Tablas dispersas. Implementación.

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.

29 de abril de 2020

1. Tablas dispersas.

- Las tablas dispersas abiertas son una buena implementación de los diccionarios cuando no es necesario recorrer los pares ⟨ clave, valor ⟩ en orden.
- Cada lista guarda en sus nodos pares ⟨ clave, valor ⟩ con claves sinónimas.
- Las operaciones de búsqueda, inserción o borrado primero convierten la clave en cuestión en una posición del vector (a través de la función de dispersión), y después realizan la operación sobre la lista en esa posición.
- Si el factor de carga se mantiene bajo y la función de distribución es uniforme, el coste de todas las operaciones es constante.

Implementación del diccionario no ordenado en el fichero `hashmap.h` .

La clase está parametrizada por el tipo de la clave, el tipo del valor, la función hash utilizada y el predicado que define cuando dos claves son iguales. Como con el tipo implementado en la STL, los tipos de la clave y el valor son obligatorios y la función hash y el predicado son opcionales. Para la función hash se utiliza por defecto la definida en la STL y para el predicado se utiliza por defecto la igualdad ofrecida por el tipo de la clave.

Se define una pareja `clave_valor` mediante un struct con dos campos, `clave` y `valor`. La clave se declara constante para que no se pueda modificar. Tiene un constructor en que el valor tiene por defecto el valor por defecto del tipo.

```
ttemplate <class Clave, class Valor,
          class Hash = std::hash<Clave>,
          class Pred = std::equal_to<Clave>>
class unordered_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) {};
   };
```

Para almacenar la información la clase define un vector de listas enlazadas simples. El tipo `Link` renombra un puntero a un nodo. La lista enlazada es un `Link` al primer nodo de la lista. El nodo se define en el struct `ListNode` consistente en un par de tipo `clave_valor` y un `Link` al nodo siguiente.

La clase define también el tamaño inicial del vector y el factor de carga permitido. La tabla aumentará de tamaño cuando se alcance el factor de carga. Tenemos también el número de elementos que nos servirá para implementar el método `size()` en tiempo constante, la función hash y el predicado que se están utilizando.

```
protected:
   using umap_t = unordered_map<Clave, Valor, Hash, Pred>;

   /*
    Clase nodo que almacena internamente la pareja <clave, valor>
    y un puntero al siguiente.
    */
   struct ListNode;
   using Link = ListNode *;
   struct ListNode {
      clave_valor cv;
      Link sig;
      ListNode(clave_valor const& e, Link s = nullptr) : cv(e), sig(s) {}
   };

   // vector de listas (el tamano se ajustara a la carga)
   std::vector<Link> array;

   static const int TAM_INICIAL = 17; // tamano inicial de la tabla
   static const int MAX_CARGA = 75;   // maxima ocupacion permitida 75 %

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

   // objeto funcion para hacer el hash de las claves
   Hash hash;

   // objeto funcion para comparar claves
   Pred pred;
```

Métodos ofrecidos por el diccionario .

- Tenemos un constructor que nos permite crear una tabla vacía con el tamaño por defecto, la función hash por defecto y el predicado por defecto. El constructor también permite modificar cualquiera de estos atributos.

  ```
  public:

     // constructor
     unordered_map(int n = TAM_INICIAL, Hash h = Hash(), Pred p = Pred()) :
         array(n, nullptr), nelems(0), hash(h), pred(p) {}
  ```

- Tenemos constructor por copia, operador de asignación y destructor ya que la implementación utiliza memoria dinámica.

  ```
     // constructor por copia
     unordered_map(umap_t const& other) {
        copia(other);
     }

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

```
}

~unordered_map() {
   libera();
};
```

- Para insertar un elemento:

  1. Se calcula la función hash de la clave y se traslada al intervalo $[0 \dots v.size()]$ con la función módulo (`%`).
  2. Se recorre la lista enlazada de la posición del vector dada por la función hash con el método privado `localizar` para comprobar si la clave ya existe en la tabla.
  3. Si se encuentra la clave en la lista enlazada se devuelve `false`.
  4. Si no se encuentra en esa lista es que la clave no existe en la tabla .
     a) Se comprueba si la tabla ha superado el factor de carga definido con la función privada `muy_llena()`. Si está muy llena se amplía y se calcula el nuevo hash de la clave que queremos insertar.
     b) Una vez que aseguramos que no superaremos el factor de carga de la tabla, añadimos un nuevo nodo al principio de la lista, reservando memoria con el operador `new` y conectando el nodo a la lista existente con el constructor del nodo.
     c) Para terminar se suma uno al número de elementos de la tabla.

