Escuela de Ingeniería Civil Informática
Facultad de Ingeniería

Estructuras de datos

Capítulo I: Análisis de algoritmos

Fabián Riquelme Csori

fabian.riquelme@uv.cl

2017-II

Index

Tipos de datos
- Tipos de datos primitivos
- Tipos de datos abstractos

Análisis de algoritmos
- Notación Big O

Tipos de datos

Un tipo de datos es una propiedad o atributo de un conjunto de datos, que determina su dominio de aplicación, qué valores pueden tomar, qué operaciones se les pueden aplicar y cómo son representados internamente por el computador.

Tipos de datos

Un tipo de datos es una propiedad o atributo de un conjunto de datos, que determina su dominio de aplicación, qué valores pueden tomar, qué operaciones se les pueden aplicar y cómo son representados internamente por el computador.

Un tipos de datos primitivo o elemental es un tipo de datos proveído por un lenguaje de programación. Ejemplos:

Tipos de datos

Un tipo de datos es una propiedad o atributo de un conjunto de datos, que determina su dominio de aplicación, qué valores pueden tomar, qué operaciones se les pueden aplicar y cómo son representados internamente por el computador.

Un tipos de datos primitivo o elemental es un tipo de datos proveído por un lenguaje de programación. Ejemplos:

- `char`
- `int`
- `float`
- `bool`
- `string`
- `pointer`

Tipos de datos

Un tipo de datos es una propiedad o atributo de un conjunto de datos, que determina su dominio de aplicación, qué valores pueden tomar, qué operaciones se les pueden aplicar y cómo son representados internamente por el computador.

Un tipos de datos primitivo o elemental es un tipo de datos proveído por un lenguaje de programación. Ejemplos:

- `char`
- `int`
- `float`
- `bool`
- `string`
- `pointer`

Estos tipos de datos primitivos se pueden mezclar para definir tipos de datos compuestos. Por ejemplo, los `struct` en C y C++.

Tipos de datos abstractos

Un tipo de datos abstractos (TDA) es un modelo matemático para tipos de datos. Provee una descripción lógica o una especificación de los componentes del dato y de las operaciones que son permitidas.

Tipos de datos abstractos

Un tipo de datos abstractos (TDA) es un modelo matemático para tipos de datos. Provee una descripción lógica o una especificación de los componentes del dato y de las operaciones que son permitidas.

- Un TDA es indepediente de la implementación.

Tipos de datos abstractos

Un tipo de datos abstractos (TDA) es un modelo matemático para tipos de datos. Provee una descripción lógica o una especificación de los componentes del dato y de las operaciones que son permitidas.

- Un TDA es indepediente de la implementación.
- Un TDA puede implementrse de diversas maneras incluso en un mismo lenguaje.

Tipos de datos abstractos

Un tipo de datos abstractos (TDA) es un modelo matemático para tipos de datos. Provee una descripción lógica o una especificación de los componentes del dato y de las operaciones que son permitidas.

- Un TDA es indepediente de la implementación.
- Un TDA puede implementrse de diversas maneras incluso en un mismo lenguaje.

Ventajas

Tipos de datos abstractos

Un tipo de datos abstractos (TDA) es un modelo matemático para tipos de datos. Provee una descripción lógica o una especificación de los componentes del dato y de las operaciones que son permitidas.

- Un TDA es indepediente de la implementación.
- Un TDA puede implementrse de diversas maneras incluso en un mismo lenguaje.

Ventajas

- Encapsulamiento
- Localización de cambios
- Flexibilidad
- Reusabilidad
- Legibilidad
- ...

Ejemplos de operaciones sobre TDA

- Constructor y destructor (en POO)

Ejemplos de operaciones sobre TDA

- Constructor y destructor (en POO)
- Iteradores: para recorrer el dato.

Ejemplos de operaciones sobre TDA

- Constructor y destructor (en POO)
- Iteradores: para recorrer el dato.
- Operadores de capacidad: determinar, verificar o modificar la capacidad de almacenamiento del dato.

Ejemplos de operaciones sobre TDA

- Constructor y destructor (en POO)
- Iteradores: para recorrer el dato.
- Operadores de capacidad: determinar, verificar o modificar la capacidad de almacenamiento del dato.
- Operadores de acceso: maneras de acceder a los elementos del dato.

Ejemplos de operaciones sobre TDA

- Constructor y destructor (en POO)
- Iteradores: para recorrer el dato.
- Operadores de capacidad: determinar, verificar o modificar la capacidad de almacenamiento del dato.
- Operadores de acceso: maneras de acceder a los elementos del dato.
- Modificadores: maneras de modificar el contenido de los elementos del dato.

