Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный технологический университет»

ФИТ

Кафедра Информационных систем и технологий 1курс

Группа 2-2

**Структуры данных**

Тема:

Сортировки. List

Лабораторная работа 12,13

**Выполнил:**

Волковец Сергей Николаевич

**Проверил:**

Белодед Николай Иванович

Оглавление

[Теоритические сведения 3](#_Toc193265094)

[Сортировка вставками 5](#_Toc193265095)

[Сортировка выбором 7](#_Toc193265096)

[Сортировка слиянием 8](#_Toc193265097)

[Быстрая сортировка 10](#_Toc193265098)

[List 11](#_Toc193265099)

[Задачи 18](#_Toc193265100)

# Теоритические сведения

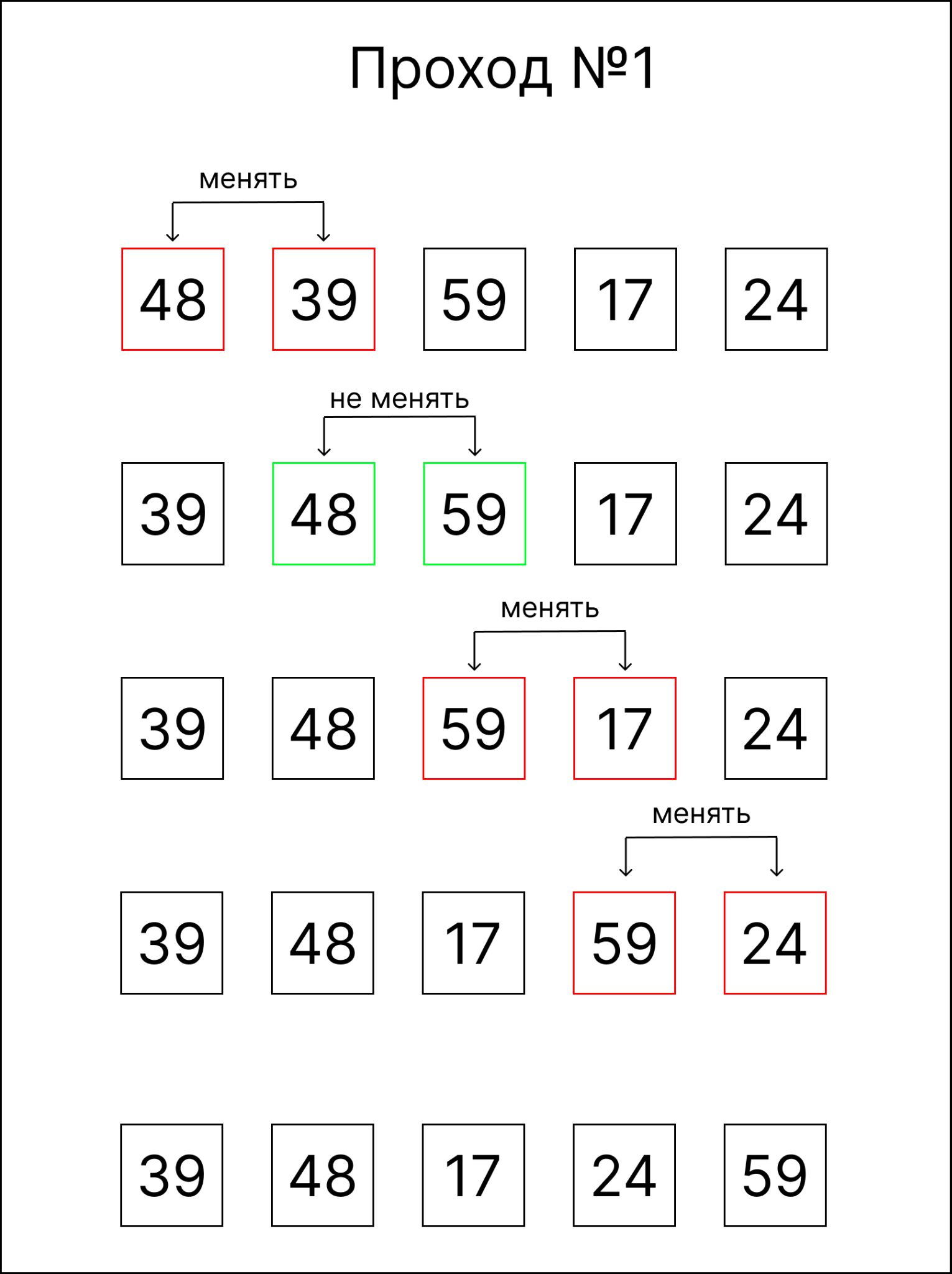
Сортировка - это процесс упорядочивания элементов в массиве или контейнере в соответствии с определенным критерием. Существует множество алгоритмов сортировки, каждый из которых имеет свои преимущества и недостатки.  
 Одной из основных причин использования сортировок является необходимость упорядочивания данных для более эффективного поиска и обработки. Например, если вы хотите найти определенный элемент в массиве, то поиск будет происходить гораздо быстрее, если элементы в массиве будут отсортированы по возрастанию или убыванию.  
 Сортировки также могут использоваться для оптимизации алгоритмов, которые работают с отсортированными данными. Например, бинарный поиск, который ищет элементы в отсортированном массиве, работает значительно быстрее, чем линейный поиск, который перебирает все элементы.  
 Кроме того, сортировки могут быть полезны для визуализации и анализа данных. Например, если вы хотите проанализировать распределение данных в массиве, то сортировка может помочь визуализировать эти данные в виде гистограммы или диаграммы.  
 Сортировки являются важным инструментом в программировании и могут помочь ускорить поиск и обработку данных, улучшить производительность алгоритмов и упростить анализ данных.

