# Тест для соискателей вакансии программист

|  |
| --- |
| Уважаемый соискатель, перед Вами тест на вакансию программист в компании Saber Interactive. Индустрия видеоигр, в которой работает компания Saber, предъявляет высокие требования к качеству, эффективности и удобству восприятия программного кода. Поэтому мы предлагаем выполнить Вам 3 несложных задания, чтобы Вы могли продемонстрировать Вашу способность писать такой код.  Просьба не использовать в первых двух задачах библиотечные функции и классы, являющиеся решением этих задач (например stl::bitset). Для третьей задачи можно использовать стандартные функции и контейнеры из STL. |

*Фамилия, имя, отчество выполнившего тест:*

*Козлов Сергей Александрович 27 декабря 1981 года рождения в Минске.*

*Дата выполнения:*

*13 октября 2020 года.*

*Примерное количество времени, затраченного на выполнение теста:*

*5 часов 25 минут.*

## Задачи

1. Напишите функцию, которая принимает на вход знаковое целое число и печатает его двоичное представление, не используя библиотечных классов или функций.

// Test1.cpp : 5 минут

//

#include <iostream>

int main()

{

signed int data = 0;

std::cout << "Enter signed int:\n";

std::cin >> data;

size\_t size\_data = sizeof(signed int) \* 8;

std::cout << "\nBinary value:\n";

for (signed int counter = size\_data - 1; counter >= 0; counter--)

{

if (data & (1 << counter))

{

std::cout << 1;

}

else

{

std::cout << 0;

}

}

}

1. Напишите функцию, удаляющую последовательно дублирующиеся символы в строке:

// Test2.cpp : 15 минут

//

#include <iostream>

#include <string>

void RemoveDups(char\* str);

int main()

{

// пример использования

char data[] = "AAA BBB AAA";

RemoveDups(data);

printf("%s\n", data); // "A B A"

}

void RemoveDups(char\* str)

{

if (str == nullptr)

{

return;

}

std::string result;

char\* previous\_symbol = str;

char\* current\_symbol = str+1;

for (; \*current\_symbol != char(0); current\_symbol++, previous\_symbol++)

{

if (\*previous\_symbol == \*current\_symbol)

{

continue;

}

else

{

result += \*previous\_symbol;

}

}

result += \*previous\_symbol;

size\_t source\_string\_length = current\_symbol - str;

memset(str, 0, source\_string\_length\*sizeof(char));

for (size\_t counter = 0; counter < result.length(); counter++)

{

str[counter] = result.at(counter);

}

for (size\_t counter = result.length(); counter < source\_string\_length; counter++)

{

str[counter] = char(0);

}

}

1. Реализуйте функции сериализации и десериализации двусвязного списка в бинарном формате в файл. Алгоритмическая сложность решения должна быть меньше квадратичной.

// Test3.cpp : 5 часов 10 минут

//

#include <iostream>

// структуру ListNode модифицировать нельзя

struct ListNode {

ListNode\* prev;

ListNode\* next;

ListNode\* rand; // указатель на произвольный элемент данного списка, либо NULL

std::string data;

};

class List {

public:

List() : head(nullptr), tail(nullptr), count(0) {}

List(List&) = delete;

List& operator=(List&) = delete;

~List();

void Serialize(FILE\* file); // сохранение в файл (файл открыт с помощью fopen(path, "wb"))

void Deserialize(FILE\* file); // загрузка из файла (файл открыт с помощью fopen(path, "rb"))

inline ListNode\* GetHead() { return head; }

inline ListNode\* GetTail() { return tail; }

inline int GetCount() { return count; }

inline void SetHead(ListNode\* p) { head = p; }

inline void SetTail(ListNode\* p) { tail = p; }

inline void SetCount(int p) { count = p; }

private:

ListNode\* head;

ListNode\* tail;

int count;

};

int main()

{

List\* local\_list\_source = new List();

if (local\_list\_source == nullptr)

{

return 1;

}

auto CurrentItemNode1 = new ListNode();

if (CurrentItemNode1 != nullptr)

{

CurrentItemNode1->next = nullptr;

CurrentItemNode1->prev = nullptr;

CurrentItemNode1->rand = CurrentItemNode1;

CurrentItemNode1->data = std::string("0001");

local\_list\_source->SetHead(CurrentItemNode1);

local\_list\_source->SetTail(CurrentItemNode1);

}

auto CurrentItemNode2 = new ListNode();

if (CurrentItemNode2 != nullptr)

{

CurrentItemNode1->next = CurrentItemNode2;

CurrentItemNode2->next = nullptr;

CurrentItemNode2->prev = CurrentItemNode1;

