Manejo de estructuras de datos en C++

Martín Knoblauch Revuelta
http://www.mkrevuelta.com @mkrevuelta m.knoblauch@indizen.com

Leganés, 11 de Febrero de 2016

Índice

Introducción

¿Por qué C++?

1. Velocidad
2. Memoria
3. Diseño
4. Escalabilidad

En resumen

C++ permite exprimir la potencia de los ordenadores al máximo

Complejidad computacional

- Caracteriza tiempo y/o memoria necesarios frente al crecimiento del problema
- Los factores constantes juegan un papel secundario...
- ...dentro de unos límites prácticos razonables
- La notación $O(f(N))$ se utiliza para clasificar los algoritmos o programas

$O(1)$

Esto tarda $O(1)$

```
x = sqrt (y);
```

También $O(1)$

(*) Con un factor constante alto

```
for (int i=0; i<42; ++i)
    x[i] = sqrt (y[i]);
```

$O(1)$

Una función que tarda $O(1)$

```
double elem_central (const double A[],
                     unsigned num)
{
    return A[num/2];
}
```

$O(1)$

(*) Factor constante **muy** alto

```
double suma_aleatoria (const double A[],
                       unsigned num)
{
    unsigned i;
    double suma;

    for (i=0, suma=0; i<(1U<<30); ++i)
        suma += A[rand()%num];

    return suma;
}
```

$O(N)$

Esto **sí** tarda $O(N)$

```
for (int i=0; i<N; ++i)
    x[i] = sqrt (y[i]);
```

- Consideramos que **N** es variable
- **N** define el tamaño del problema
- Puede haber otras variables...

$O(N^2)$

```
unsigned fii (const double A[],
              unsigned num)
{
    unsigned i, j, c;

    for (i=c=0; i<num; ++i)
        for (j=0; j<num; ++j)
            if (i<j && A[i]>A[j])
                ++ c;
    return c;
}
```

$O(N^2)$ La mitad de iteraciones, pero...

```
unsigned foo (const double A[],
              unsigned num)
{
    unsigned i, j, c;

    for (i=c=0; i<num; ++i)
        for (j=i+1; j<num; ++j)
            if (A[i]>A[j])
                ++ c;
    return c;
}
```

Tiempo polinómico en general

$O(N^3)$

```
for (i=0; i<num; ++i)
    for (j=0; j<num; ++j)
        for (k=0; k<num; ++k)
            // ... hacer algo O(1) aquí ...
```

$O(N^4)$, $O(N^5)$, $O(N^6)$...

4 bucles anidados, 5 bucles, 6...

$O(N)$ ¡Aunque no lo parece!

```
void faa (double A[],
          unsigned num)
{                                   //Ojo: --num
    unsigned i, j;

    for (i=0; i<num; ++i)
        for (j=i; j<num && A[j]!=3; ++j)
            A[--num] *= 2;
}
```

$O(\log N)$

```
double suma_pot_3 (const double A[],
                   unsigned num)
{
    unsigned i;
    double suma;

    for (i=1, suma=0; i<num; i*=3)
        suma += A[i];

    return suma;  //La base en O(log N)
}                 //es un factor cte.
```

$O(N \log N)$

```
unsigned fee (const double A[],
              unsigned num)
{
    unsigned i, c, m;

    for (c=0, m=num; m; m>>=1)
        for (i=0; i<num; ++i)
            if (i!=m-1 && A[i]==A[m-1])
                ++ c;
    return c;
}
```

Casos

Se caracteriza por separado

1. Peor caso
2. Caso medio
3. Mejor caso

Ejemplo: quicksort vs. heapsort vs. smoothsort

Amortización

A veces se habla de tiempo **amortizado**

- Es un tiempo medio
- Los casos malos se dan a un ritmo conocido
- Su efecto adverso es absorbido por los casos buenos

Contenedores estándar

Clases principales

- `std::vector`
- `std::deque`
- `std::list`
- `std::map`
- `std::set`
- `std::multi*`
- `std::unordered_*`

