**Linked Lists**

A linked list is a collection of nodes storing data and links to other nodes. In this way, nodes can be located anywhere in memory, and passing from one node of the linked structure to another is accomplished by storing the addresses of other nodes in the linked structure. Although linked lists can be implemented in various ways, the most flexible implementation is by using pointers.

**Ch 3.1 – Singly Linked Lists**

If a node contains a data member that is a pointer to another node, then many nodes can be strung together using only one variable to access the entire sequence of nodes. Such a sequence of nodes is the most frequently used implementation of a linked list, which is a data structure composed of nodes, each node holding some information and a pointer to another node in the list. If a node has a link to only its successor in this sequence, the list if called a singly linked list. An example of such a list is shown in Figure 3.1. Node that only variable p is used to access any node in the list. The last node on the list can be recognized by the null pointer. Each node in the list in Figure 3.1 is an instance of the following class definition:

Text, letter

Description automatically generated

Text

Description automatically generated with medium confidence

A picture containing diagram

Description automatically generated

A node includes two data members: info and next. The info member is used to store information, and this member is important to the user. The next member is used to link nodes to form a linked list. It is an auxiliary data member used to maintain the linked list. It is indispensable for implementation of the linked list, but less important (if at all) from the user’s perspective. Note that IntSLLNode is defined in terms of itself because one data member, next, is a pointer to a node of the same type that is just being defined. Objects that include such a data member are called self-referential objects.

The definition of a node also includes two constructors. The first constructor ini-tializes the next pointer to null and leaves the value of info unspecified. The second constructor takes two arguments: one to initialize the info member and another to initialize the next member. The second constructor also covers the case when only one numerical argument is supplied by the user. In this case, info is initialized to the argument and next to null.

Now, let us create the linked list in Figure 3.1l. One way to create this three-node linked list is to first generate the node for number 10, then the node for 8, and finally the node for 50. Each node has to be initialized properly and incorporated into the list. To see this, each step is illustrated in Figure 3.1 separately. First, we execute the declaration and assignment



which creates the first node on the list and makes the variable p a pointer to this node. This is done in four steps. In the first step, a new IntSLLNode is created (Figure 3.1a), in the second step, the info member of this node is set to 10 (Figure 3.1b), and in the third step, the node’s next member is set to null (Figure 3.1c). The null pointer is marked with a slash in the pointer data member. Note that the slash in the nextmember is not a slash character. The second and third steps—initialization of data members of the new IntSLLNode—are performed by invoking the constructor IntSLLNode(10), which turns into the constructor IntSLLNode(10,0). The fourth step is making p a pointer to the newly created node (Figure 3.1d). This pointer is the address of the node, and it is shown as an arrow from the variable p to the new node. The second node is created with the assignment



where p->next is the next member of the node pointed to by p (Figure 3.1d). As before, four steps are executed:

Text

Description automatically generated

Note that the data members of nodes pointed to by p are accessed using the ar-row notation, which is clearer than using a dot notation, as in (\*p).next. The linked list is now extended by adding a third node with the assignment



where p->next->next is the next member of the second node. This cumbersome notation has to be used because the list is accessible only through the variable p.In processing the third node, four steps are also executed: creating the node (Figure 3.1i), initializing its two data members (Figure 3.1j–k), and then incorporat-ing the node in the list (Figure 3.1l).Our linked list example illustrates a certain inconvenience in using pointers: the longer the linked list, the longer the chain of nexts to access the nodes at the end of the list. In this example, p->next->next->next allows us to access the next mem-ber of the 3rd node on the list. But what if it were the 103rd or, worse, the 1,003rd node on the list? Typing 1,003 nexts, as in p->next->...->next, would be daunt-ing. If we missed one next in this chain, then a wrong assignment is made. Also, the flexibility of using linked lists is diminished. Therefore, other ways of accessing nodes in linked lists are needed. One way is always to keep two pointers to the linked list: one to the first node and one to the last, as shown in Figure 3.2.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text

Description automatically generated

Text

Description automatically generated

The singly linked list implementation in Figure 3.2 uses two classes: one class, IntSLLNode, for nodes of the list, and another, IntSLList, for access to the list. The class IntSLList defines two data members, head and tail, which are pointers to the first and the last nodes of a list. This explains why all members of IntSLLNodeare declared public. Because particular nodes of the list are accessible through pointers, nodes are made inaccessible to outside objects by declaring head and tail private so that the information-hiding principle is not really compromised. If some of the members of IntSLLNode were declared non-public, then classes derived from IntSLList could not access them. An example of a list is shown in Figure 3.3. The list is declared with the statement:



The first object in Figure 3.3a is not part of the list; it allows for having access to the list. For simplicity, in subsequent figures, only nodes belonging to the list are shown, the access node is omitted, and the head and tail members are marked as in Figure 3.3b.