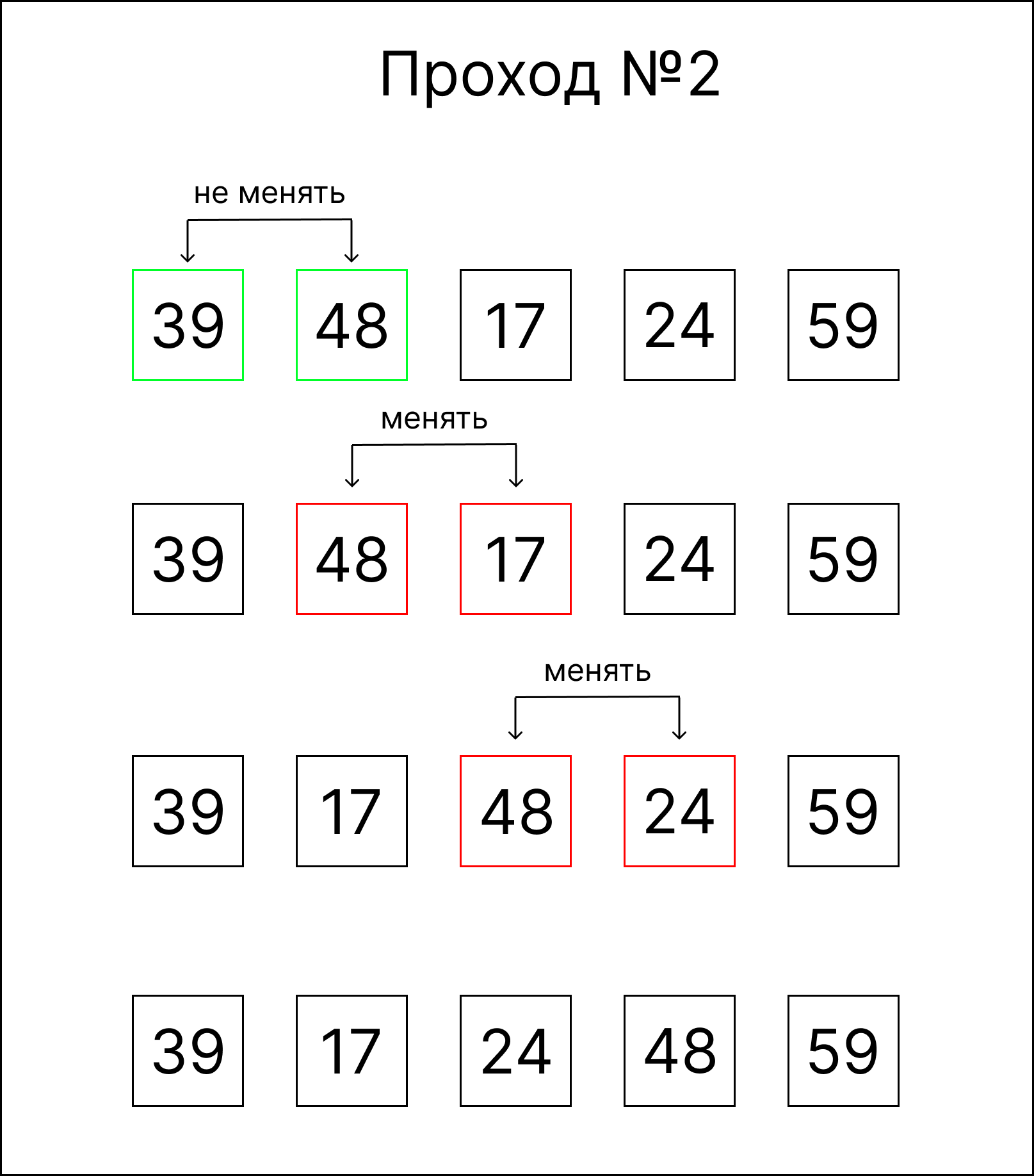
Сортировка пузырьком

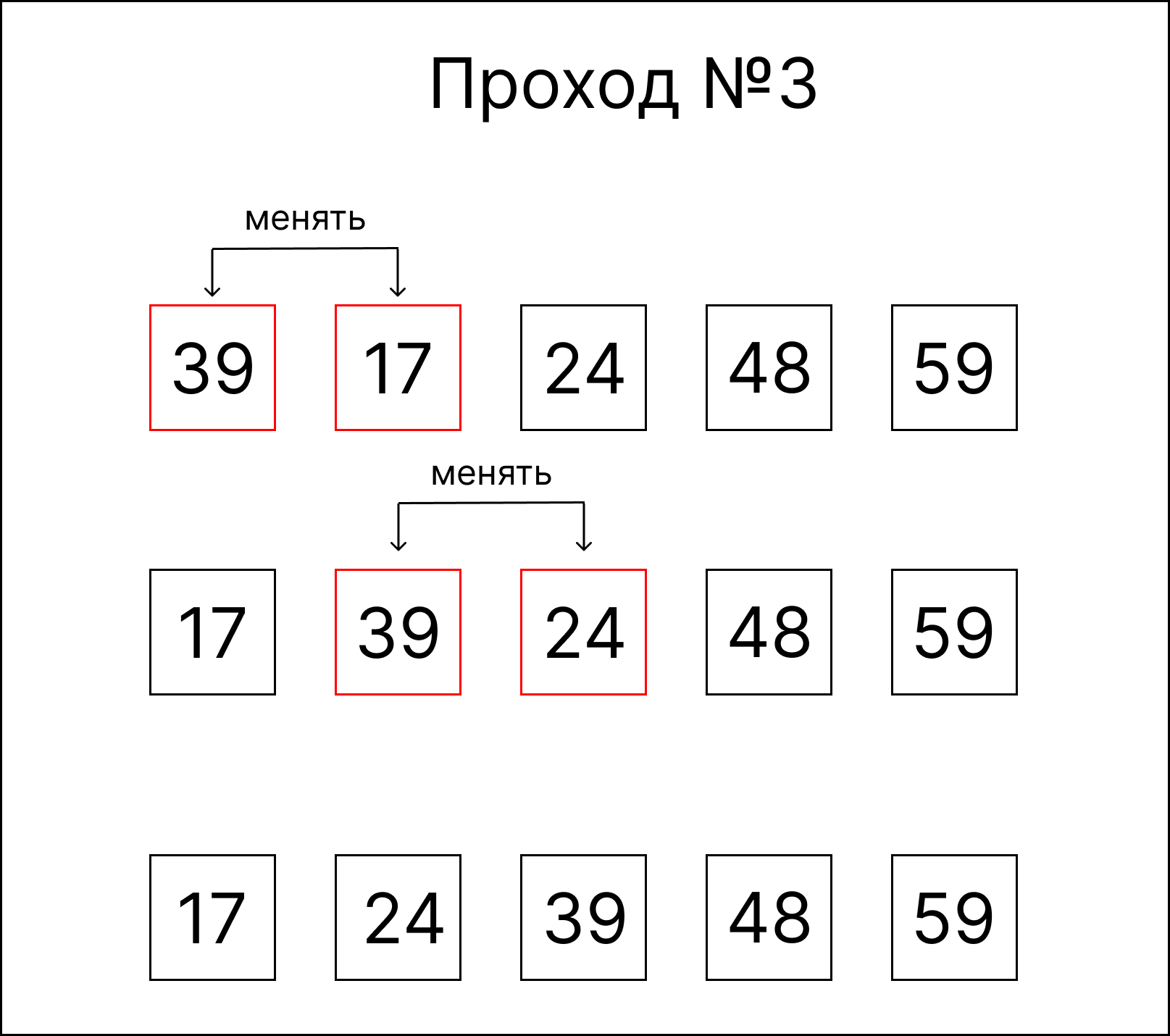
Алгоритм сортировки пузырьком (Bubble Sort) - это простой алгоритм сортировки, который работает путем сравнения и обмена соседних элементов в массиве. Алгоритм получил свое название из-за того, что большие элементы "всплывают" вверх массива, как пузырьки воды, во время выполнения сортировки.  
 Алгоритм сортировки пузырьком можно реализовать в нескольких вариантах, но основная идея остается неизменной. Вот пример реализации сортировки пузырьком:

