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

РТУ МИРЭА

Отчёт по выполнению практического задания 3
Тема: Применение стека и очереди при преобразовании арифметических выражений в постфиксную, префиксную нотации и вычисление значений выражений
Дисциплина Структуры и алгоритмы обработки данных

Выполнил студент Пак С.А.

группа ИКБО-05-20

Москва 2021

СОДЕРЖАНИЕ

ОТЧЁТ ПО ЗАДАНИЮ 1

Номер варианта: 4.

1. Упражнение 1

Задача состоит в том, чтобы провести преобразование инфиксной формы записи выражения $S = a+b-c*k/(d*e-f)$ в постфиксную нотацию.

Для решения задачи нужно использовать стек *w*, в котором будут храниться операторы и скобки, а также пусть *pst* — строка результата.

1) *pst = a*; *w = []*;
2) *pst = a*; *w = [+]*;
3) *pst = ab*; *w = [+]*;
4) *pst = ab+*; *w = [-]*;
5) *pst = ab+c*; *w = [-]*;
6) *pst = ab+c*; *w = [-, *]*;
7) *pst = ab+ck*; *w = [-, *]*;
8) *pst = ab+ck**; *w = [-, /]*;
9) *pst = ab+ck**; *w = [-, /, (]*;
10) *pst =ab+ck*d* ; *w = [-, /, (]*;
11) *pst = ab+ck*d*; *w = [-, /, (, *]*;
12) *pst = ab+ck*de*; *w = [-, /, (, *]*;
13) *pst = ab+ck*de**; *w = [-, /, (, -]*;
14) *pst = ab+ck*de*f*; *w = [-, /, (, -]*;
15) *pst = ab+ck*de*f-*; *w = [-, /]*;
15) *pst = ab+ck*de*f-/-*; *w = []*.

Таким образом, постфиксная нотация выражения *S* будет равна *ab+ck*de*f-/-*.

2. Упражнение 2

Задача состоит в том, чтобы представить постфиксную нотацию следующих выражений: $S_1 = a+b*c-d/e*h$; $S_2 = a+b*c*d+(e-f)*(g*h+i)$.

Начнём с выражения S_1:

1) *pst = a*; *w = [];*
2) *pst = a*; *w = [+]*;
3) *pst = ab*; *w = [+]*;
4) *pst = ab*; *w = [+, *]*;
5) *pst = abc*; *w = [+, *]*;
6) *pst = abc**; *w = [+]*;
7) *pst = abc*+*; *w = [-]*;
8) *pst = abc*+d*; *w = [-]*;
9) *pst = abc*+d*; *w = [-, /]*;

10) *pst = abc*+de*; *w = [-, /]*;
11) *pst = abc*+de/*; *w = [-, *]*;
12) *pst = abc*+de/h*; *w = [-*]*;
13) *pst = abc*+de/h*-; *w = []*.

Сделаем всё то же самое для S_2:
1) *pst = a*; *w = []*;
2) *pst = a*; *w = [+]*;
3) *pst = ab*; *w = [+]*;
4) *pst = ab*; *w = [+, *]*;
5) *pst = abc*; *w = [+, *]*;
6) *pst = abc**; *w = [+, *]*;
7) *pst = abc*d*; *w = [+, *]*;
8) *pst = abc*d*+; *w = [+]*;
9) *pst = abc*d*+; *w = [+, (]*;
10) *pst = abc*d*+e*; *w = [+, (]*;
11) *pst = abc*d*+e*; *w = [+, (, -]*;
12) *pst = abc*d*+ef*; *w = [+, (, -]*;
13) *pst = abc*d*+ef-*; *w = [+]*;
14) *pst = abc*d*+ef-*; *w = [+, *]*;
15) *pst = abc*d*+ef-*; *w = [+, *, (]*;
16) *pst = abc*d*+ef-g*; *w = [+, *, (]*;
17) *pst = abc*d*+ef-g*; *w = [+, *, (, *]*;
18) *pst = abc*d*+ef-gh*; *w = [+, *, (, *]*;
19) *pst = abc*d*+ef-gh**; *w = [+, *, (, +]*;
20) *pst = abc*d*+ef-gh*i*; *w = [+, *, (, +]*;
21) *pst = abc*d*+ef-gh*i+*; *w = [+, *]*;
22) *pst = abc*d*+ef-gh*i+*+*; *w = []*.

Следовательно, для выражения S_1 постфиксная нотация выглядит следующим образом: *abc*+de/h*-. В свою очередь для S_2 постфиксная нотация является выражением *abc*d*+ef-gh*i+*+*.

3. Упражнение 3

Задача состоит в том, чтобы показать процесс преобразования из инфиксной формы в префиксную для выражений: S_1 = *a+b*c-d/e*h*; S_2 = *a+b*c*d+(e-f)*(g*h+i)*.

Для решения задачи нужно использовать два стека: один w_1 из них нужен для операндов, другой w_2 — для операторов.

