

Otras estructuras de datos

Programación de Sistemas de Telecomunicación Informática II

GSyC

Universidad Rey Juan Carlos

Noviembre 2017



©2017 Grupo de Sistemas y Comunicaciones.
Algunos derechos reservados.
Este trabajo se distribuye bajo la licencia
Creative Commons Attribution Share-Alike
disponible en <http://creativecommons.org/licenses/by-sa/3.0/es>

Contenidos

- 1 Árboles autoequilibrados
- 2 Pilas y Colas
- 3 Tabla Hash
- 4 Listas doblemente enlazadas, Listas enlazadas circulares
- 5 Biblioteca predefinida de contenedores de Ada 2005
- 6 Bibliografía

Contenidos

- 1 Árboles autoequilibrados
- 2 Pilas y Colas
- 3 Tabla Hash
- 4 Listas doblemente enlazadas, Listas enlazadas circulares
- 5 Biblioteca predefinida de contenedores de Ada 2005
- 6 Bibliografía

Rendimiento de la búsqueda en un ABB (I)

- El rendimiento de las búsquedas (para insertar/extraer) en un árbol de búsqueda binaria (ABB) depende de cuán equilibrada sea su estructura
- Y el equilibrio depende del orden de inserción de nuevos nodos y del orden en el que se han ido produciendo las extracciones

Ejemplo:

```
Insertar("zappos.com" , "66.209.92.150");  
Insertar("yelp.com" , "63.251.52.110");  
Insertar("xing.com" , "213.238.60.19");  
Insertar("wings.com" , "12.155.29.35 ");  
Insertar("viacom.com" , "206.220.43.92");  
Insertar("ucla.edu" , "169.232.55.22");  
Insertar("nbc.com" , "66.77.124.26");  
Insertar("google.com" , "66.249.92.104");  
Insertar("facebook.com", "69.63.181.12");  
Insertar("edi.com" , "192.86.2.98");  
Insertar("cbs.com" , "198.99.118.37");  
Insertar("bbva.es" , "195.76.187.83");
```

Rendimiento de la búsqueda en un ABB (II)

Peor caso: árbol completamente desequilibrado

- En el peor caso cada nodo tiene exactamente un enlace no nulo, salvo un único nodo que tiene sus dos enlaces nulos (**nodo hoja**).
- En este caso el árbol tiene la estructura de una lista enlazada y por tanto no se beneficia de la búsqueda binaria.
- Ocurre cuando p.ej. insertamos los nodos siguiendo el orden de la clave.
- También los borrados pueden ocasionar desequilibrios en el árbol.

Rendimiento de la búsqueda en un ABB (IV)

Mejor caso: árbol perfectamente equilibrado

- Árbol perfectamente equilibrado de N nodos:
 - Todos los nodos hoja están a la misma distancia de la raíz: $\log_2 N$. Hay $\log_2 N$ nodos entre el nodo raíz y un nodo hoja
 - Todo nodo del árbol, o tiene dos hijos no nulos (nodo hoja), o sus dos hijos son nulos
- El coste de una búsqueda de un elemento no existente (para extraer o insertar) es $= \log_2 N$
 - Medimos el coste como el número de nodos del árbol que hay que consultar
- El coste de una búsqueda de un elemento que está en el árbol (para extraer o insertar) es $\leq \log_2 N$

Rendimiento de la búsqueda en un ABB (III)

Caso promedio:

- Si se insertan/borran elementos en un ABB de manera aleatoria de forma que el orden de inserción/borrado NO se corresponda con el orden de las claves, el árbol tiende a permanecer en equilibrio.
- El coste de una inserción o una extracción aleatorias en un ABB en el que todas las inserciones y extracciones se han realizado de manera aleatoria es $O(2\log_2 N)$
 - $O(2\log_2 N) =$ aproximadamente $2\log_2 N$ para N grandes

Árboles binarios autoequilibrados (I)

- Los árboles binarios **autoequilibrados** son árboles de búsqueda binaria que evitan el peor caso, manteniendo el árbol casi perfectamente equilibrado.
- La inserción y el borrado en los árboles autoequilibrados se realiza de manera tal que no sólo se mantenga el orden de las claves, sino también el equilibrio.
- Lo hacen con independencia del orden de inserción/extracción, garantizando un tiempo de búsqueda logarítmico.

Árboles binarios autoequilibrados (II)

- Los árboles 2-3, los árboles rojo-negros y los árboles AVL son árboles binarios autoequilibrados
- Un **árbol 2-3**:
 - o está vacío
 - o es un 2-nodo, con una clave y dos enlaces a un subárbol 2-3 izquierdo con claves menores y a un subárbol 2-3 derecho con claves mayores
 - o es un 3-nodo, con 2 claves y tres enlaces: a un subárbol 2-3 izquierdo con claves menores a las 2 del nodo, a un subárbol 2-3 central con claves entre las 2 del nodo y a un subárbol 2-3 derecho con claves mayores a las 2 del nodo
- Los **árboles rojo-negros** son equivalentes a los árboles 2-3 pero permiten una implementación más eficiente que éstos
- Los **árboles AVL** fueron los primeros árboles binarios autoequilibrados que se inventaron.

Contenidos

- 1 Árboles autoequilibrados
- 2 Pilas y Colas**
- 3 Tabla Hash
- 4 Listas doblemente enlazadas, Listas enlazadas circulares
- 5 Biblioteca predefinida de contenedores de Ada 2005
- 6 Bibliografía

Introducción

- La **pila (stack)** y la **cola (queue)** son estructuras de datos que sirven para almacenar colecciones de elementos
 - Los elementos NO tienen una clave, como si ocurre en las tablas de símbolos
- En pilas y colas se pueden realizar dos operaciones básicas:
 - Insertar un elemento en la pila/cola
 - Extraer un elemento de la pila/cola
- Pila y cola se diferencian en cómo se elige el elemento a extraer.
 - En la tabla de símbolos se busca por clave el elemento a insertar/extraer. En las pilas y colas NO

Pila (*stack*)

Una **pila** es una colección en la que se pueden insertar y extraer elementos.

