Data Structures Lab

Lab Task # 04

Rayan Ahmed

23i-0018

AI-B

Code:

array.cpp:

```cpp
#include <iostream>

using namespace std;

class patient
{
private:
    string name;
    int id;

public:
    patient()
    {
        name = "";
        id = 0;
    }
    patient(string name, int id)
    {
        this->name = name;
        this->id = id;
    }

    friend ostream &operator<<(ostream &output, const patient &other)
    {
        output << "Name: " << other.name << endl;
        output << "ID: " << other.id << endl;
    }
};

class Array
{
private:
    patient arr[20];
    int size;

public:
```

```
    Array()
    {
        size = 0;
    }

    // Adds a patient at end of queue
    void push_back(string name, int id)
    {
        if (size == 19)
        {
            cout << "Queue is already full!" << endl;
            return;
        }

        arr[size] = patient(name, id);
        size++;
    }

    // Adds a patient at start of queue
    void push_front(string name, int id)
    {
        if (size == 19)
        {
            cout << "Queue is already full!" << endl;
            return;
        }

        for (int i = size + 1; i > 0; i--)
            arr[i] = arr[i - 1];
        arr[0] = patient(name, id);

        size++;
    }

    // Adds a patient at a specific index
    void insert(int index, string name, int id)
    {
        if (size == 19)
        {
            cout << "Queue is already full!" << endl;
            return;
        }

        for (int i = size + 1; i > index; i--)
            arr[i] = arr[i - 1];
        arr[index] = patient(name, id);

        size++;
    }

    // Removes a patient in the beginning
    void pop_front()
    {
        if (size == 0)
        {
            cout << "Queue is already empty!" << endl;
            return;
        }

        for (int i = 0; i < size - 1; i++)
            arr[i] = arr[i + 1];
        size--;
    }

    // Removes a patieng in the end
    void pop_back()
```

```
    {
        if (size == 0)
        {
            cout << "Queue is already empty!" << endl;
            return;
        }

        size--;
    }

    // Removes a patient at a specific index
    void remove(int index)
    {
        if (index < 0 || index >= size)
        {
            cout << "Index out of range!" << endl;
            return;
        }
        if (size == 0)
        {
            cout << "Queue is already empty!" << endl;
            return;
        }

        for (int i = index; i < size - 1; i++)
            arr[i] = arr[i + 1];
        size--;
    }

    // Displays all the patients
    void print()
    {
        cout << "Number of Patients in queue: " << size << endl;
        for (int i = 0; i < size; i++)
            cout << arr[i] << endl;
    }
};

int main()
{
    Array arr;

    arr.push_back("Hello", 1);
    arr.push_back("Helloasdf", 2);
    arr.push_back("He", 3);
    arr.push_back("Hel", 4);
    arr.push_front("Rayan", 0);
    arr.insert(5, "AHmed", 8);
    arr.pop_front();
    arr.pop_back();
    arr.pop_back();
    arr.pop_back();

    arr.print();

    return 0;
}
```

linked_list.cpp:

```cpp
#include <iostream>

using namespace std;

class patient
{
private:
    string name;
    int id;

public:
    patient()
    {
        name = "";
        id = 0;
    }
    patient(string name, int id)
    {
        this->name = name;
        this->id = id;
    }

    friend ostream &operator<<(ostream &output, const patient &other)
    {
        output << "Name: " << other.name << endl;
        output << "ID: " << other.id << endl;
    }
};

struct Node
{
    patient data;
    Node *next;
};

class List
{
private:
    Node *head;

public:
    List() { head = NULL; }

    // Adds a patient at the end of the list
    void push_back(string name, int id)
    {
        Node *newNode = new Node();
        newNode->data = patient(name, id);
        newNode->next = NULL;
        if (head == NULL)
            head = newNode;
        else
        {
            Node *last = head;
            while (last->next != NULL)
                last = last->next;
            last->next = newNode;
        }
    }
```

```
// Adds a patient at the start of the list
void push_front(string name, int id)
{
    Node *newNode = new Node();
    newNode->data = patient(name, id);

    newNode->next = head;
    head = newNode;
}

// Inserts a patient at a specific index
void insert(int index, string name, int id)
{
    if (index == 0)
    {
        push_front(name, id);
        return;
    }

    Node *newNode = new Node();
    newNode->data = patient(name, id);
    Node *current = head;
    int currentIndex = 0;

    while (current != NULL && currentIndex < index - 1)
    {
        current = current->next;
        currentIndex++;
    }

    if (current == NULL)
        cout << "Index out of range" << endl;
    else
    {
        newNode->next = current->next;
        current->next = newNode;
    }
}

// Removes a patient in the end
void pop_back()
{
    if (head == NULL)
    {
        cout << "List is empty" << endl;
        return;
    }
    else if (head->next == NULL)
    {
        delete head;
        head = NULL;
        return;
    }

    Node *current = head;
    while (current->next->next != NULL)
        current = current->next;

    delete current->next;
    current->next = NULL;
}

// Removes a patient in the beginning
void pop_front()
{
    if (head == NULL)
```

```
            cout << "List is empty" << endl;
        else
        {
            Node *temp = head;
            head = head->next;

            delete temp;
        }
    }

    // Removes a patient at a specific index
    void remove(int index)
    {
        if (index < 0)
        {
            cout << "Index out of range" << endl;
            return;
        }

        if (index == 0)
        {
            pop_front();
            return;
        }

        Node *current = head;
        int currentIndex = 0;

        while (current != NULL && currentIndex < index - 1)
        {
            current = current->next;
            currentIndex++;
        }

        if (current == NULL || current->next == NULL)
        {
            cout << "Index out of range" << endl;
            return;
        }

        Node *nodeToDelete = current->next;
        current->next = nodeToDelete->next;
        delete nodeToDelete;
    }

    // Displays all the patients
    void print()
    {
        Node *current = head;

        while (current != NULL)
        {
            cout << current->data << endl;
            current = current->next;
        }
    }
};

int main()
{
    List list;

    list.push_back("Ryaan", 1);
    list.push_back("Ryaan", 1);
    list.push_back("Ryaan", 1);
    list.push_front("Ahmed", 2);
```

```
    list.push_front("Ahmed", 2);
    list.push_front("Ahmed", 2);
    list.insert(6, "Hola", 3);
    list.insert(7, "Hola", 3);
    list.push_back("heloo", 4);

    list.pop_back();
    list.pop_back();
    list.pop_front();
    list.pop_front();
    list.remove(4);

    list.print();

    return 0;
}
```

