ГУАП

КАФЕДРА № 43

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №3

СТЕК И ОЧЕРЕДЬ

по курсу: Структуры и алгоритмы обработки данных

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. № _____ _____ _____
подпись, дата инициалы, фамилия

Санкт-Петербург 2022

Цель работы:

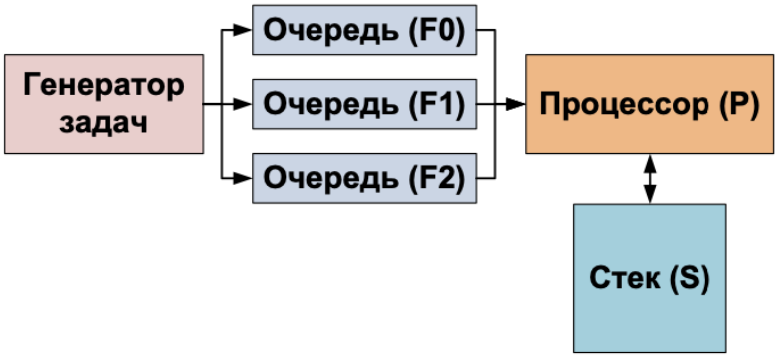
Целью работы является изучение структур данных «стек» и «очередь», а также получение практических навыков их реализации.

Задание на лабораторную работу:

4	1	Стек – динамический; очередь – динамическая

Задача 1.

Система состоит из процессора *P*, трёх очередей *F*0, *F*1, *F*2 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.

Листинг программы:

main.cpp

```
#define AUTO true

#include <iostream>
using namespace std;

#include <cmath>
#include <time.h>
#include <iomanip>

#include "structs.h"
#include "generator.h"
#include "processor.h"

// количемтво генерируемых задач
#define count_generator_tasks 20

int main() {
    // смена кодировки
  system("chcp 65001");

  srand(time(NULL));

  /* Стек - последный вошёл, первый вышел */
  MYList *stack = new MYList(-1);

  /* Очередь - первый вошёл, первый вышел */
  MYList *queue1 = new MYList(-1);
  MYList *queue2 = new MYList(-1);
  MYList *queue3 = new MYList(-1);

  /* Генератор */
  Generator *generator = new Generator(count_generator_tasks,
AUTO);

  /* Процессоры */
  Processors processors(stack, queue1, queue2, queue3, genera-
tor);
  processors.loop();

  delete stack;
  delete queue1;
  delete queue2;
  delete queue3;
  delete generator;

    return 0;
```

generator.h

```
#include <iostream>
using namespace std;
```

```
// генератор задач
class Generator {
public:
  MYList *list = new MYList(-1);

  Generator(int count, bool Auto_flag);
  ~Generator();
  Task get();
  int get_size();

};

// генерирует случайное число в диапазоне от A до B
int random_int(int a, int b) {
  return a + (rand() % ( b - a + 1 ) );
}

// конструктор
Generator::Generator(int Count, bool Auto_flag = true) {
  if (Auto_flag) {
    for (int i = 0; i < Count; i++) {
      list -> append(
        random_int(1, 3),
        i + 1,
        random_int(1, 3)
      );
    }
  } else {
    // Count = read_value("Количество задач: ", false, false,
false);
    cout << "Количество задач: ";
    cin >> Count;

    unsigned int priority;
    unsigned int taskTime;
    unsigned int durationTime;

    for (int i = 0; i < Count; i++) {
      cout << "Задача " << i << endl;

      cout << "Приоритет: ";
      cin >> priority;
      cout << "Момент поступления: ";
      cin >> taskTime;
      cout << "Длительность выполнения: ";
      cin >> durationTime;

      list -> append(
        priority,
        taskTime,
        durationTime
      );
    }
  }


  //draw_stack(list -> get_all(), list -> get_size());
}
```

```
// деструктор
Generator::~Generator() {
  delete list;
}

// получить элемент (после получения он удаляется)
Task Generator::get() {
  return list -> pop(list -> get_size() - 1);
}

// получить текущий размер
int Generator::get_size() {
  return list -> get_size();
```

