МИНОБР НАУКИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА — Российский технологический университет»

РТУ МИРЭА

Отчёт по выполнению практического задания 3
Тема: Рекурсивные алгоритмы и их реализация
Дисциплина Структуры и алгоритмы обработки данных

Выполнил студент Пак С.А.

группа ИКБО-05-20

Москва 2021

СОДЕРЖАНИЕ

ОТВЕТЫ НА ВОПРОСЫ

Дайте определение понятиям.

1) Определение рекурсивной фукнции. Рекурсивная функция — это объект, частично состоящий из самого себя или определяемый с помощью себя.

2) Шаг рекурсии. Шаг рекурсии — это активизация очередного рекурсивного выполнения алгоритма при других исходных данных.

3) Глубина рекурсии. Глубина рекурсии — это наибольшее одновременное количество рекурсивных вызовов функции, определяющее максимальное количество слоёв рекурсивного стека, в котором осуществляется хранение отложенных вычислений.

4) Условие завершения рекурсии. Условие завершения рекурсии — это условие, которое определяет завершение рекурсии и формирование конкретного простейшего решения вычислительного процесса.

5) Виды рекурсии:

Линейная рекурсия — это рекурсия, при которой каждый вызов порождает ровно один новый вызов;

Каскадная рекурсия — это рекурсия, при которой каждый вызов порождает несколько новых вызовов.

6) Прямая и косвенная рекурсия.

Прямая рекурсия имеет место, если решение задачи сводится к разделению её на меньшие подзадачи, выполняемые с помощью одного и того же алгоритма.

Косвенная рекурсия имеет место, если алгоритм А вызывает алгоритм В, и алгоритм В вновь вызывает алгоритм А.

7) Организация стека рекурсивных вызовов. При каждом новом рекурсивном вызове функции в стеке создаётся новое множество локальных переменных и форменных параметров, их имена одинаковы, но они имеют различные значения.

ОТЧЁТ ПО ЗАДАЧЕ 1

Номер варианта: 4.

1. Условие и постановка задачи

Определить, является ли текст — палиндромом.

2. Описание алгоритма — рекуррентная зависимость

Для начала нужно описать функцию и её параметры. Пусть функция будет называться *isPalindrome*, которая имеет три параметра: *str* — исследуемая строка; *start* и *end* — индексы, обозначающие границы исследуемой подстроки. То есть исследуемая подстрока будет состоять из символов под индексами со *start* до *end* не включительно.

Теперь сам алгоритм:

1) Если индекс *start* равен индексу *end*, то функция должна вернуть значение *true* (это значит, что либо строка состоит из одного символа, либо функция дошла ровно до середины строки, так как строка содержит нечётное количество символов);

2) Если индекс *start* больше индекса *end*, то функция должна вернуть значение выражения *str[start] = str[end]* (это значит, что строка содержит чётное количество символов, поэтому нужно проверить равенство двух элементов, находящихся в середине строки);

3) Если символ строки, находящийся на позиции *start*, не совпадает с символом, находящимся на позиции *end*, то нужно вернуть значение *false*, так как строка уже не является палиндромом;

4) Если условия всех пунктов с 1-го по 3-й не выполнились, то нужно вернуть значение *true && isPalindrome(str, start + 1, end)*.

Таким образом, когда рекурсия завершится, на выходе получится конъюнкт. Поэтому если хотя бы один рекурсивный вызов вернёт *false*, то и значение всего логического выражения будет равно *false*.

В результате строка будет являться палиндромом тогда и только тогда, когда все рекурсивные вызовы вернут значение *true*.

На основе алгоритма составим рекуррентную зависимость (формула 1):

$$isPalindrome(str, start, end) = \begin{cases} true, start = end, \\ str[start] = str[end], start > end, \\ false, str[start] \neq str[end], \\ true \wedge isPalindrome(str, start, end) \end{cases} \quad (1)$$

3. Коды используемых функций

Все функции, которые выполняют поставленную задачу содержатся в файле *tasks.ts* (.ts — расширение для TypeScript файлов).

