Algoritmos y estructuras de datos

Conjuntos en arreglos ordenados y desordenados

Francisco Javier Zaragoza Martínez

Universidad Autónoma Metropolitana Unidad Azcapotzalco
Departamento de Sistemas

18 de mayo de 2015

Tipo de datos concreto conjunto desordenado

¿Cómo representar un conjunto?

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 de un conjunto

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

Estructura de datos

Estructura para un conjunto

Necesitamos almacenar tres datos: el tamaño máximo del arreglo, el número de elementos del arreglo y el arreglo. 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       */
} conjunto;
```

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

Crear un conjunto

Crear un conjunto vacío de tamaño dado

```
conjunto crea(int max)
{
  conjunto s;

  s.a = (int *) malloc(max*sizeof(int));
  if (s.a != NULL) /* si hubo memoria                */
    s.max = max;   /* puede haber max elementos */
  else             /* en caso contrario             */
    s.max = 0;     /* puede haber 0 elementos   */
  s.n = 0;         /* no hay elementos              */
  return s;
}
```

Destruir un conjunto

Destruir un conjunto

Con memoria dinámica debemos hacer limpieza:

```
void destruye(conjunto *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     */
}
```

Pertenencia

Pertenencia de un elemento

Será útil que esta función nos devuelva la posición del elemento `x` en el arreglo `a[MAX]` o negativo si no está.

```
int pertenece(conjunto s, int x)
{
  int i;

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

A esto también se le llama búsqueda lineal.

Agregar un elemento

Agregar un elemento

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 agrega(conjunto *s, int x)
{
  if (pertenece(*s, x) >= 0)
    return 1;              /* x ya estaba en el arreglo */
  if (s->n < s->max) {     /* si hay espacio            */
    s->a[s->n] = x;        /* pon x al final de a       */
    s->n++;                /* hay un elemento mas       */
    return 1;
  } else return 0;         /* no hubo espacio           */
}
```

El conjunto se puede modificar, así que se mandó una referencia.

Eliminar un elemento

Eliminar un elemento

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

```
int elimina(conjunto *s, int x)
{
  int i;

  i = pertenece(*s, x);
  if (i >= 0) {                /* si encuentras x          */
    s->n--;                    /* disminuye la cuenta      */
    s->a[i] = s->a[s->n];      /* mueve el ultimo de a     */
    return 1;                  /* y termina                */
  } else return 0;             /* x no estuvo en a         */
}
```

El conjunto se puede modificar, así que se mandó una referencia.

Inclusión

Subconjunto

Cada elemento de un conjunto debe estar en el otro.

```
int subconjunto(conjunto s, conjunto t)
{
  int i;

  for (i = 0; i < s.n; i++)
    if (pertenece(t, s.a[i]) < 0)
      return 0;
  return 1;
}
```

Igualdad

Igualdad

Los dos conjuntos deben tener el mismo número de elementos y uno debe ser subconjunto del otro.

```
int igual(conjunto s, conjunto t)
{
  return (s.n == t.n && subconjunto(s, t));
}
```

Unión

Unión destructiva

Esta función agrega el contenido del segundo conjunto al primero.

```
void une(conjunto *s, conjunto t)
{
  int i;

  for (i = 0; i < t.n; i++)
    agrega(s, t.a[i]);
}
```

Ejercicios

Ejercicios

1. Escribe `recrea` que toma un conjunto y le cambia su tamaño máximo. ¿Qué problemas podrían aparecer al intentar esto?
2. Reescribe `pertenece` usando apuntadores.
3. Reescribe `une` para que avise si la operación tuvo éxito, es decir, si hubo espacio suficiente para almacenar la unión.
4. Escribe `intersecta`, `diferencia` y `simetrica` destructivas.
5. Escribe `une`, `intersecta`, `diferencia` y `simetrica` no destructivas, es decir, el resultado debe quedar en un tercer conjunto.

Problemas y soluciones

Problemas

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

Soluciones

En este curso resolveremos el segundo problema.

Ejercicios

Mínimo y máximo

Escribe funciones `min` y `max` que regresen el menor y mayor elementos de un conjunto, respectivamente.

Multiconjuntos

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.

Tipo de datos concreto conjunto ordenado

¿Cómo representar un conjunto?

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 de un conjunto

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

Pertenencia en un conjunto ordenado

Pertenencia de un elemento

Cambiemos la condición de paro del ciclo:

```
int pertenece(conjunto s, int x)
{
  int i;