Начнём с выражения S_1:
1) w_1 = *[a]*; w_2 = *[+]*;

2) $w_1 = [a, b]$; $w_2 = [+]$;
3) $w_1 = [a, b]$; $w_2 = [+, *]$;
4) $w_1 = [a, b, c]$; $w_2 = [+, *]$;
5) $w_1 = [a, *bc]$; $w_2 = [+]$;
6) $w_1 = [+a*bc]$; $w_2 = [-]$;
7) $w_1 = [+a*bc, d]$; $w_2 = [-, /]$;
8) $w_1 = [+a*bc, d, e]$; $w_2 = [-, /]$;
9) $w_1 = [+a*bc, /de]$; $w_2 = [-, *]$;
10) $w_1 = [+a*bc, /de, h]$; $w_2 = [-, *]$;
11) $w_1 = [+a*bc, */deh]$; $w_2 = [-]$;
12) $w_1 = [-+a*bc*/deh]$; $w_2 = []$.

Выражение S_2:
1) $w_1 = [a]$; $w_2 = [+]$;
2) $w_1 = [a, b]$; $w_2 = [+, *]$;
3) $w_1 = [a, b, c]$; $w_2 = [+, *]$;
4) $w_1 = [a, *bc]$; $w_2 = [+, *]$;
5) $w_1 = [a, *bc, d]$; $w_2 = [+, *]$;
6) $w_1 = [+a**bcd]$; $w_2 = [+]$;
7) $w_1 = [+a**bcd]$; $w_2 = [+, (]$;
8) $w_1 = [+a**bcd, e]$; $w_2 = [+, (]$;
9) $w_1 = [+a**bcd, e, f]$; $w_2 = [+, (, -]$;
10) $w_1 = [+a**bcd, -ef]$; $w_2 = [+]$;
11) $w_1 = [+a**bcd, -ef]$; $w_2 = [+, *]$;
12) $w_1 = [+a**bcd, -ef]$; $w_2 = [+, *, (]$;
13) $w_1 = [+a**bcd, -ef, g, h]$; $w_2 = [+, *, (, *]$;
14) $w_1 = [+a**bcd, -ef, *gh]$; $w_2 = [+, *, (, +]$;
15) $w_1 = [+a**bcd, -ef, *gh, i]$; $w_2 = [+, *, (, +]$;
16) $w_1 = [+a**bcd, -ef, +*ghi]$; $w_2 = [+, *]$;
17) $w_1 = [+a**bcd, *-ef+*ghi]$; $w_2 = [+]$;
18) $w_1 = [++a**bcd*-ef+*ghi]$; $w_2 = []$.

Таким образом, для выражения S_1 префиксной формой записи является *-+a*bc*/deh*. Соответственно, для выражения S_2 — *++a**bcd*-ef+*ghi.*

4. Упражнение 4

Задача состоит в том, чтобы вычислить значение выражения в префиксной форме.

Выражение: -*+2 3 5 * 2 3.

Начнём процесс вычисления значения выражения:

1) Оператор «-» переносим в конец строки, т.к. после него также стоит оператор → *+2 3 5 * 2 3-;

2) Оператор «*» также переносим в конец строки → +2 3 5 * 2 3-*;

3) Вычисляем значение +2 3 = 2 + 3 = 5 → 5 5 * 2 3-*;

4) Операнд «5» переносим в конец строки, т.к. всё должно начинаться с оператора → 5 * 2 3-*5;

5) Операнд «5» переносим в конец строки по той же причине, что указана в п.4 → * 2 3-*5 5;

6) Вычисляем значение *2 3 = 2 * 3 = 6 → 6-*5 5;

7) Операнд «6» переносим в конец строки по той же причине, что указана в п.4 → -*5 5 6;

8) Оператор «-» переносим в конец по той же причине, что указана в п.1 → *5 5 6-;

9) Вычисляем *5 5 = 5 * 5 = 25 → 25 6-;

10) Операнд «25» переносим в конец по той же причине, что указана в п.4 → 6-25;

11) Операнд «6» переносим в конец по той же причине, что указана в п.4 → -25 6;

12) Вычисляем -25 6 = 25-6 = 19.

Следовательно, -*+2 3 5 * 2 3 = 19.

ОТЧЁТ ПО ЗАДАНИЮ 2.1

1. Условие и постановка задачи

Необходимо выполнить программную реализацию следующих задач:

1. Реализовать операции над очередью: втолкнуть элемент в очередь, вытолкнуть элемент из очереди, вернуть значение элемента в вершине очереди, сделать очередь пустой, определить пуста ли очередь. Рассмотреть два варианта реализации очереди: на массиве (или строке); на однонаправленном списке.

- Создать класс или просто заголовочный файл с функциями;
- Применить операции для вычисления значения выражения п.4 данного варианта.

2. Разработать функцию(ии) преобразования без скобочного выражения, представленного в инфиксной форме, в префиксную форму;

3. Сложить два длинных числа, которые не могут быть размещены в переменной стандартного типа. Числа поступают либо как последовательность цифр, либо как строка.

2. Выбор подхода к реализации структуры данных.

Так как в условии говорилось рассмотреть два варианта реализации очереди, то мною было реализовано два варианта структуры данных очередь. Однако все они имеют абсолютно одинаковые методы, которые делают одни и те же вещи.

Также для программной реализации мною был выбран язык программирования TypeScript, поэтому прототипы функций и код программы будут представлены на данном языке.

