|  |
| --- |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **"МИРЭА – Российский технологический университет"**  **РТУ МИРЭА** |

Институт комплексной безопасности и специального приборостроения

Кафедра КБ-3 «Управление и моделирование систем»

**ОТЧЕТ   
о выполнении лабораторной работы №1**

**«Реализация сортировки линейных структур данных»**

**по дисциплине   
«Программная реализация нелинейных структур»**

**Вариант № 94**

Выполнил: студент 2 курса

Группы **\_\_\_\_\_\_\_\_\_\_\_**

шифр \_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
*(фио студента)*

Проверил:

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

Москва 2022 г.

**Задание на лабораторную работу № 1.**

В рамках лабораторной работы №1 требуется программно реализовать (с помощью указателей (однонаправленных/двунаправленный динамический линейный связанный список, массива или используя стандартный контейнер библиотеки STL “stack” или «queue» - по варианту) абстрактный тип данных (АТД) в соответствии с заданием (стек, дек, очередь с одной головой, очередь с головой и хвостом).

Абстрактный тип данных должен позволять осуществлять только операции, присущие типу линейного связанного списка:

* получить значение первого элемента (на выходе),
* добавить элемент (на вход),
* удалить элемент из списка (на выходе),
* проверить – список пуст,
* обнулить (проинициализировать) список (конструктор, при необходимости).
* деструктор (при необходимости)

Используя разработанный АТД и указанный набор операций, необходимо реализовать заданный алгоритм сортировки последовательности элементов заданного типа, при этом следует учитывать, что разрешен доступ (чтение/извлечение) только к элементу на выходе.

На основе исходного текста программы получить аналитическую оценку трудоемкости работы алгоритма сортировки, используя О-символику для каждого реализованного метода АТД и сортировки в целом.

**Вариант № 94.**

**Реализация связи элементов линейного списка: Массив**

**Способ организации линейного связанный список: Стек**

**Алгоритм сортировки: Фиксированное двухпутевое слияние**

**Теория о сортировках.**

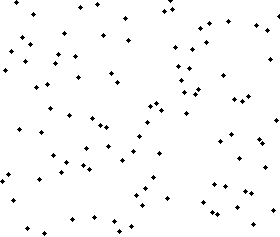
**Сортировка слиянием** (англ. *merge sort*) — алгоритм сортировки, который упорядочивает списки (или другие структуры данных, доступ к элементам которых можно получать только последовательно, например — потоки) в определённом порядке. Эта сортировка — хороший пример использования принципа «разделяй и властвуй». Сначала задача разбивается на несколько подзадач меньшего размера. Затем эти задачи решаются с помощью рекурсивного вызова или непосредственно, если их размер достаточно мал. Наконец, их решения комбинируются, и получается решение исходной задачи.

Рисунок иллюстрирует сортировку слиянием.

**Алгоритм**

1. Сортируемый массив разбивается на две части примерно одинакового размера;
2. Каждая из получившихся частей сортируется отдельно, например — тем же самым алгоритмом;
3. Два упорядоченных массива половинного размера соединяются в один.

1.1. — 2.1. Рекурсивное разбиение задачи на меньшие происходит до тех пор, пока размер массива не достигнет единицы (любой массив длины 1 можно считать упорядоченным).

3.1. Соединение двух упорядоченных массивов в один.  
Основную идею слияния двух отсортированных массивов можно объяснить на следующем примере. Пусть мы имеем два уже отсортированных по возрастанию подмассива. Тогда:  
3.2. Слияние двух подмассивов в третий результирующий массив.  
На каждом шаге мы берём меньший из двух первых элементов подмассивов и записываем его в результирующий массив. Счётчики номеров элементов результирующего массива и подмассива, из которого был взят элемент, увеличиваем на 1.  
3.3. «Прицепление» остатка.  
Когда один из подмассивов закончился, мы добавляем все оставшиеся элементы второго подмассива в результирующий массив.

**Листинг программы с расчетами.**

from typing import Generic, List, TypeVar

VT = TypeVar("VT")

class Stack(Generic[VT]):

    \_stack: List[VT]

    \_n\_op: int

    def \_\_init\_\_(self):

        self.\_stack = []

        self.\_n\_op = 0

    def push(self, el: VT):  # n + 2

        self.\_n\_op += self.size + 2

        self.\_stack = self.\_stack.copy()

        self.\_stack.append(el)

    def pop(self) -> VT:  # n + 2

        self.\_n\_op += self.size + 2

        if self.is\_empty():

            raise Exception("Can't pop from empty stack!")

        el = self.\_stack[-1]

        self.\_stack = self.\_stack[:-1]

        return el

    def is\_empty(self):  # 1

        return len(self.\_stack) == 0

    @property

    def n\_op(self):

        return self.\_n\_op

    @property

    def size(self):  # 1

        return len(self.\_stack)

