Experiment 1.3: Linked List, Operating System Algorithm

Problem 1.3.1: Merge Two Sorted Lists

Problem Statement: You are given the heads of two sorted linked lists list1 and list2.
Merge the two lists into one sorted list. The list should be made by splicing together
the nodes of the first two lists. Return the head of the merged linked list.

Algorithm:

Initialize a dummy node to serve as the starting point for the merged list.

Create a pointer current to traverse and build the new list.

Compare the elements from list1 and list2:

If the value of list1 is smaller, append it to the new list and move list1 pointer to the next node.

If the value of list2 is smaller, append it to the new list and move list2 pointer to the next node.

If one list is exhausted, append the remaining nodes of the other list to the merged list.

Return the next node of the dummy (since it's the start of the merged list).

Code:

```
#include <iostream>

// Definition for singly-linked list.
struct ListNode {
    int val;
    ListNode* next;
    ListNode() : val(0), next(nullptr) {}
    ListNode(int x) : val(x), next(nullptr) {}
    ListNode(int x, ListNode* next) : val(x), next(next) {}
};

class Solution {
public:
    ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
        // Create a dummy node to serve as the start of the merged list
        ListNode* dummy = new ListNode();
        ListNode* current = dummy;

        // Traverse both lists and merge them
        while (list1 != nullptr && list2 != nullptr) {
            if (list1->val < list2->val) {
                current->next = list1;
                list1 = list1->next;
            } else {
                current->next = list2;
                list2 = list2->next;
            }
            current = current->next;
        }

        // If any list has remaining nodes, append them
        if (list1 != nullptr) {
            current->next = list1;
        } else if (list2 != nullptr) {
            current->next = list2;
        }

        // Return the next node of dummy as the merged list head
        return dummy->next;
    }
};

// Helper function to print the linked list
void printList(ListNode* head) {
    while (head != nullptr) {
        std::cout << head->val << " ";
        head = head->next;
    }
    std::cout << std::endl;
}

// Main function for testing
int main() {
    // Create two sorted linked lists
    ListNode* list1 = new ListNode(1, new ListNode(2, new ListNode(4)));
    ListNode* list2 = new ListNode(1, new ListNode(3, new ListNode(4)));

    Solution solution;
    ListNode* mergedList = solution.mergeTwoLists(list1, list2);

    // Print the merged list
    std::cout << "Merged List: ";
    printList(mergedList);

    return 0;
}
```

Problem 1.3.2: Remove Duplicates from Sorted List II

Problem Statement: Given the head of a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list.

Return the linked list sorted as well.

Algorithm:

Initialize a dummy node to handle edge cases.

Use a pointer prev to point to the dummy node and current to traverse the list.

Check for duplicate values by comparing the current node's value with the next node.

If duplicates are found, skip all nodes with that value.

If no duplicates are found, update the prev pointer to point to the current node.

Return the next node of the dummy (which represents the start of the updated list).

Code:

```
#include <iostream>

// Definition for singly-linked list.
struct ListNode {
   int val;
   ListNode* next;
   ListNode() : val(0), next(nullptr) {}
   ListNode(int x) : val(x), next(nullptr) {}
   ListNode(int x, ListNode* next) : val(x), next(next) {}
};

class Solution {
```

```
public:
   ListNode* deleteDuplicates(ListNode* head) {
      // Create a dummy node to handle edge cases easily
      ListNode* dummy = new ListNode(0);
      dummy->next = head;

      // Pointer to track the last node before duplicates
      ListNode* prev = dummy;

      while (head != nullptr) {
         // If the current node has duplicates, skip all duplicates
         if (head->next != nullptr && head->val == head->next->val) {
            // Skip nodes with the same value
            while (head->next != nullptr && head->val == head->next->val) {
               head = head->next;
            }
            // Link the prev node to the next distinct node
            prev->next = head->next;
         } else {
            // No duplicates, just move the prev pointer
            prev = prev->next;
         }
         // Move to the next node
         head = head->next;
      }

      return dummy->next;  // Return the new head of the list
   }
};

// Helper function to print the linked list
```

```
void printList(ListNode* head) {
    while (head != nullptr) {
        std::cout << head->val << " ";
        head = head->next;
    }
    std::cout << std::endl;
}

// Main function for testing
int main() {
    // Create a sorted linked list with duplicates
    ListNode* head = new ListNode(1, new ListNode(2, new ListNode(3, new ListNode(3, new
ListNode(4, new ListNode(4, new ListNode(5)))))));

    Solution solution;
    ListNode* updatedList = solution.deleteDuplicates(head);

    // Print the updated list
    std::cout << "Updated List: ";
    printList(updatedList);

    return 0;
}
```

Problem 1.3.3: LRU Cache

Problem Statement: Design a data structure that follows the constraints of a Least Recently Used (LRU) cache.

