**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра информационных систем**

отчет

**по практической работе №2**

**по дисциплине «Программирование»**

Тема:

|  |  |  |
| --- | --- | --- |
| Студент гр. |  | Полуянов В. Н. |
| Преподаватель |  | Глущенко А. Г. |

Санкт-Петербург

2023

**Цель работы.**

Изучение свойств и организация двусвязных списков; получение практических навыков в работе с динамическими массивами и двусвязными списками; проведение сравнительной характеристики скорости вставки, получения и удаления элементов из них

**Основные теоретические положения.**

Одномерный однонаправленный список представляет собой совокупность отдельных элементов, каждый из которых содержит две части – информационную и адресную. Информационная часть предназначена для хранения полезных данных и может иметь практически любой тип. Адресная часть каждого элемента содержит адрес следующего элемента списка. Для работы со списком достаточно знать адрес его первого элемента. Зная адрес первого элемента списка, можно последовательно получить доступ к любому другому его элементу.

Одним из недостатков односвязных элементов является то, что узел имеет указатель только на следующий элемент. Вернуться из текущего элемента к предыдущему невозможно

Каждый узел двунаправленного линейного списка содержит два поля указателей – на следующий и на предыдущий узлы. Указатель на предыдущий узел корня списка содержит нулевое значение. Указатель последнего узла также содержит нулевое значение. Такая организация списка позволяет перемещаться по его элементам в двух направлениях

**Постановка задачи.**

Необходимо создать двусвязный линейный список и соответствующий ему динамический массив.

Написать функции, реализующие операции с двусвязными списками:

1. Формирование двусвязного списка размерности N.
2. Вставка элемента по значению или индексу.
3. Удаление элемента по значению или индексу.
4. Получение элемента списка по значению или индексу.
5. Обмен элементов двусвязного списка местами.
6. Сравнение скорости выполнения п. 1-5 с скоростью работы с динамическим массивом.

**Выполнение работы.**

Код программы представлен в приложении А.

1. При запуске программы пользователю выводится подсказка команды, выводящей меню доступных функций. Ожидается ввод команды с клавиатуры. (рис. 1).



Рисунок . Главное меню

1. Следующий шаг зависит от введенной команды, если пользователь ввёл:
   1. “0”, то выполнение программы завершается.
   2. “1”, то открывается меню создания/пересоздания списка (рис. 2).

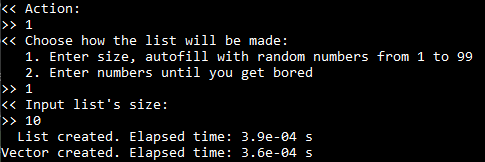


Рисунок 2. Задача 1

* 1. “2”, то открывается меню вставки элемента в двусвязный список (рис. 3).

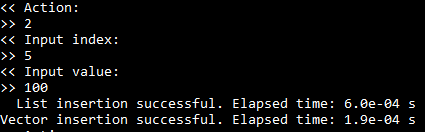


Рисунок 3. Задача 2

* 1. “3”, то выводятся все имеющиеся записи (рис. 4).

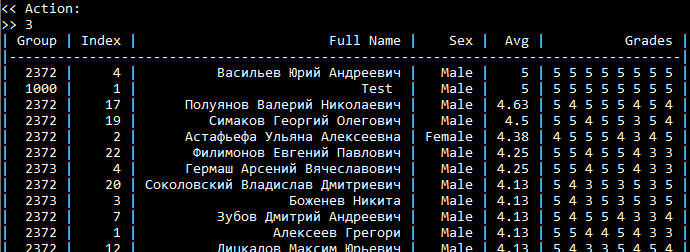


Рисунок 4. Задача 3

* 1. “4”, то открывается меню удаления элемента по индексу или по значению (рис. 5).

