Single Linked Lists

Linked Lists

Linked lists are a basic data structure used to keep track of collections of information similar to arrays. They can be used to store data, or more abstract data types such as stacks, queues, and double-ended queues can be implemented with them.

They are different from arrays in several ways. In an array, the data is stored contiguously in memory, and it is possible to use indexing to access any location in a single step. In linked lists, the data is contained in links that are connected through pointers and not stored contiguously, so it is necessary to walk sequentially down the links from the head, or starting point, to find a given item. The memory in arrays is allocated in a single block when they are created, so when they get full, it is necessary to allocate a new array twice as large and copy the data. Linked lists grow and shrink dynamically as new links are created and added to the linked list or old links are removed.

There are two basic types of linked lists, single-linked lists where there is a forward connection between two adjacent links and double-linked lists with connections both forward and backwards. This document focuses on the simpler version, a single-linked list. It shows how items can be added or removed from the head, or front of the linked list, and how it can be searched to find a given item.

It then shows how to modify the linked list to add items at the tail and how a linked list can be used to implement stacks and queues.

Basic structure

A linked list is a set of links joined using pointers between them. It can be imagined as a chain where each link of the linked list is represented by a single link in the chain.

Imagine building a paper chain. Before the first link is created, there is nothing. The first link is created and held in one hand. Then, a second link is looped through the first, and the first is dropped, no longer relevant. As each subsequent link is added to the front of the chain, the end of chain (which is the first link that was added) gets further away. The only links that are tracked are the new one that is being created and the previous one currently held in hand.

Continuing with the paper chain example, to figure out how many links of each color were in the chain, the only way is to start at the link being held and move through the chain, one link at a time, counting the links of each color, until the end is reached.

Like in the paper chain example, a linked list starts at the head and continues, link by link, to the end. Traversing the linked list can only go in a single direction, from each link to the next, because each link only contains a pointer to the next one.

Link Class

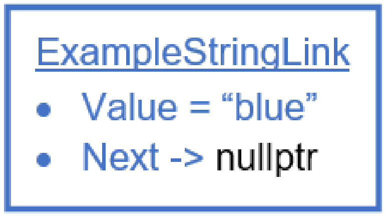
Since a linked list is composed of links, it is necessary to define the structure of a link first. A link is a single object in memory and is defined as an independent class.

A link contains at least two things. One is the value being stored in it (often an object) and the other is a reference or pointer to the next link, normally named `next`. The Link class is not used independently, instead it is only used inside of a LinkedList class.

For a single-linked list, as this document describes, only those two things are necessary. To keep the following coding examples simple, the value stored in the link will be an integer. Remember that while the values are integers in these examples, the link value can be any data type that has been defined.

Note that the following definition is pseudo code that is somewhat C++ based, although pointers are not explicitly defined or used. This documents uses nullptr as a placeholder for the language specific terminator of the list.

```
class Link
private:
   // data stored in the link
   int value

   // reference to next link in the list
   Link next

public:
   // constructor sets value and next
   Link(int value, Link next = nullptr)
      this.value = value
      this.next = next

   // access methods
   int getValue()
      return this.value

   Link getNext()
      return this.next

   // mutator methods
   void setNext(Link next)
      this.next = next
```



Note that link does not have a method to set a new value. The normal convention for linked lists is that values are not changed, the link is instead replaced. The constructor (init function) provides a default value for next so that it does not have to specified when creating a new link.

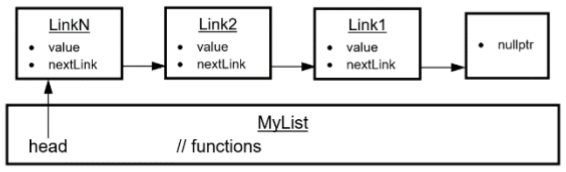
LinkedList Class

By itself, a link is not very useful. It is necessary to create a LinkedList class that includes the methods used to maintain the linked list. The LinkedList class for a single-linked list consists of a variable head that contains the address of the first link, and a series of functions which use head to interact with and access the links in the linked list. This is an important shift from structures that can be accessed via index (such as arrays), where each individual element can be directly accessed. Because of this difference, all accesses to links and their values (other than the first one) require stepping through all intermediate links and are O(N).