Ejemplos de operaciones sobre TDA

- Constructor y destructor (en POO)
- Iteradores: para recorrer el dato.
- Operadores de capacidad: determinar, verificar o modificar la capacidad de almacenamiento del dato.
- Operadores de acceso: maneras de acceder a los elementos del dato.
- Modificadores: maneras de modificar el contenido de los elementos del dato.
- Comparadores: operadores (en general binarios) para relacionar datos del mismo TDA.

Ejemplos de operaciones sobre TDA

- Constructor y destructor (en POO)
- Iteradores: para recorrer el dato.
- Operadores de capacidad: determinar, verificar o modificar la capacidad de almacenamiento del dato.
- Operadores de acceso: maneras de acceder a los elementos del dato.
- Modificadores: maneras de modificar el contenido de los elementos del dato.
- Comparadores: operadores (en general binarios) para relacionar datos del mismo TDA.
- Otros: swaps, replicadores, etc.

Ejemplos de operaciones sobre TDA

- Constructor y destructor (en POO)
- Iteradores: para recorrer el dato.
- Operadores de capacidad: determinar, verificar o modificar la capacidad de almacenamiento del dato.
- Operadores de acceso: maneras de acceder a los elementos del dato.
- Modificadores: maneras de modificar el contenido de los elementos del dato.
- Comparadores: operadores (en general binarios) para relacionar datos del mismo TDA.
- Otros: swaps, replicadores, etc.

Ejemplo: `http://www.cplusplus.com/reference/vector/`

Eficiencia y complejidad

- Se invierte mucho trabajo en mejorar la eficiencia de los TDA. De ello normalmente depende la eficiencia de todo el sistema.

Eficiencia y complejidad

- Se invierte mucho trabajo en mejorar la eficiencia de los TDA. De ello normalmente depende la eficiencia de todo el sistema.
- Pero ¿qué es la eficiencia?

Eficiencia y complejidad

- Se invierte mucho trabajo en mejorar la eficiencia de los TDA. De ello normalmente depende la eficiencia de todo el sistema.
- Pero ¿qué es la eficiencia?
- Para medir tiempo de ejecución en bash desde la shell de Linux, tenemos el comando `time`:

```
real 0m0.037s
user 0m0.004s
sys 0m0.008s
```

Eficiencia y complejidad

- Se invierte mucho trabajo en mejorar la eficiencia de los TDA. De ello normalmente depende la eficiencia de todo el sistema.
- Pero ¿qué es la eficiencia?
- Para medir tiempo de ejecución en bash desde la shell de Linux, tenemos el comando `time`:

```
real 0m0.037s
user 0m0.004s
sys 0m0.008s
```

  - `real` es el tiempo total transcurrido para ejecutar la aplicación. Incluye otros procesos que estaban en ejecución.

Eficiencia y complejidad

- Se invierte mucho trabajo en mejorar la eficiencia de los TDA. De ello normalmente depende la eficiencia de todo el sistema.
- Pero ¿qué es la eficiencia?
- Para medir tiempo de ejecución en bash desde la shell de Linux, tenemos el comando `time`:

```
real 0m0.037s
user 0m0.004s
sys 0m0.008s
```

  - `real` es el tiempo total transcurrido para ejecutar la aplicación. Incluye otros procesos que estaban en ejecución.
  - `user` es el tiempo de CPU del proceso concreto que se ejecutó. Se excluye otros procesos y retardo de disco.

Eficiencia y complejidad

- Se invierte mucho trabajo en mejorar la eficiencia de los TDA. De ello normalmente depende la eficiencia de todo el sistema.
- Pero ¿qué es la eficiencia?
- Para medir tiempo de ejecución en bash desde la shell de Linux, tenemos el comando `time`:

```
real 0m0.037s
user 0m0.004s
sys 0m0.008s
```

  - `real` es el tiempo total transcurrido para ejecutar la aplicación. Incluye otros procesos que estaban en ejecución.
  - `user` es el tiempo de CPU del proceso concreto que se ejecutó. Se excluye otros procesos y retardo de disco.
  - `sys` es el tiempo de CPU en llamadas al sistema del proceso.

Eficiencia y complejidad

- Se invierte mucho trabajo en mejorar la eficiencia de los TDA. De ello normalmente depende la eficiencia de todo el sistema.
- Pero ¿qué es la eficiencia?
- Para medir tiempo de ejecución en bash desde la shell de Linux, tenemos el comando `time`:

```
real 0m0.037s
user 0m0.004s
sys 0m0.008s
```

  - `real` es el tiempo total transcurrido para ejecutar la aplicación. Incluye otros procesos que estaban en ejecución.
  - `user` es el tiempo de CPU del proceso concreto que se ejecutó. Se excluye otros procesos y retardo de disco.
  - `sys` es el tiempo de CPU en llamadas al sistema del proceso.
  - Si no hubieran otros procesos corriendo y la lectura de disco fuera inmediata, tendríamos `user`+`sys`=`real`.

