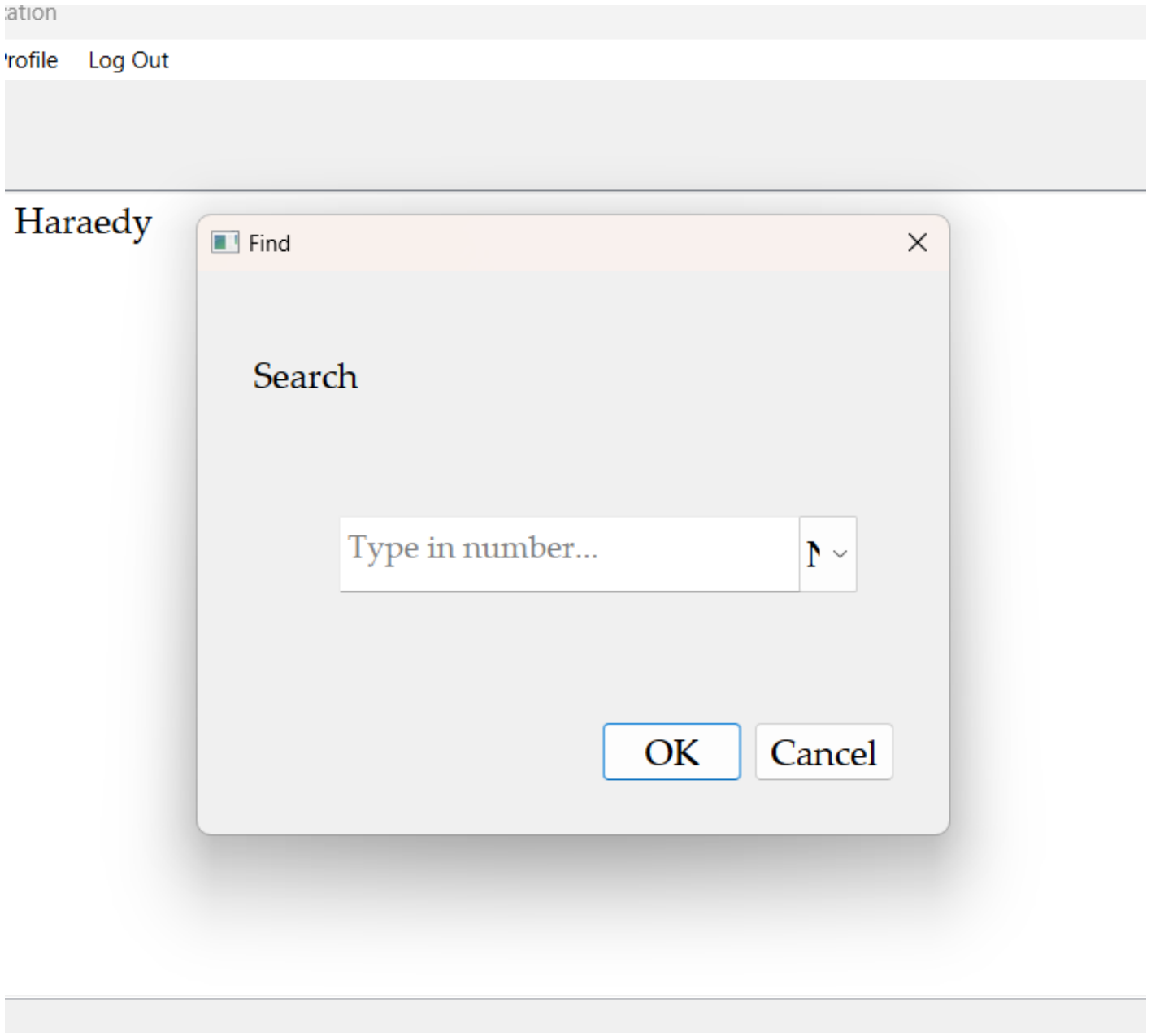
Implementation of Data Structure to a contacts app

Contact App List

We have decided to make a contact app list using C++, so we looked up the fastest and most efficient ways to store, sort, delete and prompt data



