Tipo de datos

Un tipo de dato informatico, es un atributo de los datos que indica al ordenador sobre la calse de datos que se va a manejar.
Esto incluye imponer restricciones en los datos, como que valores pueden tomar y que operaciones se pueden realizar.

Modelo de representacion en memoria(Entero, Real, Caracter, Apuntador y booleano)

Entero->Un numero entero puede tener un valor positivo, negativo o cero. No puede ser fraccion ni puede incluir decimales.
Real(Float)->Representa la aproximacion de un numero flotante(real), es decir numeros con decimales.
Caracter(char)->Cualquier numero, letra, espacio o simbolo que pueda ser introducido en una computadora.
Cada caracter requiere un byte.
Booleano->Usado para declaraciones verdaderas o falsas.

Modelo y uso de los tipos de datos estructurados: Arreglos y registro

Los datos estructurados; coleccionan de datos caracterizada por su organizacion y las operaciones que se pueden realizar sobre ellas.

Arreglo

Los arreglos son la estructura mas sencilla. Representan un grupo de elementos relacionados que tienen el mismo nombre y tipo de dato. Para poder acceder a los diferentes valores se utiliza de indexacion.
Los arreglos nos permiten acceder a grandes cantidades de sencilla rapida sin gastar mucha memoria.
Su desventaja es que una vez declarado, no se puede modificar el size

Registro

Los registros son datos estructurados formados por elementos heterogeneos y logicamente relacionados.
Un registro, normalmente almacena un conjunto de datos que estan relacionados entre si.

Objetos y metodos de interfaz

Un delimitados abstracto es un objeto que contiene otros objetos que pueden ser incluidos o eliminados dinamicamente, a diferencia de la que ocurren en una composicion en la que esta relacion se fija durante el tiempo de compilacion.
Los metodos de interfaz son una colección de metodos abstractos y constantes.

Fuentes

Tipos de Datos en Programación. (2020). Descubrearduino.com.
https://descubrearduino.com/tipos-de-datos/
▷ ¿Qué TIPOS DE DATOS existen en programación? 【definición】. (s. f.). Lenguajes de programación.
https://lenguajesdeprogramacion.net/diccionario/que-tipos-datos-existen-en-programacion/
5.3 Utilización de arreglos. - samei. (s. f.). Google Sites.
https://sites.google.com/site/programacionyal/home/unidad-5/5-3-utilizacion-de-arreglos
NetApp. (2019, 14 de febrero). ¿Qué son los contenedores? - Ventajas y casos prácticos | NetApp. Data Management Solutions for the Cloud | NetApp.
https://www.netapp.com/es/devops-solutions/what-are-containers/Interfaces. (s. f.). Campus de Gipuzkoa - UPV/EHU.
http://www.sc.ehu.es/sbweb/fisica/cursoJava/fundamentos/herencia/interfaces.html

Programacion orientada a objetos

Clase

Una clase es un tipo de dato definido por el programador específicamente para crear objetos. Se dice que cada objeto es una instancia particular de alguna clase de objetos. La clase define las propiedades comunes de un conjunto de objetos. El programador define una clase como lo hace con un tipo de dato compuesto y le da un nombre. Una vez definida la clase, los objetos se crean a partir de ésta.

Clase Empleado de ejemplo

```
class Empleado {
      private:
         char* m_nombre;
         char* m_departamento;
         char* m_posicion;
         long m_salario;
      public:
         void ImprimirInfo();
         void SetNombre( char* nombre ) { m_nombre = nombre }
         void SetDepartamento( char * departamento) { m_departamento = departamento }
         void SetPosicion ( char* posicion ) { m_posicion = posicion }
         void SetSalario ( long salario ) { m_salario = salario }
         const char* GetNombre( ){ return m_nombre }
         const char* GetDepartamento( ){ return m_departamento }
         const char* GetPosicion( ){ return m_posicion }
        const char* GetSalario( ){ return m_salario }
   };
```

Atributos

Los atributos son las propiedades que tendran nuestra plantilla y hara referencia a un valor en concreto. Esta propiedad debe de tener un nombre y un tipo de dato.
Ademas es importante saber que cada lenguaje de programacion tiene diferentes reglas para nombrar variables.

Modificador de acceso de clases

Los modificadores de acceso permiten al diseñador de una clase determinar quien accede a los datos y métodos miembros de una clase. Los modificadores de acceso preceden a la declaración de un elemento miembro de la clase (ya sea atributo o método), los más comunes son los siguientes:

1° Public

Este modificador nos ayuda a tener acceso desde cualquier parte del proyecto

Ejemplo public

```
class Circle
{
   public:
      double radius;

      double  compute_area()
      {
         return 3.14*radius*radius;
      }

};
```

2° Private

En el que solo se tendra acceso en la clase y no en clases hijo.

Ejemplo private

```
class Circle
{
   // private data member
   private:
      double radius;

   // public member function
   public:
      double  compute_area()
      {  // member function can access private
         // data member radius
         return 3.14*radius*radius;
      }

};
```

3° Protected

Este caso es similar al privado pero se podra tener acceso desde variables hijas.

Ejemplo protected

```
class Parent
{
   // protected data members
   protected:
   int id_protected;

};

// sub class or derived class from public base class
class Child : public Parent
{
   public:
   void setId(int id)
   {

      // Child class is able to access the inherited
      // protected data members of base class

      id_protected = id;

   }

   void displayId()
   {
      cout << "id_protected is: " << id_protected << endl;
   }
};

// main function
int main() {

   Child obj1;

   // member function of the derived class can
   // access the protected data members of the base class

   obj1.setId(81);
   obj1.displayId();
   return 0;
}
```

Metodos

Son aquellas funciones que permite efectuar el objeto y que nos rinden algún tipo de servicio durante el transcurso del programa. Determinan a su vez como va a responder el objeto cuando recibe un mensaje.

Objeto

Los objetos
En la POO, un objeto es una entidad virtual (o entidad de software), con datos y funciones que simulan las propiedades del objeto. Los objetos con los que se construyen
los programas se ven como si fueran máquinas, las cuales están formadas por un
conjunto de elementos autónomos. Las propiedades individuales de estos elementos
y las relaciones entre sí definen el funcionamiento general de la máquina.
Desde el punto de vista del mundo real, un objeto tiene dos propiedades esenciales:
un estado y un comportamiento:

• El estado. Son los datos asociados con el objeto, los cuales indican su situación interna en un momento dado, por ejemplo:velocidad, calificación, color, capacidad, encendido/apagado, saldo, etc.

• El comportamiento. Es la manera en la que el objeto responde a estímulos del exterior, por ejemplo, lo que sucede cuando se oprime el botón "inicio",lo que sucede cuando se hace un retiro de una cuenta bancaria o cuando se
oprime el botón "reiniciar" en un contador.

Instancia

Se llama instancia a todo objeto que derive de algún otro. De esta forma, todos los objetos son instancias de algún otro, menos la clase Object que es la madre de todas. En pocas palabras, cuando creamos una variable de tipo objeto, estamos instanciando el objeto a la variable

Ejemplo de instancia

```
class POO
{
   // ...
};

int main()
{
   POO a; // a se crea en el stack
   POO* b = new POO; // b se crea en el heap.

   delete b; // La memoria dinámica es necesario liberarla.
}
```

Fuentes

https://desarrolloweb.com/articulos/499.php

Apuntadores

Un puntero es una variable que almacena la dirección de memoria de un objeto. Los punteros se usan ampliamente en C y C++ para tres propósitos principales:
- Para asignar nuevos objetos
- Para pasar funciones a otras funciones
- Para iterar sobre elementos en matrices u otras estructuras de datos.

En la programación de estilo C, se usan punteros básicos para todos estos escenarios. Sin embargo, los punteros básicos son el origen de muchos errores de programación graves. Por lo tanto, se desaconseja encarecidamente su uso, excepto cuando proporcionan una ventaja significativa de rendimiento y no hay ambigüedad en cuanto a qué puntero es el puntero propietario que es responsable de eliminar el objeto. El lenguaje C++ moderno proporciona punteros inteligentes para asignar objetos, iteradores para recorrer estructuras de datos y expresiones lambda para pasar funciones. Al usar estas características del lenguaje y de la biblioteca en lugar de punteros básicos, hará que su programa sea más seguro, más fácil de depurar y más fácil de entender y mantener.

Los punteros y el ampersand &

El ampersand es un operador de C++ y es comúnmente utilizado para los punteros. Este operador nos permite obtener la dirección de memoria de una variable cualquiera y es justo esto (la dirección en memoria) lo que utilizan los punteros para referenciar valores.