Diagram

Description automatically generated

Besides the head and tail members, the class IntSLList also defines member functions that allow us to manipulate the lists. We now look more closely at some basic operations on linked lists presented in Figure 3.2.

Ch 3.1.1 – Insertion

Adding a node at the beginning of a linked list is performed in four steps.

A screenshot of a computer

Description automatically generated with medium confidence

A picture containing diagram

Description automatically generated

The four steps are executed by the member function addToHead() (Figure 3.2). The function executes the first three steps indirectly by calling the constructor IntSLLNode(el,head). The last step is executed directly in the function by assign-ing the address of the newly created node to head.

The member function addToHead() singles out one special case, namely, in-serting a new node in an empty linked list. In an empty linked list, both head and tail are null; therefore, both become pointers to the only node of the new list. When inserting in a nonempty list, only head needs to be updated.The process of adding a new node to the end of the list has five steps.

Text

Description automatically generated

All these steps are executed in the if clause of addToTail() (Figure 3.2). The else clause of this function is executed only if the linked list is empty. If this case were not included, the program may crash because in the if clause we make an as-signment to the next member of the node referred by tail. In the case of an empty linked list, it is a pointer to a nonexisting data member of a nonexisting node.

A picture containing diagram

Description automatically generated

The process of inserting a new node at the beginning of the list is very similar to the process of inserting a node at the end of the list. This is because the implemen-tation of IntSLList uses two pointer members: head and tail. For this reason, both addToHead() and addToTail() can be executed in constant time O(1); that is, regardless of the number of nodes in the list, the number of operations performed by these two member functions does not exceed some constant number c. Note that because the head pointer allows us to have access to a linked list, the tail pointer is not indispensable; its only role is to have immediate access to the last node of the list. With this access, a new node can be added easily at the end of the list. If the tailpointer were not used, then adding a node at the end of the list would be more com-plicated because we would first have to reach the last node in order to attach a new node to it. This requires scanning the list and requires O(n) steps to finish; that is, it is linearly proportional to the length of the list. The process of scanning lists is illus-trated when discussing deletion of the last node.

Ch 3.1.2 - Deletion

One deletion operation consists of deleting a node at the beginning of the list and returning the value stored in it. This operation is implemented by the member func-tion deleteFromHead(). In this operation, the information from the first node is temporarily stored in a local variable el, and then head is reset so that what was the second node becomes the first node. In this way, the former first node can be deleted in constant time O(1) (Figure 3.6).

Diagram

Description automatically generated

Unlike before, there are now two special cases to consider. One case is when we attempt to remove a node from an empty linked list. If such an attempt is made, the program is very likely to crash, which we don’t want to happen. The caller should also know that such an attempt is made to perform a certain action. After all, if the caller expects a number to be returned from the call to deleteFromHead() and no number can be returned, then the caller may be unable to accomplish some other operations.

There are several ways to approach this problem. One way is to use an assertstatement:

Text, application

Description automatically generated

The assert statement checks the condition !isEmpty(), and if the condition is false, the program is aborted. This is a crude solution because the caller may wish to continue even if no number is returned from deleteFromHead().

Another solution is to throw an exception and catch it by the user, as in:

Text

Description automatically generated

The throw clause with the string argument is expected to have a matching try-catch clause in the caller (or caller’s caller, etc.) also with the string argument, which catches the exception, as in

Text, letter

Description automatically generated

This solution gives the caller some control over the abnormal situation with-out making it lethal to the program as with the use of the assert statement. The user is responsible for providing an exception handler in the form of the try-catchstatement, with the solution appropriate to the particular case. If the statement is not provided, then the program crashes when the exception is thrown. The function f() may only print a message that a list is empty when an attempt is made to delete a number from an empty list, another function g() may assign a certain value to nin such a case, and yet another function h() may find such a situation detrimental to the program and abort the program altogether.

The idea that the user is responsible for providing an action in the case of an exception is also presumed in the implementation given in Figure 3.2. The mem-ber function assumes that the list is not empty. To prevent the program from crash-ing, the member function isEmpty() is added to the IntSLList class, and the user should use it as in:

Text, letter

Description automatically generated

Note that including a similar if statement in deleteFromHead() does not solve the problem. Consider this code:

Text, application

Description automatically generated

If an if statement is added, then the else clause must also be added; otherwise, the program does not compile because “not all control paths return a value.” But now, if 0 is returned, the caller does not know whether the returned 0 is a sign of failure or if it is a literal 0 retrieved from the list. To avoid any confusion, the caller must use an if statement to test whether the list is empty before calling deleteFromHead(). In this way, one if statement would be redundant.

To maintain uniformity in the interpretation of the return value, the last solu-tion can be modified so that instead of returning an integer, the function returns the pointer to an integer:

Text

Description automatically generated

**Ch 3.2 – Doubly Linked Lists**

**Ch 3.3 – Circular Linked Lists**

**Ch 3.7 – Lists in the Standard Template Library**