An Introduction to Linked Lists

Jeff Bienstadt

© December 4, 2023

Abstract

This article presents the linked list data structure. It describes the layout of the linked list and the operations that can be performed with its use. Finally, it presents annotated source code for implementations in the C and C++ languages. The full source code, in both languages, including code to demonstrate the usage of linked lists in applications, is available for download from GitHub[1][2].

Part I
Description

The linked list is a linear data structure that consists of a series of *nodes* that contain the data and which are *linked* together. The data portion of a node may contain any type of data desired. This article will use simple integer values for node data.

The first node in the linked list is the *head* node. An implementation will generally keep a pointer or reference to the head node as a way of keeping track of the linked list itself. A linked list with no head node is empty. The last node in the linked list is the *tail* node. Some implementations also keep track of the tail node, but it is not required.

Nodes in a linked list are not required to be contiguous in memory, and they are unlikely to be. Because nodes are linked rather than each residing in a specific location as in an array, a linked list may "grow" or "shrink" to any size required, without the need to move any surrounding nodes. As a result inserting a new node into the list is fast and efficient. However, locating a given node is comparatively slow as the linked list does not provide for quick random access. Instead the list must be traversed node by node until either the desired node is found or until a terminating node is reached.

There are two types of linked lists: the singly-linked list and the doubly-linked list. This article will present both of these types.

[1] https://github.com/jeffbi/data-structures/tree/master/LinkedList/C
[2] https://github.com/jeffbi/data-structures/tree/master/LinkedList/C++

1 Singly linked lists

A singly-linked list uses nodes that contain the node's data and contain a single pointer to the next node in the list. Singly-linked lists may be traversed in only one direction, toward the list's tail.

Figure 1 is a conceptual view of what a singly-linked list might look like.

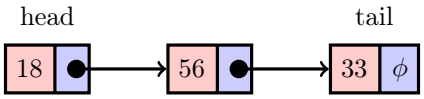

Figure 1: A singly-linked list.

The ϕ symbol in the blue section of the tail node represents a null or empty pointer or reference, depending on the implementation language, that indicates the terminating node.

Some implementations use a pointer or reference to a *dummy* node that indicates the terminating node. The dummy node is not a true node in the linked list as it contains no data—it is simply a placeholder. Such a linked list might look like figure 2.

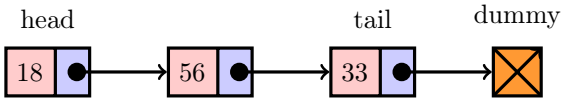

Figure 2: A singly-linked list with a dummy node.

1.1 Accessing a list element

Unlike an array, a linked list is not a *random-access* type of data structure, meaning you cannot simply access any given element with a single operation. To access the ith element of a linked list, it is necessary to *traverse* the list, node by node, until the ith element is found. This means that accessing an element has $O(n)$ time complexity. This is worsened by the fact that since a given node can exist anywhere in memory, nodes are unlikely to be in adjacent memory locations, causing cache misses.

1.2 Traversing a singly-linked list

Iterating across a singly linked list is a bit more involved than simply using an index to reach an element in an array but the mechanism is still quite simple. Start at the head node, follow the pointer to the next node, and repeat until reaching the tail node, which is identified by its *next* pointer being ϕ. The algorithm is shown in figure 3.

To locate a specific node, while iterating over the list compare the data portion of the node to the target data value.

```
1   begin
2     currentnode ← headnode
3     while currentnode.next ≠ ϕ:
4       currentnode ← currentnode.next
5   end
```

Figure 3: Algorithm for traversing a singly-linked list.

1.3 Adding nodes

Linked lists do not have a fixed length. They may grow or shrink to whatever size is needed, constrained only by available memory. There are procedures for appending nodes to the end of the list, prepending nodes at the front of the list, and for inserting a new node into a list.

One advantage of a linked list over an array is that adding a new node to the list is a constant time ($O(1)$) operation. Adding an element to an array will likely require that portions of existing data be copied to make room for a new element, and the entire array my need to be copied into newly-allocated memory if there is not enough room in the array for the new element. A new element is added to a linked list simply by creating a new node and manipulating pointers.

1.3.1 Inserting into a singly-linked list

Inserting a node into a singly-linked list requires a few steps, but is generally more efficient than inserting an element into an array. To insert an element into an array, each following element must be moved down a position to make room for the new element. This is not the case with a linked list since each node points to its next node, wherever it may be in memory.

When inserting a new node into a singly-linked list, because the nodes point in only one direction, the new node in usually inserted *after* a specified node. To insert a new node before a given node would require traversing the linked list until the node that precedes the desired node is found, and inserting after that node.

The algorithm for inserting into a singly-linked list is presented in figure 4.

```
1   procedure insert(after_node, new_node)
2   begin
3     new_node.next ← after_node.next
4     after_node.next ← new_node
5   end
```

Figure 4: Algorithm for inserting a node into a singly-linked list.

Before applying the algorithm, a new node is created. This new node may reside anywhere in memory. The node's *next* pointer does not yet have to point

to anywhere of consequence and is often set to ϕ. See figure 5.

