Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования «Нижегородский государственный университет

им. Н.И. Лобачевского»

**Институт информационных технологий, механики и математики**

**Отчет по лабораторной работе**

**ИССЛЕДОВАНИЕ РАЗЛИЧНЫХ ИГРОВЫХ АЛГОРИТМОВ ПОИСКА НАИЛУЧШЕГО ХОДА НА ПРИМЕРЕ ИГРЫ «РУССКИЕ ШАШКИ»**

**Выполнили**:

студент группы 0823-1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Грачева Е. А.

студент группы 0826-1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Панов А. А.

Подпись

**Проверил**:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Подпись

Нижний Новгород

2017

**Содержание**

Оглавление

[Введение 3](#_Toc482135543)

[Постановка задачи 4](#_Toc482135544)

[Алгоритмы 5](#_Toc482135545)

[Реализация 8](#_Toc482135546)

[Алгоритмическое ядро (С++) 10](#_Toc482135547)

[Графический интерфейс (C#) 18](#_Toc482135548)

[Апробация 23](#_Toc482135549)

[Заключение 24](#_Toc482135550)

[Литература 25](#_Toc482135551)

[Приложение. Фрагменты исходного кода программы 26](#_Toc482135552)

[template<class Data> class Stack 26](#_Toc482135553)

# Введение

В настоящее время в программировании популярно направление, связанное с компьютеризацией различных логических игр. Мотивируется это тем, что, во-первых, компьютерные игры достаточно востребованы; во-вторых, логические игры представляют собой хороший материал для исследования различных алгоритмов поиска наилучшего хода.

# Постановка задачи

Цели работы: исследовать различные алгоритмы поиска лучшего хода в игровой программе, реализовать и по возможности оптимизировать их, а также выбрать наиболее эффективные. В качестве разрабатываемой игровой программы были выбраны “русские шашки”, т.к. игра имеет сравнительно простую логику, но при этом является хорошим примером, иллюстрирующим различные алгоритмы поиска оптимального хода. Сравнение с известными программами, такими как Тундра или Аврора, позволит оценить эффективность наших алгоритмов.

# Алгоритмы

Шашки – игра с конечным числом состояний. Это означает, что, обладая бесконечными вычислительными ресурсами, мы смогли бы найти решение этой игры – то есть такую стратегию, следуя которой всегда можно было бы выиграть или, по крайней мере, сыграть вничью. Каждая позиция pos в шашках - это либо победа белых, либо победа черных, либо ничья. Определим функцию ***f:P→Z***, которая ставит в соответствие позиции ***pos*** из множества всевозможных позиций ***P*** целое число, отражающее «выгодность» этой позиции для текущего игрока***.*** Такая функция называется *оценочной функцией.*Если бы у нас был бесконечно быстрый компьютер, мы могли бы вычислить ее следующим образом:

* присвоим всем финальным позициям значения –1, 0, 1, в зависимости от исхода игры;
* применим рекурсивное правило где **pos** соответствует ходу текущего игрока, **pos‘** соответствует ходу противника, а  **pos→pos‘** обозначает все допустимые ходы из позиции **pos**.

Чем больше значение функции, тем выгодней позиция. Таким образом, ставя каждому ходу в соответствие его оценку, мы бы могли однозначно определить лучший ход.

Количество [возможных позиций](http://en.wikipedia.org/wiki/Shannon_number) в шашках составляет около 5\*1020. Вычисления такого масштаба выполнить практически невозможно (канадским ученым потребовалось 200 компьютеров и 20 лет вычислений). Следовательно, находить наилучший ход нужно другим способом.

### Алгоритм полного перебора

Самым простым способом получить если не лучший, то хотя бы осмысленный ход, является метод полного перебора с возвратом в соединении с рекурсией. Если сгенерировать все возможные ходы из данной позиции для белых, затем для каждого такого хода повторить те же действия для черных, потом снова для белых и т.д., получится простой и понятный рекурсивный алгоритм, основанный на переборе позиций. Пример дерева игры, возникающего таким образом, представлен на рис.х,х1.

Однако, как уже было сказано, рассмотреть все возможные варианты до конца невозможно, поэтому приходится ограничивать глубину рекурсии. Пусть на каждом шаге известна текущая глубина рекурсии – параметр ***depth***, первоначально равный максимальной заданной глубине. Тогда если ***depth***≠0, то рекурсия будет вызываться с параметром ***depth-1***, в противном случае вычисления следует прекратить и вызвать *оценочную функцию[[1]](#footnote-1)*.