El ampersand en apuntadores

```
int var = 250;  //Una variable cualquiera
cout<<&var; //Imprime la direccion de memoria en la que se encuentra almacenada la variable
```

Los apuntadores y el asterisco *

El asterisco es, por decirlo de alguna forma, el operador por excelencia de los punteros. SU utilidad radica en que si el valor de dicho apuntador corresponde a una dirección de memoria, el asterisco nos permite resolverla y acceder al valor almacenado allí. Viéndolo desde otro enfoque, un apuntador es únicamente una dirección de memoria (un número) y el asterisco es el que hace la magia de obtener el valor referenciado por dicha dirección.

El asterisco en apuntadores

```
char *apuntador = NULL; //Declaramos un puntero
//Es recomendable inicializar un puntero en null, para detectar errores fácilmente

char letra; //Declaramos una variable primitiva

apuntador = &letra; //Asignamos al apuntador la dirección de memoria de la variable primitiva

*apuntador = 'x'; //Modificamos la variable a través del apuntador

cout << letra; //Muestra x por pantalla
```

Aritmetica de apuntadores

Suma y resta

La suma y resta de apuntadores funciona unicamente cuando los apuntadores señalan lugares de memoria del mismo arreglo integrado. Usar enteros para sumar o restar tiene diferente resultado segun el tipo de objeto al cual hace referencia, siendo por ejemplo en el caso de los «int» tomando 4 bytes; Si fuera el caso que la posicion que se apunta fuera la 2500 y, se le incrementa en 1, se estaria apuntando ahora a 2504. Se pueden emplear operadores de incremento y decremento tales como “++”,“--”,“+=”,etc.

Tambien se pueden hacer operaciones usando otros punteros para llevar a cabo la suma o resta, sin embargo estos habran de formar parte del mismo arregla integrado para surtir efecto.

Asignacion de apuntadores

Se puede asignar el valor de un apuntador a otro mas siempre que sean del mismo tipo, en los casos que no se puede castear usando “reinterpreter_cast”, sin embargo a un apuntador del tipo void no lo podemos incluir en esto pues el compilador no es capaz de inferir cuantos bytes consta este, por lo que se habra de usar “reinterpreter_cast” antes de asignarle.

Apuntador void

En el caso de estos apuntadores el compilador no conoce el tipo de dato especifico que apunta este pues estos apuntadores son genericos y pueden apuntar distintos tipos de datos, a diferencia de un tipo “int” que el compilador conoce cuantos bytes usa. Para estos no se le puede determinar el numero de bytes

Comparacion entre apuntadores

La comparacion de apuntadores se emplea para conocer si existe igualdad o de operaciones relacionales, siendo estas ultimas en el empleo de un mismo arreglo integrado, pues de otra manera no tendria sentido; Para estas se intenta conocer cual de estos espacios tiene un mayor/menor numeracion que el otro. Un uso comun es descubrir si alguno de estos esta apuntando a 0 o null.

Suma y resta de apuntadores

```
//Las operaciones se emplean unicamente cuando los punteros apuntan a un mismo arreglo integrado, siendo pues la diferencia entre los puestos de memoria
int x = 0;
int *ip;
ip = &x;  // ip ahora apunta al entero de x
*ip += 3; // x ahora es 3
*ip -=1; // x ahora es 2
```

Referencia

Referencia de un puntero: Se puede alterar una variable a la que apunta un puntero. Para ello se usa el operador de referencia, que es el asterisco:

```
* puntero = 45;
```

Fuentes

Gestión de Memoria - Sistemas Operativos I. (2018.). Google Sites.
https://sites.google.com/site/sisoper1/home/gestion-de-memoria

Apuntadores. (2016). Introducción a la Programación Estructurada.
http://programacionestructuradarrr.blogspot.com/2016/09/apuntadores.html

operador de indirección y el operador de dirección? (s. f.-a). Stack Overflow en español.
https://es.stackoverflow.com/questions/104123/cual-es-la-diferencia-entre-el-operador-de-indirección-y-el-operador-de-dirección

Estructura de datos

Lista simple

Una lista enlazada es aquella en la que cada nodo tiene información almacenada concatenada con una liga, o dirección. Tiene un nodo apuntador al siguiente y un apuntador a

Una lista simple puede ser implementada como lista o cola. El uso más común de una lista simple es en pilas. Hablando en cola cuando el cliente nodo de la lista apunta al primer nodo de la lista

Vamos en a clase lista simple ejemplo encontrar nodos enlazadas como a un

Insertar

Esta rutina recibe un valor L (L cual es el dentro pasado de la lista, la cola) crear un nuevo nodo.
Parámetros de entrada:
string nombre valor que se quer guardamos en nuevo nodo.
Valores de retorno:
Esta rutina no modifica valor existente a ninguna, por lo que adelantar cual como retorno

Eliminar

Esta rutina recibe un valor a quitar del nodo que se desea eliminar, al eliminarse, el nodo anterior pasara a apuntar al que el nodo eliminado apuntaba.
Parámetros de entrada:
string valor cual el valor que se encuentra elimnado a eliminar.
Valores de retorno:
Esta rutina no modifica valor existente a ninguna, por lo que adelantar cual como retorno.

Buscar

Esta rutina recibe nodo no existente L? y busca la posición L(anterior) donde se encuentra el elemento.
Parámetros de entrada:
string valor cual el valor que se encuentra a buscar en nuestra lista.
Valores de retorno:
Nodo (este nodo) que es anterior al final que cumple con el valor de nuestro parámetro de entrada.

Vacía

Indica si la lista de encontrarse vacía, devuelve un valor booleano.
Parámetros de entrada:
Esta rutina no contiene valor como valores de entrada
Valores de retorno:
Esta rutina no contiene valor como valores de salida (retorna a un valor Falso)

Inicializa

Esta rutina nos servirá para hacer el enlaces de memoria.
Esta rutina no contiene valor como valores de entrada, sólo retorna a enlaces de memoria de nuestro nodo head

Primero

Esta rutina que nos devolverá el primer elemento de la lista.
Parámetros de entrada:
Esta rutina no contiene valor como valores de entrada
Valores de retorno:
Nodo (este nodo) que retornara siempre el primer elemento de la lista.

Último

Retorna que nos retorna el último elemento de la lista
Parámetros de entrada:
Esta rutina no contiene valor como valores de entrada
Valores de retorno:
Nodo (este nodo) que retornara siempre el último elemento de la lista.

Siguiente

Se desplazará nodo que nos retorna el siguiente valor de la posición
Parámetros de entrada:
en posición (Entero que es la posición) + 1 a parámetro (Entero)
Valores de retorno:
Nodo (este nodo) que retornara el siguiente elemento dada a la posición mignorada como parámetro)

Anterior

Dada una posición nos retornará el valor anterior en la lista
Parámetros de entrada:
en posición (Entero que es la posición) - 1 a parámetro (Entero)
Valores de retorno:
Nodo (este nodo) que retornara el anterior elemento dada a la posición mignorada como parámetro)

Tamaño

Retorna el tamaño actual de la lista
Parámetros de entrada:
Esta rutina no contiene valor como valores de entrada
Valores de retorno:
en este retorna nos devuelve el valores entero a el tamaño actual de nuestra lista

Eliminar todo (vaciar)

Elimina todos los elementos
Parámetros de entrada:
Esta rutina no contiene valor como valores de entrada
Valores de retorno:
Esta rutina no contiene valor como valores de retorno

Mostrar lista

Imprime la lista en pantalla
Parámetros de entrada:
Esta rutina no contiene valor como valores de entrada
Valores de retorno:
Esta rutina no contiene valor como valores de retorno

