Listas Enlazadas

Estructuras de Datos

1

Recuerdo Arreglos

- **Ventajas**
  - **Acceso es rápido – O(1)**
  - **Inserción es rápida en un arreglo desordenado O(1)**
  - **Búsqueda es rápida en un arreglo ordenado – O(log n)**
    - **Porque usamos búsqueda binaria**
- **Desventajas**
  - **La eliminación es lenta – O(n)**
  - **La búsqueda es lenta en un arreglo desordenado – O(n)**
  - **Inserción es lenta en un arreglo ordenado – O(n)**

2

Comparación Ordenamiento: Resumen

- Burbuja– apenas se usa
  - Muy lento, a menos que el número de datos sea muy pequeño
- Selección – un poco mejor
  - Útil si: numero de datos es pequeño y si la operación de intercambio consume mucho tiempo comparado con las comparaciones
- Inserción – más versátil
  - El mejor en la mayoría de las situaciones
  - Aún para grandes cantidades de datos altamente desordenados,
- Requerimientos de memoria no son tan altos para estos algoritmos

3

Una estructura de datos versátil

- **La lista enlazada**
  - **El segundo TDA (tipo de dato abstracto) más usado después de los arreglos**
- **Se utiliza para el almacenamiento de datos en el mundo real**
- **Se pueden usar vectores para las listas enlazadas!**
- **Es buena, para situaciones en las que la estructura se modifica con frecuencia**
- **Mala en situaciones con accesos frecuentes**

5

Diferentes Tipos

- **Simple**
- **De doble extremo**
- **Ordenada**
- **Doblemente enlazada**
- **Lista circular: simple o doblemente enlazada**

6

Un enlace

- Los datos de las listas enlazadas están incrustados en los vínculos o enlaces
- Cada enlace está compuesto por:
  - Los propios datos
  - Una referencia al siguiente enlace de la lista, que es nula para el último elemento

7

La clase Link (Enlace)

- **A veces tiene sentido tener un vínculo a su propia clase, ya que una lista puede definirse como una colección de objetos de enlace:**
  - **A esta clase se le llama una *clase auto-referencial*.**

```
class Link  {
  public int iData;
  public Link next;
}
```

8

Referencias

- Recuerde que en Java, todos los objetos son referencias
- Eso significa que la variable ‘next', para cada enlace sólo contiene un número entero que representa una dirección de memoria
  - Un 'número mágico' que nos dice dónde está el objeto
  - Son siempre del mismo tamaño (no hay problema)
- En un principio, cada vez que se crea un objeto de enlace, su referencia es nula
- Hasta que realmente declaramos:
  - mylink.next = new Link();

- Incluso ahora, “next” no contiene un objeto.
  - ¡Aún es una dirección!
  - Sólo que ahora es la dirección a un objeto real, a diferencia cuando es nulo

9

Memoria

- **¿Cómo se vería esto en la memoria, entonces?**

10

¡Recordemos la implicación!

- El acceso a una lista enlazada es lento en comparación con los arreglos
- Los arreglos son como hileras de casas
  - Ellos están dispuestos secuencialmente
  - Así que es fácil encontrar, por ejemplo, la tercera casa
- Con las listas enlazadas, usted tiene que seguir los eslabones de la cadena
  - Las referencias a "next"
  - ¿Cómo conseguimos el tercer elemento aquí?:

11

Enlaces de Registros

- **Podemos tener una relación de los expedientes del personal:**

```
class Link{
  public String name;
  public String address;
  public int ssn;
  public Link next;
}
```

12

Operaciones

- **Inserción**
  - **Al principio (rápido)**
  - **En el medio (más lento, aunque sigue siendo mejor que los arreglos)**
- **Eliminación**
  - **Al principio (rápido)**
  - **En el medio (más lento, aunque sigue siendo mejor que los arreglos)**
- **Búsqueda**
  - **De manera similar a los arreglos, en el peor de los casos tenemos que comprobar todos los elementos**
- **¡Vamos a construir algunos de estos métodos!**

13

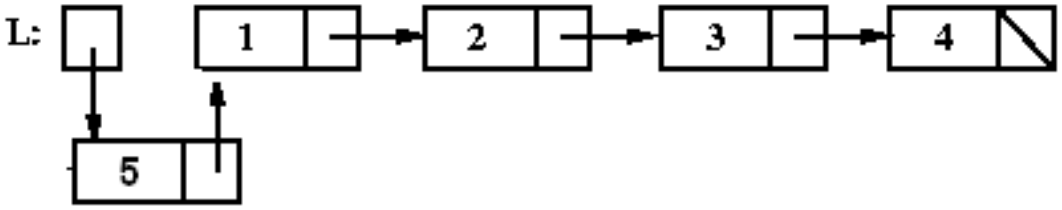
Clase LinkedList

- **Inicia con:**
  - **Un Link privado al primer elemento**
  - **Un constructor que establece esta referencia a null**
  - **Un método isEmpty(), que devuelve true si la lista está vacía**

