Algoritmos y estructuras de datos

Memoria dinámica y estructuras

Francisco Javier Zaragoza Martínez

Universidad Autónoma Metropolitana Unidad Azcapotzalco
Departamento de Sistemas

9 de agosto de 2024

Citas

Rob Pike

Si has escogido las estructuras de datos correctas y organizado las cosas bien, los algoritmos serán casi siempre evidentes.

Alan J. Perlis

Es mejor tener cien funciones que operen en una estructura de datos que diez funciones que operen en diez estructuras de datos.

Douglas Crockford

Generalmente, el arte de la programación es la factorización de un conjunto de requerimientos en un conjunto de funciones y estructuras de datos.

Cadenas

1. Una cadena se almacena en un arreglo de caracteres `char s[]`.
2. Una cadena debe terminar siempre con el caracter nulo `'\0'`.
3. Hay diversas formas de inicializar, leer y escribir cadenas.
4. Un arreglo de cadenas se almacena en un arreglo de apuntadores.

Operaciones de cadenas

1. Longitud de una cadena (`int strlen(char *s)`).
2. Copia de cadenas (`char* strcpy(char *s, char *t)`).
3. Concatenación de cadenas (`char* strcat(char *s, char *t)`).
4. Comparación de cadenas (`int strcmp(char *s, char *t)`).

Problema

Longitud de la cadena destino

Dos de las funciones de cadenas que vimos comparten un problema: cuando llamamos a `strcpy` o a `strcat` debemos garantizar que la cadena destino es suficiente larga.

Otra operación con el mismo problema

Considere una función cuyo propósito sea crear un duplicado de una cadena (`char *strdup(char *s)`).

Memoria estática y dinámica

La memoria estática se pide en tiempo de compilación, mientras que la memoria dinámica se pide en tiempo de ejecución.

Ventajas y desventajas

Con memoria dinámica podemos pedir exactamente la memoria que necesitemos (ni más ni menos) pero debemos administrarla nosotros (en particular, debemos avisar cuando ya no la necesitemos).

Las funciones de administración de la memoria dinámica se pueden usar agregando `#include <stdlib.h>` a tu programa.

Solicitud de memoria

La función `void *malloc(int n)` sirve para solicitar `n` bytes consecutivos. Regresa un apuntador al primero de esos bytes o `NULL` si no hay memoria suficiente. Esto se usa en combinación con `sizeof` que da el número de bytes que necesita una variable de tipo arbitrario para almacenarse. La función `void *calloc(int n, int t)` sirve para solicitar un arreglo lleno de ceros de `n` componentes de tamaño `t`.

Liberación de memoria

La función `void free(void *p)` sirve para liberar la memoria solicitada.

Ejemplo

Arreglos de longitud arbitraria

Para pedir un arreglo de enteros de longitud arbitraria podemos hacer:

```
int n;  // escoge un valor de n
int *a; // apuntador al arreglo
scanf("%d", &n);
a = (int *) malloc(n*sizeof(int)); // pide el arreglo
//  (int *) calloc(n, sizeof(int))  lo llena de ceros
if (a != NULL) {
  // todo bien, usa a[0]...a[n-1]
} else {
  // no hubo memoria suficiente
}
free(a);                          // libera el arreglo
```

Observa que `malloc` y `calloc` regresan un `void *` y lo convertimos a un `int *`.

Ejemplo

Duplicar una cadena

```
char *duplica(char *s) {
  char *p = (char *) malloc(longitud(s)+1); // bytes necesarios
  if (p != NULL)                            // si hubo memoria
    copia(p, s);                            // copia s en p
  return p;
}
```

1. Escribe una función `char *invierte(char *s)` que invierta la cadena `s` en una cadena nueva y regrese un apuntador a ella.
2. Escribe una función `char *concatena(char *s, char *t)` que concatene las cadenas `s` y `t` en una cadena nueva y regrese un apuntador a ella.
3. Escribe una función `char *multiplica(char *s, int n)` que concatene `n` copias de la cadena `s` en una cadena nueva y regrese un apuntador a ella.

Estructuras

Una estructura agrupa una o más variables (del mismo o varios tipos) bajo un solo nombre. Las estructuras sirven en particular para organizar datos complicados ya que permiten que un grupo de variables relacionadas se les trate como una unidad.

Ejemplo

- Un punto tiene dos coordenadas enteras.
- Un triángulo tiene tres vértices que son puntos.
- Un número complejo está formado por dos números reales.
- Un racional tiene un numerador y un denominador.
- Un monomio tiene un coeficiente, un exponente y una incógnita.
- Una fecha consta de un año, un mes y un día.

Ejemplo

Puntos de dos dimensiones

Un punto tiene dos coordenadas enteras x y y. Esto lo podemos lograr así:

```
struct punto { // nombre de la estructura
  int x;       // coordenada x
  int y;       // coordenada y
} p, q;        // dos puntos
```

Esto define dos puntos p y q, cada uno con coordenadas x y y:

```
p.x
p.y
q.x
q.y
```

El operador . da acceso a los miembros de la estructura.

