Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Факультет безопасности информационных технологий

Дисциплина:
«Алгоритмы и структуры данных»

Лабораторная работа №3

Выполнил:

Беляков Никита Андреевич N3245

(подпись)

Проверил:

Еврофеев С.А.

(отметка о выполнении)

(подпись)

Санкт-Петербург
2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 3

ХОД РАБОТЫ 4
1.1 Блок схема 4
1.2 Псевдокод 7
1.3 Маршрут 8
1.3.1 Общий Поток Программы 9
1.3.2 Маршруты Команд 9
1.3.3 Очистка Консоли 10
1.3.4 Обработка Неизвестной Команды 10
1.4 Спецификация переменных 10
1.5 Листинг программы 10
1.6 Разбор цикла по шагам 15
1.7 Расчет сложности алгоритма 15
1.7.1 Метод Enqueue 16
1.7.2 Метод Dequeue 16
1.7.3 Метод Display 17
1.7.4 Очистка консоли 17
1.7.5 Сортировка 18
1.7.6 Точка входа программы 19
1.7.7 Финальный вывод 20
1.8 Тесты 20
1.8.1 CircularQueueBetonicTest.go 20
1.8.2 Вывод результатов 22
1.8.3 Ручное тестирование 22

ЗАКЛЮЧЕНИЕ 24

ВВЕДЕНИЕ

Цель данной лабораторной работы состоит в разработке программы для битонной сортировки с использованием кольцевой очереди, основанной на связном списке. Битонная сортировка - это эффективный алгоритм сортировки, который может быть применен к различным типам данных, включая числа и строки. Использование кольцевой очереди для хранения данных позволяет эффективно реализовать операции добавления, удаления и обработки элементов в программе. Разработка такой программы позволит нам лучше понять принципы работы битонной сортировки и применить их на практике.

1 ХОД РАБОТЫ

1.1 Блок схема

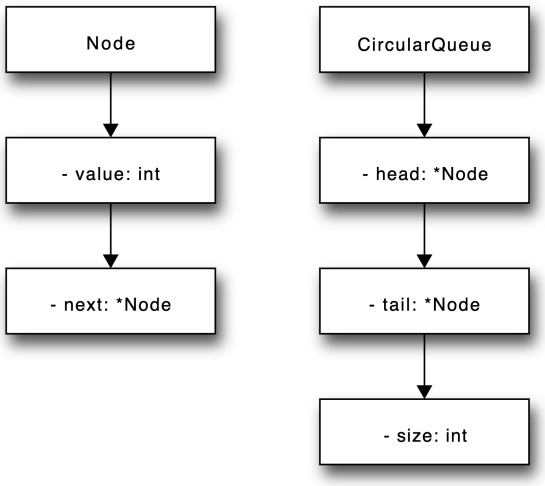


Рисунок 1.1 — Структуры.

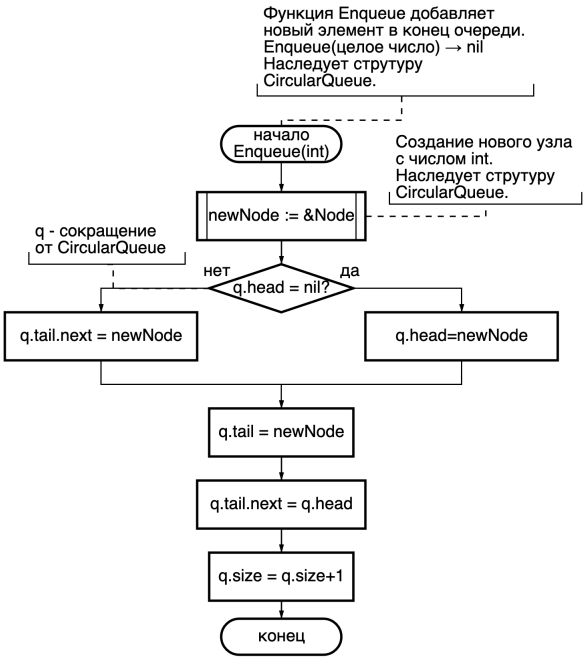


Рисунок 1.2 — Функция Enqueue().

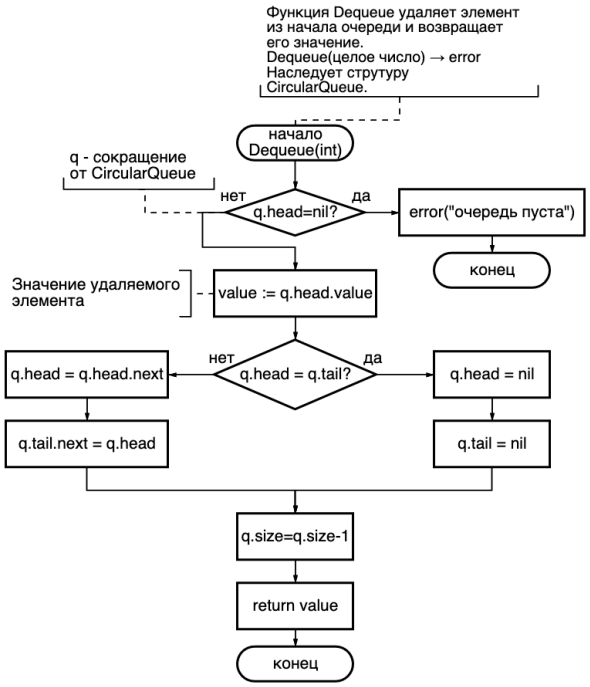


Рисунок 1.3 — Функция Dequeue().