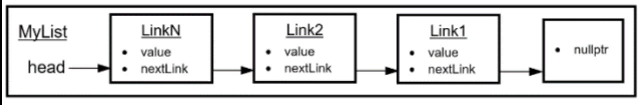
Note that all languages access list elements via pointer, regardless of whether it is done explicitly (C++) or implicitly (Python, C#).

While technically this means that a linked list does not actually contain any of the link elements, conceptually and diagrammatically a linked list is considered to be all of the links accessible via its head.

Technical Representation



Conceptual Representation



For clear communication, this conceptual representation of a linked list will be used throughout this document rather than the technical one.

For a single-linked list with a single end, the only data stored in the LinkedList class is the variable head. The methods this document covers are constructor, destructor, addHead, getHead, removeHead, findValue, and findRemove. In addition, a showList method is defined to allow displaying a logical representation of the linked list. Note that C++ additionally requires a destructor that walks down the list and deletes all the links when you are done.

```
class LinkedList
private:
   // a pointer to the start of the list
   Link head

public:
   // the only class variable is the head and it starts out as nullptr
   LinkedList()

   // add a new link containing value at the head of the list
   void addHead(int value)

   // return the value contained in the first link of the list
   int getHead()

   // remove the first link in the list
   void removeHead()

   // returns true if the list is empty
   bool isEmpty()

   // return true if value is present in the list
   bool findValue(int value)

   // return true if value is present in the list
   // after removing that link from the linkedlist
   bool findRemove(int value)

   // return a representation of the list contents
   string showList()
```

Constructor (init function)

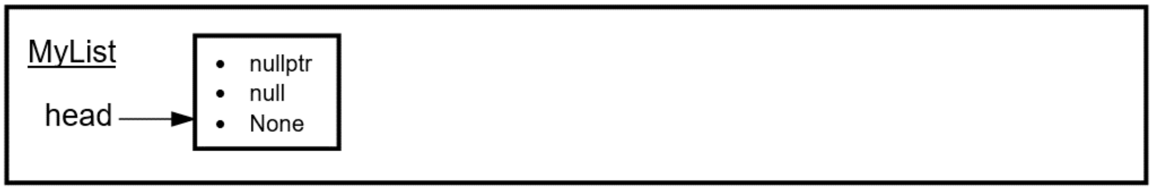
The constructor sets the head to nullptr to initialize it.

```
    LinkedList()
      head = nullptr
```

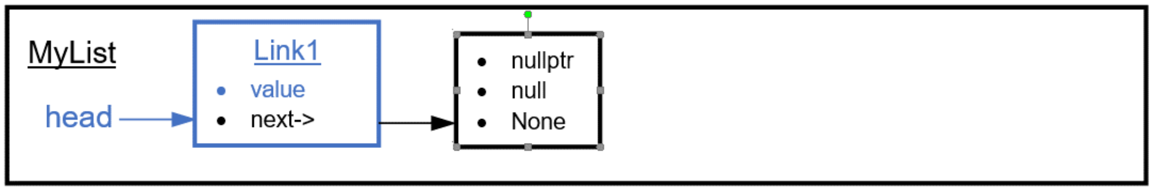
Adding a value at head

The first method to consider is adding a new link to the linked list. Since head is the only variable in the LinkedList class and all new links must be added via it the method is named addHead.

The following diagrams show the two cases to be considered when adding a new link to a linked list. The first is adding a value to an empty list, where head points to nullptr.

