



UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

Clase Lista C++ Estándar

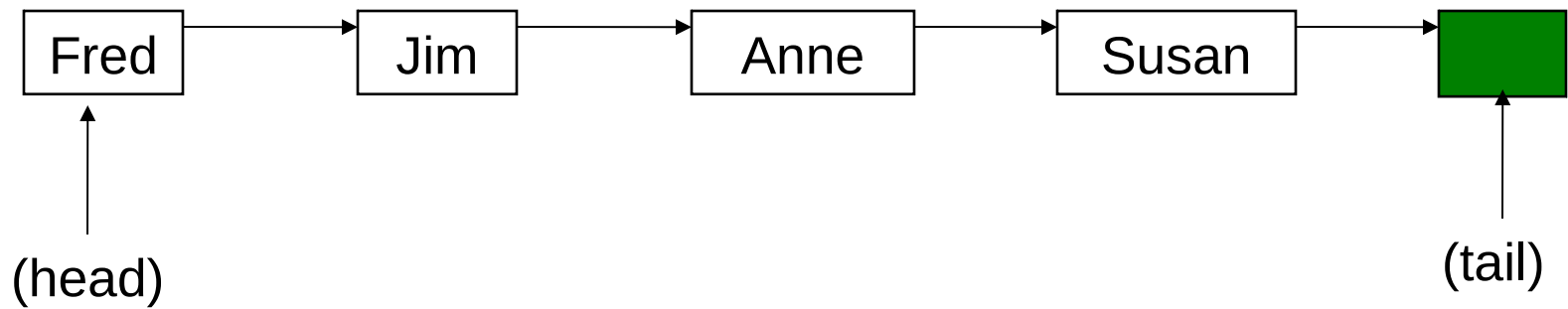
Agustín J. González
EIO329

Estructura de Datos Lista (List)

- La estructura de datos Lista es una secuencia conectada de nodes, cada uno de los cuales contiene algún dato.
- Hay un nodo al comienzo llamado la cabeza o frente (head o front).
- Hay un nodo de término llamado la cola o atrás (tail o back).
- Una Lista sólo puede ser recorrida en secuencia, usualmente hacia atrás o adelante.
- Hay varias formas de implementar una lista, como se muestra a continuación...

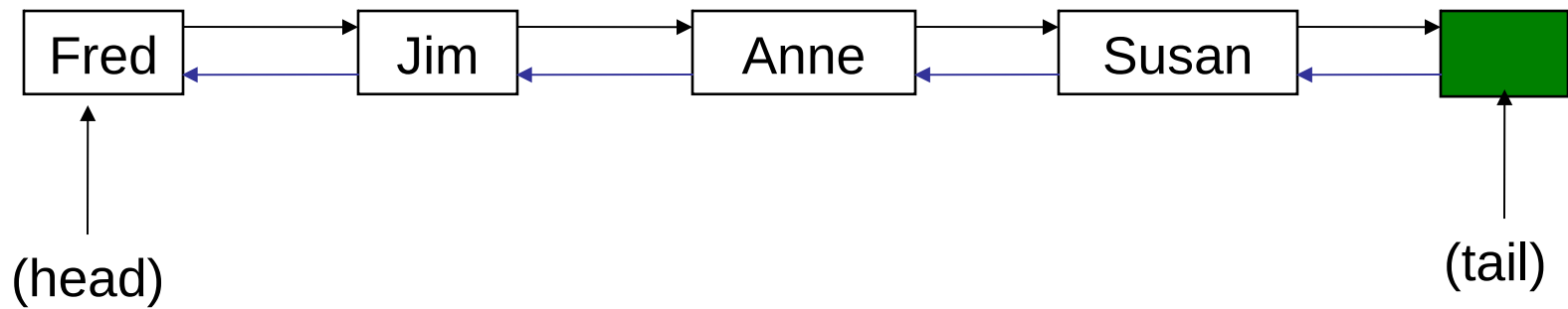
Lista Simplemente Enlazada

- Una lista simplemente enlazada tiene punteros conectando los nodos sólo en dirección hacia la cola. Cada uno de los nodos contiene un string en este ejemplo:



Lista Doblemente Enlazada

- Una lista doblemente enlazada tiene punteros conectando los nodos en ambas direcciones. Esto permite recorrer la lista en ambas direcciones:



La clase List en la biblioteca estándar usa esta implementación.