2 operaciones básicas: Push, Pop

- **Push** inserta un nuevo elemento en la pila
- **Pop** extrae el último elemento que se insertó en la pila
 - La extracción sigue el orden **LIFO** (*Last-In First-Out*)

Casos de uso de las pilas

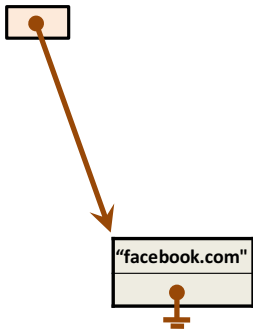
- Cada vez que un usuario selecciona un enlace en una página Web que está mostrando el navegador, la página se inserta en una **pila de páginas Web visitadas**.
 - Cada vez que pulsa el botón de volver atrás en el navegador Web se extrae de la pila de páginas Web visitadas la última que se insertó
- Las **llamadas a subprogramas** también utilizan una pila:
 - para ir insertando la información local (parámetros, variables locales) cada vez que se produce una nueva llamada a un subprograma
 - para extraer la información local a cada subprograma cada vez que retorna de una llamada

Implementación de la pila

- Implementación con Array
 - Problema: el Array tiene un tamaño máximo fijado de antemano
 - Solución: cuando el Array se llena se copia a otro un poco más grande.
- Implementación con Lista enlazada
 - De manera muy similar a como implementamos la tabla de símbolos con una lista enlazada:
 - No hay clave
 - **Push** \simeq Put de la tabla de símbolos (recuerda, insertaba por el principio)
 - **Pop**: devuelve First y actualiza First a First.Next.

Ejemplo

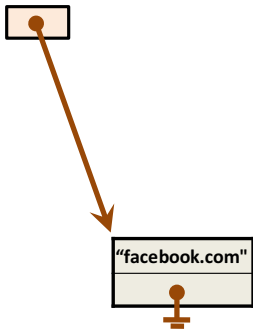
My_Stack



Ejemplo

```
1 Stacks.Push (My_Stack, ASU.To_Unbounded_String ("google.com"));
```

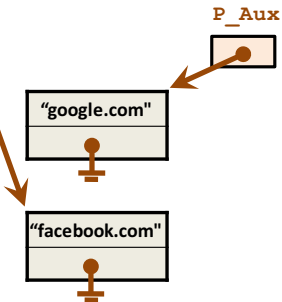
My_Stack



Ejemplo

```
1 Stacks.Push (My_Stack, ASU.To_Unbounded_String ("google.com"));
```

My_Stack



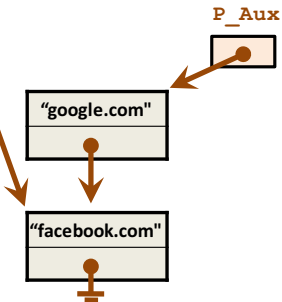
P_Aux



Ejemplo

```
1 Stacks.Push (My_Stack, ASU.To_Unbounded_String ("google.com"));
```

My_Stack



Ejemplo

```
1 Stacks.Push (My_Stack, ASU.To_Unbounded_String ("google.com"));
```

My_Stack



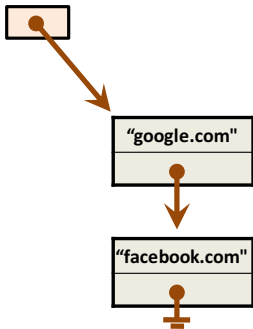
P_Aux



Ejemplo

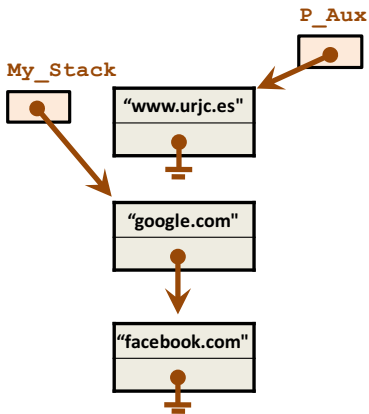
```
2 Stacks.Push (My_Stack, ASU.To_Unbounded_String ("www.urjc.es"));
```

My_Stack



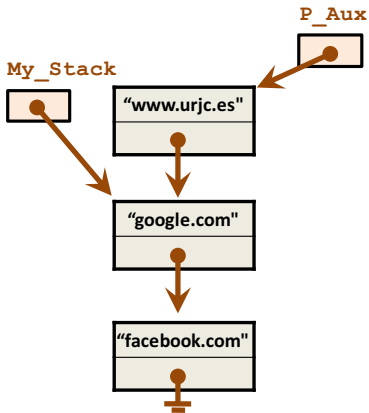
Ejemplo

```
2 Stacks.Push (My_Stack, ASU.To_Unbounded_String ("www.urjc.es"));
```



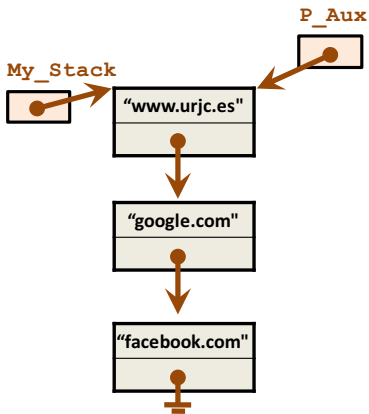
Ejemplo

```
2 Stacks.Push (My_Stack, ASU.To_Unbounded_String ("www.urjc.es"));
```



Ejemplo

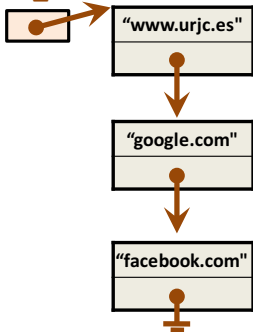
```
2 Stacks.Push (My_Stack, ASU.To_Unbounded_String ("www.urjc.es"));
```



Ejemplo

3 URL := Stacks.Pop (My_Stack);

My_Stack



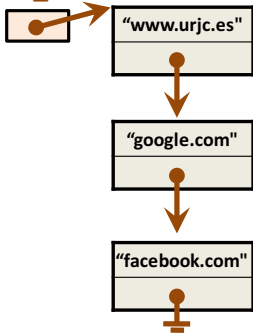
URL

?

Ejemplo

3 `URL := Stacks.Pop (My_Stack) ;`

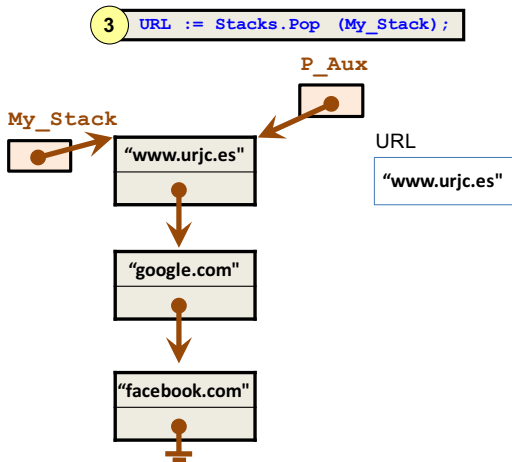
My_Stack



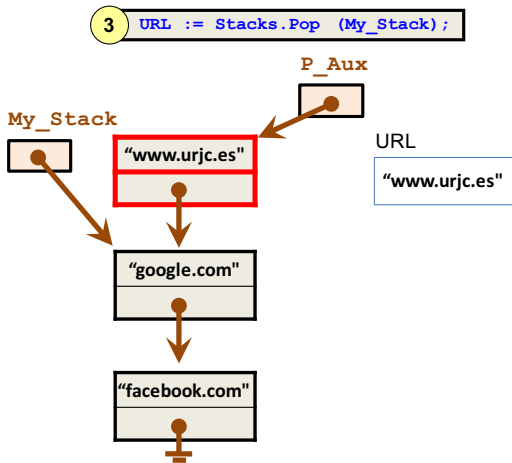
URL

"www.urjc.es"

