Figura2d
8  {
9  public:
10   Circulo(const Punto&centro,const float&radio):Figura2d("Circulo")
11   {
12         this->centro=centro;
13         this->radio=radio;
14   };
15   void trasladarorigen(float x, float y);
16   void modificarradio(float radio);
17   const Punto & devolverorigen()const;
18   const devolverradio()const;
19   float areaFigura()const;
20
21 private:
22   Punto centro;
23   float radio;
24 };
25 #endif // CIRCULOS_H
```

1.2.2.

Se implementan dos variables privadas, las cuales guardan las coordenadas del punto central del punto, y el radio del mismo.

1.2.3.

```
1  #include "Circulo.h"
2  #include <cmath>
3  #include <Punto.h>
4
5
6  void Circulo::trasladarorigen(float x, float y)
7  {
8    //traslada el centro del origen
9    centro.trasladar(x,y);
10 }
11
12 void Circulo::modificarradio(float radio)
13 {
14   //modifica el radio del circulo
```

```
15    this->radio=radio;
16  }
17
18  const Punto & Circulo::devolverorigen()const
19  {
20    //devuelve el centro del circulo
21    return centro;
22  }
23
24  const Circulo::devolverradio()const
25  {
26    //devuelve el radio
27    return radio;
28  }
29
30  float Circulo::areaFigura()const
31  {
32    //calcula el area de un circulo
33    return 3.14 * (pow( radio,2.0));
34  }
```

1.2.4. Dejado intencionalmente en blanco

1.2.5.

Métodos Públicos

- ***crearCirculo***: Se le pasa un punto y un real que cumplira la funcion de ser el radio, con esos datos se creará un círculo.Su complejidad pertenece a O(1).
- ***trasladarOrigen***: Se le pasa dos valores reales y se le suma a el punto de origen. Su complejidad pertenece a O(1).
- ***modificarRadio***: Se pide un valor real y este pasará a ser el nuevo radio del círculo.Su complejidad pertenece a O(1).
- ***devolverOrigen***: Devuelve el Punto con el que se creó el círculo. Su complejidad pertenece a O(1).
- ***areaFigura***: Devuelve el área del círculo desde el que se la llama. Su complejidad pertenece a O(1).

Métodos privados

Estructura de la clase círculo:

```
18    private:
19          Punto centro;
20          float radio;
21  };
```

La estructura del círculo está conformada por un punto y el radio del mismo.

TDA Triángulo

2. El TDA Triángulo está compuesto por 3 puntos, distintos entre sí, que forman un triángulo cuando se unen de a pares.

2.1. Especificación algebraica

2.1.1.

Class Triangulo
Inherits Figura2d
Imports Punto,Nat,Real
Basic constructors crearTriangulo
Effective
Type Triangulo
Operations

crearTriangulo: Punto(p1) * Punto(p2) * Punto(p3) -> Triangulo; constructora
pre (coordx(p1)==coordx(p2)) && (coordy(p1)==coordy(p3));
devolverPunto: Triangulo * Nat-> Punto; observadora
pre ((k>=1) && (k<=3));
modificarPunto: Triangulo * Nat* Punto -> Triangulo; modificadora
pre ((k>=2) && (k<=3));
areaFigura: Triangulo -> Real; observadora
_=_: Triangulo * Triangulo -> Triangulo; modificadora

Axioms

p1, p2, p3, p4, p5, p6: Punto; k , b , h: Nat;

(k==1) devolverPunto(crearTriangulo(p1,p2,p3),k)=p1;
(k==2) devolverPunto(crearTriangulo(p1,p2,p3),k)=p2;
(k==3) devolverPunto(crearTriangulo(p1,p2,p3),k)=p3;
((k==2) && (coordx(p1)==coordx(p4)) && (coordy(p1)!=coordy(p4)))
modificarPunto(crearTriangulo(p1,p2,p3),k,p4)=crearTriangulo(p1,p4,p3);
((k==2) && (coordx(p1)!=coordx(p4)) && (coordy(p1)!=coordy(p4)))
modificarPunto(crearTriangulo(p1,p2,p3),k,p4)=crearTriangulo(p1,p2,p4);
areaTriangulo(crearTriangulo(p1,p2,p3))= (distancia(p1,p2)*distancia(p1,p3))/2;
_=_(crearTriangulo(p1,p2,p3),crearTriangulo(p4,p5,p6))= crearTriangulo(p4,p5,p6);

End-class

2.1.2. Clasificación de las operaciones

Constructoras:

- crearTriangulo

Observadora:

- devolverPunto
- areaFigura

Modificadoras:

- modificarPunto
- _=_

2.2. Clase Triangulo en c++

2.2.1.