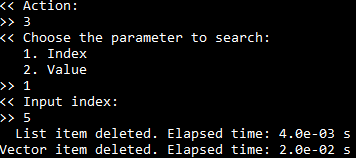


Рисунок 5. Задача 4

* 1. “5”, то открывается меню обмена элементов местами (рис. 6).

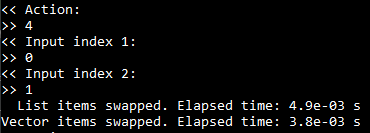


Рисунок 6. Задача 5

* 1. “i”, то открывается меню ИДЗ (вариант №16) (рис. 7).

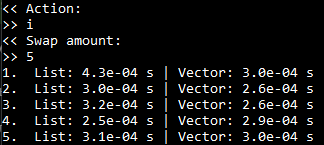


Рисунок 7. Задача 6

* 1. “h”, то выводится меню доступных команд (рис. 8).

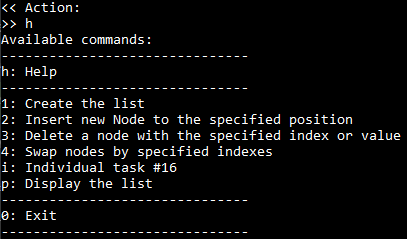


Рисунок 8. Меню

* 1. Любой символ или набор символов, отличный от существующий команд, то будет выведено сообщение об ошибке (рис. 9).



Рисунок 9. Неизвестная команда

**Выводы.**

В ходе работы были изучены свойства и организация двусвязных линейных списков. Получены практические навыки работы с динамическими массивами и двусвязными списками. Проведена сравнительная характеристика скорости вставки, получения и удаления элементов для контейнеров на 100000 элементов (табл. 1).

|  |  |  |
| --- | --- | --- |
|  | Двусвязный список | Динамический массив |
| Вставка, мс. | 33 | 1 |
| Поиск, мс. | 52 | 320 |
| Удаление, мс. | 40 | 230 |

Приложение А

рабочий код

#include <iostream>  
#include <cstdlib>  
#include <iomanip>  
#include <chrono>  
#include <vector>  
  
  
*/// Node structure represents a node in a Doubly-Linked List*struct Node{  
 int value;  
 struct Node \*next;  
 struct Node \*prev;  
};  
  
  
*/// Reads the integer input via cin*bool inputInt(int &variable, bool isUnsigned = false) {  
 std::cin >> variable;  
 if (std::cin.fail() || (isUnsigned && variable < 0)) {  
 std::cout << "Invalid input\n";  
 std::cin.clear();  
 std::cin.ignore();  
 return false;  
 }  
 return true;  
}  
  
  
*/// Print the DLL to the console*void printList(struct Node \* head) {  
 if (!head) {  
 std::cout << "List is empty\n";  
 return;  
 }  
 for ( ; head; head = head->next) std::cout << head->value << ' ';  
 std::cout << std::endl;  
}  
  
  
*/// Free the memory allocated for a DLL*void deleteList(struct Node \* &head) {  
 while (head) {  
 struct Node \*temp = head;  
 head = head->next;  
 delete temp;  
 }  
 head = nullptr;  
}  
  
  
*/\*\*  
 \* Creates a DLL of the specified size filled with randint(0, 99) and returns a pointer to the head node  
 \* @param size - List size  
 \* @return Pointer to the first element of the list  
 \*/*struct Node \* createList(unsigned size) {  
 if (size == 0) return nullptr; *// No size - no list* struct Node \*current;  
 struct Node \*next = nullptr;  
  
 *// Creates DLL starting from the LAST node* for (int i = 0; i < size; ++i) {  
 current = new struct Node;  
 current->value = 1 + std::rand() % 99;  
  
 current->next = next; *// Point the current node's next pointer to the previous one* if (next) next->prev = current; *// If there is a previous node, point its prev pointer to the current one* next = current; *// Update the pointer* }  
 current->prev = nullptr;  
 return current; *// Return head*}  
  
