Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Информатика и системы управления
КАФЕДРА Программное обеспечение ЭВМ и информационные технологии

ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №4
РАБОТА СО СТЕКОМ

Название предмета: Типы и структуры данных

Студент: Варламова Екатерина Алексеевна

Группа: ИУ7-31Б

2020г.

I. Описание условия задачи

Создать программу работы со стеком, выполняющую операции добавления, удаления элементов и вывод текущего состояния стека. Реализовать стек: а) массивом; б) списком. Оценить преимущества и недостатки каждой реализации. При реализации стека списком в вывод текущего состояния стека добавить просмотр адресов элементов стека и создать свой список или массив свободных областей (адресов освобождаемых элементов) с выводом его на экран. С помощью стека проверить правильность расстановки скобок трех типов (круглых, квадратных и фигурных) в выражении.

Техническое задание

1. Описание исходных данных

Программа ожидает:

- название файла с выражением (в аргументах командной строки)
- максимальный размер стека
- номер действия (подробнее о том, какой номер соответствует определённому действию, будет выведено при запуске программы).

Формат ввода и ограничения:

1. При добавлении элемента в стек требуется ввести элемент, который имеет символьный тип (символы из однобайтовых кодировок).
2. При вводе максимального размера стека ожидается целое неотрицательное число.

2. Описание результата программы

Результатом работы программы могут являться (в зависимости от введённого действия):

1. Извлечённый элемент стека.
2. Текущее состояние стека (если стек реализован списком, помимо самих элементов будут также выведены их адреса и список адресов всех извлечённых элементов).

3. Вывод о том, правильно ли расположены скобки в выражении. При этом при добавлении в стек и удалении элементов (скобок) из стека будет выводиться текущее состояние стека (текущее состояние определяется так же, как в предыдущем пункте).
4. Статистика по времени выполнения и объёму памяти при обработке стеков, реализованных списком и динамическим массивом.

Формат вывода:

1. Для извлечённого элемента:
   extracted value: <element>
2. Для текущего состояния стека, если он реализован:
   - массивом (в стеке 5 элементов):

   <element> <element> <element> <element> <element>
   - списком (в стеке 5 элементов, 2 были извлечены):

   element: <element> located: <address>

   element: <element> located: <address>

   element: <element> located: <address>

   element: <element> located: <address>

   element: <element> located: <address>

   free: <address> <address>

   Пример:

   element: e located: 0x7fab19604090

   element: d located: 0x7fab194058a0

   element: c located: 0x7fab19405890

   element: b located: 0x7fab19604080

   element: a located: 0x7fab195040b0

   free: 0x7fab19604090 0x7fab194058a0
3. Для вывода о расположении скобок:
   - correct –правильная расстановка скобок;
   - incorrect – одинаковое количество открытых и закрытых скобок, но какая-то скобка закрыта неправильно. Пример: ({];

- incorrect: stack overflow – при чтении выражения переполнился стек;
- incorrect: memory error – операционная система отказала в выделении памяти под узел списка или под область вектора;
- incorrect: not enough brackets – количество открытых и закрытых скобок в выражении не совпадает

4. Статистика выводится в виде таблицы.

3. *Описание задачи, реализуемой программой*

Программа показывает работу стеков, реализованных на основе динамического массива и списка. Можно добавлять элемент в стек, извлекать из него элемент и просматривать текущее состояние стека.

Кроме того, программа может считывать выражение из файла (имя должно быть указано в аргументах командной строки) и определять, правильно ли расставлены скобки в выражении. Эта функция работает с обоими стеками: реализованного на основе списка и на основе массива. Сравнить эффективность стеков можно в разделе статистика.

4. Способ обращения к программе

Способ обращения к программе - консольный. Дальнейшие инструкции будут выведены после запуска.

5. Описание возможных аварийных ситуаций и ошибок пользователя

- ошибки ввода максимального размера стека и элементов стека;
- ошибка ввода действия;
- ошибка в названии файла или его отсутствие (физическое и/или в аргументах командной строки);
- отказ операционной системы выделить запрашиваемую память;

Во всех указанных случаях программа сообщит об ошибке.

