Implementación de las Clases Pila, Cola, Lista y Arbol

Josefina Sierra Santibáñez

13 de mayo de 2018

Implementación de Estructuras de Datos

Implementaremos cada estructura de datos en dos niveles.

- El nivel superior será **una clase** con toda la **información global de la estructura** y **punteros a elementos distinguidos** (el primer nodo, el último, u otros nodos si se necesita).
- El nivel inferior será una **tupla privada (`struct`)** de la clase donde se definirá el tipo de componentes de la estructura, que denominamos **nodos**. Cada nodo constará de
  - la **información** correspondiente al elemento de la estructura que almacena, y **un puntero al siguiente** elemento de la estructura.
  - Si la estructura es arborescente, cada nodo tendrá tantos punteros como elementos siguientes (i.e. descendientes).
  - Si hemos de recorrer la estructura en sentido contrario, cada nodo también tendrá un puntero al nodo anterior en la estructura.

Implementación de Estructuras de Datos

```
class estructura_datos {

  private:

    struct Node {
      tipus_info info;
      Node* seg;
    };

    tipus_1 info_general;
    Node* element_distingit_1;
    ...

  public:
    ...
};
```

Implementación de Estructuras de Datos

Invariante de la representación

- Cada elemento de la estructura de datos tiene asociado su propio nodo, i.e. un mismo nodo no puede representar más de un elemento de un mismo objeto.
- Dos objetos diferentes (i.e. que no tengan la misma dirección de memoria) no pueden compartir nodos. Es decir, un nodo puede aparecer en la estructura de nodos enlazados de un solo objeto.

Estas restricciones evitan que la modificación de una parte de un objeto afecte a otros objetos u otras partes del mismo objeto.

Implementación de la clase Pila

```
template <class T> class Pila {

private:
   struct Node {
      T info;
      Node* seg;
   };

   int altura;
   Node* primer;
   ... // especificacion e implementacion
       // de operaciones privadas

public:
   ... // especificacion e implementacion
       // de operaciones publicas
};
```

Métodos Públicos de la clase Pila

```
Pila() {
/* Pre: cert */
/* Post: El resultat es una pila buida */
    altura = 0;
    primer = NULL;
}

bool es_buida() const {
/* Pre: cert */
/* Post: indica si el p.i. es una pila buida */
    return primer == NULL;
}

int mida() const {
/* Pre: cert */
/* Post: Retorna el nombre d'elements del p.i. */
    return altura;
}
```

Métodos Públicos de la clase Pila

```
T cim() const {
/* Pre: el p.i. es una pila no buida */
/* Post: retorna el darrer element afegit al p.i.
     return primer->info;
}

void empilar(const T& x) {
/* Pre: cert */
/* Post: El p.i. es com el p.i. original amb x
   afegit com a darrer element */
     Node* aux = new Node;
     //reserva espai pel nou element
     aux->info = x;
     aux->seg = primer;
     primer = aux;
     ++altura;
 }
```

Métodos Públicos de la clase Pila

```
void desempilar() {
/* Pre: el p.i. es una pila no buida */
/* Post: El p.i. es com el p.i. original pero
sense el darrer element afegit al p.i. original *
  Node* aux = primer;
//conserva l'acces al primer node abans d'avancar
  primer = primer->seg;
// avanca
  delete aux;
// allibera l'espai de l'antic cim
  --altura;
}
```

Métodos Públicos de la clase Pila

```
// INCORRECTO
Pila(const Pila& original) {
/* Pre: cert */
/* Post: El resultat es una copia d'original */
  altura = original.altura;
  // NO COPIA LA CADENA DE NODOS ENLAZADOS,
  // SOLO ASIGNA LOS PUNTEROS
  primer = original.primer;
}

// CORRECTO
Pila(const Pila& original) {
/* Pre: cert */
/* Post: El resultat es una copia d'original */
  altura = original.altura;
  primer = copia_cadena(original.primer);
}
```

Copia de Estructuras de Nodos Enlazados

Las estructuras de datos construidas a partir de estructuras de nodos enlazados mediante punteros requieren **implementaciones especiales** para el método **constructor de copia**, el **operador de asignación**, y el método **destructor**.

- Si se quiere crear una estructura de nodos enlazados nueva a partir de otra, se ha de definir una **operación de copia de estructura de nodos enlazados** para cada tipo de nodo (e.g. `copia_cadena`).
- Del mismo modo, es preciso definir un método **constructor de copia** para cualquier estructura de datos construida a partir de estructuras de nodos enlazados.

