Tipos de datos lineales

Alberto Verdejo

Enero 2018

Dpto. de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Bibliografía

- R. Peña. *Diseño de Programas: Formalismo y Abstracción*. Tercera edición. Pearson Prentice-Hall, 2005.
  Capítulo 6
- N. Martí Oliet, Y. Ortega Mallén y A. Verdejo. *Estructuras de datos y métodos algorítmicos: 213 ejercicios resueltos*. Segunda edición, Garceta, 2013.
  Capítulos 3, 4 y 5
- M. A. Weiss. *Data Structures and Algorithm Analysis in C++*. Fourth edition. Pearson, 2014.
  Capítulo 3

Pilas

- Estructura de datos lineal cuya característica principal es que el acceso a los elementos se realiza en orden inverso al de su almacenamiento, siguiendo el criterio de el último en entrar es el primero en salir (LIFO).
- La ventaja de las pilas es que el acceso a la estructura, tanto para su modificación (inserción y borrado) como para la consulta de los datos almacenados, se realiza en un único punto, la cima de la pila, lo que facilita implementaciones sencillas y eficientes.
- A pesar de su sencillez, se trata de una estructura con múltiples aplicaciones en el diseño de algoritmos, como la evaluación de expresiones o la implementación de la recursión.

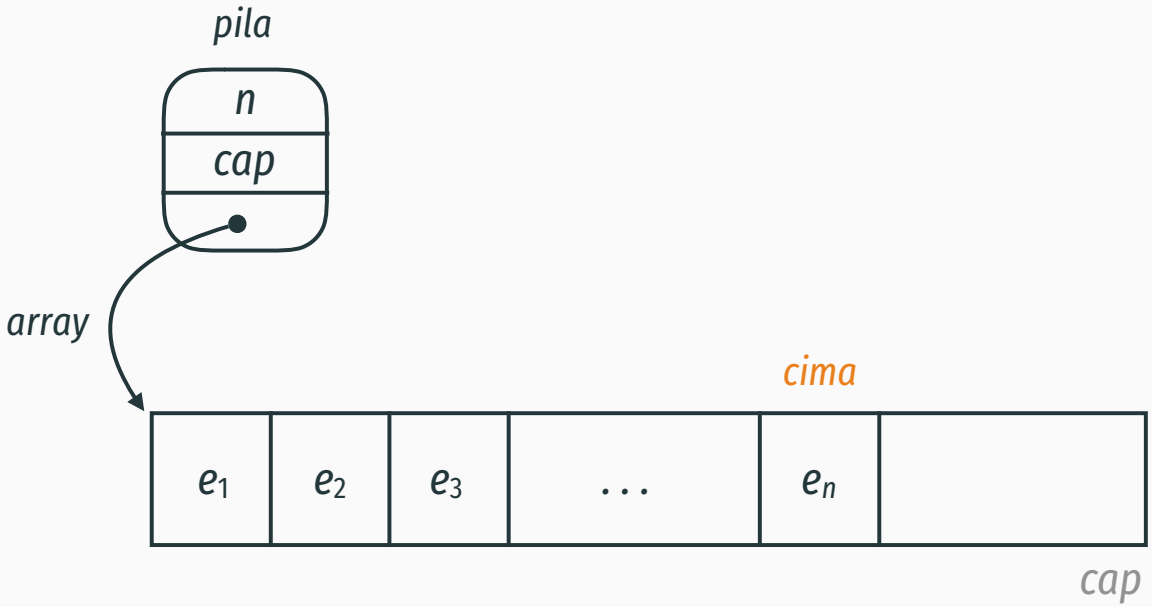
TAD de las pilas

El TAD de las pilas, `stack<T>`, cuenta con las siguientes operaciones:

- crear la pila vacía, `stack`
- apilar un elemento, `void push(T const& elem)`
- consultar el elemento en la cima, si existe, `T const& top() const`
- desapilar el elemento en la cima, si existe, `void pop()`
- determinar si la pila es vacía, `bool empty() const`

Implementación

Mediante un array dinámico



`stack_eda.h`

Evaluación de expresiones en notación postfija

Notación infija: $(8/(5-3)) * (7+2)$

Notación postfija: $8\,5\,3\,-\,/\,7\,2\,+\,*$

Evaluación de expresiones en notación postfija

Notación infija: $(8/(5-3)) * (7+2)$

Notación postfija: $8\,5\,3\,-\,/\,7\,2\,+\,*$

```
int evaluar(string const& expresion) {  // "853-/72+*"
   stack<int> pila;
   for (char c : expresion) {
      if (isdigit(c))
         pila.push(c - '0');
      else {
         int op2 = pila.top(); pila.pop();
         int op1 = pila.top(); pila.pop();
         pila.push(aplicar(c, op1, op2));
      }
   }
   return pila.top();
}
```

