|  |
| --- |
| 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 № 65**

Выполнил: студент 2 курса

Группы **\_\_\_\_\_\_\_\_**

шифр \_\_\_\_\_\_\_

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

Проверил:

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

Москва 2020 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 № 65.**

**Реализация связи элементов линейного списка: Указатели**

**Способ организации линейного связанный список: Очередь с одной головой**

**Алгоритм сортировки: Быстаря по Хоару**

**Теория о сортировках.**

**Быстрая сортировка Хоара.**

Один из самых быстрых известных универсальных алгоритмов сортировки массивов: в среднем O ( n log ⁡ n ) {\displaystyle O(n\log n)} обменов при упорядочении n {\displaystyle n} элементов; из-за наличия ряда недостатков на практике обычно используется с некоторыми доработками.

QuickSort является существенно улучшенным вариантом алгоритма сортировки с помощью прямого обмена (его варианты известны как «Пузырьковая сортировка» и «Шейкерная сортировка»), известного в том числе своей низкой эффективностью. Принципиальное отличие состоит в том, что в первую очередь производятся перестановки на наибольшем возможном расстоянии и после каждого прохода элементы делятся на две независимые группы (таким образом улучшение самого неэффективного прямого метода сортировки дало в результате один из наиболее эффективных улучшенных методов).

Рисунок иллюстрирует сортировку простым выбором.

Шаги алгоритма:

1. Выбрать элемент из массива. Назовём его опорным.

2. *Разбиение*: перераспределение элементов в массиве таким образом, что элементы, меньшие опорного, помещаются перед ним, а большие или равные - после.

3. Рекурсивно применить первые два шага к двум подмассивам слева и справа от опорного элемента. Рекурсия не применяется к массиву, в котором только один элемент или отсутствуют элементы.

**Листинг программы с расчетами.**

typedef struct *S\_Node*

{

*int32\_t* v;

    struct *S\_Node* \*prev;

} *S\_Node*;

*S\_Node* \*S\_Node\_init(*int32\_t*);

void S\_Node\_print(const *S\_Node*\*);

typedef struct *S\_Queue*

{

*S\_Node* \*head;

*size\_t* size;

*uint64\_t* N\_OP;

} *S\_Queue*;

*S\_Queue* \*S\_Queue\_init();

void S\_Queue\_push(*S\_Queue*\*, *S\_Node*\*);

*S\_Node* \*S\_Queue\_pop(*S\_Queue*\*);

const *S\_Node* \*S\_Queue\_peek(*S\_Queue*\*, *size\_t*);

void S\_Queue\_print(*S\_Queue* \*);

*size\_t* S\_Queue\_len(*S\_Queue*\*);

void S\_Queue\_rotate(*S\_Queue*\*);

#include "queue.h"

*S\_Node* \*S\_Node\_init(*int32\_t* *v*)

{

*S\_Node* \*node = (*S\_Node*\*) malloc(sizeof(*S\_Node*));

    node->prev = NULL;

    node->v = *v*;

    return node;

}

void S\_Node\_print(const *S\_Node* \**node*)

{

    printf("%i ", *node*->v);

}

*S\_Queue* \*S\_Queue\_init()

{

*S\_Queue* \*queue = (*S\_Queue*\*) malloc(sizeof(*S\_Queue*));

    queue->head = NULL;

    queue->size = 0;

    queue->N\_OP = 0;

    return queue;

}

void S\_Queue\_push(*S\_Queue* \**queue*, *S\_Node* \**node*) // 2n + 3

{

    if (!*queue*->size) // 1

    {

*queue*->head = *node*; // 2

*queue*->N\_OP += 2;

    }

    else

    {

*S\_Node* \*tail = *queue*->head; // 2

        while (tail->prev) {

            tail = tail->prev; // 2

*queue*->N\_OP += 3;

        } // 2 \* (n - 1)

        tail->prev = *node*; // 2

*node*->prev = NULL; // 2

*queue*->N\_OP += 6;

    } // 2n - 2 + 2 + 2 = 2n + 2

*queue*->size++; // 1

*queue*->N\_OP += 2;

}

*S\_Node* \*S\_Queue\_pop(*S\_Queue* \**queue*) // 9

{

    if (!*queue*->size)

    {

        fprintf(stderr, "Invalaid opertion pop! Queue is empty!\n");

        exit(EXIT\_FAILURE);

    }

*S\_Node* \*node = *queue*->head; // 2

*queue*->head = node->prev; // 3

    node->prev = NULL; // 2

*queue*->size--; // 2

*queue*->N\_OP += 9;

    return node;