Lista simple 1

```
Module ListaSpec1

Alias ClientesAlias
Alias ClientesAlias.Cliente
Alias ClientesAlias.Nodo
Alias ClientesAlias.Lista

using textexpect.ex2

Module Test.P

test ciclo1(fun1
  prueba
  fun1.(
    "Test Nodo
    "
  )
  assert(fun1.(
    ) == true
  )
end

test insertar(fun1
  prueba
  fun1.(
    "Test Nodo
    "
  )
  assert(fun1.(
    ) == "Nodo"
  )
end

test insertar_nombre(fun1, nombre)
  prueba
  fun1.(
    nombre
  )
  assert(fun1.(
    ) == nombre
  )
end

test eliminar(fun1, fun2)
  prueba
  fun1.(
    "Test Nodo
    "
  )
  fun2.(
    "Test Nodo
    "
  )
  assert(fun1.(
    ) == false
  )
end

test eliminar_nombre(fun1, fun2, nombre)
  prueba
  fun1.(
    "Test Nodo
    "
  )
  fun2.(
    "Test Nodo
    "
    nombre
  )
  assert(fun1.(
    ) == false
  )
end

test buscar(fun1, fun2)
  prueba
  fun1.(
    "Test Nodo
    "
  )
  fun2.(
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
  )
  assert(fun1.(
    ) == "Test Nodo
    "
  )
end

test buscar_nombre(fun1, fun2, nombre)
  prueba
  fun1.(
    "Test Nodo
    "
  )
  fun2.(
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    nombre
  )
  assert(fun1.(
    ) == "Test Nodo
    "
  )
end

test vacia(fun1, fun2)
  prueba
  fun1.(
    "Test Nodo
    "
  )
  fun2.(
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
  )
  assert(fun1.(
    ) == false
  )
end

test inicializa(fun1, fun2)
  prueba
  fun1.(
    "Test Nodo
    "
  )
  fun2.(
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
  )
  assert(fun1.(
    ) == "Lista vacia"
  )
end

test primero(fun1, fun2)
  prueba
  fun1.(
    "Test Nodo
    "
  )
  fun2.(
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
  )
  assert(fun1.(
    ) == "Lista vacia"
  )
end

test ultimo(fun1, fun2)
  prueba
  fun1.(
    "Test Nodo
    "
  )
  fun2.(
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
  )
  assert(fun1.(
    ) == "Lista vacia"
  )
end

test siguiente(fun1, fun2, posicion)
  prueba
  fun1.(
    "Test Nodo
    "
  )
  fun2.(
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    posicion
  )
  assert(fun1.(
    ) == "Lista vacia"
  )
end

test anterior(fun1, fun2, posicion)
  prueba
  fun1.(
    "Test Nodo
    "
  )
  fun2.(
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    posicion
  )
  assert(fun1.(
    ) == "Lista vacia"
  )
end

test tamaño(fun1, fun2, nombre)
  prueba
  fun1.(
    "Test Nodo
    "
  )
  fun2.(
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    nombre
  )
  assert(fun1.(
    ) == "Lista vacia"
  )
end

test mostrar_lista(fun1, fun2, nombre)
  prueba
  fun1.(
    "Test Nodo
    "
  )
  fun2.(
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    "Test Nodo
    "
    nombre
  )
  assert(fun1.(
    ) == "Lista vacia"
  )
end
```

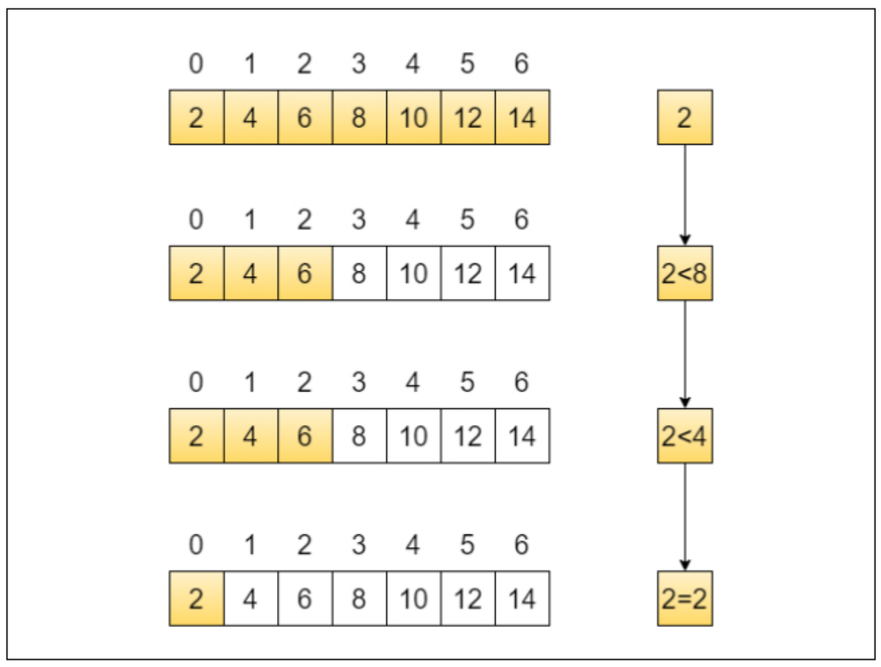
Lista Instancia 1

```
#
#
# Created by jesus on 2/10/2022
#

#MAKE PROCESS_ALIAS_ON_PLAN_DOSE_MOMENTO_LATINOTRA_O
#MAKE PROCESS_ALIAS_ON_PLAN_DOSE_MOMENTO_LATINOTRA_O

Module "Nodo.ex"
Module "Lista.ex"

defp CLIENTE_TYPEMAP(
  :id,
  :nombre,
  :edad,
  :fecha,
  :telefono,
  :email,
  :fecha_registrado_lunes

  :cantidad,

  :cantidad_domingo,
  :cantidad_domingo_2,
  :cantidad_domingo_3,
  :cantidad_domingo_4,
  :cantidad_domingo_5,
  :cantidad_domingo_6,
  :cantidad_domingo_7,
  :cantidad_domingo_8,
  :cantidad_domingo_9,
  :cantidad_domingo_10,
  :cantidad_domingo_11,
  :cantidad_domingo_12,
  :cantidad_domingo_13,
  :cantidad_domingo_14,
  :cantidad_domingo_15,
  :cantidad_domingo_16,
  :cantidad_domingo_17,
  :cantidad_domingo_18,
  :cantidad_domingo_19,
  :cantidad_domingo_20,
  :cantidad_domingo_21,
  :cantidad_domingo_22,
  :cantidad_domingo_23,
  :cantidad_domingo_24,
  :cantidad_domingo_25,
  :cantidad_domingo_26,
  :cantidad_domingo_27,
  :cantidad_domingo_28,
  :cantidad_domingo_29,
  :cantidad_domingo_30,
  :cantidad_domingo_31,
  :cantidad_domingo_32,
  :cantidad_domingo_33,
  :cantidad_domingo_34,
  :cantidad_domingo_35,
  :cantidad_domingo_36,
  :cantidad_domingo_37,
  :cantidad_domingo_38,
  :cantidad_domingo_39,
  :cantidad_domingo_40,
  :cantidad_domingo_41,
```

Busquedas

Busqueda binaria

La búsqueda binaria es el método, donde si el arreglo o vector esta bien ordenado, se reduce sucesivamente la operación eliminando repetidas veces la mitad de la lista restante.
El proceso comienza comparando el elemento central del arreglo con el elemento buscado. Si ambos coinciden finaliza la búsqueda. Si no ocurre así, el elemento buscado será mayor o menor en sentido estricto que el elemento central del arreglo. Si el elemento buscado es mayor se procede a hacer búsqueda binaria en el subarray superior, si el elemento buscado es menor que el contenido de la casilla central, se debe cambiar el segmento a considerar al segmento que está a la izquierda de tal sitio central.



Busqueda binaria

```
int binarySearch(int arr[], int l, int r, int x)
    {
       if (r >= l) {
    int mid = l + (r - l) / 2;

    // Si el elemento actual esta presente en el medio
    if (arr[mid] == x)
       return mid;
    // retornar el medio

    //si el elemento es más pequeño que el medio
    // entonces debe de estar presente en el subarray izquierdo
    if (arr[mid] > x)
       return binarySearch(arr, l, mid - 1, x);
         // si no esta el elemento solo puede estar
    //en el subarray derecho
    return binarySearch(arr, mid + 1, r, x);
       }

       //Si el elemento no se encuentra en el arreglo
       // retornamos -1
       return -1;
    }
```

Fuentes

https://uneginf05.es.tl/M-e2-todo-de-Busqueda-Secuencial-y-Binaria.htm

Busqueda Secuencial

La búsqueda secuencial, también se le conoce como búsqueda lineal.
Supongamos una colección de registros organizados como una lista lineal. El algoritmo básico de búsqueda secuencial consiste en empezar al inicio de la lista e ir a través de cada registro hasta encontrar la clave indicada (k) o hasta el final de la lista.
Este método consiste en recorrer el arreglo o vector elemento a elemento e ir comparando con el valor buscado (clave). Se empieza con la primera casilla del vector y se observa una casilla tras otra hasta que se encuentre el elemento buscado o se han visto todas las casillas. El resultado de la búsqueda es un solo valor, y será la posición del elemento buscado o cero.

