Implementación de un diccionario con orden entre los elementos.

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

Los diccionarios o tablas sirven para asociar valores a claves.

En ocasiones las claves tienen orden y los pares se mantienen ordenados según su clave y se pueden recorrer en orden.

- Los árboles binarios de búsqueda son una buena implementación de los diccionarios cuando las claves están ordenadas (fichero `treemap_eda.h`). En cada nodo del árbol guardamos un par ⟨clave, valor⟩, y los nodos se mantienen ordenados por claves: para todo nodo, las claves en los nodos de su hijo izquierdo son menores y las claves en los nodos de su hijo derecho son mayores.
- Las tablas dispersas (hash tables) son una buena implementación cuando no es necesario el orden entre claves (fichero `hashmap_eda.h`).

En este tema vamos a estudiar una posible implementación del TAD diccionario cuando existe un orden definido entre las claves. Para la implementación utilizaremos un árbol binario de búsqueda. Comenzamos definiendo las operaciones que vamos a implementar. Estas operaciones son un subconjunto de las operaciones definidas para este tipo en la STL y algunas de ellas se comportan de forma un poco diferente.

1. Operaciones básicas del TAD diccionario.

El TAD de los diccionarios que se proporciona en el campus virtual cuenta con las siguientes operaciones (ficheros `treemap_eda.h` y `hashmap_eda.h`):

- crear el diccionario vacío,
- `bool insert(clave_valor const& cv);` insertar un par ⟨ clave, valor ⟩. Si la clave ya existe en el diccionario no lo añade y devuelve el valor `false`. Si no existe, se añade al diccionario y devuelve el valor `true`.
- `int count(Clave const& c) const;` averiguar si una clave pertenece al diccionario. Devuelve uno si la clave pertenece al diccionarío y cero si no pertenece.
- `iterator find(Clave const& c);` averiguar si una clave pertenece al diccionario. Devuelve un iterador al elemento si este existe o un iterador al final de la tabla si el elemento no existe.
- `Valor const& at(Clave const& c) const;` consultar el valor asociado a una clave, si existe. Esta operación sólo permite consultar. Para modificar el valor asociado a la clave se debe utilizar la operación siguiente.
- `Valor & operator[](Clave const& c);` consultar y poder modificar el valor asociado a una clave, insertándola si no existe.
- `bool erase(Clave const& c);` eliminar una clave. Si la clave no existe en el diccionario devuelve `false`.
- `bool empty() const;` determinar si el diccionario está vacío.
- `int size() const;` devuelve el número de elementos en el diccionario.
- iteradores, constante y no constante, que nos permitan recorrer todos los pares ⟨ clave, valor ⟩. Si el diccionario tiene definido un orden, el iterador lo recorrerá según ese orden. Si el diccionario no tiene definido ningún orden no debe suponerse ningún tipo de orden en el recorrido.

Las principales diferencias con las operaciones definidas para los diccionarios en la librería STL (`map`, y `unordered_map`) son :

- La operación `insert` devuelve un valor booleano que indica si el elemento se ha añadido al diccionario o no. La correspondiente operación de la librería STL devuelve una pareja formada por un iterador y un valor booleano. El valor booleano es equivalente al devuelto por esta operación.

- Para comprobar si una clave pertenece al diccionario contamos con la operación `count`, equivalente a la ofrecida por la librería y con la operación `find` también equivalente a la de la librería.
- Para consultar el valor asociado a una clave, la operación `at` permite consultar el valor, pero no modificarlo posteriormente, ya que el valor se devuelve constante por referencia. La operación lanza una excepción del tipo `out_of_range` si la clave no está en el diccionario, igual que la operación de la librería.
- El operador `[]` es equivalente al ofrecido por la librería.
- La función `erase` permite eliminar una clave. Esta función devuelve `true` si el elemento está en el diccionario y se elimina, y `false` si el elemento no está en el diccionario y por lo tanto no fue posible eliminarlo. La función de la librería devuelve un valor entero que indica el número de elementos eliminados, dado que este valor se puede interpretar como un valor booleano se pueden utilizar ambas funciones de forma similar. La implementación que vamos a ver no tiene la posibilidad de eliminar elementos mediante un iterador.
- Las funciones `empty`, y `size` son equivalentes a las funciones de la librería.
- Se ofrece un iterador constante y uno no constante que permite modificar el valor de los elementos mientras se recorre el diccionario. En la implementación que vamos a ver no se ofrece la posibilidad de recorrer el diccionario en orden inverso como ocurre con el tipo `map`.