Структура данных очередь имеет следующие методы:

1. add(element: T): void — добавляет элемент в конец очереди

1) Предусловие: метод может быть вызван, когда в очереди нет ни одного элемента;

2) Постусловие: каждый раз, когда вызывается этот метод, количество элементов в очереди возрастает на единицу.

2. peek(): T — возвращает элемент, находящийся в начале очереди

1) Предусловие: когда метод вызывается при пустой очереди, то он ничего не возвращает;

2) Постусловие: метод не изменяет количество элементов в очереди.

3. poll(): void — удаляет элемент из начала очереди

1) Предусловие: если очередь пуста, то метод генерирует ошибку;

2) Постусловие: после вызова метода количество элементов в очереди сократится на единицу.

4. clear(): void — делает очередь пустой

1) Предусловие: метод может быть вызван при любом количестве элементов в очереди;

2) Постусловие: после вызова метода количество элементов в очереди будет равно 0.

5. isEmpty(): boolean — определяет, является ли очередь пустой

1) Предусловие: метод может быть вызван при любом состоянии очереди;

2) Постусловие: метод никак не влияет на состояние объекта класса.

Структура информационной части элемента линейного списка здесь не играет роли, так как в коде использовались так называемые обобщения, которые позволяют с помощью задать тип данных информационной части при создании объекта.

3. Код реализации структуры данных

1. Файл QueueArray.ts

Содержит реализацию очереди на массиве (.ts — это расширение для TypeScript файлов).

```
/**
 * Представляет структуру данных очередь, которая реализована с помощью массива
 */
class QueueArray<T> {
  private _size: number;          // количество элементов в очереди
  private array: Array<T>;       // вспомогательный массив

  /**
   * Конструктор по умолчанию, который создаёт пустую очередь
   */
  public constructor() {
    this._size = 0;
    this.array = [];
  }


  /**
   * Геттер, который возвращает количество элементов в очереди
   */
  public get size(): number {
    return this._size;
  }


  /**
   * Добавляет элементы в конец очереди
   * @param elements       добавляемый элемент
   */
  public add(...elements: Array<T>): void {
```

```
        this._size += elements.length;

        for (const element of elements) {
          this.array.push(element);
        }
      }


      /**
       * Возвращает элемент, который находится в начале очереди
       */
      public peek(): T | undefined {
        return (this._size === 0) ? undefined : this.array[0];
      }


      /**
       * Удаляет элемент, который находится в начале очереди
       */
      public poll(): void {
        if (this._size === 0) {
             throw new Error("[ERROR]: Нельзя удалить элемент, т.к. очередь
пуста!");
        }

        --this._size;
        this.array.shift();
      }


      /**
       * Опустошает очередь (удаляет все элементы из очереди)
       */
      public clear(): void {
        this._size = 0;
        this.array = [];
      }


      /**
       * Определяет, является ли очередь пустой
       */
      public isEmpty(): boolean {
        return (this._size === 0);
      }


      /**
       * Возвращает строковое представление очереди
       */
      public toString(): string {
        return `[${this.array.join()}]`;
      }
    }


    /* Экспорт класса для его использования в других файлах */
```

```
export default QueueArray;
```

