COSC230 Practical 3:
Linked Lists
(Practical for Week 4 and 5)

The purpose of this practical is to understand how various types of linked lists and their associated operations can be implemented in C++, how linked lists are represented in the Standard Template Library (STL), and how linked lists can be used as an underlying data structure in a simple database programming example.

Question 1: **Implementing a Linked List with a pointer to tail**

In lectures we have looked in detail at the implementation of a class for a singly linked list. The singly linked list turns out to be an efficient implementation for a stack data structure, as we shall see later in this unit. However, an efficient queue data structure requires a little more work. In this question, you will implement a singly linked list interface (given below) in C++ that includes a pointer to the tail (i.e., the last node) of the list. Don't confuse this use of tail with its common usage in functional programming, where "tail" means all nodes after the head of the list. Make use of the same `Node` class we used for `Linked_List` (the singly linked list class). Draw box and pointer diagrams to help you make sure to correctly update the tail pointer in each of your member function implementations. Use the data display debugger, `ddd`, to visualize your linked-list pointers as you step through the code you develop.

```
template<class T>
class Linked_List_Tail {
public:
    Linked_List_Tail() { head = 0, tail = 0; }
    ~Linked_List_Tail();
    bool empty() { return head == 0; }
    T front() { return head->key; }         // Extra member
    void remove_from_head();
    void insert_at_tail(T);                 // Extra member
private:
    Node<T> *head, *tail;                   // Extra member
};
```

Question 2: **Implementing a Linked List without pointers**

How do we implement a linked list without pointers? As we saw in lectures, this is possible using arrays and array indices (please see the lecture on implementing linked lists without pointers before attempting this question). In the interface provided below, `Node_Array` is a class that includes fixed-sized arrays for the members `key`, `prev`, and `next`, and replaces the `Node` class used previously. This class also initializes the "free list", representing the pool of memory available for constructing linked lists (see Lectures). Using this interface, implement its members, and directly compare them with the members of `Doubly_Linked_List`. You should notice many similarities!

```
// Node Array class (with arrays of length 4) replaces Node class

template<class T>
class Node_Array {
public:
    Node_Array(): key()
    {
        // Initialize Node Array to be "free" list
        free = 0;
        for (int i = 0; i != 4; i++) {
            next[i] = i + 1;
        }
        next[3] = -1;                   // -1 represents a null array index
    }
    int free;                           // head of "free" list
    T key[4];
    int prev[4], next[4];
};

template<class T>
class Array_Linked_List {
public:
    Array_Linked_List() { head = -1; }  // -1 represents a null array index
    bool empty() { return head == -1; }
    void insert(Node_Array<T>&, T);
    void remove(Node_Array<T>&, int);
    void remove_from_head(Node_Array<T>& NA) { remove(NA, head); }
private:
    int head;
};
```

Question 3: **Building a Simple Database using the STL**

Now that you understand how various linked lists can be implemented in `C++`, it is time to use them in real applications. This will be done using the `list` class in the Standard Template Library. This class is implemented as a doubly linked list in the STL, and comes with a number of useful member functions and algorithms. To use any of these you will need to make use of iterators. The application here is to write a simple database that manages airline reservations. To keep it simple, the only information considered here is the passenger surname. Your job is to fill out `case 2` to `case 6` in the program given below. Make sure to test your program to check it produces the expected behaviour. What is the worst-case asymptotic complexity of each operation?

```
#include<list>
#include<algorithm>
#include<iostream>
#include<string>
#include<fstream>
using namespace std;

int menu()
{
    int option;
    cout << endl;
    cout << "Enter one of the following options:" << endl;
    cout << "1. load reservations from file:" << endl;
    cout << "2. reserve a ticket" << endl;
    cout << "3. cancel a reservation" << endl;
    cout << "4. check reservation" << endl;
    cout << "5. display passenger list" << endl;
    cout << "6. save passenger list" << endl;
    cout << "7. exit" << endl << endl;
    cin >> option;
    cin.get();
    return option;
}
```

```
int main()
{
    list<string> flight_list;
    list<string>::iterator i1, i2;
    string name;

    while (true)
    {
        switch (menu())
        {
            case 1:
                {
                    ifstream input("ticket_reservations.dat");
                    while (input >> name)
                    {
                        flight_list.push_back(name);
                    }
                    input.close();
                    break;
                }

            case 2:
                .
                .
                .
            case 7:
                return 0;
        }
    }
    return 0;
}
```