Блок-схема алгоритма полного перебора представлена на рис.х.

### Оценочная функция

Пусть P - множество всевозможных позиций на доске. Функция f:P→Z, ставящая в соответствие некоторой позиции из множества P целое число, отражающее «выгодность» этой позиции для текущего игрока, называется оценочной функцией.

Простейшая оценочная функция считает шашки на доске, т.е. суммирует вес всех белых шашек и вычитает из полученного результата сумму всех черных шашек. Если шашек поровну, функция вернет 0.

Установим вес шашек следующим образом:

простая шашка – 50;

дамка – 350.

Таким образом, простая оценочная функция при расстановке шашек как на рис. для белого игрока вернет число 250, для черного игрока -250.

Более сложная оценочная функция учитывает ценность полей доски, которая определяется следующими принципами:

* надо стараться по возможности подвигаться вперед, т.к. чем ближе к последней горизонтали, тем сила полей больше;
* поскольку ценность шашек a1 и h2 является наименьшей, то их надо по возможности быстро ввести в игру;
* учитывая, что с полей c5 и f4 удобно препятствовать развитию сил противника, следует бороться за овладением центральными полями c5 и f4, стремиться к захвату центра;
* следует придерживаться принципа пропорционального, равномерного распределения шашек по обоим флангам;
* шашки с полей c1, e1, g1 лучше без особой надобности не сдвигать, чтобы не открывать поля последнего ряда;
* дамке по возможности стоит контролировать главную диагональ, т.к. она будет препятствовать продвижению противника в дамки.

Согласно этим правилам, ценность полей доски была обозначена следующим образом (рис.х):

На практике оказалось, что бот, использующий простую оценочную функцию, проигрывает боту, использующему более сложную оценочную функцию.

### Alpha-Beta с отсечениями

Рассмотрим основной алгоритм оптимизации перебора - alpha-beta c отсечениями. Суть его в том, что для получения оценки такой же точности, как и при полном переборе, совершенно не обязательно просматривать все варианты. Для определения отсекаемых вариантов не требуется знать особенностей данной игры.

В рекурсивной функции мы должны ввести две новые переменные — максимум для белых (alpha) и максимум для черных (beta). При начальном вызове обе эти величины равны минимально возможному значению. Если в какой-то позиции, например, для черных, мы получили результат, превышающий максимум для черных, достигнутый до этого, мы увеличиваем этот максимум (beta). Теперь рассмотрим некоторую строку игры. Представим, что где-то в глубине дерева перебора мы обнаружили, что максимумы белых и черных сравнялись. Допустим, мы просчитываем позицию для белых. Если мы продолжим перебирать в данной позиции, то максимум для белых может еще увеличиться, а может остаться прежним, но он уже сравнялся с максимумом черных. Это значит, что когда программа поднимется на уровень выше (рекурсивно), результат не будет записан, т. к. он не превышает максимума для черных в этой позиции. Также это значит, что в данной позиции для белых мы можем прекратить перебор и вернуть полученный результат досрочно. Дальше считать нет смысла.

Проиллюстрируем вышесказанное на примере (рис.х). Пусть на каком-либо шаге рекурсии максимум для белых (alpha) равен 4. Сделаем какой-либо ход и просчитаем возможные ходы для черных (опустимся на уровень ниже). Пусть при дальнейших вычислениях для одного из ходов результат равен 1 (beta=1). Если впоследствии найдется ход, оценка которого будет меньше 1, то он будет записан в beta, но не будет записан в alpha, и наоборот, если найдется ход, оценка которого будет больше 1, то он не будет записан даже в beta, следовательно, и не будет записан в alpha. Получается, дальнейшие вычисления бессмысленны.

Блок-схема алгоритма поиска с отсечениями представлена на рис..

На практике результаты показали, что…

### Форсированные варианты

В некоторых ситуациях, например, в случае размена, прекращение вычислений по достижении максимальной глубины рекурсии может привести к крайне неверной оценке позиции (*эффект горизонта*). Рассмотрим пример (рис.х). Пусть максимальная глубина рекурсии равна 2. Ход белых. Даже не очень опытному игроку очевидно, что лучший ход – *d8-h8*, поскольку он приводит к полному уничтожению противника. Однако бот не увидит преимуществ такого хода и сходит *h2-b8*. Чтобы избежать подобных ситуаций, отдельные ветки стоит просчитывать на большую глубину.