14

insertFirst(): O(1)

- **Aceptar un nuevo número entero**
  - **Crear un nuevo enlace (Link)**
  - **Cambiar la referencia del nuevo enlace next al actual primero**
  - **Cambie el primero para referenciar al nuevo enlace**
  - **No podemos ejecutar estos dos últimos pasos a la inversa. ¿Por qué?**



15

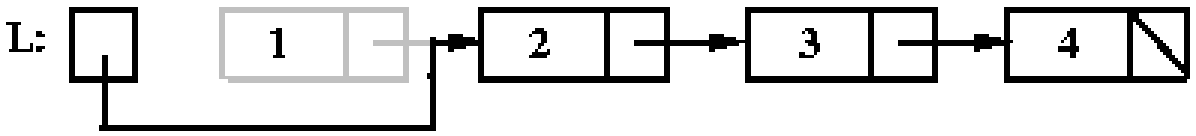
```
Algoritmo Insertar Nodo
{
        p = new Nodo
        info(p) = Elemento de datos a insertar
        sig(p) = inicio
        inicio = p
}
```

```
// operación insertar nodo
void Lista :: insertarNodo(int dato)
{
        Nodo *p;
        p = new Nodo;           // paso 1
        p -> info = dato;               // paso 2
        p -> sig = inicio;      // paso 3
        inicio = p;             // paso 4
}
```

16

deleteFirst(): O(1)

- **Retire el primer número entero en la lista**
  - **Sólo restablece la primera referencia a first.next**



17

Delete_nodo(): O(n)

El algoritmo básico es el siguiente:

```
EliminarNodo
{
        Buscar en la lista el elemento a eliminar.
        Ajusta los punteros de la lista para eliminar el nodo que
        contenga el elemento que se va a eliminar.
}
```

Como puede ver; antes de eliminar se debe de buscar.

La búsqueda de un elemento se realiza Nodo a Nodo, de principio a fin, es decir, es secuencial.
El algoritmo para buscar es el siguiente:

```
Buscar
{           if lista vacia
                    escribir("lista vacia...")
            else    // Ubicarse al comienzo de la lista
                    p = inicio
                    antp = NULL
                    encuentra = false       //encuentra es una bandera
                    while(NOT encuentra  AND  p  != NULL)
                    {           if info(p) == dato
                                        encuentra = true
                                else
                                {           //continuar busqueda
                                        antp = p
                                        p = sig(p)
                                }
                    }
}
```

19

07/06/2020

Después de utilizar el algoritmo de la búsqueda, se usan los valores de encuentra p y antp para proceder a eliminar el nodo requerido.
El algoritmo es el siguiente:

```
Eliminar
{       if(encuentra)
        {
                if(antp == NULL) //caso del 1er nodo
                        inicio = sig(p)
                else
                        sig(antp) = sig(p) //eliminacion
        }
        else
                escribir("No se encuentra en la lista...")
}
```

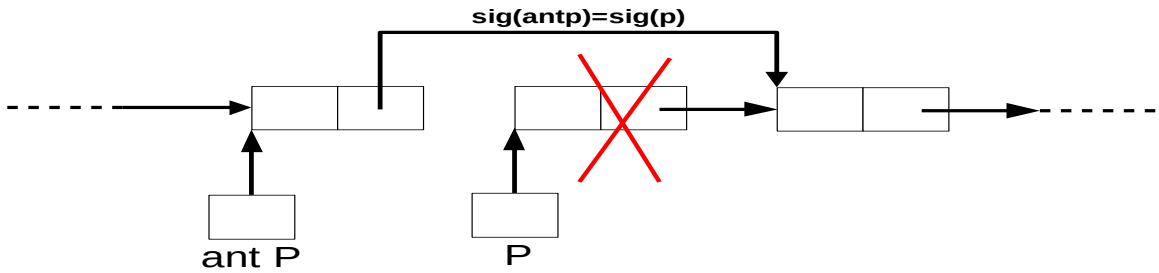
Ahora se puede ajustar la búsqueda con eliminar para escribir un solo algoritmo: Eliminar Nodo.