Otros elementos

- Iteradores
- Adaptadores
- Comparadores
- Allocators *¿"Reservadores"?*

`std::vector`

- Memoria contigua
- Acceso aleatorio en tiempo $O(1)$
- Inserción en tiempo $O(N)$ (en el caso general)
- Tiempo $O(1)$ **amortizado** al añadir/borrar **al final** (en el peor caso es $O(N)$)
- `size()`, `reserve()`, `capacity()`

std::vector

- Agranda automáticamente la memoria reservada (copiando o moviendo[C++11])
- ¡No la encoge automáticamente!
- `shrink_to_fit ()`[C++11]

Validez de iteradores

Se invalidan todos cada vez que crece la capacidad

std::vector

- Iteradores muy pequeños: sólo 1 puntero
- Vector vacío (gcc): 3 punteros a null
  - `start`
  - `finish`
  - `end_of_storage`

std::vector

Java

- **Vector**
  https://docs.oracle.com/javase/8/docs/api/java/util/Vector.html
  "[...] as components are added to the vector, the vector's storage increases in chunks the size of capacityIncrement."
- **ArrayList**
  https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html
  "[...] (This class is roughly equivalent to Vector, except that it is unsynchronized.) [...] As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost."

`std::deque`

- Bloques (gcc: de al menos 512 bytes) mantenidos con un array de punteros
- Acceso aleatorio en tiempo $O(1)$, aunque no tan rápido como `std::vector`
- Inserción en tiempo $O(N)$ (en el caso general)
- Tiempo casi $O(1)$ al añadir/borrar al principio o al final (para ser precisos, es $O(1)$ **amortizado** pero con un factor cte. pequeño)

std::deque

- Estructura principal e iteradores más pesados que los de `std::vector`
- (gcc: Siempre reserva al menos un bloque, aunque el `std::deque` esté vacío)
- (VS2012: No)

std::deque

Java

- **ArrayDeque**
  https://docs.oracle.com/javase/8/docs/api/java/util/ArrayDeque.html
  "[...] This class is likely to be faster than Stack when used as a stack, and faster than LinkedList when used as a queue."

- Ojo: No ofrece acceso aleatorio

`std::array`

- Como un array convencional: tamaño fijo
- Compatible con la interfaz de `std::vector`
  - Iteradores: `begin()`, `end()`
  - Elementos en extremos: `front()`, `back()`

Así se le puede aplicar código que en principio estaba pensado para `std::vector`

`std::stack` y `std::queue`

- Pila y cola, respectivamente
- Push y pop tardan casi $O(1)$
- Basadas en `std::deque` por defecto (de ahí el "casi")
- Ocupan exactamente lo que ocupe el contenedor encapsulado

std::stack y std::queue

Se puede usar `std::vector` en vez de `std::deque`.
Por ejemplo:

```
std::stack<int,std::vector<int>>
```

(pero entonces push y pop tardan $O(1)$ **amortizado**)

Se puede usar `std::list`... Por ejemplo:

```
std::queue<int,std::list<int>>
```

(pero entonces ocupa más memoria y es más lento)

std::priority_queue

- Heap (montón) dispuesto en forma de array
- Push y pop tardan $O(\log N)$ **amortizado**
- Basada en `std::vector` por defecto (internamente usa mucho acceso aleatorio)

Se puede usar `std::deque` en vez de `std::vector`. Por ejemplo:

```
std::priority_queue<int,std::deque<int>>
```

(pero entonces es un poco más lento en término medio)

std::priority_queue

- El comparador es parametrizable
- Ocupa lo que ocupe el contenedor encapsulado más un puntero al comparador

`std::list`

- Lista doblemente enlazada
- Inserción/extracción (también dentro de la secuencia) en tiempo $O(1)$
- Magnífica validez de iteradores: se invalidan individualmente al extraer el elemento apuntado
- Ocupa más memoria que `std::vector`
- En una competición "justa" (sólo añadir al final), es más lenta que `std::vector`
  - Hace una reserva de memoria dinámica por elemento
  - **Mind the cache!**

std::list

C++11 exige que size() tarde $O(1)$

Pero... en gcc ha sido $O(N)$ mucho tiempo y parece que se mantiene así para no romper la compatibilidad binaria. En gcc 4.8.4 y 4.9.0:

```
/**  Returns the number of elements in the %list.  */
size_type
size() const _GLIBCXX_NOEXCEPT
{ return std::distance(begin(), end()); }
```

- A cambio, se puede cortar y pegar trozos de lista (`splice()`) en tiempo $O(1)$

std::list

Java

- **LinkedList**
  `https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html`
  "Doubly-linked list implementation of the List and Deque interfaces. [...] All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index."
  ¿Hay algo parecido a `splice()`?

