

Федеральное агентство по образованию Российской Федерации
Государственное образовательное учреждение
высшего профессионального образования
Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

Отчёт по лабораторной работе

Структура хранения данных: Мультистек

Выполнил:
студент института ИТММ гр. 381908-4
Галиндо Хавьер Э.

Проверил:
ассистент каф. МОСТ, ИИТММ

Лебедев И.Г.

Нижний Новгород
2020 г.

Содержание

Введение	3
Постановка задачи	4
Руководство пользователя	5
Руководство программиста	6
Описание структуры программы.....	6
Описание структур данных.....	6
Описание алгоритмов	7
Заключение	8
Литература.....	9
Приложения.....	10

Введение

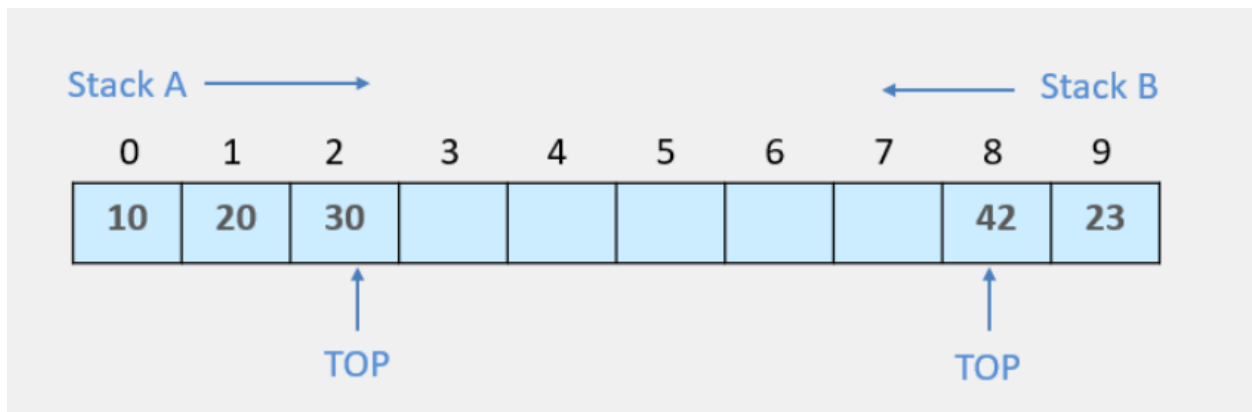
Массив или структура списка вызовов функций и параметров, используемых в современном компьютерном программировании – это «стек» - «последним пришёл — первым вышел».

Когда стек создается с использованием одного массива, мы не можем хранить большой объем данных, поэтому данные ограничения устраняются с использованием более одного стека в одном массиве, или мы можем использовать несколько стеков. Массив разделен на несколько стопок

Этот метод называется множественным стеком.

Предположим, имеется массив `stack [n]`, разделенный на два стека `stack A` и `stack B`, где $n = 10$.

- Stack A расширяется слева направо, то есть от 0-го элемента.
- Stack B расширяется справа налево, то есть с 10-го элемента.
- Общий размер `stack A` и `stack B` никогда не превышает 10.



Постановка задачи

Задача работы — этой программы является правильная реализация структуры данных на основе идеи Multistacks, а также возможность выполнять с ней базовые операции.

1. Разработка и реализация вспомогательного класса стека – Stack
2. Разработка и реализация класса мультистека - Multistack
3. Реализация тестов на базе Google Test
4. Пример использования класса StackLib

Руководство пользователя

Пользователю нужно запустить программу

Далее будет представлен пример использования списков

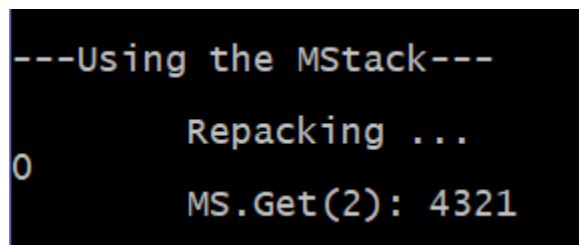
Следующим шагом – будет создан объект класса Stack и Multistack.

значения помещаются в него и элементы одного из стеков отображаются на экране.

На определенном этапе стек, в который мы пытаемся добавить элемент, переполняется, после чего будет произведена переупаковка.

Для повторного выполнения потребуется закрыть и перезапустить программу.

Результаты и описание процессов представлены на рисунке 1.



```
---Using the MStack---  
0      Repacking ...  
      MS.Get(2): 4321
```

Рисунок 1 - результат.