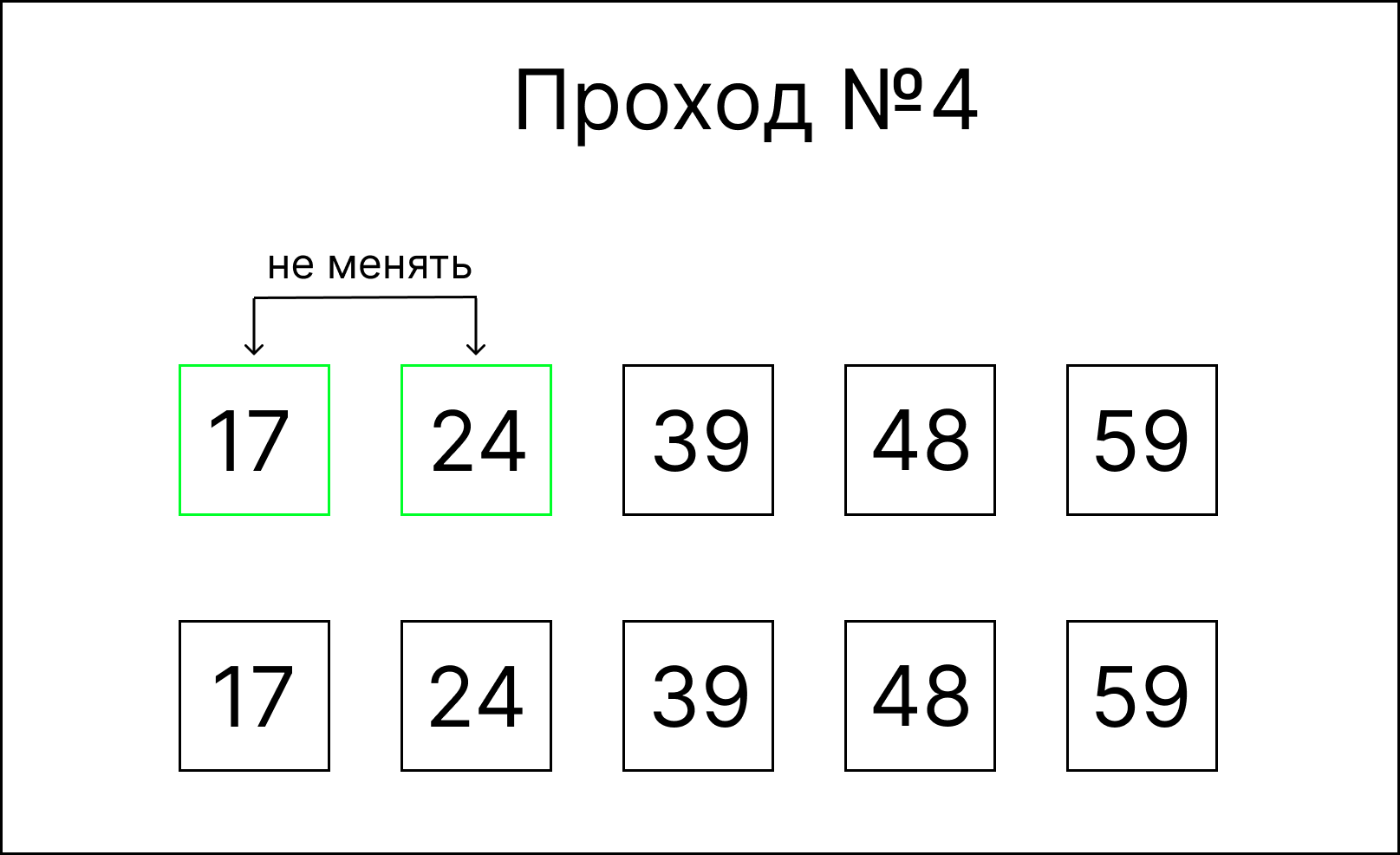
|  |
| --- |
| void bubbleSort(int arr[], int n) {  for (int i = 0; i < n - 1; i++) {  for (int j = 0; j < n - i - 1; j++) {  if (arr[j] > arr[j + 1]) {  int temp = arr[j];  arr[j] = arr[j + 1];  arr[j + 1] = temp;  }  }  } } |

В этом примере используются два вложенных цикла for для прохода по всем элементам массива и сравнения каждого элемента со своим соседом. Если элемент слева больше элемента справа, то мы меняем их местами.  
 Алгоритм сортировки пузырьком имеет сложность O(N^2), что делает его неэффективным для сортировки больших массивов. Однако, этот алгоритм все еще может быть полезным для сортировки небольших массивов или для обучения основам алгоритмов сортировки.  
 Алгоритм сортировки пузырьком - это простой и интуитивно понятный алгоритм сортировки, который может быть легко реализован на C++. Однако, из-за его низкой эффективности, в большинстве случаев лучше использовать более продвинутые алгоритмы сортировки, такие как быстрая сортировка или сортировка слиянием.







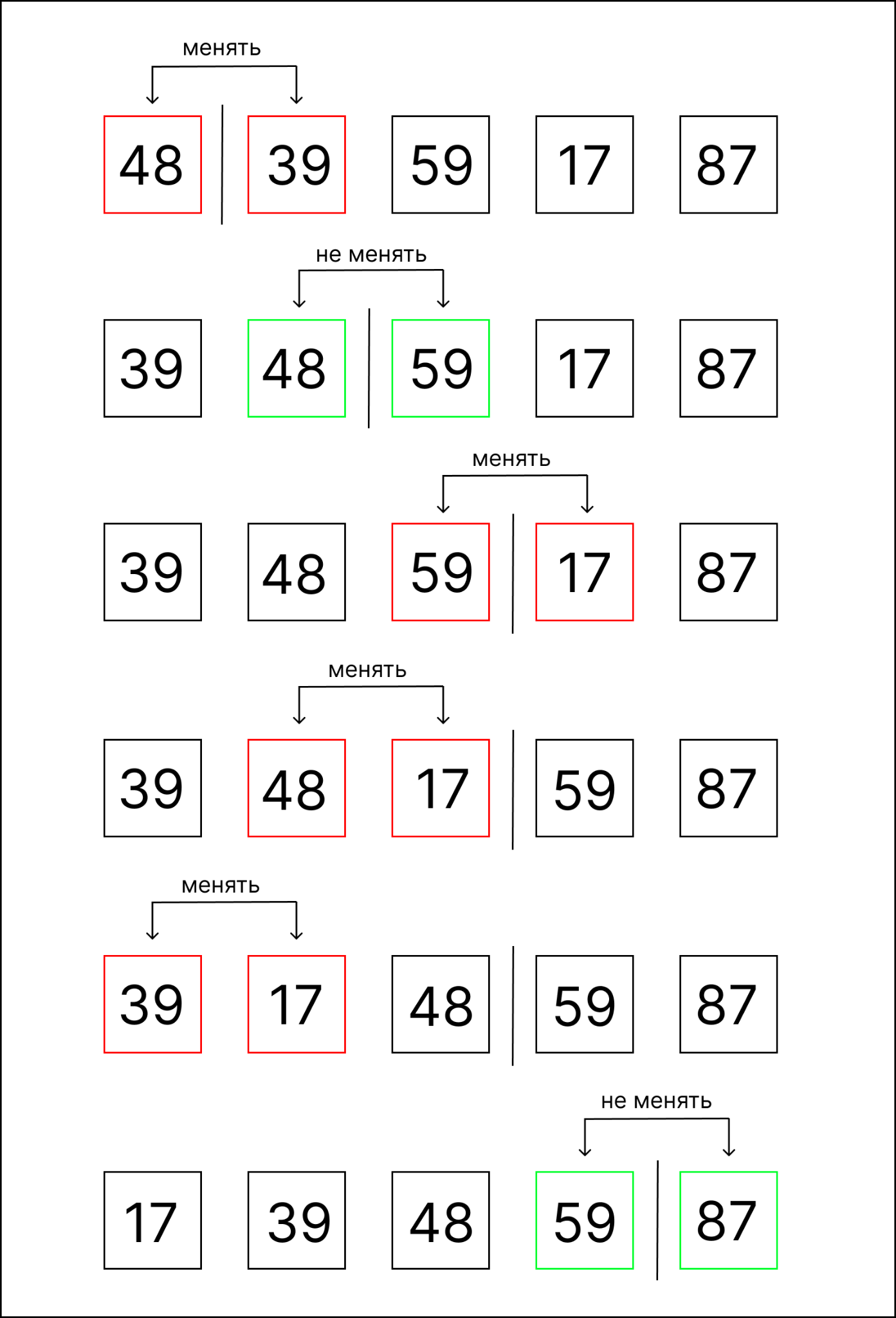


# Сортировка вставками

Алгоритм сортировки вставками (insertion sort) работает по следующему принципу: на каждом шаге мы берем очередной элемент массива и вставляем его в отсортированную часть массива так, чтобы она оставалась отсортированной. Таким образом, мы постепенно увеличиваем отсортированную часть массива до тех пор, пока не отсортируем весь массив.