Clase List Estándar C++

- La clase list es una clase template (plantilla) en la Biblioteca estándar C++
- Podemos crear listas que contengan cualquier tipo de objeto.
- Las clases list y vector comparten muchas operaciones, incluyendo: `push_back()`, `pop_back()`, `begin()`, `end()`, `size()`, y `empty()`
- El operador sub-índice (`[]`) no puede ser usado con listas.

Ver <http://www.cplusplus.com/reference/stl/>.

Métodos de clase List

Table 9.1 Summary of list operations

| | | |
|--|---|---------------|
| Constructors and Assignment | | |
| <code>list<T> v;</code> | Default constructor | $O(1)$ |
| <code>list<T> v (aList);</code> | Copy constructor | $O(n)$ |
| <code>l = aList</code> | Assignment | $O(n)$ |
| <code>l.swap (aList)</code> | Swap values with another list | $O(1)$ |
| Element Access | | |
| <code>l.front ()</code> | First element in list | $O(1)$ |
| <code>l.back ()</code> | Last element in list | $O(1)$ |
| Insertion and Removal | | |
| <code>l.push_front (value)</code> | Add value to front of list | $O(1)$ |
| <code>l.push_back (value)</code> | Add value to end of list | $O(1)$ |
| <code>l.insert (iterator, value)</code> | Insert value at specified location | $O(1)$ |
| <code>l.pop_front ()</code> | Remove value from front of list | $O(1)$ |
| <code>l.pop_back ()</code> | Remove value from end of list | $O(1)$ |
| <code>l.erase (iterator)</code> | Remove referenced element | $O(1)$ |
| <code>l.erase (iterator,iterator)</code> | Remove range of elements | $O(1)^a$ |
| <code>l.remove (value)</code> | Remove all occurrences of value | $O(n)$ |
| <code>l.remove_if (predicate)</code> | Removal all values that match condition | $O(n)$ |
| Size | | |
| <code>l.empty ()</code> | True if collection is empty | $O(1)$ |
| <code>l.size ()</code> | Return number of elements in collection | $O(n)^b$ |
| Iterators | | |
| <code>list<T>::iterator itr</code> | Declare a new iterator | $O(1)$ |
| <code>l.begin ()</code> | Starting iterator | $O(1)$ |
| <code>l.end ()</code> | Ending iterator | $O(1)$ |
| <code>l.rbegin ()</code> | Starting backwards moving iterator | $O(1)$ |
| <code>l.rend ()</code> | Ending backwards moving iterator | $O(1)$ |
| Miscellaneous | | |
| <code>l.reverse ()</code> | Reverse order of elements | $O(n)$ |
| <code>l.sort ()</code> | Place elements into ascending order | $O(n \log n)$ |
| <code>l.sort (comparison)</code> | Order using comparison function | $O(n \log n)$ |
| <code>l.merge (list)</code> | Merge with another ordered list | $O(n)$ |

a. Freeing the memory used by erased cells will require time proportional to the number of elements deleted.

b. Some implementations keep track of the number of elements in a list, and thus can determine the size in $O(1)$.

Agregar y remover nodos

El siguiente código crea una lista, agrega cuatro nodos, y remueve un nodo:

```
#include <list>
```

```
list <string> staff;
```

```
staff.push_back("Fred"); // Add element at the end
```

```
staff.push_back("Jim");
```

```
staff.push_back("Anne");
```

```
staff.push_back("Susan");
```

```
cout << staff.size() << endl; // 4
```

```
staff.pop_back(); // Delete last element
```

```
cout << staff.size() << endl; // 3
```

Iteradores en listas

- Un **iterador (iterator)**, como en vector, es un puntero que se puede mover a través de la lista y provee acceso a elementos individuales.
- El **operador desreferencia (*)** es usado cuando necesitamos obtener o fijar el valor de un elemento de la lista.

```
list<string>::iterator pos;
```

```
pos = staff.begin();  
cout << *pos << endl;      // "Fred"
```

```
*pos = "Barry";  
cout << *pos << endl;      // "Barry"
```