Busqueda binaria

```
int search(int arr[], int N, int x){
    int i;
    //se recorre el arreglo
    for (i = 0; i < N; i++)
    //Si el elemento actual es igual al valor x
    if (arr[i] == x)
        //Retornamos el valor de i (la posicion)
        return i;
    return -1;
}
```

Fuentes

https://uneginf05.es.tl/M-e2-todo-de-Busqueda-Secuencial-y-Binaria.htm

Estructura de datos

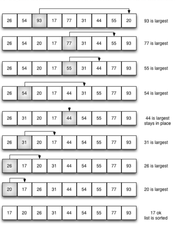
Basicos | POO | Ajustadores | Listas

Algoritmos de busqueda | Algoritmos de ordenamiento | Estructuras de datos Avanzadas | Articles y Grafos

Metodos de ordenamiento

Ordenamiento SelectSort

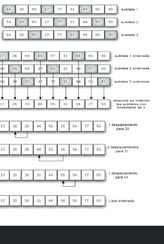
El algoritmo por selección tiene una complejidad de O(n^2), el algoritmo para realizar el mismo numero de comparaciones, es decir que cada elemento realizara una comparacion.




```
SelectSort
void swap(int *firstValue, int *secondValue){
    //Creamos una variable temporal y asignamos el primer valor recibido a este mismo
    int temp = *firstValue;
    //Cambiamos el valor de la primer variable por el de la segunda
    *firstValue = *secondValue;
    //Cambiamos el valor de la segunda por la primera
    *secondValue = firstValue;
}
void selectionSort(int arr[], int n){
    int
    int i, j, min_idx;
    //Recorremos el arreglo
    for (j = 0; j < n; j++){
    min_idx = j;
    //Recorremos el arreglo apartir de la posición i+1 para revisar los demás elementos
    for (j = i+1; j < n; j++){
        //Comparamos si el elemento en la posición j es menor que la posición index
        if (arr[j] < arr[min_idx])
            //Si se ejecutan ya menor que la posición index ter le asigna el valor de j al index
            min_idx = j;
    }
    //Es el caso de que exista uno diferente al valor de i
    if(min_idx!=j)
        //Realizamos el cambio de posición en los valores de la variable
        swap(&arr[min_idx], &arr[j]);
    }
}
```

Ordenamiento ShellSort

El ordenamiento ShellSort primero mueve los elementos mas grandes, de manera que los elementos con valores small se ordenaran con menores movimientos hasta su posicion final.




```
Shell sort
void shellSort(int arr[], int tamanoArr){
    //Recorremos el arreglo dividendo el tamaño por sin medio y por cada iteración lo dividiremos entre 2 para así obtener el valor de esmenor
    for (int gap = tamanoArr/ 2; gap > 0; gap /= 2){
    //Recorremos desde la mitad de tamaño mientras que i sea menor a tamaño
    for (int i = gap; i < tamaño; i ++){
        //asignamos el valor de la posición a una variable temporal
        int temp = arr[i];
        int j;
        //
        // cambia los elementos ordenados por valores anteriores
        // hacia que se encuentre la ubicación correcta para arr[i]
        for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
            arr[j] = arr[j - gap];
        //
        // Asignamos el valor temporal a la posición correcta
        arr[j] = temp;
    }
    }
    return 0;
}
```

Ordenamiento Burbuja mejorado

Mediante el metodo burbuja, dado un arreglo de n elementos, se requiere de n-1 pasos para ordenar el arreglo.

Ordenamiento QuickSort

El metodo quicksort fue desarrollado por el científico británico Ceilan Hoare en Colombo.

La complejidad del algoritmo en el mejor de los casos es O(n).

La forma en la que funciona este algoritmo es que simultaneamente se recorre por la izquierda a la par por la derecha. En el punto en el que se cruzan los indices sera la posicion adecuada para colocar el pivote porque sabemos que a uno de los lados todos los elementos son menores y que los del otro lado son mayores.

```
QuickSort
void swap(int *firstValue, int *secondValue){
    //Creamos una variable temporal y asignamos el primer valor recibido a este mismo
    int temp = *firstValue;
    //Cambiamos el valor de la primer variable por el de la segunda
    *firstValue = *secondValue;
    //Cambiamos el valor de la segunda por la primera
    *secondValue = firstValue;
}
int partition (int arr[], int low, int high){
    int
    // Creamos pivote que sera el elemento que se colocara en la posición correcta
    int pivot = arr[high];
    // Index del elemento más pequeño e indica la posición correcta del pivote
    int i = (low - 1);
    int j;
    for (int j = low; j <= high - 1; j++){
    //si el valor actual es más pequeño que el pivote
    if (arr[j] < pivot){
        //Increcuta incrementamos el valor de i
        i++;
        //cambiamos, cambiamos el valor de la posición a la posición j
        swap(&arr[i], &arr[j]);
    }
    }
    //Intercambiamos el valor de la posición i+1 por el valor de la posición inicial
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
//de inicio recibimos el arreglo, el valor minimo del arreglo y el tamaño del arreglo
void quickSort(int arr[], int low, int high){
    if (low < high){
    //Recibimos la posición del arreglo
    int p = partition(arr, low, high);
    //Incrementamos mediante una llamada la función quicksort con el valor más como p-1 para ordenar la primera mitad
    quickSort(arr, low, p - 1);

    //Recurrentemente mediante a llamar la función quicksort con el valor más bajo como p+1 para ordenar la segunda mitad
    quickSort(arr, p + 1, high);
    }
}
```

Ordenamiento MergeSort

El algoritmo mergesort es un algoritmo que funciona bajo el concepto divide y vencerás. Fue desarrollado en 1945 por John Von Neumann.

La complejidad del algoritmo es de O(n.Log(n))

El ordenamiento funciona dividiendo la lista en 2 sublistas. Ordena recursivamente cada sublista con el metodo merge.

```
MergeSort
//Mezclamos 2 subarreglos del arreglo principal
// O (1) primer sub arreglo es arr[begin, mid]
// O segundo sub arreglo es arr[mid + 1, end]
void merge(int arreglo[], int const left, int const mid,
           int const right){
    auto const subAryOne = mid - left + 1;
    auto const subAryTwo = right - mid;

    //Creamos los arreglos temporales
    // Crea temp arrays para almacenar
    auto *leftArray = new int[subAryOne],
        *rightArray = new int[subAryTwo];

    // Copiamos la información del arreglo a los sub arreglos leftArray[] y rightArray[]
    for (auto i = 0; i < subAryOne; i++)
        leftArray[i] = arreglo[left + i];
    for (auto j = 0; j < subAryTwo; j++)
        rightArray[j] = arreglo[mid + 1 + j];

    auto indexOfSubAryOne{ne
        = 0,// Inicializamos el index del primer subarreglo
        indexOfSubAryTwo
        = 0,// Inicializamos el index del segundo subarreglo
    int indexOfMergeArray
        = left;//Inicializamos el index del arreglo mezclado

    //Mezclamos los arreglos temporales al arreglo arr[left, right]
    while (indexOfSubAryOne < subAryOne){
        while (indexOfSubAryTwo < subAryTwo) {
        if (leftArray[indexOfSubAryOne]
            <= rightArray[indexOfSubAryTwo]) {
            arreglo[indexOfMergeArray]
                = leftArray[indexOfSubAryOne];
            indexOfSubAryOne++;
        }
        else {
            arreglo[indexOfMergeArray]
                = rightArray[indexOfSubAryTwo];
            indexOfSubAryTwo++;
        }
        indexOfMergeArray++
    }
    //Copiamos los elementos restantes del arreglo left[] si es que hay alguno
    while (indexOfSubAryOne < subAryOne) {
        arreglo[indexOfMergeArray]
            = leftArray[indexOfSubAryOne];
        indexOfSubAryOne++;
        indexOfMergeArray++;
    }
    //Copiamos los elementos restantes del arreglo right[] si es que hay alguno
    while (indexOfSubAryTwo < subAryTwo) {
        arreglo[indexOfMergeArray]
            = rightArray[indexOfSubAryTwo];
        indexOfSubAryTwo++;
        indexOfMergeArray++;
    }
    delete[] leftArray;
    delete[] rightArray;
}

// begin es para el index izquierdo
// end es para el index derecho del subarreglo
void mergeSort(int arreglo[], int const begin, int const end)
{
    if (begin >= end)
        return // Retorn para terminar la recursividad

    auto mid = begin + (end - begin) / 2;
    mergeSort(arreglo, begin, mid);
    mergeSort(arreglo, mid + 1, end);
    merge(arreglo, begin, mid, end);
}
```

Ordenamiento InsertSort

Este ordenamiento es la forma mas natural para el humano de ordenar y puede usarse para ordenar un mazo de cartas. Fue creado por Benjamin Bustos y Patricio Poblete.