|  |
| --- |
| void insertionSort(int arr[], int n) {  for (int i = 1; i < n; i++) {  int key = arr[i];  int j = i - 1;  while (j >= 0 && arr[j] > key) {  arr[j + 1] = arr[j];  j--;  }  arr[j + 1] = key;  } } |

В этом коде мы проходим по массиву начиная с первого элемента i = 1. На каждом шаге мы берем текущий элемент key, и сравниваем его со всеми элементами, которые идут перед ним, начиная с элемента j = i - 1. Если мы находим элемент, который больше key, то мы сдвигаем его на одну позицию вправо, чтобы освободить место для key. Мы продолжаем сдвигать элементы до тех пор, пока не найдем место, куда можно вставить key. Затем мы вставляем key в эту позицию.  
 Сортировка вставками имеет сложность O(n^2), что делает ее не очень эффективной для больших массивов. Однако, она имеет некоторые преимущества перед другими алгоритмами сортировки, например, она быстрее работает на уже почти отсортированных массивах.

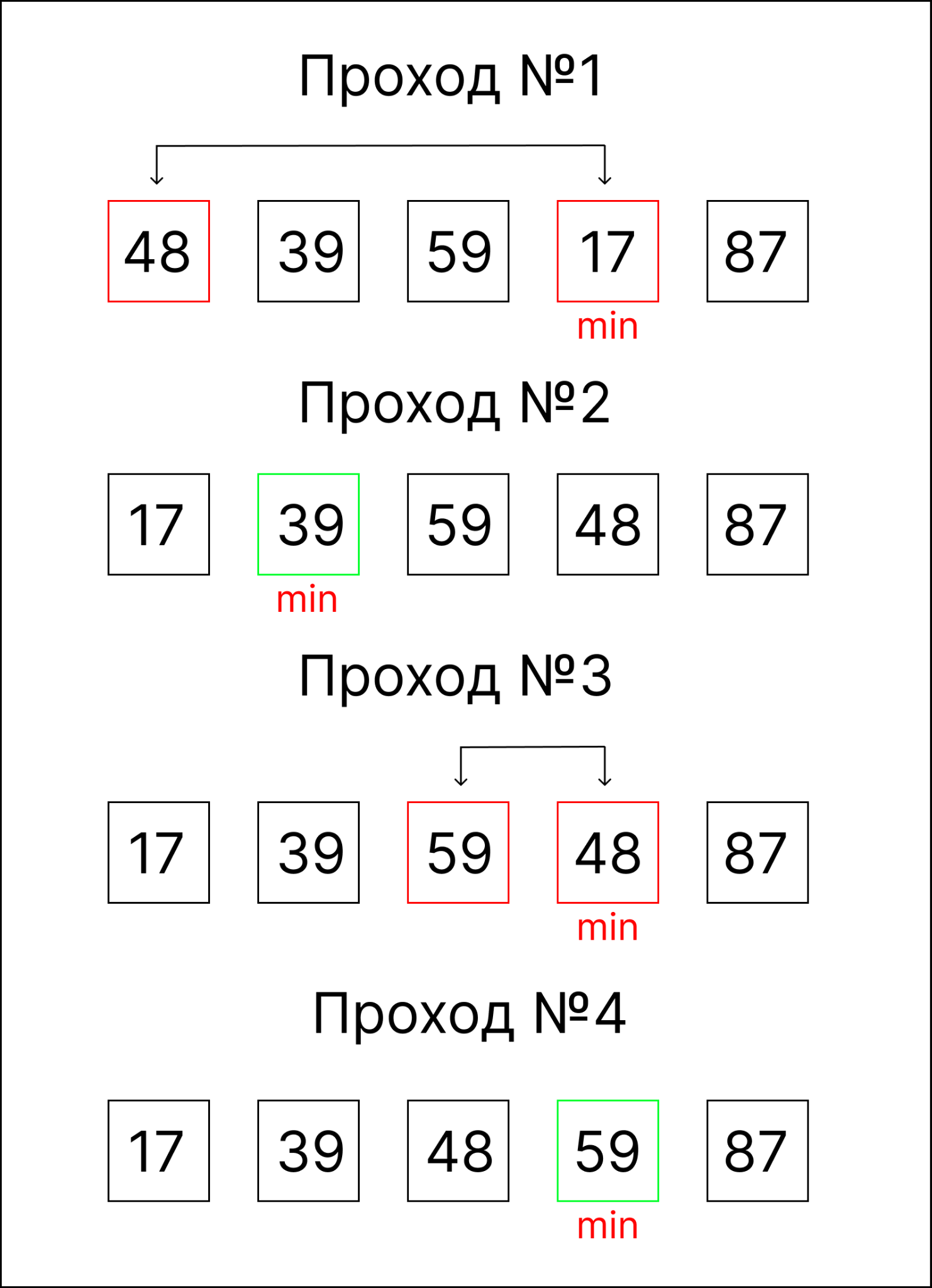


# Сортировка выбором

Алгоритм сортировки выбором (selection sort) работает по следующему принципу: на каждом шаге мы ищем минимальный элемент в неотсортированной части массива и меняем его местами с первым элементом в неотсортированной части. Таким образом, мы постепенно увеличиваем отсортированную часть массива до тех пор, пока не отсортируем весь массив.

|  |
| --- |
| void selectionSort(int arr[], int n) {  for (int i = 0; i < n - 1; i++) {  int minIndex = i;  for (int j = i + 1; j < n; j++) {  if (arr[j] < arr[minIndex]) {  minIndex = j;  }  }  swap(arr[i], arr[minIndex]);  } } |

В этом коде мы проходим по массиву начиная с первого элемента i = 0. На каждом шаге мы ищем минимальный элемент в неотсортированной части массива, начиная со следующего элемента после i, и меняем его местами с первым элементом в неотсортированной части, который имеет индекс i. Мы продолжаем так делать до тех пор, пока не отсортируем весь массив.  
 Сортировка выбором также имеет сложность O(n^2), что делает ее не очень эффективной для больших массивов. Однако, она также имеет некоторые преимущества перед другими алгоритмами сортировки, например, она менее затратна по памяти, так как не использует дополнительную память для сортировки.