Time Complexity Comparison:

- **Arrays:**
  - **Accessing Elements**: Accessing elements by index is more efficient in arrays, as it has constant time complexity O(1). You can access any patient directly using the index.
  - **Insertion at the End (push_back)**: Inserting a patient at the end of the array is efficient with a time complexity of O(1) (if there's space), but if the array is full, it can become O(n) due to resizing.
  - **Insertion at the Beginning or Middle (push_front/insert)**: Inserting at the beginning or middle is inefficient, requiring shifting elements, leading to a time complexity of O(n).
  - **Deletion at the End (pop_back)**: This operation is efficient and takes O(1).
  - **Deletion at the Beginning or Middle (pop_front/remove)**: Similar to insertion, deleting from the front or middle requires shifting elements and takes O(n).
- **Linked Lists:**
  - **Accessing Elements**: Accessing elements by index in a linked list takes O(n) because you must traverse the list to find the node.
  - **Insertion at the End (push_back)**: Inserting at the end takes O(n) unless you maintain a tail pointer, which reduces it to O(1).
  - **Insertion at the Beginning (push_front)**: This operation is efficient in linked lists and takes O(1), as no shifting is needed.
  - **Deletion at the End (pop_back)**: Deleting the last element requires traversing the list, so it takes O(n) unless a tail pointer is used.
  - **Deletion at the Beginning (pop_front)**: Deleting the first element is efficient and takes O(1).

Summary:

- **Efficient in Arrays**: Direct access, insertion, and deletion at the end.
- **Efficient in Linked Lists**: Insertion and deletion at the beginning, flexible size.

Memory Usage:

- **Arrays**:
  - Arrays require contiguous memory blocks, so they are less memory-efficient if the size isn't well managed. You must declare a fixed size (e.g., 20 patients), which can lead to either wasted space or insufficient space if more patients need to be added.
  - If the array becomes full, a new larger array must be created, and all data must be copied, which increases memory usage temporarily during resizing.
- **Linked Lists**:
  - Linked lists are more memory-efficient for dynamic data because memory is allocated as needed. However, they have extra memory overhead due to storing pointers (each node stores both patient data and a pointer to the next node).
  - In a hospital system where patient data constantly changes, linked lists are more flexible because you can add or remove patients without worrying about fixed sizes.

Ease of Implementation:

- **Arrays**:
  - Arrays are generally easier to implement since you can directly access elements by index. Managing patient data in an array is intuitive for small, fixed-size datasets.
  - However, operations like insertion and deletion, especially at the front or middle, are more complex due to the need to shift elements.
- **Linked Lists**:
  - Linked lists are more challenging to implement because you need to manage pointers, which adds complexity (e.g., handling `NULL` pointers).
  - However, for scenarios requiring frequent additions and deletions, linked lists are more efficient and intuitive once the pointer logic is clear.

Advantages and Disadvantages:

- **Arrays**:
  - **Advantages**:
    - Efficient random access with constant time complexity O(1).
    - Simple to implement for fixed-size data with minimal pointer management.
  - **Disadvantages**:
    - Fixed size can lead to memory waste or the need for resizing.
    - Inserting or deleting patients at the beginning or middle is inefficient because of the need to shift elements.
- **Linked Lists**:
  - **Advantages**:
    - Dynamic size allows easy memory management, with no need to preallocate space.
    - Efficient insertion and deletion operations at both ends of the list.
  - **Disadvantages**:
    - Random access is slow, with time complexity O(n).
    - More complex implementation with extra memory overhead due to pointers.

When to Use Linked Lists vs Arrays:

- **When Linked Lists are More Suitable**:
  - If patient records are frequently added or removed from the middle or the beginning of the list.
  - When the total number of patients is unknown or constantly changing, making a dynamic structure preferable.
  - When memory should be allocated as needed (e.g., in scenarios with fluctuating patient data in a hospital).
- **When Arrays are More Suitable**:
  - If patient records are static or only modified at the end, such as appending new patients to a waiting list.
  - If constant-time access to patient records is needed (e.g., if you need to frequently access patients by their index).
  - When you have a fixed number of patients, and memory usage is predictable.

Conclusion:

- Arrays are faster for accessing patient records directly and for operations at the end of the list. However, they are less efficient for dynamic datasets with frequent insertions and deletions.
- Linked lists are more flexible for dynamic datasets, with better performance for insertions and deletions at any position, but they come with the cost of slower access and additional memory usage for pointers.

In a hospital management system where patient data frequently changes, linked lists might be more suitable due to their flexibility and ease of handling dynamic sizes. Arrays could be used if fast access by index is needed and the patient data doesn't change frequently.