Diseño e implementación de TADs lineales[2]

Profesor: Isabel Pita

Facultad de Informática - UCM

21 de febrero de 2022

[2] Transparencias obtenidas del tema: TADs lineales de los apuntes de EDA desarrollado por Marco Antonio Gómez (2016), y de las transparencias e implementaciones desarrolladas por Alberto Verdejo

Estructuras de datos lineales

- Estructuras básicas
  - Vector de elementos.
  - Listas enlazadas.
- Las implementaciones de los TADs lineales pueden hacer uso de una u otra estructura básica;
- La elección de una u otra podrá influir en la complejidad de sus operaciones.

Vectores de elementos.

Definición de una clase vector: En la implementación se utilizan normalmente tres atributos:

- Puntero al array almacenado en memoria dinámica.
- Tamaño de ese array (o lo que es lo mismo, número de elementos que podría almacenar como máximo).
- Número de elementos ocupados actualmente. Los índices ocupados casi siempre se *condensan* al principio del array.

```
private:
    // Tamano inicial del vector dinamico.
   static const int TAM_INICIAL = 10;
    // Puntero al array que contiene los datos.
    T *_v;
    // tamano del vector _v.
    unsigned int _tam;
    // Numero de elementos reales guardados.
    unsigned int _numElems;
};
```

Listas enlazadas

- Cada elemento es almacenado en un espacio de memoria independiente (un *nodo*) y la colección completa se mantiene utilizando punteros.
- Alternativas:
  - Listas enlazadas simples (`linked_list_ed`): cada nodo mantiene un puntero al siguiente elemento.
  - Listas doblemente enlazadas (`double_linked_list_ed`): cada nodo mantiene dos punteros: uno al nodo siguiente y otro al nodo anterior.
- Todas las implementaciones tendrán la definición de la clase `Nodo` que almacena el elemento y los punteros al nodo siguiente y al nodo anterior (sólo en listas doblemente enlazadas).
- Este curso implementaremos la clase `Nodo` como un tipo interno del TAD que estemos definiendo.

Pilas (stack)

- 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)*.
- El comportamiento de las pilas es totalmente independiente del tipo de los datos almacenados en ellas, por lo tanto se trata de un tipo genérico.
- La ventaja de las pilas es que el acceso a la estructura, tanto para su modificación (insercción 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.

Pilas (stack)

- Operaciones:
  - Crear una pila vacía.
  - Apilar un nuevo elemento en una pila: `push`.
  - Desapilar el último elemento (parcial): `pop`.
  - Acceder al último elemento añadido (parcial): `top`.
  - Averiguar si una pila tiene elementos: (`empty`).

Pilas (stack)

Implementación de pilas con arrays dinámicos.

- Tipo representante: array dinámico de elementos.
- Relación de equivalencia Dos pilas son iguales si el número de elementos almacenados coincide y sus valores respectivos, uno a uno, también.

Implementación de las pilas

Implementación de las pilas con listas enlazadas simples.

- Tipo representante: Un puntero al nodo que contiene la cima (`_cima`). Si la pila está vacía, el puntero valdrá `nullptr`.
- Dado que lo único que hacemos con la lista es insertar y borrar el primer elemento las listas enlazadas simples son suficiente.
- Invariante de la representación: debe garantizar
  - que la secuencia de nodos termina en `nullptr` (eso garantiza que no hay ciclos) y
- Relación de equivalencia: dos objetos pila serán iguales si su lista enlazada contiene el mismo número de elementos y sus valores uno a uno coinciden (están en el mismo orden):

- La complejidad de las operaciones de ambas implementaciones es similar:

Operación	Vectores	Listas enlazadas
`stack`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`push`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`pop`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`top`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`empty`	$\mathcal{O}(1)$	$\mathcal{O}(1)$

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)*.
- Como las pilas, 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.