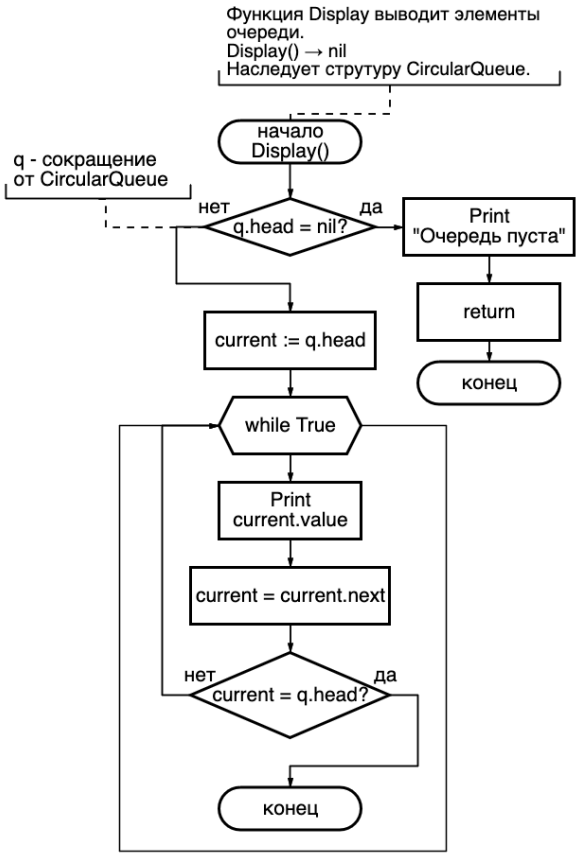


Рисунок 1.4 — Функция Display().

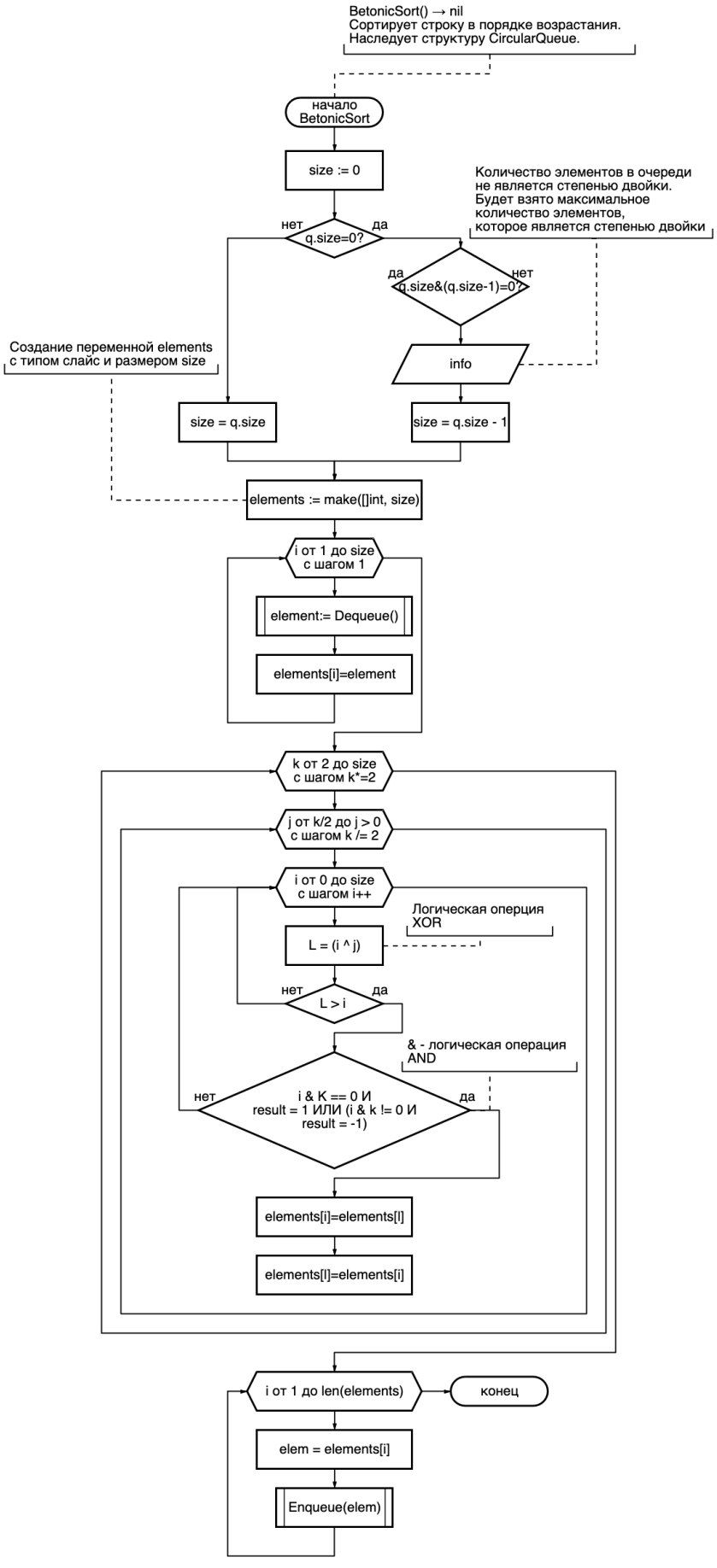


Рисунок 1.5 — Функция BetonicSort()..