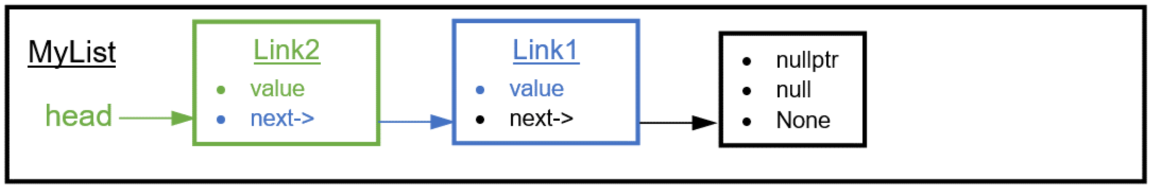


The first link is added by creating a new link, and then head is set to point to it. The Link class defined above automatically sets the link's next value equal to nullptr, so nothing further is required.



To make the connections visually easy to follow, in these diagrams all pointers are colored to match the link (or in the case of nullptr, the lack of a link) that they are currently pointing to.

The second case for adding a link is when a link is added to a non-empty list. Since there is already a link in the list, when the new link is created its next value must be set to point to the link that head points to before head can be set to point to the new link



It is always best practice to consider possible special cases when creating code. In this circumstance it is easily demonstrated that no special case code is required. In the first case, head contains the end of the list indicator (nullptr) and in the second case head contains the address of a link. This allows the code to be reduced to a single case.

Since the same basic action is performed in both cases, the code simply needs to create a new link, set the new link's next value to the value stored in head, and then set head to the new link.

```
void addHead(int value)
    // create a new link containing the value
    Link temp = new Link(value)

    // update its next field to contain whatever head contained
    temp.setNext(head)

    // update head to point to the new link
    head = temp
```

This creates a new link in memory and adds it to the front of the linked list. This is part of the dynamic nature of a linked list, as new links are added the memory usage increases.

Since the size of the linked list, the number of links, does not affect the number of steps in this method, it is O(1).

Accessing the first link

This version of a linked list implements two methods which directly access the link at head. The first returns the value contained in that link and is named getHead.

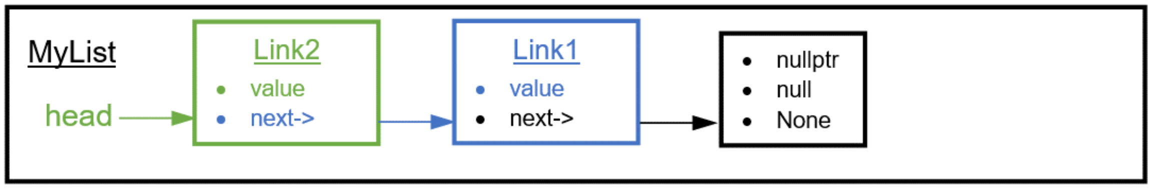
Again, both an empty linked list and a non-empty linked list must be considered. Because this method reaches into a link object to extract a value and there is nothing inside of nullptr that could be extracted, a special case must be handled with when coding this method. The exception empty_list_exception is a placeholder for either a user defined exception or a system generated exception and is not specific to any language. By throwing an exception, the class avoids a fatal error and passes the problem back to the calling program where it needs to be dealt with.

```
int getHead()
    // if list is empty, throw an exception
    if (head == nullptr)
        throw empty_list_exception

    // return the value from inside head
    return head.getValue()
```

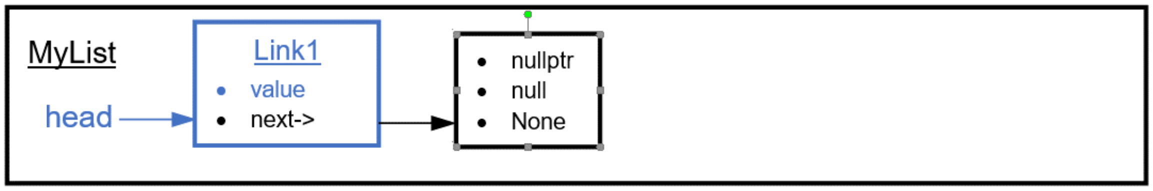
Removing at the head

The second method accessing the linked list from head removes the first link without returning a value. Since it removes the link pointed to by head, it is named removeHead.