II. Описание внутренних структур данных

В программе есть 2 основных структуры данных:

1. Стек на основе списка

Описание на языке С стека выглядит таким образом:

```
typedef struct node
{
    struct node *next;
    elem_t value;
} node_t;

typedef struct
{
    node_t *head;
    size_t count;
    size_t max_count;
    size_t size_of_element;
    vector_t adrs;
} stack_on_list_t;
```

Данные хранятся в форме односвязного линейного списка. Структура стека состоит из: указателя head на голову списка (вершина стека), текущего количества элементов и максимального count и max_count соответственно, размера элементов size_of_element. Дополнительно ввиду условия задачи хранится вектор adrs адресов освобождённых элементов списка. У данной структуры есть следующие особенности:

- данные в памяти располагаются в произвольном порядке;
- при добавлении элемента каждый раз запрашивается память под элемент;
- при удалении элемента каждый раз память освобождается;

2. Стек на основе динамического массива

Описание на языке С стека выглядит таким образом:

```
typedef struct {
    vector_t data;
    elem_t *head;
    size_t count;
    size_t max_count;
    size_t size_of_element;
} stack_on_array_t;
```

Здесь сами данные хранятся в динамически расширяемом векторе data, указатель head указывает на последний элемент в данных, count и max_count – текущее количество элементов и максимальное соответственно, size_of_element – размер элементов. У данной структуры есть следующие особенности:

- данные в памяти хранятся последовательно;
- выделение памяти под новые элементы происходит блоками (при достижении максимума размер выделенной памяти умножается на 2, а элементы копируются из старой области в новую);
- при удалении сдвигается только указатель на последний элемент;

Вспомогательной структурой является вектор, который реализует хранение данных. Его описание на языке С выглядит следующим образом:

```
typedef struct
{
    void *front;
    size_t size_of_element;
    size_t size;
    size_t alloc_size;
} vector_t;
```

III. Описание алгоритма

Определение правильности расстановки скобок в выражении реализовано с использованием возможностей стека:

1. Считывается очередной символ строки sym
   - Если sym один из {, [, (, то он добавляется в стек;
   - Если sym один из },],), то вынимается верхний элемент из стека. Если стек пуст, то скобки расставлены неверно (недостаточно скобок). Если извлеченный элемент не является обратным к sym, то скобки расставлены неверно.
2. Если все символы считаны, а стек не пуст, то скобки расставлены неверно (недостаточно скобок), иначе скобки расставлены верно.

Оценка по времени

Для обработки стека (под обработкой имеется в виду определение правильности расстановки скобок в выражении) были использованы стеки, реализованные на основе списка и на основе вектора. Исходя из особенностей каждой структуры, мы можем сделать вывод о том, что по времени при небольших размерах элементов динамически расширяемый вектор будет быстрее по сравнению со списком, поскольку в динамическом векторе память запрашивается «с запасом», а в списке каждый раз при добавлении (при извлечении в векторе просто перемещается указатель, а в списке освобождается память). Однако при больших размерах элементов копирование элементов при расширении вектора значительно повлияет на скорость работы, так что список будет более эффективен.

Оценка по памяти

Если не учитывать расходы по памяти на вспомогательные переменные, хранящиеся в структурах стеков (например, максимальное количество элементов, текущее количество элементов), то расходы по памяти будут примерно следующими (размер элемента size, количество элементов n, размер указателя u):

На стек на основе вектора:

- указатель на данные (на вершину стека): u
- данные (максимум): n * 2 * size

На стек на основе списка:

- указатель на узел: u
- данные: n * (size + u) (без учёта выравнивания)

Получаем, что в худшем случае для вектора (память выделена под много пустых элементов) выигрыш в занимаемой памяти будет зависеть от размера элемента: если элемент занимает много памяти (2 * size > size + u), то выгоднее использовать список. Иначе, если размер элементов небольшой (около размера указателей), то в любом случае выгоднее использовать вектор.

Итог

Если размер элементов небольшой, то выгоднее использовать стек на основе вектора, если же размер элементов большой, то ввиду «гибкости» списка, стек на основе списка окажется эффективнее как по памяти, так и по времени.

Проверка выводов экспериментально

Возьмём стек на 600 000 элементов. При вышеописанной реализации вектора данная цифра является почти худшим случаем (выделена память под много пустых элементов). При этом используем упаковку структуры списка для того, чтобы было проще отследить статистику по памяти (отметим, что на время это повлияет незначительно).