2. Файл QueueList.ts

Содержит реализацию очереди на однонаправленном списке. При этом использует ещё 2 структуры данных (List, Node), которые также реализованы в отдельных файлах.

```
import Node from "../node/Node";
import List from "../list/List";

/**
   *   Класс,   представляющий   очередь,   реализованную   с   помощью
однонаправленного списка
 */
class QueueList<T> {
  private _size: number;        // количество элементов в очереди
  private _list: List<T>;       // вспомогательный список


  /**
   * Конструктор, создающий пустую очередь
   */
  public constructor() {
    this._size = 0;
    this._list = new List();
  }


  /**
   * Геттер для поля _size (количество элементов в очереди)
   */
  public get size(): number {
    return this._size;
  }


  /**
   * Добавляет элемент в очередь (список)
   * @param elements        добавляемый элемент (элементы)
   */
  public add(...elements: Array<T>): void {
    this._size += elements.length;
    this._list.add(...elements);
  }


  /**
   * Возвращает элемент, находящийся в начале очереди
   */
  public peek(): Node<T> | undefined | null {
    return this._list.get(0);
  }


  /**
   * Удаляет элемент, который находится в начале очереди
```

```
     */
    public poll(): void {
      --this._size;
      this._list.remove(0);
    }


    /**
     * Удаляет все элементы из очереди
     */
    public clear(): void {
      const N: number = this._size;

      for (let i: number = 0; i < N; ++i) {
        this.poll();
      }
    }


    /**
     * Возвращает true, если очередь пуста
     */
    public isEmpty(): boolean {
      return (this._size === 0);
    }


    /**
     * Возвращает строковое представление очереди
     */
    public toString(): string {
      return this._list.toString();
    }
  }


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

3. Файл List.ts

Содержит структуру данных List, которая представляет однонаправленный список.

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

  /**
   * Класс, представляющий структуру данных однонаправленный список
   */
  class List<T> {
    private _size: number;          // количество элементов в списке
    private _header: Node<T> | undefined | null;       // ссылка на головной
элемент списка

    /**
```

```
     * Вспомогательный метод. Вычисляет "настоящий" индекс в списке для
указанного индекса
     * @param index       индекс
     */
    private calculateRealIndex(index: number): number {
      if (index < 0 || index >= this._size) {
        return -1;
      }

      return this._size - index - 1;
    }

    /**
     * Конструктор, создающий пустой список
     */
    public constructor() {
      this._size = 0;
      this._header = null;
    }


    /**
     * Геттер для поля _size (количество элементов в списке)
     */
    public get size(): number {
      return this._size;
    }


    /**
     * Сеттер для поля _size (количество элементов в списке)
     * @param size        новое количество элементов в списке
     */
    public set size(size: number) {
      this._size = size;
    }


    /**
     * Геттер для поля _header (ссылка на головной элемент списка)
     */
    public get header(): Node<T> | undefined | null {
      return this._header;
    }


    /**
     * Сеттер для поля _header (ссылка на головной элемент списка)
     * @param header        новый головной элемент списка
     */
    public set header(header: Node<T> | undefined | null) {
      this._header = header;
    }


    /**
     * Добавляет элементы или элемент в начало списка
```

```
     * @param elements        добавляемые элементы
     */
    public add(...elements: Array<T | undefined>): void {
      this._size += elements.length;

      for (const element of elements) {
        let tempNode: Node<T> = new Node(element);

        if (this._header === null) {  // список пустой
          this._header = tempNode;
          this._header.next = null;
          continue;
        }

        // список не является пустым, поэтому нужно применить трюк Вирта
                [this._header!.value,  tempNode.value]  =  [tempNode.value,
this._header!.value];

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


    /**
     * Возвращает элемент списка на заданной позиции
     * @param index       индекс элемента
     */
    public get(index: number): Node<T> | undefined | null {
      let realIndex: number = this.calculateRealIndex(index);

      if (realIndex < 0 || this._size === 0) {
        return undefined;
      }

      let copyHeader: Node<T> | undefined | null = cloneDeep(this._header);
      while (copyHeader !== null && (realIndex--) > 0) {
        copyHeader = copyHeader!.next;
      }

      return copyHeader;
    }


    /**
     * Удаляет элемент под заданным индексом из списка
     * @param index       индекс удаляемого элемента
     */
    public remove(index: number): void {
      let realIndex: number = this.calculateRealIndex(index);

      if (this._size === 0 || realIndex < 0) {
        return;
      }

      if (realIndex === 0) {  // нужно удалить головной элемент
        --this._size;
```

```
            this._header = this._header!.next;
            return;
          }

          let listValues: Array<T | undefined> = [];
          let copyHeader: Node<T> | undefined | null = cloneDeep(this._header);

          while (copyHeader !== null && (realIndex--) - 1 > 0) {
            listValues.push(copyHeader!.value);
            copyHeader = copyHeader!.next;
          }

          // в copyHeader находится ссылка на элемент до удаляемого
          listValues.push(copyHeader!.value);     // добавляем в массив значение
элемента до удаляемого
          copyHeader = copyHeader!.next!.next;

          while (copyHeader !== null) {
            listValues.push(copyHeader!.value);
            copyHeader = copyHeader!.next;
          }

          const newLength: number = listValues.length;

          let newList: List<T> = new List();

          listValues.forEach((_, index: number) => {
            newList.add(listValues[newLength - index - 1]);
          });

          this._size = newLength;
          this._header = newList.header;
        }


        /**
         * Возвращает строковое представление однонаправленного списка
         */
        public toString(): string {
          if (this._size === 0) {   // список пустой
            return "[]";
          }

          // клонирование головного объекта списка
          let list: Array<T | undefined> = [];

          let copyHeader: Node<T> | undefined | null = cloneDeep(this._header);
          list.push(copyHeader!.value);

          copyHeader = copyHeader!.next;

          while (copyHeader !== null) {
            list.push(copyHeader!.value);
            copyHeader = copyHeader!.next;
          }

          return `[${list.reverse().join(", ")}]`;
```

```
  }
}


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

4. Файл Node.ts

Содержит структуру данных Node, которая представляет элемент однонаправленного списка.

```
/**
 * Класс, представляющий элемент однонаправленного списка
 */
class Node<T> {
  private _value: T | undefined;           // информационная часть узла
  private _next: Node<T> | undefined | null;       // ссылка на следующий элемент списка

  /**
   * Конструктор, который создаёт узел
   * @param value         значение информационной части узла
   * @param next          ссылка на следующий элемент списка
   */
  public constructor(value?: T | undefined, next?: Node<T> | undefined | null) {
    this._value = value;
    this._next = next;
  }


  /**
   * Геттер для поля _value (значение информационной части)
   */
  public get value(): T | undefined {
    return this._value;
  }


  /**
   * Сеттер для поля _value (значение информационной части)
   * @param value         новое значение информационной части
   */
  public set value(value: T | undefined) {
    this._value = value;
  }


  /**
   * Геттер для поля _next (ссылка на следующий элемент списка)
   */
  public get next(): Node<T> | undefined | null {
    return this._next;
  }
```

```
    /**
     * Сеттер для поля _next (ссылка на следующий элемент списка)
     * @param next        новая ссылка на следующий элемент списка
     */
    public set next(next: Node<T> | undefined | null) {
      this._next = next;
    }


    /**
     * Возвращает строковое представление узла списка
     */
    public toString(): string {
      return `${this._value}`;
    }
  }


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

4. Код функции вычисления выражения

Здесь используются 2 функции. Одна из которых (exprToArray) преобразовывает строку арифметического выражения в массив для более удобной работы с выражением.