Руководство программиста

Описание структуры программы

В программе содержатся следующие модули:

- Модуль StackLib: статическая библиотека. Содержит файл Stack.h, в котором описан интерфейс и реализация шаблонного класса *TStack*.
- Модуль MultiStackLib – содержит файл MultiStack.h, в котором реализованы классы MultiStack и TNewStack
- Модуль MultiStack – содержит в себе файл реализации примера использования класса Stack.
- Модуль test_MultiStack – содержит в себе файл MStackTest.cpp, в котором находится набор тестов, для проверки работоспособности класса TMSack.

Описание структур данных

- Класс TNewStack

Класс TNewStack является шаблонным.

1) Элементы класса, объявленные со спецификатором public:

- TNewStack (int _size = 0, T* _mas = 0): конструктор
- TNewStack(TNewStack <T> &NS): конструктор копирования
- int GetFreeMemory(): получение свободной памяти (кол-во свободных позиций)
- int GetSize(): получение размера
- int GetTop(): получение первого элемента
- void SetMas(int _size, T* _mas): задать массив размера _size, содержащий элементы _mas.
- void Put(T _A): положить элемент _A
- T Get(): получить значение элемента
- ~ TNewStack(): деструктор.

- Класс TMSack

1) Элементы класса, объявленные со спецификатором protected:

- int size : размер
- T* mas: массив элементов
- int n: количество стеков
- TNewStack<T>** newS: массив указателей на начало каждого стека в мультистеке
- int GetFreeMemory(): возвращает число свободных элементов
- void Repack(int k): перепакровка (память k-ого стека увеличивается)

- 2) Элементы класса, объявленные со спецификатором public:
- TMSStack(int _n, int _size): конструктор
 - TMSStack(TMSStack<T> &A): конструктор копирования
 - int GetSize(): возвращает размер
 - T Get(int _n): возвращает значение из n-ого стека
 - void Set(int _n, T p): положить значение в _n-ый стек
 - bool IsFull(int _n): проверка стека на полноту _n-ого стека
 - bool IsEmpty(int _n): проверка стека на пустоту _n-ого стека
 - ~TMSStack(): деструктор

Описание алгоритмов

Добавление элемента в стек – это процедура направления элемента в стек с указанным номером, позволяет поместить его на вершину стека, на которую указывает поле top. После добавления элемента в стек, значение поля top увеличивается на 1.

Удаление элемента из стека: В процедуре удаления элемента из стека с указанным номером убирается из первой непустой ячейки. На эту непустую ячейку указывает поле top со значением, уменьшенным на 0.

.

Заключение

Целью данной работы являлось – рассмотрение структуры хранения данных: Мультистек и наглядное препарирование работы стека

Как видно результатом работы стала – реализация, хранение и операции с мультистеком.

А также реализована практическая значимость данной работы – это создание тестов для проверки работоспособности класса мультистека на базе GoogleTest. А также приведен пример работы стека.

Литература

1. Круз Р.Л. Структуры данных и проектирование программ. – М.: Бином. Лаборатория знаний, 2014.
2. Топп У., Форд У. Структуры данных в C++.- М.: Бином, 1999
3. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ.- М.: МЦМО, 1999.
4. Ахо Альфред В., Хопкрофт Джон, Ульман Джеффри Д. Структуры данных и алгоритмы- М.: Издательский дом Вильямс, 2000.
5. Гергель В.П. и др. Методы программирования. Учебное пособие. Н.Новгород: ННГУ, 2016.
6. Столлинкс, В. Структурная организация и архитектура компьютерных систем, 5-е изд.: Пер. с англ. — М.: Издательский дом «Вильямс», 2002. — 896 с.: ил. — Парал. тит. англ.
7. Страуструп Б. Язык программирования C++. – М.: Бином, 2001

