TD3: Algoritmos y Estructuras de Datos

Prof. Agustín Garassino, Gervasio Pérez, Juan Edi

Primer Semestre de 2024

Clase Teórica 10
Iteradores
Pilas y Colas

Resumen

En la clase de hoy veremos

- Concepto de iterador
- Iteradores en C++ `std`
- Estructuras básicas: Pila y Cola

C++: Recordemos Cadena

```
1     class Cadena {
2       public:
3       Cadena(); // Post: *this.longitud() = 0
4
5       int longitud() const;
6       int iesimo(int pos) const;
7       // Pre: 0 <= pos < *this.longitud()
8
9       void agregarAtras(int e);
10      // Post: this.longitud() = *this_0.longitud()+1 and
11      //       *this.iesimo(*this.longitud()-1) = e and
12      //       (ParaTodo i:int) 0 <= i < *this_0.longitud()
13      //              => *this.iesimo(i) = *this_0.iesimo(i)
14      //...
```

C++: recorrer elementos

Yo usuario de Cadena, ¿cómo la recorro de principio a fin?

```
1     int contar_pares(const Cadena& c) {
2         int res = 0;
3         for (int i = 0; i < c.longitud(); i++){
4             if (c.iesimo(i) % 2 == 0);
5               res++;
6         }
7         return res;
8     }
```

¿Qué complejidad tiene esto?

C++: Eliminar elemento

```
1     class Cadena {
2       public:
3       // ...
4
5       void eliminar_posicion(int pos);
6       // Pre: 0 <= pos < *this.longitud()
7       // Post: this.longitud() = *this_0.longitud()-1 and
8       //       (ParaTodo i:int) (0 <= i < pos
9       //              => *this.iesimo(i) = *this_0.iesimo(i))
10      //        and (pos <= i < *this.longitud())
11      //              => *this.iesimo(i) = *this_0.iesimo(i+1)
```

¿Qué complejidad tiene esta operación?

Cadena: eliminar elementos

Yo usuario de Cadena, ¿cómo elimino ciertos elementos?

```
1     void solo_pares(Cadena& c) {
2         int i = 0;
3         while (i < c.longitud()){
4             if (c.iesimo(i) % 2 != 0);
5               c.eliminar_posicion(i);
6             else
7               i++;
8         }
9     }
```

Nuevamente, ¿qué complejidad tiene esto?

Cadena: eliminar elementos

La estructura de representación de Cadena admite

- recorrer n elementos en $O(n)$
- recorrer n elementos y eliminar k de éstos en $O(n)$
- acceder/eliminar un elemento específico en $O(1)$ **si pudiera "recordar" su posición**

A la interfaz de Cadena le falta **algo**.

Cadena es un **contenedor** de elementos, como `vector`, `set` y `map`.
Todos los contenedores de elementos tienen necesidades parecidas:

- Recorrer eficientemente
- Eliminar eficientemente
- Guardarse un "puntero" a un elemento

Contenedores: el concepto de Iterador

Un iterador es un tipo asociado a un contenedor que condensa las funcionalidades de recorrer, eliminar y "apuntar".

El contenedor debe proveer en su interfaz pública operaciones que devuelvan y operen con iteradores.

Contenedores: el concepto de Iterador

Operaciones del contenedor

- `begin()`: devuelve un iterador al primer elemento
- `end()`: devuelve un iterador al **fin** (que no es un elemento)
- `erase(iterator it)`: elimina el elemento referenciado por el iterador dado

Contenedores: el concepto de Iterador

El tipo iterador debe proveer operaciones para avanzar, retroceder, acceder al elemento, y comparar iteradores.

Operaciones del iterador

- **operator**==: compara si dos iteradores apuntan al mismo elemento
- **operator***: accede al elemento "apuntado" (it. de entrada/salida)
- **operator**++: avanza al siguiente (it. unidireccional)
- **operator**--: retrocede al anterior (it. bidireccional)

En general, todas esas operaciones son $O(1)$ o $O(1)$ amortizado.
Algunos contenedores además proveen la siguiente operación:

- **operator**+, **operator**-: avanza/retrocede N lugares en O(1) (it. de acceso aleatorio)

Iteradores **const** y no **const**

Aunque mi iterador lo permita, si mi contenedor es **const**, no tengo disponibles las operaciones que modifiquen el contenedor.

