Implementación del TAD de los conjuntos. Árboles generales.

Prof. Isabel Pita

Explicación de las transparencias de clase

Código desarrollado por el profesor A. Verdejo para la asignatura de Estructuras de Datos de la Facultad de Informática de la UCM.

5 de mayo de 2020

1. Conjuntos, árboles generales.

A principio de curso vimos una implementación del TAD conjunto utilizando un array dinámico ordenado. Con dicha implementación, la operación de búsqueda tiene coste logarítmico respecto al número de elementos del conjunto, mientras que las operaciones de insertar y eliminar tienen coste lineal respecto al número de elementos del conjunto al tener que desplazar los elementos del vector para mantenerlo ordenado.

Los árboles binarios de búsqueda son un TAD que permite realizar las operaciones de buscar, insertar y eliminar en tiempo logarítmico respecto al número de elementos del árbol. Por lo tanto, la implementación con árboles binarios de búsqueda del TAD de los conjuntos (de elementos ordenables) es más eficiente que la implementación con un vector.

El TAD de los conjuntos cuenta con las siguientes operaciones:

- conjunto vacío, `set`
- insertar un elemento, `bool insert(T const& elem)`
- eliminar un elemento, `bool erase(T const& elem)`
- averiguar si un elemento pertenece al conjunto, `int count(T const& elem) const`
- averiguar si el conjunto es vacío, `bool empty() const`
- averiguar el cardinal del conjunto, `int size() const`
- iterador que permite recorrer los elementos del conjunto en orden

Los elementos del conjunto deben tener definido un orden que debe ser estricto. Haremos la implementación de la clase `set` paramétrica respecto a los elementos del conjunto y respecto al orden definido entre los elementos. Para ello implementamos la clase genérica mediante un `template` con dos parámetros, el primero indica el tipo de los elementos del conjunto y el segundo la definición del orden. Renombramos el conjunto como `set_t`.

```
template <class T, class Comparator = std::less<T>>
class set {
   using set_t = set<T, Comparator>;
....
```

Para crear un conjunto de cadenas de caracteres en el que se use como comparador el operador menor haremos:

```
set<std::string> cjto;
```

Si queremos utilizar el operador mayor como comparador usaremos

```
set<std::string,std::greater<std::string>> cjtoReves;
```

En la parte privada de la clase se definen los nodos del árbol de la misma forma en que se definían en los árboles binarios. El nodo tiene un elemento, `elem`, y un puntero a su hijo izquierdo, `iz`, y un puntero a su hijo derecho, `dr`. El árbol de búsqueda guarda un puntero a la raíz del árbol, `raiz`. También se almacenan el número de elementos del árbol, `nelems` para poder resolver la operación que calcula el cardinal del conjunto en tiempo constante y el objeto función que compara los elementos y que se da como parámetro de la clase.

```
   struct TreeNode;
   using Link = TreeNode *;
   struct TreeNode {
      T elem;
      Link iz, dr;
      TreeNode(T const& e, Link i = nullptr, Link d = nullptr) :
      elem(e), iz(i), dr(d) {}
   };

   // puntero a la raiz de la estructura jerarquica de nodos
   Link raiz;
```

```
// numero de elementos (cardinal del conjunto)
size_t nelems;

// objeto funcion que compara elementos (orden total estricto)
Comparator menor;
```

En la parte pública de la clase se definen las operaciones:

- El constructor de conjunto.

```
// constructor (conjunto vacio)
set(Comparator c = Comparator()) : raiz(nullptr), nelems(0), menor(c) {}
```

- El constructor por copia, el operador de asignación y el destructor porque en la clase se utiliza memoria dinámica.

```
// constructor por copia
set(set_t const& other) {
   copia(other);
}

// operador de asignacion
set_t & operator=(set_t const& that) {
   if (this != &that) {
      libera(raiz);
      copia(that);
   }
   return *this;
}

~set() {
   libera(raiz);
};
```

- Las operaciones de insertar, `insert`, eliminar, `erase` y buscar un elemento `count`. Las operaciones de insertar y eliminar un elemento reciben como parámetro el elemento y devuelven un booleano indicando si la operación se ha realizado. En caso de que insertemos un elemento que ya se encuentra en el conjunto o que eliminemos un elemento que no está en el conjunto el resultado de la función será `false`. La búsqueda se realiza con la función `count`, que devuelve el número de elementos que se buscan que hay en el conjunto. Como el conjunto no tiene elementos repetidos, el resultado será uno o cero dependiendo de si el elemento está en el conjunto o si no está. Se utiliza esta implementación en lugar de devolver un valor booleano por ser la implementación usada para el tipo `set` de la librería STL. Las tres funciones se implementan por medio de una función privada que vimos en la clase anterior.

```
bool insert(T const& e) {
      return inserta(e, raiz);
}

size_t count(T const& e) const  {
   return pertenece(e, raiz) ? 1 : 0;
}

bool erase(T const& e) {
   return borra(e, raiz);
}
```

