CI2612: Algoritmos y Estructuras de Datos II

Blai Bonet

Universidad Simón Bolívar, Caracas, Venezuela

Estructuras de datos elementales

Objetivos

- Pilas y colas
- Listas enlazadas
- Implementación de apuntadores y objetos
- Representación de árboles con raíz (enraizados)

Pilas y colas

Las pilas y colas son EDs que solo soportan inserción/eliminación de elementos y la eliminación es **restringida:**

– en las pilas solo puede eliminarse el último elemento insertado: k eliminaciones sucesivas eliminan los últimos k elementos insertados

– en las colas solo puede eliminarse el primer elemento insertado: k eliminaciones sucesivas eliminan los primeros k elementos insertados

En ambos casos, si decimos que un elemento entra cuando es insertado y sale cuando es eliminado, entonces:

– una pila implementa un esquema **LIFO** por *"Last In, First Out"*

– una cola implementa un esquema **FIFO** por *"First In, First Out"*

Implementación de pilas con arreglos

Es fácil implementar una pila con **capacidad** de n elementos con un arreglo de dimensión n

El arreglo tiene los atributos: `length` que denota la dimensión del arreglo, y `top` que denota el número de elementos en la pila

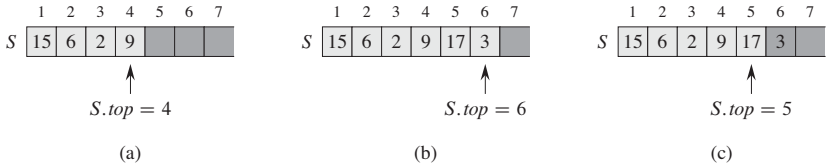

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

Implementación de pilas con arreglos

En las pilas, la inserción se llama `Push` y la eliminación `Pop`

```
1  Stack-Empty(array S)
2      return S.top == 0
3
4  Stack-Push(array S, pointer x)
5      if S.top == S.length then error "stack overflow"
6      S.top = S.top + 1
7      S[S.top] = x
8
9  Stack-Pop(array S)
10     if Stack-Empty(S) then error "stack underflow"
11     S.top = S.top - 1
12     return S[S.top + 1]
```

Estas operaciones toman **tiempo constante**

Implementación de colas con arreglos

Una cola de capacidad n elementos también puede implementarse con un arreglo de dimensión n con atributos `head`, `tail` y `nelements`:

- `head` apunta al primer elemento de la cola (aquel que será removido en la próxima eliminación)
- `tail` apunta al lugar donde el siguiente elemento será insertado. Los elementos en la cola se encuentran en las posiciones `head`, `head` $+ 1, \ldots,$ `tail` $- 1$ de **forma circular**: el siguiente de la posición `length` es la posición 1
- `nelements` mantiene el número de elementos en la cola

Implementación de colas con arreglos

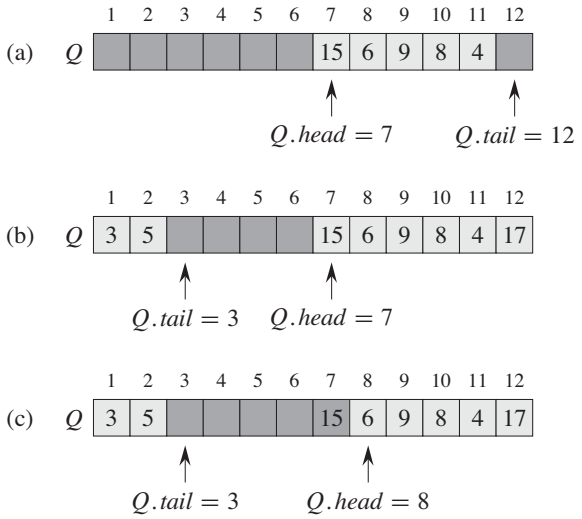

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

Implementación de colas con arreglos

En las colas, la inserción se llama Enqueue y la eliminación Dequeue

```
1  Queue-Empty(array Q)
2      return Q.nelements == 0
3
4  Enqueue(array Q, pointer x)
5      if Q.nelements == Q.length then error "queue overflow"
6      Q[Q.tail] = x
7      Q.tail = Q.tail + 1
8      if Q.tail > Q.length then Q.tail = 1
9      Q.nelements = Q.nelements + 1
10
11 Dequeue(array Q)
12     if Queue-Empty(Q) then error "queue underflow"
13     x = Q[Q.head]
14     Q.head = Q.head + 1
15     if Q.head > Q.length then Q.head = 1
16     Q.nelements = Q.nelements - 1
17     return x
```