- ¿Podemos hablar de eficiencia independientemente de los lenguajes de programación y de las máquinas?

- ¿Podemos hablar de eficiencia independientemente de los lenguajes de programación y de las máquinas?
- Complejidad temporal vs. complejidad espacial
  - ¿Recuerdan las tablas de verdad de la lógica proposicional? ¿Conocen el problema SAT?

Esfuerzo computacional

- Dada una entrada w para un algoritmo, definimos una función $T : \mathbb{N} \to \mathbb{N}$ tal que $T(|w|)$ es el tiempo o número de pasos que dicho algoritmo tarda en computar w, en función de su tamaño $|w|$.

Esfuerzo computacional

- Dada una entrada w para un algoritmo, definimos una función $T : \mathbb{N} \to \mathbb{N}$ tal que $T(|w|)$ es el tiempo o número de pasos que dicho algoritmo tarda en computar w, en función de su tamaño $|w|$.
- Dados dos algoritmos A_1 y A_2 que resuelven un mismo problema para una misma entrada w.
  Sea $|w| = n$, supongamos:

$$T_1(n) = n^3 \quad \text{y} \quad T_2(n) = 6\,T_1(n)^2 + 3n + 15$$

Esfuerzo computacional

- Dada una entrada w para un algoritmo, definimos una función $T : \mathbb{N} \to \mathbb{N}$ tal que $T(|w|)$ es el tiempo o número de pasos que dicho algoritmo tarda en computar w, en función de su tamaño $|w|$.
- Dados dos algoritmos A_1 y A_2 que resuelven un mismo problema para una misma entrada w.
  Sea $|w| = n$, supongamos:

$$T_1(n) = n^3 \quad \text{y} \quad T_2(n) = 6\,T_1(n)^2 + 3n + 15$$

- Ambos algoritmos poseen un tiempo de ejecución polinomial.

Esfuerzo computacional

- Dada una entrada w para un algoritmo, definimos una función $T : \mathbb{N} \to \mathbb{N}$ tal que $T(|w|)$ es el tiempo o número de pasos que dicho algoritmo tarda en computar w, en función de su tamaño $|w|$.
- Dados dos algoritmos A_1 y A_2 que resuelven un mismo problema para una misma entrada w.
  Sea $|w| = n$, supongamos:

$$T_1(n) = n^3 \quad \text{y} \quad T_2(n) = 6\,T_1(n)^2 + 3n + 15$$

- Ambos algoritmos poseen un tiempo de ejecución polinomial.
- No nos interesa tanto las diferencias "sutiles" entre T_1 y T_2, sino saber que ambos tiempos son polinomiales (i.e., nos concentramos en las tasas de crecimiento).

Funciones superiormente acotadas: O

- Todo polinomio está superiormente acotado por otro polinomio.

Funciones superiormente acotadas: O

- Todo polinomio está superiormente acotado por otro polinomio.
- La tasa de crecimiento de cualquier polinomio puede representarse simplemente por su grado.

Funciones superiormente acotadas: O

- Todo polinomio está superiormente acotado por otro polinomio.
- La tasa de crecimiento de cualquier polinomio puede representarse simplemente por su grado.
- Ej: Si el tiempo de complejidad de un algoritmo es $T(n) = 3n^2 + 6n$, decimos que es del orden n^2, y usamos la notación $O(n^2)$, tal que $T(n) = O(n^2)$.

Funciones superiormente acotadas: O

- Todo polinomio está superiormente acotado por otro polinomio.
- La tasa de crecimiento de cualquier polinomio puede representarse simplemente por su grado.
- Ej: Si el tiempo de complejidad de un algoritmo es $T(n) = 3n^2 + 6n$, decimos que es del orden n^2, y usamos la notación $O(n^2)$, tal que $T(n) = O(n^2)$.
- Formalmente,

$$f(x) = O(g(x))$$

si existen k, x_0 reales positivos tal que

$$|f(x)| \leq k\,|g(x)|\ \text{, para todo } x \geq x_0$$

Algunas propiedades de Big O

Adición

- Si $f_1 = O(g_1)$ y $f_2 = O(g_2)$, entonces $f_1 + f_2 = O(|g_1| + |g_2|)$

Algunas propiedades de Big O

Adición

- Si $f_1 = O(g_1)$ y $f_2 = O(g_2)$, entonces $f_1 + f_2 = O(|g_1| + |g_2|)$
- Si f y g son funciones positivas, $f + O(g) = O(f + g)$