Métodos Privados de la clase Pila

Crea una copia de la cadena de nodos enlazados a la que apunta `m`, y devuelve un puntero que apunta a la cabeza de dicha copia.

```
static node* copia_cadena(node* m) {
/* Pre: cert */
/* Post: si m es NULL, el resultat es NULL; en
cas contrari, el resultat apunta al primer node
d'una cadena de nodes que son copia de la cadena
que te el node apuntat per m com a primer */
  Node* n;
  if (m==NULL) n=NULL;
  else {
    n = new Node;
    n->info = m->info;
    n->seg = copia_cadena(m->seg);
  }
  return n;
}
```

Métodos Públicos de la clase Pila

```
// INCORRECTO
~Pila() {
// Destructora: Esborra automaticament els
// objectes locals en sortir d'un ambit de
// visibilitat
// NO BORRA LA CADENA DE NODOS ENLAZADOS,
// SOLO BORRA EL PRIMER NODO DE LA CADENA
    delete primer;
}

// CORRECTO
~Pila() {
// Destructora: Esborra automaticament els
// objectes locals en sortir d'un ambit de
// visibilitat
    esborra_cadena(primer);
}
```

Borrado de Estructuras de Nodos Enlazados

- Cuando aplicamos la operación `delete n` a un puntero `n` que apunta a una estructura de nodos enlazados **sólo liberamos la memoria del nodo al que apunta** `n`, pero no liberamos la memoria de todos los nodos de la estructura de nodos enlazados apuntada por `n`.
- Para liberar la memoria de todos los nodos de la estructura de nodos enlazados apuntada por `n` **debemos definir una operación de borrado de estructuras de nodos enlazados** para el tipo de nodos utilizado (e.g. `esborra_cadena(primer)`).
- También es preciso **definir un método destructor** para cualquier **estructura de datos construida a partir de estructuras de nodos enlazados** (e.g. `~Pila()`).

Métodos Privados de la clase Pila

Borra la cadena de nodos enlazados a la que apunta el puntero a nodo `m`.

```
static void esborra_cadena(node* m) {
/* Pre: cert */
/* Post: no fa res si m es NULL; en cas
contrari, allibera espai dels nodes de la
cadena que te el node apuntat per m com a
primer */
  if (m != NULL) {
    esborra_cadena(m->seg);
    delete m;
  }
}
```

Métodos Públicos de la clase Pila

```
Pila& operator=(const Pila& original) {
/* Pre: cert */
/* Post: El p.i. passa a ser una copia d'original
i qualsevol contingut anterior del p.i. ha estat
esborrat (si p.i. i original no son el mateix) */
  if (this != &original) {//p.i. no es un alias
                          //d'original
    esborra_cadena(primer);
    altura = original.altura;
    primer = copia_cadena(original.primer);
  }
  return *this;
}
```

El operador & devuelve la dirección de memoria de un objeto.
El operador de asignación = devuelve una referencia al parámetro implícito (i.e. un dato de tipo Pila&).

Operador de Asignación

El operador de asignación =

1. parte de un parámetro implícito existente,
2. comprueba que el parámetro implícito y original no sean el mismo objeto (i.e. *alias*),
3. libera el espacio de memoria asociado al parámetro implícito antes de hacer la copia,
4. copia la pila original al parámetro implícito, y
5. devuelve una referencia (`Pila &`) al parámetro implícito.

El uso del mecanismo de *retorno por referencia* (i.e. el tipo `Pila &` del valor de retorno) en la definición de `operator=`, en lugar de *retorno por valor* (i.e. el tipo `Pila` del valor de retorno) permite que se puedan hacer asignaciones encadenadas.

Referencias

Si `T` es un tipo determinado, `T&` es el tipo **referencia a T**.

- Una referencia es un nombre alternativo para un objeto.
- Contiene, como los punteros, la dirección de una variable.
- No es preciso usar `*` para acceder al valor de una referencia.

```
int n=3;
int& ref=n; // ref es una referencia a int
ref=4;//variables n y ref son intercambiables
cout << n << endl; // escribe 4
```

- Las referencias deben inicializarse a la vez que se declaran. Esto se debe a que las referencias son punteros constantes, siempre deben apuntar a la misma dirección.
- Se añade `&` al tipo que devuelve una función en su declaración para indicar que **la función devuelve una referencia**. Esto significa que la función devuelve el propio objeto, en lugar del valor del objeto. Lo cual permite evitar que se haga una copia del objeto. Una función también puede devolver una referencia constante: `const Tipo& nombre_funcion(parametros)`.

Métodos Públicos de la clase Pila

```
void p_buida() {
/* Pre: cert */
/* Post: El p.i. passa a ser una pila sense
elements i qualsevol contingut anterior del
p.i. ha estat esborrat */
   esborra_cadena(primer);
   altura = 0;
   primer = NULL;
}
```