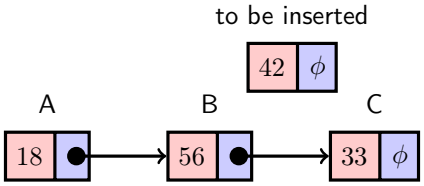

Figure 5: A new node to be inserted into a singly-linked list.

Figures 6 and 7 illustrate the process of inserting the new node into the linked list, between the second and third nodes, labeled B and C respectively, in figure 5. The list has been traversed and stopped at node B.

Node B's *next* pointer is copied into the new node's *next* pointer, resulting in two nodes pointing to the same *next* node, as shown in figure 6. This coresponds to line 3 in the insertion algorithm.

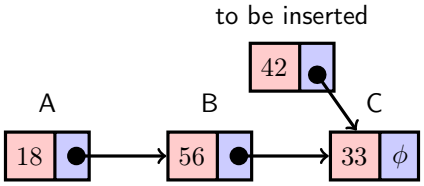

Figure 6: New node now points to its next node.

Node B's *next* pointer is then set to the location of the new node, which is line 4 in the insertion algorithm. As seen in figure 7, the new node is fully inserted into the linked list.

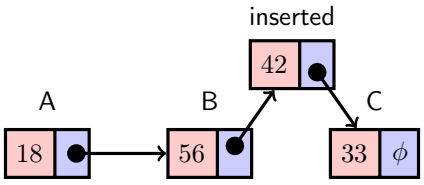

Figure 7: New node inserted, and previous node points to it.

Note that the new node was set to point at the next node *before* setting the current node to point to the new node. Otherwise for a brief moment the new node is the last node in the list and the tail node is no longer part of the list. Additionally, if the new node's *next* pointer was left unitialized on creation then that pointer may point anywhere, not at a valid list node. These issues are important in a multi-threaded environment. By having the new node point to the next node first, these problems are avoided.

An unfortunate side effect of inserting a node after a specified node is that there is no node preceding the head node, making insertion of a node to be the new head node is problematic. Fortunately prepending a node to the front of a linked list is simple.

1.3.2 Prepending to the front of a singly-linked list

Adding a new node to the front of a linked list requires the current head node and the new node to be prepended. By setting the new node's *next* pointer to the curent head node, the new node becomes the new head node.

1. Create the new node.
2. Set the new node's *next* pointer to point to the current head node.
3. Set the head node to the new node

The prepended node is now the new head node and the linked list is complete.

1.3.3 Appending to the end of a singly-linked list

Appending a new node to a linked list is simpler than inserting a node into the linked list, although the tail node must be located, which generally means either traversing the linked list or maintaining a pointer or reference to the tail node.

1. Create the new node and set its *next* pointer to a null pointer or empty reference.
2. Locate the tail node. If a large number of appends are expected, keeping a reference or pointer to the tail node can speed this process.
3. Set the tail node's *next* pointer to point to the new node.

It is important to set the new node's next pointer to ϕ before changing the original tail node's pointer, to avoid having a brief time in which the new node is not a valid tail node.

1.4 Removing nodes

Removing a node from a singly-linked list is similar to the insertion process. The actual removal is, like insertion, a $O(1)$ constant-time operation. Since there is no direct access to a node's previous node it is necessary to remove the node *after* a given node. Figure 8 shows a singly-linked list with a node to be removed, node C.

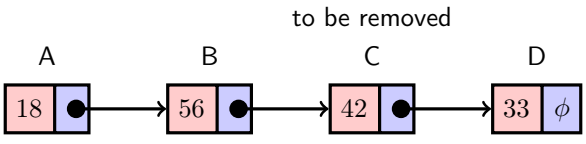

Figure 8: A singly-linked list with a node to be removed.

The following steps remove node C by removing the node that comes after node B in the linked list. Of course the first step is to locate node B, which requires traversing the linked list until a node's next node is the one to be removed, which is a $O(n)$ linear-time operation.

Once node B, the node previous to the node to be removed, is found, the *next* pointer from node C, the node to be removed, is copied to node B's *next* pointer, resulting in the *next* pointers for both nodes A and B pointing to node D, as shown in figure 9.

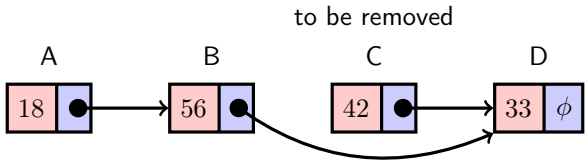

Figure 9: A singly-linked list with a Node B partially removed.

Finally, the *next* pointer of the removed node is set to ϕ, discarding that node's link and completing the removal, illustrated by figure 10

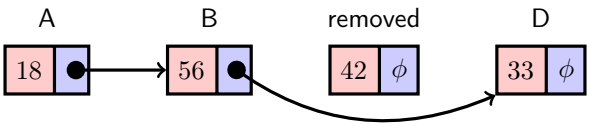

Figure 10: A singly-linked list with Node C fully removed.