Algorithm:

Use a doubly linked list to store the cache and a hash map for O(1) lookups.

The linked list stores the keys in order of use. The head of the list represents the most recently used, and the tail represents the least recently used.

For get(key), move the node to the head if it exists, and return the value. If not, return -1.

For put(key, value), if the key already exists, update the value and move it to the head. If the key doesn't exist, insert it at the head. If the cache is full, remove the node from the tail.

Code:

```
#include <unordered_map>

#include <list>

#include <iostream>

using namespace std;


class LFUCache {

   // Capacity of the LFU cache

   int capacity;


   // Min frequency tracker

   int minFreq;


   // Key to value and frequency map

   unordered_map<int, pair<int, int>> cache;


   // Frequency to list of keys map

   unordered_map<int, list<int>> freqMap;


   // Key to iterator map (for quick access to the key's location in freqMap)

   unordered_map<int, list<int>::iterator> keyIter;
```

```
public:
    LFUCache(int capacity) {
        this->capacity = capacity;
        this->minFreq = 0;
    }

    int get(int key) {
        if (cache.find(key) == cache.end())
            return -1;  // Key not found

        // Update frequency of the key
        _updateFreq(key);

        return cache[key].first;
    }

    void put(int key, int value) {
        if (capacity == 0) return;

        if (cache.find(key) != cache.end()) {
            // Key already exists, update the value
            cache[key].first = value;
            _updateFreq(key);
        } else {
            // If the cache is full, evict the least frequently used key
            if (cache.size() == capacity) {
                _evict();
            }

            // Insert the new key-value pair
            cache[key] = {value, 1};  // {value, frequency}
            freqMap[1].push_back(key);  // Add key to frequency 1 list
            keyIter[key] = --freqMap[1].end();  // Store the iterator to the key
            minFreq = 1;  // Reset minimum frequency
        }
    }

private:
    void _updateFreq(int key) {
        int freq = cache[key].second;  // Current frequency of the key
        freqMap[freq].erase(keyIter[key]);  // Remove the key from its current frequency list

        // If the current frequency list is empty and it's the minimum frequency, update minFreq
        if (freqMap[freq].empty() && freq == minFreq) {
            minFreq++;
        }

        // Increase the frequency
        cache[key].second++;
        freqMap[freq + 1].push_back(key);  // Add the key to the new frequency list
        keyIter[key] = --freqMap[freq + 1].end();  // Update the iterator for the key
    }

    void _evict() {
        // Evict the least frequently used key (the first key in the list with the minimum frequency)
        int keyToEvict = freqMap[minFreq].front();
        freqMap[minFreq].pop_front();  // Remove the key from the frequency list

        cache.erase(keyToEvict);  // Remove the key from the cache
        keyIter.erase(keyToEvict);  // Remove the iterator tracking
    }
};
```

```
// Helper function to test the LFUCache
int main() {
    LFUCache lfuCache(2);  // Capacity 2

    lfuCache.put(1, 1);
    lfuCache.put(2, 2);

    cout << "Get 1: " << lfuCache.get(1) << endl;  // Should return 1

    lfuCache.put(3, 3);  // Evicts key 2

    cout << "Get 2: " << lfuCache.get(2) << endl;  // Should return -1 (not found)
    cout << "Get 3: " << lfuCache.get(3) << endl;  // Should return 3

    lfuCache.put(4, 4);  // Evicts key 1

    cout << "Get 1: " << lfuCache.get(1) << endl;  // Should return -1 (not found)
    cout << "Get 3: " << lfuCache.get(3) << endl;  // Should return 3
    cout << "Get 4: " << lfuCache.get(4) << endl;  // Should return 4

    return 0;
}
```