  
*/\*\*  
 \* Creates a DLL from user input and returns a pointer to the head node  
 \* @return Pointer to the first element of the list or nullptr if input is wrong (or list is empty)  
 \*/*struct Node \* createListFromInput() {  
 struct Node \*head = nullptr;  
 struct Node \*tail = nullptr;  
  
 int input;  
 std::cout << "<< Enter the list elements separated by spaces (Enter 0 to finish):\n>> ";  
 while (true) {  
 if (!inputInt(input)) { *// Invalid input handler* if (head) deleteList(head);  
 return nullptr;  
 }  
 if (input == 0) return head; *// Return the first element of the DLL if 0 entered* auto \*current = new struct Node;  
 current->value = input;  
  
 current->next = nullptr; *// Next node is NULL by default* if (tail) { *// If node isn't the head, then update the pointers* tail->next = current;  
 current->prev = tail;  
 } else { *// If tail is null, then it is the first element; update the head* current->prev = nullptr;  
 head = current;  
 }  
 tail = current; *// Update tail* }  
}  
  
  
*/\*\*  
 \* Finds node by it's index in the DLL  
 \* @param head - The first element of the DLL  
 \* @param pos - Node to find index  
 \* @return Pointer to the found node  
 \*/*struct Node \* getItemByIndex(struct Node \* head, unsigned pos) {  
 if (!head) { *// Empty list check* std::cout << "List is empty\n";  
 return nullptr;  
 }  
  
 *// Traverse the list to the index or return index out of range* for (unsigned i = 0; i < pos; ++i) {  
 if (!head->next) {  
 std::cout << "Index out of range\n";  
 return nullptr;  
 }  
 head = head->next;  
 }  
  
 return head;  
}  
  
  
*/\*\* Finds first occurrence of node by it's value in the DLL  
 \*  
 \*/* struct Node \* getItemByValue(struct Node \* head, int value) {  
 if (!head) { *// Empty list check* std::cout << "List is empty\n";  
 return nullptr;  
 }  
  
 *// Traverse the list while not found or node exists* while (head) {  
 if (head->value == value) break;  
  
 if (!head->next) {  
 std::cout << "Item not found\n";  
 return nullptr;  
 }  
 head = head->next;  
 }  
  
 return head;  
 }  
  
  
*/\*\*  
 \* Inserts node at the specified position in DLL  
 \* @param head - The first element of the DLL (may be changed)  
 \* @param pos - Insertion index  
 \* @param value - Node value  
 \* @return true if insertion done successful else false  
 \*/*bool insertItem(struct Node \* &head, unsigned pos, int value) {  
 *// Find the current node* struct Node \* found = getItemByIndex(head, pos);  
 if (!found) return false;  
  
 *// Insert the new node* auto \*newNode = new struct Node;  
 newNode->value = value;  
 newNode->prev = found->prev;  
 newNode->next = found;  
 if (newNode->prev) newNode->prev->next = newNode;  
 if (newNode->next) newNode->next->prev = newNode;  
  
 if (pos == 0) head = newNode; *// Update the head if necessary* return true;  
}  
  
  
*/\*\*  
 \* Deletes the node with a specified index  
 \* @param head - First element of the DLL, could be changed in process  
 \* @param pos - Index of deletable node  
 \* @return true if deleted and false if element not found  
 \*/*bool deleteItem(struct Node \* &head, unsigned pos) {  
 if (!head) return false; *// If the list is empty, no deletion is possible  
  
 // Find the node* struct Node \*found = getItemByIndex(head, pos);  
 if (!found) return false;  
  
 *// Delete the node* found->prev ? found->prev->next = found->next : head = found->next; *// Update prev node or update the head if null* found->next ? found->next->prev = found->prev : nullptr; *// Update next node or list deleted if null* delete found;  
  
 return true;  
}  
  
*/\*\*  
 \* Deletes the node with a specified value  
 \* @param head - First element of the DLL, could be changed in process  
 \* @param value - Value of deletable node  
 \* @return true if deleted and false if element not found  
 \*/*bool deleteItemByValue(struct Node \* &head, int value) {  
 if (!head) return false; *// If the list is empty, no deletion is possible  
  