20 07/06/2020

El algoritmo básico es el siguiente:

```
EliminarNodo
{
        Buscar en la lista el elemento a eliminar.
        Ajusta los punteros de la lista para eliminar el nodo que
        contenga el elemento que se va a eliminar.
}
```

Como puede verse, antes de eliminar se debe de buscar.



21

07/06/2020

El algoritmo básico es el siguiente:

```
Algoritmo EliminarNodo
{
        Algoritmo Buscar
        Algoritmo Eliminar
}
```

El código:

```
// Operación EliminarNodo()
void Lista :: eliminarNodo(int dato)
{
        int encuentra = false;
        Nodo *p, *antP;
        p = inicio;
        antP = NULL;
```

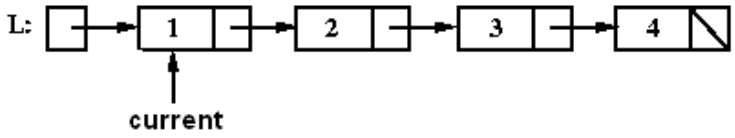
22

07/06/2020

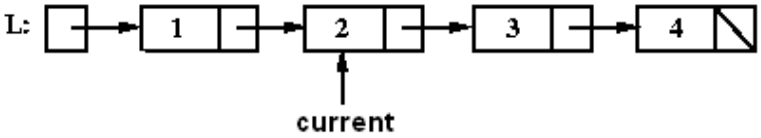
```
        // buscar elemento a eliminar
        if( listaVacia() )
                cout<<"Lista vacia!…"<<endl;
        else
        {       while(!encuentra && p!=NULL)
                {       if (p -> info == dato)
                                encuentra = true;
                        else
                        {       antP = p;
                                p = p-> sig;
                        }
                }
                //eliminar nodo si se encuentra
                if(encuentra)
                {       if(antP == NULL)
                        {       inicio = p-> sig;
                                delete p;
                        }
                        else
                        {       antP -> sig = p -> sig;
                                delete p;
                        }
                }
                else
                        cout<<"El dato"<<dato<<"no se encuentra"<<endl;
        } // fin de 1er if
}
```

displayList() – O(n)

- **Use una referencia a *current* para iterar a través de los elementos**
  - **Imprimir el valor**
  - **Asigne a *current* la referencia de *current.next***
- **Pare cuando *current* sea nulo (null)**
- **Antes de asignar *current* a *current.next*:**



- **Después de asignar *current* a *current.next*:**



24

Recorrer Lista

El recorrido de la lista es secuencial y se realiza de prinicipio a fin. El objetivo del recorrido es visitar cada nodo y enviar su(s) dato(s) hacia el flujo de salida. El seudocódigo es:

```
RecorrerLista
{           if(lista No vacia)
            {           p = inicio
                        while( p !== NULL)
                        {           mostrar info(p)
                                    p= sig(p) //ir al siguiente nodo
                        }
            }
            else
                        escribir (“Lista vacia”)
}
```

25

07/06/2020

El código

```
// Operación recorrer lista()
void Lista :: recorrerLista()
{       Nodo *p;
        p = inicio;
        if(!listaVacia())
        {       while(p!=NULL)
                {       cout<<p -> info<<"-> ";
                        p = p -> sig;
                }
                cout<<"NULO"<<endl;
        }
        else
                cout<<"lista Vacia…!"<<endl;
}
```

Lista Vacía

Consiste en averiguar si la lista tiene o no elementos. Para ello se devuelve un valor booleano.

```
Lista Vacia
{       if (inicio = NULL)
                return TRUE
         else
                return FALSE
}
```

```
// código operación listaVacia()
int lista :: listaVacia()
{     if (inicio == NULL)
            return true;
       else
            return false;
}
```

27

07/06/2020

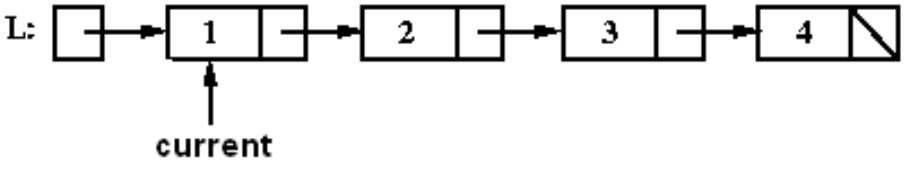
Función main()

```
LinkedList theList = new LinkedList();
theList.insertFirst(22);
theList.insertFirst(44);
theList.insertFirst(66);
theList.insertFirst(88);

theList.displayList();

while (!theList.isEmpty())
  theList.deleteFirst();

theList.displayList();
```