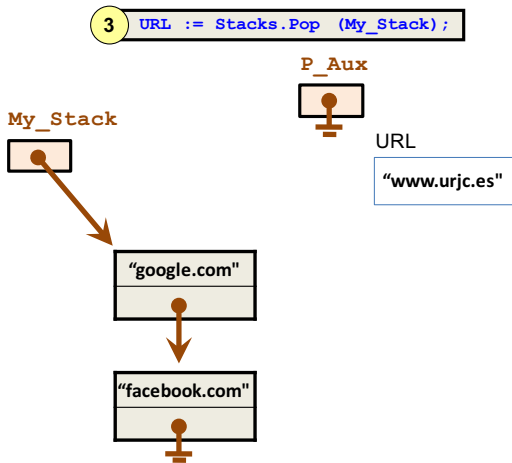
Ejemplo



Ejemplo



Ejemplo



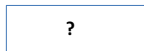
Ejemplo

4 `URL := Stacks.Pop (My_Stack) ;`

My_Stack



URL



Ejemplo

4 `URL := Stacks.Pop (My_Stack) ;`

My_Stack

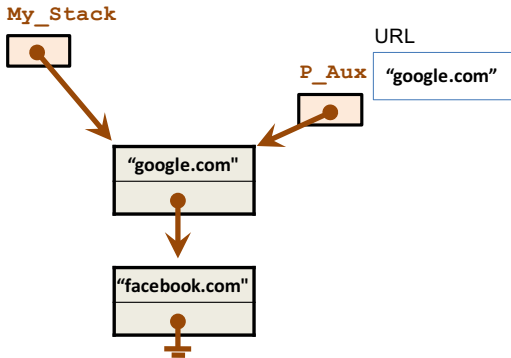


URL

"google.com"

Ejemplo

4 `URL := Stacks.Pop (My_Stack);`



Ejemplo

4

```
URL := Stacks.Pop (My_Stack);
```

My_Stack



URL

"google.com"

P_Aux



Ejemplo

4

```
URL := Stacks.Pop (My_Stack);
```

My_Stack



URL

P_Aux



"google.com"

"facebook.com"



Cola (*Queue*)

Una **cola** es una colección en la que se pueden insertar y extraer elementos.

2 operaciones básicas: Enqueue, Dequeue

- **Enqueue** inserta un nuevo elemento en la cola
- **Dequeue** extrae el elemento más antiguo de la cola
 - La extracción sigue el orden **FIFO** (*First-In First-Out*)

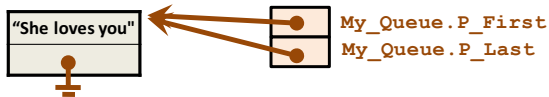
Casos de uso de las colas

- Para almacenar peticiones de uso de un servicio (uso de la impresora, uso del disco, uso de la CPU).
- Para transferir información asíncronamente entre dos entidades que no procesan información a la misma velocidad: cola de transmisión de paquetes en la tarjeta Ethernet en la que la CPU va insertando tramas para que la tarjeta las envíe cuando pueda.
- Lista de reproducción de una aplicación multimedia: cola de canciones que queremos que se vayan reproduciendo.

Implementación de la cola

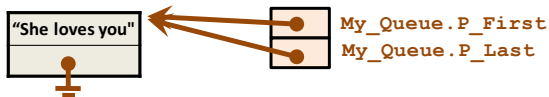
- Implementación con Array
 - Problema: el Array tiene un tamaño máximo fijado de antemano.
 - Solución: cuando el Array se llena se copia a otro un poco más grande.
- Implementación con Lista enlazada
 - De manera muy similar a como implementamos la tabla de símbolos con una lista enlazada:
 - No hay clave
 - **Enqueue**: Añade el nuevo elemento al final y no al principio, manteniendo para ello un nuevo puntero Last, que es actualizado después de enlazar el antiguo Last al nuevo elemento insertado
 - **Dequeue**: Devuelve First y actualiza First a First.next

Ejemplo



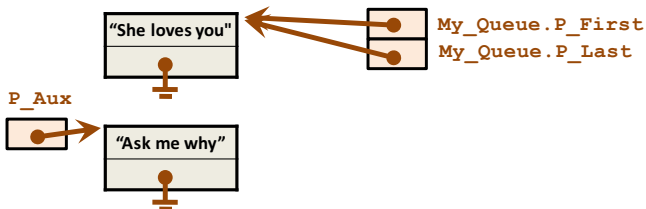
Ejemplo

1 `Queues.Enqueue(My_Queue, ASU.To_Unbounded_String ("Ask me why"));`



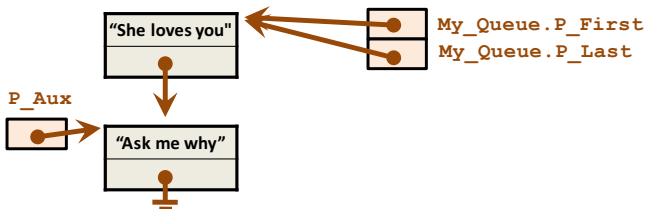
Ejemplo

```
1 Queues.Enqueue(My_Queue, ASU.To_Unbounded_String ("Ask me why"));
```



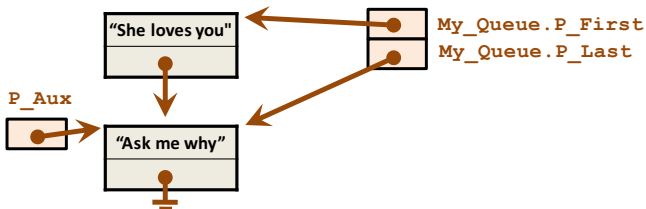
Ejemplo

```
1 Queues.Enqueue(My_Queue, ASU.To_Unbounded_String ("Ask me why"));
```



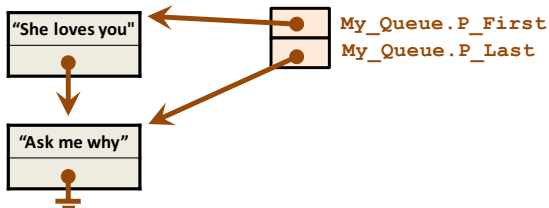
Ejemplo

```
1 Queues.Enqueue(My_Queue, ASU.To_Unbounded_String ("Ask me why"));
```