Iteradores

Podemos usar los operadores ++ y -- para manipular iteradores. El siguiente código recorre la lista y despliega los ítems usando un iterador:

```
void ShowList( list<string> & sList )
{
    list<string>::iterator pos;
    pos = sList.begin();

    while( pos != sList.end())
    {
        cout << *pos << endl;
        pos++;
    }
}
```

Iterador Constante (const_iterator)

Si pasamos una lista como constante (const list) debemos usar un **iterador constante** para recorrer la lista:

```
void ShowList( const list<string> & sList )
{
    list<string>::const_iterator pos;
    pos = sList.begin();

    while( pos != sList.end())
    {
        cout << *pos << endl;
        pos++;
    }
}
```

Iterador reverso (reverse_iterator)

Un iterador reverso (reverse_iterator) recorre la lista en dirección inversa. El siguiente lazo despliega todos los elementos en orden inverso:

```
void ShowReverse( list<string> & sList )
{
    list<string>::reverse_iterator pos;
    pos = sList.rbegin();

    while( pos != sList.rend())
    {
        cout << *pos << endl;
        pos++;
    }
}
```

Iterador constante Reverso (const_reverse_iterator)

Un const_reverse_iterator nos permite trabajar con objetos lista constantes:

```
void ShowReverse( const list<string> & sList )
{
    list<string>::const_reverse_iterator pos;
    pos = sList.rbegin();

    while( pos != sList.rend())
    {
        cout << *pos << endl;
        pos++;
    }
}
```

Inserción de Nodos

La función miembro insert() inserta un nuevo nodo antes de la posición del iterador. El iterador sigue siendo válido después de la operación.

```
list<string> staff;  
staff.push_back("Barry");  
staff.push_back("Charles");
```

```
list<string>::iterator pos;  
pos = staff.begin();  
staff.insert(pos, "Adele");  
// "Adele","Barry","Charles"
```

```
pos = staff.end();  
staff.insert(pos, "Zeke");  
// "Adele","Barry","Charles","Zeke"
```

Eliminación de Nodos

La función miembro `erase()` remueve el nodo de la posición del iterador. El iterador es **no válido** después de la operación.

```
list<string> staff;
staff.push_back("Barry");
staff.push_back("Charles");

list<string>::iterator pos = staff.begin();
staff.erase(pos);
cout << *pos;      // error:invalidated!

// erase all elements
staff.erase( staff.begin(), staff.end());

cout << staff.empty(); // true
```

Mezcla de Listas

La función miembro `merge()` combina dos listas en según el operador de orden de los objetos que contiene. Por ejemplo en este caso el orden es alfabético.

```
list <string> staff1;  
staff1.push_back("Anne");  
staff1.push_back("Fred");  
staff1.push_back("Jim");  
staff1.push_back("Susan");
```

```
list <string> staff2;  
staff2.push_back("Barry");  
staff2.push_back("Charles");  
staff2.push_back("George");  
staff2.push_back("Ted");
```

```
staff2.merge( staff1 );
```

Ordenamiento de una Lista

La función miembro `sort()` ordena la lista en orden ascendente. La función `reverse()` invierte la lista.

```
list <string> staff;
```

```
.
```

```
.
```

```
staff.sort();
```

```
staff.reverse();
```


Nota sobre sobrecarga del operador de salida (<<)

- Siempre que sea posible, la sobrecarga de operadores (+,-, etc) debería ser encapsulada como función miembros de la clase.
- Sin embargo, hay ocasiones en que esto genera una expresión difícil de interpretar. En este caso hacemos una excepción a la regla.
- Si pudiéramos:

```
class Point {  
public:  
    ostream & operator <<(ostream & os);  
    .....  
};
```

- Podríamos tener: Point p;
 p.operator <<(cout); // llamado a función
 p << cout // el mismo efecto

Nota sobre sobrecarga del operador de salida (<<)

- Obviamente esta codificación no es intuitiva.
- El usar una función no miembro de la clase nos permite disponer los operandos en el orden “normal”.
- La función debe ser implementada como:

```
ostream & operator << (ostream & os, const Point & p)
{
    os << '(' << p.GetX() << ',' << p.GetY()<<')';
    return os;
};
```

- Si necesitamos acceder a miembros privados de la clase, la declaramos dentro de la clase como función amiga.