```
bool insert(clave_valor const& cv) {
   int i = hash(cv.clave) % array.size();
   Link ant, pos = array[i];
   if (localizar(cv.clave, ant, pos)) { // la clave ya existe
      return false;
   } else {
      if (muy_llena()) {
         amplia();
         i = hash(cv.clave) % array.size();
      }
      array[i] = new ListNode(cv, array[i]);
      ++nelems;
      return true;
   }
}
```

La función `localizar` recibe la clave a buscar, un `Link ant` que es un parámetro de salida en el que la función devuelve un `Link` a la posición anterior al elemento buscado si lo encuentra (este parámetro se utiliza en la función `erase`) y un `Link pos` que es un parámetro de entrada/salida que lleva el comienzo de la lista en que se debe buscar el valor y devuelve un `Link` al elemento si lo encuentra o `nullptr` si no lo encuentra.

```
bool localizar(Clave const& c, Link & ant, Link & pos) const {
   ant = nullptr;
   while (pos != nullptr) {
      if (pred(c, pos->cv.clave))
         return true;
      else {
         ant = pos; pos = pos->sig;
      }
   }
   return false;
}
```

La función `muy_llena` comprueba que el factor de carga de la tabla no supera el máximo definido.

```
bool muy_llena() const {
   return 100.0 * nelems / array.size() > MAX_CARGA;
}
```

Por último, para ampliar el tamaño de la tabla, vemos que se busca el siguiente primo al doble del tamaño del vector. Esto es debido a que el tamaño del vector debe ser primo para que los elementos se distribuyan mejor. Cómo queremos que el tamaño del vector sea al menos el doble que el tamaño actual, buscamos el primer número primo siguiente a este valor.

Tenemos un bucle que recorre cada posición del vector. Para cada posición del vector hay un bucle interno en el que se recorre cada una de las listas moviendo los nodos a la nueva lista que les corresponda. Observad que no se copia el valor del nodo, sino que se coloca el nodo ya existente en la nueva lista por medio de los punteros. Observad también que no se pueden copiar las listas completas, porque al cambiar el tamaño del vector se modifica el valor de la función hash para cada clave, y por lo tanto cada nodo de la lista puede ir a parar a una lista diferente en la nueva tabla.

```
static int siguiente_primo(int a) {
   bool primo = false;
   while (!primo) {
      ++a;
      primo = true;
      for (int j = 2; j <= sqrt(a) && primo; ++j) {
         primo = a % j != 0;
      }
   }
   return a;
}

void amplia() {
   std::vector<Link> nuevo(siguiente_primo(array.size()*2), nullptr);
   for (int j = 0; j < array.size(); ++j) {
      Link act = array[j];
      while (act != nullptr) {
         Link a_mover = act;
         act = act->sig;
         int i = hash(a_mover->cv.clave) % nuevo.size();
         a_mover->sig = nuevo[i];
         nuevo[i] = a_mover;
      }
   }
   swap(array, nuevo);
}
```

- Para buscar una clave tenemos la función `count`. Esta función utiliza la función privada `localizar` que se explicó anteriormente.
- La función `at` permite consultar el valor asociado a una clave. En nuestra implementación esta función devuelve el valor y por lo tanto no permite modificarlo posteriormente. Utiliza la función `localizar` para encontrar la clave en la tabla y por medio del `Link pos` obtiene el valor asociado. Si el valor no existe se lanza una excepción.
- El operador `[]` nos permite modificar el valor asociado a una clave. Recordad que si la clave no existe la añade a la tabla. En la implementación utiliza la función `localizar`. Si el valor está en la tabla devuelve una referencia al valor para que se pueda modificar posteriormente. Si el valor no está lo añade a la tabla igual que se hace en la función `insertar`.
- la función `erase` elimina un par $\langle$ clave, valor $\rangle$ de la tabla. Utiliza la función `localizar` para obtener un `link` al elemento a eliminar y al elemento anterior (parámetro `ant` de la función localizar). Con estos dos Links puede borrar el nodo de la lista enlazada simple. Observad que se devuelve la memoria del nodo al sistema con el operador `delete`.

Iteradores del `unordered_map` .

Se definen dos iteradores en la clase `unordered_map`. Un iterador constante y uno no constante. Como ocurre en la implementación de la clase `map` del fichero `treemap_eda.h`, la implementación del iterador es genérica y la diferencia entre los dos iteradores se consigue mediante la instanciación del iterador.

El iterador está parametrizado por un valor booleano que indica si se trata de un iterador constante o no.

```
template <bool is_const>
class Iterador {
    ...
};
```

Se renombran algunos tipos por medio de un condicional, de forma que el tipo sea constante o una referencia dependiendo del parámetro booleano de la clase. Esto nos permite diferenciar el comportamiento del iterador constante del no constante. Por ejemplo, `reference` se refiere al tipo `clave_valor const&` si la clase esta inicializada con el valor `true`, es decir `is_cont` tiene el valor `true`.