Ejemplo

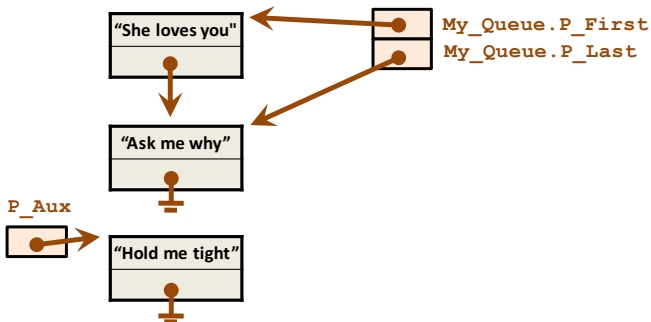
```
2 Queues.Enqueue(My_Queue, ASU.To_Unbounded_String ("Hold me tight"));
```



Ejemplo

2

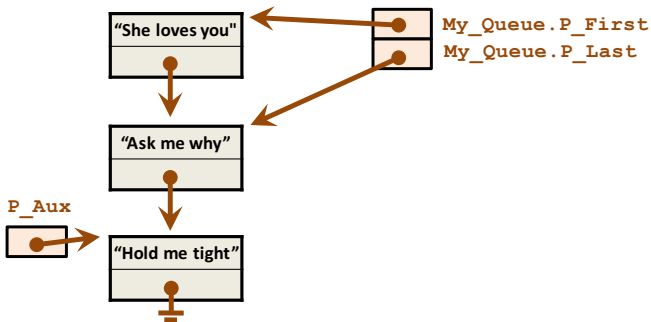
```
Queues.Enqueue(My_Queue, ASU.To_Unbounded_String ("Hold me tight"));
```



Ejemplo

2

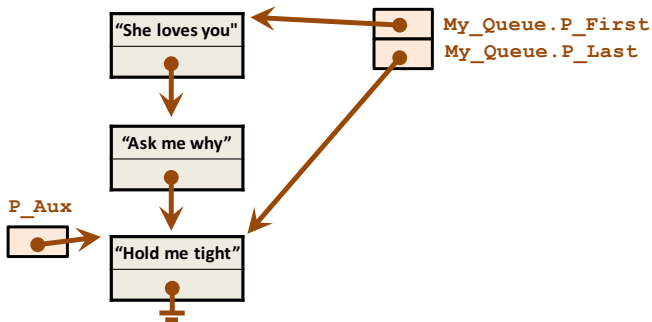
```
Queues.Enqueue(My_Queue, ASU.To_Unbounded_String ("Hold me tight"));
```



Ejemplo

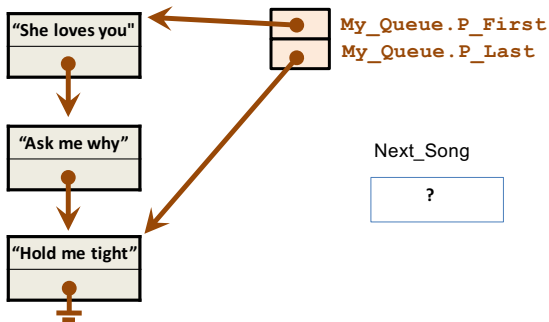
2

```
Queues.Enqueue(My_Queue, ASU.To_Unbounded_String ("Hold me tight"));
```



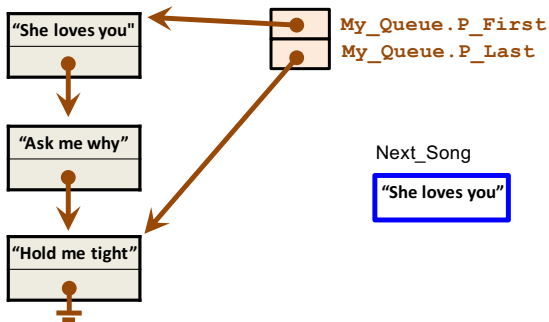
Ejemplo

3 `Next_Song := Queues.Dequeue (My_Queue) ;`



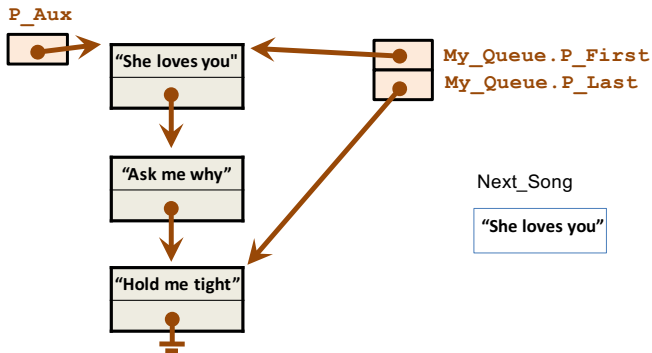
Ejemplo

3 `Next_Song := Queues.Dequeue (My_Queue);`



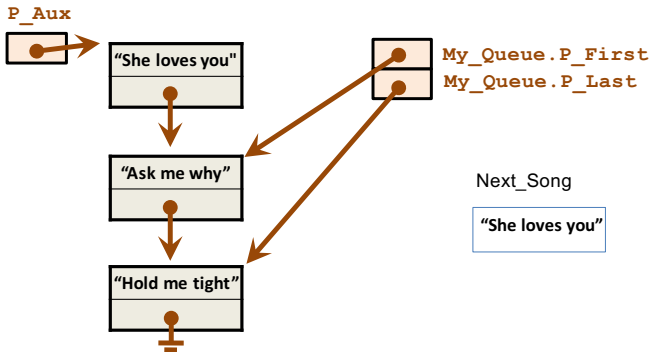
Ejemplo

```
3 Next_Song := Queues.Dequeue (My_Queue);
```



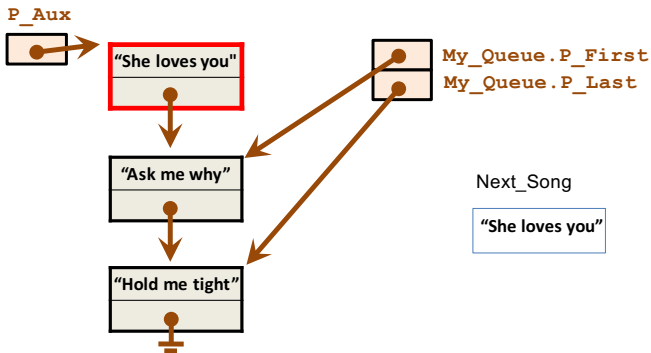
Ejemplo

3 `Next_Song := Queues.Dequeue (My_Queue);`



Ejemplo

3 `Next_Song := Queues.Dequeue (My_Queue);`



Ejemplo

3 `Next_Song := Queues.Dequeue (My_Queue);`

P_Aux



My_Queue.P_First

My_Queue.P_Last

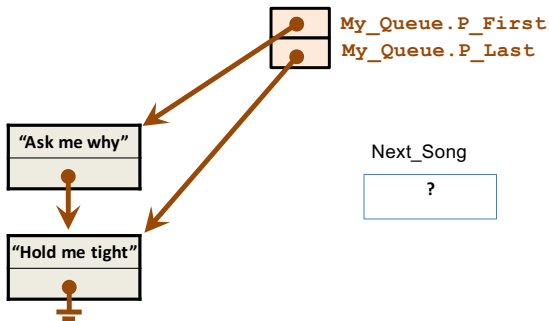


Next_Song

"She loves you"