Tiene una complejidad de O(n^2).

```
InsertSort
void insertSort(int arr[], int n){
    int
    int j, temp, j;
    for (i = 1; i < n; i++){
    j = i;
    temp = arr[i];
    j = i - 1;
    // Ralentiza los elementos del arreglo
    // el elemento más grande que key se mueve a la siguiente posicion
    while (j >= 0 && arr[j] > temp){
        arr[j + 1] = arr[j];
        j = j - 1;
    }
    arr[j + 1] = temp;
    }
}
```

Ordenamiento burbuja

El metodo de ordenamiento Burbujeor es uno de los algoritmos mas sencillos de ordenamiento. Funciona revisando cada elemento de la lista que se ordena, intercambiando los elemento en caso de no estar en el orden indicado.

El algoritmo realiza varios recorridos en la lista hasta no necesitar mas intercambios.

El autor del algoritmo no se conoce bien, pero parece que este algoritmo pertenece a Esteban H. Fraedge, por una de sus publicaciones sobre ordenamiento en sistemas de computadores electronicas.

La complejidad de este metodo depende de como se ebucuentren ordenadas los elementos, pero poderme decir que es complejidad es de O(n^2).

```
burbuja
void swap(int *firstValue, int *secondValue){
    //Creamos una variable temporal y asignamos el primer valor recibido a este mismo
    int temp = *firstValue;
    //Cambiamos el valor de la primer variable por el de la segunda
    *firstValue = *secondValue;
    //Cambiamos el valor de la segunda por la primera
    *secondValue = firstValue;
}
void bubbleSort(int arr[], int n){
    int i, j;
    //Recorremos el arreglo mientras i sea menor que n
    for (i = 0; i < n - 1; i++)
        for (j = 0; j < n - i - 1; j++)
            // Si el valor en la posición j es mayor que en la posición j+1
            if (arr[j] > arr[j + 1])
                //Cambiamos el valor de la posición j a la posición j+1 y el valor de la posición j+1 a la posición j
                swap(&arr[j], &arr[j + 1]);
}
```

Fuentes

https://stat.ebu.mx/archivos/2015/TPeltomaki/s3u5.html

TDA Pila

La estructura de datos pila, se rige con el estándar LIFO, es decir, el ultimo que entra es el primero en salir.

En las pilas encontramos las siguientes operaciones características:

1- Push: Agrega un nuevo elemento en la cima de la pila

2- Pop: Obtiene/borra el ultimo elemento agregado

TDA Pila .h

```
//
// Created by josue on 5/11/2022
//

#ifndef PROG02_AGUILAR_ISLAS_JOSUE_NORBERTO_PILA_H
#define PROG02_AGUILAR_ISLAS_JOSUE_NORBERTO_PILA_H

#include "Nodo.h"
#include "Utilities.h"
#include <string>

class Pila : Utilities{
public:
    Nodo *top;
    Pila();
    ~Pila();

    Nodo* push(Animal *animal);
    Nodo* pop();
    Nodo* buscarByID(int id);

    bool isEmpty();
    int tamanio();
    void vaciar();
    void print() const;
    void init();

};

#endif //PROG02_AGUILAR_ISLAS_JOSUE_NORBERTO_PILA_H
```

TDA Pila .cpp

```
//
// Created by josue on 5/11/2022
//

#include "Pila.h"

Pila::Pila(){
    top = nullptr;
}

Nodo* Pila::push(Animal *animal){
    Nodo *p, *q;
    p = new Nodo();
    p->data = animal;
    p->next = nullptr;
    if (top != nullptr) {
        q = top;
        while (q->next != nullptr) {
            q = q->next;
        }
        q->next = p;
    } else {
        top = p;
        p->next = nullptr;
    }
    return top;
}

Nodo* Pila::pop(){
    if(isEmpty()){
        string error = "Error, no se puede realizar la operación porque la lista se encuentra vacia";
        showError(error);
        return nullptr;
    }
    Nodo *p = top;
    top = p->next;
    return p;
}

Nodo* Pila::buscarByID(int id){
    if(isEmpty()){
        string error = "Error, la lista se encuentra vacia";
        showError(error);
        return nullptr;
    }
    Nodo *p = top;
    int cont = 0;
    while(p != nullptr){
        if(p->data->getId() == id){
            cout << cont << " " << p << " -> " << p->data->toString() << endl;
        }
        p = p->next;
        cont++;
    }
    return p;
}

bool Pila::isEmpty(){
    return top == nullptr;
}

int Pila::tamanio(){
    if(isEmpty()){
        string error = "Error, la lista se encuentra vacia";
        showError(error);
        return 0;
    }
    Nodo *p = top;
    int cont = 0;
    while(p != nullptr){
        p = p->next;
        cont++;
    }
    return cont;
}

void Pila::vaciar(){
    if(isEmpty()){
        string error = "Error, la lista se encuentra vacia";
        showError(error);
    }else{
        Nodo *p = top;
        while(p != nullptr){
            p = p->next;
        }
    }
}
void Pila::print() const{
    Nodo *p = top;
    int cont = 0;
    while(p != nullptr){
        cout << cont << " " << p << " -> " << p->data->toString() << endl;
        p = p->next;
        cont++;
    }
}

void Pila::init(){
    top = nullptr;
}
```

Fuentes

https://www.youtube.com/watch?v=XJ8cf3LHsAs4

TDA Cola

Las colas son una estructura de datos parecida a las colas en la vida real. Es decir, nos incorporaremos al final de la cola y conforme avanza llegaremos a la cabeza de la cola, donde terminara nuestro progreso.

Una cola se puede componer de una lista enlazada, solo es necesario unir nodos.

Las operaciones de una cola son parecidas a las de una lista, pero, son solo 3:

1-Encolar: La operación encolar agrega elementos al final de la cola, similar a la operación insertar en final de una lista.

2-Consultar: Obtiene el siguiente elemento de la cola.

3-Eliminar: Elimina un elemento al principio de la cola, esto se hace una vez se procesa el dato

TDA Cola .h

```
//
// Created by josue on 5/11/2022
//

#ifndef PROG02_AGUILAR_ISLAS_JOSUE_NORBERTO_COLA_H
#define PROG02_AGUILAR_ISLAS_JOSUE_NORBERTO_COLA_H

#include "Nodo.h"
#include "Utilities.h"
#include <string>

class Cola : Utilities{
public:
    Nodo *first;
    Nodo *last;

    Cola();
    ~Cola();

    Nodo* queue(Animal *animal);
    Nodo* dequeue();

    Nodo* buscarByID(int id);

    bool isEmpty();
    int tamanio();
    void vaciar();
    void print() const;
    void init();
};

#endif //PROG02_AGUILAR_ISLAS_JOSUE_NORBERTO_COLA_H
```

TDA Cola .cpp

```
//
// Created by josue on 5/11/2022
//

#include "Cola.h"
#include "Utilities.h"

Cola::Cola(){
    first = nullptr;
    last = nullptr;
}

Nodo* Cola::queue(Animal *animal){
    Nodo *p, *q;
    p = new Nodo();
    p->data = animal;
    p->next = nullptr;
    if (first != nullptr) {
        q = last;
        while (q->next != nullptr) {
            q = q->next;
        }
        q->next = p;
    } else {
        first = p;
        last = p;
        p->next = nullptr;
    }
    return last;
}

Nodo* Cola::dequeue(){
    Nodo *p, *q;
    if(last != nullptr && first != nullptr){
        p = first;
        first = p->next;
        return p;
    }else{
        string error = "Error, la lista se encuentra vacia";
        showError(error);
        return nullptr;
    }

}

Nodo* Cola::buscarByID(int id){
    if(isEmpty()){
        string error = "Error, la lista se encuentra vacia";
        showError(error);
        return nullptr;
    }
    Nodo *p = first;
    int cont = 0;
    while(p != last){
        if(p->data->getId() == id){
            cout << cont << " " << p << " -> " << p->data->toString() << endl;
        }
        p = p->next;
        cont++;
    }
    return p;
}

int Cola::tamanio(){
    if(isEmpty()){
        string error = "Error, la lista se encuentra vacia";
        showError(error);
        return 0;
    }
    Nodo *p = first;
    int cont = 0;
    while(p != nullptr){
        p = p->next;
        cont++;
    }
    return cont;
}

bool Cola::isEmpty(){
    if(first == last || (first != nullptr && last != nullptr)){
        return false;
    }
    return true;
}

void Cola::print() const{
    Nodo *p = first;
    int cont = 0;
    while(p != nullptr){
        cout << cont << " " << p << " -> " << p->data->toString() << endl;
        p = p->next;
        cont++;
    }

}

void Cola::vaciar(){
    if(isEmpty()){
        string error = "Error, la lista se encuentra vacia";
        showError(error);
    }else{
        Nodo *p = first;
        while(p != nullptr){
            p = p->next;

        }
    }
}

void Cola::init(){
    first = nullptr;
    last = nullptr;
}
```

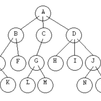
Fuentes

https://www.youtube.com/watch?v=rJDlnjEuLUrQ

Estructura de datos

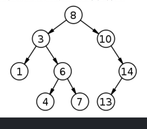
Arboles

Un árbol es parecido también como una estructura jerárquica y sin forma de bosque. aplicando todos los conceptos de elementos y objetos tenemos nodos
Un nodo es un elemento que contiene información y que puede tener un número de hijos. Un nodo puede tener cero hijos.
Los derechos son una utilización en informática como un método eficiente para interpretar grandes cantidades y complejas.
Los nodos hoja son los nodos que no tienen hijos. Los nodos que no son nodos hoja se llaman nodos internos.
En las líneas estructuras dinámicas, porque las internas pueden cambiar tanto de forma como de tamaño durante la ejecución del programa. Y estructuras no lineales porque cada elemento del dato puede tener más de un sucesor.