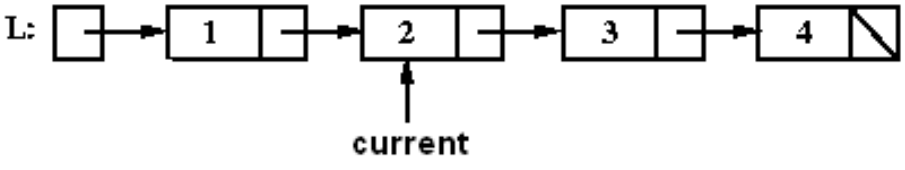
28

find() - O(n)

- Esencialmente es la misma idea que displayList()
  - Linealmente iterar a través de los elementos con una referencia a current
  - Repetidamente asignar current a current.next
  - Excepto: ¡para cuando lo hayas encontrado!
- Antes de asignar *current* a *current.next*:



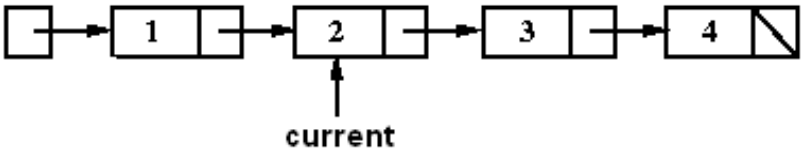
- Después de asignar *current* a *current.next*:



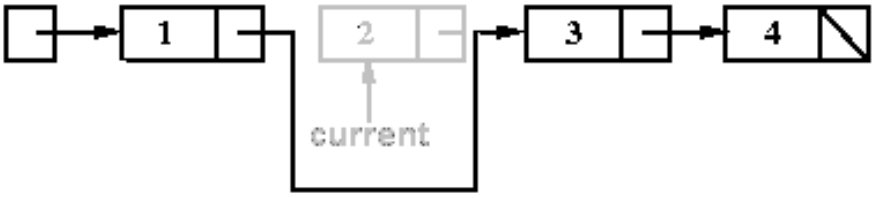
29

delete() –O(n)

- **Busque un valor almacenado en la lista, y retírelo**
- **Primero tenemos que encontrarlo, y en ese momento estará en current**
  - **Deber mantener una referencia al elemento previo a current.next**
- **C L:**



- **L:**



30

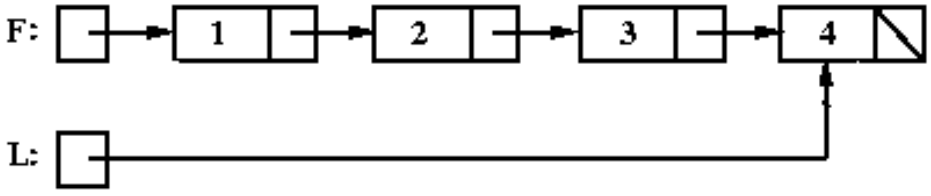
main() función - #2

```
LinkedList theList = new LinkedList();
theList.insertFirst(22);
theList.insertFirst(44);
theList.insertFirst(66);
theList.insertFirst(88);

theList.displayList();

Link f = theList.find(44);

theList.delete(66);

theList.displayList();
```

31

Listas de dos extremos

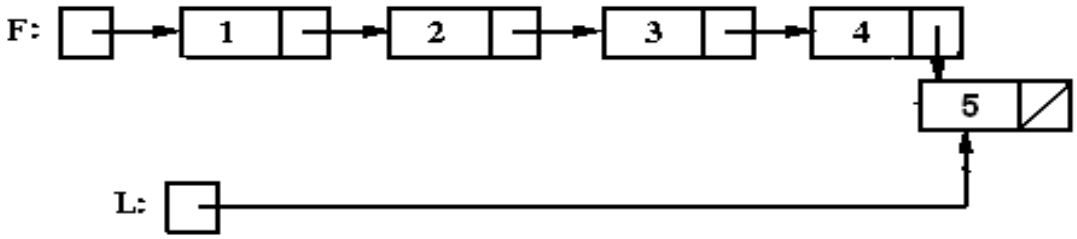
- Es igual que una lista enlazada regular, excepto que ahora hay que mantener referencias
  - Una al inicio (primera)
  - Y una hasta el final (último)
- Permite una fácil inserción en ambos extremos
  - Aun no se puede eliminar el último elemento fácilmente. ¿Por qué?
  - No se puede cambiar find () para iniciar desde el final.



