EDUCACIÓN
SECRETARÍA DE EDUCACIÓN PÚBLICA

Tecnológico Nacional de México
Instituto Tecnológico de Tijuana

SUBDIRECCIÓN ACADÉMICA

DEPARTAMENTO DE SISTEMAS Y COMPUTACIÓN

SEMESTRE: AGOSTO – DICIEMBRE 2021

ESTRUCTURA DE DATOS - 2SC3D

INVESTIGACIÓN

UNIDAD 1 – INTRODUCCIÓN A LAS ESTRUCTURAS DE DATOS

GUILLEN MARTINEZ ANTHONY #20210575

M.C.C. LUZ ELENA CORTEZ GALVAN

Tijuana B.C 09/09/2021

INDICE

1.1 – Clasificación de las estructuras de datos.

Concepto.

Una estructura de datos es una clase de datos que se puede caracterizar por su organización y operaciones definidas sobre ella. Algunas veces a estas estructuras se les llama tipos de datos.

Las estructuras de datos son una forma de organizar los datos en la computadora, de tal manera que nos permita realizar unas operaciones con ellas de forma muy eficiente.

Es decir, igual que un array introducimos un dato y eso es prácticamente inmediato, no siempre lo es, según qué estructuras de datos y qué operaciones.

Depende que algoritmo queramos ejecutar, habrá veces que sea mejor utilizar una estructura de datos u otra estructura que nos permita más velocidad.

Por este motivo es interesante conocer algo más que simplemente los arrays o los hashmaps que casi todo el mundo conoce.

Clasificaciones.

ESTRUCTURAS LÓGICAS DE DATOS:

En un programa, cada variable pertenece a alguna estructura de datos explícita o implícitamente definida, la cual determina el conjunto de operaciones válidas para ella. Las estructuras de datos que se discuten aquí son estructuras de datos lógicas. Cada estructura de datos lógica puede tener varias representaciones físicas diferentes para sus almacenamientos

ESTRUCTURAS PRIMITIVAS Y SIMPLES:

Son primitivas aquellas que no están compuestas por otras estructuras de datos, por ejemplo, enteros, booleanos y caracteres. Otras estructuras de datos se pueden construir de una o más primitivas. Las estructuras de datos simples que consideramos se construyen a partir de estructuras primitivas y son: cadenas, arreglos y registros. A estas estructuras de datos las respaldan muchos lenguajes de programación.

ESTRUCTURAS LINEALES Y NO LINEALES:

Las estructuras de datos simples se pueden combinar de varias maneras para formar estructuras más complejas. Los dos cases principales de estructuras de datos son las lineales y las no lineales, dependiendo de la complejidad de las relaciones lógicas que representan. Las estructuras de datos lineales incluyen pilas, colas y listas ligadas lineales. Las estructuras de datos no lineales incluyen grafos y árboles.

DATOS ESTATICOS
Su tamaño y forma es constante durante la ejecución de un programa y por tanto se determinan en tiempo de compilación. El ejemplo típico son los arrays. Tienen el problema de que hay que dimensionar la estructura de antemano, lo que puede conllevar desperdicio o falta de memoria.

DATOS DINAMICOS

Su tamaño y forma es variable (o puede serlo) a lo largo de un programa, por lo que se crean y destruyen en tiempo de ejecución. Esto permite dimensionar la estructura de datos de una forma precisa: se va asignando memoria en tiempo de ejecución según se va necesitando.

Ejemplos.

Clase para definir un nodo

Definimos dos constructores, el segundo permite crear un nodo y definir el nodo al que apunta como siguiente elemento en la lista.

```
class Node
{
    private object data;
    private Node next;

    public Node(object datavalue)
    {
        this.data = datavalue;
        this.next = null;
    }

    public Node(object datavalue, Node nextnode)
    {
        this.data = datavalue;
        this.next = nextnode;
    }

    public Node Next { get { return this.next; } set { this.next = value; } }
    public object Data { get { return this.data; } set { this.data = value; } }
}
```