- Se define un iterador constante que recorre el árbol en inorden. La implementación de este iterador coincide con la implementación realizada del iterador de los árboles binarios, por lo que no la veremos aqui.

Indicar que al estar el conjunto implementado mediante un árbol de búsqueda el recorrido en inorden del conjunto obtiene los elementos ordenados según el comparador que se proporcione al crear el conjunto.

Los conjuntos sólo tienen iterador constante, ya que no tiene sentido modificar un elemento del conjunto. Si el nuevo valor coincidiese con un elemento ya en el conjunto, el nuevo conjunto no sería correcto. Para evitar este tipo de incoherencias no se permite modificar el valor de los elementos.

Ejemplo de uso de los conjuntos. El siguiente código crea un conjunto, le añade una serie de elementos y los muestra en orden creciente.

```
set<std::string> cjto;
cjto.insert("hola");
cjto.insert("adios");
cjto.insert("caracola");
cjto.insert("pera");
cjto.insert("manzana");
cjto.insert("ciruela");
cjto.insert("zanahoria");
for (auto const& s : cjto)
std::cout << s << ' ';
std::cout << '\n';
```

Al ejecutarse el código se mostraría por pantalla

```
adios caracola ciruela hola manzana pera zanahoria
```

Para mostrar los valores en orden decreciente declaramos cómo comparador el operador mayor.

```
set<std::string,std::greater<std::string>> cjtoReves;
cjtoReves.insert("hola");
cjtoReves.insert("adios");
cjtoReves.insert("caracola");
cjtoReves.insert("pera");
cjtoReves.insert("manzana");
cjtoReves.insert("ciruela");
cjtoReves.insert("zanahoria");
for (auto const& s : cjtoReves)
std::cout << s << ' ';
std::cout << '\n';
```

Al ejecutarse el código se mostraría por pantalla

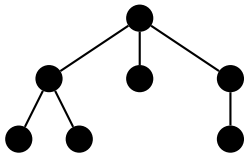
```
zanahoria pera manzana hola ciruela caracola adios
```

2. Árboles generales

Los árboles generales son árboles no vacíos y no ordenados, en el sentido de que el orden de los hijos no es importante, en los cuales el número de hijos de cada nodo es variable, pudiendo ser desde cero hasta un valor máximo llamado el grado del árbol.

En los problemas 34 *Altura de un árbol general*y 35 *Los becarios precarios* del juez, se pide resolver un problema sobre un árbol general. La descripción del árbol en la entrada se da en una línea. Primero aparece el número de hijos que tiene la raíz y a continuación aparecen las descripciones, como árboles generales, de cada uno de sus hijos, de izquierda a derecha. Las hojas son nodos con cero hijos. Observad que en la entrada de un árbol general el 0 indica una hoja, ya que los árboles generales no pueden ser vacíos.

Por ejemplo, la entrada 3 2 0 0 0 1 0 corresponde al árbol:



Implementamos una función que calcule el número de nodos de un árbol general. Resolvemos el problema sin crear el árbol en memoria. En su lugar analizamos los datos según los vamos leyendo. La función que resuelve el problema devuelve un valor entero correspondiente al número de nodos del árbol. La función comienza leyendo el número de hijos que tiene el nodo. Si el nodo no tiene hijos es el caso base y el número de nodos es uno. Si tiene hijos, hacemos un bucle en el que llamamos recursivamente a la función para que cuente el número de nodos de cada hijo. Según vamos ejecutando las llamadas recursivas vamos contando el número de nodos de cada hijo y acumulándolo en la variable `numNodos`. Para terminar devuelve la suma de los nodos de los hijos mas uno correspondiente a la raiz.

```
inline int resolver() {
    int raiz;
    std::cin >> raiz;
    if (raiz == 0) { // es un arbol vacio
        return 1;
    } else { // leer recursivamente los hijos
        int numNodos = 0;
        for (int i = 0; i < raiz; ++i)
            numNodos += resolver());
        return numNodos + 1;
    }
}
```