Transformación a notación postfija

```
string pasar_a_postfija(string const& expresion) {  // "(8/(5-3))*(7+2)"
  string postfija;  stack<char> operadores;
  for (char c : expresion) {
    if (isdigit(c)) postfija.push_back(c);
    else if (c == '(') operadores.push(c);
    else if (c == ')') {
      while (operadores.top() != '(') {
        postfija.push_back(operadores.top());  operadores.pop();
      }
      operadores.pop(); // el (
    } else {  // c es operador
        while (!operadores.empty() && es_menor_ig(c, operadores.top())) {
          postfija.push_back(operadores.top()); operadores.pop();
        }
        operadores.push(c);
    }
  }
  while (!operadores.empty()) {
    postfija.push_back(operadores.top()); operadores.pop();
  }
  return postfija;
}
```

Problemas

- ACR 141 - Paréntesis balanceados
- ACR 187 - Solitario
- ACR 384 - El juego de la linterna

Colas

- Estructura de datos lineal cuya característica principal es que el acceso a los elementos se realiza en el mismo orden en que fueron almacenados, siguiendo el criterio de el primero en entrar es el primero en salir (FIFO).
- El comportamiento de las colas es totalmente independiente del tipo de los datos almacenados en ellas, por lo que se trata de un tipo de datos genérico.
- Las colas presentan dos zonas de interés: el extremo final, por donde se incorporan los elementos, y la cabecera, por donde se consultan y se eliminan los elementos.
- El comportamiento FIFO es muy utilizado en el diseño de algoritmos para diversas aplicaciones, sobre todo en simulación, debido a la ubicuidad de las colas en toda clase de sistemas.

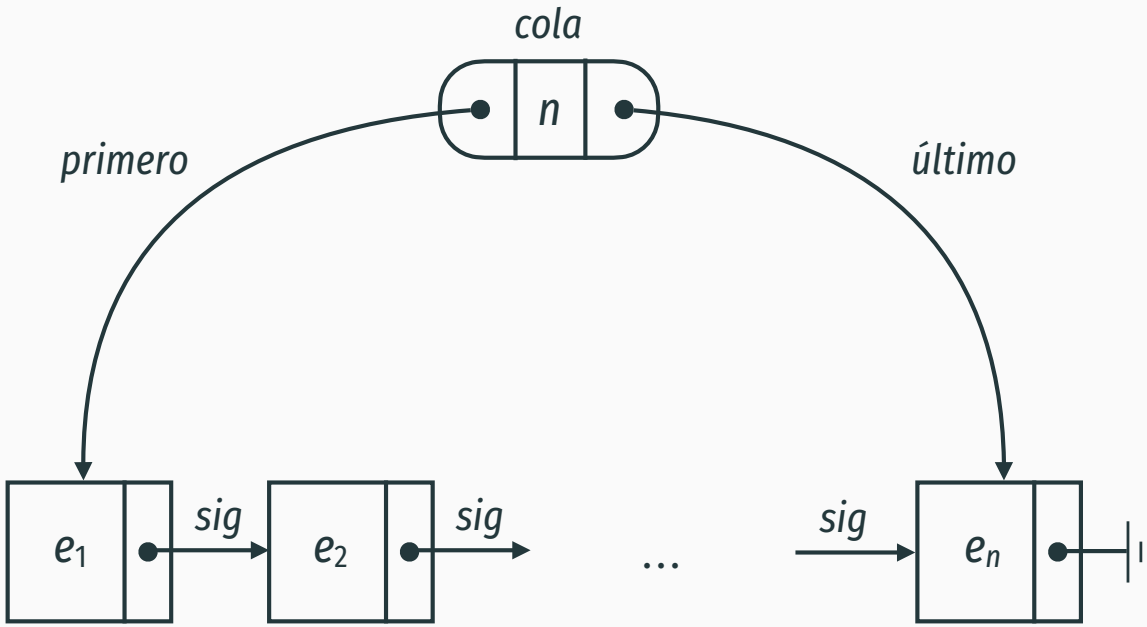
TAD de las colas

El TAD de las colas, `queue<T>`, cuenta con las siguientes operaciones:

- crear la cola vacía, `queue`
- añadir un elemento al final de la cola, `void push(T const& elem)`
- consultar el primer elemento, si existe, `T const& front() const`
- eliminar el primer elemento, si existe, `void pop()`
- determinar si la cola es vacía, `bool empty() const`

Implementación

Mediante nodos enlazados y punteros al primero y al último nodo



`queue_eda.h`

Problemas

- 06 - Duplicar una lista enlazada
- 07 - Invertir una lista enlazada

- ACR 127 - Una, dola, tela, catola...
- ACR 142 - ¿Quién empieza?
- ACR 143 - Tortitas
- ACR 198 - Evaluando expresiones

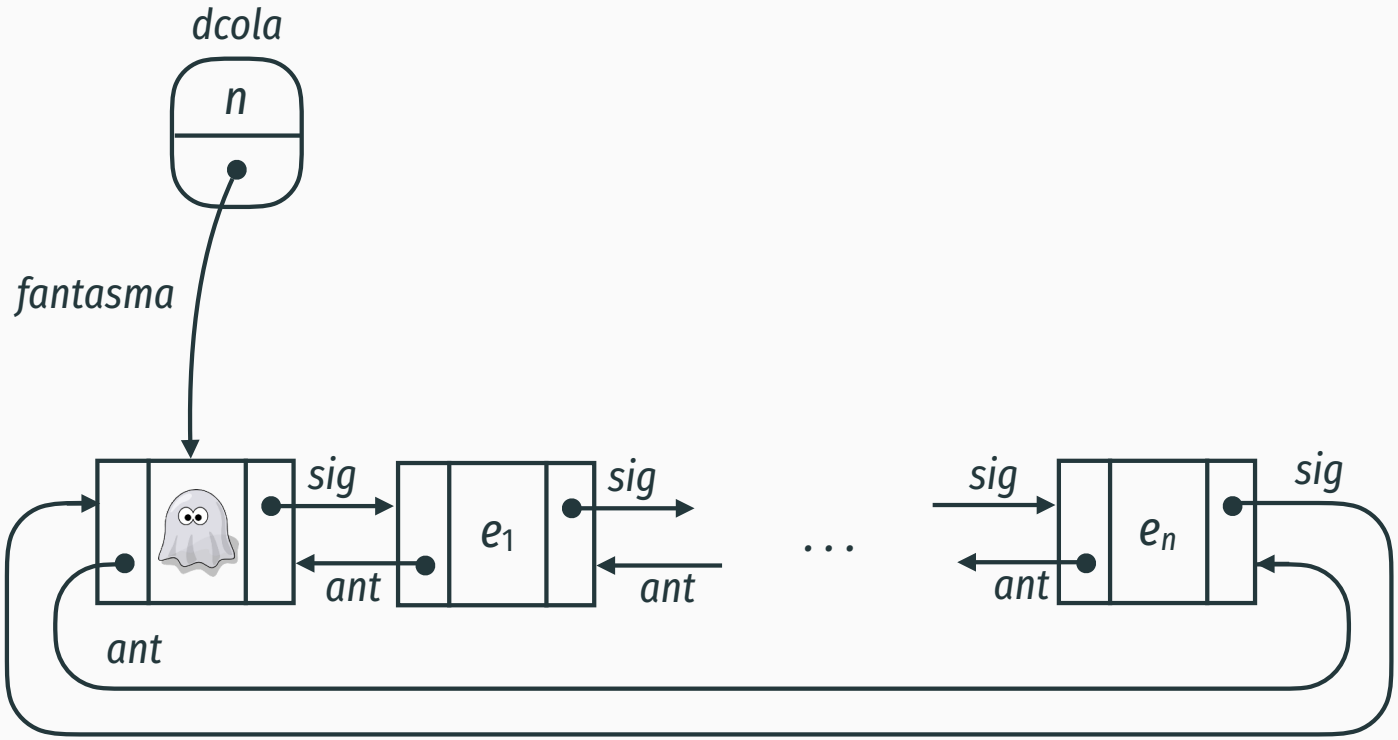
TAD de las colas dobles

El TAD de las colas dobles, `deque<T>`, cuenta con las siguientes operaciones:

- crear la cola doble vacía, `deque`
- añadir un elemento por el principio, `void push_front(T const& elem)`
- añadir un elemento por el final, `void push_back(T const& elem)`
- consultar el primer elemento, si existe, `T const& front() const`
- consultar el último elemento, si existe, `T const& back() const`
- eliminar el primer elemento, si existe, `void pop_front()`
- eliminar el último elemento, si existe, `void pop_back()`
- determinar si la cola es vacía, `bool empty() const`
- consultar el tamaño de la cola, `size_t size() const`

