CS 1704

Introduction to Data Structures and Software Engineering

Linked Lists Basics

- Arrays have so many drawbacks:
  - Wasted space
  - Wasted time
  - Clunky for insertion or deletion in the middle
  - etc.

Need a better solution
Linked-List

Idea:
- Information is stored in nodes
- Each node points to the node directly following it
- To find, insert or delete you simply manipulate nodes

Trade-offs

- Linked lists allow for random insertion and deletion
- Search, however, could take searching the entire list
- If an array is ordered, you could search less than half and find what you are seeking
- Linked data structures in general allow greater flexibility

Specifics

- A linked list contains a head node, a tail node and a current node.
- These are used to keep track of the data structure
- A linked-list class would make these private data.
- A linked list should have several different types of insertion and deletion.
Insertion/Deletion

- You can have an insert before the head node, an insert after the tail node, an insert before the current node and an insert after the current node.
- You can think up many more, but should do at least insert before and after the current node.
- You should do a remove current node.

Find

- Find simply walks down the list of nodes, starting at the head node looking for a particular piece of data.
- When it finds it, it stops and the current node pointer should be pointing to that node.

More Specifics

- The head, tail and current node pointers are dynamically allocated pointers to a node class.
- The magic word: dynamic.
- That means the linked list class needs to implement a copy constructor, assignment operator and a destructor in addition to any other methods.

Start of a Linked List class

```cpp
class LinkedList
{
    public:
        //stuff
    private:
        Node *Head;
        Node *Tail;
        Node *Current;
};
```

Question

- What is Node?
  - Node is a class
  - Node holds or points to the data.
  - Nodes also point to at least the next node in the chain.

Here's a node...let's take a look inside!!
Specifics
- Nodes should have at least two private pieces of data
  - A place to store the data
  - A pointer to the next node in the chain
- The should support various setting and getting methods of both the next pointer and the data

Node Class start
```cpp
class Node {
    public:
        // stuff
    private:
        Node* Next;
        Record* R;
};
```

Issues
- Insertion
  - Insert after current, after tail, and before head is more or less straightforward
  - You create a new node
  - Have the new node point to what you used to point to
  - Have the node you are sitting in point to the node
    - In the case of the insert before head, redirect the head pointer to point to the new head
    - In the case of insert after tail, redirect tail to point to new node
    - In any case of current, need to check to see if the current is the head or tail and take appropriate actions
More Insertion Issues
- Insert before current raises a problem
- How do you get to the node before you?
- Solution, create a temporary node pointer and walk down from head and look ahead to see if the next node is the current node.
- When you see the current node ahead of you, stop and insert after the node the temporary pointer is pointing to.
- Again taking into account if the temporary node is the head or the tail

Deletion Issues
- Same problem as insert before current
- How do you find the node that is in front of you to be able to hook up the chain again?
- Use same solution.
- May want to write a private helper function that finds the node before the current node.
- You can call it whenever you need.
- It could return a pointer to you.

Alternative
- Create a doubly linked-list.
- Each node not only points to the next node in the chain, but also the node before it in the chain.
- This simplifies the problem of how do you get the node in front of you.
- It complicates insertion and deletion because now you have more pointers to keep track of.
Forward Declarations

We can simply make a forward declaration of Node, prior to defining the Node type:

typedef int Item;
struct Node;         // forward declaration
typedef Node* NodePtr;
struct Node {        
    Item     Element;
    NodePtr Next;
};
NodePtr Head = NULL;  // head pointer for list

Node Class

class Node
{
    public:
        Node();
        Node( Record *R );
        Node( const Node& RHS); //Copy const
        const Node& operator=( const Node& RHS);
        ~Node();

    private:
};

Node Implementation

Node::Node()  
{  
    Next = 0;
    recordPtr = 0;
}

Node::Node( Record *R )  
{  
    Next = 0;
    recordPtr = R;
}

Destructor

Node::~Node()  
{  
    delete recordPtr;
    delete Next;
}

assignment Operator

const Node& Node::operator=( const Node& RHS )  
{  
    if ( this != &RHS )
    {
        delete Next;
        delete recordPtr;
        Next = 0;
        recordPtr = new (nothrow) Record("RHS.recordPtr");
    }
    return *this;
}
getNext/setNext

Node * Node::getNext()
{
    return temp;
}
void Node::setNext( Node* N )
{
    Next = N;
}

data/getData

Record Node::getData()
{
    Record R = *recordPtr;
    return R;
}
void Node::setData( Record *R )
{
    recordPtr = R;
}

Some more implementation

class LinkedList
{
    public:
        LinkedList();
        LinkedList( const LinkedList& );
        LinkedList( const LinkedList& );
        ~LinkedList();
        bool insertAfterCurrent( Record );
        bool insertBeforeCurrent( Record );

private:
    Node* findNodeBeforeCurr( );
    Node* Head;
    Node* Tail;
    Node* Current;
};

More Implementation

void gotoHead();
void gotoTail();
bool find( Record ) const;
bool deleteCurrent( );
void print( ostream& ) const;
void clear( );

More Implementation

private:
    Node* findNodeBeforeCurr( );

    Node* Head;
    Node* Tail;
    Node* Current;
};

Constructor

LinkedList::LinkedList()
{
    Head = Tail = Current = 0;
}
Destructor

```cpp
Destroyed::LinkedList()
{
    Node* temp = Head;
    Current = Head;
    while ( Current != 0 )
    {
        Current = Current->getNext();
        delete Temp;
        Temp = Current;
    }
    Head = Tail = Current = 0;
}
```

insertAfterCurrent

```cpp
bool LinkedList::insertAfterCurrent( Record R )
{
    bool success = true;
    Node* newNode = new (nothrow) Node( R );
    if ( !newNode )
    {
        success = false;
        else if ( Head == 0 )
        {
            Head = Tail = Current = newNode;
        }
        else
        {
            newNode->setNext( Current->getNext() );
            Current->setNext( newNode);
        }
    }
    return success;
}
```

deleteCurrent

```cpp
bool LinkedList::deleteCurrent()
{
    bool success = true;
    if ( Current != Head && Current != Tail )
    {
        Node* prior = findNodeBeforeCurrent();
        prior->setNext( Current->getNext() );
        delete Current;
        Current = prior;
    }
    return success;
}
```

findNodeBeforeCurrent

```cpp
Node* LinkedList::findNodeBeforeCurrent()
{
    Node* temp = Head;
    if ( Head == Current )
        temp = 0;
    else
    {
        while ( Temp->getNext() != Current )
            Temp = Temp->getNext();
    }
    return temp;
}
```

copy constructor

```cpp
LinkedList::LinkedList( const LinkedList& LL )
{
    Head = new Node( new Record( LL.Head->getData() ) );
    Current = Tail = Head = 0;
    Node* temp = LL.Head;
    while ( temp != 0 )
    {
        insertAfterCurrent( LL.Current->getData() );
        temp = temp->getNext();
    }
    //Is this all?
}
```

```cpp
LinkedList::LinkedList( const LinkedList& LL )
{
    Head = new Node( new Record( LL.Head->getData() ) );
    Current = Tail = Head = 0;
    Node* temp = LL.Head;
    while ( temp != 0 )
    {
        insertAfterCurrent( LL.Current->getData() );
        temp = temp->getNext();
    }
    //Is this all?
    //check to see if LL is empty, current should be at the
    same relative position, set tail
} //check to see if LL is empty, current should be at the
```
Other Considerations

- What about a Merge Method?
  - Merges two Linked lists