Removing the head node requires special handling since there is no node preceding it. Since most implementations maintain a reference to the head node, all that is required is that the maintained reference or pointer be set to the current head node's *next* node. Then the former head node's *next* pointer may be set to ϕ to completely disassociate that node from the linked list.

The C implementation of the singly-linked list in Part II, section 4, incorporates this logic into the `sll_remove_node` function.

2 Doubly linked lists

A doubly-linked list is very similar to a singly-linked list. The primary difference is that a node contains a pointer to the previous node as well as a pointer to the next node. This allows the linked list to be traversed both forward and backward, and makes it much simpler to reach a node's predecessor. Figure 11 illustrates a doubly-linked list.

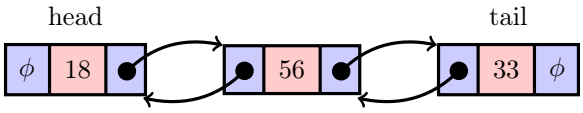

Figure 11: A doubly-linked list.

2.1 Traversing a doubly-linked list

Traversing a doubly-linked list is exactly the same as with a singly-list list: follow each node's next pointer until the node that has no next pointer. One advantage of a doubly-linked list is that it can be traversed backward as well as forward: follow each node's previous-node pointer until reaching the head node which has no previous-node pointer.

2.2 Adding nodes

Like singly-linked lists, doubly-linked lists do not have a fixed length and may grow or shrink to whatever size is needed, constrained only by available memory. New nodes may be easily appended to the end, prepended to the beginning, or inserted anywhere inside the linked list.

2.2.1 Inserting into a doubly-linked list

Inserting a node into a doubly-linked list involves essentially the same steps as used when inserting into a singly-linked list, but there are additional node pointers to contend with. However, an advantage to the additional nodes is that inserting a node before a given node is as efficient as inserting after a node.

First a new node is created, which may be created anywhere in memory. The node's previous and next pointers are usually initialized to ϕ. Figure 12 illustrates this.

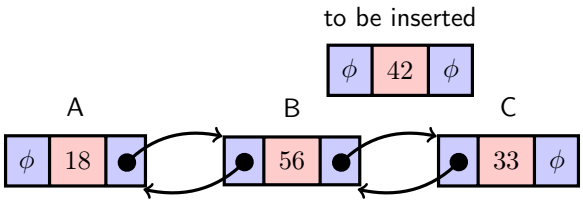

Figure 12: A new node to be inserted into a doubly-linked list.

Figures 13 and 14 demonstrate the process of inserting the new node between nodes B and C. Since nodes B and C point to each other, forward and backward, the process is the same whether inserting after node B or before node C.

The new node's previous pointer is set to point to node B, and its next pointer is set to point to node C, as shown in figure 13. It makes no difference

whether the previous pointer is set first or the next pointer, but the two steps together should be done before modifying the pointers of existing nodes in the linked list.

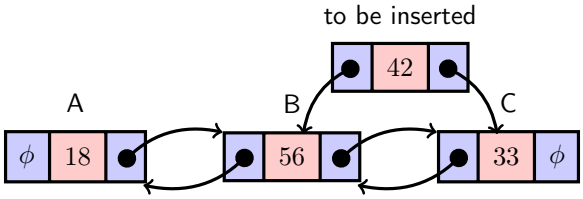

Figure 13: New node now points to its previous and next nodes.

Once the new node's pointer have been set, node B's *next* pointer and node C's *previous* pointer are both set to the new node, as shown in figure 14, completing the insertion.

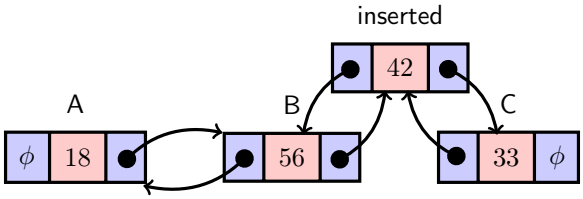

Figure 14: New node inserted, both previous and next nodes point to it.

2.2.2 Appending to the end of a doubly-linked list

Appending a node to the end of a doubly-linked list is similar to appending to a singly-linked list, with the exception of handling an additional node pointer.

- Create the new node and set its *next* pointer to a null pointer or empty reference.
- Locate the tail node.
- Set the new node's *previous* pointer to point to the current tail node.
- Set the current tail node's *next* pointer to point to the new node.

As with a singly-linked list, the tail node must be located. Some implementations maintain a pointer or reference to the tail node for this purpose.

2.2.3 Prepending to the front of a doubly-linked list

Adding a new node to the beginning of a doubly-linked list is again similar to doing so to a singly-linked list. As with appending, there is another node pointer to consider.

- Create the new node, setting its *previous* pointer to ϕ.
- Set the new node's *next* pointer to point to the current head node.
- Set the current head node's *previous* pointer to the new node.
- Set the new head node to be the new node.

2.3 Removing nodes

One advantage of a doubly-linked list over a singly-linked list is that a node can be removed directly, without the need to traverse the linked list looking for the previous node. Figure 15 is a doubly-linked list before removing node C.