1. Итеративная реализация функции

```
/**
 * Возвращает true, если строка, переданная в качестве аргумента, является
палиндромом
 * @param str         исследуемая строка
 */
export const isPalindromeIterative = (str: string): boolean => {
  let left: number = 0;
  let right: number = str.length - 1;

  while (left < right) {
    if (str[left++] !== str[right--]) {
      return false;
    }
  }

  return true;
}
```

2. Рекурсивная реализация функции

```
/**
 * Возвращает true, если строка, переданная в качестве аргумента, является
палиндромом
 * @param str         исследуемая строка
 * @param start       начальная позиция исследуемой подстроки
 * @param end         конечная позиция исследуемой подстроки (не включая эту
позицию)
 */
export const isPalindromeRecursive = (str: string, start: number, end: number):
boolean => {
  --end;

  if (start >= end) {     // строка закончилась
    return (
      (start === end)     // середина строки
        ? true
        : (str[start] === str[end])
    );
  }

  if (str[start] !== str[end]) {
    return false;
  }

  return (true && isPalindromeRecursive(str, start + 1, end));
}
```

4. Ответы на задания по задаче 1

1) Приведите итерационный алгоритм решения задачи. Алгоритм решения задачи следующий:

1. Создание двух переменных *left* и *right*, который являются индексами;

2. Значение переменной *left* — 0, а *right* — длина строки, переданной в качестве аргумента минус 1, так как индексация начинается с нуля;

3. В цикле проверяем на равенство символы под индексами *left* и *right*

4. Если они совпали, то значение *left* увеличиваем на единицу, а *right* уменьшаем на единицу;

5. Если не совпали, то возвращаем логическое значение *false*;

6. Цикл нужно прервать, когда значение переменной *left* станет больше либо равно значению *right*;

7. Таким образом, полное завершение цикла возможно только в том случае, если все символы прошли проверку п.3. Это значит, что строка является палиндромом. Следовательно нужно вернуть логическое значение *true*.

2) Реализация итеративного алгоритма решения задачи:

```
/**
 * Возвращает true, если строка, переданная в качестве аргумента, является
палиндромом
 * @param str         исследуемая строка
 */
export const isPalindromeIterative = (str: string): boolean => {
  let left: number = 0;
  let right: number = str.length - 1;

  while (left < right) {
    if (str[left++] !== str[right--]) {
      return false;
    }
  }

  return true;
}
```

3) Определение теоретической сложности итеративного алгоритма

Ниже представлен процесс вычисления теоретической сложности алгоритма (табл.1). В таблице 1 *n* — это количество символов в строке.

Таблица 1

Операторы	Время выполнения инструкции	Количество выполнений оператора
`left ← 0`	C_1	1
`right ← n - 1`	C_2	1
`while (left < right) do`	C_3	$\sum_{1}^{\frac{n}{2}} 1$
`if (str[left] != str[right]) then`	C_4	$\sum_{1}^{\frac{n}{2}} 1$
`return false`	C_5	1

Продолжение таблицы 1

Операторы	Время выполнения инструкции	Количетсво выполнений оператора
`endif`		
`left = left + 1`	C_6	$\sum_{1}^{\frac{n}{2}} 1$
`right = right - 1`	C_7	$\sum_{1}^{\frac{n}{2}} 1$
`od`		
`return true`	C_8	1

Таким образом, теоретическая сложность алгоритма будет вычисляться следующим образом (формула 2):

$$T(n)=C_1\cdot 1+C_2\cdot 1+C_3\cdot\sum_{1}^{\frac{n}{2}}1+C_4\cdot\sum_{1}^{\frac{n}{2}}1+C_5\cdot 1+C_6\cdot\sum_{1}^{\frac{n}{2}}1+C_7\cdot\sum_{1}^{\frac{n}{2}}1+C_8\cdot 1= \\ =(C_1+C_2+C_5+C_8)+(C_3+C_4+C_6+C_7)\cdot\sum_{1}^{\frac{n}{2}}1 \quad (2)$$

В <u>худшем случае</u>, когда цикл выполнится $\frac{n}{2}$ раз, сумма $\sum_{1}^{\frac{n}{2}}1=\frac{n}{2}$, поэтому формула теоретической сложности примет вид (формула 3):

$$T(n)=(C_1+C_2+C_5+C_8)+(C_3+C_4+C_6+C_7)\cdot\frac{n}{2}=A+B\cdot\frac{n}{2} \quad (3)$$

где, *A*, *B = const*.

В <u>лучшем случае</u>, когда цикл выполнится лишь один раз, так как символы на концах строки не равны друг другу, сумма $\sum_{1}^{\frac{n}{2}}1=1$, поэтому формула теоретической сложности примет вид (формула 4):

$$T(n)=(C_1+C_2+C_5+C_8)+(C_3+C_4+C_6+C_7)=A+B \quad (4)$$

uде *A*, *B = const*.

4) <u>Описание рекуррентной зависимости в решении задачи</u>

Рекуррентная зависимость уже была выведена в п.2 отчёта по задаче 1. Поэтому рекуррентное соотношение выражено формулой 1.