Implementación

Mediante nodos doblemente enlazados, nodo fantasma y circular



`deque_eda.h`

Mensajes secretos

- El *agente 0069* ha inventado un nuevo método de codificación de mensajes secretos.
- El mensaje original X se codifica en dos etapas: en primer lugar, X se transforma en X' reemplazando cada sucesión de caracteres consecutivos que no sean vocales por su inversa.
- En segundo lugar, X' se transforma en la sucesión de caracteres X'' obtenida al ir tomando sucesivamente el primer carácter de X', luego el último, luego el segundo, luego el penúltimo, etc.
- Por ejemplo,

$$
\begin{aligned}
X &= \texttt{Anacleto, agente secreto} \\
X' &= \texttt{Analceto ,agetnes erceto} \\
X'' &= \texttt{Aontaelccreet os e,natge}
\end{aligned}
$$

```
void volcar_pila(stack<char> & pila, deque<char> & dcola) {
   while (!pila.empty()) {
      dcola.push_back(pila.top());
      pila.pop();
   }
}

string codifica(string const& mensaje) {
   // primera fase, invertir consonantes entre vocales
   deque<char> X;
   stack<char> pila_consonantes; // para darles la vuelta
   for (char c : mensaje) {
      if (es_vocal(c)) {
         volcar_pila(pila_consonantes, X);
         X.push_back(c);
      } else {
         pila_consonantes.push(c);
      }
   }
   volcar_pila(pila_consonantes, X);
```

```
    // segunda fase
    string resultado;
    size_t cont = 0;
    while (!X.empty()) {
        if (cont % 2 == 0) {
            resultado.push_back(X.front()); X.pop_front();
        } else {
            resultado.push_back(X.back()); X.pop_back();
        }
        ++cont;
    }
    return resultado;
}


    assert(mensaje == descodifica(codifica(mensaje)));
```

Problemas

- ACR 197 - Mensaje interceptado
- ACR 258 - Coge el sobre y corre

TAD de las listas

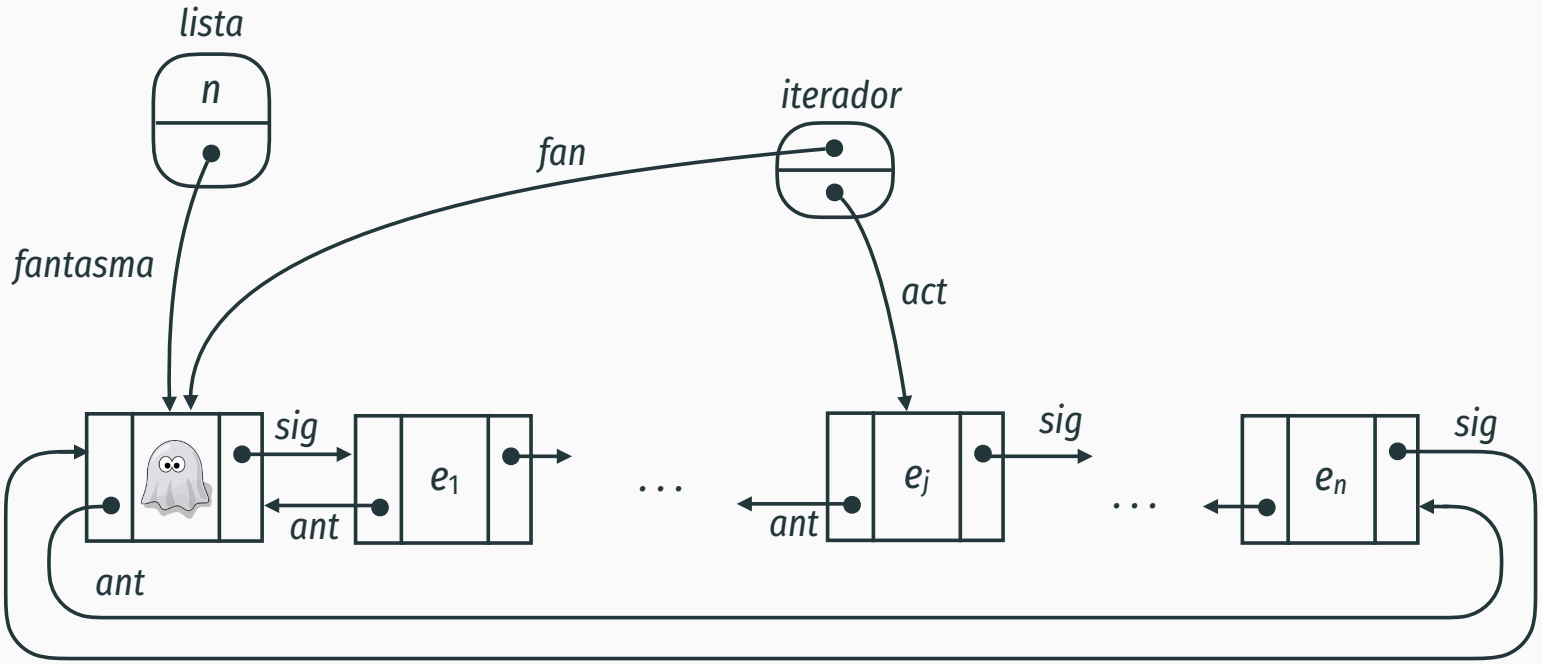
El TAD de las listas, `list<T>`, cuenta con las siguientes operaciones:

