1. **Цель работы**

   Целью работы является изучение структур данных «стек» и «очередь», а также получение практических навыков их реализации.

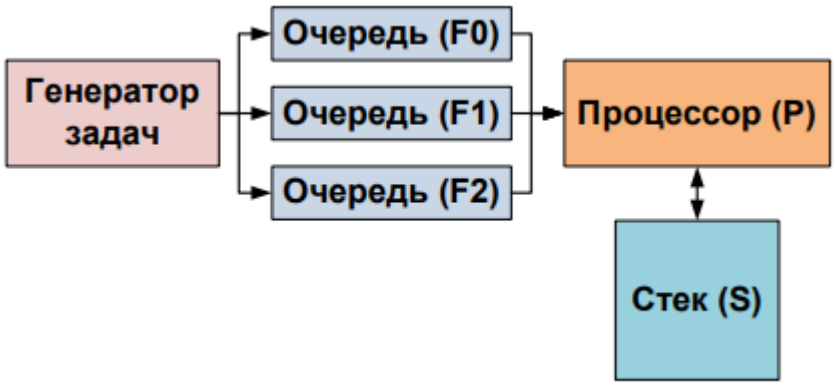
2. **Задание**

Согласно варианту №2:

Вариант	Номер задачи	Реализация
2	1	Стек – статический, очередь - динамическая

Система состоит из процессора P, трѐх очередей F0, F1, F2 и стека S.

В систему поступают запросы на выполнение задач.



Поступающие запросы ставятся в соответствующие приоритетам очереди. Сначала обрабатываются задачи из очереди F0. Если она пуста, можно обрабатывать задачи из очереди F1. Если и она пуста, то можно обрабатывать задачи из очереди F2. Если все очереди пусты, то система находится в ожидании поступающих задач (процессор свободен), либо в режиме обработки предыдущей задачи (процессор занят). Если поступает задача с более высоким приоритетом, чем обрабатываемая в данный момент, то обрабатываемая помещается в стек и может обрабатываться тогда и только тогда, когда все задачи с более высоким приоритетом уже обработаны.

