University M’hamed Bougara Boumerdes

Institute of Electrical and Electronics Engineering (ex: INELEC)

Algorithms and Data structures laboratory

LAB N°2

Saturday, March 5, 2022

Superviser:

Dr. A. Zitouni

Students:

Madaoui Zakaria	G3 Computer
Maallem Nassim	G3 Computer

Objectives:

- Write a C++ List of names data structure using array implementation
- Write a C++ List of names data structure using pointer implementation

Task 1: ArrayList implementation

In order to implement an array list of names we wrote a struct that represents the list. The struct keeps track of the list size and contains a fixed array for storing the names. After that we implemented few functions to add functionality to insert, delete, search, traverse forward and backward. Finally, we wrote a simple logic in the main() function to display a menu that allows the user to use all the previously built functionality. This was the resulting code:

```
#include <iostream>
#include <cstring>
using namespace std;

#define MAX_ELEMENTS 100
#define MAX_NAME_SIZE 20
#define DIRECTION_FORWARD 1
#define DIRECTION_BACKWARD 0

#include <iostream>
#include <cstring>
using namespace std;

#define MAX_ELEMENTS 100
#define MAX_NAME_SIZE 20
#define DIRECTION_FORWARD 1
#define DIRECTION_BACKWARD 0

//Arraylist struct
typedef struct
{
    unsigned size = 0;
    char data[MAX_ELEMENTS][MAX_NAME_SIZE];
} ArrayList;

//Arraylist prototypes
void list_insert(ArrayList &list, char item[], unsigned position);
void list_remove(ArrayList &list, unsigned position);
int list_search(const ArrayList &list, const char *item);
void list_traverse(const ArrayList &list, unsigned direction);

int main()
{
    char user_option = '\0';
    char temp_str[MAX_NAME_SIZE];
    unsigned temp_position = 0;
    ArrayList l;

    while (user_option != 'q')
    {
        cout << "> Please select an option to proceed: \n"
             << "    i: insert new element\n"
             << "    d: delete an element \n"
             << "    s: search for index of an element \n"
             << "    f: traverse list forward \n"
             << "    b: traverse list backward \n"
             << "    q: to quit the program \n";
        cin >> user_option;

        switch (user_option)
        {
        case 'i': /*insert*/
            cout << "> type the position & name to insert: ";
            cin >> temp_position >> temp_str;
            list_insert(l, temp_str, temp_position);
            break;

        case 'd': /*delete*/
            cout << "> type item position to delete: ";
            cin >> temp_position;
            list_remove(l, temp_position);
            break;

        case 's': /*serach and get index*/
            cout << "> type the name to search for: ";
            cin >> temp_str;
            cout << temp_str << " index is : " << list_search(l, temp_str) << "\n";
            break;

        case 'f': /*traverse forward*/
            list_traverse(l, DIRECTION_FORWARD);
            break;

        case 'b': /*traverse backward*/
            list_traverse(l, DIRECTION_BACKWARD);
            break;

        case 'q': /*quit*/
            cout << "Quitting program ... ";
            break;

        default:
            cout << "Wrong input ! try again !\n";
            break;
        }
        cout << "\n-------------------------------------------\n\n";
    }
    return 0;
}

//prototypes implementation
void list_insert(ArrayList &list, char item[], unsigned position)
{
    if (position >= MAX_ELEMENTS)
        cout << "Index out of bound ! position must be less than " << MAX_ELEMENTS <<
"\n";
    else if (list.size == MAX_ELEMENTS)
        cout << "List is full !\n";
    else
    {
        for (int i = list.size; i > position; i--)
        {
            strcpy(list.data[i], list.data[i - 1]);
        }
        strcpy(list.data[position], item);
        list.size++; //incerement size
    }
}

void list_remove(ArrayList &list, unsigned position)
{
    if (position >= list.size)
        cout << "element in position " << position << " doesn't exist!\n";
    else
    {
        for (int i = position; i < list.size - 1; i++)
        {
            strcpy(list.data[i], list.data[i + 1]);
        }
        list.size--; //decerement size
    }
}

void list_traverse(const ArrayList &list, unsigned direction)
{
    if (list.size == 0)
        cout << "List is empty\n";
    else if (direction == DIRECTION_FORWARD)
    {
        for (int i = 0; i < list.size; i++)
        {
            cout << "name[" << i << "]= " << list.data[i] << "\n";
        }
    }
    else
    {
        for (int i = list.size - 1; i >= 0; i--)
        {
            cout << "name[" << i << "]= " << list.data[i] << "\n";
        }
    }
}

int list_search(const ArrayList &list, const char *item)
{
    for (int i = 0; i < list.size; i++)
    {
        if (strcmp(item, list.data[i]) == 0)
            return i;
    }
    return -1;
}
```