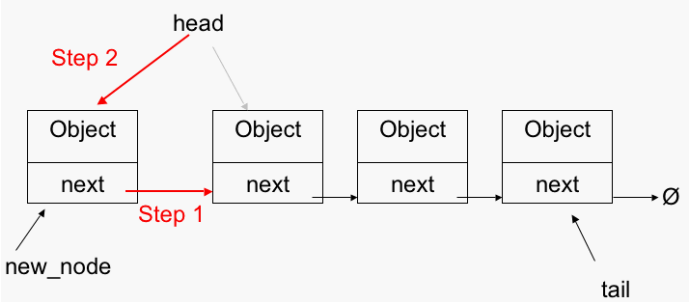
Lo más remarcable de la clase es la referencia object. Nos permite almacenar tipos de datos simples como si fueran objetos. El tipo object es un alias para Object en .NET. Todos los tipos de variables son herencia directa de este tipo. Podemos asignar valores de cualquier tipo a variables de tipo object. Cuando una variable se convierte a un tipo object a esta operación se le llama boxing o boxed. Cuando se realiza la operación inversa se denomina unboxing.

```
static void Main(string[] args)
{
    int i = 5;
    // Boxing explicito.
    object a = (object)i;
    // Boxing implicito.
    object b = i;
    // Unboxing explicito.
    int j = (int)a;

    a = new Node(i);

}
```

Clase lista



La clase lista tiene contiene como miembros head y tail. Son respectivamente referencias al primer y último nodo de la lista, también definimos una variable string para asignar un nombre a la lista.

```
public class List
{
    private Node head;
    private Node tail;
    private string name;

    public List(string listname)
    {
        name = listname;
        head = tail = null;
    }

    public List()
    {
        name = "list";
        head = tail = null;
    }
```

Definimos un método que nos resultará de utilidad más adelante, el método IsListEmpty retorna true si la cabeza de lista apunta a null.

```
public bool IsListEmpty() { return head == null; }
```

Ahora definimos un método para operar sobre la lista insertando un nuevo nodo al inicio de la misma. Si la lista esta recién creada o vacía la cabeza y la cola apunta al nuevo y único la misma. En caso contrario la cabeza apunta al nuevo nodo y le pasamos el nodo de cabeza actual como miembro para que quede en segundo lugar.

```
public void InsertHead(object Item)
{
    if (IsListEmpty())
        head = tail = new Node(Item);
    else
        head = new Node(Item, head);
}
```

Para añadir un nodo al final definimos el siguiente método:

```
public void InsertTail(object Item)
{
    if (IsListEmpty())
        head = tail = new Node(Item);
    else
        tail = tail.Next = new Node(Item);
}
```

Antes de continuar con un método para borrar un elemento del inicio de la lista vamos a definir una clase EmptyListException para lanzar una excepción cuando se producen operaciones ilegales sobra la lista, por ejemplo, si la lista está vacía. Usamos System.ApplicationException para excepciones generadas por nuestro programa.

```
public class EmptyListException : ApplicationException
{
    public EmptyListException(string name)
        : base("List named " + name + " is empty.")
    {
    }
}
```

Ahora ya podemos crear un método para borrar un elemento de la cabecera de la lista. Si la lista está vacía lanza una excepción que podemos capturar y tratar desde el programa principal. Después obtenemos el miembro del nodo de cabecera y restablece las referencias del primer y último nodo (si solo hay un nodo en la lista head y last quedaran a null, si hay más de un elemento avanzamos al siguiente nodo la cabecera).

```
public object DeleteHead()
{
    if (IsListEmpty())
        throw new EmptyListException(name);

    object DeletedObject = head.Data;

    if (head == tail)
        head = tail = null;
    else
        head = head.Next;

    return DeletedObject;
}
```

Visto el anterior ejemplo borrar el último nodo es similar. Pero en este caso el método que debemos seguir nodo no es muy eficiente (esto se solucionaría con una lista doblemente enlazada). Recorremos desde el primero nodo uno de detrás de otro hasta que el nodo siguiente no sea el último, de esta manera hacemos que apunte a null quedando fuera el último nodo.

```
public object DeleteTail()
{
    if (IsListEmpty())
        throw new EmptyListException(name);

    object DeletedObject = tail.Data;

    if (head == tail)
        head = tail = null;
    else
    {
        Node actual = head;
        while (actual.Next != tail)
            actual = actual.Next;

        tail = actual;
        actual.Next = null;
    }

    return DeletedObject;
}
```

Ahora solo nos queda un método para imprimir los nodos de la lista.

```
public void Print()
{
    if (IsListEmpty())
    {
        Console.WriteLine("La lista"+name+" esta vacía");
        return;
    }

    Console.WriteLine("Lista "+name);

    Node actual = head;

    while (actual != null)
    {
        Console.Write(actual.Data + " ");
    }

    Console.WriteLine("\n");
}
```

Ahora veamos cómo se puede utilizar:

```
static void Main(string[] args)
{
    List list = new List();

    bool mybool = true;
    char mychar = 'a';
    int myint = 2;
    string mystring = "prueba";

    list.InsertHead(mybool);
    list.InsertHead(mychar);
    list.InsertTail(myint);
    list.InsertTail(mystring);
    list.Print();

    Console.WriteLine("Press key to exit...");
    Console.ReadLine();

}
```

1.2 – Tipos de datos abstractos (TDA).