3. **Листинг программы**

```
// Вариант 2

#include <iostream>
#include<string>
#include <time.h>

using namespace std;

struct Task
{
    string name;
    int priority;
```

```
    int taskStart;
    int taskDuration;
};

struct TaskList
{
    Task* taskValues;
    TaskList* next;
};

int get_num_int() // Запрос и проверка числа на корректность
{
    int x;

    cin >> x;
    while (cin.fail() || (cin.peek() != '\n')) // Проверка на корректность
    {
        cin.clear(); // Очищение флага ошибки
        cin.ignore(numeric_limits<streamsize>::max(), '\n'); // Очистка буфера
запроса
        cout << "Повторите ввод: ";
        cin >> x;
    }

    return x;
}

bool IsEmpty(TaskList* head) // Проверка пустой ли список
{
    return head == NULL;
}

void PrintListElem(Task* head)
{
    if (head == NULL)
    {
        cout << "Процессор пуст." << endl;
        return;
    }

    Task* cur = head;
    cout << "Процесс " << cur->name << " Время поступления задачи " << cur-
>taskStart << " Приоритет задачи " << cur->priority << " Такты задачи " << cur-
>taskDuration << endl;
}

void PrintList(TaskList* head)
{
    if (IsEmpty(head))
    {
        cout << "Список процессов пуст." << endl;
        return;
    }

    TaskList* cur = head;
    while (cur != NULL)
    {
        cout << "Процесс " << cur->taskValues->name << " Время поступления задачи "
<< cur->taskValues->taskStart  << " Приоритет задачи " << cur->taskValues->priority
<< " Такты задачи " << cur->taskValues->taskDuration << endl;
        cur = cur->next;
    }
}
```

```
Task* ReturnElemByName(TaskList* head, const string name) // Возвращение элемента по
имени
{
    TaskList* cur = head;

    while (cur) {

        if (cur->taskValues->name == name)
        {
            return cur->taskValues;
        }
        cur = cur->next;
    };
}


string FindLastElem(TaskList* head) // Вывод имени последнего элемента
{
    TaskList* cur = head;

    while (cur->next) {
        cur = cur->next;
    };

    return cur->taskValues->name;
}

bool FindElemByName(TaskList* head, const string name) // Проверка на наличие по
имени
{
    TaskList* cur = head;

    while (cur) {

        if (cur->taskValues->name == name)
        {
            return true;
        }
        cur = cur->next;
    } ;

    return false;
}

bool FindElemByStart(TaskList* head, const int taskStart) // Нахождение элемента по
времени начала
{
    TaskList* cur = head;

    while (cur) {

        if (cur->taskValues->taskStart == taskStart)
        {
            return true;
        }
        cur = cur->next;
    };

    return false;
}

void AddElem(TaskList** head, string name, int priority, int taskStart, int
taskDuration) // Добавление элемента
{
    TaskList* cur = *head;
```

```
    TaskList* p = new TaskList;
    p->taskValues = new Task;

    if (IsEmpty(cur))
    {
        p->taskValues->name = name;
        p->taskValues->priority = priority;
        p->taskValues->taskStart = taskStart;
        p->taskValues->taskDuration = taskDuration;
        p->next = NULL;
        *head = p;
    }
    else
    {
        if (FindElemByName(*head, name))
        {
            cout << "Процесс с таким именем уже существут в списке" << endl;
            return;
        }

        while (cur->next)
        {
            cur = cur->next;
        }
        p->taskValues->name = name;
        p->taskValues->priority = priority;
        p->taskValues->taskStart = taskStart;
        p->taskValues->taskDuration = taskDuration;
        p->next = NULL;
        cur->next = p;
    }

}

void RemoveElem(TaskList** head, string name) // Удаление элемента
{
    TaskList* cur = *head;

    if (IsEmpty(*head))
    {
        cout << "Список процессов пуст." << endl;
        return;
    }

    if (not(FindElemByName(*head, name)))
    {
        cout << "Процесса с таким именем не существует" << endl;
        return;
    }

    if ((cur->taskValues->name == name) && (cur->next != NULL))
    {
        *head = cur->next;
        delete cur->taskValues;
        delete cur;
        return;
    }

    while (cur->next) {
        if (cur->next->taskValues->name == name)
        {
            break;
        }
        cur = cur->next;
    } ;
```

```
    if ((cur == *head) && (cur->next == NULL))
    {
        delete cur->taskValues;
        delete cur;
        *head = NULL;
    }
    else
    {
        TaskList* ptr2 = cur->next;

        cur->next = ptr2->next;
        delete ptr2->taskValues;
        delete ptr2;
    }
}

int listSize(TaskList* head)
{
    TaskList* cur = head;
    int sum = 0;
    while (cur)
    {
        sum += 1;
        cur = cur->next;
    }
    return sum;
}

void pushToQueue(TaskList** Tasks, TaskList** Queue, const string name)
{
    TaskList* cur = *Tasks;
    Task* task = ReturnElemByName(*Tasks, name);

    AddElem(Queue, task->name, task->priority, task->taskStart, task->taskDuration);
    RemoveElem(Tasks, name);
}

void getFromQueue(TaskList** Queue, Task** Processor)
{
    TaskList* cur = *Queue;
    Task* task = cur->taskValues;

    (*Processor)->name = task->name;