 // Find the node* struct Node \*found = getItemByValue(head, value);  
 if (!found) return false;  
  
 *// Delete the node* found->prev ? found->prev->next = found->next : head = found->next; *// Update prev node or update the head if null* found->next ? found->next->prev = found->prev : nullptr; *// Update next node or list deleted if null* delete found;  
  
 return true;  
 }  
  
  
*/\*\*  
 \* Swaps the node1 and node2 elements of doubly-linked list by indexes  
 \* @param head - First element of the DLL, is needed cuz it can change in the process  
 \* @param i1 - Index of the first element to be swapped  
 \* @param i2 - Index of the second element to be swapped  
 \* @return true if elements are swapped else false  
 \*/*bool swapByIndex(struct Node \* &head, unsigned i1, unsigned i2) {  
 *// If list is empty, has only one element, or indexes are the same, then no swapping required* if (!head || !head->next || i1 == i2) return false;  
 if (i1 > i2) std::swap(i1, i2); *// Index1 should be less than index2* struct Node \*node1, \*node2;  
  
 *// Traverse the list until the nodes to be swapped are found* unsigned index = 0;  
 struct Node \*temp = head;  
 while (temp) {  
 if (index == i1) node1 = temp;  
 else if (index == i2) node2 = temp;  
 *// If the second index is out of range, then throw an error message and return false* if (!temp->next && i2 > index) {  
 std::cout << "List item not found\n";  
 return false;  
 }  
 temp = temp->next;  
 index += 1;  
 }  
  
 *// If node1 is the head node, update the head to node2* if (node1 == head) head = node2;  
  
 *// Swap the next pointers of the nodes* temp = node1->next;  
 node1->next = node2->next;  
 node2->next = temp;  
 *// And update the prev pointers of the next node if it exists* if (node1->next) node1->next->prev = node1;  
 if (node2->next) node2->next->prev = node2;  
  
 *// Swap the prev pointers of the nodes* temp = node1->prev;  
 node1->prev = node2->prev;  
 node2->prev = temp;  
 *// And update the next pointer of the previous node if it exists* if (node1->prev) node1->prev->next = node1;  
 if (node2->prev) node2->prev->next = node2;  
  
 return true;  
}  
  
  
*/// Gets the start time\_point and prints the duration\_cast(now-start) in scientific format*void printTimeDurationCast(auto start, bool isEndOfLine = true) {  
 auto end = std::chrono::steady\_clock::now();  
 auto elapsed = std::chrono::duration\_cast<std::chrono::nanoseconds>(end - start);  
 std::cout << std::scientific << std::setprecision(1);  
 std::cout << elapsed.count() / 1e9 << " s";  
 if (isEndOfLine) std::cout << std::endl;  
 std::cout << std::defaultfloat;  
}  
  
  
int main() {  
  
 struct Node \*head = nullptr; *// Doubly-linked list's head* std::vector<int> vector; *// Vector for time comparison  
  
 // Main loop* std::cout << "Enter 'h' to get list of commands\n";  
 while (true) {  
  
 *// Get command from user* char userAction;  
 std::cout << "<< Action:\n>> ";  
 std::cin >> userAction;  
  
 *// Error handler (i.e. more than one symbol input)* if (std::cin.fail() || std::cin.get() != '\n') {  
 std::cout << "RuntimeError: unknown command\n";  
 std::cin.clear();  
 std::cin.ignore(100000, '\n');  
 continue;  
 }  
  
