SERIALIZABLE LINKED LIST
(Design Document)

Nitin Puranik

OBJECTIVE

This application handles creation and manipulation of singly linked lists along with the added capability of serializing the linked list structures out to disk and deserializing them back to memory. The supported operations are :

- Adding a new list
- Displaying a given list
- Adding a node to a given list
- Removing a node from a given list
- Removing the entire list
- Serialize all the lists in memory out to disk and doing a clean exit.

STRUCTURE OF THE SYSTEM

This system has three core components :

- The singly linked list component that also contains the list node subcomponent.
- The memory recycler component that maintains freed memory chunks for reuse.
- The serializer component that handles disk operations such as serialization, deserialization and disk updation.

There is a fourth component that provides a user interface and interfaces into the above three core components.

LINKED LIST (STRUCTURE)

The linked list structure used in this system is a singly linked list. Singly linked list has been used as no additional advantage would be gained by using a doubly linked list. A doubly linked list would be more suited to applications that would need list traversal in both forward and backward direction. The requirement for this problem states that the lists have to be sorted and hence a doubly linked list would result in redundant space consumption.

The list structure has been defined in the class link_list.h. This file also contains the node structure, the collection of which makes a linked list. The node structure contains the members data, which holds the actual integer data, and a pointer to the next node. The list structure contains a pointer to the head of the list. Once we dereference the head pointer of a list object, we can traverse the list by following the next pointers starting from the head.

Each list object has an integer ID, held by the member ‘mID’. This ID is initialized by the static data member ‘gID’, which incrementally assigns IDs to every new list object created, starting from 1.

MEMORY RECYCLER (STRUCTURE)

The memory recycler has the primary responsibility of retaining ownerships of memory chunks that have been released. The recycler’s structure has been defined in link_memory.h. A vector of pointers to Node objects is declared, called ‘mChunks’. This vector stores the Node pointers of all the nodes that have been deleted by the user. Every time a new data item is added to a linked list, the mChunks vector is checked for reusable node chunks and if found, the chunk is reused again by creating a new Node object on that chunk with the placement new construct.

SERIALIZER (STRUCTURE)

The serializer component is responsible for all the disk operations related to linked lists. The serializer handles such operations as serializing a linked list to disk, bringing the list from disk back into memory, updating the disk file whenever a list item is removed from the list or when the entire list is removed. The serializer also defines certain utility functionalities such as testing if a list is on disk and checking if a list has been marked as deleted.

LINKED LIST (OPERATIONS)

The linked list operations have been defined in the file link_list.cpp. The list header defines a macro called BLOCK. This macro constrains the maximum number of nodes a list can contain in memory. If the list can contain more nodes in memory, then the BLOCK value needs to be increased. If memory is a constraint, then BLOCK value has to be reduced. At present, the BLOCK value has been set to 10.

The design of the list operations has been such that part of the list could reside in memory, while portions of it could be scattered across the disk. Each of the operations are elaborated below.

1. Creation of the list

When the user creates a new list, a new list object is created and the list ID is initialized to the next highest ID, by incrementing and assigning the static gID member to the private mID.

2. Add new items to the list

A global hash table(list_table) is maintained that maps list IDs with pointers to list objects residing in memory. Every time a new list is created, a new entry is made in the list_table structure. Every removal of list causes a corresponding deletion from this table. These steps explain what happens behind the scenes when the user adds new items to the list.

- list_table structure is checked to see if the list resides in memory. If found, the insert function on that object is called.
- If the list is not found in memory, the disk file is checked for the list. If found, then a new list with the same ID is created and the insert function is called on it. Now we have part of the list in memory and part on disk.
- If the list is not found on disk, then the user has entered a non existent list ID and hence is notified with an error message.
- Once in the insert function, the memory recycler checks if a reusable chunk is present in the memory pool.
- If found, the Node object is placed on that chunk and initialized using the placement new construct.
- If not found, then the Node object is newly allocated memory.
- Now, we traverse through the sorted list to find the appropriate spot for insertion and insert the new node there.
- Once the insertion is done, if the list is now found to have reached the maximum number of nodes, then the list is serialized out to disk and removed from memory. Disk serialization is explained in the Serializer section further in the document.

3. Removing items from the list

Below are the steps that occur when the user proceeds to remove items from the list.