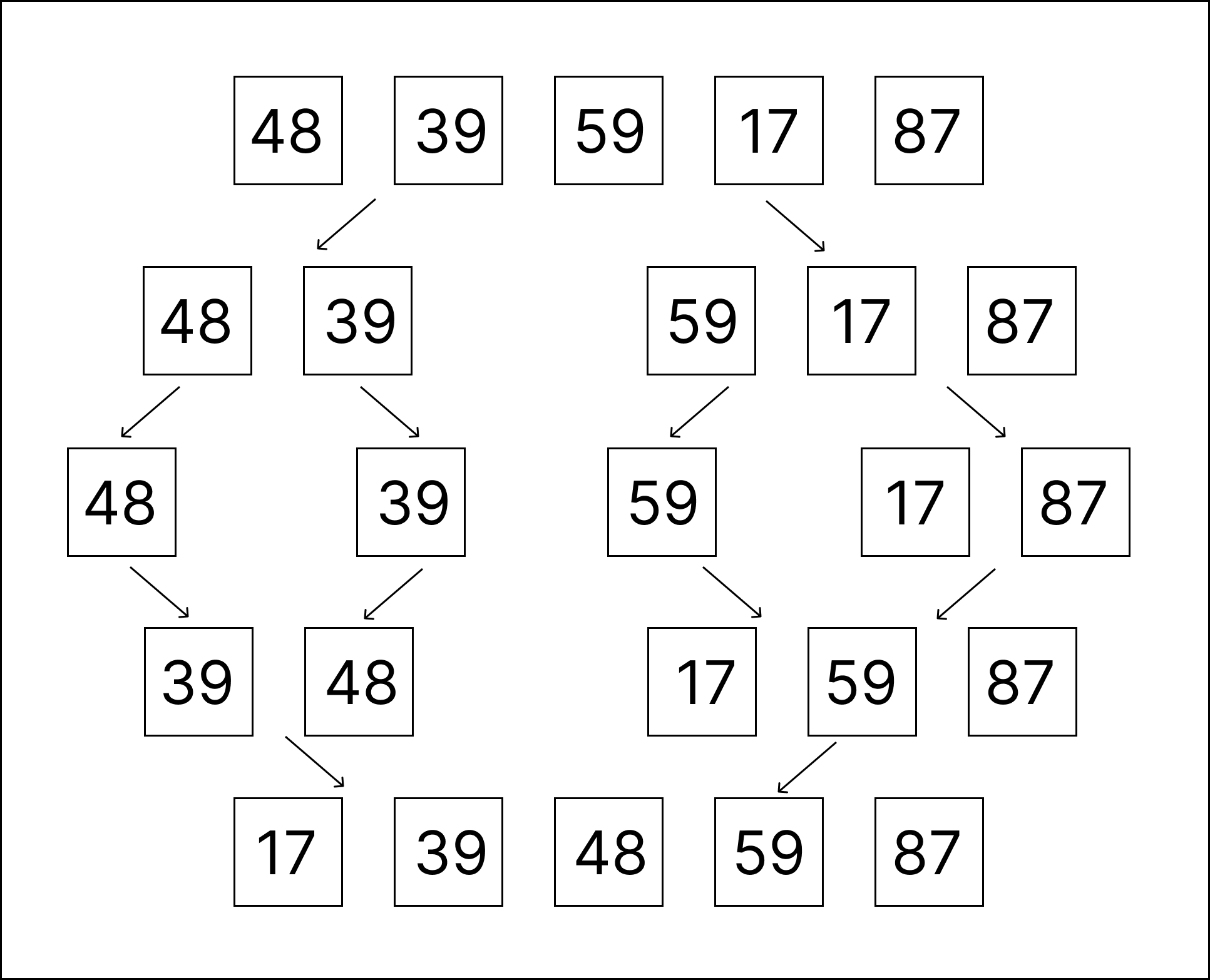


# Сортировка слиянием

Алгоритм сортировки слиянием (merge sort) работает по следующему принципу: мы делим исходный массив на две части, рекурсивно сортируем каждую из них и затем сливаем две отсортированные части в один отсортированный массив. Таким образом, мы постепенно уменьшаем размеры частей до тех пор, пока не получим отсортированные отдельные элементы, которые затем сливаем в один отсортированный массив.

|  |
| --- |
| void merge(int arr[], int left, int middle, int right) {  int size\_left = middle - left + 1;  int size\_right = right - middle;  int\* left\_arr = new int[size\_left];  int\* right\_arr = new int[size\_right];    for (int i = 0; i < size\_left; i++) {  left\_arr[i] = arr[left + i];  }  for (int j = 0; j < size\_right; j++) {  right\_arr[j] = arr[middle + 1 + j]; }   int i = 0, j = 0, k = left;  while (i < size\_left && j < size\_right) {  if (left\_arr[i] <= right\_arr[j]) {  arr[k] = left\_arr[i];  i++;  }  else {  arr[k] = right\_arr[j];  j++;  }  k++;  }    while (i < size\_left) {  arr[k] = left\_arr[i];  i++;  k++;  }    while (j < size\_right) {  arr[k] = right\_arr[j];  j++;  k++;  }  delete[] left\_arr;  delete[] right\_arr; }  void mergeSort(int arr[], int left, int right) {  if (left < right) {  int middle = (left + right) / 2;  mergeSort(arr, left, middle);  mergeSort(arr, middle + 1, right);  merge(arr, left, middle, right);  } } |

В этом коде мы используем функцию merge(), которая сливает две отсортированные части массива (от l до m и от m+1 до r) в один отсортированный массив. Затем мы используем функцию mergeSort(), которая рекурсивно разделяет исходный массив на две части и сортирует каждую из них с помощью merge(). Мы продолжаем делать это до тех пор, пока не получим отдельные отсортированные элементы, которые затем сливаем в один отсортированный массив.  
 Сортировка слиянием имеет сложность O(n log n), что делает ее более эффективной для больших массивов, чем сортировки со сложностью O(n^2). Однако, она требует дополнительной памяти для слияния частей массива, что может быть проблемой для очень больших массивов.

