|  |  |  |
| --- | --- | --- |
| МИНОБРНАУКИ РОССИИ | | |
| Федеральное государственное автономное  образовательное учреждение высшего образования  «Пермский государственный национальный  исследовательский университет» | | |
|  | Институт компьютерных наук и технологий | |
| **ОТЧЁТ**  по индивидуальной работе №2  по дисциплине «Язык программирования C++»  Вариант 6 | | |
|  | | Работу выполнил  студент группы ИТ-17,18-2024 1 курса  Поздеев А.Д.  «11» июня 2025 г. |
| Работу проверил  Кухар Д. А.  «25» июня 2025 г. |
| Пермь 2025 | | |

СОДЕРЖАНИЕ

[Постановка задачи 3](#_Toc153130027)

[Алгоритм решения 3](#_Toc153130028)

[Тестирование 3](#_Toc153130029)

[Код программы 3](#_Toc153130030)

[Инструкция по применению стилей и оформлению работы 4](#_Toc153130031)

# Постановка задачи

Для заданного подмножества набора костей домино определить, можно ли их

выложить в ряд, не нарушая правил. Если можно, то представить один любой вариант такого разложения. Например, для входных данных 31, 00, 13, получаем ответ: некорректные входные данные; для входных данных 02, 04, 42 ответ: можно, 04, 42, 20. Использовать двухсвязный список.

# Алгоритм решения

Домино в двусвязном списке представлю с помощью класса, который я решил оригинально назвать “Dominode”. Он будет хранить два значения - первая и вторая половинка кости домино, а также указатели на предыдущий и следующий элементы и булевый тег, который показывает, была ли выведена кость или нет.

class Dominode {

public:

int pattern[2];

Dominode\* prev;

Dominode\* next;

Dominode(int f, int s) : pattern{f,s}, prev(nullptr), next(nullptr) {}

bool used=0;

};

Пишем простую функцию для добавления домино в список:

void addDomino(Dominode\*& head, int first, int second){

Dominode\* newDomino = new Dominode(first, second);//новый нод

if (!head) //если список пустой, делаем нод началом списка

head = newDomino;

else{

Dominode\* temp = head;

while (temp->next)//двигаемся в конец списка

temp = temp->next;

temp->next = newDomino;//добавляем нод в список

newDomino->prev = temp;

}

}

Теперь пишем функцию для проверки, можно ли выложить домино в ряд, не нарушая правил. Номера точек у примыкающих половинок разных домино должно совпадать. Из этого следует, что количество половинок домино с одинаковым количеством точек должно быть чётное, за возможным исключением двух половинок, которые могут встречаться в нечётном количестве: они стоят в начале и в конце. Для подсчёта числа паттернов, встречающихся нечётное количество раз, используем массив degree размера 7. (degree[i]=x – паттерн с i точками встречается x раз)

bool arrangeDominoes(Dominode\* head, unsigned long long int n) {

int degree[7] = {0};

Dominode\* temp = head;

while (temp){

degree[temp->pattern[0]]++;

degree[temp->pattern[1]]++;

temp=temp->next;

}

Пока проверяем условие, сохраняем паттерны, встречающиеся нечётное количество раз.

short int nBuffer, nBuffer2;

int patbuffer;

bool bufCheck=1;

int oddCount = 0;

for (int i = 0; i < 7; i++) {

if (degree[i] % 2 != 0) {

oddCount++;

if(bufCheck){nBuffer=i; bufCheck=0;}

else nBuffer2=i;

}

}

if (oddCount>2){

cout << "Невозможно расставить кости в ряд" << endl;

exit(0);

}

Вывод домино делаем в этой же функции. Если количество паттернов, встречающиеся нечётное количество раз, равно нулю, сохранённые паттерны могут быть любыми; я возьму их из первых двух костей. Проходимся по списку и ищем кость домино с сохранённым паттерном. Проверяем, чтобы кость не содержала оба паттерна. Если паттерн стоит на втором месте, переворачиваем кость. Выводим номера точек обоих половинок на экран, сохраняем вторую половинку (*patbuffer*) и помечаем кость как использованную с помощью тега в классе Dominode.

if (oddCount==0){

nBuffer=head->pattern[0]; nBuffer=head->next->pattern[0];

}

while (head) {

if (head->pattern[0]==nBuffer||head->pattern[1]==nBuffer){

if (head->pattern[0]==nBuffer2||head->pattern[1]==nBuffer2){

cout << "Невозможно расставить кости в ряд" << endl;

exit(0);

}else{

if (head->pattern[1]==nBuffer){ // переворот

int temp=head->pattern[1];

head->pattern[1]=head->pattern[0];

head->pattern[0]=temp;

}

cout<<"Можно уложить кости так: "<<head->pattern[0]<<head->pattern[1];

patbuffer=head->pattern[1];

head->used=1; // помечаем кость как использованную

break;

}

}

head = head->next;

}

Проходимся по списку и ищем неиспользованные кости, которые содержат значение *patbuffer* в любой половинке и не содержат второй сохранённый паттерн. Если *patbuffer* находится во второй половинке, кость переворачиваем. Выводим кость на экран, обновляем *patbuffer* значением из второй половинки, помечаем домино как использованное и двигаемся дальше по списку, пока не останется всего одна неиспользованная кость - в ней будет второй сохранённый паттерн. Если список закончился, идём в начало и продолжаем.