}

void S\_Queue\_print(*S\_Queue* \**queue*)

{

    if (!*queue*->size) return;

    else

    {

*S\_Node* \*node = *queue*->head;

        while(node != NULL)

        {

            S\_Node\_print(node);

            node = node->prev;

        }

        printf("\n");

    }

}

*size\_t* S\_Queue\_len(*S\_Queue* \**queue*) // 1

{

    return *queue*->size;

}

void S\_Queue\_rotate(*S\_Queue* \**queue*) // 2n + 14

{

    if(*queue*->size < 2) return; // 2

    S\_Queue\_push(*queue*, S\_Queue\_pop(*queue*)); // 9 + 2n + 3 = 2n + 12

}

const *S\_Node* \*S\_Queue\_peek(*S\_Queue* \**queue*, *size\_t* *pos*) // 2k + 2

{

*S\_Node* \*node = *queue*->head; // 2

*queue*->N\_OP += 2;

    while (*pos*--) // 2k

    {

        node = node->prev; // 2

*queue*->N\_OP += 3;

    }

    return node;

}

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <inttypes.h>

#include "queue.h"

void fill\_test\_data(*S\_Queue* \**queue*, *size\_t* *count*)

{

    srand(time(NULL));

    while (*count*--) S\_Queue\_push(*queue*, S\_Node\_init(rand() % 300));

}

*S\_Node* \*\_pop\_by\_pos(*S\_Queue* \**queue*, *size\_t* *pos*) // 3 + 10 + 1 + 2 + 10 + 2nk + 15k + 2n^2 + 13n - 2kn - 15k - 15 = 2n^2 + 13n + 11

{

    if (*queue*->size - 1 < *pos*)  // 3

    {

        fprintf(stderr, "Out of bound! Queue size: %li, pop position: %li\n", *queue*->size, *pos*);

        exit(EXIT\_FAILURE);

    }

    if(!*pos*) return S\_Queue\_pop(*queue*); // 10

*size\_t* before = *pos*; // 1

*size\_t* after = S\_Queue\_len(*queue*) - *pos*; // 2

*S\_Node* \*node;

    while (before--) S\_Queue\_rotate(*queue*); // k \* (2n + 14 + 1) = 2nk + 15k

    node = S\_Queue\_pop(*queue*); // 10

    while (--after) S\_Queue\_rotate(*queue*); // (n - k - 1) \* (2n + 14 + 1) = 2n^2 + 13n - 2kn - 15k - 15

    return node;

}

void \_push\_by\_pos(*S\_Queue* \**queue*, *S\_Node* \**node*, *size\_t* *pos*) // 1 + 3 + 2n + 3 + 2nk + 15k + 2n^2 + 15n - 2nk - 15k = 2n^2 + 17n + 7

{

    if (*pos* >= *queue*->size) // 2

    {

        S\_Queue\_push(*queue*, *node*); // 2n + 3

        return;

    }

*size\_t* before = *pos*; // 1

*size\_t* after = S\_Queue\_len(*queue*) - *pos*; // 3

    while(before--) S\_Queue\_rotate(*queue*); // k \* (2n + 14 + 1) = 2nk + 15k

    S\_Queue\_push(*queue*, *node*); // 2n + 3

    while(after--) S\_Queue\_rotate(*queue*); // (n - k) \* (2n + 15) = 2n^2 + 15n - 2nk - 15k

}

void \_quick\_sort(*S\_Queue* \**queue*, *int32\_t* *left*, *int32\_t* *right*) // 2n^3 \* log(n)

{

*int32\_t* i = *left*; // 1

*int32\_t* j = *right*; // 1

    const *S\_Node* \*pivot = S\_Queue\_peek(*queue*, (i + j) / 2); // 2 \* ((i + j) / 2) + 1 = i + j + 1

*S\_Node* \*temp;

    do

    {

        while(S\_Queue\_peek(*queue*, i)->v < pivot->v) i++; // Σ(k=i -> (i + j) / 2) (3 + 2k + 2) ~= i + j + 5

        while(S\_Queue\_peek(*queue*, j)->v > pivot->v) j--; // Σ(k=j -> (i + j) / 2) (3 + 2k + 2) ~= i + j + 5

        if (i <= j) // 1

        {

            if (i < j) // 1

            {

                 temp = \_pop\_by\_pos(*queue*, i); // 2i^2 + 13i + 12

                \_push\_by\_pos(*queue*, \_pop\_by\_pos(*queue*, j - 1), i); // 2i^2 + 17i + 7 + 2(j - 1)^2 + 13(j - 1) + 11 = 17i + 31 + 2j^2 + 9j