32

insertLast() - O(1)

- ¿Qué se parece esto ahora? Vamos a ver:
  - Cree el nuevo enlace con el nuevo valor (next = null)
  - Establecer ultimo.siguiente para hacer referencia al nuevo enlace
  - Selecciona el último para hacer referencia al nuevo enlace



33

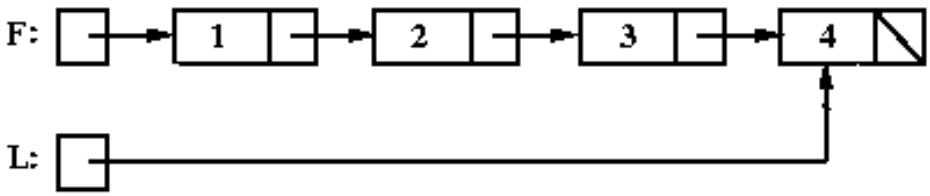
main() funcion - #3

```
LinkedList theList = new LinkedList();
theList.insertFirst(22);
theList.insertFirst(44);
theList.insertFirst(66);

theList.insertLast(11);
theList.insertLast(33);
theList.insertLast(55);

theList.displayList();

/ / Insertar elementos al principio invierte el orden
/ / Insertar elementos al final, ¡conserva su orden!
```

34

Listas con dos extremos

- **¿Tendremos también que modificar delete ()?**
- **¿Cuándo? Cómo hacerlo?.**
- **Tarea: especificar una manera de realizar la eliminación del inicio y del fin de la lista de dos extremos, e implementar la solución**



35

Eficiencia: Listas enlazadas con dos extremos

- **Inserción rápida en los extremos: O (1)**
- **Búsqueda: O (n)**
- **Eliminación de un elemento específico: O (n)**
  - **PERO, más rápido que los arreglos**
  - **La búsqueda requiere igualmente O (n)**
  - **Pero en un arreglo necesitamos O (n) desplazamientos (shifts), y para una lista necesitamos sólo copias de referencia - O (1)**
- **La inserción en un punto específico se puede hacer, con resultados similares**

36

Memoria: Resumen

- **Una lista enlazada usa (más o menos) memoria que un arreglo? ¿Por qué?**

- **Una lista enlazada utiliza la memoria de forma más eficiente que un arreglo.**
  - **El tamaño es flexible**

37

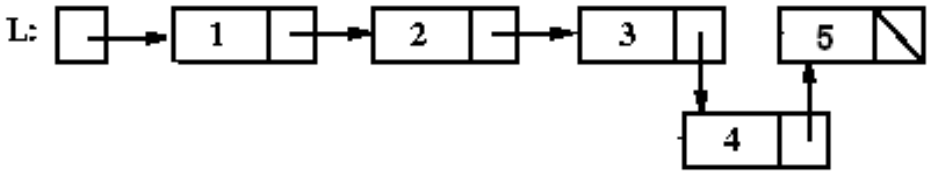
Listas Ordenadas

- Es una lista enlazada donde los datos se mantienen en forma ordenada
- Útil en algunas aplicaciones
  - Las mismas aplicaciones en las que utilizarías un arreglo ordenado
  - Pero, la inserción será más rápida!
  - Y, la memoria será utilizada de manera más eficiente
  - Pero, es un poco más difícil de poner en práctica
- Veámosla …

38

insert() - O(n)

- No hemos mirado a la inserción en el medio. Vamos a ver cómo se va a hacer:



- Así que para cada elemento, lo insertarmos en el lugar correcto
  - O (n), en el peor caso, para encontrar el punto de inserción
  - O (1) para la inserción real
    - Versus O (n) para los arreglos con desplazamiento
- ¿Podrían otras operaciones cambiar (delete, find)?

39

main() función - #4

```
LinkedList theList = new LinkedList();
theList.insert(20);
theList.insert(40);

theList.displayList();

theList.insert(10);
theList.insert(30);
theList.insert(50);

theList.displayList();

theList.remove();

theList.displayList();
```

40

Lista Enlazada Ordenada: Eficiencia

- **Inserción y eliminación son O (n) para el peor caso de búsqueda**
  - **No se puede hacer una búsqueda binaria en una lista enlazada ordenada, al contrario como lo vimos con arreglos! ¿Por qué no?**
- **El valor mínimo se encuentra en tiempo O (1)**
  - **Si la lista es de dos extremos, el máximo también es posible (por qué?)**
- **Por lo tanto, es buena si una aplicación requiere acceso frecuente al ítem mínimo (o máximo)**
  - **¿Para qué tipo de cola nos sería esto de ayuda?**
- **También es buena para un algoritmo de ordenamiento!**
  - **n inserciones, cada una requiere de O (n) comparaciones lo que da O (n^2)**
  - **Sin embargo, O (n) copias en vez de O (n^2) con la ordenación por inserción**
  - **Pero el doble de memoria (¿por qué?)**

41