processor.h

```
#include <iostream>
//#include <conio.h>
using namespace std;

// рисует линию в терминале
void draw_line(int size = 20) {
  for (int i = 0; i < size; i++)
    cout << '-';
  cout << endl;
}

// класс реализующий работу процессороа и его логику
class Processors {
public:
  int tick_num = 0;

  Task buf = {0, 0, 0};
  int durationTime = 0;
  // bool run_processor = false;
  bool last_sended = false;

  MYList *stack;

  MYList *queue1;
  MYList *queue2;
  MYList *queue3;

  Generator *generator;

  bool run = true;

  Processors(MYList *stack1, MYList* _queue1, MYList* _queue2,
MYList* _queue3, Generator* generator1);
  void tick();
  void loop();
};

// конструктор
Processors::Processors(MYList *stack1, MYList* _queue1, MYList*
_queue2, MYList* _queue3, Generator* generator1) {
  stack = stack1;
  queue1 = _queue1;
  queue2 = _queue2;
  queue3 = _queue3;
```

```
  generator = generator1;

  Task buf;
  int i;

  int count_generator_task_to = generator -> get_size() / 3;

  for (i = 0; i < count_generator_task_to; i++) {
    buf = generator -> get();
    queue1 -> append(
      buf.priority,
      buf.taskTime,
      buf.durationTime
    );
  }

  for (i = 0; i < count_generator_task_to; i++) {
    buf = generator -> get();
    queue2 -> append(
      buf.priority,
      buf.taskTime,
      buf.durationTime
    );
  }

  int generator_remainder = generator -> get_size();
  for (i = 0; i < generator_remainder; i++) {
    buf = generator -> get();
    queue3 -> append(
      buf.priority,
      buf.taskTime,
      buf.durationTime
    );
  }

  run = true;
}

// цикл процессора
void Processors::loop() {
  char c;
  while (run) {
    draw_line();
    tick();

    cout << "Процессор: ";
    draw_task(buf, durationTime);

    cout << "Генератор:" << endl;
    draw_stack(generator -> list -> get_all(), generator ->
get_size());

    cout << "Стек:" << endl;
    draw_stack(stack -> get_all(), stack -> get_size());

    cout << "Очередь #1:" << endl;
    draw_stack(queue1 -> get_all(), queue1 -> get_size());
    cout << "Очередь #2:" << endl;
    draw_stack(queue2 -> get_all(), queue2 -> get_size());
    cout << "Очередь #3:" << endl;
```

```
    draw_stack(queue3 -> get_all(), queue3 -> get_size());

    // 13 Enter
    // 27 escape
    // 8 backspace
    // 32 space
    c = getchar(); //getch();
    if (c == 8) run = false;

    if (!run) break;
  }
}

// итерция процессора
void Processors::tick() {
  tick_num++;
  cout << tick_num << " такт" << endl;

  // если очереди пусты, то берём задачи из стека
  int queue_summary_size = queue1 -> get_size() + queue2 ->
get_size() + queue3 -> get_size();
  if (queue_summary_size <= 0 && durationTime <= 0) {
    if (stack -> get_size() > 0) {
      buf = stack -> pop((stack -> get_size()) - 1);
      durationTime = buf.durationTime;
      return;
    } else {
      buf = {0, 0, 0};

    }
  }

  // если процессор пуст, то берём новую задачу из очередей
  if (queue_summary_size > 0 && durationTime <= 0) {
    if (queue1 -> get_size() > 0) {
      buf = queue1 -> pop(0);
    } else if (queue2 -> get_size() > 0) {
      buf = queue2 -> pop(0);
    } else if (queue3 -> get_size() > 0) {
      buf = queue3 -> pop(0);
    }
    durationTime = buf.durationTime;
    return;
  }

  Task buf_priority;
  bool priority = false;

  // проверка на преоритет задачи в стеках
  if (queue1 -> get_size() > 0 && queue1 -> get(0).priority >
buf.priority) {
    buf_priority = queue1 -> pop(0);
    priority = true;
  } else if (queue2 -> get_size() > 0 && queue2 -> get(0).prior-
ity > buf.priority) {
    priority = true;
    buf_priority = queue2 -> pop(0);
  } else if (queue3 -> get_size() > 0 && queue3 -> get(0).prior-
ity > buf.priority) {
    priority = true;
```

```
    buf_priority = queue3 -> pop(0);
  }

  // если имеется задача с более высоким приоритетом, то
добавляем старую в стек и берём новую
  if (priority) {
    stack -> append(
      buf.priority,
      buf.taskTime,
      buf.durationTime
    );

    buf = buf_priority;
    durationTime = buf.durationTime;
    return;
  }


  //работа процессора
  if ((durationTime > 0)) {
    run = true;
    durationTime -= 1;
  } else {
    run = false;
  }

}
```

