Tema 4 resumen/mapa conceptual. Listas, pilas colas.

Alejandro Ruiz López
PROGRAMACIÓN CON ESTRUCTURAS LINEALES

1 Introducción a la creación de listas lista lineal doblemente enlazada LLDE

Nodos: estructura de datos en la que los elementos se encuentran dispersos en memoria y cada uno almacena la posición del elemento inmediatamente posterior y, opcionalmente, del elemento inmediatamente anterior.

Motivos para el diseño de una c:

- Por su versatilidad
- Su separación formal entre las operaciones de inserción, eliminación e intercambio de nodos y las funciones miembro de la lista.

Estructura NODE_BASE: Contiene dos punteros, uno al nodo anterior(prev) y otro al nodo posterior(next), y es utilizada para definir las operaciones de inserción, eliminación e intercambio de nodos en la lista.

2 Operaciones en las listas.

Ligadura / Inserción (Hook): inserta el nodo que la invoca delante del nodo pasado por el argumento.

Desligadura (Unhook): aisla al nodo que la invoca y enlaza los nodos anterior y posterior en la lista original.

Intercambio (Swap): Intercambia dos nodos, distinguiendo entre tres casos:

- Cuando ambos nodos forman parte de la lista.
- Cuando un nodo está aislado.
- Cuando ambos nodos están aislados.

3 Uso de la estructura NODE_BASE.

Implementación: utiliza la estructura NODE_BASE para representar una lista lineal doblemente enlazada (LLDE).
Las Funciones miembro de la clase List<> utilizan la estructura NODE_BASE para realizar operaciones de inserción, eliminación e intercambio de nodos en la lista.
La ventajas de la separación entre estructura NODE_BASE y la clase List <> es que simplifica el diseño del contenedor y evita replicaciones de código.
dnb_ Nodo centinela: Subobjeto dentro de la lista que no almacena datos indica el final de las interacciones en la estructura.

Nota: Se recomienda esbozar gráficos para ver de forma gráfica la correcta implementación de los métodos de inserción, eliminación e intercambio de nodos.

4 Estructura detail::Node_base

Se usa la herencia como mecanismo básico para añadir la información contenida en la estructura detail::Node_base a una nueva estructura privada List<>::Node que contenga la área de datos.

Constructor:

- Inicia una lista vacía.
- Aisla al nodo centinela (prev y next apuntan al propio nodo)
- Introduce mediante push_back() nodos.
- Si hay excepción:
    - Se borra el contenido acumulado clear()
    - Se lanza una excepción.

5. Pilas y colas

Pila (**Stack**) esquema de datos LIFO (Last-In, First Out)
- Último elemento en entrar, primero en salir.
- Funciones:
  - top(): accede al elemento de la cima de la pila
  - push(): añade elemento a la cima de la pila.
  - pop(): elimina el elemento de la cima de la pils.
  - empty(): devuelve si la pila está vacía.
  - size(); devuelve el tamaño de la pila.
	Cola (**Queue**) esquema de datos FIFO (First In, First Out)
	- Primer elemento en entrar, es el primero en salir.
	- Funciones:
	  - front(): acceder al elemento al frente de la cola.
	  - back(): acceder al elemento en el fondo.
	  - push(): insertar un elemento en el fondo.
	  - pop(): eliminar el elemento en el frente.

6. Tipos de Iteradores

Permiten el acceso secuencial a elementos almacenados de una colección.

vector: - v_ - space_ - last_ - begin() - end()	listas: - prev - next - dnb_ - begin() - last()	Condiciones que deben cumplir: - Debe ser construido como copia de otro (CopyConstructible) - Reasignado como copia de otro (CopyAssignable) - Destructible (Destructible) - Intercambiable (Swappable) - Se debe desreferenciar dentro del rango al que accede. - Ejemplo: *it, debe poder ++it debe apuntar al posterior

Categorías:

- InputIterator: solo lectura,sentido ascendente ,acceso, comparación.
- OutputIterator: solo escritura, sentido ascendente, no comparable ni acceso con operador ->
- ForwardIteraor: Acceso ->, lectura y escritura, sentido ascendente, comparable.
- BidirecciionalIterator: Acceso ->, lectura y escritura, sentido ascendente y descendente, comparable. LIST<>
- RandomAccessIterator; Acceso ->, lectura y escritura, sentido ascendente y descendente, offset arbitrario, es comparable. VECTOR<>