    @property

    def top(self):  # 1

        return self.\_stack[-1]

    def \_\_str\_\_(self) -> str:

        return "Stack[" + ", ".join(map(str, self.\_stack)) + "]"

def print\_stack(stack: Stack):

    print("Stack[" + ", ".join(map(str, stack.\_stack)) + "]")

def seek(stack: Stack, pos: int):  # ~= 2n^2 + 3n - 8

    swap\_stack = Stack()  # 1

    stack\_size = stack.size  # 2

    for \_ in range(stack\_size - pos - 1):  # (n - pos - 1) \* (

        swap\_stack.push(stack.pop())  # k + 2 + n + 2

    # ) ~= n^2 + 4n - n\*pos - 4pos - n - 4

    el = stack.top  # 2

    for \_ in range(stack\_size - pos - 1):  # (n - pos - 1) \* (

        stack.push(swap\_stack.pop())  # k + 2 + n + 2

    # ) ~= n^2 + 4n - n\*pos - 4pos - n - 4

    return el

def push\_by\_pos(stack: Stack[VT], el: VT, pos: int):  # ~= 2n^2 + 4n + 2

    swap\_stack = Stack()  # 1

    stack\_size = stack.size  # 2

    for \_ in range(stack\_size - pos):  # (n - pos) \* (

        swap\_stack.push(stack.pop())  # k + n + 4

    # ) ~= n^2 + 4n - n \* pos - 4pos

    stack.push(el)  # n + 2

    for \_ in range(stack\_size - pos):  # (n - pos) \* (

        stack.push(swap\_stack.pop())  # k + n + 4

    # ) ~= n^2 + 4n - n \* pos - 4pos

def pop\_by\_pos(stack: Stack[VT], pos: int) -> VT:  # ~= 2n^2 + 3n - 8

    swap\_stack = Stack()  # 1

    stack\_size = stack.size  # 2

    for \_ in range(stack\_size - pos - 1):  # (n - pos - 1) \* (

        swap\_stack.push(stack.pop())  # k + 2 + n + 2

    # ) ~= n^2 + 4n - n\*pos - 4pos - n - 4

    el = stack.pop()  # n + 2

    for \_ in range(stack\_size - pos - 1):  # (n - pos - 1) \* (

        stack.push(swap\_stack.pop())  # k + 2 + n + 2

    # ) ~= n^2 + 4n - n\*pos - 4pos - n - 4

    return el

def swap(stack: Stack, pos1: int, pos2: int):  # 8n^2 + 14n - 12

    temp = pop\_by\_pos(stack, pos1)  # 2n^2 + 3n - 8

    push\_by\_pos(

        stack, pop\_by\_pos(stack, pos2 - 1), pos1

    )  # 2n^2 + 4n + 2 + 2n^2 + 3n - 8

    push\_by\_pos(stack, temp, pos2)  # 2n^2 + 4n + 2

def slice\_stack(

    stack: Stack[VT], l: int = 0, r: int = -1

) -> Stack[VT]:  # 4 + 4 + n^2 + 3n - 5 + n + n^2 + 2n = 2n^2 + 6n + 3

    stack\_size = stack.size  # 2

    slice\_stack = Stack[VT]()  # 1

    swap\_stack = Stack[VT]()  # 1

    if r == -1:  # 1

        r = stack.size - 1  # 3

    for \_ in range(stack\_size - r - 1):  # (n - r - 1) \* (

        swap\_stack.push(stack.pop())  # k + n + 4

    # ) ~= n^2 + 3n - 5

    for \_ in range(r - l + 1):  # (r - l + 1) \* (

        slice\_stack.push(stack.top)  # k + 3

        swap\_stack.push(stack.pop())  # s + n + 4

    # ) ~= n

    for \_ in range(swap\_stack.size):  # n \* (

        stack.push(swap\_stack.pop())  # k + n + 2

    # ) ~= n^2 + 2n

    return reverse(slice\_stack)  # 2n^2 + 2n + 1

def reverse(stack: Stack[VT]) -> Stack[VT]:  # 2n^2 + 2n + 1

    out = Stack[VT]()  # 1

    for \_ in range(stack.size):  # n \* (

        out.push(stack.pop())  # n + n + 2

    # ) = 2n^2 + 2n

    return out

def merge(stack1: Stack[VT], stack2: Stack[VT]) -> Stack[VT]:

    out = Stack[VT]()  # 1

    while stack1.size > 0 and stack2.size > 0:  # n \* (

        if stack1.top > stack2.top:  # 3

            out.push(stack1.pop())  # n + n + 3

        else:  # 1

            out.push(stack2.pop())  # n + n + 3

    # ) = 4n^2 + 6n

    while stack2.size > 0:  # n \* (

        out.push(stack2.pop())  # n + n + 3

    # ) = 2n^2 + 3n

    while stack1.size > 0:  # n \* (

        out.push(stack1.pop())  # n + n + 3

    # ) = 2n^2 + 3n

    return reverse(out)  # 1

def fixed\_two\_way\_merge\_sort(stack: Stack[VT]) -> Stack[VT]: # ~= 6n^2 + 6n + 3