1.2 Псевдокод

```
АЛГОРИТМ КольцеваяОчередь
НАЧАЛО
    ОПРЕДЕЛИТЬ структуру Узел
        ЦЕЛОЕ значение
        УКАЗАТЕЛЬ на Узел следующий

    ОПРЕДЕЛИТЬ структуру КольцеваяОчередь
        УКАЗАТЕЛЬ на Узел голова
        УКАЗАТЕЛЬ на Узел хвост
        ЦЕЛОЕ размер

    ФУНКЦИЯ Добавить(значение ЦЕЛОЕ)
        СОЗДАТЬ новыйУзел = НОВЫЙ Узел(значение)
        ЕСЛИ голова = НУЛЬ ТО
            голова = новыйУзел
        ИНАЧЕ
            хвост.следующий = новыйУзел
        КОНЕЦ ЕСЛИ
        хвост = новыйУзел
        хвост.следующий = голова
        увеличить размер на 1
    КОНЕЦ ФУНКЦИИ

    ФУНКЦИЯ Удалить()
        ЕСЛИ голова = НУЛЬ ТО
            ВОЗВРАТИТЬ ошибка "Очередь пуста"
        значение = голова.значение
        ЕСЛИ голова = хвост ТО
            голова = НУЛЬ
            хвост = НУЛЬ
        ИНАЧЕ
            голова = голова.следующий
            хвост.следующий = голова
        КОНЕЦ ЕСЛИ
        уменьшить размер на 1
        ВОЗВРАТИТЬ значение
    КОНЕЦ ФУНКЦИИ

    ФУНКЦИЯ Показать()
        ЕСЛИ голова = НУЛЬ ТО
            ВЫВОД "Очередь пуста"
            ВОЗВРАТ
        ТЕКУЩИЙ = голова
        ПОКА ТЕКУЩИЙ != голова ИЛИ ПЕРВЫЙ ПРОХОД
            ВЫВОД ТЕКУЩИЙ.значение
            ТЕКУЩИЙ = ТЕКУЩИЙ.следующий
            ПЕРВЫЙ ПРОХОД = ЛОЖЬ
        КОНЕЦ ПОКА
```

```
49          ВЫВОД новая строка
50      КОНЕЦ ФУНКЦИИ
51
52      ФУНКЦИЯ ОчиститьКонсоль()
53          ВЫБОР ОС
54              СЛУЧАЙ "linux", "darwin":
55                  выполнить команду "clear"
56              СЛУЧАЙ "windows":
57                  выполнить команду "cmd /c cls"
58              ИНАЧЕ
59                  ВОЗВРАТИТЬ ошибка "ОС не поддерживается"
60          КОНЕЦ ВЫБОРА
61      КОНЕЦ ФУНКЦИИ
62
63      ФУНКЦИЯ БитоннаяСортировка()
64          определить максимальный размер, являющийся степенью двойки
65          извлечь элементы в массив
66          выполнить битонную сортировку
67          вернуть элементы в очередь
68      КОНЕЦ ФУНКЦИИ
69
70      ГЛАВНАЯ ПРОГРАММА
71          СОЗДАТЬ очередь
72          В ЦИКЛЕ
73              ОчиститьКонсоль()
74              ВВОД команды
75              ВЫБОР команды
76                  СЛУЧАЙ "enqueue":
77                      Добавить(значение)
78                  СЛУЧАЙ "dequeue":
79                      Удалить()
80                  СЛУЧАЙ "sort":
81                      БитоннаяСортировка()
82                  СЛУЧАЙ "display":
83                      Показать()
84                  СЛУЧАЙ "exit":
85                      ВЫХОД ИЗ ПРОГРАММЫ
86                  ИНАЧЕ
87                      ВЫВОД "Неизвестная команда"
88              КОНЕЦ ВЫБОРА
89          КОНЕЦ ЦИКЛА
90      КОНЕЦ ГЛАВНОЙ ПРОГРАММЫ
91  КОНЕЦ
92
```

1.3 Маршрут

Маршрут выполнения программы.

1.3.1 Общий Поток Программы

1. **Инициализация**: Создается экземпляр кольцевой очереди.
2. **Ожидание Команды**: Программа выводит доступные команды и ожидает ввода пользователя.
3. **Обработка Команды**: В зависимости от команды выполняется соответствующий блок действий.
4. **Повторение или Завершение**: Если команда не является `exit`, программа повторяет цикл ожидания и обработки команды. В противном случае программа завершает свою работу.

1.3.2 Маршруты Команд

Команда Enqueue

- **Ввод**: Пользователь вводит `enqueue [число]`.
- **Действие**: Добавляет число в конец кольцевой очереди.
- **Вывод**: Сообщение об успешном добавлении числа.

Команда Dequeue

- **Ввод**: Пользователь вводит `dequeue`.
- **Действие**: Удаляет число из начала кольцевой очереди.
- **Вывод**: Выводит значение удаленного числа или сообщение об ошибке, если очередь пуста.

Команда Sort

- **Ввод**: Пользователь вводит `sort`.
- **Действие**: Выполняет битонную сортировку элементов в очереди.
- **Вывод**: Сообщение об успешной сортировке очереди.

Команда Display

- **Ввод**: Пользователь вводит `display`.
- **Действие**: Отображает элементы очереди.
- **Вывод**: Выводит все элементы очереди или сообщает, что очередь пуста.

Команда Exit

- **Ввод**: Пользователь вводит `exit`.
- **Действие**: Завершает выполнение программы.
- **Вывод**: Сообщение о завершении работы программы.

