Algoritmos y estructuras de datos

Conjuntos en arreglos ordenados

Francisco Javier Zaragoza Martínez

Universidad Autónoma Metropolitana Unidad Azcapotzalco
Departamento de Sistemas

12 de agosto de 2024

Citas

Henri Poincaré

Las generaciones futuras considerarán que la teoría de conjuntos es una enfermedad de la que nos hemos curado.

Paul Cohen

Una vez que vamos más allá de los enteros ¿se refiere la teoría de conjuntos a una realidad existente? o, como algunos formalistas lo consideran, ¿debemos considerarla como un interesante juego formal?

Conjuntos como arreglos desordenados

Si queremos representar un conjunto en un programa debemos saber:

1. El tipo de sus elementos (digamos `int`).
2. La cantidad de elementos (digamos `int n`).
3. Dónde guardar los elementos (digamos `int a[MAX]`).
4. Cómo guardar los elementos (desordenados en `a[0..n-1]`).

Ejemplo

```
int n = 3;          // tres elementos en el conjunto
int a[10] = {3, 1, 4}; // arreglo de diez entradas
```

Conjuntos como arreglos desordenados

Estructura de datos

Necesitamos almacenar tres datos: el tamaño del arreglo, el número de elementos en el arreglo y el arreglo con los elementos. Esto lo podemos hacer así:

```
typedef struct {
  int max; // maxima cantidad de elementos
  int n;   // cantidad actual de elementos
  int *a;  // apuntador a un arreglo
} arreglo;
```

Nos haremos cargo por separado de pedir y liberar la memoria del arreglo.

Conjuntos como arreglos desordenados

Crear un conjunto $S \leftarrow \emptyset$

```
arreglo creaArreglo(int max) {
  arreglo s;
  s.a = (int *) malloc(max*sizeof(int));
  s.max = (s.a != NULL) ? max : 0;
  s.n = 0;
  return s;
}
```

Destruir un conjunto

Con memoria dinámica debemos hacer limpieza:

```
void destruyeArreglo(arreglo *s) {
  free(s->a);  // libera el arreglo
  s->a = NULL; // no lo uses otra vez
  s->max = 0;  // no le caben elementos
  s->n = 0;    // no tiene elementos
}
```

La estructura del conjunto se modifica, así que se manda por referencia.

Conjuntos como arreglos desordenados

Pertenencia de un elemento $x \in S$

Esta función devuelve la posición del elemento `x` en el arreglo `a[MAX]` o `-1` si no está.

```
int enArrDes(arreglo s, int x) {
  for (int i = 0; i < s.n; i++)
    if (s.a[i] == x)
      return i;
  return -1;
}
```

Esto es un ejemplo de búsqueda lineal, que tarda hasta n pasos.

Conjuntos como arreglos desordenados

Agregar un elemento $S \leftarrow S \cup x$

Hagamos una función que agregue un elemento a un conjunto si es que no está y cabe. Debe avisar si no lo puede agregar por falta de espacio.

```
int agregaArrDes(arreglo *s, int x) {
  if (enArrDes(*s, x) >= 0)
    return 1;          // x ya estaba en el arreglo
  if (s->n == s->max) // si el arreglo esta lleno
    return 0;          // no se pudo agregar x
  s->a[s->n] = x;      // pon x al final de a
  s->n++;              // un elemento mas en s
  return 1;            // si se pudo agregar x
}
```

La estructura del conjunto se modifica, así que se manda por referencia.

Eliminar un elemento $S \leftarrow S \setminus x$

Hagamos una función que elimine un elemento de un conjunto si está. Debe avisar si no lo puede eliminar por no estar.

```
int eliminaArrDes(arreglo *s, int x) {
  int i = enArrDes(*s, x);
  if (i < 0)              // si no encuentras x
    return 0;             // no se elimina
  s->n--;                 // disminuye la cuenta
  s->a[i] = s->a[s->n]; // mueve el ultimo de a
  return 1;               // y termina
}
```

La estructura del conjunto se modifica, así que se manda por referencia.