Приложения

Multistack.cpp

```
#include "Multistack.h"
#include <math.h>
#include "Stack.h"

using namespace std;

template<class T>
inline void TMultiStack<T>::StackRelocation(int index)
{
    int freeSize = 0;
    for (int i = 0; i < stackCount; i++)
    {
        freeSize += stacks[i].GetSize() - stacks[i].GetCount();
    }

    if (freeSize == 0)
    {
        throw logic_error("Error");
    }

    int count = int(floor(double(freeSize) / stackCount));
    int* sizes = new int[this->stackCount];

    for (int i = 0; i < (stackCount - 1); i++)
    {
        sizes[i] = stacks[i].GetCount() + count;
    }
    int c = stacks[stackCount - 1].GetCount();
    sizes[this->stackCount - 1] = c + (freeSize - (count * (this->stackCount - 1)));

    T** newData = new T * [stackCount];
    int k = 0;

    for (int i = 0; i < stackCount; i++)
    {
        newData[i] = &(data[k]);
        k = k + sizes[i];
    }

    for (int i = 0; i < stackCount; i++)
    {
        if (newData[i] == oldData[i])
        {
            stacks[i].SetData(newData[i], sizes[i], stacks[i].GetCount());
        }
        else if (newData[i] < oldData[i])
        {
            for (int j = 0; j < stacks[i].GetCount(); j++)
            {
                newData[i][j] = oldData[i][j];
            }
            stacks[i].SetData(newData[i], sizes[i], stacks[i].GetCount());
        }
    }
}
```

```

else if (newData[i] > oldData[i])
{
    int k = i;
    for (; k < stackCount; k++)
    {
        if (newData[k] > oldData[k])
        {
            continue;
        }
        else
        {
            break;
        }
    }
    k--;

    for (int s = k; s <= i; s--)
    {
        for (int j = stacks[s].GetCount() - 1; j >= 0; j--)
        {
            newData[s][j] = oldData[s][j];
        }
        stacks[s].SetData(newData[s], sizes[s], stacks[s].GetCount());
    }
}

T** buf = oldData;
oldData = newData;
delete[] buf;
delete[] sizes;
}

template<class T>
inline TMultiStack<T>::TMultiStack(int size, int _stackCount)
{
    if ((size > 0) && (_stackCount > 0))
    {
        this->length = size;
        this->stackCount = _stackCount;

        data = new T[length];
        for (int i = 0; i < length; i++)
        {
            data[i] = 0;
        }

        int count = int(floor(double(size) / stackCount));
        int* sizes = new int[this->stackCount];

        for (int i = 0; i < (stackCount - 1); i++)
        {
            sizes[i] = count;
        }

        sizes[stackCount - 1] = size - (count * (stackCount - 1));
        oldData = new T * [stackCount];
    }
}

```

```

        this->stacks = new TStack<T>[stackCount];
        int k = 0;

        for (int i = 0; i < stackCount; i++)
        {
            this->stacks[i].SetData(&(data[k]), sizes[i], 0);
            this->oldData[i] = &(data[k]);
            k = k + sizes[i];
        }
    }
    else
    {
        throw logic_error("Error");
    }
}

template <class T>
TMultiStack<T>::TMultiStack(TMultiStack<T>& _v)
{
    length = _v.length;
    stackCount = _v.stackCount;

    data = new T[length];
    for (int i = 0; i < length; i++)
    {
        data[i] = _v.data[i];
    }

    stacks = new TStack<T>[stackCount];
    for (int i = 0; i < stackCount; i++)
    {
        stacks[i] = _v.stacks[i];
    }
    oldData = _v.oldData;
}

template <class T>
TMultiStack<T>::~~TMultiStack()
{
    length = 0;
    stackCount = 0;

    if (data == NULL)
    {
        delete[] data;
        data = NULL;
    }

    if (stacks == NULL)
    {
        delete[] stacks;
        stacks = NULL;
    }
}

template <class T>
TMultiStack<T>& TMultiStack<T>::operator =(TMultiStack<T>& _v)
{

```

```

        if (this == &_v)
        {
            return *this;
        }

        this->length = _v.length;
        if (data != NULL)
        {
            delete[] data;
        }
        if (stacks != NULL)
        {
            delete[] stacks;
        }

        data = new T[length];
        for (int i = 0; i < length; i++)
        {
            data[i] = _v.data[i];
        }

        stacks = new TStack<T>[stackCount];
        for (int i = 0; i < stackCount; i++)
        {
            stacks[i] = _v.stacks[i];
        }

        return *this;
    }

template<class T>
inline void TMultiStack<T>::Push(T d, int i)
{
    if (i < 0 || i >= stackCount)
    {
        throw logic_error("Error");
    }

    if (stacks[i].IsFull())
    {
        StackRelocation(i);
    }

    stacks[i].Push(d);
}

template<class T>
inline T TMultiStack<T>::Get(int i)
{
    if (i < 0 || i > stackCount)
    {
        throw logic_error("Error");
    }

    if (stacks[i].IsEmpty())
    {
        throw logic_error("Error memory");
    }
}

```

```

        T d = stacks[i].Get();
        return d;
    }

template<class T>
inline bool TMultiStack<T>::IsEmpty(int i) const
{
    if (i < 0 || i > stackCount)
    {
        throw logic_error("Error");
    }

    return stacks[i].IsEmpty();
}

template<class T>
inline bool TMultiStack<T>::IsFull(int i) const
{
    if (i < 0 || i > stackCount)
    {
        throw logic_error("Error");
    }

    return stacks[i].IsFull();
}

template <class T>
int TMultiStack<T>::GetSize()
{
    return length;
}

template<class T>
inline void TMultiStack<T>::Resize(int size, int stackCount)
{
    stacks[stackCount].Resize(size);
}

```