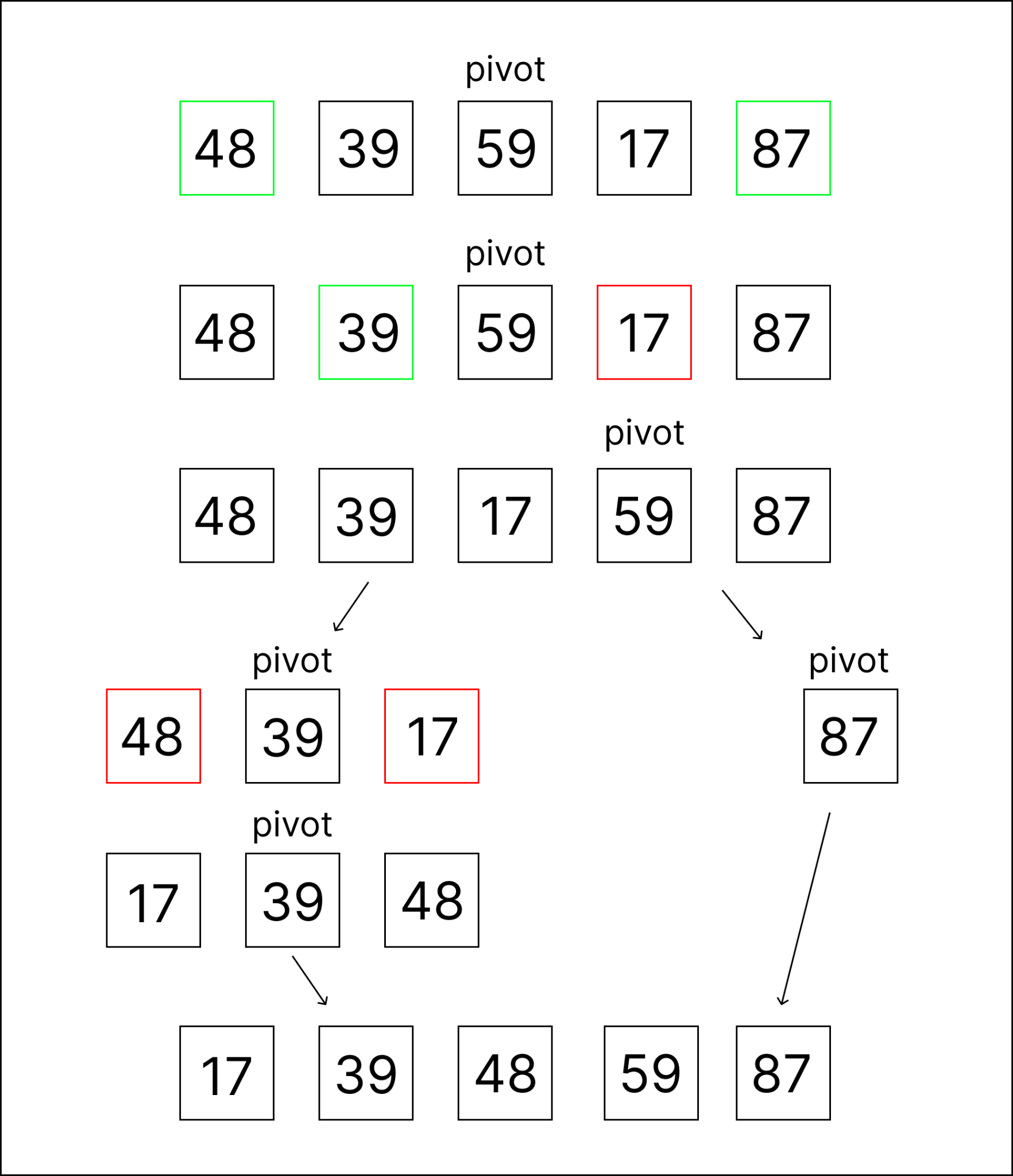


# Быстрая сортировка

Алгоритм быстрой сортировки (quick sort) основан на методе "разделяй и властвуй". Он работает по следующему принципу: мы выбираем опорный элемент из массива, затем перемещаем все элементы, меньшие опорного, на его левую сторону, а все элементы, большие опорного, на правую сторону. Таким образом, мы разбиваем массив на две части и рекурсивно повторяем этот процесс для каждой из них. На каждом шаге выбирается новый опорный элемент, который помогает дальше разбивать массив на две части. Когда подмассивы достигают размера 1 или 0, сортировка завершается.

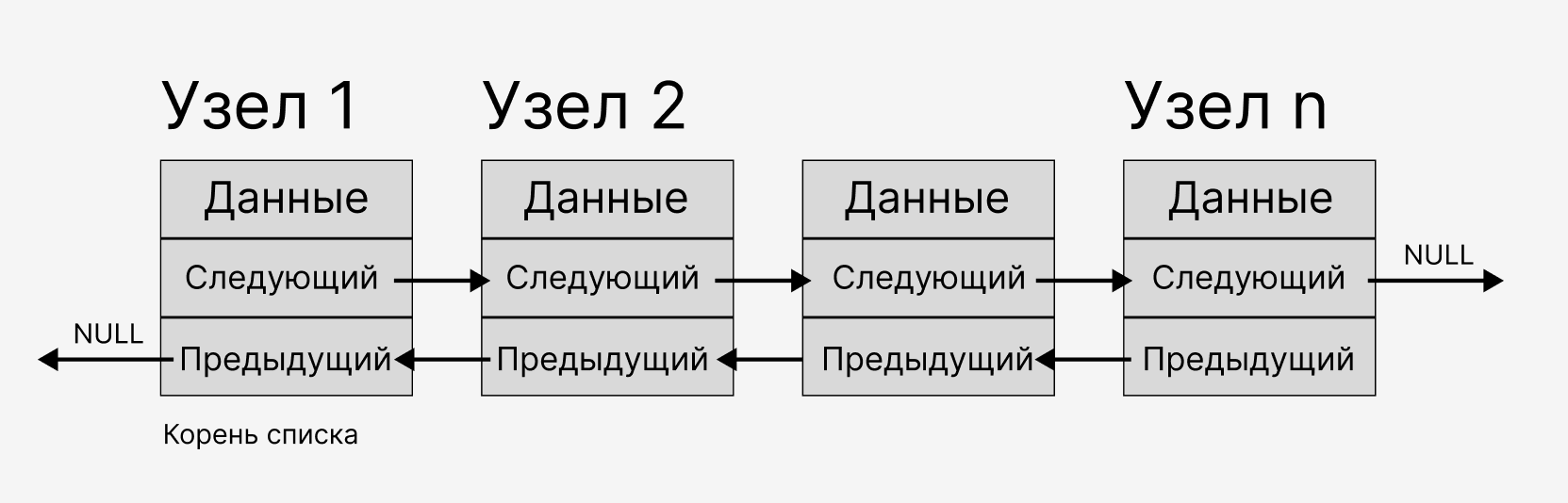
|  |
| --- |
| int partOfQuickSort(int arr[], int left, int right) {  int opora = arr[(left + right) / 2];  while (left <= right) {  while (arr[left] < opora) left++;  while (arr[right] > opora) right--;  if (left <= right) {  int temp = arr[left];  arr[left] = arr[right];  arr[right] = temp;  left++;  right--;  }  }  return left;  }  void quickSort(int arr[], int start, int end) {  if (start >= end) return;  int rightStart = partOfQuickSort(arr, start, end);  quickSort(arr, start, rightStart - 1); quickSort(arr, rightStart, end); } |

Эта реализация сортирует массив arr в диапазоне от left до right. Опорный элемент выбирается как средний элемент из диапазона.  
 Сложность быстрой сортировки в среднем случае O(n log n), в худшем случае O(n^2). Однако, благодаря эффективной реализации и небольшому количеству дополнительной памяти, быстрая сортировка является одним из наиболее широко используемых алгоритмов сортировки в программировании.



**Теоретические сведения № 2**

List — это двусвязный список, каждый элемент которого содержит 2 указателя: один указывает на следующий элемент списка, а другой — на предыдущий элемент списка.   
 List предоставляет доступ только к началу и к концу списка — произвольный доступ запрещен. Если вы хотите найти значение где-то в середине, то вы должны начать с одного конца и перебирать каждый элемент списка до тех пор, пока не найдете то, что ищете.  
 Преимуществом двусвязного списка является то, что добавление элементов происходит очень быстро, если вы, конечно, знаете, куда хотите добавлять. Обычно для перебора элементов двусвязного списка используются итераторы.



Для использования данного типа списка необходимо подключить заголовочный файл list.

|  |
| --- |
| #include <list> |

Чтобы объявить двусвязный список, нужно пользоваться конструкцией ниже:

|  |
| --- |
| list < тип данных > <имя контейнера>; |

Способы создания двусвязного списка:

|  |
| --- |
| std::list<int> list1; // пустой список std::list<int> list2(6); // список list2 состоит из 6 чисел, каждый элемент имеет значение по умолчанию std::list<int> list3(4, 3); // список list3 состоит из 4 чисел, каждое число равно 3 std::list<int> list4{ 1, 2, 4, 5 }; // список list4 состоит из чисел 1, 2, 4, 5 std::list<int> list5 = { 1, 2, 3, 5 }; // список list5 состоит из чисел 1, 2, 4, 5 std::list<int> list6(list4); // список list6 - копия списка list4 std::list<int> list7 = list4; // список list7 - копия списка list4 |

В отличие от других контейнеров для типа list не определена операция обращения по индексу или функция **at()**, которая выполняет похожую задачу.