    (*Processor)->priority = task->priority;
    (*Processor)->taskStart = task->taskStart;
    (*Processor)->taskDuration = task->taskDuration;

    RemoveElem(Queue, cur->taskValues->name);
}

void pushToStack(Task** Processor, TaskList** Stack)
{
    Task* task = *Processor;

    AddElem(Stack, task->name, task->priority, task->taskStart, task->taskDuration);

    task->name = "";
    task->taskStart = 0;
    task->taskDuration = 0;
    task->priority = 0;
}

void getFromStack(Task** Processor, TaskList** Stack)
{
```

```
    Task* task = ReturnElemByName(*Stack, FindLastElem(*Stack));

    (*Processor)->name = task->name;
    (*Processor)->priority = task->priority;
    (*Processor)->taskStart = task->taskStart;
    (*Processor)->taskDuration = task->taskDuration;

    RemoveElem(Stack, task->name);
}

void StartSequence(TaskList** Tasks, int StackSize, bool &flag)
{

    if (IsEmpty(*Tasks))
    {
        cout << "Список входящих процессов пуст." << endl;
        return;
    }
    clock_t start = clock();
    Task* Processor = new Task;
    TaskList* cur = *Tasks;
    TaskList* Stack = NULL;
    TaskList* Queue_1 = NULL;
    TaskList* Queue_2 = NULL;
    TaskList* Queue_3 = NULL;

    Processor->priority = 0;
    Processor->taskDuration = 0;

    int timer = 1;

    bool ProcessorFree = true;

    while (true)
    {
        cur = *Tasks;
        cout << "Идёт " << timer << " такт" << endl;
        cout << endl;

        while (FindElemByStart(*Tasks, timer))
        {
            if (cur->taskValues->taskStart == timer)
            {
                int priority = cur->taskValues->priority;
                if (priority == 1)
                {
                    pushToQueue(Tasks, &Queue_1, cur->taskValues->name);
                }
                else if (priority == 2)
                {
                    pushToQueue(Tasks, &Queue_2, cur->taskValues->name);
                }
                else if (priority == 3)
                {
                    pushToQueue(Tasks, &Queue_3, cur->taskValues->name);
                }
                cur = *Tasks;
            }
            else
                cur = cur->next;
        }

        if ((ProcessorFree) && (IsEmpty(Stack)))
        {
            if (not(IsEmpty(Queue_1)))
```

```
            {
                getFromQueue(&Queue_1, &Processor);
                ProcessorFree = false;
            }
            else if (not(IsEmpty(Queue_2)))
            {
                getFromQueue(&Queue_2, &Processor);
                ProcessorFree = false;
            }
            else if (not(IsEmpty(Queue_3)))
            {
                getFromQueue(&Queue_3, &Processor);
                ProcessorFree = false;
            }
        }
        while (((not(IsEmpty(Queue_1)) && Processor->priority > 1) ||
(not(IsEmpty(Queue_2)) && Processor->priority > 2)) && not(ProcessorFree))
        {

            if (not(IsEmpty(Queue_1)) && Processor->priority > 1)
            {

                pushToStack(&Processor, &Stack);

                if (listSize(Stack) > StackSize)
                {
                    flag = false;
                    return;
                }
                getFromQueue(&Queue_1, &Processor);
            }
            else if (not(IsEmpty(Queue_2)) && Processor->priority > 2)
            {
                pushToStack(&Processor, &Stack);
                if (listSize(Stack) > StackSize)
                {
                    flag = false;
                    return;
                }
                getFromQueue(&Queue_2, &Processor);
            }
        }

        if (not(IsEmpty(Stack)))
        {
            Task* FirstStack = ReturnElemByName(Stack, FindLastElem(Stack));

            if ((FirstStack->priority <= Processor->priority) && not(ProcessorFree))
            {
                pushToStack(&Processor, &Stack);
                Processor->name = FirstStack->name;
                Processor->priority = FirstStack->priority;
                Processor->taskStart = FirstStack->taskStart;
                Processor->taskDuration = FirstStack->taskDuration;
            }
            if (ProcessorFree)
            {
                getFromStack(&Processor, &Stack);
                ProcessorFree = false;
            }
        }
        if (not(IsEmpty(*Tasks)))
        {
            cout << "Входные задания:" << endl;
            PrintList(*Tasks);
```

```
        }
        if (not(IsEmpty(Queue_1)))
        {
            cout << "Содержимое очереди 1:" << endl;
            PrintList(Queue_1);
        }
        if (not(IsEmpty(Queue_2)))
        {
            cout << "Содержимое очереди 2:" << endl;
            PrintList(Queue_2);
        }
        if (not(IsEmpty(Queue_3)))
        {
            cout << "Содержимое очереди 3:" << endl;
            PrintList(Queue_3);
        }
        if (not(IsEmpty(Stack)))
        {
            cout << "Содержимое стека:" << endl;
            PrintList(Stack);
        }
        if (not(ProcessorFree))
        {
            cout << "Содержимое процессора:" << endl;