 *// Exit command* if (userAction == '0') {  
 if (head) deleteList(head);  
 break;  
 }  
  
 switch (userAction) {  
  
 *// Create or recreate the list* case '1': {  
 *// List already exists. Ask for recreation* if (head) {  
 std::cout << "<< List already exists. Do you want to recreate it? [1/0]:\n>> ";  
 int input;  
 if (!inputInt(input)) { *// NaN error handler* continue;  
 } else if (input != 1 && input != 0) { *// Input is not [1/0]* std::cout << "That's illegal... Input just zero or one [1/0] next time\n";  
 continue;  
 } else if (input == 0) { *// Input is 0, goto userAction* std::cout << "Okay.\n";  
 continue;  
 }  
 deleteList(head); *// Else remove old list and create new* }  
  
 *// Get the way list will be created* std::cout << "<< Choose how the list will be made:\n";  
 std::cout << " 1. Enter size, autofill with random numbers from 1 to 99\n";  
 std::cout << " 2. Enter numbers until you get bored\n>> ";  
 int input;  
 if (!inputInt(input)) { *// NaN error handler* continue;  
 } else if (input != 2 && input != 1) { *// Input is not [2/1]* std::cout << "Nah, your input should be 2 or 1. Isn't it simple?\n";  
 continue;  
 }  
  
 *// Create the list* if (input == 1) {  
 int size;  
 std::cout << "<< Input list's size:\n>> ";  
 if (!inputInt(size, true)) continue;  
  
 auto start = std::chrono::steady\_clock::now();  
 head = createList(size);  
 if (!head) { *// Size == 0, list is empty* std::cout << "Just kidding, right? The list is empty.\n";  
 continue;  
 }  
 std::cout << " List created. Elapsed time: ";  
 printTimeDurationCast(start);  
  
 *// Compare with dynamic array* start = std::chrono::steady\_clock::now();  
 std::vector<int> compareArr(size);  
 std::cout << "Vector created. Elapsed time: ";  
 printTimeDurationCast(start);  
 } else {  
 head = createListFromInput();  
 if (!head) { *// No input, list is empty* std::cout << "Just kidding, right? The list is empty.\n";  
 continue;  
 }  
 }  
  
 vector.clear();  
 for (auto \*curr = head; curr; curr = curr->next) {  
 vector.push\_back(curr->value);  
 }  
  
 break;  
 }  
  
 *// Insert new node* case '2': {  
 if (!head) {  
 std::cout << "List is empty\n";  
 continue;  
 }  
  
 *// Get index & value and insert new node = value into index* int index, value;  
 std::cout << "<< Input index:\n>> ";  
 if (!inputInt(index, true)) continue;  
 std::cout << "<< Input value:\n>> ";  
 if (!inputInt(value)) continue;  
  
 auto start = std::chrono::steady\_clock::now();  
 if (!insertItem(head, index, value)) continue;  
 std::cout << " List insertion successful. Elapsed time: ";  
 printTimeDurationCast(start);  
  
 start = std::chrono::steady\_clock::now();  
 vector.insert(vector.begin() + index, value);  
 std::cout << "Vector insertion successful. Elapsed time: ";  
 printTimeDurationCast(start);  
  
 break;  
 }  
  
 *// Delete list node by index or by value* case '3': {  
 if (!head) {  
 std::cout << "List is empty\n";  
 continue;  
 }  
  
 *// Choose a way to search* std::cout << "<< Choose the parameter to search:\n";  
 std::cout << " 1. Index\n";  
 std::cout << " 2. Value\n>> ";  
 int input;  
 if (!inputInt(input)) { *// NaN error handler* continue;  
 } else if (input != 2 && input != 1) { *// Input is not [2/1]* std::cout << "Nah, your input should be 2 or 1. Isn't it simple?\n";  
 continue;  
 }  
  