1.3.3 Очистка Консоли

- **Действие**: Очищает текст консоли перед каждым новым циклом ожидания команды.
- **Особенность**: Выполняется автоматически перед выводом доступных команд.

1.3.4 Обработка Неизвестной Команды

- **Ввод**: Пользователь вводит нераспознанную команду.
- **Действие**: Игнорирует команду.
- **Вывод**: Сообщение о неизвестной команде и список доступных команд.

1.4 Спецификация переменных

Ниже представлена спецификация переменных.

Переменная	Тип	Минимум	Максимум	Значение
value	int	-2^{31}	$2^{31}-1$	Значение узла в очереди
head, tail	*Node	–	–	Указатели на начало и конец очереди
size	int	0	$2^{31}-1$	Размер кольцевой очереди
newNode	*Node	–	–	Новый узел для добавления в очередь
current	*Node	–	–	Текущий узел при обходе очереди
cmd	*exec.Cmd	–	–	Команда для очистки консоли
elements	[]int	–	–	Массив элементов для сортировки
i, j, k, l	int	-2^{31}	$2^{31}-1$	Индексы для итерации в алгоритме сортировки
size	int	0	$2^{31}-1$	Переменная для хранения размера очереди в сортировке
cmdString	string	0	2^{31}	Строка с введенной командой
cmdParts	[]string	–	–	Разделенная на части команда

1.5 Листинг программы

Ниже представлен полный листинг программы:

```
1  package main
2
3  import (
4          "bufio"
5          "errors"
6          "fmt"
7          "github.com/fatih/color" // Импорт пакета для работы с цветом вывода
8          "os"
9          "os/exec"
10         "runtime"
11         "strconv"
12         "strings"
13 )
14
15 // Node представляет собой узел в связном списке.
16 type Node struct {
17         value int   // Значение узла
```

```
18          next  *Node // Следующий узел в списке
19  }
20
21  // CircularQueue представляет собой структуру кольцевой очереди.
22  type CircularQueue struct {
23          head *Node // Начало очереди
24          tail *Node // Конец очереди
25          size int   // Размер очереди
26  }
27
28  // Enqueue добавляет новый элемент в конец очереди.
29  func (q *CircularQueue) Enqueue(value int) {
30          newNode := &Node{value: value} // Создание нового узла
31          if q.head == nil {             // Если очередь пуста
32                  q.head = newNode
33          } else { // Если в очереди уже есть элементы
34                  q.tail.next = newNode
35          }
36          q.tail = newNode // Обновление указателя на конец очереди
37          // Связываем последний и первый элементы для обеспечения кольцевой структуры
38          q.tail.next = q.head
39          q.size++ // Увеличиваем размер очереди
40  }
41
42  // Dequeue удаляет элемент из начала очереди и возвращает его значение.
43  func (q *CircularQueue) Dequeue() (int, error) {
44          if q.head == nil { // Если очередь пуста
45                  return 0, errors.New("очередь пуста")
46          }
47          value := q.head.value // Значение удаляемого элемента
48          if q.head == q.tail { // Если в очереди остался один элемент
49                  q.head = nil
50                  q.tail = nil
51          } else {
52                  // Обновление указателя на начало очереди
53                  q.head = q.head.next
54                  // Обновление указателя последнего элемента на новый первый элемент
55                  q.tail.next = q.head
56          }
57          q.size-- // Уменьшаем размер очереди
58          return value, nil
59  }
60
61  // Display выводит элементы очереди.
62  func (q *CircularQueue) Display() {
63          if q.head == nil { // Если очередь пуста
64                  fmt.Println("Очередь пуста")
65                  return
66          }
67          current := q.head
68          for {
```

```
69                  fmt.Printf("%d ", current.value)
70                  current = current.next // Переход к следующему узлу
71                  if current == q.head { // Проверка на завершение обхода очереди
72                          break
73                  }
74          }
75          fmt.Println()
76  }
77
78  // ClearConsole очищает консоль в зависимости от операционной системы.
79  func ClearConsole() {
80          switch runtime.GOOS {
81          case "linux", "darwin": // Для Linux и macOS
82                  cmd := exec.Command("clear") // Команда для очистки консоли
83                  cmd.Stdout = os.Stdout
84                  err := cmd.Run() // Выполнение команды
85                  if err != nil {
86                          // Обработка ошибки очистки консоли
87                          _ = errors.New("Не удалось очистить консоль: %s\n")
88                          return
89                  }
90          case "windows": // Для Windows
91                  // Команда для очистки консоли
92                  cmd := exec.Command("cmd", "/c", "cls")
93                  cmd.Stdout = os.Stdout
94                  // Выполнение команды
95                  err := cmd.Run()
96                  if err != nil {
97                          // Обработка ошибки очистки консоли
98                          _ = errors.New("Не удалось очистить консоль: %s\n")
99                          return
100                 }
101         }
102 }
103
104 // BetonicSort начинает процесс сортировки элементов в очереди.
105 func (q *CircularQueue) BetonicSort() {
106         var size int // Переменная для хранения размера очереди
107
108         // Определение максимального количества элементов, которое является степенью
        ↪  двойки
109         if q.size == 0 || (q.size&(q.size-1)) != 0 {
110                 fmt.Println("Количество элементов в очереди не является степенью двойки")
111                 fmt.Println("Будет взято максимальное количество элементов, которое
                ↪  является степенью двойки")
112                 size = q.size - 1
113         } else {
114                 size = q.size
115         }
116
117         // Извлечение элементов из очереди в массив
```

```
118         elements := make([]int, size)
119
120         for i := 0; i < size; i++ {
121                 element, _ := q.Dequeue()
122                 elements[i] = element
123         }
124
125         // Битонная сортировка
126         for k := 2; k <= size; k = k * 2 {
127                 for j := k / 2; j > 0; j = j / 2 {
128                         for i := 0; i < size; i++ {
129                                 l := i ^ j
130                                 if l > i {
131                                         if ((i&k) == 0 && elements[i] > elements[l]) ||
                                        ↪  ((i&k) != 0 && elements[i] < elements[l]) {
132                                                 // Обмен элементами
133                                                 elements[i], elements[l] = elements[l],
                                                ↪  elements[i]
134                                         }
135                                 }
136                         }
137                 }
138         }
139         // Возвращение отсортированных элементов в очередь
140         for _, elem := range elements {
141                 q.Enqueue(elem)
142         }
143 }
144
145 // main является точкой входа программы.
146 func main() {
147         reader := bufio.NewReader(os.Stdin)
148         queue := &CircularQueue{}
149
150         // Функция для вывода сообщений об ошибке красным цветом
151         errMsg := color.New(color.FgHiRed).PrintfFunc()
152         // Функция для вывода успешных сообщений зеленым цветом
153         success := color.New(color.FgHiGreen).PrintfFunc()
154
155         for {
156                 ClearConsole() // Очистить консоль
157                 fmt.Printf("\nВведите команду. Доступные команды:\n1. enqueue [число]\n2.
                ↪  dequeue\n3. sort\n4. display\n5. exit\n")
158                 cmdString, _ := reader.ReadString('\n')
159                 cmdString = strings.TrimSpace(cmdString)
160                 cmdParts := strings.Split(cmdString, " ")
161
162                 switch cmdParts[0] {
163                 case "1":
164                         if len(cmdParts) != 2 {
165                                 // Вывод сообщения об ошибке
```

```
166                                 errMsg("Необходимо указать число для добавления в
                                    ↪  очередь\n")
167                                 continue
168                         }
169                         value, err := strconv.Atoi(cmdParts[1])
170                         if err != nil {
171                                 // Вывод сообщения об ошибке
172                                 errMsg("Введите корректное число\n")
173                                 continue
174                         }
175                         queue.Enqueue(value) // Добавление элемента в очередь
176                         // Вывод успешного сообщения
177                         success("Элемент добавлен в очередь\n")
178                 case "2":
179                         value, err := queue.Dequeue()
180                         if err != nil {
181                                 // Вывод сообщения об ошибке
182                                 errMsg("%s\n", err)
183                         } else {
184                                 // Вывод успешного сообщения
185                                 success("Из очереди удален элемент: %d\n", value)
186                         }
187                 case "3":
188                         queue.BetonicSort() // Битонная сортировка очереди
189                         // Вывод успешного сообщения
190                         success("Очередь отсортирована\n")
191                 case "4":
192                         queue.Display() // Отображение элементов очереди
193                 case "5":
194                         fmt.Print("Программа завершена\n")
195                         return
196                 default:
197                         // Вывод сообщения об ошибке
198                         errMsg("Неизвестная команда\n")
199                 }
200                 fmt.Printf("\nНажмите Enter для продолжения...")
201                 _, err := reader.ReadString('\n')
202                 if err != nil {
203                         // Вывод сообщения об ошибке
204                         errMsg("Не удалось прочитать строку: %s\n", err)
205                         return
206                 }
207         }
208 }
```

