Farhan Zaman 10/6/18
CS21200 10:00AM

HW 3 Report

This homework was to teach us about linked lists, and how they can be more effective with massive data storage, and how to save memory, and do a multitude of changes that can happen in real life. We get a better idea of memory allocation and pointers in general.

For question 1 I wrote:

```
node::node(const nodeDatatype& init_data, node* init_next)
{
	data=init_data;
	next=init_next;
}

void node::setdata(const nodeDatatype& init_data)
{
	data=init_data;
}

void node::setnext(node* init_next)
{
	next=init_next;
}

nodeDatatype node::getdata() const
{
	return data;
}
const node* node::getnext() const
{
	return next;
}

node* node::getnext()
{
	return next;
}
```

These are all the basic set and get functions, to access the 2 private members, data and next, along with one const getnext function to add reliability if we ever need to keep it constant. The precondition of these functions don't exist. The post condition for node constructor is that a node is created with either data 0 and next=NULL, or it has user inputted values and links. Post

condition for set data is just that data private member is assigned the user inputted value. The post condition is similar for setnext. For get next and get data, the post condition is that the respective private member value is returned. There is no worst time complexity, its just O(1).

For question 2a I wrote:

```
void insertfront(node* &head, nodeDatatype data)
{
	head=new node(data,head);
}
```

The insertfront function takes user inputter data, and uses a constructor to set the link from the new node to NULL, which is what head was pointing to before, and now head is pointing to new node. The overloaded insertfront function, takes a node as its argument, and calls the other function to perform the same action. The precondition is that head is pointing to a list. The post condition is that a new node is added to the front of my list and now head points to the new node and the list gets one node longer. Worst time complexity is just O(1), since it's just one action.

For question 2b I wrote:

```
void deletefront(node* &head)
{
	node* curr;
	curr=head;
	head=head->getnext();
	delete curr;
}
```

The deletefront function takes a head point to a list, and then deletes the first node of the list, by using a temporary variable and sets it to head which is pointing to the first node and then deletes it, after making head point to the 2nd thing in the list if there is one. The precondition of this would be head points to a list, regardless whether or not it's empty or not this works. The post condition is the first node is deleted, and the list is 1 node shorter. The worst case time complexity is O(1), since constant amount of operators occur.

For question 2c I wrote:

```
void insertend(node*& head, nodeDatatype data)
{
	if(head==NULL)
	{
		insertfront(head,data);
```

```
		return;
	}
	node* place=new node(data);
	node*curr;
	curr=head;
	while(curr->getnext()!=NULL)
	{
		curr=curr->getnext();
	}
	curr->setnext(place);
}
```

The insertend function inputs a node at the end of the linked list, but it first checks if the head is equal to NULL, since then the insertfront can just be called to do the same thing. If the list isn't empty then we use a loop to get the last node and then set its link to a node with the user inputted data. We know curr is the last node, since the loop stops it when the next one is NULL. The precondition is that head points to a linked list, and the postcondition is that the list gets 1 longer, and a new node is added to the end. The worst case time complexity is O(n), which is for the while loop if the size of the list was just n.

For question 2d I wrote:

```
void deleteend(node* &head)
{
	node* curr, *prev;
	curr=head;
	if(head==NULL)
	{
		cout<<"Empty list"<<endl;
	}
	else if(head->getnext()==NULL)
	{
		delete head;
	}
	else
	{
		while(curr->getnext()!=NULL)
		{
			prev=curr;
			curr=curr->getnext();
		}
		prev->setnext(NULL);
		delete curr;
	}
}
```

For the deleteend function, it deletes the last node by first checking if the list is empty and if it is to do nothing since there is nothing to delete. If the list only has 1 thing to delete whatever the headpoint is pointing to, and if the list is longer than 1 node, to use our while loop and reach the last node, and have a previous tracker, so when current is the last node, we have the 2nd to last node as well, which we use to point to NULL, and then the memory for current is deleted. The precondition is that head points to a linked list and the postcondition is that the last node is deleted if applicable. The worst case time complexity is O(n), which is for the while loop and a list of size n.