Arboles binarios

Un árbol binario es aquel en el que cada elemento siguiente como máximo a otros 2 elementos, comúnmente llamados hijo izquierdo y hijo derecho.




```
Arboles ejemplo

package  com.kaizen
Package  com.kaizen
using namespace  std;

template  <class T>
class  ArbolBinario;

template  <class T>
class  ArbolBinarioNodo
{
    private:
        T  dato;
        int  nivel;
        ArbolBinarioNodo<T.*>*  hijo;
        ArbolBinarioNodo<T.*>*  padre;
    public:
        ArbolBinarioNodo(T.&);
        ~ArbolBinarioNodo();
        void  setDato(T.&);
        T.&  getDato();
        friend class ArbolBinarioNodo<T.>;
        friend class ArbolBinarioArbol<T.>;
};

template  <class T>
ArbolBinarioNodo<T.>::ArbolBinarioNodo(T.&  d)
{
    dato = d;
    padre = NULL;
    hijo = NULL;
}

template  <class T>
void  ArbolBinarioNodo<T.>::setDato(T.&  d)
{
    dato = d;
}

template  <class T>
T.&  ArbolBinarioNodo<T.>::getDato()
{
    return dato;
}

template  <class T>
ArbolBinarioArbol<T.>::ArbolBinarioArbol(T.&  d)
{
    raiz = NULL;
    ultimo = NULL;
    cantidad = 0;
    ArbolBinarioNodo<T.>*  nodo = new ArbolBinarioNodo<T.>(d);
    if(nodo != NULL)
    {
        raiz = nodo;
        ultimo = nodo;
        cantidad = 1;
    }
}

template  <class T>
ArbolBinarioArbol<T.>::~ArbolBinarioArbol()
{
    while(cantidad > 0)
    {
        ArbolBinarioNodo<T.>*  nodo = ultimo->getPadre();
        if(ultimo->getPadre().getIzquierdo() == ultimo)
            nodo->setIzquierdo(NULL);
        else
            nodo->setDerecho(NULL);
        delete ultimo;
        ultimo = nodo;
        cantidad--;
    }
}

template  <class T>
void  ArbolBinarioArbol<T.>::setRaiz(T.&  d)
{
    if(cantidad == 0)
    {
        ArbolBinarioNodo<T.>*  nodo = new ArbolBinarioNodo<T.>(d);
        if(nodo != NULL)
        {
            raiz = nodo;
            ultimo = nodo;
            cantidad = 1;
        }
    }
    else
    {
        cout << "Ya existe la raiz" << endl;
    }
}

template  <class T>
void  ArbolBinarioArbol<T.>::setIzquierdo(T.&  d, ArbolBinarioNodo<T.>*  padre)
{
    if(padre != NULL)
    {
        ArbolBinarioNodo<T.>*  nodo = new ArbolBinarioNodo<T.>(d);
        if(nodo != NULL)
        {
            if(padre->getIzquierdo() == NULL)
            {
                padre->setIzquierdo(nodo);
                if(padre == ultimo)
                    ultimo = nodo;
                cantidad++;
                if(cantidad == 2)
                    raiz->setPadre(nodo);
                else
                    nodo->setPadre(padre);
            }
            else
            {
                cout << "Ya tiene un hijo izquierdo" << endl;
                delete nodo;
                nodo = NULL;
            }
        }
        else
            cout << "Error de memoria" << endl;
    }
    else
        cout << "Nodo padre no existe" << endl;
}

template  <class T>
void  ArbolBinarioArbol<T.>::setDerecho(T.&  d, ArbolBinarioNodo<T.>*  padre)
{
    if(padre != NULL)
    {
        ArbolBinarioNodo<T.>*  nodo = new ArbolBinarioNodo<T.>(d);
        if(nodo != NULL)
        {
            if(padre->getDerecho() == NULL)
            {
                padre->setDerecho(nodo);
                if(padre == ultimo)
                    ultimo = nodo;
                cantidad++;
                if(cantidad == 2)
                    raiz->setPadre(nodo);
                else
                    nodo->setPadre(padre);
            }
            else
            {
                cout << "Ya tiene un hijo derecho" << endl;
                delete nodo;
                nodo = NULL;
            }
        }
        else
            cout << "Error de memoria" << endl;
    }
    else
        cout << "Nodo padre no existe" << endl;
}

template  <class T>
void  ArbolBinarioArbol<T.>::setPadre(T.&  d, ArbolBinarioNodo<T.>*  hijo)
{
    if(hijo != NULL)
    {
        ArbolBinarioNodo<T.>*  nodo = new ArbolBinarioNodo<T.>(d);
        if(nodo != NULL)
        {
            if(hijo == raiz)
            {
                raiz->setPadre(nodo);
                nodo->setIzquierdo(raiz);
                raiz = nodo;
                if(cantidad == 1)
                    ultimo = nodo;
                cantidad++;
                if(cantidad == 2)
                    raiz->setPadre(nodo);
                else
                    nodo->setPadre(hijo->getPadre());
            }
            else
            {
                cout << "Nodo no es la raiz" << endl;
                delete nodo;
                nodo = NULL;
            }
        }
        else
            cout << "Error de memoria" << endl;
    }
    else
        cout << "Nodo hijo no existe" << endl;
}

template  <class T>
void  ArbolBinarioArbol<T.>::setIzquierdo(T.&  d, ArbolBinarioNodo<T.>*  padre, ArbolBinarioNodo<T.>*  hijo)
{
    if(padre != NULL && hijo != NULL)
    {
        ArbolBinarioNodo<T.>*  nodo = new ArbolBinarioNodo<T.>(d);
        if(nodo != NULL)
        {
            if(padre->getIzquierdo() == hijo)
            {
                nodo->setIzquierdo(hijo);
                padre->setIzquierdo(nodo);
                if(hijo == ultimo)
                    ultimo = nodo;
                cantidad++;
                if(cantidad == 2)
                    raiz->setPadre(nodo);
                else
                    nodo->setPadre(padre);
                if(cantidad == 3)
                    hijo->setPadre(nodo);
                else
                    nodo->setPadre(hijo->getPadre());
            }
            else
            {
                cout << "Nodo no es hijo izquierdo" << endl;
                delete nodo;
                nodo = NULL;
            }
        }
        else
            cout << "Error de memoria" << endl;
    }
    else
        cout << "Nodo padre o nodo hijo no existe" << endl;
}

template  <class T>
void  ArbolBinarioArbol<T.>::setDerecho(T.&  d, ArbolBinarioNodo<T.>*  padre, ArbolBinarioNodo<T.>*  hijo)
{
    if(padre != NULL && hijo != NULL)
    {
        ArbolBinarioNodo<T.>*  nodo = new ArbolBinarioNodo<T.>(d);
        if(nodo != NULL)
        {
            if(padre->getDerecho() == hijo)
            {
                nodo->setDerecho(hijo);
                padre->setDerecho(nodo);
                if(hijo == ultimo)
                    ultimo = nodo;
                cantidad++;
                if(cantidad == 2)
                    raiz->setPadre(nodo);
                else
                    nodo->setPadre(padre);
                if(cantidad == 3)
                    hijo->setPadre(nodo);
                else
                    nodo->setPadre(hijo->getPadre());
            }
            else
            {
                cout << "Nodo no es hijo derecho" << endl;
                delete nodo;
                nodo = NULL;
            }
        }
        else
            cout << "Error de memoria" << endl;
    }
    else
        cout << "Nodo padre o nodo hijo no existe" << endl;
}

template  <class T>
void  ArbolBinarioArbol<T.>::setPadre(T.&  d, ArbolBinarioNodo<T.>*  hijo, ArbolBinarioNodo<T.>*  padre)
{
    if(hijo != NULL && padre != NULL)
    {
        ArbolBinarioNodo<T.>*  nodo = new ArbolBinarioNodo<T.>(d);
        if(nodo != NULL)
        {
            if(hijo == raiz)
            {
                raiz->setPadre(nodo);
                nodo->setIzquierdo(raiz);
                raiz = nodo;
                if(cantidad == 1)
                    ultimo = nodo;
                cantidad++;
                if(cantidad == 2)
                    raiz->setPadre(nodo);
                else
                    nodo->setPadre(hijo->getPadre());
                if(padre->getIzquierdo() == hijo)
                    padre->setIzquierdo(nodo);
                else
                    padre->setDerecho(nodo);
                if(padre == ultimo)
                    ultimo = nodo;
                if(cantidad == 3)
                    hijo->setPadre(nodo);
                else
                    nodo->setPadre(hijo->getPadre());
                if(padre->getIzquierdo() == hijo)
                    padre->setIzquierdo(NULL);
                else
                    padre->setDerecho(NULL);
                delete hijo;
                hijo = NULL;
            }
            else
            {
                cout << "Nodo no es la raiz" << endl;
                delete nodo;
                nodo = NULL;
                delete hijo;
                hijo = NULL;
                delete padre;
                padre = NULL;
            }
        }
        else
            cout << "Error de memoria" << endl;
    }
    else
        cout << "Nodo hijo o nodo padre no existe" << endl;
}

template  <class T>