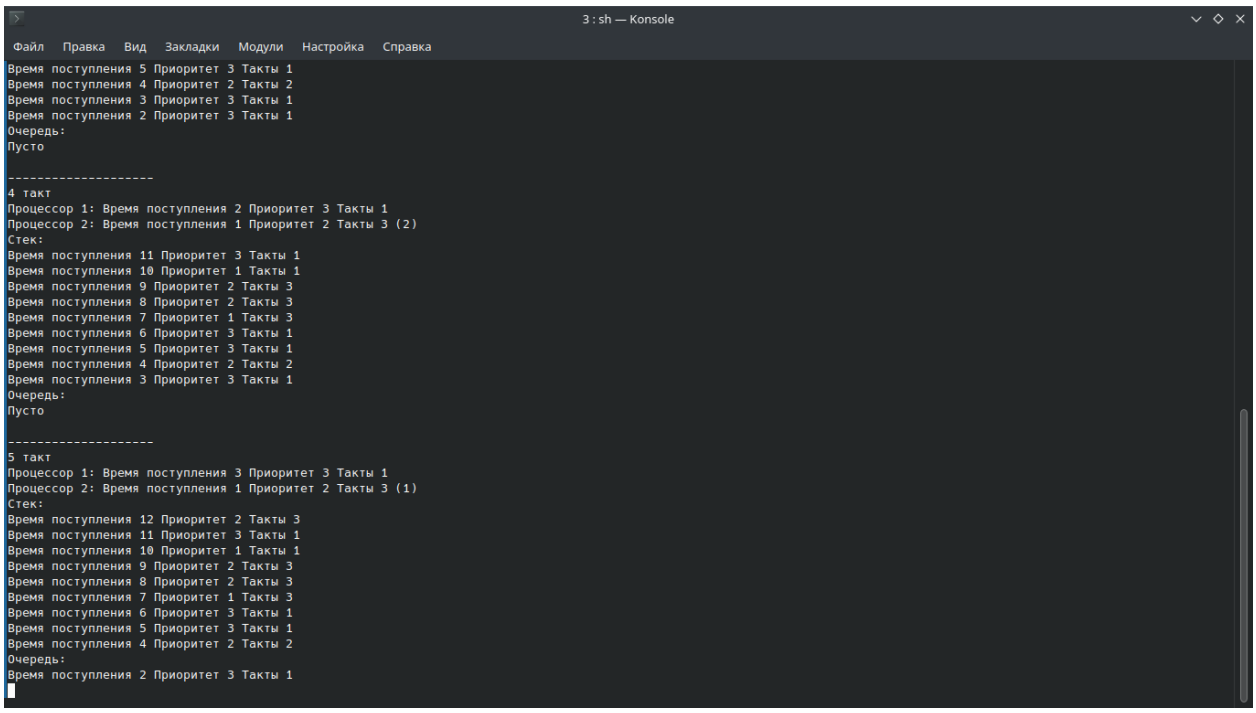
structs.h

```
#include <iostream>
using namespace std;

// элемент стека или очереди
struct Task {
  unsigned int priority; // приоритет
  unsigned int taskTime; // момент поступления
  unsigned int durationTime; // длительность выполнения
};

// вывод элемента стека или очереди
void draw_task(Task task, int tick = 0) {
  if (task.durationTime != 0) {
    cout << " - ";
    cout << "Время поступления " << task.taskTime;
    cout << " Приоритет " << task.priority;
    cout << " Такты " << task.durationTime;
    if (tick != 0) cout << " (" << tick << ")";
    cout << endl;
  } else {
    cout << "Пусто" << endl;
  }
}

// вывод всего стека или очереди
void draw_stack(Task *stack, int size) {
  if (size > 0) {
    for (int i = 0; i < size; i++) {
```

```
      draw_task(stack[i]);
    }
  } else {
    cout << "Пусто" << endl;
  }

}

// класс реализующий Стек и Очередь
class MYList {
public:
  Task *tmp = nullptr;
  int count = 0;
  int max_count;

  MYList(int Max_count);
  ~MYList();
  bool append(int priority, int taskTime, int durationTime, bool
end);
  Task pop(int id);
  Task get(int id);
  Task *get_all();
  int get_size();

};

// конструктор
MYList::MYList(int Max_count) {
  max_count = Max_count;
  count = 0;
  tmp = nullptr;

}

// деструктор
MYList::~MYList() {
  free(tmp);
}

// добавление элемента
bool MYList::append(int priority, int taskTime, int duration-
Time, bool end = false) {
  if ((max_count == -1) || (count < max_count)) {
    if (tmp != nullptr) {
      tmp = (Task*)realloc(tmp, (++count) * sizeof(Task));
    } else {
      tmp = (Task*)malloc(sizeof(Task));
      count = 1;
    }

    if (end) {
      tmp[count - 1].priority = priority;
      tmp[count - 1].taskTime = taskTime;
      tmp[count - 1].durationTime = durationTime;

    } else {
      Task *buf = (Task*)malloc((count) * sizeof(Task));

      buf[0].priority = priority;
      buf[0].taskTime = taskTime;
```

```
      buf[0].durationTime = durationTime;

      for (int i = 0; i < count - 1; i++) {
        buf[i + 1].priority = tmp[i].priority;
        buf[i + 1].taskTime = tmp[i].taskTime;
        buf[i + 1].durationTime = tmp[i].durationTime;
      }

      tmp = buf;

    }

    return true;
  } else return false;

}

// вырезание элемента с возвратом
Task MYList::pop(int id) {
  Task *buf = tmp;

  tmp = (Task*)malloc((count - 1) * sizeof(Task));

  int index = 0;
  for (int i = 0; i < count; i++) {
    if (i != id)
      tmp[index++] = buf[i];
  }

  --count;

  return buf[id];

}

// получить 1 элемент
Task MYList::get(int id) {
  return tmp[id];
}

// получить все элементы
Task *MYList::get_all() {
  return tmp;
}

// получить текущий размер
int MYList::get_size() {
  return count;
}
```

Скриншот выполнения программы:



Вывод:

Мы изучили структуры данных «стек» и «очередь», а также полученили практические навыки их реализации.