Algunas propiedades de Big O

Adición

- Si $f_1 = O(g_1)$ y $f_2 = O(g_2)$, entonces $f_1 + f_2 = O(|g_1| + |g_2|)$
- Si f y g son funciones positivas, $f + O(g) = O(f + g)$

Multiplicación

- Si $f_1 = O(g_1)$ y $f_2 = O(g_2)$, entonces $f_1 f_2 = O(g_1 g_2)$

Algunas propiedades de Big O

Adición

- Si $f_1 = O(g_1)$ y $f_2 = O(g_2)$, entonces $f_1 + f_2 = O(|g_1| + |g_2|)$
- Si f y g son funciones positivas, $f + O(g) = O(f + g)$

Multiplicación

- Si $f_1 = O(g_1)$ y $f_2 = O(g_2)$, entonces $f_1 f_2 = O(g_1 g_2)$
- $f \cdot O(g) = O(fg)$

Algunas propiedades de Big O

Adición

- Si $f_1 = O(g_1)$ y $f_2 = O(g_2)$, entonces $f_1 + f_2 = O(|g_1| + |g_2|)$
- Si f y g son funciones positivas, $f + O(g) = O(f + g)$

Multiplicación

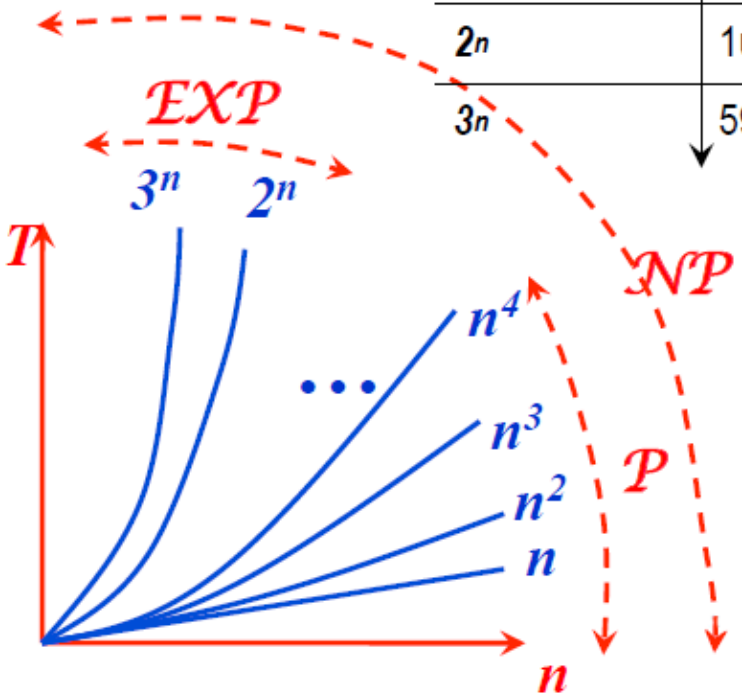
- Si $f_1 = O(g_1)$ y $f_2 = O(g_2)$, entonces $f_1 f_2 = O(g_1 g_2)$
- $f \cdot O(g) = O(fg)$
- $O(kg) = O(g)$, para toda constante $k > 0$.

Cotas usuales

- $O(1)$ es un crecimiento constante
- $O(\log n)$ es un crecimiento logarítmico
- $O(n)$ es un crecimiento lineal
- $O(n^2)$ es un crecimiento cuadrático
- $O(n^c)$ es un crecimiento polinomial
- $O(c^n)$, $c > 1$ es un crecimiento exponencial
- $O(n!)$ es un crecimiento factorial

Algoritmos polinomiales vs. exponenciales

Complejidad temporal	10	20	30	40	50
n	10ns	20ns	30ns	40ns	50ns
n^2	100ns	400ns	900ns	1,6us	2,5us
n^3	1us	8us	27us	64us	125us
2^n	1us	1ms	1s	18min	13días
3^n	59us	3,48s	2,3días	385años	$2{,}2 \times 10^7$ años



mayor tamaño de entrada resoluble en 1 hora:

tiempo	computador de hoy	100 veces más rápido	1000 veces más rápido
m	$N_1 = 24 \cdot 10^{11}$	$100N_1$	$1000N_1$
m^2	$N_2 = 1550000$	$10N_2$	$31{,}6N_2$
m^3	$N_3 = 13200$	$4{,}64N_3$	$10N_3$
m^5	$N_4 = 300$	$2{,}5N_4$	$3{,}98N_4$
2^m	$N_5 = 41{,}1$	N_5+6,64	N_5+9,97
3^m	$N_6 = 25{,}9$	N_6+4,19	N_6+6,29

Valorizando de forma General.