Task 2: LinkedList (pointer) implementation

In order to implement a linked list of names we wrote a struct that represents a node. Each node contains a **c-string** , a pointer to the next node in the list , and a pointer to the previous node in the list. This way we have what is called a **Doubly Linked List** and that makes traversing the list in both directions very efficient. We also wrote another stuct called **LinkedList** which acts as a wrapper for the list and it keeps track of the list size, first and last node. After that we implemented similar functions as before to add the functionality to insert, delete, search, traverse forward and backward. Finally, we used the same logic in the main() function to display a menu that allows the user to use all the previously built functionality. This was the resulting code:

```
#include <iostream>
```

```cpp
#include <cstring>
using namespace std;

#define MAX_ELEMENTS 100
#define MAX_NAME_SIZE 20
#define DIRECTION_FORWARD 1
#define DIRECTION_BACKWARD 0

struct node{
    char data[MAX_NAME_SIZE];
    struct node* next = nullptr;
    struct node* prev = nullptr;
};

typedef struct node node;

//LinkedList wrapper struct
typedef struct
{
    unsigned size = 0;
    node* first = nullptr;
    node* last  = nullptr;
} LinkedList;

//Arraylist prototypes
void list_insert(LinkedList &list, char item[], unsigned position);
void list_remove(LinkedList &list, unsigned position);
int  list_search(const LinkedList &list, const char *item);
void list_traverse(const LinkedList &list, unsigned direction);

int main()
{
   //same as previous main
}

//prototypes implementation
void list_insert(LinkedList &list, char item[], unsigned position)
{

    node *temp_1 = list.first;

    if( position > list.size || position < 0){ /*Wrong input*/
        cout << "no such position \n";
        return; //do not increment, retuern directly
    }
    else if (list.first == nullptr && position == 0){ /*inserting for first time*/
            list.first = new node;                // create new node
            strcpy(list.first->data, item);     // put data in new node
            list.first->next = list.first->prev = nullptr;
    }
    else if(position == 0){ /*inserting at index 0*/
        node *temp_2 = new node;     // create new node
        strcpy(temp_2->data, item);  // put data in new node
        temp_2->prev = nullptr;
        temp_2->next = temp_1;
        temp_1->prev = temp_2;
        list.first    = temp_2;         //keep track of first element
    }
    else{ /*inserting at index greater than 0 */
        for (int i = 1; i < position ; i++) /*get node[position-1]*/
        {
            temp_1 = temp_1->next;
        }
        node *temp_2 = new node;     // create new node
        strcpy(temp_2->data, item);  // put data in new node
        temp_2->next = temp_1->next;
        temp_2->prev = temp_1;
        temp_1->next = temp_2;

        if (position == list.size) list.last = temp_2; //track last element aswell
        else temp_2->next->prev = temp_2;              // if next element is not null
correct it's backward link
    }

    list.size++; //incerement size
}

void list_remove(LinkedList &list, unsigned position)
{
    node *temp_1 = list.first;

    if( position >= list.size || position < 0){ /*Wrong input*/
        cout << "no such position \n";
        return; //do not decrement, retuern directly
    }
    else if(position == 0){ /* removing element at index 0*/
        node *temp_2 = temp_1->next;
        list.first = temp_2; //keep track of first element when deleting
        delete temp_1;

        if(list.size != 1) temp_2->prev = nullptr; //correct back link if first
element is not the last
    }
    else{ /*deleting element at index greater than 0 */
        for (int i = 0; i < position ; i++) /*get node[position]*/
        {
            temp_1 = temp_1->next;
        }
        node *temp_0 = temp_1->prev;     // get prev node pointer
        node *temp_2 = temp_1->next;     // get next node pointer
        temp_0->next = temp_2;
        delete temp_1;

        if (position == (list.size-1)) list.last = temp_0; //keep track of last when
deleting
        else temp_2->prev = temp_0;     //if next element isn't null correct it's back
link
    }

    list.size--; //decerement size
}

void list_traverse(const LinkedList &list, unsigned direction)
{
    int index;
    if(list.size == 0) cout << "List is empty\n";
    else if (direction == DIRECTION_FORWARD)
    {
        node* temp = list.first;
        index = 0;
        while (temp != nullptr)
        {
            cout << "element[" << index++ << "]= " << temp->data << "\n";
            temp = temp->next;
        }

    }
    else
    {
        node* temp = list.last;
        index = list.size;
        while (temp != nullptr)
        {
            cout << "element[" << --index << "]= " << temp->data << "\n";
            temp = temp->prev;
        }
    }
}

int list_search(const LinkedList &list, const char *item)
{
    node* temp = list.first;
    int index  = 0;
    while (temp != nullptr)
    {
        if(strcmp(temp->data, item) == 0) return index;
        temp = temp->next; //traverse forward
        index++;
    }

    return -1;
}
```

Conclusion:

By the end of this experiment, we come to the conclusion that an array implementation of a list is very simple to write (required less logic), but it imposes the restriction of fixed size. On the other hand, the pointer implementation is more complex and takes more memory (next pointer and name) but it allows for a flexible size and allocates memory only when needed.