void  ArbolBinarioArbol<T.>::setIzquierdo(T.&  d, ArbolBinarioNodo<T.>*  hijo)
{
    if(hijo != NULL)
    {
        ArbolBinarioNodo<T.>*  nodo = new ArbolBinarioNodo<T.>(d);
        if(nodo != NULL)
        {
            if(hijo->getIzquierdo() == NULL)
            {
                hijo->setIzquierdo(nodo);
                if(hijo == ultimo)
                    ultimo = nodo;
                cantidad++;
                if(cantidad == 2)
                    raiz->setPadre(nodo);
                else
                    nodo->setPadre(hijo);
                if(cantidad == 3)
                    hijo->setPadre(nodo);
                else
                    nodo->setPadre(hijo->getPadre());
                if(hijo == hijo->getPadre()->getIzquierdo())
                    hijo->getPadre()->setIzquierdo(nodo);
                else
                    hijo->getPadre()->setDerecho(nodo);
                if(hijo->getPadre() == ultimo)
                    ultimo = nodo;
                if(hijo == raiz)
                    raiz = nodo;
                delete hijo;
                hijo = NULL;
            }
            else
            {
                cout << "Nodo hijo ya tiene hijo izquierdo" << endl;
                delete nodo;
                nodo = NULL;
                delete hijo;
                hijo = NULL;
                delete padre;
                padre = NULL;
            }
        }
        else
            cout << "Error de memoria" << endl;
    }
    else
        cout << "Nodo hijo o nodo padre no existe" << endl;
}

template  <class T>
void  ArbolBinarioArbol<T.>::setDerecho(T.&  d, ArbolBinarioNodo<T.>*  hijo)
{
    if(hijo != NULL)
    {
        ArbolBinarioNodo<T.>*  nodo = new ArbolBinarioNodo<T.>(d);
        if(nodo != NULL)
        {
            if(hijo->getDerecho() == NULL)
            {
                hijo->setDerecho(nodo);
                if(hijo == ultimo)
                    ultimo = nodo;
                cantidad++;
                if(cantidad == 2)
                    raiz->setPadre(nodo);
                else
                    nodo->setPadre(hijo);
                if(cantidad == 3)
                    hijo->setPadre(nodo);
                else
                    nodo->setPadre(hijo->getPadre());
                if(hijo == hijo->getPadre()->getIzquierdo())
                    hijo->getPadre()->setIzquierdo(nodo);
                else
                    hijo->getPadre()->setDerecho(nodo);
                if(hijo->getPadre() == ultimo)
                    ultimo = nodo;
                if(hijo == raiz)
                    raiz = nodo;
                delete hijo;
                hijo = NULL;
            }
            else
            {
                cout << "Nodo hijo ya tiene hijo derecho" << endl;
                delete nodo;
                nodo = NULL;
                delete hijo;
                hijo = NULL;
                delete padre;
                padre = NULL;
            }
        }
        else
            cout << "Error de memoria" << endl;
    }
    else
        cout << "Nodo hijo o nodo padre no existe" << endl;
}

template  <class T>
void  ArbolBinarioArbol<T.>::setPadre(T.&  d, ArbolBinarioNodo<T.>*  hijo, ArbolBinarioNodo<T.>*  padre)
{
    if(hijo != NULL && padre != NULL)
    {
        ArbolBinarioNodo<T.>*  nodo = new ArbolBinarioNodo<T.>(d);
        if(nodo != NULL)
        {
            if(hijo == raiz)
            {
                raiz->setPadre(nodo);
                nodo->setIzquierdo(raiz);
                raiz = nodo;
                if(cantidad == 1)
                    ultimo = nodo;
                cantidad++;
                if(cantidad == 2)
                    raiz->setPadre(nodo);
                else
                    nodo->setPadre(hijo->getPadre());
                if(padre->getIzquierdo() == hijo)
                    padre->setIzquierdo(nodo);
                else
                    padre->setDerecho(nodo);
                if(padre == ultimo)
                    ultimo = nodo;
                if(cantidad == 3)
                    hijo->setPadre(nodo);
                else
                    nodo->setPadre(hijo->getPadre());
                if(padre->getIzquierdo() == hijo)
                    padre->setIzquierdo(NULL);
                else
                    padre->setDerecho(NULL);
                delete hijo;
                hijo = NULL;
                delete padre;
                padre = NULL;
            }
            else
            {
                cout << "Nodo no es la raiz" << endl;
                delete nodo;
                nodo = NULL;
                delete hijo;
                hijo = NULL;
                delete padre;
                padre = NULL;
            }
        }
        else
            cout << "Error de memoria" << endl;
    }
    else
        cout << "Nodo hijo o nodo padre no existe" << endl;
}

template  <class T>
void  ArbolBinarioArbol<T.>::setIzquierdo(T.&  d, ArbolBinarioNodo<T.>*  padre)
{
    if(padre != NULL)
    {
        ArbolBinarioNodo<T.>*  nodo = new ArbolBinarioNodo<T.>(d);
        if(nodo != NULL)
        {
            if(padre->getIzquierdo() == NULL)
            {
                padre->setIzquierdo(nodo);
                if(padre == ultimo)
                    ultimo = nodo;
                cantidad++;
                if(cantidad == 2)
                    raiz->setPadre(nodo);
                else
                    nodo->setPadre(padre);
                if(cantidad == 3)
                    padre->setPadre(nodo);
                else
                    nodo->setPadre(padre->getPadre());
                if(padre == padre->getPadre()->getIzquierdo())
                    padre->getPadre()->setIzquierdo(nodo);
                else
                    padre->getPadre()->setDerecho(nodo);
                if(padre->getPadre() == ultimo)
                    ultimo = nodo;
                if(padre == raiz)
                    raiz = nodo;
                delete padre;
                padre = NULL;
            }
            else
            {
                cout << "Nodo padre ya tiene hijo izquierdo" << endl;
                delete nodo;
                nodo = NULL;
                delete padre;
                padre = NULL;
            }
        }
        else
            cout << "Error de memoria" << endl;
    }
    else
        cout << "Nodo padre no existe" << endl;
}

template  <class T>
void  ArbolBinarioArbol<T.>::setDerecho(T.&  d, ArbolBinarioNodo<T.>*  padre)
{
    if(padre != NULL)
    {
        ArbolBinarioNodo<T.>*  nodo = new ArbolBinarioNodo<T.>(d);
        if(nodo != NULL)
        {
            if(padre->getDerecho() == NULL)
            {
                padre->setDerecho(nodo);
                if(padre == ultimo)
                    ultimo = nodo;
                cantidad++;
                if(cantidad == 2)
                    raiz->setPadre(nodo);
                else
                    nodo->setPadre(padre);
                if(cantidad == 3)
                    padre->setPadre(nodo);
                else
                    nodo->setPadre(padre->getPadre());
                if(padre == padre->getPadre()->getIzquierdo)
                    padre->getPadre()->setIzquierdo(nodo);
                else
                    padre->getPadre()->setDerecho(nodo);
                if(padre->getPadre() == ultimo)
                    ultimo = nodo;
                if(padre == raiz)
                    raiz = nodo;
                delete padre;
                padre = NULL;
            }
            else
            {
                cout << "Nodo padre ya tiene hijo derecho" << endl;
                delete nodo;
                nodo = NULL;
                delete padre;
                padre = NULL;
            }
        }
        else
            cout << "Error de memoria" << endl;
    }
    else
        cout << "Nodo padre no existe" << endl;
}

template  <class T>
void  ArbolBinarioArbol<T.>::setPadre(T.&  d, ArbolBinarioNodo<T.>*  padre)
{
    if(padre != NULL)
    {
        ArbolBinarioNodo<T.>*  nodo = new ArbolBinarioNodo<T.>(d);
        if(nodo != NULL)
        {
            if(padre == raiz)
            {
                raiz->setPadre(nodo);
                nodo->setIzquierdo(raiz);
                raiz = nodo;
                if(cantidad == 1)
                    ultimo = nodo;
                cantidad++;
                if(cantidad == 2)
                    raiz->setPadre(nodo);
                else
                    nodo->setPadre(padre->getPadre());
                if(cantidad == 3)
                    padre->setPadre(nodo);
                else
                    nodo->setPadre(padre->getPadre());