1- Vectors

Vectors are used in hash tables to provide flexibility, dynamic resizing, and efficient collision handling, making them a practical choice for implementing separate chaining and managing variable-sized datasets

```
    if (size >= N) {

        resize( new_N: (int (1.5*N)) + 7);
    }

    cntct = new T[N];

    move(cntct_elems.begin(), cntct_elems.end(), cntct);
}

~Vector() {

    delete[] cntct;
}

void resize(int new_N) {

    T* new_cntct = new T[new_N];

    for (int i { }; i < size; ++i) {

        new_cntct[i] = move(cntct[i]);
    }

    delete[] cntct;

    cntct = new_cntct;
    N = new_N;
}

void push(const initializer_list<T>& cntct_elems) {

    if ((size + cntct_elems.size()) >= N) {

        resize( new_N: (int (1.5*N)) + 7);
    }

    for (auto i : cntct_elems) {
```

The data structures we used

2- Hash tables

A hash table is a data structure used to store and retrieve data efficiently using key-value pairs. It employs a hash function to map keys to specific indices in an array, enabling fast access to data.

- Efficiency: Provides O(1) average time complexity for insertion, deletion, and search operations.

- Key-Value Mapping: Data is stored based on unique keys, allowing quick lookups.

- Collision Handling: Uses techniques like chaining or open addressing to resolve hash conflicts.

```
namespace HashTableNamespace {
    template<class DataType>
    class HashTable {
    private:
        vector<Node<DataType>*> table;
        int capacity;

        int hashFunction(int key) const;

    public:
        // Constructor
        HashTable(int size = 10);


        // methods
        bool remove(int key);
        DataType search(int key) const;
        void insert(int key, const DataType& value);

        void display() const;


        // Destructor
        ~HashTable();
    };
} // namespace HashTable

#include "../source/HashTable.cpp" // Include the implementation

#endif // HASH_TABLE_HPP
```

3- Stack implementation

In the contacts app, the stack is utilized as a data structure to handle specific tasks that require Last In, First Out (LIFO) operations.

Key Uses of Stack in the App:

Navigation or Backtracking: The stack can store previously accessed contacts, allowing the user to move back through recent searches or views.

```
    int N { inpt_N > MAX_SIZE ? MAX_SIZE : inpt_N }, size { }, top { -1 };

    void resize(int new_N) {
        T* new_cntct = new T[new_N];

        for (int i = 0; i < size; ++i) {

            new_cntct[i] = move(cntct[i]);
        }

        delete[] cntct;

        cntct = new_cntct;
        N = new_N;
    }

public:
    Stack() {

        cntct = new T[N];
    }

    ~Stack() {

        delete[]
    }

    void push(T&& value) {

        if (size >= N) {
            resize((int) N * 1.5)) + 1);
        }

        cntct[++top] = std::move(value);
        ++size;
    }
```

Why we chose those structures?

- Hash tables on average can offer O(1) , they are great for saving time in big programs, making it wise to use.
- Vectors memory management is great, they increase in size when needed, they store elements contiguously, meaning they don’t have gaps, making it a great choice for our app
- Vectors handle memory internally and grow when needed
- A stack allows data to be added (push) and removed (pop) in reverse order of insertion, making it ideal for tasks like backtracking
- The stack time complexity is efficient, those functions have O(1) time complixity, giving us the least time possible

Why didn’t we choose other structures?

- Linked list was our initial choice for handling collisions, but we found out vectors are much better and faster
- A linked list does not store anything, it only stores a pointer, and then whatever you plan to give it exactly. It has terrible random access (O(n))
- Linked lists require frequent dynamic memory allocations and deallocations for each node, which can be slower and more error-prone
- Array wasn’t used because of its fixed size, making it a bad choice for dealing with a lot numbers
- Queue would be a bad choice since it’s designed for ins and outs with order, giving us less flexibility to work with

Vectors VS Linked list

	Vector	Linked List
Memory Layout	Contiguous	Non-contiguous
Access Time	O(1)	O(n)
Insertion (Beginning)	O(n)	O(1)
Insertion (End)	O(1) amortized	O(n)
Deletion (Beginning)	O(n)	O(1)
Deletion (End)	O(1) amortized	O(n)
Memory Overhead	Minimal	High (due to pointers)
Cache Efficiency	High	Low
Dynamic Resizing	Automatic	Manual (via pointers)

4- Deletion

In the contacts app, the deletion operation is designed to efficiently remove contacts based on user input, ensuring data integrity and ease of use.

Key Features of Deletion Operation:

Search and Remove: The app first searches for the contact using a key, such as a name or phone number, and removes it if found.

