Double Linked Lists

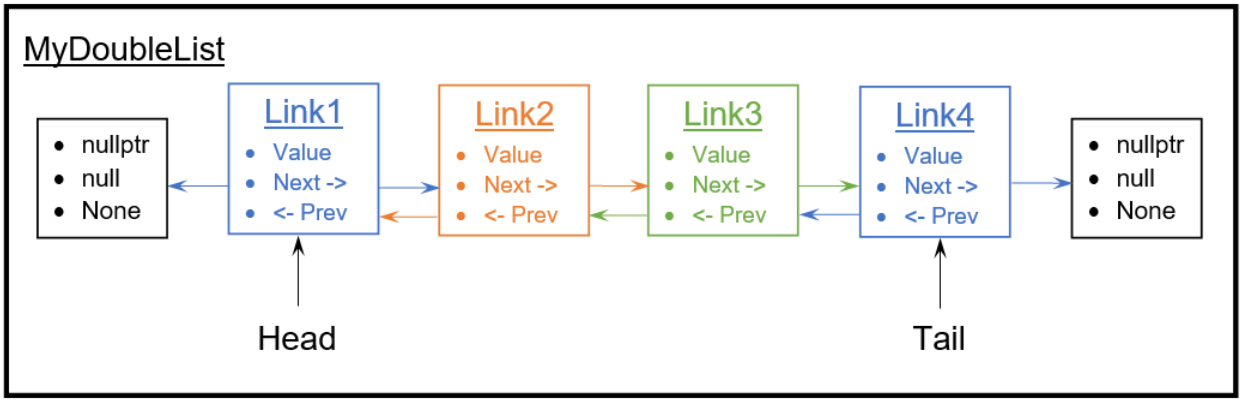
Double Linked Lists

Adding a second variable to each link, so the link references both the previous and next links in the list, creates a double linked list instead of a single linked list. Double linked lists allow moving in both directions in the list. They also support removing items from the tail in a single step, so they readily support a deque.

This document will cover the changes that must be made to a single linked list to make it a double linked list. It will assume the single linked list that already has variables allowing direct access to both the head and tail of the list.

Basic structure

A double linked list is similar to a single linked list; it just adds links to previous as well as next. Like a single linked list, it has variables for head and tail that refer to the first and last links of the list respectively.



Link Definition

A double linked list adds a reference to previous to the single linked list's link. That is, it has a variable for the value being stored and two references, next and previous, for accessing the next and previous links in the list. This document uses a pseudocode that is somewhat C++ in style. Also, nullptr is used as a placeholder for the class specific list terminator (nullptr, null, or None).

```
class Link
    // data contained in link
    int value

    // reference to the next link in the list
    Link next

    // reference to the previous link in the list
    Link prev

  public:
    // constructor
      Link(int value, Link next = nullptr, Link prev = nullptr)
        this->value = value
        this->next = next
        this->prev = prev
```

```
    // access methods
int getValue()
    return this->value

Link getNext()
    return this->next

Link getPrev()
    return this->prev

void setNext(Link next)
    this->next = next

void setPrev(Link prev)
    this->prev = prev
```

List Definition

In addition to the methods described previously for a single linked list, there is an added method, removeTail. Adding previous also requires modifying some single linked list methods.

```
class List
    // a reference to the start of the list
    Link head
    // a reference to the end of the list
    Link tail

  public:
    // constructor initializes the head and tail to nullptr
    List()

     // destructor for C++, deletes any remaining links
    ~List()

    // add a new link to the start of the list
    void addHead(int value)

    // add a new link to the end of the list
    void addTail(int value)

    // return the value from the start of the list
    // Throw an exception on an empty list
    int getHead()

    // return the value from the end of the list
    // Throw an exception on an empty list
    int getTail()

    // remove a link from the start of the list
    // Throw an exception on an empty list
    void removeHead()

    // remove a link from the end of the list
    // Throw an exception on an empty list
    void removeTail()

    // see if a value is present in the list
    bool findValue(int value)
```

```
// delete a value if present
bool findRemove(int value)
```