1.6 Разбор цикла по шагам

case1	case2	case3	case4
47384920	3647383917253679	9443	35748234
Step 0: 47834920	Step 0: 3674383917253679	Step 0: 4943	Step 0: 35742834
Step 1: 43874920	Step 1: 3674389317253679	Step 1: 4349	Step 1: 35742843
Step 2: 34874920	Step 2: 3674389317523679	Step 2: 3449	Step 2: 34752843
Step 3: 34784920	Step 3: 3674389317523697	Sorted array: 3449	Step 3: 34754823
Step 4: 34789420	Step 4: 3476389317523697		Step 4: 34574823
Step 5: 34289470	Step 5: 3476983317523697		Step 5: 34578423
Step 6: 34209478	Step 6: 3476983312573697		Step 6: 34578432
Step 7: 24309478	Step 7: 3476983312579637		Step 7: 34378452
Step 8: 20349478	Step 8: 3476983312579736		Step 8: 34328457
Step 9: 20347498	Step 9: 3467983312579736		Step 9: 32348457
Step 10: 02347498	Step 10: 3467983312579763		Step 10: 32345487
Step 11: 02344798	Step 11: 3437986312579763		Step 11: 23345487
Step 12: 02344789	Step 12: 3433986712579763		Step 12: 23344587
Sorted array: 02344789	Step 13: 3433986792571763		Step 13: 23344578
	Step 14: 3433986797571263		Sorted array: 23344578
	Step 15: 3433986797671253		
	Step 16: 3334986797671253		
	Step 17: 3334689797671253		
	Step 18: 3334679897671253		
	Step 19: 3334679897675213		
	Step 20: 3334679897675312		
	Step 21: 3334678997675312		
	Step 22: 3334678997765312		
	Step 23: 3334678997765321		
	Step 24: 3334578997766321		
	Step 25: 3334538997766721		
	Step 26: 3334532997766781		
	Step 27: 3334532197766789		
	Step 28: 3324533197766789		
	Step 29: 3321533497766789		
	Step 30: 3321533467769789		
	Step 31: 2331533467769789		
	Step 32: 2133533467769789		
	Step 33: 2133335467769789		
	Step 34: 2133335466779789		
	Step 35: 2133335466778799		
	Step 36: 1233335466778799		
	Sorted array: 1233334566778799		