Qué son los Tipos de Datos Abstractos

Un TAD se define como una estructura algebraica compuesta por un conjunto de objetos abstractos que modelan elementos del mundo real, y un conjunto de operaciones para su manipulación.

Un TAD es un ente cerrado y autosuficiente, que no requiere de un contexto específico para que pueda ser utilizado en un programa. Esto garantiza portabilidad y reutilización del software. Las partes que forman un TAD son:

1. Atributos (tipos de datos, identificadores, etc.)
2. Funciones (rutinas) que definen las operaciones válidas para manipular los datos (atributos).

Las operaciones de un TAD se clasifican en 3 grupos, según su función sobre el objeto abstracto:

- Constructora: es la operación encargada de crear elementos del TAD.
- Modificadora: es la operación que puede alterar el estado de un elemento de un TAD.
- Analizadora: es una operación que no altera el estado del objeto, sino que tiene como misión consultar su estado y retornar algún tipo de información.

Cuáles son los TDA que maneja C#

Entre las aplicaciones de los TAD's se encuentran el TAD lista, el TAD pila, el TAD cola, etc.

TAD's Estáticos.

La creación y mantenimiento de un TAD estático requiere de memoria no dinámica, es decir, el espacio en memoria para almacenar los datos es reservado en tiempo de compilación (Arreglos).

TAD's Dinámicos (Asignación dinámica de memoria).

La creación y mantenimiento de estructuras dinámicas de datos (TAD's dinámicos), requiere de obtener más espacio de memoria (reservar memoria) en tiempo de ejecución para almacenar datos o para almacenar el tipo de clase "Nodo".

El TAD Lista

Una lista está formada por una serie de elementos llamados nodos los cuales son objetos que contiene como variable miembro un puntero asignado y variables de cualquier tipo para manejar datos.

El puntero sirve para enlazar cada nodo con el resto de nodos que conforman la lista. De esto podemos deducir que una lista enlazada (lista) es una secuencia de nodos en el que cada nodo esta enlazado o conectado con el siguiente (por medio del puntero mencionado anteriormente). El primer nodo de la lista se denomina cabeza de la lista y el último nodo cola de la lista. Este último nodo suele tener su puntero igualado a NULL Para indicar que es el fin de la lista.

El TAD pila.

Una pila (stack) es una secuencia de cero o más elementos de un mismo tipo, que puede crecer y decrecer por uno de sus extremos (el tope de la pila).

Las pilas se denominan también estructuras LIFO (Last In First Out), porque su característica principal es que el último elemento en llegar es el primero en salir. Son muy utilizadas en programación para evaluar expresiones.

Programa ejemplo en C# que utilice un TDA.

Implementación de pilas a través de TAD predeterminado por C#.
Cree un proyecto modo consola en Visual C# y agregue al inicio el nombre de espacio Collections de la siguiente manera: Using System.Collections;

```
string valor;
Stack mipila = new Stack( );
//Ingreso de elementos a la pila
mipila.Push(“z”);
mipila.Push(“ba”);
mipila.Push(“nom”);
//imprimir elementos de la pila pueden ser de tipo char, int, string o
//cualquier otro tipo ya que es un tipo object dependerá de lo que el
//usuario necesite
foreach (string var in mipila)
{
        Console.WriteLine(var);
}
Console.WriteLine(“\n\n”);
//Peek Retorna el valor que está al tope de la pila sin eliminarlo
Console.WriteLine(“El tope de la pila es”);
Console.WriteLine(“mipila.Peek()”);
//retorna el valor del tope eliminándolo
valor = mipila.Pop().ToString();
Console.WriteLine(“eliminado de la pila” + valor);
Console.WriteLine(“\n\n”);
//mostrando contenido de la pila
foreach (string var in mipila)
{
        Console.WriteLine(var);
}
Console.ReadLine();
```