 *// Delete the node* if (input == 1) {  
 int index;  
 std::cout << "<< Input index:\n>> ";  
 if (!inputInt(index, true)) continue;  
 auto start = std::chrono::steady\_clock::now();  
 if (deleteItem(head, index)) {  
 std::cout << " List item deleted. Elapsed time: ";  
 printTimeDurationCast(start);  
  
 start = std::chrono::steady\_clock::now();  
 vector.erase(vector.begin() + index);  
 std::cout << "Vector item deleted. Elapsed time: ";  
 printTimeDurationCast(start);  
 }  
 } else {  
 int value;  
 std::cout << "<< Input value:\n>> ";  
 if (!inputInt(value)) continue;  
 auto start = std::chrono::steady\_clock::now();  
 if (deleteItemByValue(head, value)) {  
 std::cout << " List item deleted. Elapsed time: ";  
 printTimeDurationCast(start);  
  
 start = std::chrono::steady\_clock::now();  
 vector.erase(std::remove(vector.begin(), vector.end(), value), vector.end());  
 std::cout << "Vector item deleted. Elapsed time: ";  
 printTimeDurationCast(start);  
 }  
 }  
  
 break;  
 }  
  
 *// Swap nodes* case '4': {  
 if (!head) {  
 std::cout << "List is empty\n";  
 continue;  
 }  
  
 *// Get indexes* int index1, index2;  
 std::cout << "<< Input index 1:\n>> ";  
 if (!inputInt(index1, true)) continue;  
 std::cout << "<< Input index 2:\n>> ";  
 if (!inputInt(index2, true)) continue;  
  
 *// Swap* auto start = std::chrono::steady\_clock::now();  
 if (swapByIndex(head, index1, index2)) {  
 std::cout << " List items swapped. Elapsed time: ";  
 printTimeDurationCast(start);  
 start = std::chrono::steady\_clock::now();  
 std::swap(vector[index1], vector[index2]);  
 std::cout << "Vector items swapped. Elapsed time: ";  
 printTimeDurationCast(start);  
 }  
  
 break;  
 }  
  
 *// Individual task* case 'i': {  
 *// Поменяйте k случайных узлов исходного двусвязного списка и k значений  
 // исходного динамического массива. Сравните скорость работы.  
 // k - введенное пользователем число (k должно быть меньше N)  
  
 // Get amount of swaps* int swapCount;  
 std::cout << "<< Swap amount:\n>> ";  
 if (!inputInt(swapCount, true)) continue;  
  
 int size = vector.size();  
 for (int i = 1; i <= swapCount; ++i) {  
 int index1, index2;  
 do {  
 index1 = std::rand() % size;  
 index2 = std::rand() % size;  
 } while (index1 == index2);  
  
 std::cout << i << '.';  
  
 auto start = std::chrono::steady\_clock::now();  
 swapByIndex(head, index1, index2);  
 std::cout << " List: ";  
 printTimeDurationCast(start, false);  
  
 start = std::chrono::steady\_clock::now();  
 std::swap(vector[index1], vector[index2]);  
 std::cout << " | Vector: ";  
 printTimeDurationCast(start);  
 }  
  
 break;  
 }  
  
 *// Print the list* case 'p': {  
 printList(head);  
 break;  
 }  
  
 *// Help* case 'h': {  
 std::cout << "Available commands:\n";  
 std::cout << std::setw(32) << std::setfill('-') << '\n';  
 std::cout << "h: Help\n";  
 std::cout << std::setw(32) << std::setfill('-') << '\n';  
 std::cout << "1: Create the list\n";  
 std::cout << "2: Insert new Node to the specified position\n";  
 std::cout << "3: Delete a node with the specified index or value\n";  
 std::cout << "4: Swap nodes by specified indexes\n";  
 std::cout << "i: Individual task #16\n";  
 std::cout << "p: Display the list\n";  
 std::cout << std::setw(32) << std::setfill('-') << '\n';  
 std::cout << "0: Exit\n";  
 std::cout << std::setw(32) << std::setfill('-') << '\n';  
 std::cout << std::setfill(' ');  
 break;  
 }  
  
 *// Unknown command error* default: std::cout << "RuntimeError: unknown command\n";  
 }  
 }  
  
 return 0;  
}