5) Реализация рекурсивной функции решения задачи

```
/**
 * Возвращает true, если строка, переданная в качестве аргумента, является
палиндромом
 * @param str         исследуемая строка
 * @param start       начальная позиция исследуемой подстроки
 * @param end         конечная позиция исследуемой подстроки (не включая эту
позицию)
 */
export const isPalindromeRecursive = (str: string, start: number, end: number):
boolean => {
  --end;

  if (start >= end) {     // строка закончилась
    return (
      (start === end)     // середина строки
        ? true
        : (str[start] === str[end])
    );
  }

  if (str[start] !== str[end]) {
    return false;
  }

  return (true && isPalindromeRecursive(str, start + 1, end));
```

6) Определение глубины рекурсии

Пусть *«aba»* — строка, которая подаётся на вход рекурсивной функции (рис.1).

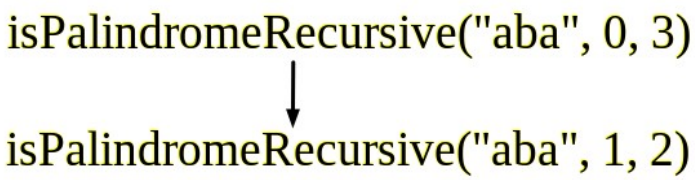

Рис.1 Дерево вызовов рекурсивной функции ч.1

Получается, что вершин в дереве 2 штуки, а листьев также 2 штуки, но два вызова расположены друг за другом без всяких ветвлений, поэтому глубина рекурсии будет равна 2.

Пусть *«abba»* — строка, которая подаётся на вход рекурсивной функции (рис.2).

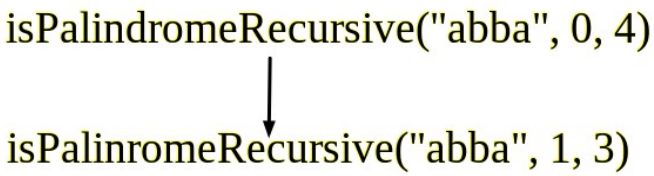

Рис.2 Дерево вызовов рекурсивной функции ч.2

Опять вершин в дереве 2, и колчиство листьев также равно 2, но два вызова расположены друг за другом, поэтому глубина рекурсии будет равна 2.

7) Определение сложности рекурсивного алгоритма

Для начала определим рекуррентное соотношение для времени выполнения (формула 5):

$$T(n)=\begin{cases}\theta(1), & start \geq end, \\ \theta(1), & str[start] \neq str[end], \\ T(n-2) & \end{cases} \quad (5)$$

Определим верхнюю границу рекуррентного соотношения (формула 6):

$$T(n)=T(n-2) \quad (6)$$

Теперь используем метод подстановки. Пусть *c = const*. Предположим, что уравнение $T(n)=T(n-2)$ имеет решение $T(n)=O(n)$. Докажем, что при подходящем выборе константы *c > 0* выполняется неравенство $T(n)\leq c\cdot n$.

Предположим справедливость этого неравенства для величины *n-2*, т.е. что выполняется соотношение $T(n-2)\leq c\cdot(n-2)$. После подстановки данного выражения в (6) получаем:

$$T(n)\leq c\cdot(n-2)\leq c\cdot n \quad (7)$$

Таким образом выражение (7) выполняется при всех *n* и *c*. Поэтому сложность данного рекурсивного алгоритма будет равна *O(n)*.

Определим теоретическую сложность алгоритма методом деревьев рекурсий. Пусть *c = const*.

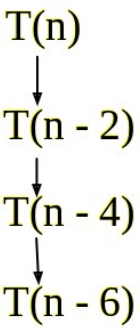

Рис.3 Метод деревьев рекурсии

Пусть глубина рекурсии — *k*, т.е. в полном дереве рекурсии *k* уровней. Значение каждого из узлов на уровне *k*: $T(n-2k)$. Время выполнения на каждом уровне: *c*. Общее время: *ck*. Найдём *k* (формула 8): т.к. $n-2k=1$, то

$$k=\frac{n-1}{2} \quad (8)$$

Следовательно общее время выполнения (формула 9):

$$ck=c\frac{(n-1)}{2} \quad (9)$$

Соответственно, теоретическая сложность рекурсивная алгоритма будет следующей (формула 10):

$$T(n)=O(n) \quad (10)$$

8) Пример схемы рекурсивных вызовов

Пусть строка *«mygym»* подаётся на вход рекурсивной функции. Тогда схема рекурсивных вызовов будет следующей: *isPalindrome(«mygym», 0, 5)* → *isPalindrome(«mygym», 1, 4)* → *isPalindrome(«mygym», 2, 3)*. Последний вызов вовращает логическое значени *true*.