1. Файл tasks.ts

Содержит все функции, которые решают поставленные в варианте задачи.

```
/**
 * Преобразовывает строку выражения в массив, удаляя при этом пробелы
 * @param expr        строка арифметического выраженияв префиксной форме
 */
const exprToArray: (expr: string) => Array<string> = (expr: string) => {
  let exprArray: Array<string> = [];

  expr.split(" ").forEach((item: string) => {
    if (item === " ") {
      return;
    }

    if (
      (item.length !== 1) &&
          (item.includes("+")  ||  item.includes("-")  ||  item.includes("/")  ||
item.includes("*"))
    ) {
      item.split("").forEach((item: string) => {
        exprArray.push(item);
      });

      return;
    }
```

```
    exprArray.push(item);
  });

  return exprArray;
}


/**
 * Вычисляет значение выражения, представленного в виде префиксной нотации
 * @param expr        выражение в префиксной нотации
 */
export const calcInPrefixForm: (expr: string) => number = (expr: string) => {
  let exp: Array<string> = exprToArray(expr);

  while (exp.length > 1) {
    const char: string = exp[0];
    const nextChar: string = exp[1];

    const left: boolean = (char === "+" || char === "-" || char === "/" || char
=== "*");
    const right: boolean = (nextChar === "+" || nextChar === "-" || nextChar ===
"/" || nextChar === "*");


    if (left && right || !left) { // оба символа - операции или текущий символ -
операнд
      exp.push(char);
      exp = exp.slice(1);
      continue;
    }

    if (left && !right) { // следующий символ - операнд
      const num1: number = +nextChar;
      const num2: number = +exp[2];

      let res: number;

      switch (char) {
        case "+":
          res = num1 + num2;
          break;

        case "-":
          res = num1 - num2;
          break;

        case "/":
          res = num1 / num2;
          break;

        case "*":
          res = num1 * num2;
          break;

        default:
          throw new Error("[ERROR]: Неопознанная операция!");
      }

      exp.push(res.toString());
```

```
      exp = exp.slice(3);
      continue;
    }
  }

  return +exp[0];
}
```

5. Код основной программы

Основная программа содержится в файле index.ts. Так как в TypeScript нет средств для ввода данных с клавиатуры, то в программе используется файловый ввод.

Поэтому в перменной STDIN содержится всё содержимое файла с входными данными, а затем, используя функции TypeScript, эти входные данные извлекаются и присваиваются различным переменным.

1. Файл index.ts

Содержит код основной программы, которая использует структуры данных и функции, содержащиеся в программе.

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

const STDIN: string = fs.readFileSync(consts.PATH_TO_INPUT + consts.INPUT_FILE,
"utf-8");

/**
 * Основная функция
 */
const main: () => Promise<void> = async () => {
  const prefixExpr: string = STDIN;
  console.log(`${prefixExpr} = ${tasks.calcInPrefixForm(prefixExpr)}`);
}


/* Запуск функции с отловом ошибок */
main()
  .catch((err: Error) => {
    console.log(err.message);
  });
```

6. Результаты тестирования

На рис.1 представлены входные данные в отдельном файле под названием file.txt. Как уже было упомянуто, TypeScript не имеет встроенных средств для ввода данных с клавиатуры, поэтому приходится использовать файловый ввод.

Рис.1 Входные данные для функции вычисления выражения

На рис.2 представлены результаты тестирования функции вычисления выражения в префиксной форме.

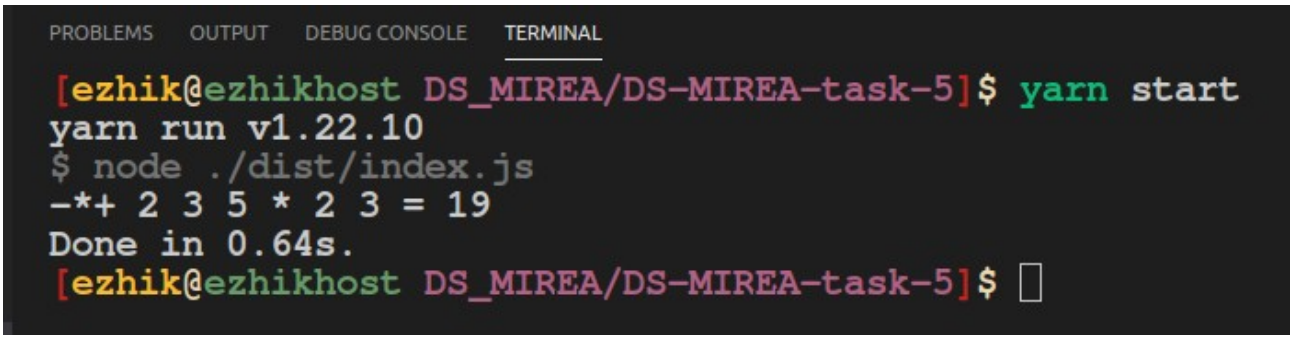

Рис.2 Результаты тестирования функции

ОТЧЁТ ПО ЗАДАНИЮ 2.2

1. Условие и постановка задачи

Разработать функцию(ии) преобразования без скобочного выражения, представленного в инфиксной форме, в префиксную форму.

2. Описание подхода решения задачи

Для решения данной задачи понадобится использовать два стека. Один стек будет для арифметических операций, а другой стек будет для операндов.

Алгоритм преобразования выражения из инфиксной формы в префиксную форму, взятый из раздела «Материалы практических занятий» на сайте СДО МИРЭА:

1. Используются два стека: стек операторов и стек операндов;

2. Операнды по мере выбора из инфиксного выражения заполняют стек операндов;

3. Операция помещается в стек операций по алгоритму формирования постфиксной записи выражения;

4. Формирование префиксного выражения вида: <операция> <операнд2> <операнд1>:

1) Операция извлекается из стека или (в случае, когда в стеке операция ниже по приоритету, чем выбранная из инфиксного выражения) выбирается из выражения;

2) Из стека операндов извлекаются два операнда и последовательно сцепляются со *строкой операции* по правилу: ***операция***, **операнд2**, **операнд1**. Полученное значение записывается в стек операндов;

3) Процесс продолжается с пункта 4 пока стек операций не станет пустым.

3. Алгоритм на псевдокоде