For question 2e I wrote:

```
int length(const node* head)
{
	int count=0;
	const node* curr;
	curr=head;
	while(curr!=NULL)
	{
		curr=curr->getnext();
		count++;
	}
	return count;
}
```

For the length function, it just returns the length of the linked list, by using a counter and running a temp variable curr through a while loop until it reaches NULL, which means it is at the last node. The precondition is that head points to a linked list, regardless if it is empty or not this works. The post condition, is the length of the list is returned. The worst case time complexity would be O(n), which is for the while loop and the size of the list n.

For question 2f I wrote:

```
void deleteall(node* &head)
{
	while(head!=NULL)
	{
		deletefront(head);
	}
}
```

The deleteall function deletes the entirety of a list, by just calling the deletefront function in a while loop, so that once head=NULL, we know no nodes remain in the list. The precondition is

that head points to a list. This works regardless of the list is empty or not, and the post condition is that head points to NULL, and the list is empty and has no length and there is no more memory allocation for it. The worst case time complexity is O(n), for the size n of the linked list inside of the while loop.

For question 2g I wrote:

```
void print(node* &head)
{
	node* curr;
	curr=head;
	while(curr!=NULL)
	{
		cout<<curr->getdata()<<endl;
		curr=curr->getnext();
	}
}
```

The print function, prints the data of every node in the list sequentially to the user's screen. It does this by have a temporary curr variable iterate forward until NULL and cout the data at every step, and this works even if the list is empty, and it is important to cout first then iterate so the first thing curr points to is shown and not skipped. The precondition is that head points to a list, and the postcondition is that the list is printed to the user's screen. The worst case time complexity is O(n), since the length of the list is n and it's in a while loop.

For question 2h I wrote:

```
void insertpos(node* &head, nodeDatatype data, int position)
{
	assert(head!=NULL && position>0);
	if(position==1)
	{
		insertfront(head,data);
		return;
	}

	node* curr=head;
	node* prev=head;
	int count=0;
	prev=head;
	while(count!=position-1)
	{
		prev=curr;
		curr=curr->getnext();
```

```
            count++;
        }
        node* place=new node(data);
        prev->setnext(place);
        prev=prev->getnext();
        prev->setnext(curr);
}
```

The insertpos function inserts a node at a position from user input. It does this by first checking that the list isn't empty and that position is a legal place. If the position is the first one, we use insert front, if not we have a tracker of curr and prev, where we use our loop and use count to make sure our curr one is on the position, and our prev is on the one before, so we can appropriately set the links The precondition is that head points to a list, and that position is less than equal to size of list, which I forgot to include in the assert statement. The postcondition is the list is 1 node longer and the newnode with user inputted data is not at position i. The worst case time complexity is O(n), for the while loop and size n list.

For question 2i I wrote;

```
void deletepos(node* &head, int position)
{
        assert(head!=NULL && position>0);
        if(position==1)
        {
            deletefront(head);
            return;
        }

        node* curr, *prev;
        curr=head;
        int count=0;
        while(count!=position-1)
        {
            prev=curr;
            curr=curr->getnext();
            count++;
        }
        prev->setnext(curr->getnext());
        delete curr;
}
```

The delete post function deletes a node at position i, and similiary to insertpos, it checks that the position is legal and the list isn't empty, and I also forgot to check that positon is less equal to the size of the list. It has 2 trackers to find the position, and then has prev link to the one after curr,

and just delete curr. The precondition is that the list points to a list, and that position is less than size of the list. The worst case time complexity is O(n), which is for the while loop and list size n.

For question 2j I wrote:

```
void insertfront(node* &head, node* place)
{
	insertfront(head, place->getdata());
}

void insertend(node* &head, node* place)
{
	insertend(head, place->getdata());
}
void insertpos(node* &head, node* place, int position)
{
	insertpos(head,place->getdata(),position);
}
```

These are the overloaded insert functions which use the original insert functions, but convert the node parameter, into data by using the member function. It is a very slick and time efficient method. The precondition for all of these is that both nodes aren't NULL, and that head points to a list. The postcondition is the save as the original insert functions. And the worst case time complexity is the same as the original insert functions.

.