В шашках форсированными вариантами являются взятия. Поэтому в конце каждой ветки рекурсии вызывается упрощенная функция поиска, рассматривающая только взятия. По завершению размена вызывается оценочная функция. Если же взятий не было, оценочная функция вызывается сразу.

# Реализация

Проект состоит из двух частей — вычислительного ядра и графического интерфейса. В вычислительной части написаны различные функции поиска лучшего хода, то есть искусственный интеллект для бота. Графическая оболочка предоставляет визуализацию игры, предоставляет настройки бота и дает возможность ходить реальному игроку.

Вычислительное ядро

Графическая оболочка

find best move

best move

Нахождение наилучшего хода требует большого объема вычислений и для обеспечения наилучшей производительности вычислительное ядро написано в виде динамической библиотеки на C++

Графическая оболочка, написана на C#, так как данный язык предоставляет удобный графический инструментарий.

Итоговое приложение отображает игру с помощью графической оболочки и находит лучший ход с помощью вычислительного ядра.

## Алгоритмическое ядро (С++)

### Описание структур данных

Далее представлены основные классы, используемые в части, отвечающей за ход искусственного интеллекта.

1. Класс Checker

Класс представляет собой структуру хранения для шашки.

class Checker

{

int Color; //цвет шашки, 0-белый или 1-черный

int Type; //тип шашки, 0-шашка или 1-дамка

int Coord; //координаты шашки (3 бита – координата по x, 3 бита – координата по y)

int Num; //идентификатор шашки

int NextNum; //номер следующей шашки, для реализации списка шашек на массиве[[2]](#footnote-2)

int PrevNum; //номер предыдущей шашки

public:

Checker(): Color(0), Type(0), Coord(0), Num(0), NextNum(0), PrevNum(0) { }

~Checker() {}

void SetColor(int color);

void SetType(int type);

void SetCoord(int coord);

void SetNum(int num); //устанавливается 1 раз

int GetColor();

int GetType();

int GetCoord();

int GetNum();

//изменяют поле на противоположное значение

void ChangeColor();

void ChangeType();

//дружественные классы

friend class ListOfCheckers;//список шашек

friend class LChIterator;//итератор для перемещения по списку шашек

protected://методы, предназначенные для использования классом ListOfCheckers

void SetNextNum(int num);

void SetPrevNum(int num);

int GetNextNum();

int GetPrevNum();

};

1. Класс ListOfCheckers и класс LChIterator

Класс *ListOfCheckers* является типом, представляющим список шашек на массиве. Необходимость его создания обусловлена тем, что в программе необходимо как обращаться к шашке по индексу, так и часто удалять и добавлять шашки. Класс *LChIterator* является итератором к этому классу.

class ListOfCheckers //список шашек

{

Checker List[13]; //12 шашек + голова, чтобы список никогда не был пустым

//шашки нумеруются от 1 до 12

public:

ListOfCheckers();

~ListOfCheckers() {}

//генерирует первоначальную позицию по заданным массивам типов и координат

void GenerateInitialPosition(int color, int\* types, int\* coords, int n);

void Insert(int num); //вставляет шашку в список

void Delete(int num); //удаляет шашку из списка

int IsEmpty();//проверяет, пустой ли список

Checker& operator[] (int i); //оператор индексации

void Clean(); //очищает стек

void Bind(int n);//связывает список друг за другом

//итератор

typedef LChIterator iterator;

iterator begin() { return iterator(&(List[List[0].GetNextNum()]),List); }

iterator end() { return iterator(List, List); }

};

class LChIterator //итератор

{

Checker\* begin; //указатель на голову списка

Checker\* p; //шашка, на которую указывает итератор

public:

LChIterator() {}

//инициализатор полей

LChIterator(Checker\* p\_, Checker\* begin\_) :p(p\_), begin(begin\_) {};

//арифметические операторы

LChIterator& operator++();

LChIterator& operator--();

LChIterator& operator++(int);

LChIterator& operator--(int);

//операторы приведения типов

operator Checker\* ();

operator int();

//операторы разыменования

Checker\* operator->();

Checker operator\* (LChIterator it);

//операторы сравнения

friend int operator==(LChIterator it, Checker\* pch);

friend int operator!=(LChIterator it1, LChIterator it2);