    if stack.size <= 1:  # 1

        return stack  # 1

    stack\_size = stack.size  # 2

    left\_stack = Stack[VT]()  # 1

    right\_stack = Stack[VT]()  # 1

    for \_ in range(stack\_size // 2):  # (n // 2) \* (

        left\_stack.push(stack.pop())  # 2n + 4

    # ) ~= n^2 + 2n

    for \_ in range(stack\_size - stack\_size // 2):  # (n - n // 2) \* (

        right\_stack.push(stack.pop())  # 2n + 4

    # ) ~= n^2 + 2n

    left\_stack = fixed\_two\_way\_merge\_sort(left\_stack)  # 2n^2 + 2n + 1

    right\_stack = fixed\_two\_way\_merge\_sort(right\_stack)  # 2n^2 + 2n + 1

    return merge(left\_stack, right\_stack)  # 2n^2 + 2n + 1

if \_\_name\_\_ == "\_\_main\_\_":

    import time

    from random import randint

    tests = 10

    step = 5000

    for test\_num in range(1, tests + 1):

        stack = Stack[int]()

        for \_ in range(test\_num \* step):

            stack.push(randint(-10000, 10000))

        start\_time = time.time()

        fixed\_two\_way\_merge\_sort(stack)

        total\_time = time.time() - start\_time

        print(f"Test: {test\_num}")

        print(f"Elems count: {test\_num \* step}")

        print(f"Total time: {total\_time}")

        print(f"N\_OP: {stack.n\_op}")

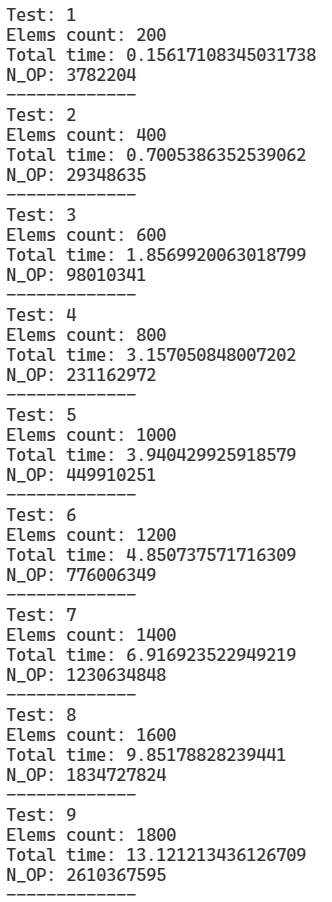
        print("-------------")

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кол-во элементов | F(n) | O(F(n)) | Т(n) (сек) | N\_op |
| 200 | 1843721 | 305754 | 0,1561 | 3782204 |
| 400 | 8318873 | 1383017 | 0,7005 | 29348635 |
| 600 | 19967500 | 3322375 | 1,8569 | 98010341 |
| 800 | 37078727 | 6172068 | 3,157 | 231162972 |
| 1000 | 59854530 | 9965784 | 3,9404 | 449910251 |
| 1200 | 88450672 | 14729499 | 4,8507 | 776006349 |
| 1400 | 122994064 | 20484374 | 6,9169 | 1230634848 |
| 1600 | 163591844 | 27248272 | 9,8517 | 1834727824 |
| 1800 | 210336728 | 35036651 | 13,1212 | 2610367595 |
| 2000 | 263310445 | 43863137 | 17,0707 | 3578072339 |

|  |  |  |  |
| --- | --- | --- | --- |
| С1=F(n)/T(n) | С2=O(F(n))/T(n) | С3=F(n)/N\_op | С4=O(F(n))/N\_op |
| 11811153,39 | 1958707,544 | 0,48747266 | 0,080840232 |
| 11875621,88 | 1974328,323 | 0,283450086 | 0,047123725 |
| 10753136,84 | 1789204,981 | 0,2037285 | 0,033898206 |
| 11744924,68 | 1955042,116 | 0,160400807 | 0,026700072 |
| 15189963,03 | 2529130,11 | 0,1330366 | 0,022150605 |
| 18234620,09 | 3036571,817 | 0,113981892 | 0,018981158 |
| 17781674,48 | 2961496,303 | 0,09994359 | 0,016645371 |
| 16605443,12 | 2765844,661 | 0,089164094 | 0,014851397 |
| 16030296,59 | 2670232,224 | 0,080577436 | 0,013422114 |
| 15424701,1 | 2569498,447 | 0,073590028 | 0,012258874 |

**Таблица результата экспериментов и графики зависимостей**

**Скриншот работы пограммы:**



**Выводы.**

По результатам данной работы был реализован класс для работы со стеком. Стек был реализован через массив. Были реализованы базовые операции для стека и рассчитана их алгоритмическая сложность. Так же был реализован алгоритм сортировки слиянием. После аналитической оценки и экспериментов было установлено, алгоритм сортировки слиянием для стека через массив имеет зависимость от числа элементов.

**Литература:**

1. Левитин А. В. Глава 6. Метод декомпозиции: Сортировка слиянием // Алгоритмы. Введение в разработку и анализ — М.: Вильямс, 2006.