while(n>1){

if((head->pattern[0]==patbuffer||head->pattern[1]==patbuffer)&&(!head->used)&&(head->pattern[0]!=nBuffer2&&head->pattern[1]!=nBuffer2)){

if (head->pattern[1]==patbuffer){

int temp=head->pattern[1];

head->pattern[1]=head->pattern[0];

head->pattern[0]=temp;

}

cout<<", "<<head->pattern[0]<<head->pattern[1];

patbuffer=head->pattern[1];

head->used=1;

n--;

}

if (!head->next){

while (head->prev) head=head->prev;

}

else head = head->next;

}

Наконец, ищем в списке кость со вторым сохранённым паттерном, он должен замыкать список. Если он находится в первой половинке, переворачиваем кость. Выводим значения паттернов. Если достигнут конец списка, идём в начало.

while (head) {

if (head->pattern[0]==nBuffer2||head->pattern[1]==nBuffer2){

if (head->pattern[0]==nBuffer2){

int temp=head->pattern[1];

head->pattern[1]=head->pattern[0];

head->pattern[0]=temp;

}

cout<<", "<<head->pattern[0]<<head->pattern[1];

break;

}

if (!head->next){

while (head->prev) head=head->prev;

}

else head = head->next;

}

Пишем дополнительные функции для проверки введённых пользователем значений. Функция проверки на номер смотрит каждый символ строки, если хотя бы один не является цифрой, выводит 0. Отрицательные числа содержат минус и будут ошибочно выдавать ноль, но для нашей задачи введённые числа могут быть только положительными, так что это не проблема.

bool isNumber(const string& s) {

if (s.empty())

return 0;

for (char c : s)

if (!isdigit(c))

return 0;

return 1;

}

Для проверки значения домино смотрим, является ли строка цифрой от 0 до 6.

bool dominoCheck(string s){//проверка, является ли строка числом от 0 до 6

if(s=="0"||s=="1"||s=="2"||s=="3"||s=="4"||s=="5"||s=="6")

return 1;

else

return 0;

}

Функции готовы, теперь пишем main. Спрашиваем у пользователя количество костей домино; проверяем, чтобы пользователь ввёл число не меньше 2 и не больше 10^15. Для каждой кости домино спрашиваем паттерн (число точек) на каждой половинке домино; проверяем, чтобы введённое число было от 0 до 6 включительно.

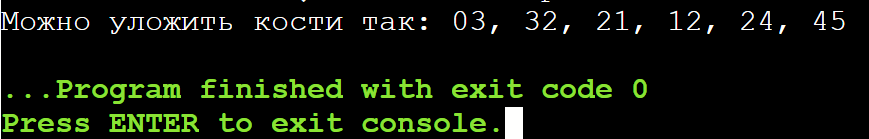
int main(){

Dominode\* head = nullptr;//начало списка cout<<"Введите количество костей домино (от 2 и больше): "; string nstring; cin>>nstring;  
if(!isNumber(nstring)){  
 cout<<"Данные не являются числом или число отрицательное";  
 exit(0);  
}  
if(nstring=="0"||nstring=="1"){  
 cout<<"Для составления ряда нужно две кости или больше";  
 exit(0);  
}  
if(nstring.size()>15){  
 cout<<"Число костей слишком большое";  
 exit(0);  
}  
unsigned long long int n=stoi(nstring);  
for (int i=0; i<n; i++){  
 cout<<"Введите количество точек на первой половине "<<i+1<<"-й кости: ";  
 string temp; cin>>temp;  
 if (!dominoCheck(temp)){  
 cout<<"Количество точек может быть только от 0 до 6 включительно.";  
 exit(0);  
 }  
 int first=stoi(temp);  
 cout<<"Введите количество точек на второй половине "<<i+1<<"-й кости: ";  
 cin>>temp;  
 if (!dominoCheck(temp)){  
 cout<<"Количество точек может быть только от 0 до 6.";  
 exit(0);  
 }  
 int second=stoi(temp);  
 addDomino(head, first, second);  
}  
  
arrangeDominoes(head,n);  
}

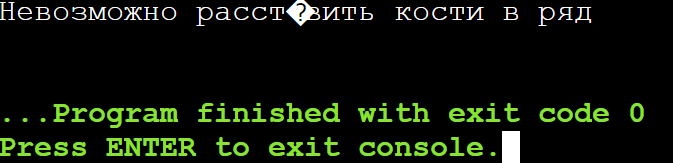
# Тестирование

Для тестирования решил ввести следующие кости домино: 54 23 03 21 12 24. Кости были подобраны так, чтобы они имели ровно два паттерна, встречающихся нечётное число раз.

Результат:



Дальше я выбрал комбинацию, где два паттерна, встречающиеся нечётное число раз, расположены в одном домино, из-за чего собрать домино правильно нельзя. Комбинация: 12 21 54



Код программы

https://github.com/dabmasterars/IKM-CPP-ArsPzd2