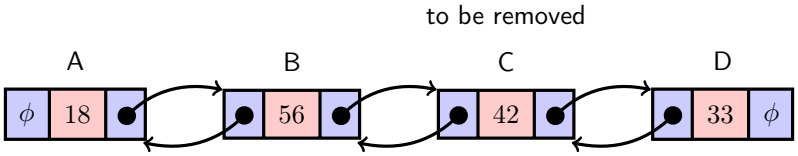

Figure 15: Doubly-linked list, node C to be removed.

First, node C's *previous* and *next* pointers are used to locate the nodes prior to and following it. Node B's *next* pointer is set to point to node D, and node D's *previous* pointer is set to point to node B, as shown in figure 16.

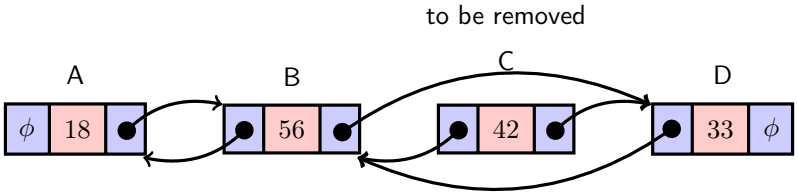

Figure 16: Nodes B and D now point to each other.

Once nodes B and C point to each other, node C has effectively been removed. To ensure the removed node no longer refers to any node still in the linked list, its *previous* and *next* pointer are set to ϕ, as illustrated in figure 17.

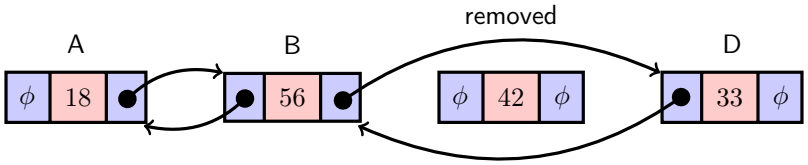

Figure 17: Node C fully removed.

If the node to be removed is the head or tail node, the implementation must take care to not attempt to have a non-existant node point somewhere.

3 Using Linked Lists

Linked lists are used in a variety of situations, such as when insertion or removal of an item into a collection needs to be fast, or when the size of a collection is not known in advance.

It is common in C-language APIs for a function to return a pointer to the beginning of a linked list. Imagine a function that retrieves a list of printer names. Rather than require one function call to determine how many names will be returned, require the user to allocate memory for an array of the given size, and another function call to fill the array, the API can provide a single function that returns the head of a linked list of names. Each list node might look like

```
typedef struct NameNode_ {
  const char        *name;
  struct NameNode_  *next;
} NameNode;
```

The code to retrieve and process the printer names would look something like

```
NameNode *list = get_printer_names();
NameNode *node = list;  /* start at the head of the list */
while (node != NULL) {
  /* do something with the name in node->name */
  ...
  node = node->next;
}
```

Of course, the API would require a companion function to free the linked list when the user is done with it. The implementation of such a function might look like:

```
void free_name_list(NameNode *head) {
  NameNode *node = head;
  while (node) {
    NameNode *next = node->next;
    free(node);
    node = next;
  }
}
```

Linked lists are also used when implementing other data structures such as stacks and queues, acting as the underlying data structure.

Part II
Implementation

This part provides sample implementations of linked lists, in both the C and C++ languages. The C program implements a singly-linked list, and is simple and straightforward, while the C++ implementation utilizes an Object Oriented Programming approach to create a doubly-linked list.

4 C Implementation

Here we present an implementation of a singly-linked list in the C language. This implementation uses simple integers for the type of data stored in a node. The code can be easily modified to work with any data type.

The implementation consists of two C source files, `single_linked_list.h` and `single_linked_list.c`. It begins in `single_linked_list.h` by defining the linked list node type.

```
1  typedef struct _sll_node
2  {
3    int data;
4    struct _sll_node *next;
5  } sll_node;
```

The `data` member contains the linked list node data. The `next` member is a pointer to the next node in the linked list.

Following the node type definition are prototypes for the functions operating on the linked list.

```
6  sll_node *sll_create_node(int data);
7  sll_node *sll_find_data(sll_node *start, int data);
8  sll_node *sll_find_tail_node(sll_node *start);
9  sll_node *sll_append_node(sll_node **head, sll_node *node);
10 sll_node *sll_append_data(sll_node **head, int data);
11 sll_node *sll_insert_node_after(sll_node *after, sll_node *node);
12 sll_node *sll_insert_data_after(sll_node *after, int data);
13 sll_node *sll_prepend_node(sll_node **head, sll_node *node);
14 sll_node *sll_prepend_data(sll_node **head, int data);
15 sll_node *sll_remove_after(sll_node **head, sll_node *node);
16 sll_node *sll_remove_node(sll_node **head, sll_node *node);
17 void sll_erase(sll_node **head);
```

Many of these functions take the head node as a pointer-to-pointer to the `ssl_node` structure. This is because these operations have at least the potential to modify the linked list such that there would be a new head node and it is necessary to keep track of the current head node.