В результате вся функция возвращает логическое значение *true*.

9) Код программы, демонстрирующей выполнение обеих функций

Коды рекурсивной и итеративной функции можно найти в п.3 отчёта по задаче 1. Поэтому ниже показан код основной программы, которая содержится в файле *index.ts*.

```
import * as tasks from "./modules/tasks/tasks";
import * as consts from "./constants";
```

```
import * as fs from "fs";

// подобие потока ввода
const STDIN: string = fs.readFileSync(consts.PATH_TO_INPUT_FILES +
consts.INPUT_FILE_NAME, "utf-8");


/**
 * Возвращает строку, в которой удалены все пробелы, и все буквы в нижнем
регистре
 * @param str         нормализуемая строка
 */
const normalizeString = (str: string): string => {
  return str.split("").filter((char: string) => !(char === "
")).join("").toLowerCase();
}


/**
 * Основная программа
 */
const main = async (): Promise<void> => {
  STDIN.split("\n").forEach((line: string) => {
    const nLine: string = normalizeString(line);

    if (tasks.isPalindromeRecursive(nLine, 0, nLine.length)) {
      console.log(`[РЕК.]: Текст {${line}} является палиндромом`);
      return;
    }

    console.log(`[РЕК.]: Текст {${line}} не является палиндромом`);
  });

  console.log();

  STDIN.split("\n").forEach((line: string) => {
    const nLine: string = normalizeString(line);

    if (tasks.isPalindromeIterative(nLine)) {
      console.log(`[ИТЕР.]: Текст {${line}} является палиндромом`);
      return;
    }

    console.log(`[ИТЕР.]: Текст {${line}} не является палиндромом`);
  });
}


// запуск функции с отловом ошибок
main()
  .catch((err: Error) => {
    console.error("[ERROR]:", err.message);
  });
```

ОТЧЁТ ПО ЗАДАЧЕ 2

1. Условие и постановка задачи

Удалить из связанного однонаправленного списка все элементы, равные заданному.

2. Описание алгоритма — рекуррентная зависимость

Для начала нужно определить рекурсивную функцию и её параметры. Пусь рекурсивная функция *r* имеет два параметра: *node* (элемент списка, с которого начинается процесс удаления); *el* (информационная часть элементов, которые нужно удалить).

Теперь сам алгоритм:

1) Если *node* имеет значение *undefined*, либо элемент, следующий за *node* является последним элементом в списке, и при этом его не нужно удалять, то нужно завершить исполнение вызова функции;

2) Если *node* является последним элементом, и его нужно удалить, то этому *node* нужно присвоить значение *undefined*. Таким образом, этот элемент станет «окончанием» списка;

3) Если нужно удалить *node* (это возможно только если *node* на данный момент является головным элементом), то *node* нужно присвоить значение элемента, следующего за *node*. Таким образом, текущее значение *node* теряется, и получается своеобразное удаление.

Затем нужно вызвать функцию *r* с аргументами *node*, *el*;

4) Если нужно удалить элемент, который находится после *node*, то всё, что следует за удаляемым элементом, теперь должно следовать после *node*. Таким образом, удаляемый элемент «потеряется».

Затем нужно вызвать функцию *r* с аргументами *node*, *el*;

5) Если не выполняется ни одно из условий п.1-4, то нужно перейти к проверку следующего за *node*, элемента. Поэтому нужно вызвать функцию *r* с аргументами *node.next* (узел, который находится после текущего узла) и *el*.

Теперь на основе алгоритма, определим рекуррентную зависимость (формула 11):

$$r(node,el)=\begin{cases} \varnothing, \ node=undefined, \\ \varnothing, \ node.next=undefined \text{ и } node.val=el, \\ node=undefined,\ node.next=undefined \text{ и } node.val=el, \\ node=node.next \Rightarrow r(node,el),\ node.val=el, \\ node.next=node.next.next \Rightarrow r(node,el),\ node.next.val=el, \\ r(node.next,el) \end{cases} \tag{11}$$