Esto se maneja en C++ con dos tipos distintos: `iterator` y `const_iterator`

Iterador en Cadena

Así cambiaría la interfaz de Cadena si quisieramos proveer iteradores al usuario de la clase.

```
1  class Cadena {
2    public:
3    class const_iterator {
4        public:
5        const int& operator*() const;
6        const_iterator operator++();
7        // ...
8    }
9
10   const_iterator begin() const;
11   const_iterator end() const;
12   const_iterator erase(const_iterator pos);
13   // Post: res es un iterador al elemento posterior al borrado
14 }
```

Cadena: uso de iteradores

Así podría aprovechar un usuario de Cadena el iterador provisto.

```
1  void solo_pares(Cadena & c) {
2      // it es de tipo Cadena::const_iterator
3      auto it = c.begin();
4      while (it != c.end()) {
5          if ((*it) % 2 != 0)
6              it = c.erase(it);
7          else
8              ++it;
9      }
10 }
```

Ejemplo con std::list<T>

std::list<T> tiene una interfaz y estructura análoga a Cadena.

```
1  void solo_pares(std::list<int> & l) {
2      // it es de tipo std::list<int>::iterator
3      auto it = l.begin();
4      while (it != l.end()) {
5          if ((*it) % 2 != 0)
6              it = l.erase(it);
7          else
8              ++it;
9      }
10 }
```

Iteradores como valores

- Un iterador es un valor como cualquier otro: se puede almacenar para usar más adelante.
- Algunos contenedores devuelven un iterador cuando se agrega un elemento.

```
1    list<int> elems;
2    vector<list<int>::const_iterator> its_orden_llegada;
3    int n;
4    while(cin >> n) {
5        auto it = elems.insert(elems.end(), n);
6        its_orden_llegada.push_back(it);
7    }
8    elems.sort();
9    elems.erase(its_orden_llegada[its_orden_llegada.size()/2]);
```

Si el usuario ingresa {33, 1, 22, 3000, 0, 99, 5}, ¿cuál es el estado final de elems?

Invalidando Iteradores

- Un iterador se asemaja a un puntero: puede invalidarse. Esto depende de qué operaciones se realicen sobre el contenedor asociado y cómo está representado el iterador.
- El resultado de acceder a o avanzar un iterador inválido no está definido. Esto es relevante cuando almacenamos iteradores para su uso posterior.

Ejemplos de invalidación de iteradores:

- En general: cuando se elimina el elemento apuntado por el iterador.
- `vector`: si cambia la capacidad del vector se invalidan todos los iteradores. El método `erase` invalida iteradores al elemento eliminado y a los elementos posteriores.
- Ieer referencia C++ para conocer los casos de cada contenedor.

Problema: expresión balanceada

Dado un `string` que contiene corchetes `"[]"` y paréntesis `"()"` queremos determinar si el string está bien balanceado o no.

- `"(as[b]x)"` está bien balanceado,
- `"asbx"` está bien balanceado,
- `"(as[bx)z]"` no está bien balanceado.
- `"(as[bxz"` no está bien balanceado.

Problema: expresión balanceada

bool *esta_balanceado*(`string` *s*)

- Inicializo un contenedor `cont` vacío de tipo `T`.
- Por cada **char** ch en s:
  Si ch es "(" o "[", agregar ch al final de cont.
  Si ch es "]":
    Si el último elemento de cont es "[", sacarlo de cont.
    Si no, devolver `false`.
  Si ch es ")":
    Si el último elemento de cont es "(", sacarlo de cont.
    Si no, devolver `false`.
- Si cont es vacío devolver `true`.
  Si no devolver `false`

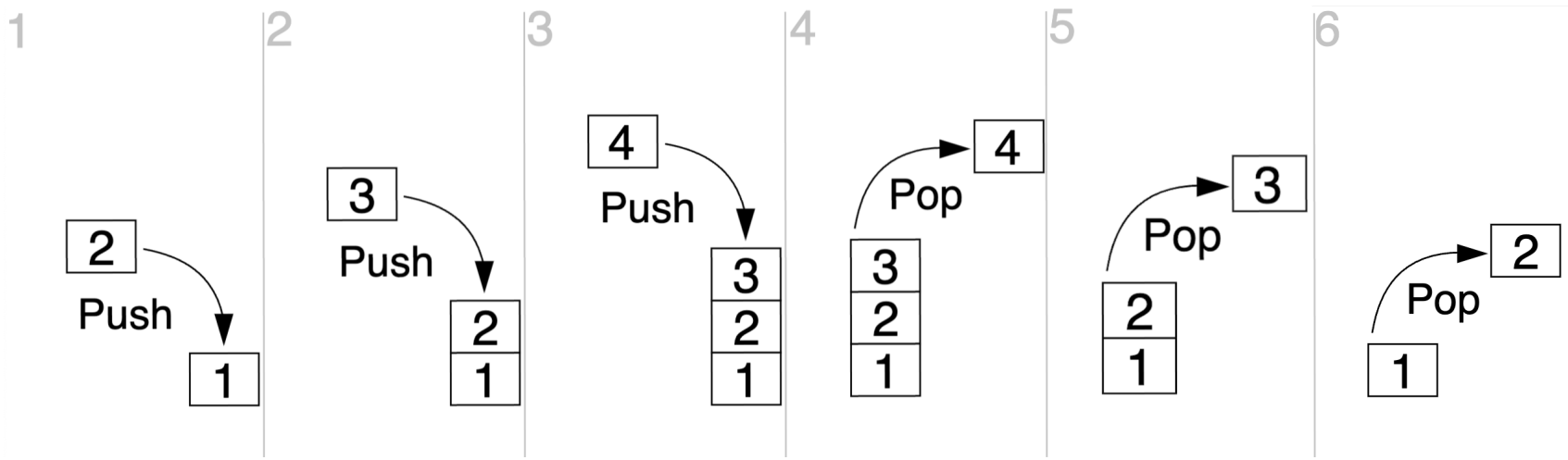
El tipo `T` debe proveer operaciones para: ver si es vacío, agregar un ítem nuevo al final, ver el último item que se agregó y eliminarlo.