                if(padre == padre->getPadre()->getIzquierdo())
                    padre->getPadre()->setIzquierdo(nodo);
                else
                    padre->getPadre()->setDerecho(nodo);
                if(padre->getPadre() == ultimo)
                    ultimo = nodo;
                if(padre == raiz)
                    raiz = nodo;
                delete padre;
                padre = NULL;
            }
            else
            {
                cout << "Nodo padre no es la raiz" << endl;
                delete nodo;
                nodo = NULL;
                delete padre;
                padre = NULL;
            }
        }
        else
            cout << "Error de memoria" << endl;
    }
    else
        cout << "Nodo padre no existe" << endl;
}

template  <class T>
void  ArbolBinarioArbol<T.>::setIzquierdo(T.&  d, ArbolBinarioNodo<T.>*  padre, int  n)
{
    if(n > 0 && padre != NULL)
    {
        int  i = 1;
        ArbolBinarioNodo<T.>*  nodo = new ArbolBinarioNodo<T.>(d);
        if(nodo != NULL)
        {
            while(padre->getIzquierdo() != NULL && i < n)
            {
                padre = padre->getIzquierdo();
                i++;
            }
            if(i == n)
            {
                if(padre->getIzquierdo() == NULL)
                {
                    padre->setIzquierdo(nodo);
                    if(padre == ultimo)
                        ultimo = nodo;
                    cantidad++;
                    if(cantidad == 2)
                        raiz->setPadre(nodo);
                    else
                        nodo->setPadre(padre);
                    if(cantidad == 3)
                        padre->setPadre(nodo);
                    else
                        nodo->setPadre(padre->getPadre());
                    if(padre == padre->getPadre()->getIzquierdo())
                        padre->getPadre()->setIzquierdo(nodo);
                    else
                        padre->getPadre()->setDerecho(nodo);
                    if(padre->getPadre() == ultimo)
                        ultimo = nodo;
                    if(padre == raiz)
                        raiz = nodo;
                    delete padre;
                    padre = NULL;
                }
                else
                {
                    cout << "Nodo padre ya tiene hijo izquierdo" << endl;
                    delete nodo;
                    nodo = NULL;
                    delete padre;
                    padre = NULL;
                }
            }
            else
                cout << "Nodo padre no existe" << endl;
        }
        else
            cout << "Error de memoria" << endl;
    }
    else
        cout << "Nodo padre o n menor o igual a cero" << endl;
}

template  <class T>
void  ArbolBinarioArbol<T.>::setDerecho(T.&  d, ArbolBinarioNodo<T.>*  padre, int  n)
{
    if(n > 0 && padre != NULL)
    {
        int  i = 1;
        ArbolBinarioNodo<T.>*  nodo = new ArbolBinarioNodo<T.>(d);
        if(nodo != NULL)
        {
            while(padre->getDerecho() != NULL && i < n)
            {
                padre = padre->getDerecho();
                i++;
            }
            if(i == n)
            {
                if(padre->getDerecho() == NULL)
                {
                    padre->setDerecho(nodo);
                    if(padre == ultimo)
                        ultimo = nodo;
                    cantidad++;
                    if(cantidad == 2)
                        raiz->setPadre(nodo);
                    else
                        nodo->setPadre(padre);
                    if(cantidad == 3)
                        padre->setPadre(nodo);
                    else
                        nodo->setPadre(padre->getPadre());
                    if(padre == padre->getPadre()->getIzquierdo())
                        padre->getPadre()->setIzquierdo(nodo);
                    else
                        padre->getPadre()->setDerecho(nodo);
                    if(padre->getPadre() == ultimo)
                        ultimo = nodo;
                    if(padre == raiz)
                        raiz = nodo;
                    delete padre;
                    padre = NULL;
                }
                else
                {
                    cout << "Nodo padre ya tiene hijo derecho" << endl;
                    delete nodo;
                    nodo = NULL;
                    delete padre;
                    padre = NULL;
                }
            }
            else
                cout << "Nodo padre no existe" << endl;
        }
        else
            cout << "Error de memoria" << endl;
    }
    else
        cout << "Nodo padre o n menor o igual a cero" << endl;
}

template  <class T>
void  ArbolBinarioArbol<T.>::setPadre(T.&  d, ArbolBinarioNodo<T.>*  padre, int  n)
{
    if(n > 0 && padre != NULL)
    {
        int  i = 1;
        ArbolBinarioNodo<T.>*  nodo = new ArbolBinarioNodo<T.>(d);
        if(nodo != NULL)
        {
            while(padre->getPadre() != NULL && i < n)
            {
                padre = padre->getPadre();
                i++;
            }
            if(i == n)
            {
                if(padre == raiz)
                {
                    raiz->setPadre(nodo);
                    nodo->setIzquierdo(raiz);
                    raiz = nodo;
                    if(cantidad == 1)
                        ultimo = nodo;
                    cantidad++;
                    if(cantidad == 2)
                        raiz->setPadre(nodo);
                    else
                        nodo->setPadre(padre->getPadre());
                    if(cantidad == 3)
                        padre->setPadre(nodo);
                    else
                        nodo->setPadre(padre->getPadre());
                    if(padre == padre->getPadre()->getIzquierdo())
                        padre->getPadre()->setIzquierdo(nodo);
                    else
                        padre->getPadre()->setDerecho(nodo);
                    if(padre->getPadre() == ultimo)
                        ultimo = nodo;
                    if(padre == raiz)
                        raiz = nodo;
                    delete padre;
                    padre = NULL;
                }
                else
                {
                    cout << "Nodo padre no es la raiz" << endl;
                    delete nodo;
                    nodo = NULL;
                    delete padre;
                    padre = NULL;
                }
            }
            else
                cout << "Nodo padre no existe" << endl;
        }
        else
            cout << "Error de memoria" << endl;
    }
    else
        cout << "Nodo padre o n menor o igual a cero" << endl;
}

template  <class T>
void  ArbolBinarioArbol<T.>::setIzquierdo(int  n, int  m)
{
    if(n > 0 && m > 0 && n != m)
    {
        int  i = 1, j = 1;
        ArbolBinarioNodo<T.>*  nodo = NULL, *padre = NULL;
        while(raiz->getIzquierdo() != NULL && i < n)
            raiz = raiz->getIzquierdo();
        if(i == n)
            nodo = raiz;
        while(raiz->getIzquierdo() != NULL && j < m)
            raiz = raiz->getIzquierdo();
        if(j == m)
            padre = raiz;
        if(nodo != NULL && padre != NULL)
        {
            if(nodo == raiz)
            {
                raiz->setPadre(padre);
                padre->setIzquierdo(raiz);
                raiz = padre;
                if(cantidad == 1)
                    ultimo = padre;
                cantidad++;
                if(cantidad == 2)
                    raiz->setPadre(padre);
                else
                    padre->setPadre(nodo->getPadre());
                if(cantidad == 3)
                    nodo->setPadre(padre);
                else
                    padre->setPadre(nodo->getPadre());
                if(nodo == nodo->getPadre()->getIzquierdo())
                    nodo->getPadre()->setIzquierdo(padre);
                else
                    nodo->getPadre()->setDerecho(padre);
                if(nodo->getPadre() == ultimo)
                    ultimo = padre;
                if(nodo == raiz)
                    raiz = padre;
                delete nodo;
                nodo = NULL;
                delete padre;
                padre = NULL;
            }
            else
            {
                cout << "Nodo no es la raiz" << endl;
                delete nodo;
                nodo = NULL;
                delete padre;
                padre = NULL;
                delete raiz;
                raiz = NULL;
                delete ultimo;
                ultimo = NULL;
                delete cantidad;
                cantidad = NULL
```