1.7 Расчет сложности алгоритма

```
import (
    "bufio"
    "errors"
    "fmt"
    "github.com/fatih/color"
    "os"
    "os/exec"
    "runtime"
    "strconv"
    "strings"
```

```
)

type Node struct {
    value int
    next  *Node
}

type CircularQueue struct {
    head *Node
    tail *Node
    size int
}
```

Здесь мы импортируем различные библиотеки и объявляем две структуры данных: ‘Node‘ и ‘CircularQueue‘. Общая сложность этого участка кода составляет $1 + 10 = 11$. Так как мы объявляем указатели на структуры.

1.7.1 Метод Enqueue

```
func (q *CircularQueue) Enqueue(value int) {
    newNode := &Node{value: value} // 2
    if q.head == nil {             // 2
        q.head = newNode           // 1
    } else {                       // 1
        q.tail.next = newNode      // 2
    }
    q.tail = newNode               // 1
    q.tail.next = q.head           // 2
    q.size++                       // 1
}
```

Здесь мы добавляем элемент в конец кольцевой очереди. Общая сложность этого метода составляет $2 + 2 + 1 + 1 + 2 + 1 = 9$.

1.7.2 Метод Dequeue

```
func (q *CircularQueue) Dequeue() (int, error) {
    if q.head == nil { // 1
        return 0, errors.New("очередь пуста") // 2
    }
    value := q.head.value // 1
    if q.head == q.tail { // 1
        q.head = nil      // 1
        q.tail = nil      // 1
    } else {              // 1
```

```
        q.head = q.head.next // 2
        q.tail.next = q.head // 2
    }
    q.size-- // 1
    return value, nil // 1
}
```

Этот метод удаляет элемент из начала очереди. Общая сложность составляет $1 + 2 + 1 + 1 + 1 + 1 + 2 + 2 + 1 = 12$.

1.7.3 Метод Display

```
func (q *CircularQueue) Display() {
    if q.head == nil { // 1
        fmt.Println("Очередь пуста") // 1
        return // 1
    }
    current := q.head // 1
    for { // n
        fmt.Printf("%d ", current.value) // n
        current = current.next // n
        if current == q.head { // n
            break // 1
        }
    }
    fmt.Println() // 1
}
```

Метод отображает элементы в очереди. Общая сложность в этом методе будет зависеть от количества элементов в очереди и составляет $1 + 1 + 1 + 1 + n(1 + 2 + n + 1) = 4n + 4$.

1.7.4 Очистка консоли

```
func ClearConsole() {
    switch runtime.GOOS {
    case "linux", "darwin": // 1
        cmd := exec.Command("clear") // 1
        cmd.Stdout = os.Stdout // 1
        err := cmd.Run() // 1
        if err != nil { // 1
            _ = errors.New("Не удалось очистить консоль: %s\n") // 1
            return // 1
        }
    case "windows": // 1
        cmd := exec.Command("cmd", "/c", "cls") // 1
        cmd.Stdout = os.Stdout // 1
```

```
        err := cmd.Run() // 1
        if err != nil { // 1
            _ = errors.New("Не удалось очистить консоль: %s\n") // 1
            return // 1
        }
    }
}
```

Здесь мы имеем условную конструкцию, которая выполняет команду очистки консоли в зависимости от операционной системы. Общая сложность этого метода равна $1 + 7 \times (1 + 1 + 1 + 1 + 1 + 1 + 1) = 56$.

1.7.5 Сортировка

“‘latex

```
func (q *CircularQueue) BetonicSort() {
    // Количество элементов
    var size int // 1

    if q.size == 0 || (q.size&(q.size-1)) != 0 { // 2
        fmt.Println("Количество элементов в очереди не является степенью двойки") // 1
        fmt.Println("Будет взято максимальное количество элементов, которое является
        ↪  степенью двойки") // 1
        size = q.size - 1 // 1
    } else { // 1
        size = q.size // 1
    }

    elements := make([]int, size) // 2

    for i := 0; i < size; i++ { // 1 + 2n + n(1 + 1)
        element, _ := q.Dequeue() // n(12)
        elements[i] = element // n
    }

    for k := 2; k <= size; k = k * 2 { // logn * (1 + logn * (1 + n(2 + 2 + 2n)))
        for j := k / 2; j > 0; j = j / 2 { // logn * (1 + logn * (1 + n(1 + 2n)))
            for i := 0; i < size; i++ { // logn * n * (1 + 2n)
                l := i ^ j // n
                if l > i { // n
                    if ((i&k) == 0 && elements[i] > elements[l]) || ((i&k) != 0 &&
                    ↪  elements[i] < elements[l]) { // n * (2 + 2)
                        elements[i], elements[l] = elements[l], elements[i] // n
                    }
                }
            }
        }
    }
```

```
32
33      for _, elem := range elements { // n(2)
34          q.Enqueue(elem) // n(9)
35      }
36  }
```

Этот метод выполняет битонную сортировку элементов в очереди. Общая сложность этого метода зависит от размера очереди и составляет $(2 + 1 + 2 + n(12) + 1 + 2 + \log n \times (1 + \log n \times (1 + n(1 + 2n)) + 2) + n(2 + 9)) = 14n^2\log^2 n + 25n\log n + 12n + 10$.