- crear la lista vacía, `list`
- añadir, consultar o eliminar un elemento por el principio o por el final,
- determinar si la lista es vacía, `bool empty() const`
- consultar el tamaño de la lista, `size_t size() const`
- consultar un elemento dada su posición, $0 \leq$ `ind` $<$ `size()`,
  `T const& at(size_t ind) const`

- obtener un iterador al comienzo de la lista, `iterator begin()`
- obtener un iterador tras el final de la lista, `iterator end()`
- insertar un elemento delante del apuntado por un iterador,
  `iterator insert(iterator const& it, T const& elem)`
- eliminar el elemento apuntado por un iterador,
  `iterator erase(iterator const& it)`

Implementación

Heredando de la estructura para colas dobles (nodos doblemente enlazados, nodo fantasma y circular). Iteradores



`list_eda.h`

Uso de iteradores

```
template <class T>
void mostrar(list<T> const& lista) {  // O(N^2)
   for (size_t i = 0; i < lista.size(); ++i)
      cout << lista.at(i);
   cout << '\n';
}

template <class T>
void mostrar(list<T> const& lista) {  // O(N)
   for (auto it = lista.cbegin(); it != lista.cend(); ++it)
      cout << *it;
   cout << '\n';
}

template <class T>
void mostrar(list<T> const& lista) {  // O(N)
   for (T const& elem : lista)
      cout << elem;
   cout << '\n';
}
```

```
int main() {
   const string digitos = "123456789";

   list<char> lista;
   for (char c : digitos)
      lista.push_back(c);
   mostrar(lista);

   for (auto it = lista.begin(); it != lista.end(); ++it)
      --(*it);
   mostrar(lista);

   for (char & c : lista)
      ++c;
   mostrar(lista);

   auto it2 = lista.begin();
   ++it2; ++it2; ++it2;
   cout << *it2 << '\n';
   it2 = lista.erase(it2); it2 = lista.erase(it2); ++it2;
   lista.insert(it2, 'z');
   it2 = lista.insert(lista.end(), 'y');
   mostrar(lista);
}
```

Función genérica para buscar el máximo

```
// devuelve un iterador apuntando al elemento mayor en [ini..fin)
template <class Iterator>
Iterator find_max(Iterator ini, Iterator fin) {
   if (ini == fin) throw range_error("intervalo vacio");
   Iterator mayor = ini;
   while (++ini != fin)
      if (*mayor < *ini)
         mayor = ini;
   return mayor;
}

template <class Iterator, class Comparator>
Iterator find_max(Iterator ini, Iterator fin, Comparator comp) {
   if (ini == fin) throw range_error("intervalo vacio");
   Iterator mayor = ini;
   while (++ini != fin)
      if (comp(*mayor, *ini))
         mayor = ini;
   return mayor;
}
```

```cpp
#include <functional>  // std::greater

int main() {
   const string digitos = "123456789";

   list<char> lista;
   for (char c : digitos)
      lista.push_back(c);

   cout << *find_max(lista.begin(), lista.end()) << '\n';

   auto it = find_max(digitos.begin(), digitos.end(), greater<char>());
   cout << *it << '\n';

   vector<int> v = { 45, 12, 123, 34 };
   cout << *find_max(v.begin(), v.end()) << '\n';

   int arr[] = { 8, 7, 6, 5 };
   cout << *find_max(arr, arr + 4) << '\n';
   cout << *find_max(begin(arr), end(arr)) << '\n';
}
```

Problemas

- 08 - Un viaje a la luna
- ACR 144 - Teclado estropeado