Implementación de los diccionarios con árboles de busqueda (fichero `treemap_eda.h`)

La clase tiene tres parámetros, la clave, el valor, que no tiene porque ser del mismo tipo que la clave, y el método de comparación. Es obligatorio dar los dos primeros parámetros al crear un objeto de la clase. El tercer parámetro es opcional y en caso de no darse se utiliza el operador menor.

Se define la pareja ⟨ clave, valor ⟩ mediante un struct con dos campos. La `clave` que es del tipo definido como primer parámetro de la clase `Clave` y `valor` que es del tipo definido como segundo parámetro de la clase `Valor`. En el struct se proporciona un constructor con dos parámetros. De esta forma podemos construir valores de tipo `clave_valor` utilizando el operador paréntesis.

El árbol de búsqueda que almacena los valores del diccionario se define mediante un estructura de nodos enlazados con punteros. El nodo se define mediante el struct `TreeNode` que tiene un campo del tipo `clave_valor` visto anteriormente y dos punteros al hijo izquierdo y al hijo derecho. El árbol se define mediante un `Link` a la raíz, el número de elementos del árbol y el comparador. Esta implementación es igual a la utilizada para implementar la clase `set` salvo que el nodo tiene elementos de tipo `clave_valor`.

```
template <class Clave, class Valor, class Comparador = std::less<Clave>>
class map {
public:
   // parejas <clave, valor>
   struct clave_valor {
      const Clave clave;
      Valor valor;
      clave_valor(Clave const& c, Valor const& v = Valor()) : clave(c), valor(v) {};
   };

protected:
   using map_t = map<Clave, Valor, Comparador>; // Renombramiento del mapa

   /*
    Clase nodo que almacena internamente la pareja <clave, valor>
    y punteros al hijo izquierdo y al hijo derecho.
    */
   struct TreeNode;
   using Link = TreeNode *;
   struct TreeNode {
      clave_valor cv;
      Link iz, dr;
```

```
    TreeNode(clave_valor const& e, Link i = nullptr, Link d = nullptr) : cv(e), iz(i), dr(d) {}
};

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

// numero de parejas <clave, valor>
int nelems;

// objeto funcion que compara elementos
Comparador menor;
```

La implementación de las operaciones es semejante a la realizada para los conjuntos (clase `set`) añadiendo el campo `valor`, por lo que no lo vemos aqui. El estudiante interesado puede consultarlo en el fichero con la implementación.

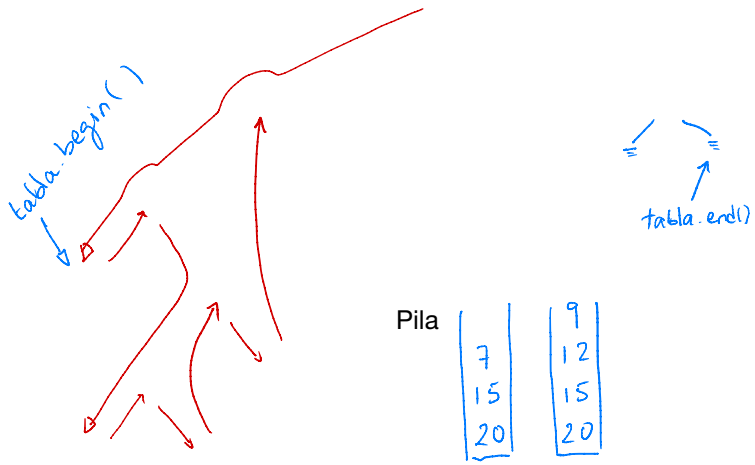
El iterador recorre el árbol de búsqueda en inorden, por lo que las claves se muestran ordenadas según el orden definido al crear el objeto. Definimos dos iteradores, uno constante que permite recorrer el árbol y uno no constante que permite modificar el valor asociado a la clave. La clase `Iterador` es una clase genérica única para los dos iteradores. La diferencia entre los dos iteradores se consigue mediante la instanciación del parámetro de la clase.

```
using const_iterator = Iterador<clave_valor const>;
using iterator = Iterador<clave_valor>;
```

El tipo `const_iterator` define el parámetro `clave_valor` cómo constante y por lo tanto no permite modificar los elementos, mientras que el tipo `iterator` define el parámetro `clave_valor` cómo no constante.

