Conjuntos Disjuntos

Prof. Héctor Navarro

Actualizado por Prof. Rhadamés Carmona,
última revisión 11/04/2019

Agenda

- Motivación
- Estructura de datos
- Operaciones
- Compresión de camino
- Complejidad
- Problemas
- Tarea

Motivación

- Hay problemas en donde las operaciones de unión, y verificar si dos elementos pertenecen al mismo conjunto son muy utilizadas.
- Para estos problemas, se debe utiliza una estructura de datos adecuada para que las operaciones de pertenencia y unión sean eficientes.
- En ocasiones un conjunto puede representar una clase de equivalencia.
- Estructura de datos de Conjuntos disjuntos: llamada Disjoin sets, union-find, merge-find set.

Estructura de datos

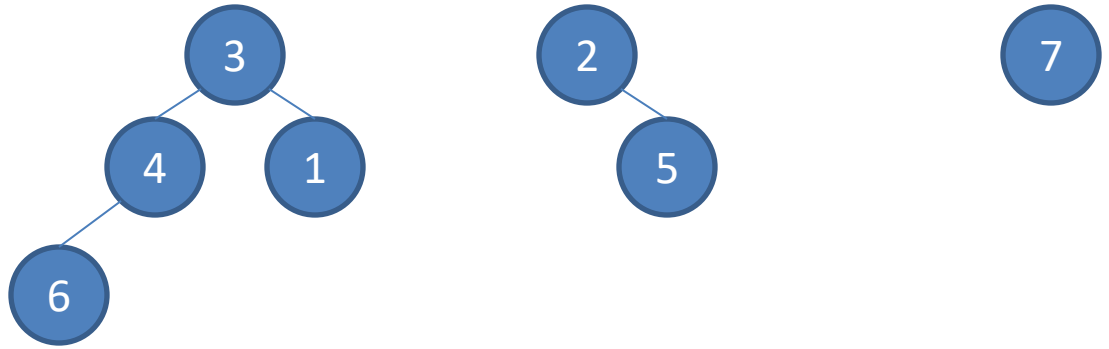
- Cada conjunto se representa como un árbol general.
- Todos los conjuntos se almacenan como un bosque.
- La unión es simplemente unir dos árboles.
- Verificar si dos elementos pertenecen al mismo conjunto (pertenencia) se reduce a verificar si están en el mimos árbol.

Estructura de datos

- Para saber si 2 elementos están en el mismo árbol, basta saber si tienen un ancestro común. Para reducir chequeos, se verifica únicamente la raíz del árbol al cuál pertenecen (debe ser la misma).
- Para esta operación, es más útil que los nodos tengan un apuntador al padre, y no apuntadores a sus hijos.

Estructura de datos

- Basta almacenar, por cada elemento del árbol, la posición de su padre. Haremos abstracción del valor del nodo, y nos centraremos únicamente en su posición en el arreglo.



padre	3	0	0	3	2	4	0
0	1	2	3	4	5	6	7

Operaciones

- Find(x): devuelve la raíz del árbol al cual pertenece x.
- Unión-pertenencia: la operación de pertenencia busca la raíz de los árboles en donde están ambos elementos. Si la raíz coincide retorna verdad. En caso contrario falso. El parámetro unión indica si los conjuntos deben unirse.

Operaciones

```
bool pertenencia(int x, int y, bool u)
{
        int i=x, j=y;
        while(padre[i]>0) i=padre[i];
        while(padre[j]>0) j=padre[j];
        if (u && (i!=j)) padre[j]=i;
        return i==j;
}
```

Operaciones

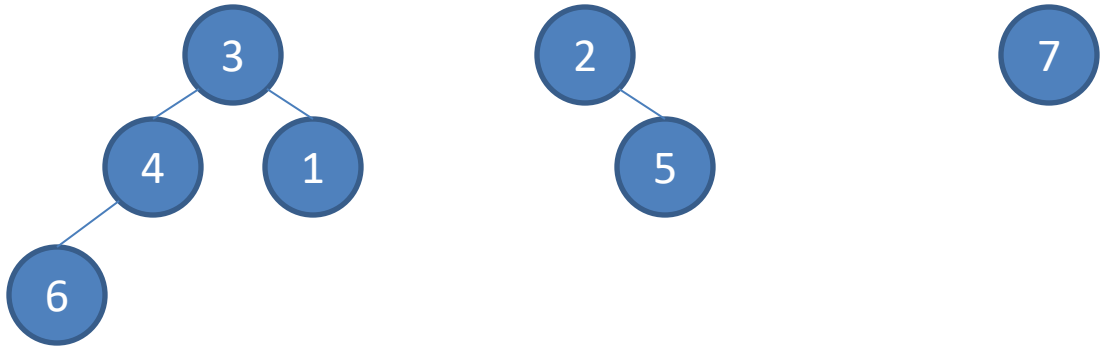
- La implementación es muy sencilla, aunque su comportamiento en el peor caso es muy malo.
- La complejidad depende de la altura del árbol. Para mantener la complejidad en lo mínimo la idea es mantener siempre el árbol con altura mínima.
- Cuando se unen dos árboles, uno queda como raíz, y el otro baja un nivel. Para minimizar la altura del árbol, parece lógico que quede como raíz el árbol con mayor cantidad de descendientes .

Operaciones

- Con este fin se debe mantener siempre la información sobre la cantidad de descendientes de cada nodo raíz (size-1).
- Para evitar el uso de otro arreglo, en lugar de usar 0 para nodos raíces, se coloca un número negativo cuyo valor absoluto es igual al número de descendientes de este nodo.