Colas

Operaciones:

- Crear una cola vacía.
- Añadir un nuevo elemento a la cola: `push`.
- Eliminar el primer elemento de la cola. Falla si la cola está vacía: `pop`.
- Devolver el primer elemento de la cola (el más antiguo). Falla si la cola está vacía: `front`
- Averiguar si la cola tiene elementos: `empty`

Implementación de colas con un vector

Tipo representante:

- Vector dinámico, el primer elemento de la cola está siempre en la posición 0 del vector.
- Inconveniente: el coste de la operación `pop` está en $\mathcal{O}(n)$, ya que se deben desplazar todos los elementos válidos una posición a la izquierda.
- Solución: Colas circulares.

Implementación de colas con una lista enlazada

- Tipo representante:
  - una lista enlazada simple en la que el primer nodo contiene el elemento que hay en la cabecera de la cola,
  - un puntero al primer nodo y
  - otro puntero al último nodo.
  - La cola vacía se representa con los dos punteros a `nullptr`.
- Invariante de la representación: similar al de las pilas.
- Relación de equivalencia: similar a las pilas.

- Para evitar el caso especial de la cola vacía en la implementación de las funciones se puede utilizar un *nodo fantasma* o *cabecera* que no guarda ningún elemento.
- La complejidad de las operaciones no varía, pero su programación es más sencilla.
- La misma técnica la utilizaremos posteriormente para las colas dobles.

- Complejidad de las operaciones con cada representación:

Operación	Vectores	Vectores circulares	Listas enlazadas
`queue`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`push`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`front`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`pop`	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`empty`	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Colas dobles (Deque)

- Las colas dobles son una generalización de las colas que permiten operar en los dos extremos. Las operaciones serán:
  - Crear una cola doble vacía.
  - Añadir un nuevo elemento al final: `push_back`.
  - Añadir un nuevo elemento al principio: `push_front`.
  - Eliminar el primer elemento de la cola. Falla si la cola está vacía: `pop_front`.
  - Devolver el primer elemento de la cola (el más antiguo). Falla si la cola está vacía: `front`.
  - Eliminar el último elemento de la cola. Falla si la cola está vacía: `pop_back`.
  - Devolver el último elemento de la cola. Falla si la cola está vacía: `back`.
  - Comprobar si la cola tiene elementos: `empty`.

- Tipo representante:
  - Lista circular doblemente enlazada con nodo cabecera.
  - Un puntero al nodo cabecera.
  - La cola vacía está representada por un nodo cabecera que no contiene ningún elemento y cuyos punteros anterior y siguiente apuntan a él mismo.
  - La implementación hará que *el siguiente* al nodo fantasma sea el primero de la cola (la cabecera), mientras que el *anterior* será el último.
- Relación de equivalencia dos objetos son iguales si las listas enlazadas contienen el mismo número de elementos y sus valores uno a uno coinciden, empezando en el nodo cabecera y exceptuando este.

- Invariante de la representación:
  - El conjunto de nodos alcanzables desde el nodo cabecera por un lado y por otro debe ser el mismo.
  - Dado que la lista es circular, el nodo cabecera debe aparecer en el conjunto de nodos alcanzables a partir de él.
  - Todos esos nodos deben estar ubicados y tener los enlaces al nodo anterior y al nodo siguiente correctos (lo que implica que si vamos al nodo anterior de n y luego pasamos a su siguiente deberíamos volver a n y al contrario).
  - Por último, todos los nodos (excepto el nodo cabecera) deben contener elementos válidos del tipo base.

- La complejidad de las operaciones en esta implementación es:

Operación	Listas enlazadas
`Deque`	$\mathcal{O}(1)$
`push_back`	$\mathcal{O}(1)$
`front`	$\mathcal{O}(1)$
`pop_front`	$\mathcal{O}(1)$
`push_front`	$\mathcal{O}(1)$
`back`	$\mathcal{O}(1)$
`pop_back`	$\mathcal{O}(1)$
`empty`	$\mathcal{O}(1)$

Listas

- Las listas son la estructura lineal más general, su única característica es imponer un orden relativo entre los elementos almacenados en ellas.
- Permiten la consulta y modificación de los dos extremos (como las colas dobles) pero también recorrerlas y actuar en cualquier punto intermedio utilizando *iteradores*.