- list_table structure is checked to see if the list resides in memory. If found, then we traverse through the list to find the item. Once found, the node object containing the item is removed from the list and the previous node has its next node pointer adjusted. The size data member of the list is decremented and we return.
- If the list is not found to be in memory, or if the data item is not found on the list, then we would have to check if the list exists on disk and if it does, then to search through the disk block containing the list. Once the item is found, the item is marked deleted on file, without proceeding to resize the list. This leaves the file fragmented, but also saves us the overhead of resizing the file.
- If the list is not found on disk, then the user has entered a non existent list ID and hence is notified with an error message.
- If the list is on disk but doesn't contain the data item, the user is notified that the list doesn't contain the data item.

4. Removing the list

When the user chooses to remove the list, the system should make sure that all traces of the list are gone, whether the list exists on memory, on disk or is distributed partially across both the disk and memory. Below are the steps that ensure the list is cleanly removed.

- The list is searched on the disk file and if found, then the list's header variable (flag, explained later) is simply set to zero to indicate that the list has been deleted. The bytes representing the list are not removed from the file and hence the file remains fragmented.
- The list is searched in memory by looking up the list_table structure. If found, then the list object is destroyed. All the node objects in the list are sent off to the recycler for further reuse and the list_table entry is removed. This removes all traces of the list in memory.
- If the list is not found both on disk and in memory, then the user has entered a non existent list ID and hence is notified with an error message.

MEMORY RECYCLER (OPERATIONS)

As mentioned in the memory recycler structure section, the recycler concerns itself with maintaining a container of pointers to Node objects that have been freed before. The objective is to reuse these freed memory chunks while creating new Node objects during data item insertions.

The memory recycler class is instantiated only once, which makes it a singleton class. One global singleton object 'pool' is instantiated during system startup and is used throughout the program. The pool object is accessed during the following scenarios.

- When a new node is created, the pool object is accessed to check if there is a previously deleted node's memory chunk lying around available for reuse. If found, then the chunk is removed from the pool's container and the new Node
object is placed on that chunk using the placement new construct.

- When an existing node is deleted, the node object's memory chunk is dispatched off to the pool's container for future reuse.
- When a list is deleted, all the nodes that make up the list are destroyed and each of their chunks are dispatched off to the pool.
- When the program exits, the pool object is destroyed as a cleanup act.

SERIALIZER (OPERATIONS)

The serializer component is an important part of the system that interfaces the system to disk. The serializer handles all of the disk operations connected to this application. The operations that the serializer handles are :

- Serialize the list out to disk
- Deserialize the list back into memory
- Update the list on disk when the list is removed
- Update the list on disk when a node inside the list is removed
- Check if a list exists on disk
- Check if the list has been marked as removed.

The Serializer class, similar to the Memory recycler class, is instantiated just once, making it a Singleton. During system boot, an object 'serial' is instantiated. The list object is stored on disk in blocks. Each list can be broken up into multiple blocks scattered all over the disk. To process the entire list, one has to start from the first block for the list and follow the offsets stored in the block header to get to the next block, all the way until the last block for the given list is reached. Each block corresponds to a portion of the list and has the following format.

- The block begins with the list ID.

- This is followed by a flag variable that can contain one of the three enumerations :
0 : This means that this list has been deleted. If a list is deleted, the very first block for this list will have the flag set to zero and hence we could stop there, without having to search for further blocks.

1 : This means that this is the last block in the file for the given list. Once we get to this block during list deserialization, it would mean that there are no further blocks for the list and we can stop here. Every time a list grows longer, we need to get to the last block that has the flag value set to 1 and update the flag value to point to the newer block. The newer block will now have the flag value 1 and the one before will point to this block.
<offset> : If this block is not the last block for a given list, then the flag will contain the absolute offset to the next block in the file. This way, for a list that has multiple split blocks, we can jump from block to block until we reach the last block for the list.

- The ID and flag are followed by the actual list elements. Each block will have a preconfigured number of elements or less, as configured by the BLOCK macro in the link_list.h header. If the number of items in the list is less than BLOCK, then the block is padded with fixed values. This ensures that every block in the file is of a fixed width and hence allows us the freedom to jump from block to block in a uniform fashion. If a node in a given block is deleted, then the node is marked deleted, rather than wiping off the bytes off the face of the file. This saves us the overhead of reshuffling the file contents but leaves the file fragmented.

ENVIRONMENT

This application builds and runs on Unix / Linux platforms and compiles with the GNU g++ compiler.