Estas operaciones toman **tiempo constante**

Lista enlazada

Una lista enlazada es una ED en donde los objetos se guardan en orden lineal utilizando **apuntadores**

En una **lista doble-enlazada**, cada objeto tiene, además de los datos satélite, una clave y dos atributos `prev` y `next` que apuntan a los elementos anterior y próximo en la lista. Si `x.prev` es null, el elemento `x` es el primero de la lista, y `x` es el último cuando `x.next` es null

La ED es representada por un objeto L que tiene atributo `head` el cual apunta al primero de la lista

Lista doble-enlazada

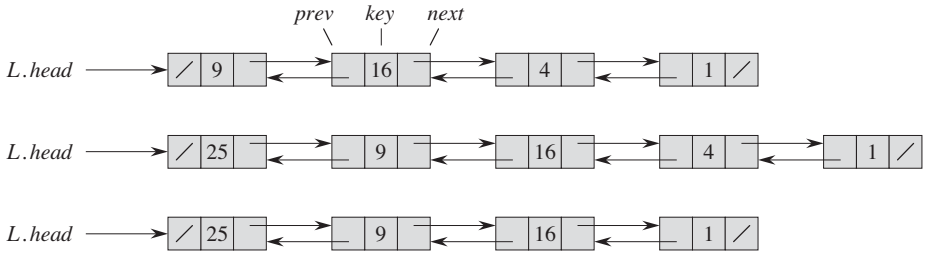

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

Tipos de listas enlazadas

Existen varios tipos de listas enlazadas. Las más comunes son:

- **simplemente enlazada:** cada elemento tiene un atributo `next` que apunta al próximo de la lista
- **doblemente enlazada:** cada elemento tiene dos atributos `prev` y `next` que apuntan al anterior y próximo de la lista respectivamente
- **ordenada:** los elementos están ordenados según las claves
- **circular:** los elementos están ordenadas de forma circular (el próximo del "último" es el "primero" y el anterior al "primero" es el "último")

A continuación asumimos listas doblemente enlazadas, no ordenadas y no circulares

Búsqueda en una lista enlazada

`List-Search(L,k)` busca en la lista L el primer elemento que tenga clave igual a k y retorna un apuntador a tal elemento si lo encuentra y `null` si no existe

```
1  List-Search(list L, key k)
2      x = L.head
3      while x != null & x.key != k do
4          x = x.next
5      return x
```

En el peor caso, la clave buscada no se encuentra en la lista o se encuentra en el último elemento por lo que `List-Search` toma tiempo $\Theta(n)$ en el peor caso donde n es el número de elementos en L

Inserción en una lista enlazada

Para insertar un elemento en una **lista no ordenada**, basta colocarlo al "frente" de la lista

```
1  List-Insert(list L, pointer x)
2      x.next = L.head
3      if L.head != null then L.head.prev = x
4      x.prev = null
5      L.head = x
```

`List-Insert` toma tiempo constante

Eliminación en una lista enlazada

Para eliminar un elemento de la lista necesitamos un apuntador al elemento

La idea es "pegar" lo anterior al elemento con lo próximo al elemento, pero tenemos que considerar los **casos borde**

```
1  List-Delete(list L, pointer x)
2      if x.prev != null then x.prev.next = x.next
3      if x.next != null then x.next.prev = x.prev
4      if x.prev == null then L.head = x.next
```

`List-Delete` toma tiempo constante

Uso de sentinelas

Un sentinela es un objeto **espurio** que se utiliza como "marcador" dentro de la lista y que permite simplificar la implementación de operaciones (sobretodo los casos bordes)

Mostramos como utilizar un sentinela insertado de forma **circular** entre el primero y último de la lista

El sentinela es apuntado por `L.nil` y es tal que `L.nil.next` es el primero de la lista y `L.nil.prev` es el último de la lista. También:

- `L.nil.next.prev = L.nil`
- `L.nil.prev.next = L.nil`

El apuntador `L.head` no se necesita

Listas circulares con sentinelas

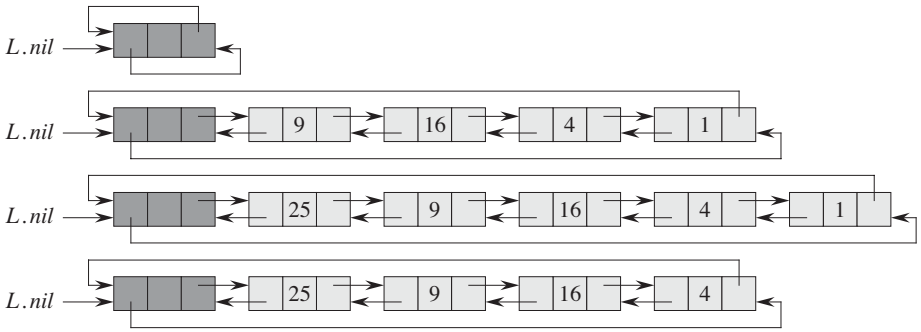

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

Uso de sentinelas

```
1  List-Search'(list L, key k)
2      x = L.nil.next
3      while x != L.nil & x.key != k do
4          x = x.next
5      return x
6
7  List-Insert'(list L, pointer x)
8      x.next = L.nil.next
9      L.nil.next.prev = x
10     L.nil.next = x
11     x.prev = L.nil
12
13 List-Delete'(list L, pointer x)
14     x.prev.next = x.next
15     x.next.prev = x.prev
```


Implementación de apuntadores

Mostramos como implementar apuntadores y atributos de objetos en lenguajes de programación que solo soporta arreglos (e.g. FORTRAN)

Representación de objetos con arreglos múltiples

Una colección de n objetos con m atributos puede representarse con m arreglos de dimensión n, un arreglo por atributo

Por ejemplo, para representar una lista con los atributos `key`, `prev` y `next` utilizamos 3 arreglos de dimensión n:

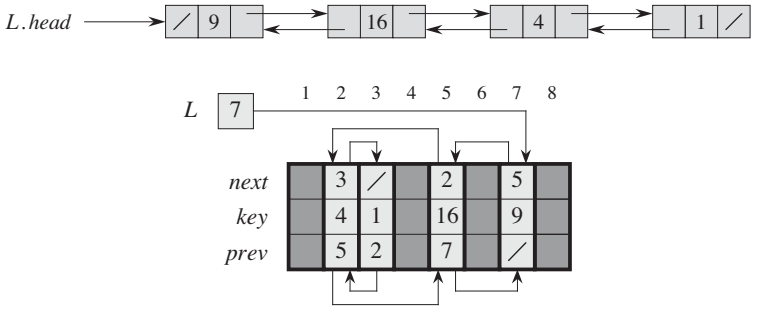

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

Representación de objetos con arreglo único

La memoria en un computador se direcciona como un arreglo, desde la posición 1 hasta la posición M

Utilizamos la misma idea para guardar todos los objetos en un arreglo A suficientemente grande. Cuando todos los objetos tienen el **mismo tamaño** (i.e. mismo número de atributos), cada objeto se guarda en un segmento contiguo $A[j \ldots k]$ de igual tamaño

Dado el comienzo j de un objeto, los atributos son guardados en **desplazamientos fijos** a partir de j. En el caso de la lista, cada objeto tiene 3 elementos y podemos utilizar el desplazamiento 0 para guardar `key`, 1 para guardar `next` y 2 para guardar `prev`

Por ejemplo, para leer el atributo `prev` del objeto apuntado por el índice i, leemos la entrada $A[i + 2]$ en el arreglo

Representación de objetos con arreglo único

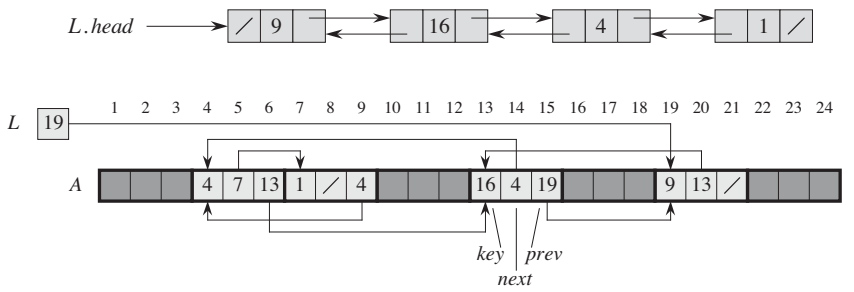

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

Reclamando y liberando espacio (objetos)

Cada vez que se inserta un objeto en un conjunto dinámico se necesita reservar/reclamar ("allocate") espacio para guardar el objeto, y cada vez que se elimina el objeto se debe liberar el espacio utilizado

Para el caso sencillo de objetos homogéneos, mostramos como implementar el **manejo de espacio** para el caso de una **lista doble enlazada representada con 3 arreglos**