```
1  #ifndef TRIANGULO_H
 2  #define TRIANGULO_H
 3  #include "Punto.h"
 4  #include "Figura2d.h"
 5  #include <cmath>
 6
 7  class Triangulo : public Figura2d
 8  {
 9  public:
10      Triangulo() : Figura2d("Triangulo") {};
11      Triangulo(const Punto&a, const Punto&b, const
Punto&c):Figura2d("Triangulo")
12    {
13          if ((a!=b) && (b!=c) && (c!=a))
14          {
15                this->a=a;
16                this->b=b;
17                this->c=c;
18          }
19    };
20    ~Triangulo(){};
21    void modificarPunto(const int&x,const Punto&a);
22    float areaFigura()const;
23    const Punto & devolverPunto(int x)const;
24      Triangulo & operator=(const Triangulo&Tri);
25  private:
26    Punto a;
27    Punto b;
28    Punto c;
29  };
30
31  #endif // TRIANGULO_H
```

2.2.2.

Se implementan 3 variables privadas, las cuales guardan las coordenadas de cada punto del triángulo.

2.2.3.

```
 1  #include "Triangulo.h"
 2
 3  void Triangulo::modificarPunto(const int&x,const Punto&a)
 4  {
 5    //Modifica el punto x(1,2 o 3) como la suma entre el original, y el
nuevo
 6    switch (x)
 7    {
```

```
 8     case 1:
 9           this->a + a;
10           break;
11     case 2:
12           this->b + b;
13           break;
14     case 3:
15           this->c + c;
16           break;
17     default:
18           break;
19     }
20   }
21   float Triangulo::areaFigura()const
22   {
23     //calcula el area del triangulo dado, a partir de la formula de Heron
24     float a=this->a.distancia(this->b);
25     float b=this->b.distancia(this->c);
26     float c=this->c.distancia(this->a);
27     float s=(a+b+c)/2;
28     float x=sqrt(s*(s-a)*(s-b)*(s-c));
29     return x;
30   }
31   const Punto & Triangulo::devolverPunto(int x)const
32   {
33     //Devuelve uno de los 3 puntos del triangulo, segun la eleccion del
usuario
34     switch (x)
35     {
36     case 1:
37           return this->a;
38           break;
39     case 2:
40           return this->b;
41           break;
42     case 3:
43           return this->c;
44           break;
45     default:
46           break;
47     }
48   }
49   Triangulo & Triangulo::operator=(const Triangulo&Tri)
50   {
51     this->a=Tri.devolverPunto(1);
52     this->b=Tri.devolverPunto(2);
53     this->c=Tri.devolverPunto(3);
54   }
```

2.2.4. Dejado intencionalmente en blanco

2.2.5.

Métodos públicos

- ***crearTriangulo***: Se le pasan tres puntos y son asignados al triángulo en caso de ser distintos. Su complejidad pertenece a O(1).
- ***modificarpunto***: Se le pasa un punto y un natural, y modifica el punto que indique el valor del natural. Su complejidad pertenece a O(1).
- ***areaFigura***: Calcula el area del triangulo desde el que se la llama. Su complejidad pertenece a O(1).
- ***devolverPunto***: Se le pasa un natural y devuelve el punto que pida ese natural. Su complejidad pertenece a O(1).
- _=_: Se le pasa un triangulo y modifica el primero con los valores del segundo. Su complejidad pertenece a O(1).

Métodos privados

Estructura privada del Triángulo:

```
15  private:
16    Punto a;
17    Punto b;
18    Punto c;
19  };
```

La estructura del triángulo está conformada por tres puntos.

TDA Cuadrado

3. El TDA Cuadrado está compuesto por 4 puntos, distintos entre sí, que forman un cuadrado cuando se unen de a pares.

.

3.1. Especificación algebraica

3.1.1.

Class Cuadrado
Inherits Figura2d
Imports Punto,Triangulo,Real,Nat
Basic Constructors CrearCuadrado
Effective
Type Cuadrado
Operations

crearCuadrado: Punto(p1) * Punto(p2) * Punto(p3) * Punto(p4) -> Cuadrado;
pre (coordy(p1)==coordy(p2)) and (coordx(p1)==coordx(p3)) and
(coordx(p2)==coordx(p4)) and (coordy(p3)==coordy(p4));

trasladarCuadrado: Cuadrado * Punto -> Cuadrado;
areaFigura: Cuadrado -> Real;
devolverPunto: Cuadrado * Nat(k) -> Punto;
pre(k>=1 && k<=4)

Axioms

p1, p2, p3, p4, p5: Punto; k: Nat

TrasladarCuadrado(CrearCuadrado(p1,p2,p3,p4),p5)=CrearCuadrado(p1+p5,p2+p5, p3+p5,p4+p5),p5);
AreaCuadrado(CrearCuadrado(p1,p2,p3,p4))=
AreaTriangulo(CrearTriangulo(p1,p2,p3)) +
AreaTriangulo(CrearTriangulo(p2,p3,p4));
(k==1) DevolverPunto(CrearCuadrado(p1,p2,p3,p4),k)=p1;
(k==2) DevolverPunto(CrearCuadrado(p1,p2,p3,p4),k)=p2;
(k==3) DevolverPunto(CrearCuadrado(p1,p2,p3,p4),k)=p3;
(k==4) DevolverPunto(CrearCuadrado(p1,p2,p3,p4),k)=p4;
End-class

3.1.2. Clasificación de las operaciones

Constructoras:

- crearCuadrado

Observadora:

- devolverPunto
- areaFigura

Modificadoras:

- trasladarCuadrado

3.2. Clase Cuadrado en c++

3.2.1.

```
1  #ifndef CUADRADO_H
 2  #define CUADRADO_H
 3  #include "Triangulo.h"
 4
 5  class Cuadrado : public Figura2d
 6  {
 7  public:
 8    Cuadrado(const Punto&a,const Punto&b,const Punto&c,const
Punto&d):Figura2d("Cuadrado")
 9    {
10          //  if ((a!=b) && (a!=c) && (a!=d) && (b!=c) && (b!=d) &&
(c!=d))
11          {
12                this->a=a;
13                this->b=b;
14                this->c=c;
15                this->d=d;
16          }
17    };
18       Cuadrado():Figura2d("Cuadrado") {};
19    ~Cuadrado() {};
20    void trasladarCuadrado(const Punto&a);
21    float areaFigura()const;
22    const Punto & devolverPunto(int x)const;
23    Cuadrado & operator=(const Cuadrado&Cua);
24  private:
25    Punto a;
26    Punto b;
27    Punto c;
28    Punto d;
29  };
30
31  #endif // CUADRADO_H
```

3.2.2. Implementamos 3 variables privadas, las cuales guardan las coordenadas de cada punto del cuadrado.

3.2.3.

```
 1  #include "Cuadrado.h"
 2
 3  void Cuadrado::trasladarCuadrado(const Punto&a)
 4  {
 5
 6    //Suma el punto dado, con los 4 originales del objeto, para
trasladarlo a ese punto
 7    this->a+a;
 8    this->b+a;
 9    this->c+a;
10    this->d+a;
```