```
toPrefixForm(string expr)
BEGIN

stackOperations ← stack;
stackOperands ← stack;

priorities ← Map<string, int>;
priorities[„+“] = 1;
priorities[„-“] = 1;
priorities[„*“] = 2;
priorites[„/“] = 2;

prefixExpression ← ““;

foreach (char in expr) do
   if  (char >= „a“ and char <= „z“) then
      stackOperands.add(char);
```

```
      continue;
   endif

   if (
      (stackOperations.length == 0) or
      (priorities[char] <= priorities[stackOperators.peek()]) then
      stackOperations.add(char);
      continue;
   endif

   while (priorities[char] <= priorities[stackOperations.peek()]) do
      op ← stackOperations.peek();
      stackOperations.poll();

      first ← stackOperands.peek();
      stackOperands.poll();

      second ← stackOperands.peek();
      stackOperands.poll();

      stackOperands.add(op + second + first);
   od

   stackOperations.add(char);
od

while (stackOperations.length > 0) do
  op ← stackOperations.peek();
  stackOperations.poll();

  first ← stackOperands.peek();
  stackOperands.poll();

  second ← stackOperands.peek();
  stackOperands.poll();

  stackOperands.add(op + second + first);
od

prefixExpression = stackOperands.peek();
return prefixExpression;

END
```

1. stackOperations — стек для операторов;
2. stackOperands — стек для операндов;
3. priorities — хэш-таблица для распределения приоритетов операторов;
4. prefixExpression — строка, в которой после завершения работы функции будет храниться выражения в префиксной форме;
5. char — символ строки expr;
6. expr — строка, содержащая выражение в инфиксной форме;
7. op — операция, доставаемая из вершины стека;
8. first, second — операнды, доставаемые из вершины стеков.

4. Код реализации функции и основной программы

1. Файл tasks.ts

Содержит функцию toPrefixForm, которая и решает поставленную задачу.

```
/**
 * Преобразовывает выражение в инфиксной форме в префиксную
 * @param expr        выражение в инфиксной форме
 */
export const toPrefixForm: (expr: string) => string = (expr: string) => {
  if (expr.includes("(") || expr.includes(")")) {
    throw new Error("[ERROR]: Выражение не должно содержать скобок!");
  }

  // распределяет приоритет арифметических операций
  const PRIORITIES: Map<string, number> = new Map([
    ["+", 1],
    ["-", 1],
    ["*", 2],
    ["/", 2],
  ]);


  let stackOperands: Array<string> = [];        // стек операндов
  let stackOperations: Array<string> = [];      // стек операций

  expr.split("").forEach((char: string) => {
     if (char.toLowerCase() >= "a" && char.toLowerCase() <= "z") { // символ -
это операнд
      stackOperands.unshift(char);
      return;
    }

    if (
      (stackOperations.length === 0) ||
      (PRIORITIES.get(char)! > PRIORITIES.get(stackOperations[0])!)
     ) { // если стек операций пуст или приоритет текущей операции выше, чем
приоритет последней операции в стеке
      stackOperations.unshift(char);
      return;
    }

      // символ - операция, приоритет которой ниже, чем приоритет последней
операции в стеке
    while (PRIORITIES.get(char)! <= PRIORITIES.get(stackOperations[0])!) {
      const operation: string = stackOperations.shift()!;
      const firstOperand: string = stackOperands.shift()!;
      const secondOperand: string = stackOperands.shift()!;
      stackOperands.unshift(`${operation}${secondOperand}${firstOperand}`);
    }

    stackOperations.unshift(char);
  });

  while (stackOperations.length > 0) {
    const operation: string = stackOperations.shift()!;
    const firstOperand: string = stackOperands.shift()!;
    const secondOperand: string = stackOperands.shift()!;
```

```
    stackOperands.unshift(`${operation}${secondOperand}${firstOperand}`);
  }

  return stackOperands[0];
}
```

2. Файл index.ts

Содержит основную программу, которая использует функцию, упомянутую выше.

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

const STDIN: string = fs.readFileSync(consts.PATH_TO_INPUT + consts.INPUT_FILE,
"utf-8");

/**
 * Основная функция
 */
const main: () => Promise<void> = async () => {
  const infixExpression: string = STDIN;
  console.log(`${infixExpression} = ${tasks.toPrefixForm(infixExpression)}`);
}


/* Запуск функции с отловом ошибок */
main()
  .catch((err: Error) => {
    console.log(err.message);
  });
```

5. Результаты тестирования функции

На рис.3 изображены входные данные для функции, которые представляют собой арифметическое выражения.