Suponga que los arreglos tienen dimensión m y la lista contiene $n \leq m$ elementos: cada arreglo contiene $m - n$ posiciones no utilizadas o libres

Los objetos libres se mantienen en una lista llamada **"free list"** de la cual se obtiene el espacio para los objetos nuevos

Free list

Free list es una lista simplemente enlazada que se mantiene con el arreglo `next`

La lista libre se utiliza como una pila:

- cada vez que se necesita espacio para un objeto nuevo, el primer elemento de **free list** es utilizado
- cada vez que se elimina un objeto, el espacio es retornado al frente de **free list**

Free list

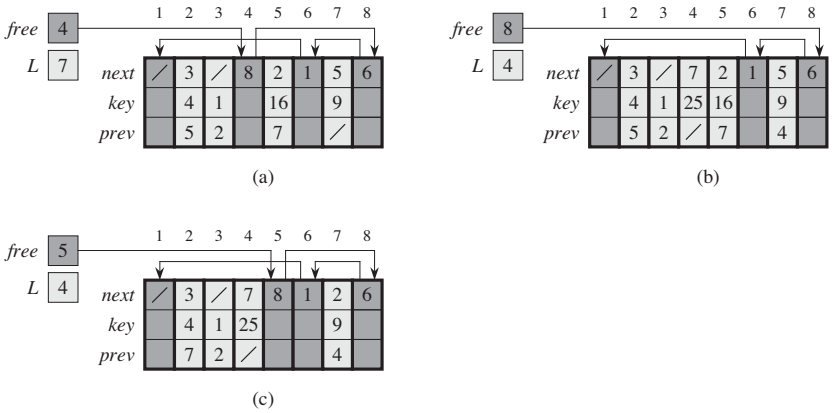

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

Manejo del free list

```
1  Allocate-Object()
2      if free == null then error "out of space"
3      x = free
4      free = free.next
5      return x
6
7  Free-Object(pointer x)
8      x.next = free
9      free = x
```


Representación de árboles enraizados

Las técnicas de representación de objetos y apuntadores las podemos utilizar para representar árboles enraizados. Mostramos como representar:

- árboles binarios
- árboles con número variable de hijos

Árboles binarios

Para un árbol binario basta mantener 3 atributos por cada nodo:

- atributo `p` que apunta al nodo "padre" del nodo
- atributo `left` que apunta al hijo izquierdo del nodo
- atributo `right` que apunta al hijo derecho del nodo

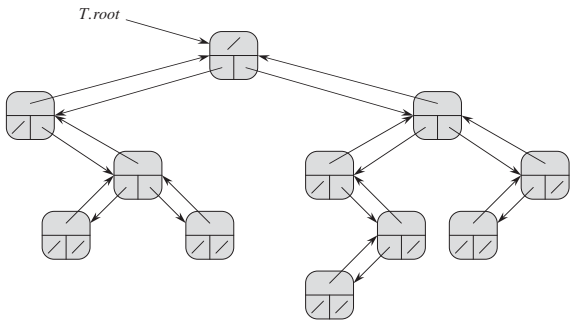

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

Árboles con número variable de hijos

Para árboles con un número variable de hijos, utilizamos 3 atributos:

- atributo `p` que apunta al nodo "padre" del nodo
- `left-child` que apunta al hijo más a la izquierda del nodo
- `next-sibling` que apunta al "hermano" a la derecha del nodo

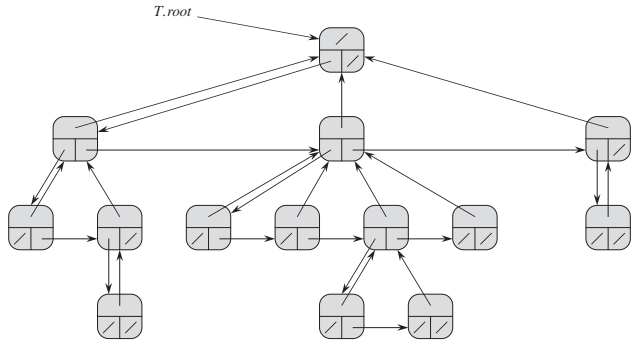

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

Ejercicios (1 de 5)