};

1. Класс Board

Доска представляет собой одномерный массив из указателей на Checker. Это необходимо для того, чтобы, зная координату, сразу находить соответвствующую ей шашку.

class Board

{

Checker\* board[64];

public:

Board();

Board(ListOfCheckers& white, ListOfCheckers& black);//заполняет доску фигурами

void Set(ListOfCheckers& white, ListOfCheckers& black); //заполняет доску фигурами

int IsEmpty(int cell);//проверяет, пустая ли клетка

Checker\*& operator [](int cell);//индексация

void Clean();//очищает доску

};

1. Класс Move

При реализации алгоритмов выясняется, что нужно как-то хранить ход шашки. Следующий класс предоставляет все необходимые для этого поля и методы.

class Move

{

int Color;//цвет шашки, которая ходит

int StartCoord;//начальные координаты хода

int FinalCoord;//конечные координаты хода

int Num;//идентификатор перемещающейся шашки

int Type;//изменился ли тип шашки во время хода

int NEaten;//число съеденных шашек

int Eaten[12] = { 0 };//массив съеденных шашек

public:

Move() :Color(0), Type(0), StartCoord(0), FinalCoord(0), Num(0), NEaten(0) {}

//операторы сравнения

friend int operator==(const Move& m1, const Move& m2);

friend int operator!=(const Move& m1, const Move& m2);

~Move() {}

//устанавливают поля класса

void Set(int \_color, int \_s\_coord, int \_f\_coord, int \_num, int \_type);

void SetColor(int color);

void SetStartCoord(int coord);

void SetFinalCoord(int coord);

void SetNum(int num);

void SetType(int type);

void SetNEaten(int neaten);

void SetEaten(int\* arrEaten);//записывает номера съеденных шашек в массив

//возвращают значения полей класса

int GetColor();

int GetStartCoord();

int GetFinalCoord();

int GetNum();

int GetType();

int GetNEaten();

void GetEaten(int\* arrEaten);//возвращает съеденные шашки в массив arrEaten

};

1. Класс Cache

На каждом шаге рекурсии необходимо где-то хранить возможные ходы. Можно было бы создавать на каждом шаге рекурсии массив ходов, но, во-первых, это может занимать некоторое время, во-вторых, выделять место пришлось бы с большим запасом, поскольку неизвестно, сколько ходов может быть сгенерировано для определенной позиции. Поэтому мы выделяем место для хранения ходов в виде большого массива один раз в самом начале, а во время работы рекурсии просто записываем туда ходы. Очищение происходит посредством перемещения указателя на конец массива в сохраненную точку.

Такая структура хранения представлена следующим классом.

class Cache

{

private:

int Size;//размер кэша

Move \*pData;//указатель на начало

Move \*pLast;//указатель на конец

public:

Cache():Size(0), pData(0), pLast(0){};

Cache(int size) : Size(size);//выделяет динамически массив размера size

void Push(Move data); //запись данных в кэш

Move& Pop();//извлечение из кэша

Move\* GetpLast();//получить указатель на конец кэша

void Rollback(Move \*saved);//очищение кэша до сохраненной позиции saved

int CurPos(); //получить текущий размер кэша

Move operator[](int i);//индексирование

bool IsFull();//проверяет, заполнен ли кэш

bool IsEmpty(); //проверяет, пуст ли кэш

void Clean(); //очищает кэш полностью

~Cache();

};

### Описание алгоритмов

Основные алгоритмы, используемые в программе (полный перебор[[3]](#footnote-3), перебор с отсечениями[[4]](#footnote-4), форсирование вариантов[[5]](#footnote-5), использование различных оценочных функций[[6]](#footnote-6)) описаны в разделе Алгоритмы. Здесь мы представим вспомогательные алгоритмы, необходимые для реализации поиска.

1. Генерация всевозможных ходов*[[7]](#footnote-7)*

Генерация ходов для позиции в шашках не является тривиальной задачей, поскольку, во-первых, возможны взятия сразу нескольких шашек, во-вторых, передвижения дамки (особенно с взятиями) требуют внимательного изучения правил игры. Алгоритм взятий для шашек можно описать следующим образом:

1. исследуем все четыре направления (по диагонали влево-вверх, вправо-вверх, вправо-вниз и влево-вниз) на наличие взятий (рассматриваются только соседние клетки);
2. если возможно взятие, «перепрыгнем» через битую шашку и поменяем ее цвет на противоположный (это необходимо, поскольку «есть» шашку дважды и убирать ее с доски до окончания хода нельзя по правилам), иначе завершим поиск взятий;
3. если текущая позиция находится на последнем ряду, изменим тип шашки на дамку и применим алгоритм взятий для дамки, начиная с п.3 (см. далее);
4. в противном случае исследуем все вышеперечисленные направления, кроме того, откуда пришла шашка, на наличие взятий;
5. повторим алгоритм, начиная с п.2.