Operaciones con estructuras

Operaciones válidas con estructuras

1. Una estructura se puede inicializar `struct punto p = {10, 20}`.
2. Una estructura se puede copiar con `p = q`.
3. La dirección de inicio de la estructura `p` es `&p`.
4. Se puede tener acceso a los miembros de `p` con `p.x` y `p.y`.

Operación inválida con estructuras

Las estructuras no se pueden comparar. Es ilegal preguntar si `p == q` o `p != q`.

Una función puede regresar una estructura:

```
struct punto creaPunto(int x, int y) {
  struct punto t;
  t.x = x;
  t.y = y;
  return t;
}
```

Una función puede recibir parámetros que sean estructuras:

```
int igualPunto(struct punto p, struct punto q) {
  return (p.x == q.x) && (p.y == q.y);
}
```

Una función puede regresar y recibir parámetros que sean estructuras:

```
struct punto sumaPunto(struct punto p, struct punto q) {
  struct punto t;
  t.x = p.x + q.x;
  t.y = p.y + q.y;
  return t;
}
```

Definición de tipos

La palabra reservada **typedef** nos permite definir tipos nuevos.

```
typedef int entero;
```

define el tipo entero como sinónimo de **int**.

```
typedef char cadena[10];
```

define el tipo cadena como un arreglo de diez **char**.

```
typedef struct punto {
  int x;
  int y;
} punto;
```

define el tipo punto como sinónimo de **struct punto**.

Usando tipos definidos se simplifica la escritura de funciones:

```
punto sumaPunto(punto p, punto q) {
  punto t;
  t.x = p.x + q.x;
  t.y = p.y + q.y;
  return t;
}
```

Ya no es necesario anotar una y otra vez que eran **struct**.

Con frecuencia se usan parámetros por referencia para las estructuras:

```
punto sumaPunto(punto *p, punto *q) {
  punto t;
  t.x = (*p).x + (*q).x;
  t.y = (*p).y + (*q).y;
  return t;
}
```

Esto es para evitar la copia de la estructura (que pudiera ser muy lenta si es grande).

Por supuesto, también se pueden modificar los parámetros por referencia:

```
void  sumaPunto(punto *p, punto *q, punto *t) {
  // modificaremos la estructura *t
  (*t).x = (*p).x + (*q).x;
  (*t).y = (*p).y + (*q).y;

}
```

Esto requiere una estructura ya existente para colocar el resultado.

La expresión `(*p).x` es tan común, que existe otra forma de escribirla:

```
void  sumaPunto(punto *p, punto *q, punto *t) {
  // modificaremos la estructura *t
  t->x = p->x + q->x;
  t->y = p->y + q->y;

}
```

La expresión `p->x` es equivalente a `(*p).x`.

Una estructura puede tener miembros que son estructuras:

```
typedef struct {
  float area;     // area de un triangulo
  float peri;     // perimetro del mismo
  punto a, b, c; // vertices del triangulo
} triangulo;
```

Esto también se pudo hacer así:

```
typedef struct {
  float area;     // area de un triangulo
  float peri;     // perimetro del mismo
  punto a[3];     // vertices del triangulo
} triangulo;
```

1. Escribe la función `int dominaPunto(punto p, punto q)` que diga si las dos coordenadas de `p` son menores a las dos coordenadas de `q`.
2. Escribe la función `punto restaPunto(punto p, punto q)` que reste los puntos `p` y `q`.
3. Reescribe estas funciones usando referencias.
4. Escribe la función `triangulo creaTriangulo(punto a, punto b, punto c)` que llene todos los miembros de una estructura `triangulo`. ¿Cómo se calcula el área y el perímetro de un triángulo? Reescribe esta función usando referencias.
5. Escribe la función `void relojTriangulo(triangulo *t)` que asegure que los tres vértices del triángulo están en el orden de las manecillas del reloj.

Complejos

```
typedef struct {
  double re, im;
} complejo;
```

Racionales

```
typedef struct {
  long long a, b;
} racional;
```

1. Escribe funciones que sumen, resten, multipliquen y dividan complejos. ¿Qué hacer con la división por cero?
2. Escribe una función `complejo potencia(complejo z, int n)` que calcule la potencia `n` de `z` (suponga que `n >= 0`).
3. Escribe una función `void simplifica(racional *r)` que simplifique `r`, es decir, el numerador y denominador no deben tener factores comunes y el denominador debe ser positivo.
4. Escribe funciones que sumen, resten, multipliquen y dividan racionales. ¿Qué hacer con la división por cero?

Conjuntos como mapas de bits

Si queremos representar un conjunto en un programa debemos saber:

1. La cantidad de elementos (digamos `int n`).
2. El tipo de sus elementos (digamos `int` de `0` a `n-1`).
3. Dónde almacenar el conjunto (digamos `int a[n]`).
4. Cómo guardar los elementos (ceros y unos en `a[0..n-1]`).