`std::forward_list` C++11

- Lista [simplemente] enlazada
- No mantiene un puntero al último elemento
- No proporciona: [..._]`back()`, `size()`
- Sí proporciona:
  - `front()`, `push_front()`, `pop_front()`
  - `insert/emplace_after()` C++11, `erase_after()`

`forward_list` C++11

¿Se pueden crear `std::stack` y `std::queue` basados en `std::forward_list`?

1. No es trivial porque usan `push_back()` y `pop_back()`
2. Para `std::queue` faltaría además un puntero al último elemento
3. Ocuparía más memoria y sería más lento

forward_list C++11

Java

- ¿Hay alguna lista simplemente enlazada en Java?
- ¿Tiene sentido?

`forward_list` C++11

Entonces... ¿Cuándo usar listas?

- ¡Sólo cuando vayamos a insertar/extraer muchas veces dentro de la secuencia!
- ...pero **no** vayamos a usar acceso aleatorio

std::map

- Árbol de búsqueda autobalanceado (típicamente rojo-negro)
- Elemento: { Clave, Valor }
- Operador [] muy cómodo:

```
mapa["foo"] = 32.5;
```

- Búsqueda (por clave) en tiempo $O(\log N)$
- Inserción y extracción en tiempo $O(\log N)$

std::map

- El valor se puede cambiar (tiempo $O(1)$ en lo que respecta al `std::map`)
- La clave no se puede cambiar
- Elementos ordenados según la clave
- Comparador parametrizable (podemos especificar cuál es ese orden)
- Siguiente/anterior en $O(1)$ **amortizado** (el peor caso es $O(\log N)$)

std::map

- El iterador apunta a un std::pair<clave,valor> (uso: it->first, it->second)
- Magnífica validez de iteradores: (como en las listas)
- Construcción copia a partir de otra secuencia en $O(N)$ si está ordenada (si no, en $O(NlogN)$)

std::map

Java

- **TreeMap**
  https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html
  "A Red-Black tree based NavigableMap implementation. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.
  This implementation provides guaranteed log(n) time cost for the containsKey, get, put and remove operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's Introduction to Algorithms."

std::multimap

- Tolera claves repetidas
- No hay operador [] (sería horriblemente ambiguo)
- Búsquedas (cuando varios elementos tienen la misma clave):
  - `find()` iterador a cualquiera de ellos
  - `lower_bound()` iterador al primero
  - `upper_bound()` iterador **posterior** al último
  - `equal_range()` ambos en un `std::pair<iterador,iterador>`

std::set

- Sólo claves
- Una clave está o no está, pero no tiene ningún otro valor asociado

std::set

Java

- **TreeSet**

  http://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html

  "A NavigableSet implementation based on a TreeMap. The elements are ordered using their natural ordering, or by a Comparator provided at set creation time, depending on which constructor is used.
  This implementation provides guaranteed log(n) time cost for the basic operations (add, remove and contains)."

std::multiset

- Tolera claves repetidas (*)
- `find()`, `lower_bound()`, `upper_bound()`, `equal_range()`... como `std::multimap`

No. No es lo mismo que un `std::map<T,unsigned>`.
Léase la letra pequeña:
(*): claves “repetidas” significa “iguales” o “equivalentes” según la función de comparación escogida

Algunos detalles

- Los rangos se especifican como:
  [desde, hasta) $\Leftarrow$ ¡Intervalo semicerrado!
- Los iteradores y el valor especial `end()` se pueden implementar de varias maneras diferentes
- Comparación por defecto vs. comparación a medida $\rightarrow$ tiempo

Hay mucho más

Tablas hash

- `std::unordered_map`
- `std::unordered_multimap`
- `std::unordered_set`
- `std::unordered_multiset`

Otros temas

- Algoritmos
- “Allocators”
- Punteros inteligentes
- Semántica de movimiento

Boost

- Multi-index
- Flyweight
- The Boost Graph Library (BGL)
- Heap (`binomial_heap`, `d_ary_heap`, `fibonacci_heap`)
- MultiArray (Arrays de varias dimensiones. Interesante: vistas)
- ...

Muchas gracias

¿Preguntas?