1.3 - Ejemplos de Estructuras de Datos y TDA's

En el ejemplo siguiente, se inicializa struct con la palabra clave new, se llama al constructor predeterminado sin parámetros y, a continuación, se establecen los miembros de la instancia.

```
public struct Customer
{
    public int ID;
    public string Name;

    public Customer(int customerID, string customerName)
    {
        ID = customerID;
        Name = customerName;
    }
}

class TestCustomer
{
    static void Main()
    {
        Customer c1 = new Customer();  //using the default constructor

        System.Console.WriteLine("Struct values before initialization:");
        System.Console.WriteLine("ID = {0}, Name = {1}", c1.ID, c1.Name);
        System.Console.WriteLine();

        c1.ID = 100;
        c1.Name = "Robert";

        System.Console.WriteLine("Struct values after initialization:");
        System.Console.WriteLine("ID = {0}, Name = {1}", c1.ID, c1.Name);
    }
}
```

Resultado

Cuando se compila y ejecuta el código anterior, el resultado muestra que las variables struct se inicializan de manera predeterminada. La variable int se inicializa en 0 y la variable string se inicializa en una cadena vacía.

```
Struct values before initialization:

ID = 0, Name =


Struct values after initialization:

ID = 100, Name = Robert
```

Ejemplo de Tipo de Datos Abstractos

```
class PilaArreglo
    {
        private Object[] arreglo;
        private int tope;
        private int MAX_ELEM = 100; // maximo numero de elementos en la pila

        public PilaArreglo()
        {
            arreglo = new Object[MAX_ELEM];
            tope = -1; // inicialmente la pila esta vacía
        }

        public void apilar(Object x)
        {
            if (tope + 1 < MAX_ELEM) // si esta llena se produce OVERFLOW
            {
                tope++;
                arreglo[tope] = x;
            }
        }

        public Object desapilar()
        {
            if (!estaVacia()) // si esta vacia se produce UNDERFLOW
            {
                Object x = arreglo[tope];
                tope--;
                return x;
            }
        }

        public Object tope()
        {
            if (!estaVacia()) // si esta vacia es un error
            {
                Object x = arreglo[tope];
                return x;
            }
        }

        public boolean estaVacia()
        {
            if (tope == -1)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
    }
```

1.4.1 - Manejo de Memoria Estática.

Cuál es la memoria estática

Es la que no se puede modificar o ejecutar en momento de ejecución. También se debe conocer con anticipo el tamaño de la estructura que está en ejecución por Ej. Un vector, matriz, cubo etc. Estos ejemplos que con anticipación se deben conocer el tamaño de la estructura. Algunos lenguajes de programación utilizan la palabra static para especificar elementos del programa que deben almacenarse en memoria estática.

Los objetos son creados en ese momento y destruidos al finalizar el programa. Mantienen la misma localización en memoria durante todo el transcurso del programa.

Quienes la utilizan y manejan en C#

- Código del programa.
- Las variables definidas en la sección principal del programa, las cuales pueden solo cambiar su contenido no su tamaño.
- Todas aquellas variables declaradas como estáticas en otras clases o módulos.

Los objetos administrados de ese modo son:

a) Variables static
b) Variables globales
c) Miembros static de clases
d) Literales de cualquier tipo

Ventajas y Desventajas del uso de memoria estática.

Ventajas

- La velocidad de acceso es alta.
- Para retener los datos solo necesita estar energizada.
- Lógica simple.
- Son más fáciles de diseñar.

Desventajas:

- No se puede modificar el tamaño de la estructura en tiempo de ejecución.
- No es óptimo con grandes cantidades de datos.
- Desperdicio de memoria cuando no se utiliza en su totalidad del tamaño v[100] .
- Menor capacidad, debido a que cada celda de almacenamiento requiere más transistores.
- Mayor costo por bit.
- Mayor consumo de Potencia.