Since, it is impossible to remove something from nothing, if the linked list is empty the code should throw an exception. After verifying that there are links to remove, it is good to consider the steps involved in taking a link out. The list may have a single link or multiple links, but the removal process is identical.



In a singly linked list, removing a link is a simpler process than adding a link, insofar as the list itself is concerned. Simply change head to point to the next link in the chain.



As before, it does not matter if it has only a single link or multiple links. The value contained in the variable next in the first link is stored in head, this is either nullptr or the address of the next link in the list.

```
void removeHead()
// if list is empty, throw an exception
if (head == nullptr)
   throw empty_list_exception

// C++ save the link to delete later
Link temp = head

// update head to point at the next link in the list
head = head.getNext()

// C++ delete the old link
delete temp
```

Note that in languages without garbage collection, such as C++, it is additionally necessary to delete the link you are removing to avoid a memory leak. This requires saving the address of the link, updating head to point to the next link, then deleting the link being removed using the saved address.

Since the number of links in the linked list does not affect the number of steps in this method, it is also O(1).

Detecting an empty list

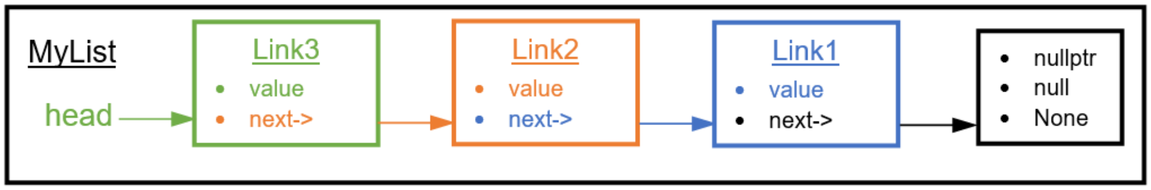
Since code readability is typically desirable, it is easy to replace the test for an empty linked list with a method that communicates the intent of the code more clearly. Here is a simple method that does so.

```
// return true if the list is empty
bool isEmpty()
   return head == nullptr
```

Traversing a linked list

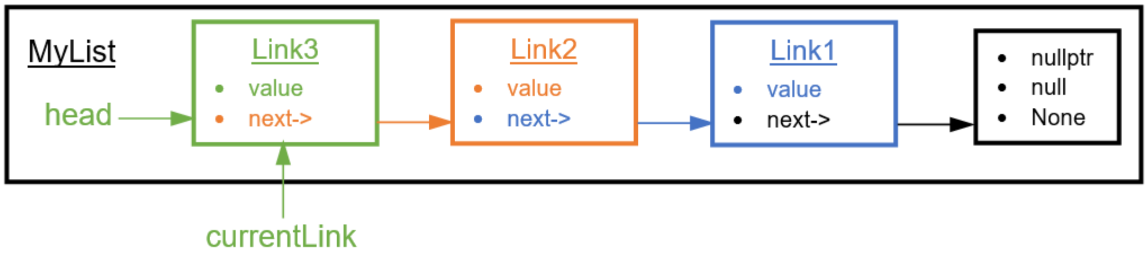
One of the features of an array is the ability to access any value directly via an index. Linked lists have the benefit of automatic resizing as needed, but links are tracked in a relativistic manner instead of via indices and links other than the head require traversing any intervening links.

The following series of images show how links are traversed when walking down a linked list.