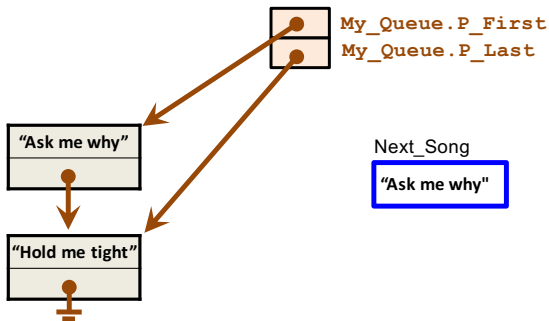
Ejemplo

4 `Next_Song := Queues.Dequeue (My_Queue) ;`



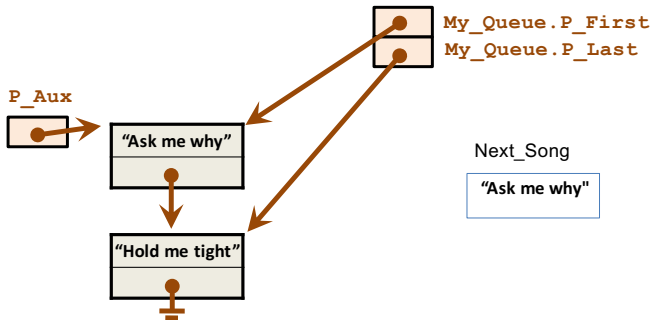
Ejemplo

4 `Next_Song := Queues.Dequeue (My_Queue) ;`



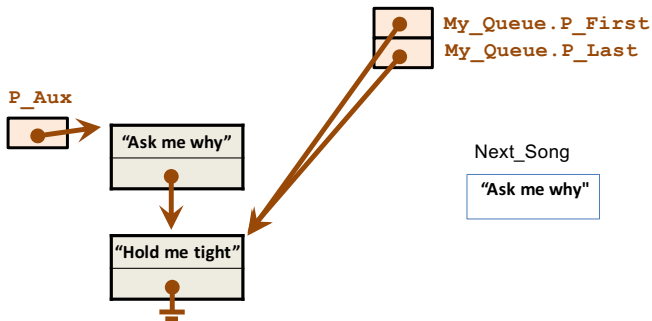
Ejemplo

4 `Next_Song := Queues.Dequeue (My_Queue);`



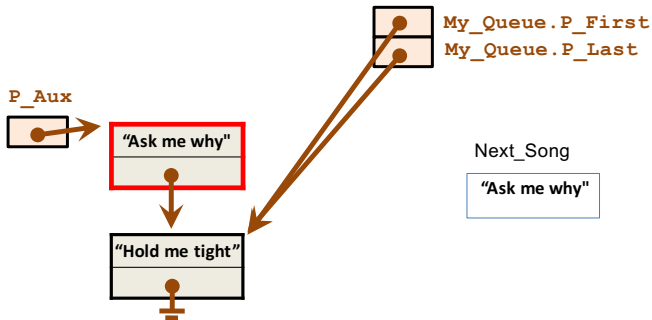
Ejemplo

4 `Next_Song := Queues.Dequeue (My_Queue);`



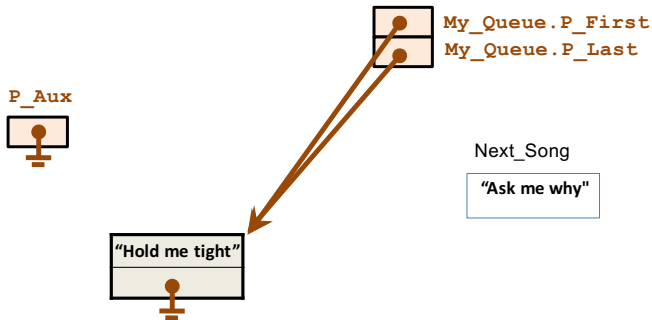
Ejemplo

4 `Next_Song := Queues.Dequeue (My_Queue);`



Ejemplo

4 `Next_Song := Queues.Dequeue (My_Queue);`



Contenidos

- 1 Árboles autoequilibrados
- 2 Pilas y Colas
- 3 Tabla Hash**
- 4 Listas doblemente enlazadas, Listas enlazadas circulares
- 5 Biblioteca predefinida de contenedores de Ada 2005
- 6 Bibliografía

Tablas hash

- Una tabla de símbolos en la que las claves sean números enteros es fácilmente implementable con un array:
 - Se inserta en el índice del array que indica la clave
 - Se extrae del índice del array que indica la clave
- ¡El tiempo de acceso sería $O(1)$ tanto para las inserciones como para las extracciones!

La tabla hash es una generalización de esta implementación mediante un Array para claves más complejas: **mediante operaciones aritméticas realizadas sobre la clave se obtiene un índice del array**

Tablas hash

En una tabla hash se necesita un mecanismo para convertir claves en índices del Array (la función hash) y un mecanismo para evitar colisiones cuando la función hash genera el mismo índice para claves distintas:

- **Función hash:** convierte cualquier clave (strings, claves compuestas,...) en un índice del Array.
 - La función hash debería ser poco costosa computacionalmente y distribuir uniformemente las claves entre los índices del Array.
- **Resolución de colisiones:**
 - Si la cantidad disponible de índices en el Array es menor que el número de claves distintas, puede haber colisiones: ¡la función hash devuelve el mismo índice para dos claves distintas!