размер элемента 1 байт, статистика по обработке стеков выглядит так:

stack	time	memory
on linked list	127363.10	5400032
on vector	62742.20	1048640

размер элемента 8 байт, статистика по обработке стеков выглядит так:

stack	time	memory
on linked list	130430.50	9600032
on vector	66564.50	8388672

размер элемента 11 байт, статистика по обработке стеков выглядит так:

stack	time	memory
on linked list	136515.10	11400032
on vector	66271.90	11534400

размер элемента 16 байт, статистика по обработке стеков выглядит так:

stack	time	memory
on linked list	137498.00	14400032
on vector	69736.20	16777280

размер элемента 25 байт, статистика по обработке стеков выглядит так:

stack	time	memory
on linked list	141762.70	19800032
on vector	71901.20	26214464

размер элемента 80 байт, статистика по обработке стеков выглядит так:

stack	time	memory
on linked list	156527.50	52800032
on vector	96201.40	83886144

размер элемента 200 байт, статистика по обработке стеков выглядит так:

stack	time	memory
on linked list	194107.70	124800032
on vector	170409.10	209715264

размер элемента 400 байт, статистика по обработке стеков выглядит так:

stack	time	memory
on linked list	269326.80	244800032
on vector	349096.50	419430464

размер элемента 1000 байт, статистика по обработке стеков выглядит так:

stack	time	memory
on linked list	536600.70	604800032
on vector	920450.90	1048576064

размер элемента 2000 байт, статистика по обработке стеков выглядит так:

stack	time	memory
on linked list	977658.40	1204800032
on vector	1787016.40	2097152064

размер элемента 5000 байт, статистика по обработке стеков выглядит так:

stack	time	memory
on linked list	2269342.20	3004800032
on vector	5177964.60	5242880064

Итак, в худшем случае для вектора и при упаковке структуры списка:

- при маленьком размере элементов (1, 8) стек на основе вектора эффективнее стека на основе списка как по времени, так и по памяти.
- при размере в 11 байт стеки показывают примерно одинаковую эффективность по памяти; по времени стек на основе вектора всё ещё выигрывает.
- при размерах элемента от 11 байт до 400 стек на основе вектора выигрывает по времени, но проигрывает по памяти.
- при размере элемента в 400 байт стек на основе вектора начинает проигрывать не только по памяти, но и по времени.
- при больших размерах (> 400) стек на основе списка значительно эффективнее и по памяти, и по времени.

Возьмём теперь стек на 1 000 000 элементов. При вышеописанной реализации вектора данная цифра является почти лучшим случаем (выделенная память почти полностью используется). При этом используем упаковку структуры списка для того, чтобы было проще отследить статистику по памяти (отметим, что на время это повлияет незначительно).

размер элемента 1 байт, статистика по обработке стеков выглядит так:

stack	time	memory
on linked list	212458.80	9000032
on vector	103564.00	1048640

размер элемента 8 байт, статистика по обработке стеков выглядит так:

stack	time	memory
on linked list	215698.00	16000032
on vector	108698.10	8388672

размер элемента 16 байт, статистика по обработке стеков выглядит так:

stack	time	memory
on linked list	222951.00	24000032
on vector	112915.60	16777280

размер элемента 64 байт, статистика по обработке стеков выглядит так:

stack	time	memory
on linked list	250671.50	72000032
on vector	133314.90	67108928

размер элемента 80 байт, статистика по обработке стеков выглядит так:

stack	time	memory
on linked list	260004.10	88000032
on vector	138944.80	83886144

размер элемента 170 байт, статистика по обработке стеков выглядит так:

stack	time	memory
on linked list	311467.20	178000032
on vector	223426.00	178257984

размер элемента 200 байт, статистика по обработке стеков выглядит так:

stack	time	memory
on linked list	328279.90	208000032
on vector	248312.50	209715264

размер элемента 400 байт, статистика по обработке стеков выглядит так:

stack	time	memory
on linked list	437960.40	408000032
on vector	449106.10	419430464

размер элемента 1000 байт, статистика по обработке стеков выглядит так:

stack	time	memory
on linked list	873659.80	1008000032
on vector	1114939.50	1048576064