Listas. Implementación

- Tipo representante: lista doblemente enlazada con nodo cabecera y circular; un entero con el número de elementos de la lista; y un iterador al nodo que se está recorriendo.

- Función de abstracción y la relación de equivalencia son similares a las vistas para las colas dobles.

- La complejidad de las operaciones es la misma que para las colas dobles:

Operación	Listas enlazadas
`list`	$\mathcal{O}(1)$
`push_front`	$\mathcal{O}(1)$
`front`	$\mathcal{O}(1)$
`pop_front`	$\mathcal{O}(1)$
`push_back`	$\mathcal{O}(1)$
`back`	$\mathcal{O}(1)$
`pop_back`	$\mathcal{O}(1)$
`empty`	$\mathcal{O}(1)$
`size`	$\mathcal{O}(1)$

Recorridos de listas

- Las listas no tienen acceso directo a los elementos como los vectores. Una operación sobre las listas que acceda al elemento n-ésimo, `lista.at(n)`, debe recorrer internamente la lista desde el comienzo hasta la posición buscada, tiene coste $\mathcal{O}(n)$.
- Con una operación como la anterior, un bucle que escribe uno a uno todos los elementos tiene coste cuadrático.

```
list<int> l;
for (int i = 0; i < l.size(); ++i)
  std::cout << l.at(i) << '\n';
```

- Para evitar comenzar a recorrer la lista en cada iteración del bucle y por lo tanto el coste cuadrático, se utilizan *iteradores*.

Iteradores

- Un *iterador* es un objeto de una clase que:
  - Representa un punto intermedio en el recorrido de una colección de datos (una lista en este caso).
  - Tiene un operador $*$ que devuelve el elemento por el que va el recorrido (y tendrá el tipo base utilizado en la colección). La operación será *parcial* si el recorrido ya ha terminado.
  - Tiene un operador ++ que hace que el iterador pase al siguiente elemento del recorrido.
  - Tiene implementada la operación de comparación `==`, `!=` , de forma que se puede saber si dos iteradores son iguales o diferentes. Dos iteradores son iguales si: representan el mismo punto en el recorrido de una lista concreta o los dos representan el final del recorrido.

Operaciones de manejo de iteradores

Operaciones de las listas para el manejo de los iteradores:

- Obtener un iterador al comienzo de la lista, `iterator begin()` o `iterator cbegin()`.
- Obtener un iterador al final de la lista, `iterator end()` o `iterator cend()`.
- Insertar un elemento delante del apuntado por un iterador, `iterator insert (iterator cons& it, T const& elem)`.
- Eliminar el elemento apuntado por un iterador `iterator erase (iterator const& it)`.

Uso de iteradores

Mostrar los elementos de una lista

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

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

Uso de iteradores

```
class incrementa {
  public:
    char operator() (char a) {
      return ++a;
    }
};
template <typename T, typename OP>
void aplicar (list<T> const& lista, OP trans){ // O(N)
  for (T & elem : lista)
    trans(elem);
  }
int main() {
  std::list<char> lista;
  // ... crear la lista con valores
  //aplicar(lista, [](char & c) {++c});
  aplicar(lista, incrementa());
  mostrar(lista);
}
```

Uso de iteradores

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

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

Función genérica para buscar el máximo

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

Función genérica para buscar el máximo

```
#include <functional> // std::greater
int main() {
  const std::string digitos = "123456789";
  std::list<char> lista;
  for (char c : digitos) lista.push_back(c);

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

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

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

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

Peligros de los iteradores

- El uso de iteradores conlleva un riesgo debido a la existencia de *efectos laterales* en las operaciones, ya que un iterador abre la puerta a acceder a los elementos de la lista *desde fuera* de la propia lista.
- Cambios que ocurran en la lista pueden afectar al resultado de las operaciones del iterador.
- Por ejemplo el código siguiente fallará:

```
std::list<int> lista;
lista.push_front(3);
std::list<int>::iterator it = lista.begin();
lista.pop_front();  // Quitamos el primer elemento
std::cout << *it << '\n'; // Accedemos a el... CRASH
```