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

Agregar un elemento

Agregar un elemento

```
int agrega(conjunto *s, int x)
{
  int i;

  if (pertenece(*s, x) >= 0)
    return 1;              /* x ya estaba en el arreglo */
  if (s->n < s->max) { /* si hay espacio                  */
    for (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++;                /* hay un elemento mas        */
    return 1;
  } else return 0;         /* no hubo espacio            */
}
```

Eliminar un elemento

Eliminar un elemento

```
int elimina(conjunto *s, int x)
{
  int i;

  i = pertenece(*s, x);
  if (i >= 0) {                  /* si encuentras x          */
    s->n--;                      /* disminuye la cuenta      */
    for (; i < s->n; i++)
      s->a[i] = s->a[i+1];       /* recorre los elementos    */
    return 1;                    /* y termina                */
  } else return 0;               /* x no estuvo en a         */
}
```

Igualdad de conjuntos ordenados

Igualdad

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

```
int igual(conjunto s, conjunto t)
{
  int i;

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

Otras operaciones de conjuntos

Ejercicios

- Reescribe `agrega` y `elimina` usando apuntadores.
- Reescribe `subconjunto` de modo que se tarde $\approx n + m$ pasos.
- Reescribe `une`, `intersecta`, `diferencia` y `simetrica` no destructivas de modo que se tarden $\approx n + m$ pasos.

Tres representaciones de conjuntos

Resumen de resultados

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

Operación	Símbolo	Arreglo 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

Tres representaciones de conjuntos

Resumen de resultados

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	Arreglo 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.
3. Queremos agregar y eliminar elementos en menos de n pasos.

Se puede lograr todo al mismo tiempo, pero comenzaremos con decidir pertenencia rápidamente en un conjunto ordenado.

Búsqueda binaria

Idea principal

Tenemos un arreglo ordenado `a[0] <= a[1] <= ... <= a[n-1]` en el que queremos saber si está `x`. Sea `m = (n-1)/2` y compare `x` con `a[m]`.

Búsqueda binaria

Idea principal

Tenemos un arreglo ordenado `a[0] <= a[1] <= ... <= a[n-1]` en el que queremos saber si está `x`. Sea `m = (n-1)/2` y compare `x` con `a[m]`.

1. Si `x == a[m]` ya acabamos.
2. Si `x < a[m]` entonces `x` no está en `a[m]...a[n-1]`.
3. Si `x > a[m]` entonces `x` no está en `a[0]...a[m]`.

Búsqueda binaria

Ejemplo: buscando 42

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	11	14	23	26	31	39	40	42	43	51	55	67	70	72	79

Búsqueda binaria

Ejemplo: buscando 42

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	11	14	23	26	31	39	40	42	43	51	55	67	70	72	79
								42	43	51	55	67	70	72	79

Búsqueda binaria

Ejemplo: buscando 42

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	11	14	23	26	31	39	40	42	43	51	55	67	70	72	79
								42	43	51	55	67	70	72	79
								42	43	51					

Búsqueda binaria

Ejemplo: buscando 42

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	11	14	23	26	31	39	40	42	43	51	55	67	70	72	79
								42	43	51	55	67	70	72	79
								42	43	51					
								42							

Búsqueda binaria

Idea más general

Tenemos un arreglo ordenado `a[i] <= a[i+1] <= ... <= a[j]` en el que queremos saber si está `x`. Sea `m = (i+j)/2` y compare `x` con `a[m]`.

1. Si `x == a[m]` ya acabamos.
2. Si `x < a[m]` entonces `x` no está en `a[m]...a[j]`.
3. Si `x > a[m]` entonces `x` no está en `a[i]...a[m]`.

Tiempo de ejecución

Sea $T(n)$ el número de comparaciones que hacemos en el peor de los casos. Es obvio que $T(1) = 1$. Si $n \geq 2$ entonces $T(n) = T(\frac{n}{2}) + 1$. En particular, si n es una potencia de 2 entonces $T(n) = \log_2(n) + 1$.

Búsqueda binaria

Con arreglos

```
int binaria(int n, int a[], int x)
{
  int i = 0, j = n-1, m;

  while (i <= j) {
    m = (i+j)/2;
    if (x == a[m])
      return m;
    if (x < a[m])
      j = m-1;
    else
      i = m+1;
  }
  return -1;
}
```

Ejercicios

Ejercicios

- Reescribe `binaria` para que devuelva la posición en la que está el elemento buscado o la posición en la que debería estar.
- Reescribe `binaria` usando apuntadores.
- Reescribe `pertenece` usando `binaria`.
- ¿Cómo cambia el tiempo de ejecución de `agrega` y `elimina` usando búsqueda binaria, según el elemento esté o no en el conjunto?