Алгоритм взятий для дамок будет выглядеть несколько сложнее:

1. исследуем все четыре направления на наличие взятий (рассматриваются не только соседние клетки);
2. если возможно взятие, «перепрыгнем» через битую шашку (поменяв ее цвет) и поставим дамку на первую свободную клетку, иначе завершим поиск взятий;
3. исследуем направления, перпендикулярные направлению, с которого пришла дамка, на наличие взятий;
4. если в этих направлениях есть взятия, повторим алгоритм, начиная с п.2, и вернемся в эту же точку;
5. в противном случае передвинем шашку на 1 клетку в первоначальном направлении (если это невозможно, то заверши поиск взятий);
6. повторим алгоритм, начиная с п.2, до тех пор, пока впереди не возникнет преграда (шашка или граница доски);
7. если впереди возможно взятие, повторим алгоритм, начиная с п.2.

Алгоритм поиска возможных ходов реализован в виде следующих функций:

int SearchEatChecker(Checker& ch);//поиск и запись взятий для шашки, возвращает 0

int SearchEatDamka(Checker& ch);//поиск и запись взятий для дамки, возвращает 0

int SearchMoveChecker(Checker& ch);//поиск и запись простых ходов для шашки, возвращает 1, //если возможны взятия, иначе возвращает 0

int SearchMoveDamka(Checker& ch);//поиск и запись простых ходов для дамки, возвращает 1, //если возможны взятия, иначе возвращает 0

void Generate(ListOfCheckers& list);//обобщает все вышеперечисленные функции

void Generate(ListOfCheckers& list);//генерирует только взятия (для форсирования)

Функция *Generate* выполняет следующий алгоритм:

1. сохраним текущее состояние кэша;
2. рассмотрим шашку из списка шашек, если такая есть, иначе завершим алгоритм;
3. если это простая шашка, то вызовем для нее функцию *SearchMoveChecker*, иначе вызовем функцию *SearchMoveDamka*;
4. если функция вернула значение 0, то повторим алгоритм для следующей шашки, начиная с п.2;
5. если функция вернула значение 1, то вернем кэш в сохраненное состояние;
6. запустим для текущей шашки функцию *SearchEatChecker* или *SearchEatDamka,* в зависимости от ее типа;
7. для следующей шашки повторим алгоритм, начиная с п.6;
8. если такой шашки нет, то завершим алгоритм.
9. Совершение хода и отмена хода[[8]](#footnote-8)

Также для выполнения рекурсивного поиска необходимо уметь делать ход и отменять его. Ход совершается по следующему алгоритму:

1. удалим все съеденные шашки с доски;
2. удалим основную шашку с доски;
3. поставим шашку на новое место;
4. присвоим полю *Coord* текущей шашки новое значение;
5. если необходимо, изменим тип шашки.

Отмена хода выполняется в обратном порядке.

## Графический интерфейс (C#)

Для удобства графическая часть выполнена по схеме MVC:

View

Controller

Model

I’m update!

user’s change

change date

В ней данные разделяются на три отдельных компонента: модель, представление и контроллер — таким образом, что модификация каждого компонента может осуществляться независимо. Также изменение и отображение данных происходит строго по заданной схеме:

* ***Модель*** (*Model*) предоставляет данные и реагирует на команды контроллера, изменяя своё состояние;
* ***Представление*** (*View*) отвечает за отображение данных модели пользователю, реагируя на изменения модели;
* ***Контроллер*** (*Controller*) интерпретирует действия пользователя, оповещая модель о необходимости изменений;

### Описание структур данных

#### Класс Game

Класс реализует модель и содержит информацию и алгоритмы необходимые для игры.