Funciones hash: ejemplos

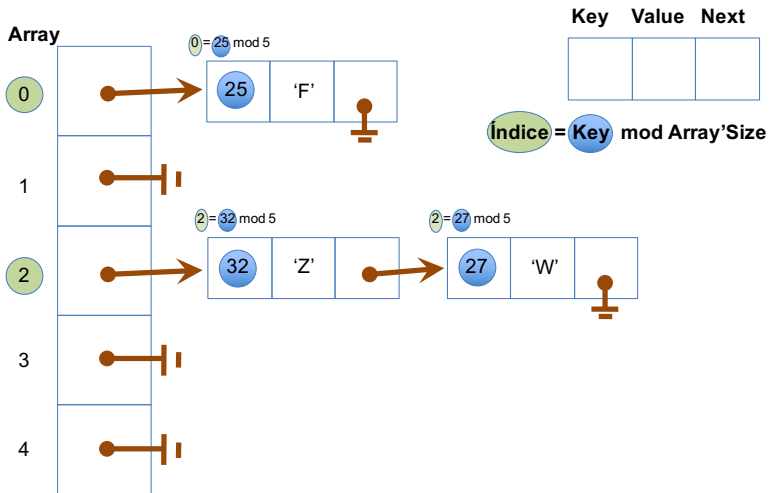
- Claves que son números:
 - Función hash: resto de dividir la clave entre el tamaño del Array
 - Escoger un número primo como tamaño del Array puede ayudar en la dispersión.
- Claves que son números reales:
 - Función hash: se puede hacer lo mismo pero utilizando la representación binaria del número obtenida concatenando la mantisa con el exponente
- Claves que son Strings:
 - Función hash: se obtiene un número a partir de la representación binaria de cada carácter del String, y se obtiene el resto de dividirlo entre el tamaño del Array

Resolución de colisiones mediante encadenamiento

Resolución de colisiones mediante encadenamiento

- En cada posición del Array se almacenan en una lista enlazada las parejas (clave, valor) de las colisiones
- Para encontrar un elemento:
 - 1 Se aplica la función hash a la clave.
 - 2 Se busca linealmente en la lista enlazada de la posición del Array correspondiente al índice devuelto por la función hash

Ejemplo



Resolución de colisiones mediante direccionamiento abierto

Resolución de colisiones mediante direccionamiento abierto

- Si se quiere almacenar un máximo de N elementos, se utiliza una tabla de $M > N$ posiciones que almacena parejas (clave, valor)
- Para encontrar un elemento:
 - 1 Se aplica la función hash a la clave
 - 2 Si en la posición devuelta está la clave: un Get devuelve el valor y un Put actualiza el valor
 - 3 Si en la posición devuelta no hay almacenado ningún elemento, no está la clave: Get no obtiene el valor, Put puede insertar ahí el valor
 - 4 Si en la posición devuelta hay almacenado un elemento con una clave distinta, se va probando con la siguiente posición en el Array hasta encontrar el elemento o encontrar un hueco

Borrado para direccionamiento abierto (alternativa I)

- Cuando borramos no podemos dejar un hueco, pues podría haber después elementos que no están en donde les corresponde, y que no se podrían encontrar al dejar un hueco por delante de ellos
- Hay que recorrer todos los elementos que hay detrás del borrado y antes del siguiente hueco, y comprobar si se pueden mover al hueco

Borrado para direccionamiento abierto (alternativa II)

- Cuando borramos lo hacemos **perezosamente**
- Se deja una marca que indica que en esa posición hubo un elemento
- Si hay demasiadas marcas de borrado se pueden recalcular las posiciones de todos los elementos, para eliminar las marcas de borrado perezoso.

Borrado para direccionamiento abierto (alternativa III)

- Cuando borramos lo hacemos **perezosamente**
- Se deja una marca que indica que en esa posición hubo un elemento
- Cuando se está buscando un elemento y se consulta una posición en la que hay una marca de borrado, se considera igual que si hubiera un elemento distinto del que se está buscando: se sigue buscando en el siguiente
 - Tras múltiples inserciones y borrados, las marcas de borrado que van quedando hacen que cada vez haya elementos más lejos de donde les corresponde por su tabla hash
 - Por ello, cuando se busca un elemento, si se encuentra, se borra de donde está (dejando la marca de borrado) y se mueve a la primera posición con una marca de borrado que se ha consultado mientras que se buscaba el elemento
- Cuando se está insertando un elemento: primero se busca (marca de borrado se considera igual que si hubiera un elemento distinto del que se busca), y si no se encuentra, se puede insertar en la primera posición con marca de borrado consultada mientras se buscaba.

Ejemplo

Array**Key****Value****M=8, N=5**

0

25

'F'

1

2

32

'Z'

3

4

49

'H'

5

6

7

Put (30, 'K')

Ejemplo

Array

Key

Value

M=8, N=5

0

25

'F'

1

2

32

'Z'

3

4

49

'H'

5

6

7

30 != 25

Put (30, 'K')

30 mod 5 = 0

Ejemplo

Array

Key

Value

M=8, N=5

0	25	'F'
1	30	'K'
2	32	'Z'
3		
4	49	'H'
5		
6		
7		

Put (30, 'K')

 $30 \bmod 5 = 0$

Ejemplo

Array**Key****Value****M=8, N=5**

0

25

'F'

1

30

'K'

2

32

'Z'

3

4

49

'H'

5

6

7

Put (15, 'A')

Ejemplo

Array

Key

Value

M=8, N=5

0

25

'F'

1

30

'K'

2

32

'Z'

3

4

49

'H'

5

6

7

15 != 25

Put (15, 'A')

 $15 \bmod 5 = 0$

Ejemplo

Array

Key

Value

M=8, N=5

0

25

'F'

1

30

'K'

15 /= 30

2

32

'Z'

3

4

49

'H'

5

6

7

Put (15, 'A')**15 mod 5 = 0**

Ejemplo

Array

Key

Value

M=8, N=5

0

25

'F'

1

30

'K'

2

32

'Z'

15 /= 32

3

4

49

'H'

5

6

7

Put (15, 'A')**15 mod 5 = 0**

Ejemplo

Array

Key

Value

M=8, N=5

0

25

'F'

1

30

'K'

2

32

'Z'

3

15**'A'**

4

49

'H'

5

6

7

Put (15, 'A') **$15 \bmod 5 = 0$**

Ejemplo

Array	Key	Value
0	25	'F'
1	30	'K'
2	32	'Z'
3	15	'A'
4	49	'H'
5		
6		
7		

M=8, N=5

Success

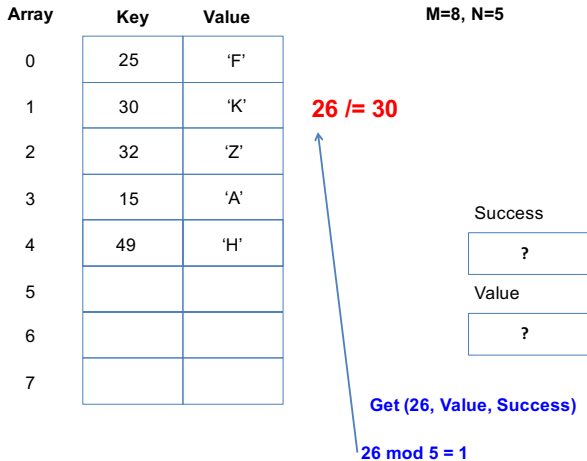
?

Value

?

Get (26, Value, Success)

Ejemplo



Ejemplo

Array

Key

Value

M=8, N=5

0

25

'F'

1

30

'K'

2

32

'Z'

26 != 32

3

15

'A'

4

49

'H'

Success

?

