Linked List

CSE 4202 - Structured Programming II

Submitted by

Sadman Shaharier Mahim

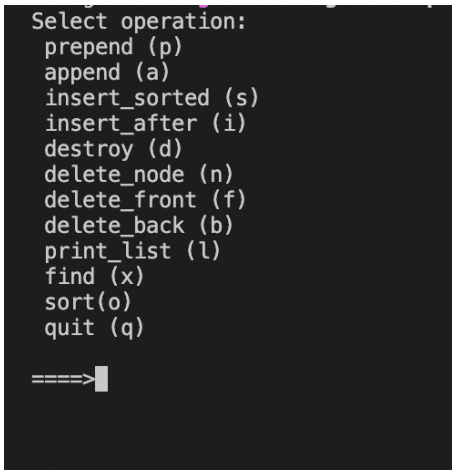
ID 220041133

Section 1

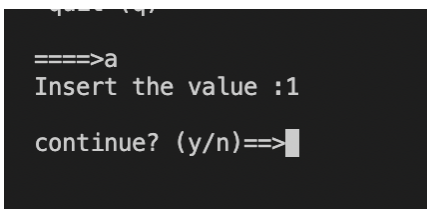
Group 1A

I have made the program interactive by giving the user to select what operation to perform on the linked list.

Here I initialize the list with the list_init() function. Then I give the user the options to choose form and using switch cases I execute the relevant function with the necessary inputs from the user.



The program also doesn't stop after the execution of one of the function and rather asks the user of they want to stop the function or continue working on the link list.



Now lets see the inner mechanisms of the functions I made:

Structure definition

We define the structure and use typedef to shorten the syntax needed to use the structure. Inside the structure we use self-referential structure to make two pointers that will point to the next and previous nodes

```c
typedef struct node
{
    int data;
    struct node* prev;
    struct node* next;

}node;
```

Prepend Function

First w check if the list is empty. If the list is empty then the new node is allocated and then initialized onto the head element or “list”

If the list is not empty then we set the next pointer of the node to point to the currently head node and then we update the head node to point to the new node. We also initialise the previous pointer of the new node as NULL.

```c
void prepend(int element){

    if (list==NULL)
    {
        node *temp=(node*)malloc(sizeof(node));
        temp->data=element;
        temp->next=NULL;
        temp->prev=NULL;
        list=temp;
    }
    else{

        if (isSorted)
        {
            if (element>list->data)
            {
                printf("The list is now unsorted\n");
                isSorted=false;
            }

        }
        node *temp=(node*)malloc(sizeof(node));
        temp->data=element;
        temp->prev=NULL;
        temp->next=list;
        list->prev=temp;
        list=temp;
    }

}
```

Append Function

Just like in the prepend function we check if the list is empty. If so then we do exactly as does in the prepend function.

Now if the list is not empty then we traverse until the “next” pointer of the current node is a null pointer meaning that we have reached the end of the list.

Also we check if the lust was previously sorted before using the sort() function. If so then we set the flag to false and also give a message to the user.

Then we append the new node by setting the next and previous pointers as necessary.

```
void append(int element){

    if (list==NULL)
    {
        node *temp=(node*)malloc(sizeof(node));
        temp->data=element;
        temp->next=NULL;
        temp->prev=NULL;
        list=temp;
    }
    else{

        node *temp=(node*)malloc(sizeof(node));
        temp->data=element;
        node* last=list;
        while (last->next!=NULL)
        {
            last=last->next;
        }

        if (isSorted)
        {
            if (element<last->data)
            {
                printf("The list is now unsorted\n");
                isSorted=false;
            }

        }
        last->next=temp;
        temp->next=NULL;
        temp->prev=last;
    }

}
```

Insert Sorted

Before doing anything in the insert sorted function we run the sort function. This is because the user may not have entered nodes into the list in sorted order till now. So if the list is not sorted already then adding elements in sorted order will have unexpected behaviour. The sort function is first described below:

The sort first checks if the list is empty or a single element list. If so then the function returns. If not then it counts the number of elements in the list and starts a nested loop with 2 of out node pointes which iterates over the list following a bubble sort algorithm until the list is fully sorted.

```
void sort() {
    if (list == NULL || list->next == NULL)
        return; // Nothing to sort if list is empty or has only one element

    int n = 0;
    node* temp;
    for (temp = list; temp != NULL; temp = temp->next)
        n++;

    node *a1, *a2;
    for (int i = 0; i < n - 1; i++) {
        int swap = 0;
        a1 = list;
        a2 = list->next;
        for (int j = 0; j < n - 1 - i; j++) {
            if (a1->data > a2->data) {
                //DEBUG printf("Swapping %d and %d\n", a1->data, a2->data);

                // Swap data without modifying next and prev pointers
                int tempData = a1->data;
                a1->data = a2->data;
                a2->data = tempData;

                swap = 1;
            }
            a1 = a1->next;
            a2 = a2->next;
        }
        if (!swap)
            break;
    }
}
```

```
void insert_sorted(int element){

    if (!isSorted)
    {
        sort();
        isSorted=true;
    }

    node* n = (node*)malloc(sizeof(node));
    n->data = element;
    n->next = NULL;

    if(list == NULL)
    {
        list = n;
        return;
    }
    else if(n->data < list->data)
    {
        list->prev=n;
        n->next = list;
        n->prev=NULL;
        list = n;
    }
    else
    {
        for(node* curr = list; curr != NULL; curr = curr->next)
        {
            if(curr->next == NULL)
            {
                curr->next = n;
                n->prev=curr;
                break;
            }
            if(n->data < curr->next->data)
            {
                n->next=curr->next;
                n->prev=curr;
                curr->next->prev=n;
                curr->next = n;

                break;
            }
        }
    }
}
```

Now the in Insert sorted function we first create a new node for the list. Usually we check if the list is empty and if so then we initialize the head element as the new node.

On the other case we iterate over the sorted list until we find that the new element is smaller than the next element in the iteration. Then we set prev and next pointers as necessary.

Insert After

```
void insert_after(int element, int pred)
{
    node *n = (node*)malloc(sizeof(node));
    n->data = element;
    n->next = NULL;
    n->prev = NULL;

    node* curr = list;
    while(curr->data != pred)
    {
        curr = curr->next;
    }
    if (isSorted)
    {
        if (element<curr->data || element>curr->next->data)
        {
            printf("The list is now unsorted\n");
            isSorted=false;
        }

    }
    curr->next->prev=n;
    n->next = curr->next;
    curr->next = n;
    n->prev = curr;
}
```

Here we first make a new node with malloc and then look for our targeted element. Then we insert the new element after the targeted element. And if the list was sorted before and our new element breaks the sort the the user is let know of that.

```
void destroy(node* n)
{
    if(n == NULL)
    {
        printf("List destroyed, all data deleted\n");
        return;
    }
    destroy(n->next);
    free(n);
}
```

Destroy

The Destroy function recursively frees all the memorry allocated for the different nodes.

Delete Node

The delete node function will look for the mentioned element and if found will remove any reference of the nodes' pointers from other nodes and then free the memory allocated for it.

```
void delete_node(int element)
{
    node* curr=list;
    for(node* trav = list; trav != NULL; trav = trav->next)
    {
        if(trav->data == element)
        {
            curr=trav;
            //DEBUG printf("found the element: %d\n",trav->data);
            //DEBUG printf("found the element: %d\n",curr->data);
            break;
        }
    }
    if (curr==NULL)
    {
        printf("The targeted element is not present");
        return;
    }

    //DEBUG else printf("found the element: %d",curr->data);

    curr->next->prev = (node*)curr->prev;
    curr->prev->next = (node*)curr->next;
    free(curr);
    //DEBUG printf("\n%d",list->next->next->prev->data);
    //DEBUG printf("\n%d",list->next->next->next->data);

}
```

```
void delete_front(){

    if(list == NULL)
    {
        printf("List is empty!\n");
        return;
    }
    node* temp;
    temp = list;
    list = list->next;
    list->prev=NULL;
    free(temp);
}


void delete_back(){

    if(list == NULL)
    {
        printf("List is empty!\n");
        return;
    }
    node* curr = list;
    while(curr->next != NULL)
    {
        curr = curr->next;
    }
    curr->prev->next=NULL;
    free(curr);
}
```

Delete front

This function just makes the 2nd node the first node and frees the memory of the original first node.

Delete back

This function iterates over to the last node and then removes its reference from the previous node and then deallocates the memory.

Find

The find function just linearly iterates over the list and if found, returns a bool true and if not found returns a false statement.

```
bool find(int element){

    node* curr=list;
    int i=0;
    for(node* trav = list; trav != NULL; trav = trav->next)
    {
        if(trav->data == element)
        {
            curr=trav;
            //DEBUG printf("found the element: %d\n",trav->data);
            //DEBUG printf("found the element: %d\n",curr->data);
            break;
        }
        i++;
    }
    if (curr==NULL)
    {
        printf("The targeted element is not present");
        return false;
    }
    else {
        printf("found the element= %d at index=%d",curr->data,i);
        return true;
    }
}
```

The End