```
using reference = typename std::conditional<is_const, clave_valor const&, clave_valor &>::t
using pointer = typename std::conditional<is_const, clave_valor const*, clave_valor *>::typ
using pointer_tabla = typename std::conditional<is_const, umap_t const*, umap_t *>::type;
```

Los operadores que ofrece el iterador son los de costumbre:

- Operador `*`, devuelve una referencia al par $\langle$ clave, valor $\rangle$ apuntado por el iterador. Dependiendo del tipo de iterador de que se trata la referencia es constante o no.
- Operador `->`, devuelve un puntero al par $\langle$ clave, valor $\rangle$ apuntado por el iterador. Se utiliza para acceder a los campos de un registro.
- Operador `++`, se utiliza para avanzar el iterador. Se implementa por medio de la función privada `next` que vemos más tarde.
- Operadores `==` y `!=`. Se utilizan para comparar iteradores

```
public:
  reference operator*() const {
     if (act == nullptr)
        throw std::out_of_range("No hay elemento a consultar");
     return act->cv;
  }

  pointer 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));
  }
```

Para recorrer la tabla necesitamos recorrer las listas de todas las posiciones del vector que forma la tabla. Se necesita información sobre la tabla que se está recorriendo, la posición del vector en que nos encontramos (el índice del vector) y dentro de esta posición, el nodo de la lista en que estamos. Los atributos que guardan esta información son:

- `tabla`, un puntero a la tabla que se está recorriendo.
- `act` un `Link` al nodo actual.
- `ind` el índice del vector que tiene la lista que estamos recorriendo.

```
protected:
    friend class unordered_map;
    pointer_tabla tabla;   // la tabla que se esta recorriendo
    Link act;         // nodo actual
    int ind;       // indice de la lista actual
```

Tenemos dos constructores:

- Un constructor que construye un iterador al primer elemento de la tabla o al último dependiendo de una variable booleana que recibe como parámetro.
  - Para construir el iterador al primer elemento debe situarse en el primer nodo de la primera lista de la tabla. Esta lista no tiene porque estar en el primer índice del vector. por ello realizamos un bucle que *salta* todas las componentes iniciales del vector cuyas listas están vacías. Por último diferencia si ha encontrado una componente con una lista no vacía y devuelve un puntero al principio de la lista, de si la tabla es vacía y todas las componentes del vector son vacías.
  - El iterador al último elemento devuelve el puntero `act` al nodo como `nullptr` y el índice `ind` del vector al tamaño de la tabla.
- Un constructor que crea un iterador a una clave concreta. Este constructor se utiliza en el método `find` de las tablas. En este constructor se calcula el índice del vector en que debe estar el valor mediante la función hash. A continuación, se utiliza la función `localizar` de las tablas para obtener un `Link` al nodo que contiene la clave. Si la función `localizar` no encuentra el elemento se devuelve un iterador al final de la tabla.

```
// iterador al primer elemento o al ultimo
Iterador(pointer_tabla t, bool first = true) : tabla(t) {
   if (first) {
      ind = 0;
      while (ind < tabla->array.size() && tabla->array[ind] == nullptr) {
         ++ind;
      }
      act = (ind < tabla->array.size() ? tabla->array[ind] : nullptr);
   } else {
      act = nullptr;
      ind = tabla->array.size();
   }
}

// iterador a una clave
Iterador(pointer_tabla t, Clave const& c) : tabla(t) {
   ind = tabla->hash(c) % tabla->array.size();
   Link ant;
   act = tabla->array[ind];
   if (!tabla->localizar(c, ant, act)) { // se devuelve iterador al final
      act = nullptr; ind = tabla->array.size();
   }
}
```

Por último la función privada `next` permite acceder al siguiente elemento. En la implementación, si el `Link act` tiene el valor `nullptr` el iterador está al final de la tabla y por lo tanto no tiene siguiente. Se lanza una excepción. El valor siguiente será el siguiente en la lista enlazada. Si nos encontrábamos al final de una lista enlazada debemos buscar la siguiente lista. Esto se hace con un bucle que recorre los índices del vector hasta encontrar el final de la tabla o un índice cuya lista no sea vacía.

```
    void next() {
      if (act == nullptr)
         throw std::out_of_range("El iterador no puede avanzar");
      act = act->sig;
      while (act == nullptr && ++ind < tabla->array.size()) {
         act = tabla->array[ind];
      }
   }
```

Métodos `begin()` y `end()` para construir iteradores .

La tabla ofrece los métodos `cbegin()` y `cend()`, `begin()` y `end()` para crear iteradores constantes y no constantes al comienzo y final de la tabla. La operación `find` del iterador no constante nos permite crear un iterador a una cierta clave y luego usar este iterador para modificar el valor asociado.

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

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

   const_iterator cend() const {
      return const_iterator(this, false);
   }
   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<false>;

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

   iterator end() {
      return iterator(this, false);
   }

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