Operaciones

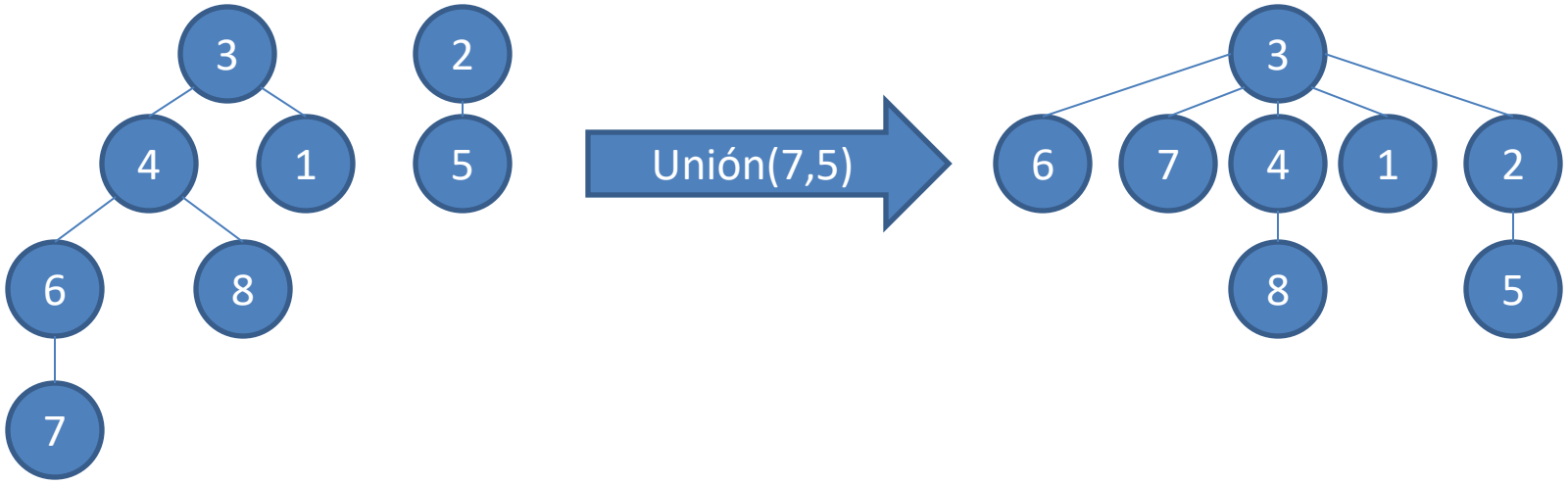
- En lugar de usar 0 para nodos raíces, se coloca un número negativo cuyo valor absoluto es igual al número de descendientes de este nodo.



padre	3	-1	-3	3	2	4	0
0	1	2	3	4	5	6	7

Operaciones

- Otra optimización: cada vez que se hace una operación de unión pertenencia, todos los nodos visitados se colocan como hijos directos de la raíz del árbol respectivo (compresión de caminos)



Operaciones

```
int pertenencia(int x, int y, bool u) {   // con comp de camino
  int i=x, j=y;
  while(padre[i]>0) i=padre[i]; // buscar raíz de x (find root)
  while(padre[j]>0) j=padre[j]; // buscar raíz de y (find root)
  while(padre[x]>0) {t=x; x=padre[x]; padre[t]=i;}  // c.c.
  while(padre[y]>0) {t=y; y=padre[y]; padre[t]=j;}  // c.c.
   if (u && (i!=j))  {
            if (padre[j] < padre[i])   // más negativo➔larger
             {    padre[j]+=padre[i]-1; padre[i]=j;    }
            else
             {    padre[i]+=padre[j]-1; padre[j]=i;    }
   }
  return i!=j;
}
```

Complejidad

- Hopcroft y Ullman demostraron que realizar una operación de unión-pertenencia se ejecuta en tiempo proporcional a log*(*n*) – logaritmo iterado de *n* (¿cuántas veces hay que aplicarle logaritmo a *n* para llegar a 1?).
- Como log*(*n*) crece tan lentamente, en la práctica se considera una operación de O(1).

Problemas

- Algoritmo de Kruskal para encontrar el mínimo spanning tree de un grafo.
- Calcular componentes conexas de un grafo.
- Conectividad en redes.
- Equivalencia de autómatas de estados finitos.
- Juego (Go, Hex).
- Mantenimiento de listas de copias duplicadas de páginas web.

Problemas

- Contar componentes conexas: recorrer todos los vértices y contar cuántos nodos tienen un índice 0 o negativo.
- Tamaño de cada componente conexa: buscar los vértices “no positivos” (raíces), obtener módulo, y sumar 1 .
- Otros problemas (UVA): 459, 599, 793 (Network Connections), 10158(war), 10178 (counting Faces), 10685 (Nature), 11503 (Virtual Friends), 11987.

Tarea

- Implementar la clase de conjuntos disjuntos. Incluir constructor, find y union-pertenece.
- Resolver algún problema de la UVA o SPOJ referente a conjuntos disjuntos, utilizando la clase. Probar en el site. Intentar pasar todos los tests (tiempo, eficacia, memoria consumida).
- Hacer un pequeño informe, que incluya el código, problema resuelto, explicación de la solución, evidencia de haber pasado todos los tests, y de que la cuenta creada es suya.