CurrentItemNode2->rand = CurrentItemNode1;

CurrentItemNode2->data = std::string("0002");

local\_list\_source->SetTail(CurrentItemNode2);

}

auto CurrentItemNode3 = new ListNode();

if (CurrentItemNode3 != nullptr)

{

CurrentItemNode2->next = CurrentItemNode3;

CurrentItemNode3->next = nullptr;

CurrentItemNode3->prev = CurrentItemNode2;

CurrentItemNode3->rand = CurrentItemNode1;

CurrentItemNode3->data = std::string("0003");

local\_list\_source->SetTail(CurrentItemNode3);

}

auto CurrentItemNode4 = new ListNode();

if (CurrentItemNode4 != nullptr)

{

CurrentItemNode3->next = CurrentItemNode4;

CurrentItemNode4->next = nullptr;

CurrentItemNode4->prev = CurrentItemNode3;

CurrentItemNode4->rand = CurrentItemNode1;

CurrentItemNode4->data = std::string("0004");

local\_list\_source->SetTail(CurrentItemNode4);

}

List\* local\_list\_destination = new List();

if (local\_list\_source == nullptr)

{

delete local\_list\_source;

return 1;

}

FILE\* local\_file = nullptr;

const char local\_path[] = "data\_file.txt";

auto result\_1 = fopen\_s(&local\_file, local\_path, "wb");

if (local\_file == nullptr || result\_1 != 0)

{

delete local\_list\_destination;

delete local\_list\_source;

return 1;

}

local\_list\_source->Serialize(local\_file);

fclose(local\_file);

local\_file = nullptr;

auto result\_2 = fopen\_s(&local\_file, local\_path, "rb");

if (local\_file == nullptr || result\_2 != 0)

{

delete local\_list\_destination;

delete local\_list\_source;

return 1;

}

local\_list\_destination->Deserialize(local\_file);

if (local\_file != nullptr)

{

fclose(local\_file);

}

const char local\_path\_test[] = "data\_file\_test.txt";

local\_file = nullptr;

auto result\_3 = fopen\_s(&local\_file, local\_path\_test, "wb");

if (local\_file == nullptr || result\_2 != 0)

{

delete local\_list\_destination;

delete local\_list\_source;

return 1;

}

local\_list\_destination->Serialize(local\_file);

delete local\_list\_destination;

delete local\_list\_source;

return 0;

}

List::~List()

{

for (ListNode\* current = head; current != tail; )

{

if (current != nullptr)

{

auto current\_save = current;

current = current->next;

delete current\_save;

}

else

{

break;

}

}

if (tail != nullptr)

{

delete tail;

}

}

class ListNodeToStore {

public:

ListNodeToStore() : previous(0), next(0), random(0), link\_forward(nullptr), link\_back(nullptr) {}

unsigned int previous;

unsigned int next;

unsigned int random;

std::string data;

ListNodeToStore\* link\_forward;

ListNodeToStore\* link\_back;

};

class ListToStore {

public:

ListToStore() : head(nullptr), tail(nullptr), count(0) {}

ListToStore(ListToStore&) = delete;

ListToStore& operator=(ListToStore&) = delete;

~ListToStore();

inline ListNodeToStore\* GetHead() { return head; }

inline ListNodeToStore\* GetTail() { return tail; }

inline int Count() { return count; }

inline void SetHead(ListNodeToStore\* p) { head = p; }

inline void SetTail(ListNodeToStore\* p) { tail = p; }

inline void SetCount(int p) { count = p; }

private:

ListNodeToStore\* head;

ListNodeToStore\* tail;

int count;

};

ListToStore::~ListToStore()

{

for (ListNodeToStore\* current = head; current != tail; )

{

if (current != nullptr)

{

auto current\_save = current;

current = current->link\_forward;

delete current\_save;

}

else

{

break;

}

}

if (tail != nullptr)

{

delete tail;

}

}

void List::Serialize(FILE\* file)

{

if (file == nullptr)

{

return;

}

ListNode\* CurrentListNode = head;

if (CurrentListNode != nullptr)

{

unsigned int CurrentNumber = 1;

for (; CurrentListNode != nullptr;)

{

if (CurrentListNode != nullptr)

{

ListNodeToStore CurrentListNodeToStore;

CurrentListNodeToStore.data = CurrentListNode->data;

CurrentListNodeToStore.next = CurrentNumber + 1;

CurrentListNodeToStore.previous = CurrentNumber - 1;

CurrentListNode = CurrentListNode->next;

CurrentNumber++;

ListNode\* CurrentListNodeRandom = head;

unsigned int CurrentNumberRandom = 1;

for (; CurrentListNodeRandom != CurrentListNode;)