Ejemplo

```
int n = 10; // hasta diez elementos en el conjunto
int a[10] = {0, 1, 0, 1, 1, 1, 0, 1, 0, 1};
int b[10] = {1, 0, 1, 1, 1, 0, 1, 0, 1, 0};
int c[10]; // conjunto sin inicializar
```

Problemas

1. Todos los elementos deben ser del mismo tipo.
2. Todos los elementos deben ser enteros en el rango 0 a $n - 1$.
3. Debemos saber al principio la cantidad máxima de elementos.
4. No podemos cambiar esa cantidad en tiempo de ejecución.
5. Algunas funciones como `uneMapa` y `cardMapa` son `muy` lentas.
6. Las funciones requieren pasar al menos dos parámetros por conjunto.

Soluciones

Hoy vamos a resolver dos de estos problemas (el tercero y el último).

Conjuntos como mapas de bits

Crear un conjunto vacío $A \leftarrow \emptyset$

Para construir un mapa de bits a vacío, inicializamos en 0 todas las entradas de a:

```
void  creaMapa(int n, int a[]) {
  for (int i = 0; i < n; i++)
    a[i] = 0; // i no esta
}
```

Recuerda que en este caso a ya era un arreglo que existía.

También lo podemos hacer pidiendo un arreglo lleno de ceros del tamaño solicitado:

```
int *creaMapa(int n) {
  int *a = (int *) calloc(n, sizeof(int));
  return a;
}
```

No debemos olvidar liberarlo después con `free`.

Un mapa de bits consta de dos componentes: la cantidad de elementos y el arreglo. Podemos declarar una estructura conveniente para esto.

```
typedef struct {
  int   n; // cantidad de elementos
  int *a; // apuntador al arreglo
} mapa;
```

Observa que no declaramos un arreglo dentro de la estructura; lo haremos con memoria dinámica.

Conjuntos como mapas de bits

Crear un conjunto vacío $A \leftarrow \emptyset$

```
mapa creaMapa(int n) {
  mapa s;
  s.a = (int *) calloc(n, sizeof(int));
  s.n = (s.a != NULL) ? n : 0;
  return s;
}
```

Conjuntos como mapas de bits

Agregar y eliminar elementos, pasando el conjunto por valor

Agregar un elemento

```
void agregaMapa(mapa s, int x) {
  if (0 <= x && x < s.n)
    s.a[x] = 1;
}
```

Eliminar un elemento

```
void eliminaMapa(mapa s, int x) {
  if (0 <= x && x < s.n)
    s.a[x] = 0;
}
```

Conjuntos como mapas de bits

Agregar y eliminar elementos, pasando el conjunto por referencia

Agregar un elemento

```
void agregaMapa(mapa *s, int x) {
  if (0 <= x && x < s->n)
    s->a[x] = 1;
}
```

Eliminar un elemento

```
void eliminaMapa(mapa *s, int x) {
  if (0 <= x && x < s->n)
    s->a[x] = 0;
}
```

Conjuntos como mapas de bits

Igualdad de conjuntos $S = T$, pasando conjuntos por valor

```
int igualMapa(mapa s, mapa t) {
  if (s.n != t.n)
    return 0;
  for (int i = 0; i < s.n; i++)
    if (s.a[i] != t.a[i])
      return 0;
  return 1;
}
```

Conjuntos como mapas de bits

Igualdad de conjuntos $S = T$, pasando conjuntos por referencia

```
int igualMapa(mapa *s, mapa *t) {
  if (s->n != t->n)
    return 0;
  for (int i = 0; i < s->n; i++)
    if (s->a[i] != t->a[i])
      return 0;
  return 1;
}
```

Conjuntos como mapas de bits

Ejercicios

1. Escribe `void destruyeMapa(mapa s)` que libere la memoria dinámica pedida para `s`.
2. Escribe una función `int cardMapa(mapa s)` que regrese la cardinalidad de `s`. ¿Cómo se podría modificar lo que hicimos para que esta función sólo haga un paso?
3. Escribe una función `void compMapa(mapa s)` que complemente los elementos de `s`.
4. Escribe `mapa uneMapa(mapa s, mapa t)`, `mapa intersectaMapa(mapa s, mapa t)`, `mapa diferenciaMapa(mapa s, mapa t)` y `mapa difsimMapa(mapa s, mapa t)` que calculen la unión, la intersección, la diferencia y la diferencia simétrica de dos mapas de bits. Reescribe estas funciones con un tercer parámetro por referencia.
5. Escribe funciones `int minMapa(mapa s)` e `int maxMapa(mapa t)` que regresen el menor y mayor elementos de un mapa de bits, respectivamente.
6. En un multiconjunto cada elemento puede aparecer una o más veces. ¿Cómo modificar lo que hicimos para implementar multiconjuntos?