3. Коды используемых функций

Для решения данной задачи используется лишь одна функция *removeAllElements*, которая является аналогом функции *r(node, el)* из предыдущего пункта. Она также как и функция *r* имеет два параметра *node* и *element*. *removeAllElements* содержится в файле *tasks.ts*.

```
/**
 * Удаляет все вхождения элемента в списке
 * @param node      головной элемент списка
 * @param element   информационная часть удаляемого элемента
 */
export const removeAllElements = <T>(node: ListNode<T> | undefined, element: T):
void => {
  if (node === undefined || (node.next === undefined && node.value !== element))
{ // нельзя удалить элемент со значением undefined, либо функция дошла до
последнего элемента, и его не нужно удалять
    return;
  }

  if (node.next === undefined && node.value === element) {
    node = undefined;
    return;
  }

  if (node.value === element) { // нужно удалить головной элемент
    // казалось бы
    // node = node.next   должно сработать
    // но TypeScript решил иначе))

    node.value = node.next!.value;  // обмен информационными частями со
следующим элементом
    node.next = node.next!.next;
    removeAllElements(node, element);
    return;
  }

  if (node.next!.value === element) {  // нужно удалить элемент после текущего
    // node.next - удаляемый узел в списке
    // node - элемент, который находится перед удаляемым
    node.next = node.next!.next;
    removeAllElements(node, element);
    return;
  }

  removeAllElements(node.next, element);
}
```

4. Ответы на задания по задаче 2

1) Разработайте рекурсивную функцию для обработки списковой структуры

Код функции, которая обрабатывает списковую структуру и, согласно варианту, удаляет все вхождения элемента с заданной информационной частью, можно посмотреть в предыдущем пункте (п.3) отчёта по задаче 2.

2) Функция создания списка

Для создания списка используется метод *add* класса *List*. Он принимает в качестве аргумента элемент(ы), которые нужно добавить, и добавляет его (их) в начало списка.

Данный метод содержится в файле *List.ts*, который также содержит реализацию связанного однонаправленного списка.

```
/**
 * Добавляет элемент или элементы в начало списка
 * @param elements        добавляемый(е) элемент(ы)
 */
public add(...elements: Array<T>): void {
  elements.forEach((element: T) => {
    let tempNode: ListNode<T> = new ListNode(element);

    if (this._header === undefined) {   // список является пустым
      this._header = tempNode;
      this._header.next = undefined;
      return;
    }

    // this._header на данном этапе не может быть равен undefined
    [this._header!.value, tempNode.value] = [tempNode.value,
this._header!.value];

    tempNode.next = this._header!.next;
    this._header!.next = tempNode;
  });
}
```

3) Определение глубины рекурсии

Если посмотреть на исходный код функции, то можно заметить, что каждый из рекурсивных вызовов функции будет идти один за другим без всяких ветвлений, потому что за каждый вызов функция вызывает саму себя не более чем один раз.

Поэтому глубина рекурсии будет равна $n-1$, где n — количество вызовов функции (рис.4).

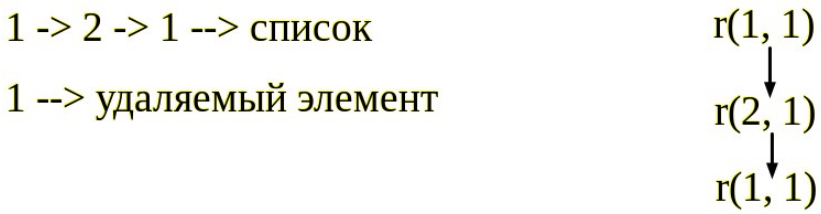

Рис.4 Глубина рекурсии для функции *r*

4) Определение теоретической сложности алгоритма

Пусть глубина рекурсии — *k*, т.е. в полном дереве рекурсии *k* уровней. Значение каждого из узлов на уровне *k*: $T(n-k-1)$. Время выполнения на каждом уровне: *c*. Общее время: *ck*. Найдём *k* (формула 12): т.к. $n-k-1=1$, то

$$k = n - 2 \quad (12)$$

Следовательно общее время выполнения (формула 13):

$$ck = c(n-2) \quad (13)$$

Соответственно, теоретическая сложность рекурсивная алгоритма будет следующей (формула 14):

$$T(n) = O(n) \quad (14)$$

5) Разработка программы и результаты тестирования

Чтобы в очередной раз не повторяться, код программы и результаты тестирования можно посмотреть в п.5 отчёта по задаче 5.

Здесь лишь отмечу, что код основной программы находится в файле *index.ts*. Также стоит сказать, что *TypeScript* не имеет встроенных средств для ввода данных с клавиатуры, поэтому пришлось использовать файловый ввода.

Ещё один момент. В реализации списка и узла списка используются так называемые «обобщения», который позволяют указываеть любой тип данных для информационной части элементов списка.