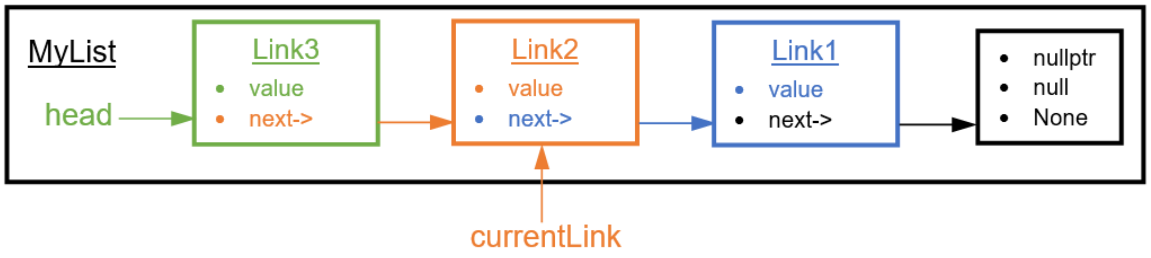
It is necessary to define a variable to keep track of which link is being accessed at a given time. In these diagrams it is labeled currentLink for clarity.

When traversing a linked list, currentLink always starts with the address stored in head. In this example, that means that it starts out pointing to Link3.



Since the only way to access links further away from head is via the pointers between them, moving currentLink to the next link requires setting it equal to the address in the next variable of the link it is currently pointing to. In other words, if currentLink is equal to head and head contains the address for Link3, currentLink also contains the address for Link3. This means that currentLink.getNext() is the same as Link3.getNext(). To move forward one link, use the following statement.

```
currentLink = currentLink.getNext()
```



To continue traversing the list, the same pattern of moving currentLink from the link that it is currently pointing at to point to the next link in the chain is repeated until currentLink reaches nullptr.

Searching

Since linked lists do not have indices, all searches are linear, starting at the head and checking each link for the desired value then quitting when it is found, or the end is reached. This means that all searches on linked lists are O(N).

findValue

A linear search can be done with a simple loop, using a variable that starts at the link pointed to by head and is then updated to walk down the list until the terminating nullptr or the value is found. This example uses a while loop; however it is equally easy to use a for loop or recursion.

```
// Iterative version
bool findValue(int value)
    // variable to track status of find
    bool found = false

    // start at the head
    Link currentLink = head

    // continue until the end
    while (currentLink != nullptr and not found)

        // found it, go home happy
        if (currentLink.getValue() == value)
            found = true

        // not found, continue with next link
        else
            currentLink = currentLink.getNext()

    // end of list or found, return
    return found
```

findRemove

To delete a link containing a value, the simplest way is to find it and delete it. The basic search is similar in nature; simply add code to delete the link when it is found instead of just returning true. The problem is, when searching, currentLink points to the link being examined and, unfortunately, when deleting a link, it is necessary to update the link before it in the linked list.

The way to solve this problem is to have two variables that walk down the linked list in tandem. CurrentLink points to the link to change and nextLink points to the link which is being compared to the search value. Note that it is necessary to create a special case for the situation when the link to remove is the first one in the list.

```
// find and remove a value if present
bool findRemove(int value)

    // special case empty list
    if (isEmpty())
        return false

    // next check head for value
    if (head.getValue() == value)
        removeHead()
        return true
    // variable to track status of find
    bool found = false

    // now walk down list, start by pointing at the head
    Link currentLink = head

    // until done with list or value is found
    while (currentLink.getNext() != nullptr and not found)
        // create a reference to the next link
        Link nextLink = currentLink.getNext()

        // if nextLink is the one wanted
        if (nextLink->getValue() == value)
            // update list to look past it
            currentLink.setNext(nextLink.getNext())

            // C++ delete old link

            // set status as found
            found = true

        // not there, move on down the line
        else
            currentLink = currentLink.getNext()

    // found or at end, return result
    return found
```

Compare this with an array where it is necessary to move all values with larger indexes down one to remove the desired value. In both cases, the search is O(N). In a linked list, it only takes a single step to remove the link and compress the linked list, while in an array it takes O(N) steps to shift the remaining items. So, even though the algorithms are of the same order, the linked list is faster.