```
11  }
12
13  float Cuadrado::areaFigura()const
14  {
15    //Calcula el area del cuadrado utilizando triangulacion de
polinomios,  creando 2 triangulos y calculando sus respectivas areas
16      Triangulo* t1= new Triangulo(this->a,this->b,this->c);
17      Triangulo* t2= new Triangulo(this->b,this->c,this->d);
18    return (t1->areaFigura() + t2->areaFigura());
19  }
20  const Punto & Cuadrado::devolverPunto(int x)const
21  {
22    //Devuelve uno de los 3 puntos del triangulo, segun la eleccion del
usuario
23    switch (x)
24    {
25    case 1:
26          return this->a;
27          break;
28    case 2:
29          return this->b;
30          break;
31    case 3:
32          return this->c;
33          break;
34    case 4:
35          return this->d;
36          break;
37    default:
38          break;
39    }
40  }
41  Cuadrado & Cuadrado::operator=(const Cuadrado&Cua)
42  {
43      this->a=Cua.devolverPunto(1);
44      this->b=Cua.devolverPunto(2);
45      this->c=Cua.devolverPunto(3);
46      this->d=Cua.devolverPunto(4);
47  }
```

3.2.4. Dejado intencionalmente en blanco

3.2.5.

Métodos públicos

- ***crearCuadrado***: Se le pasan cuatro puntos y si son distintos entre si los asigna al objeto cuadrado. Su complejidad pertenece a O(1).
- ***trasladarCuadrado***: Se le pasa un punto y se le suma ese punto a cada uno de los cuatro del objeto cuadrado. Su complejidad pertenece a O(1).

- ***areaFigura***: Devuelve el área del cuadrado utilizando la clase Triángulo. Su complejidad pertenece a O(1).
- ***devolverPunto***: Se le pasa un natural entre 1 y 4 y devuelve el punto que esté en ese orden. Su complejidad pertenece a O(1).

Métodos privados

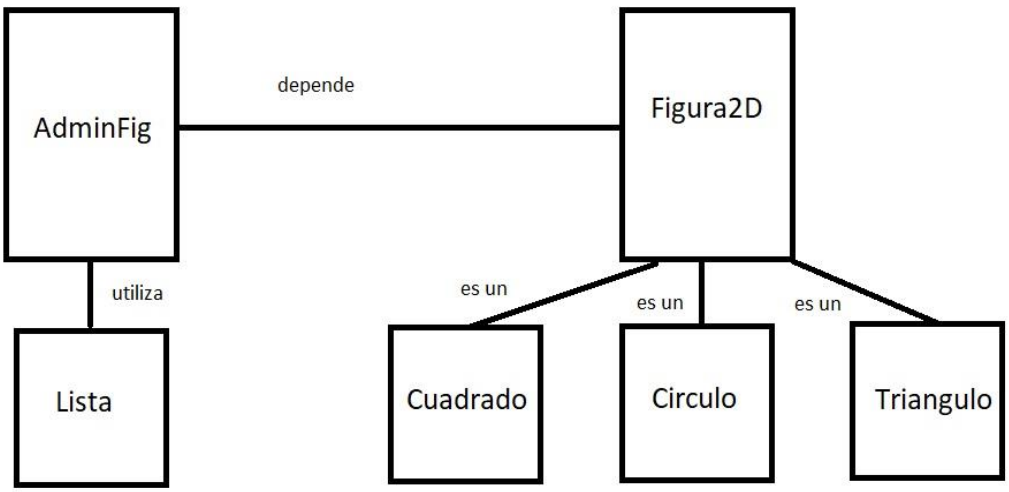
Estructura privada del cuadrado:

```
15  private:
16    Punto a;
17    Punto b;
18    Punto c;
19    Punto d;
20  };
```

La estructura privada del cuadrado está conformada por cuatro puntos.

3.3. Resolucion del problema

3.3.1.



3.3.2. Codificacion en c++

```
39    //resolucion del cohete
40    Punto r1(6,0);
41    Punto r2(0,12);
42    Punto m3(2,0);
```