Tipo de datos: Pila

Una Pila representa a un conjunto de pendientes a procesar en orden inverso al de llegada con las siguientes dos operaciones elementales:

- apilar: agrega un elemento al tope de la pila en $O(1)$
- desapilar: quita y devuelve el tope de la pila en $O(1)$

En C++: `std::stack` implementa pila usando `std::deque` como estructura de representación.



Tipo de datos: Pila

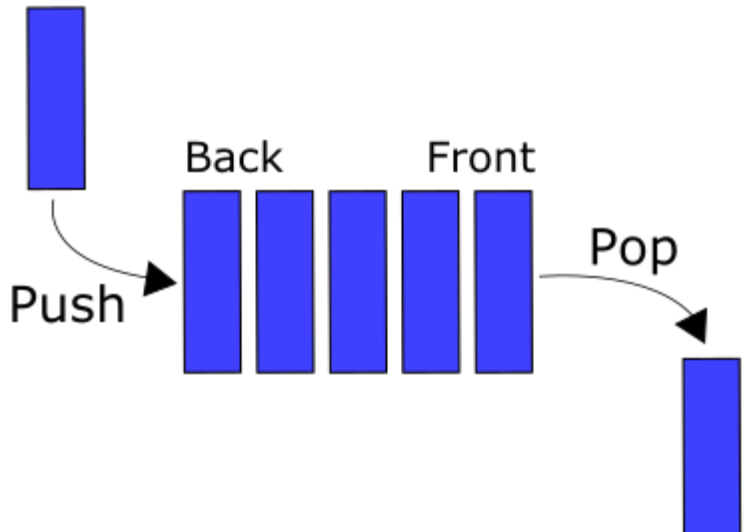
```
1       stack<int> elems;
2       while(cin >> n) {
3           elems.push(n);
4       } // ingresa 1 2 3 4 5 6
5       while(elems.size() > 0) {
6           cout << elems.top() << " ";
7           elems.pop();
8    }
9       // imprime 6 5 4 3 2 1
```

Tipo de datos: Cola

Una Cola es similar a una Pila pero representa a una lista de espera a procesar en el mismo orden de llegada. Tiene las siguientes dos operaciones elementales:

- encolar: agrega un elemento al final de la cola en O(1)
- desencolar: quita de la cola y devuelve el próximo elemento a ser procesado en O(1)

En C++: `std::queue` implementa cola usando `std::deque` como estructura de representación.



Tipo de datos: Cola

```
1        queue<int> elems;
2        while(cin >> n) {
3            elems.push(n);
4        } // ingresa "1 2 3 4 5 6"
5        while(elems.size() > 0) {
6            cout << elems.front() << " ";
7          elems.pop_front();
8          // escribe "1 2 3 4 5 6"
9    }
```

Escenarios

¿Qué estructura de datos utilizarían para...?

1. invertir un string
2. implementar un mecanismo de para deshacer el último cambio en un editor de texto
3. testear si un string es un palíndromo
4. coordinar los trabajos en una impresora compartida de red
5. implementar llamados a funciones en una computadora
6. listar los nodos de un árbol, ordenados por nivel
7. implementar el historial de navegación de un browser, permitiendo ir hacia atrás y hacia adelante
8. implementar una calculadora con notación polaca inversa

Ejercicios

1. Implementar el método `bool es_palindromo(string s)` utilizando una `std::stack`.
2. Implementar el método `bool es_balanceado(string s)` que chequea si los paréntesis s están bien balanceados. Utilizar una `std::stack`.

T11: Resumen de hoy

Iteradores:

- Motivación y Concepto
- Implementacion en C++
- Ejemplo: `Cadena` y `std::list`

Tipos de datos:

- Pila: `https://en.cppreference.com/w/cpp/container/stack`
- Cola: `https://en.cppreference.com/w/cpp/container/queue`

Guías de ejercicios

- Esta semana pueden comenzar la Guía 7 (¡la última!).