Displaying the Linked List

For debugging and understanding purposes, here is a simple function that returns a printable version of the linked list and the values that are stored in it.

```
string showList()
    string buffer = ""

    if (isEmpty())
        buffer = "Empty List"
    else
        buffer = "head -> "

        Link ptr = head
        while (ptr)
            buffer += to_string(ptr.getValue())
            ptr = ptr.getNext()
            buffer += " -> "

        buffer += "nullptr"

    return buffer
```

Destructor

As shown in the technical representation of a linked list earlier, the links in a linked list are not actually contained within the linked list itself. Therefore, in languages without garbage collection deleting the list will not automatically delete the links from it, nor will deleting head. Instead, it is necessary to code a destructor that walks down the list and deletes all the links.

```
virtual ~LinkedList()
    // while the list is not empty, remove the head
    while (!isEmpty())
        removeHead()
```

Adding to Tail

A useful extension to this class is to add a variable, tail, that points to the last link in the linked list. This allows adding a value at the end without walking down the entire linked list to get there.

Note that four methods need to be modified to do this; the constructor to set the new tail variable to nullptr, addHead to set tail when adding to an empty linked list, removeHead to also update tail when removing the last item from a linked list, findRemove to head with possibly deleting the tail link, and the new method addTail . Note that the following methods have additionally been updated to use the isEmpty method in place of testing head == nullptr.

```
LinkedList()
    head = nullptr
    tail = nullptr

void addHead(int value)

    // create a new link containing the value
    Link temp = new Link(value)

    // if the list was empty
    if (isEmpty())
        // set tail to point to singleton link
        tail = temp
    else
        // update its next field to contain whatever head contained
        temp.setNext(head)

    // update head to point to the new link
    head = temp

void removeHead()
    // if list is empty, throw an exception
    if (isEmpty())
        throw empty_list_exception

    // update head to point at the next link in the list
    head = head.getNext()

    // if now empty, set tail properly
    if (isEmpty())
        tail = nullptr

bool findRemove(int value)
```

```
    // special case empty list
    if (isEmpty())
        return false

    // next check head for the value
    if (head.getValue() == value)
        removeHead()
        return true

    // variable to track status of find
    bool found = false

    // now walk down the list, starting at the head
    Link curLink = head

    // until done or value is found
    while (curLink.getNext() != nullptr and not found)

        // create a reference to the next link
        Link nextLink = curLink.getNext()

        // if nextlink is the one we want
        if (nextLink.getValue() == value)

            // update list to point past it
            curLink.setNext(nextLink.getNext())

            // added code to deal with deleting tail
            if (nextLink == tail)
                tail = curLink

            // C++ free up memory as needed
            delete nextLink

            // set status as found
            found = true

        // not there, move on down the line
        else
            curLink = nextLink;

    // found or at end, return result
    return found

void addTail(int value)
    // create a new link containing value
    Link * temp = new Link(value)

    // if list was empty
    if (isEmpty())
        // set head to point to singleton link
        head = temp
    else
        // update last link to point to new link
        tail.setNext(temp)

    // update tail to point to node
    tail = temp
```

FIFO and LIFO

Two commonly used data structures are a stack (LIFO) and a queue (FIFO). In a stack, items are added in such a way that only the most recently added item can be retrieved, then the previously added one, and so forth. Imagine a stack of plates where plates can be added to the top of the stack or removed from the top, but not inserted in the middle. In a queue, items are added at the end of the queue and removed from the front, for example the line of people waiting to check out in a store.

With a double-ended, single-linked list as described here, it is trivial to implement either of these two classes using class composition. Notice that the only difference between them is where new items are added. The Stack adds them to the head and the Queue adds them to the tail.

```
class myStack
private:
   // This stack uses class composition by including a linked list
   LinkedList theList

public:
   // the stack hides the linked list methods behind its own
   void push(int value)
      theList.addHead(value)
   int peek()
      return theList.getHead()
   void pop()
      theList.removeHead()

class myQueue
private:
   // This queue uses class composition by including a linked list
   LinkedList theList

public:
   // the queue hides the linked list methods behind its own
   void addLast(int value)
      theList.addTail(value)
   int getFirst()
      return theList.getHead()
   void removeFirst()
      theList.removeHead()
```