размер элемента 2000 байт, статистика по обработке стеков выглядит так:

stack	time	memory
on linked list	1634306.40	2008000032
on vector	2224671.30	2097152064

размер элемента 5000 байт, статистика по обработке стеков выглядит так:

stack	time	memory
on linked list	1634306.40	2008000032
on vector	2224671.30	2097152064

Итак, в лучшем случае для вектора и при упаковке структуры списка:

- при размере элементов от 1 до 170 байт стек на основе вектора эффективнее стека на основе списка как по времени, так и по памяти.
- при размере в 170 байт стеки показывают примерно одинаковую эффективность по памяти; по времени стек на основе вектора всё ещё выигрывает.
- при размерах элемента от 170 байт до 400 стек на основе вектора выигрывает по времени, но проигрывает по памяти (незначительно).
- при размере элемента в 400 байт стек на основе вектора начинает проигрывать не только по памяти, но и по времени.
- при больших размерах (> 400) стек на основе списка эффективен и по памяти (незначительно), и по времени (значительно).

Итог

Мы экспериментально подтвердили сделанный ранее вывод о том, что при небольших размерах элементов выгоднее использовать стек на основе вектора, а при больших – стек на основе списка.

IV. Тестирование

1. Позитивные тесты

Входные данные	Описание теста	Результат
[{[([{}])]}()]	Проверка скобок на основе вектора	[{ [[{ [([{ [[([{ [{ [([{ [[([{ [([{ [[{ [{ [[([[correct
[{[]}()]	Проверка скобок на основе списка	element: [located: 0x100706300 free: element: { located: 0x100704e30 element: [located: 0x100706300 free: element: [located: 0x100705c20 element: { located: 0x100704e30 element: [located: 0x100706300 free: element: { located: 0x100704e30 element: [located: 0x100706300 free: 0x100705c20 element: [located: 0x100706300 free: 0x100705c20 0x100704e30 element: (located: 0x100704e30 element: [located: 0x100706300 free: 0x100705c20 0x100704e30 element: [located: 0x100706300 free: 0x100705c20 0x100704e30 0x100704e30 free: 0x100705c20 0x100704e30 0x100704e30 0x100706300 correct

[{[]}()	Проверка скобок на любом стеке на неверное количество скобок (не хватает закрывающей)	incorrect: not enough brackets
[{[]}()]]	Проверка скобок на любом стеке на неверное количество скобок (не хватает открывающей)	incorrect: not enough brackets
[{[]}())	Проверка скобок на любом стеке на неверное закрытие одной из скобок	incorrect
[{[]}()]	Проверка скобок на любом стеке на переполнение (максимальный размер стека = 2)	incorrect: stack overflow

V. Выводы по проделанной работе

В ходе работы был проведён сравнительный анализ реализаций стеков (на основе односвязного списка и на основе вектора), в результате которого установилось, что при небольших размерах элементов выгоднее использовать стек на основе вектора, а при больших – стек на основе списка.

VI. Ответы на вопросы

1. Что такое стек?

Стек – это последовательный список с переменной длиной, в котором включение и исключение элементов происходит только с одной стороны – с его вершины. Стек функционирует по принципу: последним пришел – первым ушел, Last In – First Out (LIFO).

2. Каким образом и сколько памяти выделяется под хранение стека при различной его реализации?

См. “Описание алгоритма” пункт “Оценка по памяти”

3. Каким образом освобождается память при удалении элемента стека при различной реализации стека?

При извлечении элемента в векторе просто перемещается указатель, а в списке освобождается память под этим элементом. Память под вектором освобождается тогда, когда стек больше не нужен.

4. Что происходит с элементами стека при его просмотре?

При классической реализации стека просмотр стека предполагает удаление всех элементов, поскольку по определению в стеке доступен только верхний элемент. В нашей реализации просмотр осуществляется без удаления.

5. Каким образом эффективнее реализовывать стек? От чего это зависит?

Если элементы будут небольшими, то эффективнее использовать стек на основе вектора, поскольку:

- копирование элементов при расширении вектора займёт меньше времени, чем создание каждого элемента списка;
- дополнительная память при реализации вектором в среднем будет примерно равна дополнительной памяти при реализации списком.

Если элементы стека будут большими, то эффективнее использовать стек на основе списка как по памяти, так и по времени.