Рис.3 Входные данные для функции преобразования выражения ч.1

На рис.4 можно увидеть результат работы функции при входных данных, изображённых на рис.3.

Рис.4 Результат работы функции преобразования выражения ч.1

На рис.5 представлены входные данные для функции, которые представляют собой немного более сложное выражение, чем на рис.3.

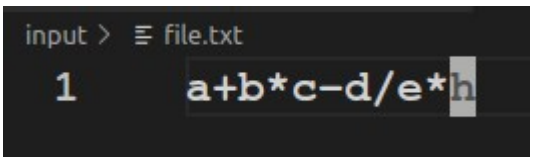

Рис.5 Входные данные для функции преобразования выражения ч.2

На рис.6 представлены результаты работы функции для входных данных, показанных на рис.5.

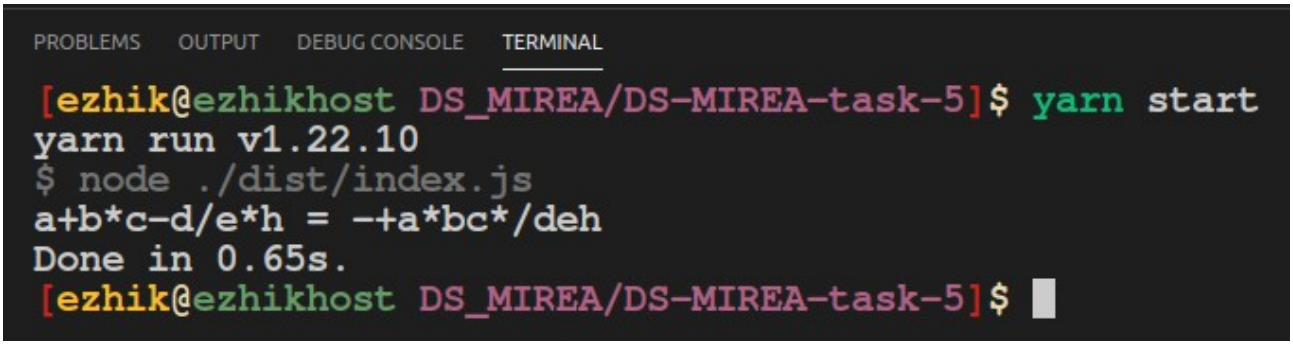

Рис.6 Результат работы функции преобразования выражения ч.2

ОТЧЁТ ПО ЗАДАНИЮ 2.3

1. Условие и постановка задачи

Сложить 2 длинных числа, которые не могут быть размещены в переменной стандартного типа. Числа поступают либо как последовательность цифр, либо как строка.

2. Описание подхода решения задачи

Для решения данной задачи, на мой взгляд, удобнее всего считать, что числа поступают как строка. Также для каждого числа нужно будет создать стек, состоящий из цифр числа, потому что когда люди складывают в столбик числа, они начинают этот процесс делать с конца (с самых последних цифр). Поэтому стек для этой задачи подходит лучше всего.

Ещё нужно учитывать, что при сложении чисел в столбик, сверху находится наибольшее по длине число. Если числа имеют одинаковую длину, то не имеет значения, что к чему прибавлять.

Результаты поочерёдного сложения цифр чисел нужно заносить либо стек, либо в массив, но потом элементы нужно переставить местами в обратном порядке.

Ну и в конец необходимо из массива или стека слепить строку, которая и будет результатом сложения двух чисел.

3. Алгоритм на псевдокоде

```
addTwoLargeNumbers(string leftNumber, string rightNumber)
BEGIN

if (leftNumber.length < rightNumber.length) then
      swap(leftNumber, rightNumber);
endif

leftStack ← stack;
rightStack ← stack;

foreach (char in leftNumber) do
   leftStack.add(to_integer(char));
od

foreach (char in rightNumber) do
   rightStack.add(to_integer(char));
od

carry ← 0;
resStack ← stack;

for i ← 0 to leftNumber.length do
   digit ← to_integer(leftNumber[i]);

    if (i >= rightStack.length) then
      oDigit ← 0;
```

```
   else
      oDigit ← rightNumber[i];
   endif

    tempRes ← digit + carry + oDigit;

   if (tempRes >= 10) then
      carry = 1;
      resStack.add(tempRes – 10);
    else
      carry = 0;
      resStack.add(tempRes);
   endif
od

res ← „“;
while (!resStack.isEmpty()) do
   res ← res + resStack.peek();
   resStack.poll();
od

reverse(res);
return res;

END
```

1. leftNumber, rightNumber — строковые представления тех самых длинных чисел;
2. leftStack, rightStack — стек для цифр первого числа и второго числа соответственно;
3. char — строковое представление цифры числа;
4. carry — остаток, который обычно держат в уме при сложении;
5. resStack — стек для числа, которое будет результатом сложения;
6. i — переменная цикла;
7. digit, oDigit — цифры чисел, стоящие на одинаковых позициях;
8. tempRes — результат сложения двух цифр;
9. res — строковое представление числа, которое является результатом сложения чисел;
10. to_integer() - функция, возвращающая целочисленное представление строки, переданной в качестве аргумента.