```
43    Punto m2(1,8);
44    Punto m1(8,16);
45    Punto a2(16,16);
46    Punto b2(16,0);
47    Punto c2(0,16);
48    Punto d2(0,0);
49    Punto a1(4,4);
50    Punto b1(4,0);
51    Punto c1(0,4);
52    Punto d1(0,0);
53    Figura2d* Cchico= new Cuadrado(a1,b1,c1,d1);
54    Figura2d* Cgrande= new Cuadrado(a2,b2,c2,d2);
55    Figura2d* Circ= new Circulo(a1,6);
56    Figura2d* TriIsoG= new Triangulo(d1,b2,m1);
57    Figura2d* TriIsoC= new Triangulo(d1,m3,m2);
58    Figura2d* TriRec1= new Triangulo(d1,r1,r2);
59    float areacchico= Cchico->areaFigura();
60    float areacircu= Circ->areaFigura();
61    AdminFig admin;
62    admin.agregarFig(TriIsoG);
63    admin.agregarFig(Circ);
64    admin.agregarFig(TriIsoC);
65    admin.agregarFig(TriIsoC);
66    admin.agregarFig(TriIsoC);
67    admin.agregarFig(TriIsoC);
68    admin.agregarFig(Cchico);
69    admin.agregarFig(Cchico);
70    admin.agregarFig(Cchico);
71    admin.agregarFig(Cgrande);
72    admin.agregarFig(Cgrande);
73    admin.agregarFig(TriRec1);
74    admin.agregarFig(TriRec1);
75    Figura2d* fnueva;
76    int k= admin.devuelvetam();
77    int resultado=0;
78    for(int i=0; i<k; i++)
79    {
80          fnueva=admin.obtenerFig(i);
81          float area= fnueva->areaFigura();
82          if ((areacchico!=area) && (areacircu!=area))
83              resultado+=area;
84          else
85              resultado-=area;
86    }
87    cout << "el area del cohete es de : " << resultado << endl;
88
89    return 0;
90  }
```

3.3.3. Complejidad temporal

La resolución del problema planteado pertenece a una complejidad de O(n) ya que se debe recorrer toda la lista para sumar o restar cada área.

Conclusiones

En conclusión aprendimos a crear TDAs en NEREUS, un lenguaje formal de pseudocódigo completamente nuevo para nosotros, el cual nos trajo problemas a la hora de especificar cada función, pero nos ayudó a pasar de pseudocódigo a c++ con mayor facilidad.

Podemos decir que aprendimos a separar el código en partes reutilizables para otros programas (clases), y pudimos ver un lenguaje más cercano al uso profesional, lo cual nos permitió aprender más del mismo a través de páginas de internet con muchísimo contenido, tales como www.cplusplus.com

Apéndice

Semana 1:

TDA AdminFig----------TPE aydaI\\semana1\\include\\AdminFig.h
TPE aydaI\\semana1\\src\\AdminFig.cpp

TDA Círculo ----------TPE aydaI\\semana1\\include\\Circulo.h
TPE aydaI\\semana1\\src\\Circulo.cpp

TDA Cuadrado----------TPE aydaI\\semana1\\include\\Cuadrado.h
TPE aydaI\\semana1\\src\\Cuadrado.cpp

TDA Figura2d----------TPE aydaI\\semana1\\include\\Figura2d.h
TPE aydaI\\semana1\\src\\Figura2d.cpp

TDA Lista----------TPE aydaI\\semana1\\include\\Lista.h
TPE aydaI\\semana1\\src\\Lista.cpp

TDA Punto----------TPE aydaI\\semana1\\include\\Punto.h
TPE aydaI\\semana1\\src\\Punto.cpp

TDA Segmento----------TPE aydaI\\semana1\\include\\Segmento.h
TPE aydaI\\semana1\\src\\Segmento.cpp

TDA Triángulo----------TPE aydaI\\semana1\\include\\Triangulo.h
TPE aydaI\\semana1\\src\\Triangulo.cpp

Semana 2:

TDA Conjunto----------TPE aydaI\\semana2\\include\\Conjunto.h
TPE aydaI\\semana2\\src\\Conjunto.cpp

TDA Fila----------TPE aydaI\\semana2\\include\\Fila.h
TPE aydaI\\semana2\\src\\Fila.cpp

TDA Lista----------TPE aydaI\\semana2\\include\\Lista.h
TPE aydaI\\semana2\\src\\Lista.cpp

TDA Pila----------TPE aydaI\\semana2\\include\\Pila.h
TPE aydaI\\semana2\\src\\Pila.cpp

Ordenamientos semana 3:

TDA Heap----------TPE aydaI\\semana3\\include\\Heap.h