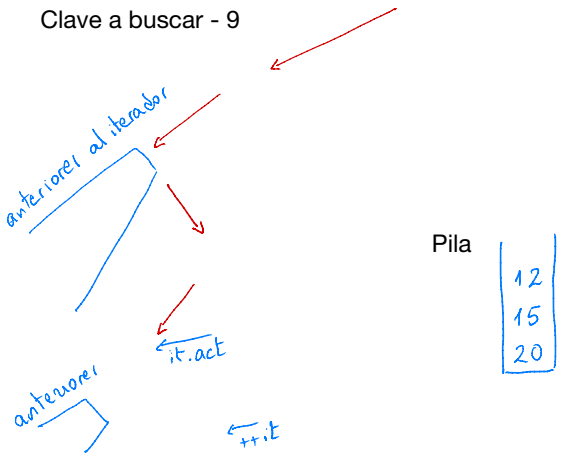
La implementación del iterador es semejante al iterador implementado en la clase `bintree` que vimos en una clase anterior. El iterador guarda en un `Link` información del nodo al cual esta *apuntando* y en una pila los `Link` a los nodos por los que se ha pasado pero que todavía no se han visitado.

Recordando cómo se realiza el recorrido, vemos que el iterador debe situarse inicialmente en el elemento más a la izquierda del árbol, esto es en el primero en el recorrido en inorden, el 5 en la figura. Para alcanzar este nodo pasa por los nodos 20, 15 y 7 sin visitarlos. Los `Link` a estos nodos se guardan en la pila del iterador para visitarlos posteriormente. Una vez visitado el nodo más a la izquierda pasamos a visitar el árbol derecho de este nodo. En el ejemplo el árbol derecho es vacío, por lo que pasamos a visitar la cima de la pila, esto es el nodo 7. A continuación recorremos el hijo derecho del 7, para ello buscamos el elemento menor, el 8. Al buscar el elemento 8 guardamos en la pila los `Link` a los nodos por los que pasamos, ya que tenemos pendiente visitarlos. Estos Links son a los nodos 12 y 9. Cómo el nodo 8 no tiene hijo derecho, visitamos la cima de la pila, el 9 y a continuación su hijo derecho en este caso el 10.



Las operaciones que va a ofrecer el iterador son:

- Operador `*`, accede al contenido de lo apuntado por el iterador. Vemos que el tipo de retorno de la función es una referencia al parámetro de la clase. Si este parámetro es constante, la referencia es constante y no se podrá modificar el contenido. Si el parámetro no es constante la referencia permitirá modificarlo. Si el `Link` que define el iterador no apunta a un elemento del árbol se lanza una excepción.
- El operador `->` permite acceder al contenido de un elemento cuando estamos utilizando punteros a structs.
- El operador `++` avanza el iterador al siguiente elemento en el orden. Para ello utiliza la función privada `next` que busca el siguiente en el hijo derecho si este existe (busca el menor del hijo derecho) y va almacenando los nodos por los que pasa en la pila. Si el hijo derecho es vacío obtiene la cima de la pila. Si la pila es vacía el elemento siguiente es el final `nullptr`.
- Operadores para comparar el valor de dos punteros `==` y `!=`. Comparan si el elemento apuntado por el iterador es el mismo en ambos iteradores. Comparan el valor del `Link`, por lo tanto la dirección de memoria en que está almacenado el elemento.
- Se implementan tres constructores.
  - El primero recibe como parámetro un `Link` y se utiliza para obtener un iterador al principio del árbol.
  - El segundo no recibe parámetros y se utiliza para obtener un iterador al final del árbol. Link `act` al valor `nullptr`.
  - El tercero recibe una tabla y una clave y se utiliza para construir un iterador a una clave concreta de la tabla. Vamos a ver su implementación. Observamos que comienza en la raíz del árbol de búsqueda que implementa la tabla, en el dibujo el nodo 20. Luego mediante un bucle busca la clave que le dan como parámetro. Si el valor buscado es menor que la clave del nodo que estoy mirando debo buscar en el hijo izquierdo. En este caso debo guardar el `Link` al nodo en el que estoy, en el dibujo el nodo 20, en la pila de nodos por los que he pasado pero todavía no he visitado. Esto es necesario para que el iterador que voy a devolver como resultado pueda continuar con el recorrido correcto del árbol desde el punto en que se encuentra. En el ejemplo, como el valor buscado, 9, es menor que el 15 se añade este a la pila y continuamos buscando en el lado izquierdo. Si el valor buscado es mayor que la clave del nodo en que me encuentro, entonces pasaré a recorrer el hijo derecho. En este caso no se guarda el nodo en la pila, ya que el nodo en que estoy ya estará recorrido por el iterador. En el ejemplo el nodo con valor 7 es un nodo ya recorrido por un iterador que se encuentre ya en el 9. Cuando coincide el valor que busco con la clave del nodo en el que estoy se da por finalizada la búsqueda. Si en la búsqueda llegamos a un hijo vacío porque no se encuentra el valor, terminamos vaciando la pila. Observad que en este caso el valor de `act` es `nullptr`.



```
protected:
   // iteradores que recorren los pares <clave, valor> de menor a mayor clave
   template <class Apuntado>
   class Iterador {
   public:
      Apuntado & operator*() const {
         if (act == nullptr)
            throw std::out_of_range("No hay elemento a consultar");
         return act->cv;
      }

      Apuntado * operator->() const {
         return &operator*();
      }

      Iterador & operator++() {  // ++ prefijo
         next();
         return *this;
      }

      bool operator==(Iterador const& that) const {
         return act == that.act;
      }
      bool operator!=(Iterador const& that) const {
         return !(this->operator==(that));
      }

   protected:
      friend class map;
      Link act;
      std::stack<Link> ancestros;  // antecesores no visitados

      // construye el iterador al primero
      Iterador(Link raiz) { act = first(raiz); }

      // construye el iterador al ultimo
      Iterador() : act(nullptr) {}

      // construye el iterador a una clave concreta (para find)
      Iterador(map_t const* m, Clave const& c) {
         act = m->raiz;
         bool encontrado = false;
         while (act != nullptr && !encontrado) {
            if (m->menor(c, act->cv.clave)) {
               ancestros.push(act);
               act = act->iz;
            } else if (m->menor(act->cv.clave, c)) {
               act = act->dr;
            } else
               encontrado = true;
         }
         if (!encontrado) { // vaciamos la pila
            ancestros = std::stack<Link>();
         }
      }

      Link first(Link ptr) {
         if (ptr == nullptr) {
            return nullptr;
         } else { // buscamos el nodo mas a la izquierda
            while (ptr->iz != nullptr) {
               ancestros.push(ptr);
```

```
            ptr = ptr->iz;
         }
         return ptr;
      }
   }

   void next() {
      if (act == nullptr) {
         throw std::out_of_range("El iterador no puede avanzar");
      } else if (act->dr != nullptr) { // primero del hijo derecho
         act = first(act->dr);
      } else if (ancestros.empty()) { // hemos llegado al final
         act = nullptr;
      } else { // podemos retroceder
         act = ancestros.top();
         ancestros.pop();
      }
   }
};
```

En la clase `map` definimos las operaciones `begin()` y `end()` para el iterador no constante y las operaciones `cbegin()`, `cend()`, `begin()` y `end()` para el iterador constante. En cada una de ellas se llama al constructor del iterador correspondiente.

Se define también la operación `find` que construye un iterador a una clave que se le pasa como parámetro, como se ha explicado anteriormente.

```
public:
   // iterador que no permite modificar el elemento apuntado
   using const_iterator = Iterador<clave_valor const>;

   const_iterator cbegin() const {
      return const_iterator(raiz);
   }
   const_iterator begin() const {
      return cbegin();
   }

   const_iterator cend() const {
      return const_iterator();
   }
   const_iterator end() const {
      return cend();
   }

   const_iterator find(Clave const& c) const {
      return const_iterator(this, c);
   }

   // iterador que si permite modificar el elemento apuntado (su valor)
   using iterator = Iterador<clave_valor>;

   iterator begin() {
      return iterator(raiz);
   }

   iterator end()  {
      return iterator();
   }

   iterator find(Clave const& c) {
      return iterator(this, c);
```

```
    }
};
```