Single Linked Lists

Linked Lists

Linked lists are a basic data structure used to keep track of information similar to arrays. They have several major differences from arrays. First, they do not occupy contiguous sections of memory like an array does, so you can't use indexes to reference arbitrary items in a single step. Instead, you must walk down the array from the beginning to find a given item. Second, they can dynamically grow and shrink in the memory they take up as you add and remove items from them. Third, you can add and remove items from the middle of a linked list without having to move other items.

This document will look at single linked lists, the simplest form of linked lists.

Basic structure

A linked list is a set of links joined using pointers between them. A link is the basic unit, much in the same way that a chain is composed of separate links joined together. Links can be implemented as structs or objects, in this example we are showing using structs to create the list.

The list consists of a pointer to the start of the list, typically called the head. Each link then contains some data and a pointer to the next item, typically labeled next. Finally the last link in the list points to **nullptr** to terminate the list. This is shown in the following diagram:

head -> link -> link -> link -> … -> link -> nullptr

Link Struct

Here we will be defining a structure that contains the data a pointer to the next Link. Note that we can do this sort of circular references because all pointers are of the same length, so the compiler knows how much space to allocate for the entire structure

```
struct Link {
    int value;
    Link * next;
};
```

For demonstration purposes here, value is defined to be an integer. In the real world, you could have it be any data type or combination of data types.

List

For this example of a simple linked list, all we need to start with is a head as a pointer to the beginning of the list

```
Link * head;
```

Access Methods

Here we are need to declare the required methods. Note, any method that modifies the list must return the head, since it may have changed by adding or removing a link.

```
        // add a new link to the start of the list
Link* addItem(Link * head, int value);

        // return a string that displays the list
        // starting at the head
std::string display(Link * head);

        // return true if a value is present in the list
bool isThere(Link * head, int value);

        // delete all links in the list
void cleanUp(Link * head);

        // return the number of links in the list
int getLength(Link * head);

        // if a value is present, delete its link
bool deleteItem(Link * &head, int value);

        // Use selection sort to sort the list
        // this version changes the contents of links
        // rather than changing the order of the links themselves
void sorter(Link * head);
```

When you are using a list, the particular set of methods you would implement depends on how you are using the list. For this document, we are implementing a range of methods that show the possibilities of a single linked list.

CleanUp Method

Since we are using dynamic memory, it is necessary to have a method to delete all links when we are done with the list. It starts at the head and walk down the list, deleting any items that are remaining.

```
        // delete all links in the list
        // recursive version
void cleanUp(Link * head)
{
    if ( head )
    {
        cleanUp(head->next);
        delete head;
    }
}

        // delete all links in the list
        // Iterative version
void cleanUp(Link * head)
{
    Link * ptr = head;
    while ( head != nullptr )
    {
        Link * temp = ptr;
        ptr = ptr->next;
        delete temp;
    }
}
```

Add at the head

The next method we need is one that adds a new value to the list. We add it at the head since that is the fastest way to add an item.

There are two cases for adding at the head, first the list is empty and you just create a new link and set the head equal to it.

head -> nullptr;
head -> newLink(value) -> nullptr;

The second is when you already have links in the list. Now you need to create the new link and set it to point to the existing link and then set head to point to the new link.

head -> link1 -> ... -> nullptr;
head -> newLink(value) -> link1 -> ... -> nullptr;

Since they both set next of the newLink to the current value of head, you only need to code a single case:

```
        // add a new link to the start of the list
  Link * addItem(Link * head, int value)
  {
      Link * temp = new Link;
      temp->value = value;
      temp->next = head;
      head = temp;
      return head;
  }
```

Searching

There are two ways that we search an array, depending on whether it is ordered or not. If it is ordered, we can do a binary search. This requires the ability to continually divide the array into smaller segments using indices. Since linked lists do not have indices, we have to use the second method (linear) for searches, whether the list is ordered or not. Thus to do a search, we start at the head, check each link for the desired value and quit when we either find it or we reach the end.

```
  // see if a value is present in the list
  // iterative function
bool isThere((Link * head, int value)
{
        // walk down the list from head until reach terminator
    for(Link * ptr = head; ptr != nullptr; ptr = ptr->next)
            //check each link for the desired value
        if ( ptr->value == value )
                // and return true if it is found
            return true;

        // if you reach the end without finding the value, not there
    return false;
}

  // see if a value is present in the list
  // recursive function
bool isThere)
{
    if ( head == nullptr )
        return false;
    if ( head->value == value )
        return true;
    return isThere(head->next, value);
}
```