Adding to head and tail

The methods for adding to head and tail need to be updated to deal with having a reference to the previous link as well as a reference to the next one. This actually makes the code more symmetrical.

```
// add a new link to the start of the list
// take advantage of constructor for link
void addHead(int value)

    // if the list is empty, initialize both head and tail
    if (head == nullptr)
        head = tail = new Link(value)

    // otherwise, just add a new link at the head
    else
        // create new link with nullptr for prev and head for next
        Link temp = new Link(value, head, nullptr)

        // set first link and head to point to the new link
        head.setPrev(temp)
        head = temp
```

Adding a new link to the end of the list is similar to adding one to the head.

```
// add a new link to the end of the list
// take advantage of constructor for link
void addTail(int value)

    // if the list is empty, initialize both head and tail
    if (tail == nullptr)
        head = tail = new Link(value)

    // otherwise, just add a new link at the tail
    else
        // create the new link with nullptr for next and tail for prev
        Link temp = new Link(value, nullptr, tail)

        // set last link and tail to point to the new link
        tail.setNext(temp)
        tail = temp
```

Accessing the first and last links

This version of a linked list implements two methods which directly access the link at head or tail. The first returns the value contained in that link and is named getHead or getTail. The second removes the relevant link and is named removeHead or removeTail.

The methods to get a value are identical for single and double linked lists, so are not shown here. For these, refer back to the single linked list document.

The only change from the single linked version of removeHeHead is to deal with the references to the previous links. Again, 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.

```
// remove a link from the start of the list
// Throw an exception on an empty list
// take care of setting tail when list becomes empty
int removeHead()

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

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

    // update head
    head = head.getNext()

     // if list is now empty, update tail
    if (head == nullptr)
        tail = nullptr

    // otherwise, update prev on new first link
    else
        head.setPrev(nullptr)

    // delete the old link (C++ only)
    delete temp
```

The new method for removeTail is again similar.

```
// remove a link from the end of the list
// Throw an exception on an empty list
// take care of setting head when list becomes empty
int removeTail()

    // if list is empty, throw an exception
    if (tail == nullptr)
        throw empty_list_exception

    // save the link to delete later (C++ only)
    Link temp = tail

    // update tail
    tail = tail.getPrev()

    // if list is now empty, update head
    if (tail == nullptr)
        head = nullptr

    // otherwise, update next on new last
    else
        tail.setNext(nullptr)

    // delete the old link (C++ only)
    delete temp
```

Searching

There is no change to the find from that used for a single linked list, which only used the value and the next variables of the link.

Find and Remove

Removing an item in the middle of list is simpler than with a single linked list. Rather than having to track the current location while testing the next one, this version simply tracks the current location and then removes the link when found.

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

    // walk down the list, looking for the value
    // start at the head
    Link ptr = head

    // continue until we run out of links
    while (ptr != nullptr)

        // see if this link is what we want
        if (ptr.getValue() == value)

            // special case head use existing code
            if (ptr == head)
                removeHead()
                return true

                // special case tail use existing code
            if (ptr == tail)
                removeTail()
                return true

            // typical link, set prev and next to point around it
            ptr.getPrev().setNext(ptr.getNext())
            ptr.getNext().setPrev(ptr.getPrev())

            // delete the old link (C++ only)
            delete temp

            // and return
            return true

        // not there, keep looking
        else
            ptr = ptr.getNext()


    // done with list without finding it
    return false
```

Double Ended Queue

A double linked, double ended list can still implement a stack or queue, just as a single linked list can. In addition, since it can add and remove from both the head and tail, it is trivial to implement a double ended queue. The exact algorithm is left as an exercise for the student.

Most standard libraries implement a double ended queue, along with a FIFO and LIFO using a double ended linked list. The Python list is also implemented this way. All that is needed is to optimize the underlying structure and the higher-level abstract structures are available free.