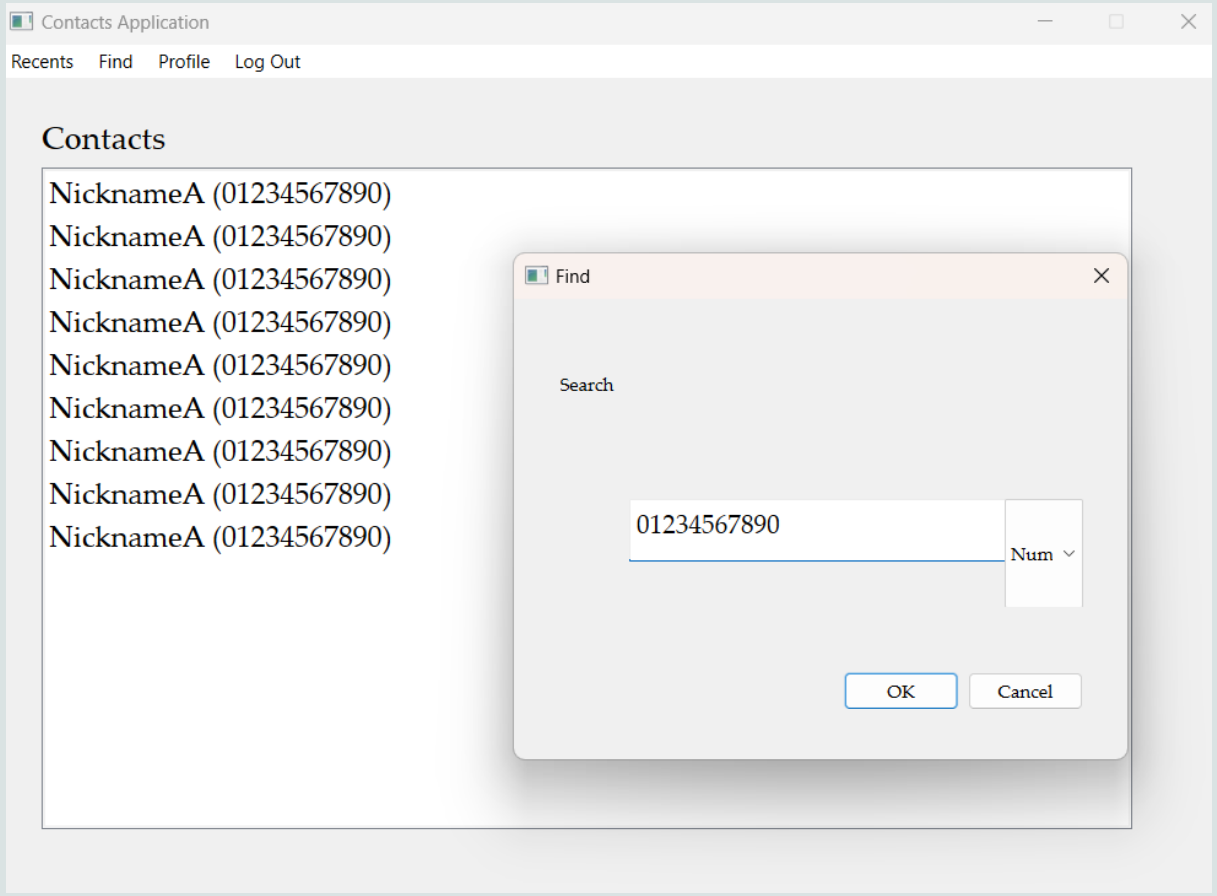
```
// remove index
void remove(const int& index){
    for (int x = index+1; x < size; x++){
       cntct[x-1] = cntct[x];
    }
    size--;
}




// this will give you time-constant access
// but we can implement display function for this.
// Overload operator[] for non-const access
T& operator[](int index) {
    if (index < 0 || index >= size) {
        throw std::out_of_range("Index out of range");
    }
    return cntct[index];
}

// Overload operator[] for const access
const T& operator[](int index) const {
    if (index < 0 || index >= size) {
        throw std::out_of_range("Index out of range");
    }
    return cntct[index];
}
```

App showcase



Mahros Mohamed 223106831

Kevin Joseph Magdy 223105947

Abdelrahman Ahmed 223102769

Sherif Mohamed 223107334

Saif Amer Mohammed 223105417

Thank you

References:

https://github.com/elqabasy/HashTable
https://github.com/winter-semester-projects/contactsApp