            PrintListElem(Processor);
        }

        if (not(ProcessorFree) && Processor->taskDuration)
        {
            Processor->taskDuration -= 1;
            if (Processor->taskDuration <= 0)
            {
                Processor->name = "DONE";
                Processor->taskStart = 0;
                Processor->taskDuration = 0;
                Processor->priority = 0;
                ProcessorFree = true;
            }
        }
        cout << endl;
        if (ProcessorFree && IsEmpty(*Tasks) && (IsEmpty(Stack)) && IsEmpty(Queue_1)
&& IsEmpty(Queue_2) && IsEmpty(Queue_3))
        {
            break;
        }
        timer++;
    }
    clock_t end = clock();
    double seconds = (double)(end - start) / CLOCKS_PER_SEC;
    cout << "Время работы программы обработки: " << seconds << "сек" << endl;
}

void RandomizeProc(TaskList** Tasks)
{
    cout << "Введите количество процессов: ";
    int x = get_num_int() + 1;
    for (int i = 1; i < x; i++)
    {
        int pr = 1 + rand() % 3;
        int st = i;
        int ln = 1 + rand() % 4;
        AddElem(Tasks, to_string(i), pr, st, ln);
    }
}
```

```
int main()
{
    setlocale(LC_ALL, "RUS");
    TaskList* AllTasks = NULL;
    string name;
    int StackSize = 1;
    int choice;
    int queue_choice;
    int taskStart;
    int taskDuration;


    cout << "Добро пожаловать в программу обработки процессов." << endl;
    cout << "Очередь - динамическая. Стек - статический." << endl;

    while (true) // Меню
    {
        cout << "Размер стека - " << StackSize << endl;
        cout << "Выберите дальнейшее действие:" << endl;
        cout << "1. Добавить процесс" << endl;
        cout << "2. Удалить процесс" << endl;
        cout << "3. Задать размер стека" << endl;
        cout << "4. Вывести список процессов" << endl;
        cout << "5. Начать выполнение" << endl;
        cout << "6. Загрузить шаблон процессов" << endl;
        cout << "7. Загрузить шаблон процессов 2" << endl;
        cout << "8. Добавить случайные процессы" << endl;
        cout << "9. Выход" << endl;
        cin >> choice;

        if (choice == 1)
        {
            system("cls");
            cout << "Введите имя процесса" << endl;
            cin.ignore();
            getline(cin, name);
            choice:
                cout << "Введите приоритет процесса(1-3) 1-выс; 3-низ" << endl;
                queue_choice = get_num_int();
                if (queue_choice > 3)
                    goto choice;
            cout << "Введите время начала процесса" << endl;
            taskStart = get_num_int();
            cout << "Введите время выполения процесса" << endl;
            taskDuration = get_num_int();
            AddElem(&AllTasks, name, queue_choice, taskStart, taskDuration);
        }
        else if (choice == 2)
        {
            system("cls");
            cout << "Введите имя процесса" << endl;
            cin.ignore();
            getline(cin, name);
            RemoveElem(&AllTasks, name);
        }
        else if (choice == 3)
        {
            system("cls");
            cout << "Введите желаемый размер стэка: " << endl;
            StackSize = get_num_int();
        }
        else if (choice == 4)
        {
            system("cls");
            PrintList(AllTasks);
```

```
        }
        else if (choice == 5)
        {
            system("cls");
            bool flag = true;
            StartSequence(&AllTasks, StackSize, flag);
            if (not(flag))
                cout << "Stack overflow" << endl;
        }
        else if (choice == 6)
        {
            system("cls");
            AddElem(&AllTasks, "1", 1, 1, 2);
            AddElem(&AllTasks, "2", 1, 2, 1);
            AddElem(&AllTasks, "3", 3, 3, 2);
            AddElem(&AllTasks, "4", 2, 4, 3);
            AddElem(&AllTasks, "5", 1, 5, 3);
            cout << "Загружено пять процессов." << endl;
            cout << "Необходимый размер стека – 1." << endl;
            cout << "Вы можете увидеть список, выбрав пункт 4." << endl;
        }
        else if (choice == 7)
        {
            system("cls");
            AddElem(&AllTasks, "1", 3, 1, 10);
            AddElem(&AllTasks, "2", 2, 2, 10);
            AddElem(&AllTasks, "3", 1, 3, 10);
            cout << "Загружено три процесса." << endl;
            cout << "Необходимый размер стека – 2." << endl;
            cout << "Вы можете увидеть список, выбрав пункт 4." << endl;
        }
        else if (choice == 8)
        {
            system("cls");
            RandomizeProc(&AllTasks);
        }
        else if(choice == 9)
        {
            cout << "Заверщение работы..." << endl;
            break;
        }
        else
        {
            system("cls");
        }
    }
}
```

4. Контрольные примеры

- Контрольный пример 1 доступный в меню в пункте 6.

  Стек будет переполнен, если меньше 1.

- Контрольный пример 2 доступный в меню в пункте 7.

  Стек будет переполнен, если меньше 2.

5. Выводы

В процессе лабораторной работы были изучены структуры данных стек и очередь, а так же получены навыки в их реализации.