Тем не менее для контейнера list можно использовать функции **front()** и **back()**, которые возвращают соответственно первый и последний элементы.

|  |
| --- |
| #include <iostream> #include <list>  using namespace std;  int main() {  list<int> numbers{ 1, 2, 3, 4, 5 };  int first{ numbers.front() };  int last{ numbers.back() };   cout << "First: " << first << endl;// 1  cout << "Last: " << last << endl;//5 } |

Для получения размера списка можно использовать функцию **size()**:

|  |
| --- |
| #include <iostream> #include <list>  using namespace std;  int main() {  list<int> numbers{ 1, 2, 3, 4 };  cout << "Size is: " << numbers.size() << endl;// 4 } |

С помощью функции **resize()** можно изменить размер списка. Эта функция имеет две формы:

* **resize(n)**: оставляет в списке n первых элементов. Если список содержит больше элементов, то он усекается до первых n элементов. Если размер списка меньше n, то добавляются недостающие элементы и инициализируются значением по умолчанию

|  |
| --- |
| #include <iostream> #include <list>  using namespace std;  int main() {  list<int> numbers{ 1, 2, 3, 4 };  for (int n : numbers)//1 2 3 4  cout << n << "\t";  cout << endl;  numbers.resize(2);  for (int n : numbers)// 1 2   cout << n << "\t";  cout << endl;  } |

* **resize(n, value)**: также оставляет в списке n первых элементов. Если размер списка меньше n, то добавляются недостающие элементы со значением value

|  |
| --- |
| #include <iostream> #include <list>  using namespace std;  int main() { list<int> numbers{ 1, 2, 3, 4 };  for (int n : numbers)// 1 2 3 4  cout << n << "\t";  cout << endl;  numbers.resize(6, 8);  for (int n : numbers)// 1 2 3 4 8 8  cout << n << "\t";  cout << endl; } |

Функция **assign()** позволяет заменить все элементы списка определенным набором. Она имеет следующие формы:

* **assign(il):** заменяет содержимое контейнера элементами из списка инициализации il

|  |
| --- |
| #include <iostream> #include <list>  using namespace std;  int main() {  list<int> numbers{ 1, 2, 3, 4 };  numbers.assign({ 5, 6, 7, 8, 9 });  for (int n : numbers)// 5 6 7 8 9  cout << n << "\t";  cout << endl; } |

* **assign(n, value)**: заменяет содержимое контейнера n элементами, которые имеют значение value

|  |
| --- |
| #include <iostream> #include <list>  using namespace std;  int main() {  list<int> numbers{ 1, 2, 3, 4 };  numbers.assign(4, 9);  for (int n : numbers)// 9 9 9 9  cout << n << "\t";  cout << endl; } |

* **assign(begin, end)**: заменяет содержимое контейнера элементами из диапазона, на начало и конец которого указывают итераторы begin и end

|  |
| --- |
| #include <iostream> #include <list>  using namespace std;  int main() {  list<int> numbers{ 1, 2, 3, 4 };  list<int> values{ 5, 6, 7, 8, 9, 10 };  auto start = ++values.begin();   auto end = values.end();  numbers.assign(start, end);  cout <<"List of numbers:" << endl;  for (int n : numbers)// 6 7 8 9 10  cout << n << "\t";  cout << endl; } |

Функция **swap()** обменивает значениями два списка:

|  |
| --- |
| #include <iostream> #include <list>  using namespace std;  int main() {  list<int> numbers{ 1, 2, 3, 4 };  list<int> values{ 5, 6, 7, 8, 9, 10 };  numbers.swap(values);  cout <<"List of numbers:" << endl;  for (int n : numbers)// 5 6 7 8 9 10  cout << n << "\t";  cout << endl;  cout << "List of values:" << endl;  for (int v : values)//1 2 3 4  cout << v << "\t";  cout << endl; } |

Для добавления элементов в контейнер list применяется ряд функций.

* **push\_back(val)**: добавляет значение val в конец списка
* **push\_front(val):** добавляет значение val в начало списка

|  |
| --- |
| #include <iostream> #include <list>  using namespace std;  int main() {  list<int> numbers{ 1, 2, 3, 4 };  numbers.push\_back(6);  numbers.push\_front(0);  cout <<"List of numbers:" << endl;  for (int n : numbers)// 0 1 2 3 4 6  cout << n << "\t";  cout << endl; } |

* **emplace\_back(val)**: добавляет значение val в конец списка
* **emplace\_front(val)**: добавляет значение val в начало списка
* **emplace(pos, val)**: вставляет элемент val на позицию, на которую указывает итератор pos. Возвращает итератор на добавленный элемент

|  |
| --- |
| #include <iostream> #include <list>  using namespace std;  int main()  {  list<int> numbers{ 1, 2, 3, 4 };  numbers.emplace\_back(9);  numbers.emplace\_front(10);   auto iter = ++numbers.cbegin();  numbers.emplace(iter, 13);  cout <<"List of numbers:" << endl;  for (int n : numbers)// 10 13 1 2 3 4 9  cout << n << "\t";  cout << endl; } |

* **insert(pos, val)**: вставляет элемент val на позицию, на которую указывает итератор pos, аналогично функции emplace. Возвращает итератор на добавленный элемент
* **insert(pos, n, val)**: вставляет n элементов val начиная с позиции, на которую указывает итератор pos. Возвращает итератор на первый добавленный элемент. Если n = 0, то возвращается итератор pos.
* i**nsert(pos, begin, end)**: вставляет начиная с позиции, на которую указывает итератор pos, элементы из другого контейнера из диапазона между итераторами begin и end. Возвращает итератор на первый добавленный элемент. Если между итераторами begin и end нет элементов, то возвращается итератор pos.
* **insert(pos, values)**: вставляет список значений values начиная с позиции, на которую указывает итератор pos. Возвращает итератор на первый добавленный элемент. Если values не содержит элементов, то возвращается итератор pos.

|  |
| --- |
| #include <iostream> #include <list>  using namespace std;  int main() {  std::list<int> numbers{ 1, 2, 3, 4, 5 };  std::list<int> values{ 10, 20, 30, 40, 50 };  auto iter1 = numbers.cbegin(); // итератор указывает на первый элемент  numbers.insert(iter1, 0); // добавляем начало списка   auto iter2 = numbers.cbegin(); // итератор указывает на первый элемент  numbers.insert(++iter2, 3, 4); // добавляем после первого элемента три четверки   auto iter3 = numbers.cbegin(); // итератор указывает на первый элемент  numbers.insert(iter3, values.begin(), values.end()); // добавляем в начало все элементы из values  auto iter4 = numbers.cend(); // итератор указывает на позицию за последним элементом  numbers.insert(iter4, { 21, 22, 23 });// добавляем в конец список из трех элементов  cout <<"List of numbers:" << endl;  for (int n : numbers)//10 20 30 40 50 0 4 4 4 1 2 3 4 5 21 22 23  cout << n << " ";  cout << endl; } |

Для удаления элементов из контейнера list могут применяться следующие функции:

* **clear(p)**: удаляет все элементы
* **pop\_back()**: удаляет последний элемент
* **pop\_front()**: удаляет первый элемент

|  |
| --- |
| #include <iostream> #include <list>  using namespace std;  int main() {  list<int> numbers{ 1, 2, 3, 4, 5 };  numbers.pop\_front(); // numbers = { 2, 3, 4, 5 }  numbers.pop\_back(); // numbers = { 2, 3, 4 }  cout <<"List of numbers:" << endl;  for (int n : numbers)// 2, 3, 4   cout << n << " ";  cout << endl;  } |