For question 2k I wrote:

```
node* locate(node* head, int position)
{
	assert(position<=length(head));
	node* curr;
	curr=head;
	int count=0;
	while(count!=position-1)
	{
		curr=curr->getnext();
		count++;
	}

	return curr;
}

const node* locate(const node* head, const int position)
```

```
{
	assert(position<=length(head));
	const node* curr;
	curr=head;
	int count=0;
	while(count!=position-1)
	{
		curr=curr->getnext();
		count++;
	}

	return curr;
}
```

The locate function, finds the node as user inputted position and returns the node or shows its data for it to make sense to the user. It does this by having the curr tracker go through the list, position-1 times which we keep track of by using a count, and iterate count in the loop. Then we just return curr. The precondition is that head isn't empty, and even if it is, NULL would be returned and position is bigger than 0 and smaller than length. The postcondition is that the position they asked over is located and returned or it isn't and NULL is returned. The constant version of the function does the same, but it is there for reliability purposes.The worst case time complexity is O(n), for the user inputted position.

For question 2l I wrote:

```
node* search(node* head, nodeDatatype data)
{
	assert(head!=NULL);
	node* curr;
	curr=head;
	while(curr->getnext()!=NULL)
	{
		if(curr->getdata()==data)
		{
			return curr;
		}
		else
		{
			curr=curr->getnext();
		}
	}
	return NULL;
}

const node* search(const node* head, const nodeDatatype data)
```

```
{
	assert(head!=NULL);
	const node* curr;
	curr=head;
	while(curr->getnext()!=NULL)
	{
		if(curr->getdata()==data)
		{
			return curr;
		}
		else
		{
			curr=curr->getnext();
		}
	}
	return NULL;
}
```

The search function searches if the user inputted data is anywhere in the list. It does this to a similar manner of locate, but this iterates through the whole list to check everything, and if found it returns the node, if not NULL is returned. The precondition is that head points to a list and that the list isn't empty. The postcondition is that either a node is returned because it was found or NULL was because it wasn't. The worst case time complexity would be O(n) for the size of the list and the while loop. The constant version is the same thing and is here for reliability purposes.

For question 2m I wrote:

```
bool cycle(node* &head)
{
	if(head==NULL || head->getnext()==NULL || head->getnext()->getnext()==NULL)
	{
		return false;
	}
	node* prev=head;
	node* curr=head->getnext();
	curr=curr->getnext();
	while(curr!=NULL && prev!=NULL)
	{
		if(curr==prev)
		{
			return true;
		}
		curr=curr->getnext();
		if(curr==NULL)
		{
			return false;
```

```
            }
            curr=curr->getnext();
            prev=prev->getnext();
      }
      return false;
}
```

The cycle function checks is there is a loop going on in the linked list, and does this by 1 seeing if there is no NULL is the list, and 2 if our 2 trackers equal each other. We have one tracker moving up one at a time, anda second tracker moving two up at a time, and sooner or later they will reach if there is a cycle or one reaches NULL. The precondition is that head points to a list. We use assert to make sure it isn't empty, or the next one isn't empty or the next one isn't empty, which could have just as easily been a precondition. The post condition is that a truth value is returned depending on is the cycle was found or not. The worst time case complexity is O(n), for the size of the list in the while loop.

For question 2n I wrote:

```
void swapadj(node* &head, int position)
{
      int len=length(head);
      assert(position+1<len && len>2);
      assert(head!=NULL && head->getnext()!=NULL);

      if(position=1)
      {
            node* curr=head->getnext();
            head->setnext(curr->getnext());
            insertfront(head,curr->getdata());
            delete curr;
            return;
      }
      node* prev=locate(head,position);
      node* place=prev->getnext();
      if(prev->getnext()==NULL || place->getnext()==NULL)
      {
            return;
      }

      node* after=place->getnext();
      node* temp=place;
      prev->setnext(after);
      place->setnext(after->getnext());
      after->setnext(temp);
}
```

The swapadj function swaps 2 adjacent nodes, i and i+1, where i is given by user input. It does this by checking if the position is just one, which requires a different algorithm since we don't have a previous point to head. If the list is bigger than 1, we use the locate function to move our pointer to the position and then we set up the link so i -1 points to i+1, and then i+1 points to i and i points to i+2, which can be done with 2-4 trackers. The precondition is that head points to a list, the list isn't empty. The postcondition is that the 2 adjacent nodes are swapped not the data in them. The worst case time complexity is O(n) for the length of the list.