Search and Remove

If we want to delete a link containing a value, the simplest way is to find it and delete it when found. The basic search is similar in nature; we simply change the code for what to do when the desired link is found to allow us to delete it. The problem is that when we are searching, we have a pointer to the link we want to check. Unfortunately, when we are looking to delete a link, we need to change the link before it.

head -> link1 -> link2 -> link3 -> link4 -> ... -> nullptr;

In order to remove link3, we need to change the next value for link2.

head -> -> link1 -> link2-> link4 -> ... -> nullptr;

The way we solve this problem is to have a pointer that walks down the list one link before the one that we are looking at. This requires that we special case the head of the list as shown in the following code. Then we walk down the list, at each step we create a pointer to the next link and check it.

```
  // delete a value if present
  // note, we are returning the value of head with a pass by reference
bool deleteItem(Link * &head, int value)
{

        // First case, the list is empty
    if ( head == nullptr )
        return false;

        // Second case, the first link has the desired value
    if ( head->value == value )
    {
        Link * temp = head;
        head = head->next;
        delete temp;
        return true;
    }


        // General case, it is somewhere down the line (or not there at all)
    Link * ptr = head->next;
    Link * prev = head;
    while ( ptr != nullptr )
    {

            // We found it, now delete the link
        if (ptr->value == value)
        {
            prev->next = ptr->next;
            delete ptr;
            return true;
        }


            // Not found yet, increment pointers
        prev = ptr;
        ptr = ptr->next;
    }


        // Reached end of list, go home empty handed
    return false;
}
```

Displaying

In our search method, we used a simple for loop to walk through the list. This same approach can be used to create a string containing the elements of the list. This same approach can be used for any list that has items that can be output with the stream operator <<. Note this for loop uses the fact that **nullptr** is **false** to end.

```
// display list starting at head
std::string displayList(Link * head)
{
        // define a string stream to gather the information
    std::stringstream ss;

        // walk down the list, until we reach terminating nullptr
    for(Link * ptr = head; ptr; ptr = ptr->next)
            // add next value to the stream
        ss << ptr->value << " ";

        // add terminating endline
    ss << "\n";

        // return the string out of the stringstream
    return ss.str();
}
```

Sorted Lists

Although ordered lists do not have the same utility as ordered arrays, since they do not support binary search, there are times that you want to maintain data in an ordered manner. Also, note that inserting an item in an ordered list might take looking at n/2 items, but you can insert the new link in one step – you do not have to move the other items out of the way first.

To maintain a sorted list, all you need to do is to insert a new link just before the first link that is larger (smaller) than it is. The following code will do this for you.

```
// support list sorted smallest to largest
Link * addSorted(Link * head, int value)
{
        // if list is empty, insert at head
    if ( head == nullptr )
    {
        return addHead(head, value);
    }

        // if head is larger, special case
    if ( head->value > value )
    {
        return addHead(head, value);
    }

        // otherwise, walk down the list to find the right place to insert it
        // again, we have to look ahead as we go down the list
    Link * ptr = head;
    while ( ptr->next != nullptr and ptr->next->value < value  )
        ptr = ptr->next

        // ptr now points to the last link <= value
        // so create a new link and put it after ptr
      Link * temp = new Link;
      temp->value = value;
      temp->next = ptr->next;
      ptr->next = temp;

        // done, go away
      return head;
}
```

Sorting a Linked List

Although it is not common to need to sort a linked list, you can sort it. Just remember that this is not like an array where you have access to any value in a single step via its index.

The following code does a selection sort using a recursive method. It does this by swapping values in links instead of changing the order of the links themselves. The smallest value is placed at the head, then the function is called on the remainder of the list to sort it.

```
  // recursive sorting using a selction sort
void sorter(Link * head)
{
        // if the list has data
        // you move the smallest item to the head
    if ( head != nullptr )
    {
        Link * ptr = head;
        Link * smallest = head;
        while ( ptr != nullptr )
        {
            if ( ptr->value < smallest->value )
                smallest = ptr;
            ptr = ptr->next;
        }
            // now you know the smallest one
            // so exchange values with std::swap
        std::swap(head->value, smallest->value);

            // the head is now smallest
            // go on and do the rest of the list
        sorter(head->next);
    }

    // if head is nullptr, you have finished with the list
    // and just return, doing nothing more
}
```