1.7.6 Точка входа программы

```
1   func main() {
2       reader := bufio.NewReader(os.Stdin) // 1
3       queue := &CircularQueue{} // 1
4
5       errMsg := color.New(color.FgHiRed).PrintfFunc() // 1
6       success := color.New(color.FgHiGreen).PrintfFunc() // 1
7
8       for { // n
9           ClearConsole() // 56
10          fmt.Printf("\nВведите команду. Доступные команды:\n1. enqueue [число]\n2.
            ↪  dequeue\n3. sort\n4. display\n5. exit\n") // n(1)
11          cmdString, _ := reader.ReadString('\n') // n
12          cmdString = strings.TrimSpace(cmdString) // n
13          cmdParts := strings.Split(cmdString, " ") // n
14
15          switch cmdParts[0] { // n
16          case "1": // 1
17              if len(cmdParts) != 2 { // 1
18                  errMsg("Необходимо указать число для добавления в очередь\n") // 1
19                  continue // 1
20              }
21              value, err := strconv.Atoi(cmdParts[1]) // 2
22              if err != nil { // 1
23                  errMsg("Введите корректное число\n") // 1
24                  continue // 1
25              }
26              queue.Enqueue(value) // 9
27              success("Элемент добавлен в очередь\n") // 1
28          case "2": // 1
29              value, err := queue.Dequeue() // 12
30              if err != nil { // 1
31                  errMsg("%s\n", err) // 1
32              } else { // 1
33                  success("Из очереди удален элемент: %d\n", value) // 1
34              }
35          case "3": // 1
36              queue.BetonicSort() // 14n^2 log^2n + 25n logn + 12n + 10
```

```
37              success("Очередь отсортирована\n") // 1
38          case "4": // 1
39              queue.Display() // 4n + 4
40          case "5": // 1
41              fmt.Print("Программа завершена\n") // 1
42              return // 1
43          default: // 1
44              errMsg("Неизвестная команда\n") // 1
45          }
46          fmt.Printf("\nНажмите Enter для продолжения...") // n
47          _, err := reader.ReadString('\n') // n
48          if err != nil { // 1
49              errMsg("Не удалось прочитать строку: %s\n", err) // 1
50              return // 1
51          }
52      }
53  }
```

Это основная функция программы, которая обрабатывает ввод пользователя и вызывает соответствующие методы. Общая сложность этой функции зависит от числа операций в цикле и составляет $(1+1+1+1+n(1+1+1+1+1+1+9+1+1+1+1)+1+n+n(1+1)+1) = 14n + 10$.

1.7.7 Финальный вывод

Теперь, чтобы вычислить финальную асимптотическую сложность, нужно сложить сложности всех участков кода:

Суммарная сложность $= 11+9+12+(4n+4)+56+(14n^2 \log^2 n+25n \log n+12n+10) + (14n + 10) = 14n^2 \log^2 n + 25n \log n + 41n + 112 + 90$

Таким образом, финальная сложность алгоритма будет $O(n^2 \log^2 n)$.

1.8 Тесты

1.8.1 CircularQueueBetonicTest.go

Ниже представлен полный листинг unit тестов.

```
1   package main
2
3   import (
4           "testing"
5   )
6
7   // Функция для создания очереди с n элементами, где n - степень двойки.
8   func createQueueWithNElements(n int) *CircularQueue {
```

```
9           q := &CircularQueue{}
10          for i := 0; i < n; i++ {
11                  q.Enqueue(i)
12          }
13          return q
14  }
15
16  // TestEnqueueDequeue проверяет корректность добавления и удаления элементов из очереди.
17  func TestEnqueueDequeue(t *testing.T) {
18          q := createQueueWithNElements(8) // Создаем очередь с 8 элементами
19
20          // Проверяем размер очереди после добавления элементов
21          if q.size != 8 {
22                  t.Errorf("Expected queue size of 8, got %d", q.size)
23          }
24
25          // Удаляем элементы и проверяем их значения
26          for i := 0; i < 8; i++ {
27                  val, err := q.Dequeue()
28                  if err != nil {
29                          t.Fatalf("Dequeue error: %v", err)
30                  }
31                  if val != i {
32                          t.Errorf("Expected value %d, got %d", i, val)
33                  }
34          }
35
36          // Проверяем, что очередь пуста после удаления всех элементов
37          if q.size != 0 {
38                  t.Errorf("Expected empty queue, got size %d", q.size)
39          }
40  }
41
42  // TestBetonicSort проверяет корректность битонной сортировки.
43  func TestBetonicSort(t *testing.T) {
44          q := createQueueWithNElements(8) // Создаем очередь с 8 элементами
45          q.BetonicSort()                  // Сортировка
46
47          // Проверяем, что элементы отсортированы
48          prevVal, _ := q.Dequeue()
49          for q.size > 0 {
50                  val, _ := q.Dequeue()
51                  if prevVal > val {
52                          t.Errorf("Queue is not sorted: %d > %d", prevVal, val)
53                  }
54                  prevVal = val
55          }
56  }
57
58  // BenchmarkBetonicSort бенчмарк для битонной сортировки.
59  func BenchmarkBetonicSort(b *testing.B) {
```

```
60          for i := 0; i < b.N; i++ {
61                  q := createQueueWithNElements(8) // Создаем очередь с 8 элементами
62                  q.BetonicSort()                  // Сортировка
63          }
64  }
```