4. Код реализации функции и основной программы

1. Файл tasks.ts

Содержит функцию, которая решает поставленную задачу.

```
/**
 * Складывает 2 числа, которые не могут быть представлены в виде примитивных типов
 * @param leftNumber        // первое из складываемых чисел
```

```
 * @param rightNumber       // второе из складываемых чисел
 */
export const addTwoLargeNumbers: (_: string, __: string) => string =
(leftNumber: string, rightNumber: string) => {
  if (leftNumber.length < rightNumber.length) {   // к большему по длине числу
должно прибавляться меньшее
    [leftNumber, rightNumber] = [rightNumber, leftNumber];
  }

      let leftStack: Array<number> = leftNumber.split("").map(digit =>
+digit).reverse();
     let rightStack: Array<number> = rightNumber.split("").map(digit =>
+digit).reverse();

  let carry: number = 0;

  // начинаем складывать с конца
  return leftStack.map((digit: number, index: number) => {
    const oDigit: number = (index >= rightStack.length) ? 0 : rightStack[index];
// к числу по сути добавляются дополнительные нули
    const tempRes: number = digit + oDigit + carry;

    if (tempRes >= 10) {
      carry = 1;
      return tempRes - 10;
    }

    carry = 0;
    return tempRes;
  })
    .reverse()
    .join("");
}
```

2. Файл tasks.ts

Содержит код основной программы, которая тестирует функцию сложения двух длинных чисел.

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

const STDIN: string = fs.readFileSync(consts.PATH_TO_INPUT + consts.INPUT_FILE,
"utf-8");

/**
 * Основная функция
 */
const main: () => Promise<void> = async () => {
  const [lNumber, rNumber] = STDIN.split(" ");
   console.log(`${lNumber} + ${rNumber} = ${tasks.addTwoLargeNumbers(lNumber,
rNumber)}`);
}


/* Запуск функции с отловом ошибок */
main()
```

```
  .catch((err: Error) => {
    console.log(err.message);
  });
```

5. Результаты тестирования функции

На рис.7 видны входные данные для функции, которая складывает два длинных числа.

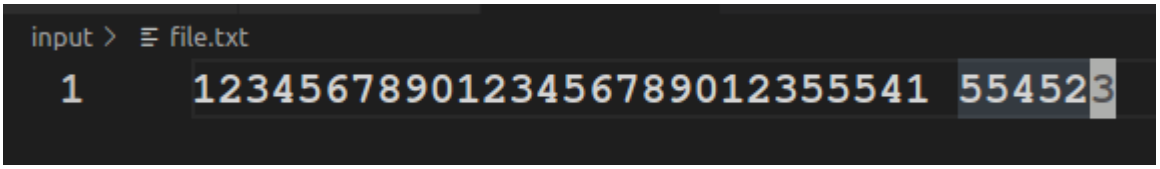

Рис.7 Входные данные для функции сложения двух длинных чисел ч.1

На рис.8 показаны результаты работы функции при входных данных на рис.7.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[ezhik@ezhikhost DS_MIREA/DS-MIREA-task-5]$ yarn start
yarn run v1.22.10
$ node ./dist/index.js
12345678901234567890123555541 + 554523 = 12345678901234567890129100064
Done in 0.65s.
[ezhik@ezhikhost DS_MIREA/DS-MIREA-task-5]$
[ezhik@ezhikhost DS_MIREA/DS-MIREA-task-5]$
```

Рис.8 Результаты тестирования функции сложения двух длинных чисел ч.1

На рис.9 представлены очередные входные данные для функции, которая складывает два длинных числа. Правда на этот раз числа совсем не большие, чтобы доказать, что функция работает правильно даже на небольших числах.

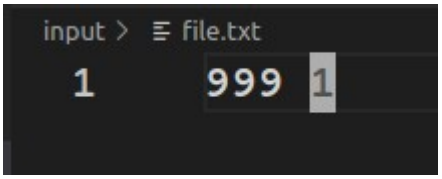

Рис.9 Входные данные для функции сложения двух длинных чисел ч.2

На рис.10 показаны очередные результаты тестирования функции, которая складывает два длинных числа.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
[ezhik@ezhikhost DS_MIREA/DS-MIREA-task-5]$ yarn start
yarn run v1.22.10
$ node ./dist/index.js
a+b*c-d/e*h = -+a*bc*/deh
Done in 0.65s.
[ezhik@ezhikhost DS_MIREA/DS-MIREA-task-5]$
```

Рис.10 Результаты тестирования функции сложения двух длинных чисел ч.2

ВЫВОДЫ

В ходе работы получил знания и навыки по реализации структуры данных стек и очередь и их применения в решении задач. Стеки и очереди являются крайне полезными структурами, позволяющими реализовать множество алгоритмов. Например, поиск в ширину, поиск в глубину, сложение длинных чисел и т.д.