##### Структуры данных, хранящие информацию об игре:

* static LogicBoard — структура для хранения игровой доски
* Checker — структура для хранения фигуры
* List<Checker>[] checkers;

checkers[0]—список шашек для белого игрока;

checkers[1]—список шашек для черного игрока;

* List<Checker>selectedCheckers; — выделенные на игровой доске шашки
* static Moves — структура для хранения ходов игрока
* static BotMove — структура для хранения ходов бота

##### Состояние игры:

* Color — структура для хранения цвета игрока/фигуры

Принимает следующие значения:

white

black

* StatusPlayer — структура для хранения статуса игрока

Принимает следующие значения:

bot — игрок это компьютер

human — игрок это человек

empty — игрок не установлен

* StatusApplication statusApplication; — структура для хранения статуса приложения

Принимает следующие значения:

game — идет процесс игры

constructor — выбран конструктор и идет конструирование игры

menu — выбрано меню

##### Настройки поиска лучшего хода:

* Search — структура для хранения типа поиска лучшего хода

Принимает следующие значения:

FullSearch — тип поиска [Search](#_Search)

AlphaBetaSearch — тип поиска [Alpha-Beta](#_Alpha-Beta)

ForcedSearch — тип поиска [Alpha-Beta forcing](#_Форсирование)

empty — тип поиска не установлен

* Evaluate — структура для хранения типа оценочной функции

Принимает следующие значения:

SimpleEvaluate — тип оценочной функции

SmartEvaluate — тип оценочной функции

empty — тип оценочной функции не установлен

#### Класс Form1

Класс является частью отображения и содержит графическую часть игры.

##### Структуры данных хранящие, визуализацию игры:

* ViewBoard — структура для хранения игровой доски

##### Структуры данных, реализующие MVC:

* iController — интерфейс контроллера

##### Структуры данных, осуществляющие управление (кнопки и проч)

#### Класс Controller

Класс обрабатывает запросы отображения и изменяет модель.

##### Структуры данных, реализующие MVC:

* iGame — интерфейс модели
* Form1 — отображение
* SettingForm — отображение

### Описание алгоритмов

##### Алгоритм игры

# Апробация

# Заключение

В данной лабораторной работе вычислялись значения арифметических выражений. Для этого выражения разбивались на *лексемы* и проверялись на *корректность* ввода с помощью простейшего конечного автомата. Затем выполнялась необходимая сортировка лексем и итоговое вычисление.

# Литература

* Гергель В. П., Лабутина А. А. Учебно-образовательный комплекс по методам программирования //Нижний Новгород: ННГУ им. НИ Лобачевского. – 2007.

# Приложение. Фрагменты исходного кода программы

## template<class Data> class Stack

Stack(int\_size)//конструкторинициализации

{

if (\_size>= 0)

top = -1;

elsethrowNEGATIVE\_INDEX;

size = \_size;

ar = newData[size];

}/\*-------------------------------------------------------------------------\*/

void push(Datanum)

{

if (top + 1 < size)

ar[++top] = num;

elsethrowSTACK\_FULL;

}/\*-------------------------------------------------------------------------\*/

voidquickPush(Datanum)

{

ar[++top] = num;

}/\*-------------------------------------------------------------------------\*/

Datapop()

{

if (top > 0)

return ar[top--];

elsethrowSTACK\_EMPTY;

}/\*-------------------------------------------------------------------------\*/

DataquickPop()

{

return ar[top--];

}/\*-------------------------------------------------------------------------\*/

Data&showTop()

{

return ar[top];

}/\*-------------------------------------------------------------------------\*/

DatagetElem(inti)

{

return ar[i];

}/\*-------------------------------------------------------------------------\*/

boolisEmty()

{

return (top == -1);

}/\*-------------------------------------------------------------------------\*/

boolisFull()

{

return (top + 1 == size);

}/\*-------------------------------------------------------------------------\*/

intgetNumTop()

{

return top;

}/\*-------------------------------------------------------------------------\*/

voidsetTop(int\_top)

{

top = \_top;

}/\*-------------------------------------------------------------------------\*/

~Stack()//деструктор

{

if (ar != 0)

delete[] ar;

}/\*-------------------------------------------------------------------------\*/

1. См … [↑](#footnote-ref-1)
2. См п.2 [↑](#footnote-ref-2)
3. Реализация см такое-то приложение [↑](#footnote-ref-3)
4. [↑](#footnote-ref-4)
5. [↑](#footnote-ref-5)
6. [↑](#footnote-ref-6)
7. [↑](#footnote-ref-7)
8. [↑](#footnote-ref-8)