Solutions

Question 1:

```
template<class T>
Linked_List_Tail<T>::~Linked_List_Tail()
{
    while (!empty())
        remove_from_head();
}

// Delete node from head

template<class T>
void Linked_List_Tail<T>::remove_from_head()
{
    Node<T>* p = head;
    head = head->next;
    delete p;
    if (head == 0)
        tail = 0;
}

// Insert new node with key "k" at tail

template<class T>
void Linked_List_Tail<T>::insert_at_tail(T k)
{
    if (tail != 0) {
        // Head exists
        tail->next = new Node<T>(k, 0);
        tail = tail->next;
    }
    else {
        // No head exists
        head = new Node<T>(k, head);
        tail = head;
    }
}
```

Question 2:

```
// Insert key at head of list

template<class T>
void Array_Linked_List<T>::insert(Node_Array<T>& NA, T k)
{
    // Allocate object
    if (NA.free == -1)
        throw;              // free list is empty!
    int p = NA.free;        // p is now an index into a Node Array
    NA.free = NA.next[p];   // update head of free list

    // Insert k into list
    NA.key[p] = k;
    NA.next[p] = head;
    if (!empty())
        NA.prev[head] = p;
    head = p;
    NA.prev[p] = -1;
}

// Delete arbitary node p

template<class T>
void Array_Linked_List<T>::remove(Node_Array<T>& NA, int p)
{
    // Reassign next and prev indices
    if (NA.prev[p] != -1)
        NA.next[NA.prev[p]] = NA.next[p];
    else
        head = NA.next[p];
    if (NA.next[p] != -1)
        NA.prev[NA.next[p]] = NA.prev[p];

    // Free object
    NA.next[p] = NA.free;   // update "next" member of free list
    NA.free = p;            // update head of free list
}
```

Question 3:

The complete database program is given as:

```
#include<list>
#include<algorithm>
#include<iostream>
#include<string>
#include<fstream>
using namespace std;

int menu()
{
    int option;
    cout << endl;
    cout << "Enter one of the following options:" << endl;
    cout << "1. load reservations from file:" << endl;
    cout << "2. reserve a ticket" << endl;
    cout << "3. cancel a reservation" << endl;
    cout << "4. check reservation" << endl;
    cout << "5. display passenger list" << endl;
    cout << "6. save passenger list" << endl;
    cout << "7. exit" << endl << endl;
    cin >> option;
    cin.get();
    return option;
}

int main()
{
    list<string> flight_list;
    list<string>::iterator i1, i2;
    string name;

    while (true)
    {
        switch (menu())
        {
            case 1:
                {
                    ifstream input("ticket_reservations.dat");
                    while (input >> name)
                    {
```

```
            flight_list.push_back(name);
        }
        input.close();
        break;
    }

case 2:
    {
        cout << "name of passenger:" << endl;
        cin >> name;
        flight_list.push_back(name);
        break;
    }

case 3:
    {
        cout << "name of passenger:" << endl;
        cin >> name;
        flight_list.remove(name);
        break;
    }

case 4:
    {
        cout << "name of passenger:" << endl;
        cin >> name;
        i1 = flight_list.begin();
        i2 = flight_list.end();
        if (find(i1, i2, name) != i2)
            cout << "this passenger has a ticket reservation" << endl;
        else
            cout << "this passenger does not have
                        a ticket reservation" << endl;
        break;
    }

case 5:
    {
        flight_list.sort();
        i1 = flight_list.begin();
        i2 = flight_list.end();
        for ( ; i1 != i2; ++i1) {
```

```
                    cout << *i1 << endl;
                }
                break;
            }

        case 6:
            {
                flight_list.sort();
                i1 = flight_list.begin();
                i2 = flight_list.end();
                ofstream output("ticket_reservations.dat");
                for ( ; i1 != i2; ++i1) {
                    output << *i1 << " ";
                }
                output.close();
            }
            break;

        case 7:
            return 0; // returns false to break out of while loop
    }
}

    return 0;
}
```

The worst-case asymptotic complexity to reserve a ticket is $\Theta(1)$. All of the other operations are $\Omega(n)$ (as some involve sorting). Check an online reference for `STL` and `std` for further details.