DATA STRUCTURES

&

ALGORITHMS

Assignment

Submitted By:

Ghulam Mustafa

Fa-22/BSCS-188

Section: ‘E’

Submitted To:

Ms. Sahar Moin

Linked Lists Problems

Problem 1: Suppose there is a singly linked list with values 10->20->30
Write a pseudocode to Reverse it. (not printing in reverse order. Actually, reverse it).

Algorithm	Pseudocode

1. Initialize pointers:
• Set current to the head of the list.
• Set previous to NULL.
• Set the following to NULL.
2. Reverse the list:
• While current is not NULL:
• Save the next node in following.
• Reverse the link by setting current's next to previous.
• Move previous to current.
• Move current to following.
3. Update the head:
• Set the head of the list to previous.
	function reverse_list():
	// Step 1
	current = head
	previous = NULL
	following = NULL
	
	// Step 2
	while current is not NULL:
	following = current.next
	current.next = previous
	previous = current
	current = following
	
	Step 3
	head = previous

Program:

```
#include<iostream>
using namespace std;
struct Node {
   int data;
   Node* next;
};
class LinkedList {
public:
   Node* head;

   LinkedList() {
      head = NULL;
   }

        void insert_at_end(int newElement) {
      Node* newNode = new Node();
      newNode->data = newElement;

      if (head == NULL) {
         head = newNode;
         return;
      }

      Node* temp = head;
      while (temp->next != NULL) {
         temp = temp->next;
      }
      temp->next = newNode;
   }
      void reverse_list() {
      Node* current = head;
      Node* previous = NULL;
 Node* following = NULL;
 while (current != NULL) {
         following = current->next;
         current->next = previous;
         previous = current;
         current = following;
      }
      head = previous;
   }
   void display() {
      Node* temp = head;
      if (head == NULL) {
         cout << "List is empty." << endl;
         return;
      }
      while (temp != NULL) {
         cout << temp->data << "\t";
         temp = temp->next;
      }
      cout << endl;
   }
};
int main() {
           LinkedList MyList;
           for(int i=10;i<=30;i+=10){
      MyList.insert_at_end(i);
   }
   cout << "Original List:" << endl;
   MyList.display();
   cout << "Reversed List: "<< endl;
   MyList.reverse_list();
   MyList.display();

   return 0;
}
```

Problem 2: Suppose there are two singly linked lists with values 10->20->30 and 40->50->60. Merge these two in one list!

Algorithm

1. Initialize merged list:

   - Create a new linked list named mergedList.

2. Merge elements from the lists

   - Set a temporary pointer temp to the head of the first list (head1).
   - While temp is not NULL:
   - Insert the data of the current node into mergedList.
   - Move temp to the next node.
   - Repeat for Second List(head 2)

3. Return the merged list.

Pseudocode

```
function merge_lists(head1, head2):
    // Step 1
    mergedList = new LinkedList()

    // Step 2
    temp = head1
    while temp is not NULL:
        mergedList.insert_at_end(temp.data)
        temp = temp.next

    // Step 3
    temp = head2
    while temp is not NULL:
        mergedList.insert_at_end(temp.data)
        temp = temp.next

    // Step 4
    return mergedList
```

Program:

```
#include<iostream>
using namespace std;
struct Node {
    int data;
    Node* next;
};
class LinkedList {
public:
    Node* head;
    LinkedList() {
        head = NULL;
    }
        void insert_at_end(int newElement) {
        Node* newNode = new Node();
        newNode->data = newElement;
        if (head == NULL) {
            head = newNode;
            return;
        }
        Node* temp = head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
    void display() {
        Node* temp = head;
        if (head == NULL) {
            cout << "List is empty." << endl;
            return;
        }
        while (temp != NULL) {
            cout << temp->data << "\t";
            temp = temp->next;
        }
        cout << endl;
    } };
```

```
LinkedList merge_lists(Node* head1, Node* head2) {
    LinkedList mergedList;
    Node* temp = head1;
    while (temp != NULL) {
        mergedList.insert_at_end(temp->data);
        temp = temp->next;
    }
    temp = head2;
    while (temp != NULL) {
        mergedList.insert_at_end(temp->data);
        temp = temp->next;
    }
    return mergedList;
}
int main() {
    LinkedList MyList1 , MyList2 , MergedList;
    int i = 1;
    for(int i=10;i<=30;i+=10) {
        MyList1.insert_at_end(i);
    }
    for(i=40;i<=60;i+=10) {
        MyList2.insert_at_end(i);
    }
    cout << "Original List 1:" << endl;
    MyList1.display();

            cout << "Original List 2:" << endl;
    MyList2.display();

    cout<< "Merged List: "<<endl;
    MergedList = merge_lists(MyList1.head , MyList2.head);
            MergedList.display();
    return 0;
}
```

**Problem 3: Suppose we have a doubly linked list with values 12->34->55
Write a function to delete all nodes of the linked list.**