5

Value

6

?

7

Get (26, Value, Success)**26 mod 5 = 1**

Ejemplo

Array

Key

Value

M=8, N=5

0

25

'F'

1

30

'K'

2

32

'Z'

3

15

'A'

4

49

'H'

5

6

7

26 /= 15

Success

?

Value

?

Get (26, Value, Success)**26 mod 5 = 1**

Ejemplo

Array

Key

Value

M=8, N=5

0

25

'F'

1

30

'K'

2

32

'Z'

3

15

'A'

4

49

'H'

5

6

7

26 != 49

Success

?

Value

?

Get (26, Value, Success)**26 mod 5 = 1**

Ejemplo

Array

Key

Value

M=8, N=5

0

25

'F'

1

30

'K'

2

32

'Z'

3

15

'A'

4

49

'H'

5

6

7

Success

False

Value

?

X

Get (26, Value, Success)

 $26 \bmod 5 = 1$

Ejemplo

Array**Key****Value****M=8, N=5**

0

25

'F'

1

30

'K'

2

32

'Z'

3

15

'A'

4

49

'H'

5

6

7

Delete (32)

Ejemplo

Array

Key

Value

M=8, N=5

0

25

'F'

1

30

'K'

2

32

'Z'

3

15

'A'

4

49

'H'

5

6

7

Delete (32)

 $32 \bmod 5 = 2$

Ejemplo

Array

Key

Value

0

25

'F'

1

30

'K'

2

3

15

'A'

4

49

'H'

5

6

7

M=8, N=5

¡No podemos dejar un hueco
en la posición 2, pues el 15
nunca lo encontraríamos!

Delete (32)

 $32 \bmod 5 = 2$

Ejemplo

Array

Key

Value

M=8, N=5

0

25

'F'

1

30

'K'

2

3

15

'A'

4

49

'H'

5

6

7

Alternativa I: Ahora habría que comprobar si hay que mover los elementos que hay antes del primer hueco (15 y 49): habría que mover el 15 a la posición 2, y dejar el hueco en la posición 3, pues el 49 está donde le corresponde.

Delete (32)

 $32 \bmod 5 = 2$

Ejemplo

Array

Key

Value

M=8, N=5

0

25

'F'

1

30

'K'

2

3

15

'A'

4

49

'H'

5

6

7

Alternativa II: borrado
perezoso, dejando una marca
de borrado (----)

Delete (32)

$32 \bmod 5 = 2$

Ejemplo

Array

Key

Value

M=8, N=5

0

25

'F'

1

30

'K'

2

3

15

'A'

4

49

'H'

5

6

7

Alternativa III: borrado perezoso, dejando una marca de borrado (----), pero ir moviendo a la marca de borrado elementos cuando se ejecuten Get y Put

Delete (32)

 $32 \bmod 5 = 2$

Ejemplo

Array	Key	Value
0	25	'F'
1	30	'K'
2	---	---
3	15	'A'
4	49	'H'
5		
6		
7		

M=8, N=5

Success

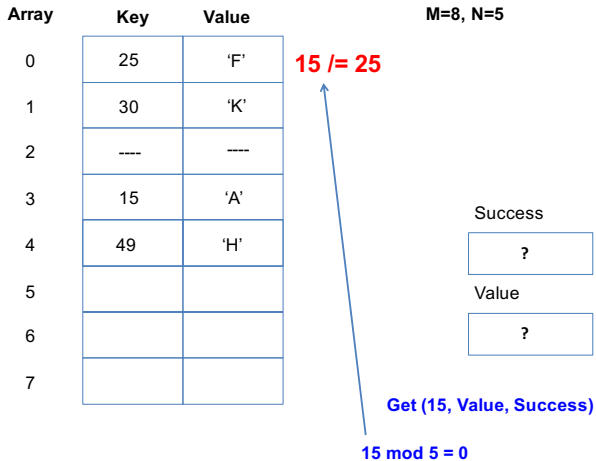
?

Value

?

Get (15, Value, Success)

Ejemplo



Ejemplo

Array

Key

Value

M=8, N=5

0

25

'F'

1

30

'K'

15 != 30

2

3

15

'A'

4

49

'H'

5

6

7

Success

?

Value

?

Get (15, Value, Success)**15 mod 5 = 0**

Ejemplo

Array	Key	Value
0	25	'F'
1	30	'K'
2	----	----
3	15	'A'
4	49	'H'
5		
6		
7		

M=8, N=5

¡Cuando estamos buscando, la marca dejada por un delete perezoso no se considera como un hueco, por lo que seguimos buscando!

15 != ----

Success

?

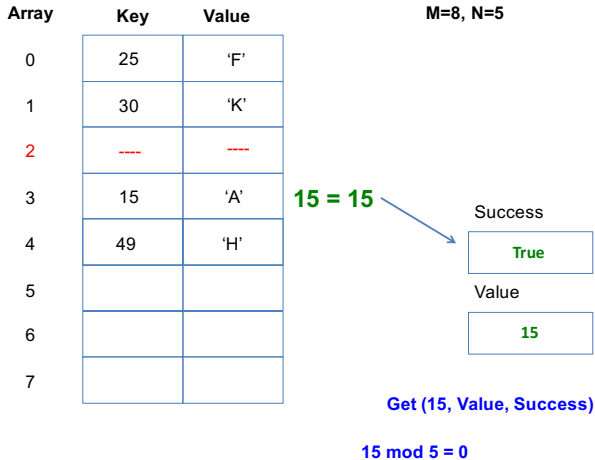
Value

?

Get (15, Value, Success)

$15 \bmod 5 = 0$

Ejemplo



Ejemplo

Array

Key

Value

M=8, N=5

0

25

'F'

1

30

'K'

2

3

15

'A'

4

49


'H'

5

6

7

Tras encontrar un elemento lo recolocamos en el primer hueco generado por un delete que hemos visitado en la cadena de búsqueda



Success

True

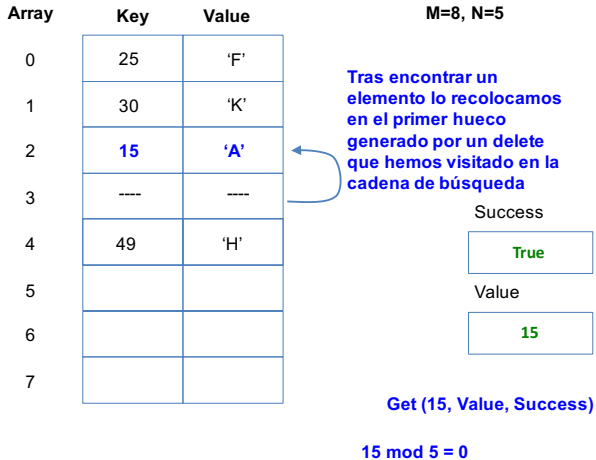
Value

15

Get (15, Value, Success)

$$15 \bmod 5 = 0$$

Ejemplo



Ejemplo

Array

Key

Value

0

25

'F'