{

CurrentListNodeRandom = CurrentListNodeRandom->next;

CurrentNumberRandom++;

}

CurrentListNodeToStore.random = CurrentNumberRandom;

if (CurrentListNode == tail)

{

CurrentListNodeToStore.next = 0;

}

fwrite(&CurrentListNodeToStore.previous, sizeof(unsigned int), 1, file);

fwrite(&CurrentListNodeToStore.next, sizeof(unsigned int), 1, file);

fwrite(&CurrentListNodeToStore.random, sizeof(unsigned int), 1, file);

size\_t CurrentDataLength = CurrentListNodeToStore.data.length();

fwrite(&CurrentDataLength, sizeof(size\_t), 1, file);

fwrite(CurrentListNodeToStore.data.c\_str(), sizeof(char), CurrentDataLength, file);

}

else

{

break;

}

}

}

}

void List::Deserialize(FILE\* file)

{

if (file == nullptr)

{

return;

}

ListToStore ListToStoreData;

for (; feof(file) == 0;)

{

ListNodeToStore CurrentListNodeToStore;

auto SizePrevious = fread(&CurrentListNodeToStore.previous, sizeof(unsigned int), 1, file);

auto SizeNext = fread(&CurrentListNodeToStore.next, sizeof(unsigned int), 1, file);

auto SizeRandom = fread(&CurrentListNodeToStore.random, sizeof(unsigned int), 1, file);

size\_t CurrentDataLength = 0;

auto SizeCurrentDataLength = fread(&CurrentDataLength, sizeof(size\_t), 1, file);

char\* data\_buffer = nullptr;

if (CurrentDataLength != 0)

{

data\_buffer = new char[CurrentDataLength+1];

}

if (data\_buffer == nullptr)

{

break;

}

auto SizeCurrentData = fread(data\_buffer, sizeof(char), CurrentDataLength, file);

data\_buffer[CurrentDataLength] = char(0);

CurrentListNodeToStore.data = std::string(data\_buffer);

delete[] data\_buffer;

CurrentListNodeToStore.link\_forward = nullptr;

CurrentListNodeToStore.link\_back = nullptr;

ListNode\* CurrentListNode = new ListNode();

if (CurrentListNode == nullptr)

{

return;

}

auto CurrentStore = ListToStoreData.GetHead();

if (CurrentStore == nullptr)

{

auto Current = new ListNodeToStore();

if (Current != nullptr)

{

Current->data = CurrentListNodeToStore.data;

Current->next = CurrentListNodeToStore.next;

Current->previous = CurrentListNodeToStore.previous;

Current->random = CurrentListNodeToStore.random;

ListToStoreData.SetHead(Current);

ListToStoreData.SetTail(Current);

}

}

else

{

auto Tail = ListToStoreData.GetTail();

auto Current = new ListNodeToStore();

Tail->link\_forward = Current;

Current->data = CurrentListNodeToStore.data;

Current->link\_forward = nullptr;

Current->link\_back = Tail;

Current->next = CurrentListNodeToStore.next;

Current->previous = CurrentListNodeToStore.previous;

Current->random = CurrentListNodeToStore.random;

ListToStoreData.SetTail(Current);

}

}

if (ListToStoreData.GetHead() != nullptr)

{

for (auto CurrentToStore = ListToStoreData.GetHead(); CurrentToStore != nullptr; CurrentToStore = CurrentToStore->link\_forward)

{

auto CurrentItem = head;

if (CurrentItem == nullptr)

{

CurrentItem = new ListNode();

if (CurrentItem == nullptr)

{

return;

}

CurrentItem->next = CurrentItem->prev = CurrentItem->rand = nullptr;

CurrentItem->data = CurrentToStore->data;

head = CurrentItem;

tail = CurrentItem;

}

else

{

auto Current = new ListNode();

if (Current == nullptr)

{

return;

}

auto Tail = tail;

Tail->next = Current;

Current->data = CurrentToStore->data;

Current->next = nullptr;

Current->prev = Tail;

Current->rand = nullptr;

tail = Current;

}

}

auto Current = head;

for (auto CurrentToStore = ListToStoreData.GetHead(); CurrentToStore != nullptr && Current != nullptr; CurrentToStore = CurrentToStore->link\_forward, Current = Current->next)

{

unsigned int CounterRandom = 1;

auto CurrentItemRandom = head;

if (CounterRandom != 0)

{

for (; CurrentItemRandom != nullptr; CurrentItemRandom = CurrentItemRandom->next)

{

if (CounterRandom < CurrentToStore->random)

{

CounterRandom++;

}

else

{

break;

}

}

Current->rand = CurrentItemRandom;

}

}

}

}