1. ¿Cómo tienen que ser inicializados los atributos en una pila y cola vacía?
2. (10.1-1) Ilustre la siguiente secuencia de operaciones sobre una pila vacía implementada en un arreglo $S[1 \ldots 6]$: `Push(S,4)`, `Push(S,1)`, `Push(S,3)`, `Pop(S)`, `Push(S,8)`, `Pop(S)`
3. (10.1-2) Explique como implementar dos pilas sobre un arreglo $A[1 \ldots n]$ de tal forma que ninguna inserción haga "overflow" a menos que el número total de elementos en las dos pilas sea n. Todas las operaciones deben correr en tiempo constante
4. (10.1-3) Ilustre la siguiente secuencia de operaciones sobre una cola vacía implementda en un arreglo $Q[1 \ldots 6]$: `Enqueue(Q,4)`, `Enqueue(Q,1)`, `Enqueue(Q,3)`, `Dequeue(Q)`, `Enqueue(Q,8)`, `Dequeue(Q)`

Ejercicios (2 de 5)

5. (10.1-5) Mientras las colas y pilas permiten la inserción/eliminación de elementos en un solo lado, una **deque** (double-ended queue) permite la inserción/eliminación de elemenetos en ambos lados. Escriba 4 operaciones de tiempo constante para insertar/eliminar elementos de ambos lados de un dequeu implementado sobre un arreglo
6. (10.1-6) Asumiendo que solo tiene pilas disponibles (no arreglos), muestre como implementar una cola utilizando dos pilas. Analice el tiempo de ejecucion de cada operación de pila
7. (10.1-7) Asumiendo que solo tiene colas disponibles (no arreglos), muestre como implementar una pila utilizando dos colas. Analice el tiempo de ejecucion de cada operación de pila
8. (10.2-1) ¿Se puede implementar una operación `Insert` de tiempo constante en una lista simplemente enlazada? ¿Y qué de `Delete`?

Ejercicios (3 de 5)

9. (10.2-2) Implemente una pila utilizando una lista *simplemente enlazada*. Las operaciones `Push` y `Pop` deben corren en tiempo constante
10. (10.2-2) Implemente una cola utilizando una lista *simplemente enlazada*. Las operaciones `Enqueue` y `Dequeue` deben corren en tiempo constante
11. (10.2-4) Modifique `List-Search'(L,k)` de forma tal que la condición de lazo solo sea `x.key != k`
12. (10.2-5) Implemente un diccionario utilizando una lista circular simplemente enlazada. ¿Cuáles son los tiempos de corrida de las operaciones?
13. (10.2-6) La operacion `Union` de dos conjuntos dinámicos S_1 y S_2 retorna un conjunto dinámico $S_1 \cup S_2$. Los conjuntos S_1 y S_2 son destruidos durante la operación. Muestre como soportar dicha operación en tiempo constante utilizando una lista de tipo apropiado

Ejercicios (4 de 5)

14. (10.2-7) Diseñe un procedimiento iterativo de $\Theta(n)$ tiempo que invierta el orden de los elementos en una lista simplemente enlazada con n elementos. El procedimiento debe utilizar espacio constante adicional al espacio de la lista

15. (10.3-2) Escriba `Allocate-Object` y `Free-Object` para una colección homogenea de objetos implementados con un arreglo único

16. (10.3-5) Sea L una lista doble enlazada con n elementos guardada en 3 arreglos de largo m. Suponga que los arreglos son manejados por `Allocate-Object` y `Free-Object` con una **lista doble enlazada** F de libres, y que solo n objetos de los arreglos son utilizados (i.e. por los objetos en L). Escriba `Compatify-List(L,F)` que dadas la listas L y F mueve los elementos en L y F tal que las primeras n posiciones de los arreglos guarden los elementos de L y la lista F de libres refieraa a las últimas $m - n$ posiciones en los arreglos. El procedimiento debe correr en tiempo $O(n)$ y utilizar espacio constante

Ejercicios (5 de 5)

17. (10.4-2) Escriba un procedimiento **recursivo** que corra en tiempo $O(n)$ e imprima todas las claves en un árbol binario de n elementos

18. (10.4-3) Escriba un procedimiento **iterativo** que corra en tiempo $O(n)$ e imprima todas las claves en un árbol binario de n elementos. Su procedimiento debe utilizar una pila

19. Escriba un procedimiento **iterativo** que corrar en tiempo $O(n)$ y cuente el número de nodos en un árbol binario. Su procedimiento no puede modificar el árbol y debe utilizar espacio constante

20. Escriba un procedimiento **iterativo** que corrar en tiempo $O(n)$ y cuente el número de nodos en un árbol con número variable de hijos. Su procedimiento no puede modificar el árbol y debe utilizar espacio constante