5. Код программы и результаты тестирования

1. Файл index.ts

```
import * as fs from "fs";

import * as consts from "./constants";
import List from "./modules/list/List";
import { removeAllElements } from "./modules/tasks/tasks";

// подобие потока ввода
const STDIN: string = fs.readFileSync(consts.PATH_TO_INPUT_FILES +
consts.INPUT_FILE_NAME, "utf-8");


/**
 * Основная программа
 */
const main = async (): Promise<void> => {
  let list: List<number> = new List();

  const addingElements: Array<number> = STDIN.split("\n")[0].split("
").map((num: string) => +num);
  const deletingElement: number = +STDIN.split("\n")[1];

  list.add(...addingElements);
```

```
  console.log(`Элемент, который нужно удалить: ${deletingElement}`)
  console.log();

  console.log(`До удаления: ${list.toString()}`);
  removeAllElements(list.header, deletingElement);

  console.log(`После удаления: ${list.toString()}`);
  console.log();
}


// запуск функции с отловом ошибок
main()
  .catch((err: Error) => {
    console.error("[ERROR]:", err.message);
  });
```

2. Файл tasks.ts

```
import ListNode from "../node/ListNode";

/**
 * Возвращает true, если строка, переданная в качестве аргумента, является
палиндромом
 * @param str         исследуемая строка
 * @param start       начальная позиция исследуемой подстроки
 * @param end         конечная позиция исследуемой подстроки (не включая эту
позицию)
 */
export const isPalindromeRecursive = (str: string, start: number, end: number):
boolean => {
  --end;

  if (start >= end) {     // строка закончилась
    return (
      (start === end)     // середина строки
        ? true
        : (str[start] === str[end])
    );
  }

  if (str[start] !== str[end]) {
    return false;
  }

  return (true && isPalindromeRecursive(str, start + 1, end));
}


/**
 * Возвращает true, если строка, переданная в качестве аргумента, является
палиндромом
 * @param str         исследуемая строка
 */
export const isPalindromeIterative = (str: string): boolean => {
  let left: number = 0;
  let right: number = str.length - 1;
```

```
  while (left < right) {
    if (str[left++] !== str[right--]) {
      return false;
    }
  }

  return true;
}


/**
 * Удаляет все вхождения элемента в списке
 * @param node      головной элемент списка
 * @param element   информационная часть удаляемого элемента
 */
export const removeAllElements = <T>(node: ListNode<T> | undefined, element: T):
void => {
  if (node === undefined || (node.next === undefined && node.value !== element))
{ // нельзя удалить элемент со значением undefined, либо функция дошла до
последнего элемента, и его не нужно удалять
    return;
  }

  if (node.next === undefined && node.value === element) {
    node = undefined;
    return;
  }

  if (node.value === element) { // нужно удалить головной элемент
    // казалось бы
    // node = node.next   должно сработать
    // но TypeScript решил иначе))

    node.value = node.next!.value;  // обмен информационными частями со
следующим элементом
    node.next = node.next!.next;
    removeAllElements(node, element);
    return;
  }

  if (node.next!.value === element) {  // нужно удалить элемент после текущего
    // node.next - удаляемый узел в списке
    // node - элемент, который находится перед удаляемым
    node.next = node.next!.next;
    removeAllElements(node, element);
    return;
  }

  removeAllElements(node.next, element);
}
```

3. Файл List.ts

```
import ListNode from "../node/ListNode";
import { cloneDeep } from "lodash";

/**
 * Представляет структуру данных однонаправленный список
```

```
 */
class List<T> {
  private _header: ListNode<T> | undefined;


  /**
   * Создаёт пустой список
   */
  public constructor() {
    this._header = undefined;
  }


  /**
   * Возвращает ссылку на головной элемент списка
   */
  public get header(): ListNode<T> | undefined {
    return this._header;
  }


  /**
   * Устанавливает новую ссылку на головной элемент списка
   * @param header        новая ссылка на головной элемент списка
   */
  public set header(header: ListNode<T> | undefined) {
    this._header = header;
  }


  /**
   * Добавляет элемент или элементы в начало списка
   * @param elements        добавляемый(е) элемент(ы)
   */
  public add(...elements: Array<T>): void {
    elements.forEach((element: T) => {
      let tempNode: ListNode<T> = new ListNode(element);

      if (this._header === undefined) {   // список является пустым
        this._header = tempNode;
        this._header.next = undefined;
        return;
      }

      // this._header на данном этапе не может быть равен undefined
      [this._header!.value, tempNode.value] = [tempNode.value,
this._header!.value];

      tempNode.next = this._header!.next;
      this._header!.next = tempNode;
    });
  }


  /**
   * Возвращает строковое представление однонаправленного списка
   */
  public toString(): string {
    if (this._header === undefined) { // список пуст
```

```
      return "[]";
    }

    let copyHeader: ListNode<T> | undefined = cloneDeep(this._header);
    let strList: string = `[${copyHeader!.value}`;

    copyHeader = copyHeader!.next;

    while (copyHeader !== undefined) {
      strList += `, ${copyHeader.value}`;
      copyHeader = copyHeader.next;
    }

    strList += `]`;
    return strList;
  }
}


/* Экспорт для использования структуры в других файлах */
export default List;
```