Programa ejemplo en C# que utilice memoria estática.

```
public static class ConvertirTemperatura
{
    public static double CelsiusToFahrenheit(string TemperaturaCelsius)
    {
        double celsius = Double.Parse(TemperaturaCelsius);

        double fahrenheit = (celsius * 9 / 5) + 32;

        return fahrenheit;
    }

    public static double FahrenheitToCelsius(string TemperaturaFahrenheit)
    {
        double fahrenheit = Double.Parse(TemperaturaFahrenheit);

        double celsius = (fahrenheit - 32) * 5 / 9;

        return celsius;
    }
}
class TestTemperatureConverter
{
    static void Main()
    {
        Console.WriteLine("Convertidor de temperaturas");
        Console.WriteLine("1. De Celsius a Fahrenheit.");
        Console.WriteLine("2. De Fahrenheit a Celsius.");
        Console.Write("Seleccione una opcion:");

        string selection = Console.ReadLine();
        double F, C = 0;

        switch (selection)
        {
            case "1":
                Console.Write("Ingrese la temperatura en Celsius: ");
                F = ConvertirTemperatura.CelsiusToFahrenheit(Console.ReadLine());
                Console.WriteLine("Temperatura en Fahrenheit: {0:F2}", F);
                break;

            case "2":
                Console.Write("Ingrese la temperatura en Fahrenheit: ");
                C = ConvertirTemperatura.FahrenheitToCelsius(Console.ReadLine());
                Console.WriteLine("Temperatura en Celsius: {0:F2}", C);
                break;

            default:
                Console.WriteLine("Porfavor seleccione el convertidor");
                break;
        }
        Console.WriteLine("Presione cualquier letra para salir.");
        Console.ReadKey();
    }
}
```

1.4.2 - Manejo de Memoria Dinámica

A) CUÁL ES LA MEMORIA DINÁMICA?

Es memoria que se reserva en tiempo de ejecución. Su principal ventaja frente a la estática, es que su tamaño puede variar durante la ejecución del programa. (En C, el programador es encargado de liberar esta memoria cuando no la utilice más). El uso de memoria dinámica es necesario cuando a priori no conocemos el número de datos/elementos a tratar.

B) QUIÉNES LA UTILIZAN Y MANEJAN EN C#?

La utilizan para guardar lo que devuelve la función de reserva de memoria llamada malloc. Esta función tiene como argumento el número de bytes que deseamos y devuelve la dirección del comienzo de la porción de memoria

C) VENTAJAS Y DESVENTAJAS DEL USO DE MEMORIA DINÁMICA.

Ventajas:

- Se puede ir incrementando durante la ejecución del programa. Esto permite, por ejemplo, trabajar con arreglos dinámicos.
- El tamaño que la memoria reserva en tiempo de ejecución puede variar durante la ejecución del programa.

Desventajas:

- Es difícil implementar estructuras de datos complejas como son los tipos recursivos (arboles, grafos, etc.).
- Puede afectar el rendimiento del sistema.
- Busca un bloque de memoria libre y almacena la posición y tamaño de la memoria asignada, de manera que pueda ser liberada más adelante.

D) PROGRAMA EJEMPLO EN C# CONSOLA QUE UTILICE MEMORIA DINÁMICA

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApplication1
{
    class Program
    {
        void imprime_binario(int n)
        {
            if (n >= 2)
            {
                imprime_binario(n / 2);
                Console.Write("{0}", n % 2);
            }
            else
            {
                Console.Write("{0}", n);
            }
        }
        static void Main(string[] args)
        {
            Console.Write("Alimenta un Numero Entero: ");
            int numero = Int32.Parse(Console.ReadLine());

            Program p = new Program();

            Console.Write("\nNumero Entero: {0}, en Codigo Binario: ", numero);
            p.imprime_binario(numero);