- El análisis de programas sencillos se puede hacer contando los bucles anidados que contiene el programa. Un sólo bucle sobre n ítems genera f(n)=n. Un bucle dentro de otro bucle f(n) = n2. Un bucle dentro de un bucle que está dentro de otro bucle genera f(n) = n3.
- Dado un conjunto de bucles que son secuenciales, el más lento de ellos determina el comportamiento asintótico del programa. Dos bucles anidados, seguidos por un solo bucle, asintóticamente es lo mismo que los bucles anidados por sí solos, ya que los bucles anidados dominan el bucle individual.

Valorizando de forma General.

- Regla General: Dado un conjunto de bucles que son secuenciales, el más lento de ellos determina el comportamiento asintótico del programa. Dos bucles anidados, seguidos por un solo bucle, asintóticamente es lo mismo que los bucles anidados por sí solos, ya que los bucles anidados dominan el bucle individual.
- Regla general: Programas con un mayor corren más lentamente que programas con un menor.

Valorizando de forma General.

- Este filtro de eliminar todos los factores y de mantener el término de mayor crecimiento, como describimos anteriormente, es lo que denominamos asymptotic behavior. Entonces el comportamiento asintótico de f(n) = 2n + 8 es descrito por la función f(n) = n.

Ejemplo.

```
[1] var M = A[ 0 ];
[2] for ( var i = 0; i < n; ++i ) {
[3]     if ( A[ i ] >= M ) {
[4]         M = A[ i ];
[5]     }
[6] }
```

Solución

1. Contar cuantas instrucciones fundamentales ejecuta este trozo de código
2. En [1], requiere 2 instrucciones: una para buscar A[0]y otra para asignar el valor a M (bajo supuesto que n siempre es al menos 1).
3. En [2], el código de iniciación de bucle o for loop también tiene que correr siempre. Esto nos da dos instrucciones más: asignación i=0 y una comparación i<n:
4. En [2], Después de cada iteración for loop, se ejecutan dos instrucciones más: un incremento de i y una comparación para chequear si nos mantenemos en el bucle: ++i; i<n;
5. Entonces hasta el momento sin incluir el interior del for se tiene: $\mathbf{f(\ n\ ) = 4 + 2n}$
6. Ahora en [3], tenemos una operación de búsqueda en array y una comparación que ocurre siempre A[i] >= M

Solución

7. Debido a que esto se encuentra dentro de un loop, ocurrirá n veces centrando siempre el análisis en el peor caso 4n.
8. Entonces: **f(n) = 4 + 2n + 4n = 6n + 4**
9. Finalmente por Comportamiento Asintótico se tiene: **f(n) = n**

Ejemplo 2.

```
bool duplicate = false;
for ( int i = 0; i < n; ++i ) {
    for ( int j = 0; j < n; ++j ) {
        if ( i != j && A[ i ] == A[ j ] ) {
            duplicate = true;
            break;
        }
    }
    if ( duplicate ) {
    break;
    }
}
```

Solución

1. Como aquí se tienen dos bucles anidados entre sí, se tiene un comportamiento asintótico descrito por f(n) = n^2

Ejemplo 3.

```
def binarySearch( A, n, value ):
if n = 1:
if A[ 0 ] = value:
        return true
    else:
        return false
    if value < A[ n / 2 ]:
        return binarySearch( A[ 0...( n / 2 - 1 ) ]
    else if value > A[ n / 2 ]:
        return binarySearch( A[ ( n / 2 + 1 )...n ]
    else:
return true
```

Solución

1. Asumamos que el array tiene un tamaño que es una potencia exacta de 2.
2. El peor escenario para este problema podría ocurrir cuando el valor que buscamos no existe en nuestro array.
3. En general, el array se divide por la mitad en cada llamado, hasta que llegamos a 1, entonces:

   0th iteration: n
   1^{st} iteration: n / 2
   2^{nd} iteration: n / 4
   last iteration: 1

Solución

- Ahora se aplica función log. Resolviendo para i, se obtiene: i= log(n). de la sisiguiente forma:
  - $2i = n$
  - $i = \log(\ n\ )$

Bibliografía

- Weiss, M., Estructura de datos y algoritmos, Addison-Wesley, 1995.
- Aho, Hopcroft y Ullman, Estructuras de datos y algoritmos, Addison-Wesley, 1988.

Recursos

- Wikipedia y Wikimedia Commons.