                \_push\_by\_pos(*queue*, temp, j); // 2j^2 + 17j + 7

            }

            i++; // 1

            j--; // 1

        }

    } while (i < j);

    // ((i + j) / 2) \* (2i + 2j + 10 + 2i^2 + 13i + 12 + 17i + 31+ 2j^2 + 9j + 2j^2 + 17j + 12) =

    // = 16i^2 + 30ij + 32i + i^3 + 2ij^2 + 14j^2 + 32j + i^2j + 2j^3 ~= n^3

    if (*left* < j)

        \_quick\_sort(*queue*, *left*, j); // n3 \* log(n)

    if (i < *right*)

        \_quick\_sort(*queue*, i, *right*); // n3 \* log(n)

}

void quick\_sort(*S\_Queue* \**queue*)

{

    \_quick\_sort(*queue*, 0, S\_Queue\_len(*queue*) - 1);

}

*int64\_t* millis()

{

    struct *timespec* now;

    timespec\_get(&now, TIME\_UTC);

    return ((*int64\_t*) now.tv\_sec) \* 1000 + ((*int64\_t*) now.tv\_nsec) / 1000000;

}

int main(int *argc*, char \*\**argv*)

{

    int step = 300;

    if (*argc* > 2) exit(EXIT\_FAILURE);

    if (*argc* == 1) step = 300;

    else step = atoi(*argv*[1]);

    for (int i = step; i < 10 \* step; i += step)

    {

*S\_Queue* \*queue = S\_Queue\_init();

        fill\_test\_data(queue, i);

*int64\_t* t\_start = millis();

        quick\_sort(queue);

*int64\_t* t\_stop = millis(NULL);

        printf("Elems: %d\nN\_OP: %li\nTime: %"PRId64"ms\n", i, queue->N\_OP, t\_stop - t\_start);

    }

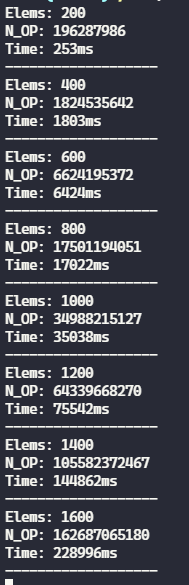
    return 0;

}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кол-во элементов | F(n) | O(F(n)) | Т(n) (сек) | N\_op |
| 200 | 13287717 | 6643856 | 0,258 | 191225235 |
| 400 | 122301704 | 61150850 | 1,761 | 1812176256 |
| 600 | 444356214 | 222178105 | 6,255 | 6701772667 |
| 800 | 1106413597 | 553206796 | 16,542 | 17063604672 |
| 1000 | 2241446076 | 1120723036 | 35,081 | 35296166169 |
| 1200 | 3986849679 | 1993424837 | 74,313 | 63002688577 |
| 1400 | 6483530828 | 3241765411 | 135,686 | 103485479633 |
| 1600 | 9875308743 | 4937654369 | 229,307 | 160603757934 |
| 1800 | 14308492982 | 7154246488 | 341,111 | 232581586799 |
| 2000 | 19931568574 | 9965784285 | 514,632 | 333361708033 |

|  |  |  |  |
| --- | --- | --- | --- |
| С1=F(n)/T(n) | С2=O(F(n))/T(n) | С3=F(n)/N\_op | С4=O(F(n))/N\_op |
| 51502780,54 | 25751380,58 | 0,069487259 | 0,034743616 |
| 69450144,26 | 34725070,71 | 0,067488857 | 0,033744427 |
| 71040162,16 | 35520080,68 | 0,06630428 | 0,03315214 |
| 66885116,51 | 33442558,1 | 0,064840555 | 0,032420277 |
| 63893448,77 | 31946724,31 | 0,063503953 | 0,031751976 |
| 53649424,45 | 26824712,19 | 0,063280628 | 0,031640314 |
| 47783344,1 | 23891672,03 | 0,0626516 | 0,0313258 |
| 43065884,35 | 21532942,17 | 0,061488653 | 0,030744326 |
| 41946735,76 | 20973367,87 | 0,061520317 | 0,030760159 |
| 38729749,75 | 19364874,87 | 0,059789616 | 0,029894808 |

**Таблица результата экспериментов и графики зависимостей**

**Скриншот работы программы:**

**Выводы.**

По результатам экспериментов было установлено, что алгоритм сортировки по Хоару для очереди с одной головой имеет зависимость от числа элементов

**Литература:**

1. Левитин А. В. Глава 4. Метод декомпозиции: Быстрая сортировка // Алгоритмы. Введение в разработку и анализ — М.: Вильямс, 2006.