            Console.ReadLine();
        }
    }
}
```

1.4.3 - Manejo de Memoria Local.

CUÁL ES LA MEMORIA LOCAL

A partir de C# 7.0, C# admite funciones locales. Las funciones locales son métodos privados de un tipo que están anidados en otro miembro. Solo se pueden llamar desde su miembro contenedor. Las funciones locales se pueden declarar en y llamar desde:

Métodos, especialmente los métodos de iterador y asincrónicos

- Constructores
- Descriptores de acceso de propiedad
- Descriptores de acceso de un evento
- Métodos anónimos
- Expresiones lambda
- Finalizadores
- Otras funciones locales

En cambio, las funciones locales no se pueden declarar dentro de un miembro con forma de expresión.

QUIENES LA UTILIZAN Y MANEJAN EN C#

En la mayoría de los lenguajes de programación, las variables locales son variables automáticas almacenadas directamente en la pila de llamadas. Esto significa que cuando una función recursiva se llama a sí misma, las variables locales recibirán un espacio de direccionamiento de memoria separado en cada instancia de la función. De esta forma, las variables con este alcance se pueden declarar, reescribir y leer sin riesgo de efectos secundarios en procesos fuera del bloque donde se declaran.

VENTAJAS Y DESVENTAJAS DEL USO DE MEMORIA LOCAL.

Ventajas

- Velocidad: almacenar datos en un disco duro externo es más rápido que cargar datos en la nube. Además, tiene un control total de las copias de seguridad y un mejor control sobre quién accede a sus datos.
- Capacidad: una de las principales ventajas del almacenamiento local es la capacidad. Aunque 15 GB es definitivamente una buena cantidad para almacenamiento en la nube, los discos duros pueden proporcionar fácilmente 2 TB.
- Control de seguridad: puede controlar completamente el método de almacenamiento, la autoridad de acceso y el protocolo de seguridad de la información de los datos.
- No depende de Internet: el almacenamiento local no depende de Internet. Una vez que guarde algo localmente y necesite acceder a él, lo encontrará de inmediato.

Desventajas

- Accesibilidad: una de las limitaciones del almacenamiento local es que los datos no son fácilmente accesibles y más difíciles de compartir con otros. Para hacer esto, debe cargarlos en el servidor de alojamiento y luego enviar correos electrónicos o enlaces a los usuarios deseados.
- Costo: el costo del hardware y la infraestructura es alto, agregar espacio y actualizar solo aumentará los costos adicionales.
- Recuperación y respaldo: si su servidor falla, los datos morirán con él. Esta puede ser la principal preocupación de la empresa al transferir datos a la nube. Los datos almacenados localmente son más susceptibles a accidentes como incendios e inundaciones, y el almacenamiento local y las copias de seguridad locales se pierden fácilmente.

Programa ejemplo en C# que utilice memoria local.

```
using System;
using System.IO;

class Example
{
    static void Main()
    {
        string contents = GetText(@"C:\temp", "example.txt");
        Console.WriteLine("Contents of the file:\n" + contents);
    }

    private static string GetText(string path, string filename)
    {
         var sr = File.OpenText(AppendPathSeparator(path) + filename);
         var text = sr.ReadToEnd();
         return text;

         // Declare a local function.
         string AppendPathSeparator(string filepath)
         {
            if (! filepath.EndsWith(@"\"))
               filepath += @"\";