This completes `single_linked_list.h`. The function implementations are in `single_linked_list.c`.

The first thing to do is to `#include` the C library's `stdlib.h` file and the `single_linked_list.h` file defined above. The `stdlib.h` file contains declarations for the functions `malloc` and `free` which will be used to allocate and deallocate memory for the linked list nodes.

```
1  #include <stdlib.h>
2  #include "single_linked_list.h"
```

Next is the implementation code for the functions that operate on the linked list, starting with `sll_create_node`.

```
3  sll_node *sll_create_node(int data) {
4    sll_node *node = (sll_node *)malloc(sizeof(sll_node));
5    if (node != NULL) {
6      node->next = NULL;
7      node->data = data;
8    }
9
10   return node;
11 }
```

On line 4 memory is allocated for the new node using the C library's `malloc` function. Line 5 tests that the allocation was successful, then lines 6 and 7 set the node's `next` pointer to `NULL` and copies the data into the node. Setting the node's `next` pointer to `NULL` is not strictly required, but doing so is a safety measure against a possible runaway list. Finally, on line 10, the function returns a pointer to the newly created node.

Next is the definition of the `sll_find_data` function which traverses the list, begining at the head node, looking for the specified data and returning a pointer to the node containing that data, or `NULL` if the data was not found.

```
12 sll_node *sll_find_data(sll_node *start, int data) {
13   while (start != NULL) {
14     if (start->data == data)
15       break;
16     start = start->next;
17   }
18
19   return start; // Will be NULL if data is not found
20 }
```

The `while` statement on line 13 moves the `start` pointer from node to node until either the data node is found or until `start` contains the `NULL` pointer value from the `next` pointer of the tail node. Line 16 advances the `start` pointer to the next node in the linked list.

Next, the `find_tail_node` function is defined. This function traverses the linked list starting at a specified node until the tail node is found. The starting node might usually be the head node, but that is not required.

```
21 sll_node *sll_find_tail_node(sll_node *start) {
22   if (start != NULL)
```

```
23      while (start->next != NULL)
24        start = start->next;
25
26    return start;
27  }
```

Line 22, tests the `start` node to ensure that it is not `NULL`. If the `start` node is `NULL`, control falls to the `return` statment on line 26.

Lines 23 and 24 use the `start` parameter to traverse the list, updating `start` to point to the next node in the linked list until the node's `next` pointer is `NULL`, indicating that the node is the tail node. Line 26 returns a pointer to that node.

The `sll_append_node` function appends a new node to the end of the list.

```
28  sll_node *sll_append_node(sll_node **head, sll_node *node) {
29    if (*head == NULL)
30      *head = node;
31    else
32      list_find_tail_node(*head)->next = node;
33
34    return node;
35  }
```

The function parameter list on line 28 takes `head` as a pointer-to-pointer to an `ssl_node` structure. This allows the pointer to the head node to be modified if necessary. In this case, if the linked list is empty as indicated by the head pointer being `NULL` (line 29) then the new node becomes the new head node on line 31.

If the linked list is not empty, then line 32 locates the tail node and sets its `next` pointer to point to the new node. A pointer to the new node is returned.

The `sll_append_data` function is very simple.

```
36  sll_node *sll_append_data(sll_node **head, int data) {
37    return sll_append_node(head, sll_create_node(data));
38  }
```

A new node is created by calling `sll_create_node` with the specified data, then that node is appended to the end of the linked list by calling `sll_append_node`. A pointer to the newly created and appended node is returned. Again, the head of the linked list may be changed to point to a new node.

`sll_insert_node_after` inserts a new node immediately following a given existing node.

```
39  sll_node *sll_insert_node_after(sll_node *after, sll_node *node) {
40    node->next = after->next;
41    after->next = node;
42
43    return node;
44  }
```

On line 40 the new node's `next` pointer is set to `after`'s `next` pointer. There are now two nodes pointing to the same following node. Then, on line 41, the `next`

pointer of the `after` node is modified to point to the new node, completing the link. These are the steps illustrated in figures 6 and 7 in Part I.

The `sll_insert_data_after` function is similar to the `sll_append_data` function. It calls `sll_create_node` to create a new node, then passes that node pointer on to `sll_insert_node_after` and returns a pointer to the new node.

```
45 sll_node *sll_insert_data_after(sll_node *after, int data) {
46   return sll_insert_node_after(after, sll_create_node(data));
47 }
```

The `sll_prepend_node` function adds a new node to the front of the linked list, making that node the new head node.

```
48 sll_node *prepend_node(sll_node **head, sll_node *node) {
49   node->next = *head;
50   *head = node;
51
52   return node;
53 }
```

Line 49 sets the new node's `next` pointer to point to the current head node.

Line 50 directly changes the head node by setting `*head` to the new node pointer. Again, note the additional level of indirection. `node` is now the new head node of the linked list.