1.8.2 Вывод результатов

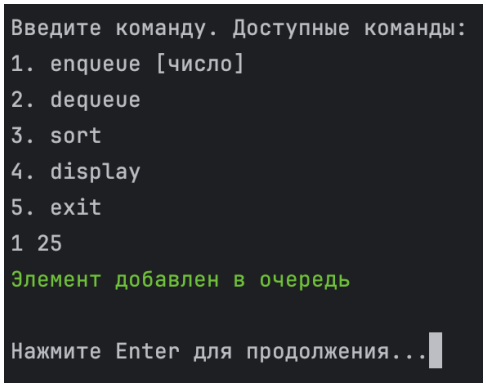
Тесты

```
1  /usr/local/go/bin/go tool test2json -t
   ↪  /Users/nikitabelekov/Library/Caches/JetBrains/GoLand2023.3/tmp/GoLand/___go_test_github_com_17
   ↪  -test.v -test.paniconexit0
2  === RUN   TestEnqueueDequeue
3  --- PASS: TestEnqueueDequeue (0.00s)
4  === RUN   TestBetonicSort
5  --- PASS: TestBetonicSort (0.00s)
6  PASS
7
8  Process finished with the exit code 0
```

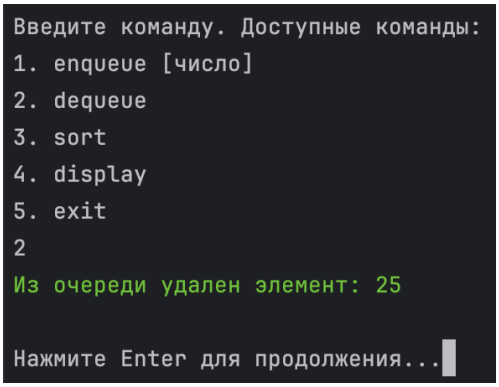
Бенчмарки

```
1  /Users/nikitabelekov/Library/Caches/JetBrains/GoLand2023.3/tmp/GoLand/___BenchmarkBetonicSort_in_g
   ↪  -test.v -test.paniconexit0 -test.bench ^\QBenchmarkBetonicSort\E$ -test.run ^$
2  goos: darwin
3  goarch: arm64
4  pkg: github.com/17HIERARCH70/algosITMO_BIT/lab4
5  BenchmarkBetonicSort
6  BenchmarkBetonicSort-8            2941936               406.5 ns/op
7  PASS
8
9  Process finished with the exit code 0
```

1.8.3 Ручное тестирование

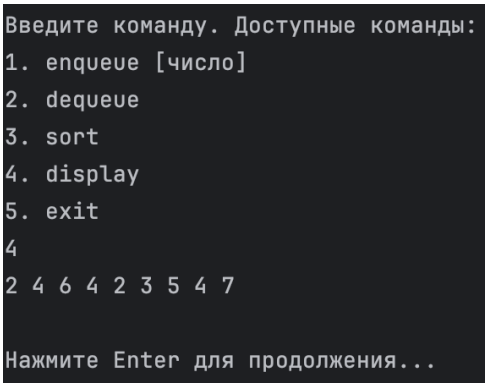


(а)

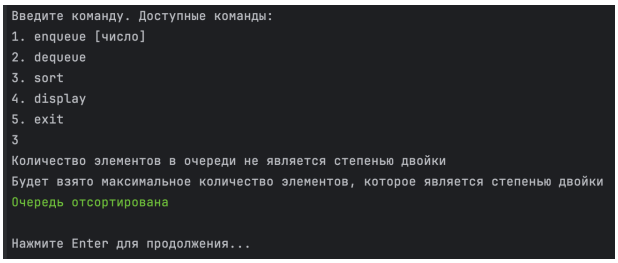


(б)

Рисунок 1.6

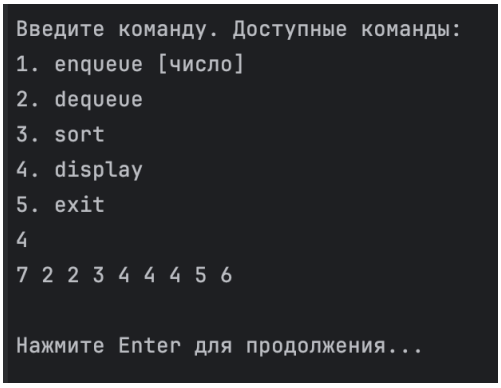


(а)

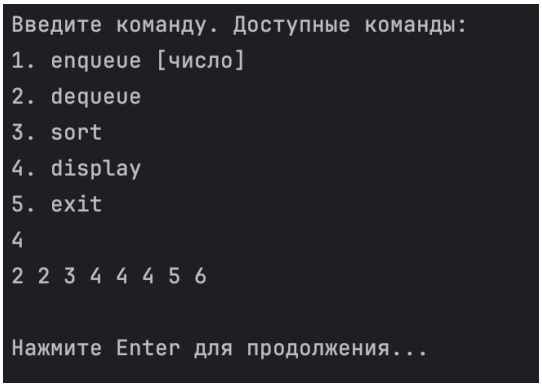


(б)

Рисунок 1.7



(а)



(б)

Рисунок 1.8

ЗАКЛЮЧЕНИЕ

Задача: Разработать программу битонной сортировки, используя для хранения данных кольцевую очередь на базе связного списка.

Реализованы функции:

1. Добавление элемента в очередь.
2. Взятие элемента из очереди.
3. Вывод всех элементов в очереди.
4. Битонная сортировка для очереди.

Среда запуска: Go 1.22.0 (выпущена 2024-02-06)

Редактор: Visual Studio Code

Все тесты были успешно пройдены.