Conjuntos como arreglos desordenados

Inclusión de conjuntos $S \subseteq T$

Cada elemento de un conjunto debe estar en el otro.

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

Esto es muy lento, pues si $n = |S|$ y $m = |T|$, puede tardar hasta nm pasos.

Igualdad de conjuntos $S = T$

Los dos conjuntos tienen la misma cantidad de elementos y uno es subconjunto del otro.

```
int igualArrDes(arreglo s, arreglo t)
{
  return (s.n == t.n && subArrDes(s, t));
}
```

Intersección de conjuntos $S \cap T$

Pide u suficiente para el menor de s y t y agrega los elementos de s que están en t.

```
arreglo intersectaArrDes(arreglo s, arreglo t) {
  arreglo u = creaArreglo((s.n < t.n) ? s.n : t.n);
  for (int i = 0; i < s.n; i++)
    if (enArrDes(t, s.a[i]) >= 0)
      u.a[u.n++] = s.a[i];
  return u;
}
```

Esto es muy lento: observa que hay dos ciclos anidados.

Conjuntos como arreglos desordenados

Ejercicios

1. Escribe una función `int recreaArreglo(arreglo *s, int max)` que toma un arreglo y le intenta cambiar su tamaño máximo a `max`. ¿Qué problemas podrían aparecer al hacer esto?
2. Reescribe la función `int enArrDes(arreglo s, int x)` usando apuntadores.
3. Escribe funciones `arreglo uneArrDes(arreglo s, arreglo t)`, `arreglo diferenciaArrDes(arreglo s, arreglo t)` y `arreglo difsimArrDes(arreglo s, arreglo t)`.
4. Escribe funciones `int minArrDes(arreglo s)` e `int maxArrDes(arreglo t)` que regresen el menor y mayor elementos de un conjunto, respectivamente.
5. En un multiconjunto cada elemento puede aparecer una o más veces. Reescribe las funciones de conjuntos para que implementen un multiconjunto de enteros almacenado en un arreglo desordenado.

Problemas

1. Todos los elementos deben ser del mismo tipo.
2. Algunas funciones como `enArrDes` y `subArrDes` son muy lentas.

Soluciones

En este curso resolveremos el segundo problema.

Conjuntos como arreglos ordenados

Si queremos representar un conjunto en un programa debemos saber:

1. El tipo de sus elementos (digamos `int`).
2. La cantidad de elementos (digamos `int n`).
3. Dónde guardar los elementos (digamos `int a[MAX]`).
4. Cómo guardar los elementos (ordenados en `a[0..n-1]`).

Ejemplo

```
int n = 3;          // tres elementos en el conjunto
int a[10] = {1, 3, 4}; // arreglo de diez entradas
```

Cambiemos la condición de paro del ciclo:

```
int enArrOrd(arreglo s, int x) {
  for (int i = 0; (i < s.n) && (s.a[i] <= x); i++)
    if (s.a[i] == x)
      return i;
  return -1;
}
```

Esto sigue siendo búsqueda lineal ($\approx n$ pasos). Sin embargo, como el arreglo a está ordenado, se podría usar un algoritmo llamado búsqueda binaria ($\approx \log_2 n$ pasos).

Conjuntos como arreglos ordenados

Agregar un elemento $S \leftarrow S \cup x$

```
int agregaArrOrd(arreglo *s, int x) {
  if (enArrOrd(*s, x) >= 0)
    return 1;              // x ya estaba en el arreglo
  if (s->n == s->max)      // si el arreglo esta lleno
    return 0;              // no se pudo agregar x
  for (int i = s->n; i > 0 && s->a[i-1] > x; i--)
    s->a[i] = s->a[i-1]; // recorre los elementos
  s->a[i] = x;             // inserta x en su lugar
  s->n++;                  // un elemento mas en s
  return 1;                // si se pudo agregar x
}
```