The `sll_prepend_data` function calls `sll_create_node` and `sll_prepend_node` to create a new node with the specified data and make that node the new head of the linked list.

```
54 sll_node *sll_prepend_data(sll_node **head, int data) {
55   return sll_prepend_node(head, sll_create_node(data));
56 }
```

The implementation provides two functions for removing a linked list node. The first, `sll_remove_node_after` removes the node *following* the given node.

```
57 sll_node *sll_remove_node_after(sll_node **head, sll_node *node) {
58   if (node->next != NULL) {
59     sll_node *next = node->next;
60
61     node->next = next->next;
62     free(next);
63   }
64
65   return node->next;
66 }
```

Line 58 determines whether the passed-in node has a node following it. If so, that node will be removed and line 59 saves a pointer to that node. Line 61 sets the passed-in node's next pointer to point to the node following the node to be removed. This unlinks the desired node from the linked list. That node's next pointer still points to the following node, but it is not necessary to reset that

pointer since line 66 disposes of the removed node with the C Library's `free` function.

Note that only the memory for each node is deallocated. If the `data` portion of the node is a pointer or contains pointers to allocated memory, it is the user's responsibility to perform any required memory management before removing the node.

Whether there was a next node to remove or not, on line 65 the function returns a pointer to the node now following the passed-in node, which might be `NULL`.

The `sll_remove_node` function removes a node from a linked list, and is the longest of this implementation. If the node to be removed is the head node, the next node in the list must be made the new head node. Otherwise the list must be traversed to locate the node that appears before the one to be removed.

```
67  sll_node *sll_remove_node(sll_node **head, sll_node *node) {
68    if (node == *head) {
69      *head = node->next;
70      free(node);
71      return *head;
72    }
73    else {
74      sll_node *previous = *head;
75
76      while (previous != NULL) {
77        if (previous->next == node)
78          return sll_remove_node_after(head, node);
79
80        previous = previous->next;
81      }
82    }
83
84    return NULL;
85  }
```

Line 68 determines whether the node to be removed is the head node and if so line 69 resets the current head node to the next node, unlinking the node from the linked list. The removed node is then disposed of with the standard C library function `free` and the new head node is returned.

If the node to be removed is not the head node, the list must be traversed from the beginning to locate the node previous to the one being removed. A new variable is created to traverse the linked list on line 74, and is initialized to point to the head node. A `while` loop is employed on line 83 to traverse the list until the `previous` variable becomes NULL, in which case the node was not found.

Inside the `while` loop, on line 77, if `previous`'s `next` pointer points to the node to be removed then `node`'s previous node has been found. That node is passed to the `sll_remove_node_after` function to perform the actual removal, and the result of that function is returned.

The final function in the C implementation is `sll_erase`, which traverses the

linked list, freeing the memory allocated for each node.

```
86  void sll_erase(sll_node **head) {
87    sll_node *current = *head;
88
89    while (current != NULL) {
90      sll_node *next = current->next;
91
92      free(current);
93      current = next;
94    }
95
96    *head = NULL;
97  }
```

The function is passed a pointer-to-pointer to the head node because the head node will ultimately be set to `NULL` to indicate that the linked list is empty.

On line 87 a new node variable is created to point to the current node as the linked list is being traversed. Line 89 begins the `while` loop that traverses the linked list. The `current` node's `next` pointer is stored in a temporary variable and the memory allocated to the `current` node is deallocated by calling the C library's `free` function, on line 92. `current` is set to point to the next node in the linked list, the value stored in the temporary variable, and the loop continues until there are no more nodes in the list.

Finally, the head node is set to NULL, indicating that the linked list is now empty.

This completes the implementation of a singly-linked list in C, produced in under 100 lines of code.

Implementing a doubly-linked list in C is similar to a singly-linked list. The main difference in a doubly-linked list is that the node structure contains an additional pointer to point to the previous node, making it possible to traverse the linked list in both directions.

A complete implementation of a doubly-linked list in C is available for download from the git repository.

5 C++ Implementation

This section presents an implementation of a doubly-linked list in the C++ language. The implementation utilizes Object Oriented Programming principles, particularly encapsulation. Templates are also used so a linked list can use any copyable data type.

This is a header-only implementation consisting only of `DoubleLinkedList.h`. The user need only include the header file. No additional library or object code is required.

The code produces a basic linked list. There are no exceptions thrown as there is no real error checking.

The implementation begins by defining the `DoubleLinkedList` class.

```
1  template<typename T>
2  class DoubleLinkedList {
```

`DoubleLinkedList` is a class template, allowing the linked list to be used with any copyable data type.

The core of a linked list is the *node*. In this implementation, the node is defined as a nested, public class type.

```
3  public:
4    class node_t {
5    private:
6      T       _data;
7      node_t *_next{nullptr};
8      node_t *_prev{nullptr};
9
10   public:
11     explicit node_t(const T &data)
12       : _data{data}
13     {}
14
15     node_t(const node_t &) = delete;
16     node_t & operator=(const node_t &) = delete;
```

Line 3 begins the definition of the `node_t` class which defines the linked list node. The `node_t` class defines three private data members on lines 6, 7 and 8. The `_data` member is of type `T`, the declared template type, and holds the node's copy of the data stored in the linked list. The `_next` and `_prev` members, of type `node_t *`, are pointers to the next and previous nodes, respectively, in the linked list. Lines 11–13 define the `node_t` type's `explicit` constructor, which takes a reference to an object of type `T`. A node is not copyable and it is not assignable, as specified by lines 15 and 16.