4. Файл ListNode.ts

```
/**
 * Представляет элемент однонаправленного списка
 */
class ListNode<T> {
  private _value: T;
  private _next: ListNode<T> | undefined;


  /**
   * Создаёт элемент списка
   * @param value         значение информационной части
   * @param next          ссылка на следующий элемент списка
   */
  public constructor(value: T, next?: ListNode<T>) {
    this._value = value;
    this._next = next;
  }


  /**
   * Возвращает значение информационной части элемента списка
   */
  public get value(): T {
    return this._value;
  }


  /**
   * Устанавливает новое значение информационной части элемента списка
   * @param value         новой значение информационной части
   */
  public set value(value: T) {
    this._value = value;
  }
```

```
  /**
   * Возвращает ссылку на следующий элемент списка
   */
  public get next(): ListNode<T> | undefined {
    return this._next;
  }


  /**
   * Устанавливает новую ссылку на следующий элемент списка
   * @param next        новая ссылка на следующий элемент списка
   */
  public set next(next: ListNode<T> | undefined) {
    this._next = next;
  }
}


/* Экспорт для использования класса в других файлах */
export default ListNode;
```

5. Результаты тестирования

Ниже показаны результаты тестирования функции (рис.5-8). Опять данные для списка и удаляемого элемента считываются из файла.

