Apuntes Estructura de datos.

10 de marzo de 2017

1. Estructura de datos

1.1. Tema 1 - Función de eficiencia.

El proceso que vamos a realiar es extraer la función que marca la eficiencia del código de nuestro programa. Existe una funcion "timer"que me permite ver el tiempo que ha transcurrido en una seccion del codigo que desee.

1.1.1. Notacion O grande.

Se dice que una f(n) = O(g(n)) si a partir de un m, f(n)<= k · g(n), donde k es una constante. Para usar este tipo de comparación podemos utilizar simplificacion de funciones a su estructura básica.

1.1.2. Hallar la función de eficiencia.

Lo primero es dividir el codigo en trozos, y tenemos en cuenta que si:

1. Los troxos son independientes, la función de la union es O del máximo de las funciones de cada uno.
2. Los trozos son anidados o dependientes, la función de la unión es O del producto de las funciones de cada uno.

Teniendo en cuenta que cada accion cuenta como la unidad, es decir:

```
1 for(int i = 0; i < n; i++){
2   cout << "Lmao";
3 }
```

Serian n iteraciones de valor 1. Sabiendo que la iteracion i++ tambien es una operación pero no tiene sentido tenerlo en cuenta ya que 2n = O(n), de igual forma que la asignacióndel principio del bucle, y la comparación que se realiza, es decir, contaremos las operaciones simples como una única.

1.2. Análisis de sentencias

1.2.1. Bucles

1.2.2. Sentencias IF-ELSE

Es una sencencia en la que usamos la regla del máximo, ocurrirá el if o el else y el máximo de los 2 marca la eficiencia del bloque.

Puede ser que la condición no sea una sentencia elemental, entonces habría que tenerla en cuenta, pero si es una sentencia trivial, la sentencia será O(1)

```
if(A[0][0]== 0)
{
  for(i = 0; i < n; i++)
  {
    for(j = 0; j < n; j++)
    A[i][j] = 0;
  }
}
else
{
  for(k = 0; k < n; k++)
  {
    A[k][k] = 1;
  }
}
```

En este caso, tenemos que tomar máximo entre $O(n^2)$ (lo que cuesta el if) y O(n)(lo que cuesta el else). Por ello, la eficiencia del código es $O(n^2)$.

1.2.3. Bloques de Sentencias

En este caso, si tenemos bloques independientes se va tomando la regla del máximo para todos los bloques

1.2.4. Llamadas a funciones

Hay que mirar cuánto cuesta la llamada a la función, es muy importante para la ejecución del código.

Analizamos un ejemplo de un código de 30 líneas de las transparencias del profesor.

1. Ejemplos

   Cuando en los bucles haya i =2, i*=4,i*=n... entonces la eficiencia será logaritmo en base n de lo que haya dentro del bucle.

1.3. Tema 2 - Abstracción.

1.3.1. Uso de Template.

Nos permite seleccionar el tipo de dato que vamos a utilizar en tiempo de ejecución. Declarando:

```
template <class T, int n>

class array_n {
private:
    T items[n];
  };
```

Creando un objeto de esta clase de la forma:

```
array_n<int,1000> w
```

Creando un metodo de la forma:

```
template <class T>
T VectorDinamico::componente(int i) const
{
  return datos[i];
}
```

La compilación a la hora de usar templates es distinta a la que estamos acostumbrados. En lugar de hacer un #include çlase.h"_ en el archivo _.cpp, se incluirá el archivo #include çlase.cpp"_ al final del archivo _.h.

1.3.2. Clase vector dinámico

Vamos a estudiar una clase que tenga un vector de datos y el número de elementos.Un primer ejemplo con tipo de dato float sería:

```
class VectorDinamico{

  float* datos;
  int ns;
}
```

1.3.3. Iteradores

Pretendemos hacer recorridos mucho más rápido. No volvermos a recorrer vectores haciendo v[i]. Usaremos los punteros para iterar, de la forma:

```
double *v = &a.
double *p;
double * fin;
fin = v+n;
for(p = v; p!= fin; ++p)
  cout << *p << endl;
```

Nota: en un compilador moderno, simplemente activando la optimización de código se consigue el mismo aumento en el rendimiento.

Definiremos incluso un nuevo tipo de dato llamado iterator, haciéndolo de la forma:

```
typedef double* iterator;
iterator begin(double* v, int n){
  return v;
}
iterator end(double* v, int n){
  return v+n;
}

/**--------------------------*/
iterator p;
for(p=begin(v,n); p!=end(v,n);++p)
  cout << *p << endl;
```

1.3.4. Pilas

Son estructuras de datos lineales, secuencias de elementos dispuestos en una dimensión. Tienen estructura *LIFO* (last in, first out)

No usaremos el concepto de posición, sólo trabajaremos con el tope de la estructura. En realidad, sí podemos recorrerla pero debemos salvaguardar los elementos , pues para acceder al siguiente elemento tenemos que borrar el anterior.

- Para insertar un elemento, se inserta siempre al principio de la pila, añadiendo físicamente
- Para borrar un elemento, se elimina y el puntero que había al primero se lleva al segundo
- Como la pila no tiene recorridos, no tienen iteradores.
- Las funciones básicas son Tope, Poner, Quitar, Vaciar.

Hay que recordar que los elementos de la pila se guardan en orden contrario al que fueron insertados. Por tanto si los imprimimos, por ser de tipo *LIFO* salen del primero que hemos insertado al último. En la práctica, usaremos las pilas con Celdas Enlazadas.

```
1  #ifndef __PILA_H__
2  #define __PILA_H__
3
4  typedef char Tbase;
5
6  struct CeldaPila{
7      Tbase elemento;
8      CeldaPila *sig;
9  };
10
11 class Pila{
12
13     CeldaPila *primera;
14
15   public:
16
17     Pila();
18     Pila(const Pila& p);
19     ~Pila();
20     Pila& operator= (const Pila& pila);
21
22     void poner(Tbase c);
23     void quitar();
24     Tbase tope() const;
25     bool vacia() const;
26 };
```