The `node_t` class contains three public functions, `data()`, `next()`, and `prev()` which provide access to the content of the linked list item data and read access to the node's *next* and *previous* pointers. The class also provides two private functions, `next(const node_t *node)` and `prev(const node_t *prev)`. These functions are private because they are meant to be called only by the `DoubleLinkedList` class to set the values of the node's *next* and *prev* pointers.

```
17     T &data() noexcept {
18         return _data;
19     }
20
21     node_t *next() const noexcept {
22         return _next;
23     }
24
25     node_t *prev() const noexcept {
26         return _prev;
27     }
28
```

```
29    private:
30      void next(node_t *node) {
31        _next = node;
32      }
33
34      void prev(node_t *node) {
35        _prev = node;
36      }
37
38      friend DoubleLinkedList;
39    };
```

Finally, on line 38 the `DoubleLinkedList` class is declared to be a `friend` of the `node_t` class. This allows `DoubleLinkedList` access to the node's private functions and data members.

That is the entire definition of the nested `node_t` class. The definition of the `DoubleLinkedList` class continues with the definition of three private data members.

```
40  private:
41    node_t *_head_node{nullptr};
42    node_t *_tail_node{nullptr};
43    size_t  _count{0};
```

The `DoubleLinkedList` class maintains the pointer to the head node so the user does not have to. To increase the speed of some operations, and as a convenience to the user, the class maintains a pointer to the tail node, and a count of the number of nodes currently in the linked list.

The implementation of the `DoubleLinkedList` class continues, beginning with construction and destruction.

```
44  public:
45    DoubleLinkedList() noexcept = default;
46
47    DoubleLinkedList(const DoubleLinkedList &) = delete;
48    DoubleLinkedList & operator=(const DoubleLinkedList &) = delete;
49
50    ~DoubleLinkedList() {
51      erase();
52    }
```

The default constructor is not required to perform any specific initialization, so it is declared as `default` on line 45 to have the compiler generate it. The class is not copyable and not assignable, so lines 47 and 48 use the `delete` keyword to tell the compiler to not generate those functions.

The destructor, on line 50, is implemented because it must delete all of the linked list's nodes. The `erase` function, which is defined later, performs this task.

Following the constructors and destructor are the operational functions of `DoubleLinkedList`, beginning with the `head`, `tail`, `size`, and `is_empty` functions.

```
53   node *head() const noexcept {
54     return _head_node;
55   }
56
57   node_t *tail() const noexcept {
58     return _tail_node;
59   }
60
61   size_t size() const noexcept {
62     return _count;
63   }
64
65   bool is_empty() const noexcept {
66     return head() == nullptr;
67   }
```

The `head` function simply returns the `_head_node` member. On line 57 the `tail` function returns the `_tail_node` data member. The `size` function returns the `_count` member and the `is_empty` function returns a `bool` indicating whether the `_head_node` is equal to `nullptr`.

Next are the functions that add and remove items from the linked list. The first is the `prepend` function, which adds an item to the front of the linked list, before the current head node. The added item becomes the new head node.

```
68   node_t *prepend(const T &data) {
69     node_t *new_node{new node_t(data)};
70
71     new_node->next(head());
72     if (is_empty())
73         _tail_node = new_node;
74     else
75         head()->prev(new_node);
76     _head_node = new_node;
77
78     ++_count;
79
80     return new_node;
81   }
```

The `prepend` function takes a `const` reference to an object of type `T`, which is the type given to the class' template parameter. On line 69 a new node is allocated and constructed using the `new` operator. The new node's *next* pointer is then set to point to the current head node, which may or may not be `nullptr`. If the linked list is empty, line 73 sets the `_tail_node` member to the new node since the new node will be the only node in the list. Otherwise there is a head node and its *prev* node is set to the new node on line 75. In either case, line 76 sets the current head node to be the new node. The count is incremented and the pointer to the new node is returned.

Note that this function invokes the node's private `next(node_t *node)` and `prev(node_t *node)` member functions. This is possible because the `node_t`

class declared `DoubleLinkedList` to be a `friend`.

The next function to be defined is `insert_after`, which inserts a new item into the linked list immediately following a given node.

```
82  node_t *insert_after(const T &data, node_t *node) {
83    node_t *new_node{new node_t(data)};
84
85    new_node->_next = node->_next;
86    new_node->_prev = node;
87    node->_next = new_node;
88    if (new_node->_next == nullptr)
89      _tail_node = new_node;
90    else
91      new_node->_next->_prev = new_node;
92
93    ++_count;
94
95    return new_node;
96  }
```

`insert_after` takes two parameters, a `const` reference to the data to be inserted and a pointer to a node after which the new data will be inserted.