For question 2o I wrote:

```
void swap(node* &head, int position1, int position2)
{

	if(position1==position2)
	{
		return;
	}
	if(position1=position2+1)
	{
		swapadj(head, position2);
		return;
	}
	if(position2=position1+1)
	{
		swapadj(head,position1);
		return;
	}
	assert(position1<length(head) && position2<length(head));

	node* prev1, *curr1, *prev2, *curr2,*after2;
	int count1=0,count2=0;
	curr1=head;
	curr2=head;
	while(prev1->getnext()!=NULL && count1!=position1)
	{
		prev1=curr1;
		curr1=curr1->getnext();
		count1++;
	}

	while(prev2->getnext()!=NULL && count2!=position2)
	{
		prev2=curr2;
		curr2=curr2->getnext();
		count2++;
```

```
        }


        after2=curr2->getnext();
        prev1->setnext(curr2);
        curr2->setnext(curr1->getnext());
        curr1->setnext(after2);
        prev2->setnext(curr1);

        node* temp=curr2->getnext();
        curr2->setnext(curr1->getnext());
        curr1->setnext(temp);

}
```

The swap function swaps any 2 position nodes from user input. It does this in a similar manner but used 5-6 trackers of i-1, i, i+1, j-1, j, ,j+1, where the link changes so its i-1 pointing to j, j pointing to i+1, and j-1 points to i and i pointing to j+1. We check cases of the value of the positions to see if we can use swapadj, and if not we use 2 for loops so we have i-1,i, and i+1 is accessed from i->getnext(), and then have j-1,j j+1 from a similar manner and set the links as described before. The precondition is that the positions are on the list, the list isn't empty, and the head points to a list. The postcondition is the position i and j node are swapped in their links, not the data. The worst case time complexity is O(n), for the size of the list.

For question 2p I wrote:

```
void reverse(node*& head)
{
        node* reverse=NULL;
        node* curr;
        curr=head;
        while(curr!=NULL)
        {
                insertfront(reverse,curr->getdata());
                curr=curr->getnext();
        }
        head=reverse;
}
```

The reverse function reverses the whole list, by inserting the data to the front of a new list, and just having head point to the new head pointer so now head points to this new reversed list. This isn't efficient since we have memory allocation from the original list not deleted. The precondition is that head points to a list. The postcondition is that the list is reversed . The worst case time complexity is O(n), which is for the length of the list.

There are improvements to be made, such as having a tail pointer so we always have access to it, or just using a doubly list since it have more features for us to exploit. Some of the means of my functions were slick methods, or brute force methods. The time complexity for everything is O(n), which is due to the constant usage of the while loop to access positions to do something, when if we had a tail pointer or a doubly list, we could always half the loop time, or choose the loop, whether we start from head or tail or reach our destination and use our functions.

```
#ifndef __ZAMAN_FARHAN_HW3__H__
#define __ZAMAN_FARHAN_HW3__H__
#include <iostream>
#include <cstdlib>
#include <cassert>

using namespace std;
typedef int nodeDatatype;

class node
{
	public:

		node(const nodeDatatype& init_data=nodeDatatype(), node*
init_next=NULL);
		void setdata(const nodeDatatype& init_data);
		void setnext(node* init_next);
		nodeDatatype getdata() const;
		const node* getnext() const;
		node* getnext();

	private:

		node* next;
		nodeDatatype data;
};

void insertfront(node* &head, node* place);
void insertfront(node* &head, nodeDatatype data);
void deletefront(node* &head);
void insertend(node* &head, node* place);
void insertend(node* &head, nodeDatatype data);
void deleteend(node* &head);
int length(const node* head);
void deleteall(node* &head);
void print(node* &head);
void insertpos(node* &head, node* place, int position);
void insertpos(node* &head, nodeDatatype data, int position);
```

```
void deletepos(node* &head, int position);
node* locate(node* head, int position);
const node* locate(const node* head, const int position);
node* search(node* head, nodeDatatype data);
const node* search(const node* head, const nodeDatatype data);
bool cycle(node* &head);
void swapadj(node* &head, int position);
void swap(node* &head, int position1, int position2);
void reverse(node* &head);



#endif
```