            return filepath;
         }
    }
}
```

1.5.1 - Complejidad en el Tiempo.

Qué es la complejidad en el tiempo

Una medida que suele ser útil conocer es el tiempo de ejecución de un programa en función de N, lo que denominaremos T(N). Esta función se puede medir físicamente (ejecutando el programa, reloj en mano), o calcularse sobre el código contando instrucciones a ejecutar y multiplicando por el tiempo requerido por cada instrucción

Cómo se mide en C#

Una medida que suele ser útil conocer es el tiempo de ejecución de un programa en función de N, lo que denominaremos T(N).

Esta función se puede medir físicamente (ejecutando el programa, reloj en mano), o calcularse sobre el código contando instrucciones a ejecutar y multiplicando por el tiempo requerido por cada instrucción.

Prácticamente todos los programas reales incluyen alguna sentencia condicional, haciendo que las sentencias efectivamente ejecutadas dependan de los datos concretos que se le presenten.

Esto hace que más que un valor T(N) debamos hablar de un rango de valores.

1.5.2 - Complejidad en el Espacio

Qué es la complejidad en el espacio

se refiere a la cantidad de memoria requerida para ejecutar un programa. Usando la complejidad espacial del programa, es posible estimar de antemano cuánta memoria requiere el programa. Además del espacio de almacenamiento y las instrucciones, constantes, variables y datos de entrada utilizados por un programa, un programa necesita algunas unidades de trabajo para operar en los datos y el espacio auxiliar para almacenar cierta información necesaria para el cálculo real.

Cómo se mide en C#

Debemos considerar el proceso del marco de la pila de funciones, por ejemplo, cuando buscamos el quinto número de Fibonacci, esta vez Necesita abrir espacio para almacenar el cuarto número, y luego abrir espacio para almacenar el tercer número. Se mide cuando abre espacio para el segundo y primer número, el tercer número obtiene el resultado y vuelve al cuarto número, El valor del cuarto número se conoce y se devuelve al quinto número. En este proceso, el espacio máximo ocupado es el número de capas menos uno. Como se muestra abajo

1.5.3 - Eficiencia de los Algoritmos.

¿Cómo se mide la eficiencia en un algoritmo?

Una forma de medir la eficiencia de un algoritmo es contar cuántas operaciones necesita para encontrar la respuesta con diferentes tamaños de la entrada.

Uno de los métodos más sobresalientes es el análisis de algoritmos, que permite medir la dificultad inherente de un problema. Los restantes capítulos utilizan con frecuencia la técnica de análisis de algoritmos siempre que estos se diseñan. Esta característica le permitirá comparar algoritmos para la resolución de problemas en términos de eficiencia.

Aunque las máquinas actuales son capaces de ejecutar millones de instrucciones por segundo, la eficiencia permanece como un reto o preocupación a resolver. Con frecuencia, la elección entre algoritmos eficientes e ineficientes pueden mostrar la diferencia entre una solución práctica a un problema y una no práctica.

¿Cuáles son las medidas prácticas para hacerlo?

Existen diferentes métodos con los que se trata de medir la eficiencia de los algoritmos; entre ellos, los que se basan en el número de operaciones que debe efectuar un algoritmo para realizar una tarea; otros métodos se centran en tratar de medir el tiempo que se emplea en llevar a cabo una determinada tarea, ya que lo importante para el usuario final es que ésta se efectúe de forma correcta y en el menor tiempo posible.

Las dos medidas más comunes son:

Complejidad temporal: cuanto se demora un algoritmo en terminar.

Complejidad espacial: cuanta memoria operativa (RAM usualmente) es requerida por el algoritmo. Esto tiene dos apartados, la cantidad de memoria que necesita el código y la cantidad que necesitan los datos sobre los que opera el algoritmo.

Fuentes de Información:

1.1 - Clasificación de las estructuras de datos.

http://tesciedd.blogspot.com/2011/01/unidad-1-clasificacion-de-estructuras.html

http://decsai.ugr.es/~jfv/ed1/c/cdrom/cap7/cap71.htm

https://openwebinars.net/blog/que-son-las-estructuras-de-datos-y-por-que-son-tan-utiles/

1.2 - Tipos de datos abstractos (TDA).

http://www.cartagena99.com/recursos/programacion/apuntes/Tema1TADs.pdf

https://docplayer.es/44195023-Tema-tipos-abstractos-de-datos-tad-s-en-c.html

https://users.dcc.uchile.cl/~bebustos/apuntes/cc30a/TDA/#:~:text=Un%20Tipo%20de%20dato%20abstracto,como%20est%C3%A9n%20implementadas%20dichas%20operaciones.

1.3 - Ejemplos de Estructuras de Datos y TDA's.

http://dominiotic.com/creando-estructuras-en-c/

https://users.dcc.uchile.cl/~bebustos/apuntes/cc30a/TDA/

1.4.1 - Manejo de Memoria Estática.

https://es.slideshare.net/Tell123Say123/memoria-estatica#:~:text=La%20memoria%20est%C3%A1tica%20es%20la%20que%20no%20se%20puede%20modificar,el%20tama%C3%B1o%20de%20la%20estructura.

https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/classes-and-structs/static-classes-and-static-class-members

1.4.2 - Manejo de Memoria Dinámica.

http://juegosprove.blogspot.com/2017/02/memoria-estatica-y-memoria-dinamica.html

http://adrian-estructuradedatos.blogspot.com/2011/04/memoria-estatica-y-dinamica.html

1.4.3 - Manejo de Memoria Local

https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/classes-and-structs/local-functions

https://mydatascope.com/blog/es/almacenamiento-en-la-nube-o-al-almacenamiento-local/

1.5.1 - Complejidad en el Tiempo

https://docs.google.com/document/d/1nE6HlQ7t95SjUIwTCqN9-uOklaPQ-18MyfdxH8mHXMc/edit

https://itslr.edu.mx/archivos2013/TPM/temas/s3u7.html

https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch?view=netcore-3.1

1.5.2 - Complejidad en el espacio

http://rhomarycristobal.blogspot.com/2011/12/septima-unidad-analisis-de-algoritmos.html

http://artemisa.unicauca.edu.co/~nediaz/EDDI/cap01.htm

1.5.3 - Eficiencia de los Algoritmos

https://es.slideshare.net/VanessaRamirez20/eficiencia-de-algoritmos-vanessa-ramrez