1

30

'K'

2

15

'A'

3

4

49

'H'

5

6

7

M=8, N=5

Success

True

Value

15

Get (15, Value, Success)

 $15 \bmod 5 = 0$

Ejemplo

Array**Key****Value****M=8, N=5**

0

25

'F'

1

30

'K'

2

15

'A'

3

4

49

'H'

5

6

7

Put (11, 'S')

Ejemplo

Array

Key

Value

M=8, N=5

0

25

'F'

1

30

'K'

11 != 30

2

15

'A'

3

4

49

'H'

5

6

7

Put (11, 'S')

11 mod 5 = 1

Ejemplo

Array

Key

Value

M=8, N=5

0

25

'F'

1

30

'K'

2

15

'A'

11 /= 15

3

4

49

'H'

5

6

7

Put (11, 'S')**11 mod 5 = 1**

Ejemplo

Array

Key

Value

M=8, N=5

0

25

'F'

1

30

'K'

2

15

'A'

3

4

49

'H'

5

6

7

11 /= ----

Para insertar hay que buscar primero, por lo que la marca dejada por un delete se considera como ocupada, y seguimos buscando ...

Put (11, 'S')

11 mod 5 = 1

Ejemplo

Array

Key

Value

M=8, N=5

0

25

'F'

1

30

'K'

2

15

'A'

3

4

49

'H'

11 /= 49

5

6

7

Put (11, 'S')

11 mod 5 = 1

Ejemplo

Array	Key	Value
0	25	'F'
1	30	'K'
2	15	'A'
3	---	---
4	49	'H'
5		
6		
7		

M=8, N=5

No está, por lo que lo insertamos.
Podríamos insertarlo en la posición 5 pero
...

X

Put (11, 'S')

$11 \bmod 5 = 1$

Ejemplo

Array

Key

Value

M=8, N=5

0

25

'F'

1

30

'K'

2

15

'A'

3

4

49

'H'

5

6

7

No está, por lo que lo insertamos.

Podríamos insertarlo en la posición 5 pero

...

podemos insertar en el primer hueco generado por un delete que hemos visitado en la cadena de búsqueda

Put (11, 'S')

$11 \bmod 5 = 1$

Ejemplo

Array	Key	Value
0	25	'F'
1	30	'K'
2	15	'A'
3	11	'S'
4	49	'H'
5		
6		
7		

M=8, N=5

No está, por lo que lo insertamos.

Podríamos insertarlo en la posición 5 pero

...

Podemos insertar en el primer hueco generado por un delete que hemos visitado en la cadena de búsqueda

Put (11, 'S')

$11 \bmod 5 = 1$

Contenidos

- 1 Árboles autoequilibrados
- 2 Pilas y Colas
- 3 Tabla Hash
- 4 Listas doblemente enlazadas, Listas enlazadas circulares**
- 5 Biblioteca predefinida de contenedores de Ada 2005
- 6 Bibliografía

Introducción

Las listas doblemente enlazadas y las listas enlazadas circulares son implementaciones alternativas de las listas enlazadas

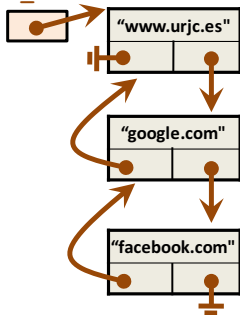
Al igual que las listas enlazadas se utilizan en la implementación de otras estructuras de datos como tablas de símbolos, pilas, colas, tablas hash

Listas doblemente enlazadas

- Una lista doblemente enlazada es una lista enlazada en la que cada nodo tiene, además de un enlace al nodo siguiente (Next), un enlace al nodo anterior que le precede en la lista: [Prev](#).
- Puede facilitar la implementación de las operaciones de tablas de símbolos, pilas, colas o tablas hash.
- Por ejemplo: para borrar un elemento no necesitamos mantener el puntero al que le precede.

Ejemplo

P_First

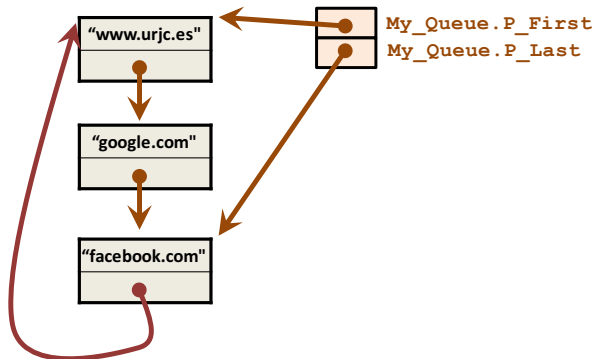


Prev	Next

Listas enlazadas circulares

- En una lista enlazada circular: $\text{Last.Next} = \text{First}$
- En una lista doblemente enlazada circular: $\text{Last.Next} = \text{First}$ y $\text{First.Prev} = \text{Last}$
- Permiten recorrer todos los elementos de la lista empezando en cualquiera de los elementos

Ejemplo de lista enlazada circular



Contenidos

- 1 Árboles autoequilibrados
- 2 Pilas y Colas
- 3 Tabla Hash
- 4 Listas doblemente enlazadas, Listas enlazadas circulares
- 5 Biblioteca predefinida de contenedores de Ada 2005**
- 6 Bibliografía

Ada.Containers

Biblioteca predefinida de contenedores de Ada 2005

En Ada 2005 se introdujo en la biblioteca predefinida del lenguaje (Ada.Containers), que proporciona implementaciones de:

- Listas doblemente enlazadas
- Arrays con búsqueda binaria
- Tablas de símbolos (*Maps* en Ada.Containers)
- Conjuntos (colecciones de elementos no repetidos no ordenados)
- Algoritmos de búsqueda

GNAT implementa algunas de estas estructuras de datos usando árboles binarios autobalanceados de tipo rojo-negro.

En otros lenguajes de programación como Java, C++, Smalltalk,... existen bibliotecas de contenedores similares.

Contenidos

- 1 Árboles autoequilibrados
- 2 Pilas y Colas
- 3 Tabla Hash
- 4 Listas doblemente enlazadas, Listas enlazadas circulares
- 5 Biblioteca predefinida de contenedores de Ada 2005
- 6 Bibliografía

- *Software Construction and Data Structures With Ada 95*. Michael B. Feldman. Addison Wesley 1996.
- *Programming in Ada 2005*. John Barnes. Addison Wesley 2006.
- *Algorithms. 4th edition*. Robert Sedgewick, Kevin Wayne. Addison Wesley 2011.
 - <http://www.cs.princeton.edu/algs4>