Conjuntos como arreglos ordenados

Eliminar un elemento $S \leftarrow S \setminus x$

```
int eliminaArrOrd(arreglo *s, int x) {
  int i = enArrOrd(*s, x);
  if (i < 0)              // si no encuentras x
    return 0;             // no se elimina
  s->n--;                 // disminuye la cuenta
  for (; i < s->n; i++)
    s->a[i] = s->a[i+1]; // recorre los elementos
  return 1;               // y termina
}
```

Esta función hace $\approx n$ pasos si los conjuntos `s` y `t` son iguales (comparado con $\approx n^2$ pasos si los vectores no estuvieran ordenados).

```
int igualArrOrd(arreglo s, arreglo t) {
  if (s.n != t.n)          // si miden distinto
    return 0;              // no son iguales
  for (int i = 0; i < s.n; i++)
    if (s.a[i] != t.a[i]) // elemento distinto
      return 0;            // no son iguales
  return 1;
}
```

Conjuntos como arreglos ordenados

Intersección de conjuntos $S \cap T$

Pide un arreglo u suficiente para el menor de s y t y avanza sobre los tres arreglos.

```
arreglo intersectaArrOrd(arreglo s, arreglo t) {
  arreglo u = creaArreglo((s.n < t.n) ? s.n : t.n);
  int i = 0, j = 0;
  while (i < s.n && j < t.n) {
    if (s.a[i] == t.a[j]) {     // elementos iguales
      u.a[u.n] = s.a[i];        // copia uno
      u.n++; i++; j++;          // avanza en los tres arreglos
    } else if (s.a[i] < t.a[j]) {
      i++;                      // s contiene al menor
    } else j++;                 // t contiene al menor
  }
  return u;
}
```

1. Reescribe las funciones `int agregaArrOrd(arreglo *s, int x)` y `int eliminaArrOrd(arreglo *s, int x)` usando apuntadores.
2. Escribe `int subArrOrd(arreglo s, arreglo t)` que tarde $\approx n + m$ pasos.
3. Escribe funciones `arreglo uneArrOrd(arreglo s, arreglo t)`, `arreglo diferenciaArrOrd(arreglo s, arreglo t)` y `arreglo difsimArrOrd(arreglo s, arreglo t)` que tarden $\approx n + m$ pasos.

Resumen de resultados de operaciones sobre un conjunto

Número de pasos en el peor de los casos, actuando sobre un conjunto A de hasta n elementos y un elemento x.

Operación	Símbolo	Mapa de bits	Desordenado	Ordenado
crear	$\emptyset$	n	1	1
destruir		1	1	1
cardinalidad	$\lvert A \rvert$	n	1	1
complemento	$\overline{A}$	n	—	—
pertenencia	$x \in A$	1	n	n
agregar	$A \cup x$	1	n	n
eliminar	$A \setminus x$	1	n	n

Número de pasos en el peor de los casos, actuando sobre dos conjuntos A de hasta n elementos y B de hasta m elementos.

Operación	Símbolo	Mapa de bits	Desordenado	Ordenado
igualdad	$A = B$	$n + m$	nm	$n + m$
inclusión	$A \subset B$	$n + m$	nm	$n + m$
unión	$A \cup B$	$n + m$	nm	$n + m$
intersección	$A \cap B$	$n + m$	nm	$n + m$
diferencia	$A \setminus B$	$n + m$	nm	$n + m$
simétrica	$A \triangle B$	$n + m$	nm	$n + m$

¿Qué se puede mejorar?

Lo que no se puede mejorar

Las operaciones que hacen un paso ya no se pueden mejorar. Tampoco se pueden mejorar aquellas que necesitan ver cada elemento al menos una vez. Por lo tanto, no se pueden mejorar las que se tardan $n + m$ pasos.

Lo que sí se puede mejorar

Eso nos deja algunas funciones candidatas:

1. Queremos crear conjuntos vacíos en un paso.
2. Queremos decidir pertenencia en menos de n pasos (con búsqueda binaria).
3. Queremos agregar y eliminar elementos en menos de n pasos.

Lo interesante es que se puede lograr todo al mismo tiempo.