Algorithm	Pseudocode

1. Check if the list is empty:
- If the list is empty (head is NULL), inform that the list is empty and stop.
2. Initialize pointers:
- Set a temporary pointer (temp) to the node after the head.
- Initialize another pointer (nextNode) to NULL.)
3. Set head to NULL:
- Set the head to NULL, effectively keeping only the head node in the list.
4. Loop through the remaining nodes and delete each one:
- While the temporary pointer (temp) is not NULL:
- Save the next node in nextNode to avoid losing the connection.
- Delete the current node (temp).
- Move to the next node by updating temp with the saved next node (nextNode)
	procedure delete_all():
	    if head is NULL:
	        output "List is empty."
	        return
	
	    temp = head.next
	    nextNode = NULL
	
	    head = NULL
	
	    while temp is not NULL:
	        nextNode = temp.next
	
	        delete temp
	        temp = nextNode

Program:

```
void delete_all() {
        if (head == NULL) {
            cout << "List is empty." << endl;
            return;
        }
        Node* temp = head->next;
        Node* aglaNode = NULL;
        head = NULL;
        while (temp != NULL) {
            aglaNode = temp->next;
            delete temp;
            temp = aglaNode;
        }
    }

    void display() {

        Node* temp = head;
        if (head == NULL) {
            cout << "List is empty." << endl;
            return;

        }
```

```
while (temp != NULL) {
            cout << temp->data << "\t";
            temp = temp->next;
        }
        cout << endl;
    }
};
int main() {
    DoublyLinkedList myList;
    myList.insert_at_end(12);
    myList.insert_at_end(34);
    myList.insert_at_end(55);

        cout << "Original List: ";
        myList.display();

        myList.delete_all();

        cout << "Empty List: ";
        myList.display();

    return 0;
}
```

Problem 4: Suppose we have a doubly linked list with values 12->34->55S. Write a function to search value 55 in this linked list. If it is present delete it and display the new linked list.

Algorithm

1. **Check if the list is empty:**
- If the head is NULL, output an error message indicating that the list is empty and return.
2. **Initialize pointers and variables:**
- Set a temporary pointer (temp) to the head.
- Initialize a variable (position) to 1 to keep track of the node position.
- Initialize a boolean variable (found) to false, indicating whether the element is found.
3. **Search for the element:**
- **While** the temporary pointer (temp) is not NULL:
- If the data in the current node is equal to the target element:
- Output a message indicating the value is found at the current position.
- Set found to **true**.
- Call the **delete_at_position** procedure with the current position.
- **Exit** the **loop**.
4. **Handle not found case:**
- If the element is not found (found is false), output a message indicating that the value was not found.
5. **delete_at_position Procedure:**

❖ **Validate position:**
- Check if the specified position is <= to 0.
- If true, output an error message about an invalid position for deletion and return.

❖ **Check if the list is empty:**
- If the head is NULL, output an error message indicating that the list is empty and return.

❖ **Initialize pointers and variables:**
- Set a temporary pointer (temp) to the head.
- Initialize a pointer (prev) to NULL to keep track of the previous node.
- Initialize a counter (counter) to 1 to keep track of the current position.

❖ **Traverse to the specified position:**
- While the temporary pointer (temp) is not NULL and the counter is less than the specified position:
- Update the prev pointer to the current node.
- Move to the next node by updating temp.
- Increment the counter.

❖ **Handle invalid position:**
- If the temporary pointer (temp) is NULL after traversal, output an error message about an invalid position for deletion and return.

❖ **Update pointers to delete the node at the specified position:**
- If prev is not NULL, update its next pointer to skip the current node.
- If the next node after the current node is not NULL, update its prev pointer to skip the current node.

❖ **Update head if necessary:**
- If prev is NULL, update the head to the next node after the current node.

6. **Delete the node:** Delete the current node (temp).

Pseudocode

```
procedure search_and_delete(element):
    if head is NULL:
        output "Error: List is empty."
        return

    temp = head
    position = 1
    found = false

    while temp is not NULL:
        if temp.data is equal to element:
            output "Value", element, "found at Node:", position
            found = true
            delete_at_position(position)
            exit loop

        temp = temp.next
        position = position + 1

    if not found:
        output "Value Not Found!"

procedure delete_at_position(position):
    if position is less than or equal to 0:
        output "Error: Invalid position for deletion."
        return

    if head is NULL:
        output "Error: List is empty. Cannot delete from the specified position."
        return

    temp = head
    prev = NULL
    counter = 1

    while temp is not NULL and counter is less than position:
        prev = temp
        temp = temp.next
        counter = counter + 1

    if temp is NULL:
        output "Error: Invalid position for deletion."
        return

    if prev is not NULL:
        prev.next = temp.next
        if temp.next is not NULL:
            temp.next.prev = prev
    else:
        head = temp.next
        if temp.next is not NULL:
            temp.next.prev = NULL

    delete temp
```

Program:

```
void search_and_delete(int element) {
   if (head == NULL) {
      cout << "Error: List is empty." << endl;
      return;
   }
   Node* temp = head;
   int position = 1;
   bool found = false;

   while (temp != NULL) {
      if (temp->data == element) {
         cout << " Value " << element <<
                " found at Node: " << position << endl;
         found = true;
         delete_at_position(position); // in this case 55
         break;
      }
      temp = temp->next;
      position++;
   }
   if (!found)
      cout << "Value Not Found!" << endl;
 }

   void delete_at_position(int position) {
         if (position <= 0) {
         cout<<"Error: Invalid position for deletion."<<endl;
         return;
   }
   if (head == NULL) {
     cout << "Error: List is empty. Cannot delete from
          the specified position." << endl;
     return;
                  }
                  Node* temp = head;
                  Node* prev = NULL;
                  int counter = 1;
       while (temp != NULL && counter <position) {
                       prev = temp;
                       temp = temp->next;
                       counter++;
        }
                     if (temp == NULL) {
     cout << "Error: Invalid position for deletion." << endl;
                       return;
                     }
                     if (prev != NULL) {
                       prev->next = temp->next;
                       if (temp->next != NULL) {
                          temp->next->prev = prev;
                       }
                     } else {
                       head = temp->next;
                       if (temp->next != NULL) {
                          temp->next->prev = NULL;
                       }
                     }                   delete temp; }
```