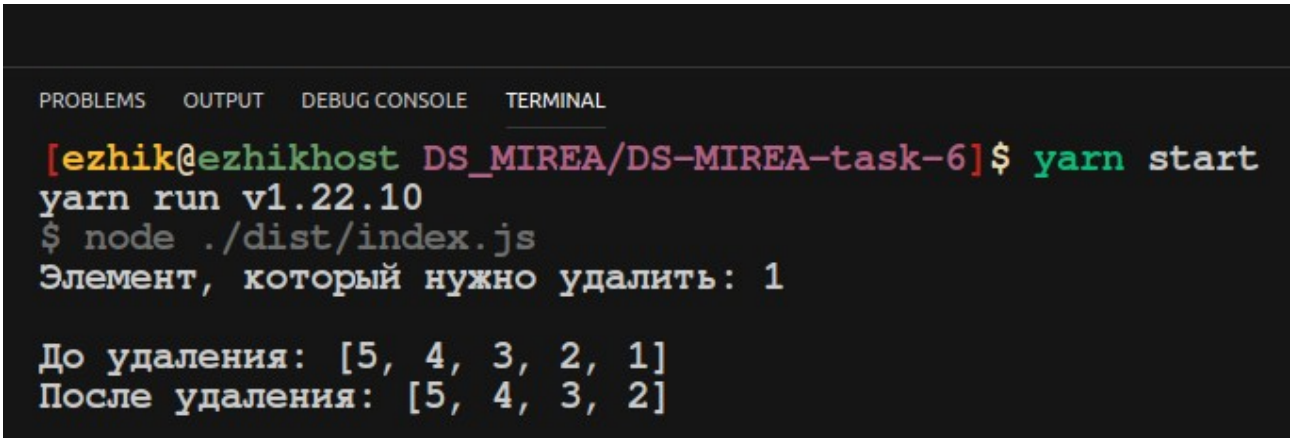

Рис. 5 Тест функции удаления ч.1

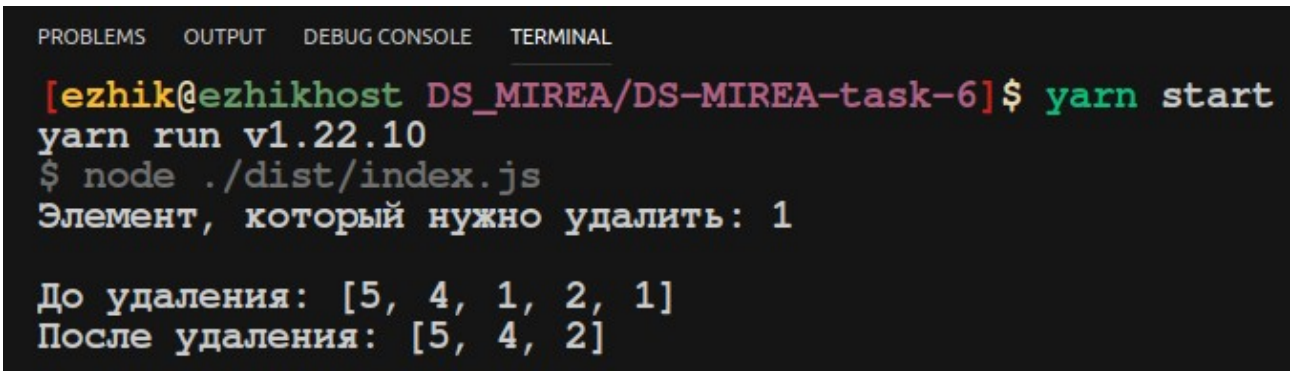

Рис.6 Тест функции удаления ч.2

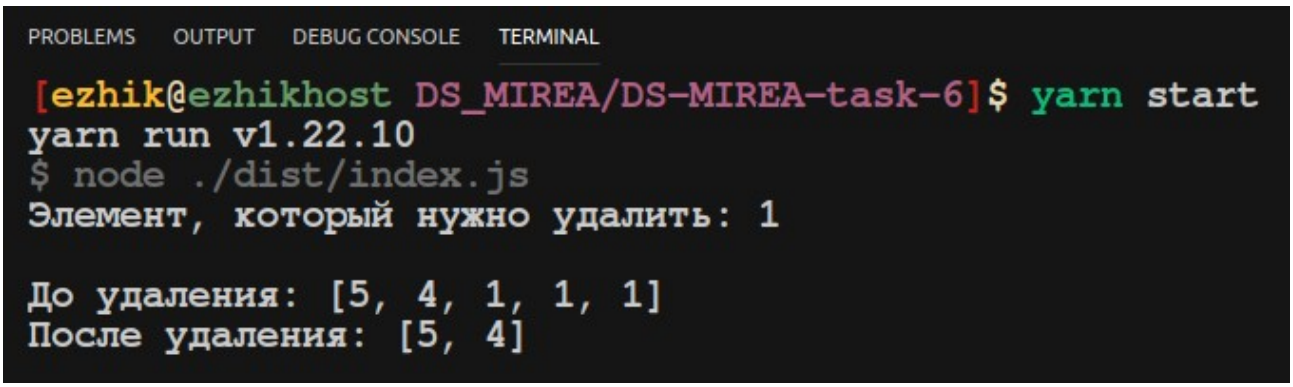

Рис.7 Тест функции удаления ч.3

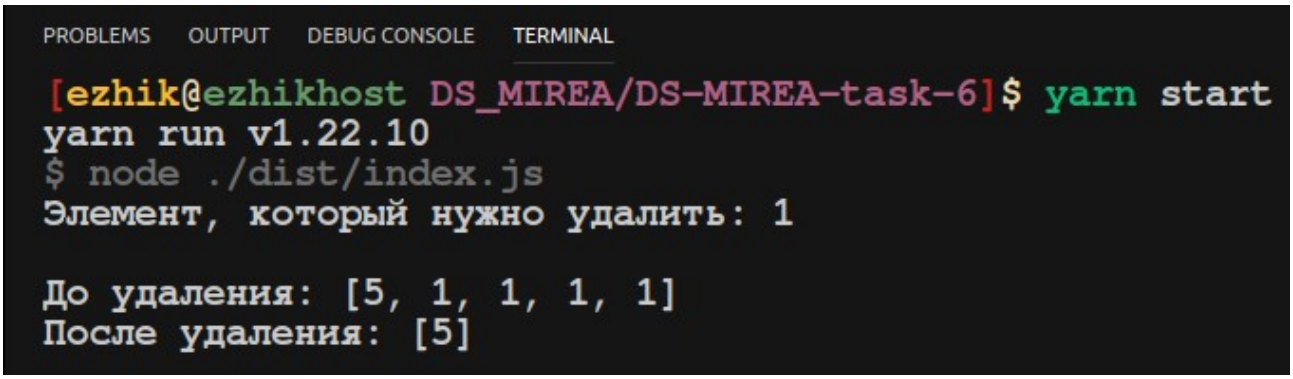

Рис.8 Тест функции удаления ч.4

ВЫВОДЫ

В ходе работы получил знания и практические навыки по разработке и реализации рекурсивных процессов.