* **erase(p):** удаляет элемент, на который указывает итератор p. Возвращает итератор на элемент, следующий после удаленного, или на конец контейнера, если удален последний элемент
* **erase(begin, end)**: удаляет элементы из диапазона, на начало и конец которого указывают итераторы begin и end. Возвращает итератор на элемент, следующий после последнего удаленного, или на конец контейнера, если удален последний элемент

|  |
| --- |
| #include <iostream> #include <list>  using namespace std;  int main() {  list<int> numbers{ 1, 2, 3, 4, 5 };  auto iter = numbers.cbegin(); // указатель на первый элемент  numbers.erase(iter); // удаляем первый элемент  auto begin = numbers.begin(); // указатель на первый элемент  auto end = numbers.end(); // указатель на последний элемент numbers.erase(++begin, --end); // удаляем со второго элемента до последнего  cout <<"List of numbers:" << endl;  for (int n : numbers) // 2 5  cout << n << " ";  cout << endl; } |

## Задачи

**Задание 1.**

Реализовать возможность просмотр отсортированной структуры (использовать все рассмотренные выше сортировки). Реализовать сортировки как по убыванию, так и по возрастанию

Пример для 1 вариант. Клиент в меню помимо обычного просмотра списка номеров, может просмотреть отсортированный список номеров по их стоимости.

Код:

#include <iostream>

#include <string>

#include <fstream>

#include <windows.h>

using namespace std;

struct client {

string surname;

string pasd;

int date\_join;

string date\_out;

string type\_off\_lat;

};

struct client List\_of\_hotel\_clients[100];

void insertionSortAscending(client arr[], int n) {

for (int i = 1; i < n; i++) {

int j = i;

while (j > 0 && arr[j - 1].date\_join > arr[j].date\_join) {

swap(arr[j], arr[j - 1]);

j--;

}

}

}

void insertionSortDescending(client arr[], int n) {

for (int i = 1; i < n; i++) {

int j = i;

while (j > 0 && arr[j - 1].date\_join < arr[j].date\_join) {

swap(arr[j], arr[j - 1]);

j--;

}

}

}

int main() {

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

setlocale(LC\_ALL, "Russian");

int n = 0;

int choice = 1;

while (choice != 0) {

cout << "1 - ввод элементов структуры с клавиатуры\n";

cout << "2 - вывод элементов структуры в консольное окно\n";

cout << "3 - удаление заданной структурированной переменной\n";

cout << "4 - отсортировать по дате прибытия (возрастание)\n";

cout << "5 - запись информации в файл\n";

cout << "6 - чтение данных из файла\n";

cout << "7 - отсортировать по дате прибытия (убывание)\n";

cout << "0 - Выход из программы\n\n";

cout << "Введите номер операции: ";

cin >> choice;

if (choice == 1) {

cout << "Введите данные: Фамилию, Паспортные данные, дату приезда, дату отъезда, тип размещения: ";

cin >> List\_of\_hotel\_clients[n].surname;

cin >> List\_of\_hotel\_clients[n].pasd;

cin >> List\_of\_hotel\_clients[n].date\_join;

cin >> List\_of\_hotel\_clients[n].date\_out;

cin >> List\_of\_hotel\_clients[n].type\_off\_lat;

n++;

}

if (choice == 2) {

cout << "Список проживающих: " << endl;

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

if (List\_of\_hotel\_clients[i].surname.empty()) continue;

cout << List\_of\_hotel\_clients[i].surname << " ";

cout << List\_of\_hotel\_clients[i].pasd << " ";

cout << List\_of\_hotel\_clients[i].date\_join << " ";

cout << List\_of\_hotel\_clients[i].date\_out << " ";

cout << List\_of\_hotel\_clients[i].type\_off\_lat << " " << endl;

}

}

if (choice == 3) {

cout << "Введите порядковый номер клиента, которого хотите удалить: ";

int nn;

cin >> nn;

if (nn >= 0 && nn < n) {

List\_of\_hotel\_clients[nn].surname = "";

List\_of\_hotel\_clients[nn].pasd = "";

List\_of\_hotel\_clients[nn].date\_join = 0;

List\_of\_hotel\_clients[nn].date\_out = "";

List\_of\_hotel\_clients[nn].type\_off\_lat = "";

}

else {

cout << "Некорректный номер клиента!" << endl;

}

}

if (choice == 4) {

insertionSortAscending(List\_of\_hotel\_clients, n);

cout << "Сортировка по возрастанию выполнена." << endl;

}

if (choice == 5) {

fstream one("clients.txt", ios::out);

if (!one.is\_open()) {

cout << "Ошибка при открытии файла!" << endl;

return 1;

}

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

if (List\_of\_hotel\_clients[i].surname.empty()) continue;

one << List\_of\_hotel\_clients[i].surname << " "

<< List\_of\_hotel\_clients[i].pasd << " "

<< List\_of\_hotel\_clients[i].date\_join << " "

<< List\_of\_hotel\_clients[i].date\_out << " "

<< List\_of\_hotel\_clients[i].type\_off\_lat << endl;

}

one.close();

cout << "Информация записана в файл." << endl;

}

if (choice == 6) {

fstream one("clients.txt", ios::in);

if (!one.is\_open()) {

cout << "Ошибка при открытии файла!" << endl;

return 1;

}

n = 0;

while (one >> List\_of\_hotel\_clients[n].surname) {

one >> List\_of\_hotel\_clients[n].pasd;

one >> List\_of\_hotel\_clients[n].date\_join;

one >> List\_of\_hotel\_clients[n].date\_out;

one >> List\_of\_hotel\_clients[n].type\_off\_lat;

n++;

}

one.close();

cout << "Данные прочитаны из файла." << endl;

}

if (choice == 7) {

insertionSortDescending(List\_of\_hotel\_clients, n);

cout << "Сортировка по убыванию выполнена." << endl;

}

}

}

**Задание 2.**

Написать программу, в которой в начало двусвязного списка добавляется n элементов и вывести на экран.

Код:

//Написать программу, в которой в начало

// двусвязного списка добавляется n элементов и вывести на экран.

#include <iostream>

#include <list>

using namespace std;

int main()

{

setlocale(LC\_ALL, "ru");

list <int> arr = { 1,2,3,4,5};

cout << "Введите число n";

int n;

cin >> n;

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

arr.push\_front(rand() % 100);

}

cout << "Список: ";

for (auto it = arr.begin(); it != arr.end(); ++it) {

cout << \*it << " ";

}

return 0;

}

**Задание 3.**

Написать программу, в которой вывести содержимое двусвязного списка в обратном порядке.

#include <iostream>

#include <list>

using namespace std;

int main() {

setlocale(LC\_ALL, "ru");

list<int> arr = { 1, 2, 3, 4, 5 };

cout << "Список в обратном порядке: ";

for (auto it = arr.rbegin(); it != arr.rend(); ++it) {

cout << \*it << " ";

}

return 0;

}

**Задание 4.**

Написать программу, в которой удалить все четные элементы двусвязного списка.

#include <iostream>

#include <list>

using namespace std;

int main() {

list<int> arr = { 1, 2, 3, 4, 5 };

for (auto it = arr.begin(); it != arr.end(); ) {

if (\*it % 2 == 0) {

it = arr.erase(it);

}

else {

++it;

}

}

for (int x : arr) {

cout << x << " ";

}

return 0;

}