On line 83 a new node is allocated and constructed. Lines 85–87 stitch the nodes together into the linked list by setting the new node's `_next` pointer to the passed-in node's `_next` pointer, setting the new node's `_prev` pointer to the passed in node, then finally setting the passed-in node's `_next` pointer to point to the new node, completing the insertion. These are the steps illustrated in figures 13 and 14 in Part I. If the new node has no next pointer then it is the new tail node as well, and line 89 sets the linked list's tail node to be the new node. Otherwise the new node is being inserted not only after the specified node, but also before the following node, so the following node's *prev* pointer is set to the new node on line 91. The node count is incremented and a pointer to the new node is returned.

A doubly-linked list provides the ability to easily insert a new element before a given node, something that is more involved with a singly-linked list.

```
97  node_t *insert_before(const T &data, node_t *node) {
98    return node == head() ? prepend(data)
99                          : insert_after(data, node->prev());
100 }
```

The `insert_before` function tests whether the specified node is the head node. If true, the new node is prepended to the front of the linked list and becomes the new head node. Otherwise, there is a node preceding the specified node and `insert_after` is called with that preceding node.

The `append` function adds a new item to the end of the linked list, following the tail node.

```
101 node_t *append(const T &data) {
102   return is_empty() ? prepend(data)
```

```
103                       : insert_after(data, tail());
104   }
```

Again, the `append` function takes a `const` reference to an object of type `T`. If the linked list is empty then the data object is passed on to the `prepend` function defined above. Otherwise the data is given to the `insert_after` function to be inserted after the tail node.

A doubly-linked list provides for a relatively simple mechanism for removing a node directly, eliminating the need for a `remove_after` function. The `remove` function takes as a parameter the node to be removed, and returns a pointer to the following node if any.

```
105   node_t *remove(node_t *node) {
106     node_t *next_node{node->next()};
107
108     if (node == head()) {
109       if (next_node == nullptr) {
110         _head_node = _tail_node = nullptr;
111       }
112       else {
113         next_node->prev(nullptr);
114         _head_node = next_node;
115       }
116     }
117     else if (node == tail()) {
118       node->prev()->next(nullptr);
119       _tail_node = node->prev();
120     }
121     else {
122       node->prev()->next(next_node);
123       next_node->prev(node->prev());
124     }
125
126     delete node;
127     --_count;
128
129     return next_node;
130   }
```

On line 106 a pointer to the specified node's *next* node is stored.

Line 108 determines if the node to be removed is the head node. If so, then if the head node is the only node in the linked list (line 109) then the head and tail nodes are set to `nullptr` making the linked list empty. Otherwise, there is a next node and that node's *prev* pointer is set to `nullptr`, making it the new head node.

If the node to be removed is not the head node but is instead the tail node (line 117), then the node prior to the specified node is made the new tail node.

If the node to be removed is neither the head nor tail node (the `else` on line 121), the node is extricated from the linked list by adjusting the *next* pointer of the previous node and the *next* pointer on the next node.

Finally, the memory for the node to be removed is reclaimed, the node count is decremented, and the following node is returned. That node might be `nullptr`.

Next is the `erase` function, which was used in the destructor to remove all nodes in the linked list. It is a public member function so it is available for users of the class as well.

```
131   void erase() {
132     node_t *current{head()};
133
134     while (current != nullptr) {
135       node_t *next{current->next()};
136
137       delete current;
138       current = next;
139     }
140
141     _head_node = _tail_node = nullptr;
142     _count = 0;
143   }
```

The `erase` function iterates all nodes in the list, beginning with the head node on line 132. The `while` statement line 134 loops until the current node is `nullptr`, indicating the end of the linked list. For each node in the linked list, lines 135–138 save the value of the current node's *next* pointer, free memory allocated to the node with the `delete` operator, then re-assign the current node to the saved pointer to the next node. Once all nodes have been freed the head and tail nodes are set to `nullptr` and the node counter is set to 0 signifying that the linked list is empty.

One more function in the public interface remains. The `find` function traverses the linked list and returns the first node that contains a specified data value.

```
144   node_t *find(const T &data) {
145     node_t *current{head()};
146
147     while (current != nullptr) {
148       if (current->data() == data)
149         break;
150
151       current = current->next();
152     }
153
154     return current;
155   }
```

The search begins at the head node on line 145. While the current node pointer in the search points to an existing node (line 147), if the current node's data member is equal to the `data` parameter the loop terminates (lines 148 and 149), otherwise the current node pointer is set to the next node in the linked list. Regardless of how the loop terminated, the current node is either the node

containing the data or is `nullptr`, indicating that the end of the linked list was reached without finding the data. On line 154 that node value is returned.

This completes the implementation of a doubly-linked list in C++, produced in 155 lines of code.

Implementing a singly-linked list in C++ is similar to a doubly-linked list. The main difference in a singly-linked list is that the node structure contains only the node's data and a pointer to the next node, so it is possible to traverse such a linked in only in the forward direction.

A complete implementation of a singly-linked list in C++ is available for download from the git repository.

As an exercise, you are encouraged to implement both singly- and doubly-linked list in the programming languages of your choice.