Лабораторная работа №4. "Работа со стеком"

Студент Ивахненко Дмитрий - ИУ7-36Б

Описание условия задачи

Создать программу работы со стеком, выполняющую операции:

1. добавление элемента,
2. удаления элемента
3. вывод текущего состояния стека.

Реализовать стек:

1. массивом
2. списком.

Все стандартные операции со стеком должны быть оформлены подпрограммами. 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.

Проверить правильность расстановки скобок трех типов (круглых, квадратных и фигурных) в выражении.

Техническое задание

Входные данные:

1. Номер команды - целое число в диапазоне от **0** **до** **16** включительно.
2. **Командно-зависимые данные:**
   - Добавить элемент в стек
     - элемент, добавляемый в стек, - символ.
   - Проверка правильности расстановки скобок в выражении
     - последовательность любых символов, которую необходимо проверить **(Примечание: последовательность, не содержащая ни одной скобки, считается корректной)**

Выходные данные:

1. Просмотреть последний элемент стека (или извлечь последний элемент)
   - значение последнего элемент стека
2. Просмотреть содержимое стека
   - значение всех элементов стека
3. Просмотреть список высвобожденных адресов
   - значение всех высвобожденных адресов
4. Проверить расстановку скобок в выражении
   - Информация об итоге проверки выражения на правильность расстановки скобок в формате:
     - **::Скобки расставлены верно::**
     - **::Скобки расставлены не верно::**
5. Провести сравнительный анализ
   - результат сравнительного анализа стеков, представленный в виде таблицы

Команды программы

`:::Работа со стеком:::`

`::Стек на односвязном списке::`

1. Добавить элемент
2. Просмотреть последний элемент
3. Извлечь последний элемент
4. Удалить последний элемент (без вывода)
5. Просмотреть сожержимое стека
6. Изменить размер стека

`::Стек на векторе::`

7. Добавить элемент
8. Просмотреть последний элемент
9. Извлечь последний элемент
10. Удалить последний элемент (без вывода)
11. Просмотреть содержимое стека
12. Изменить размер стека

`:::Задача на правильную скобочную последовательность:::`

13. Проверить расстановку скобок в строке (Стек-Список)
14. Проверить расстановку скобок в строке (Стек-Вектор)

`:::Дополнительные функции:::`

15. Просмотреть список высвобожденных адресов
16. Провести сравнительный анализ (Стек-Вектор и Стек-Список)

Обращение к программе:

Запускается из терминала при помощи комманды `./bin/app`.

Аварийные ситуации:

1. Некорректный ввод номера команды.

   На входе: число, большее, чем максмальный индекс команды или меньшее, чем минимальный.

   На выходе: `[Ошибка] ::Некорректный ввод пункта меню::`

2. Некорректный ввод элемента, добавляемого в стек.

   На входе: несколько символов / пустой ввод

   На выходе: `[Ошибка] ::Некорректный ввод::`

3. Указание некорректного размера стека при его изменении

   На входе: символ, не являющийся натуральным числом.

   На выходе: `[Ошибка] ::Некорректный размер стека::`

4. Изменение размера непустого стека

На входе: попытка изменить размер стека, в котором находится хотя бы один элемент

На выходе: `[Ошибка] ::Изменение размера возможно только для пустого стека::`

5. Пустая последовательность

   На входе: пустой ввод при проверке расстановки скобок в строке

   На выходе: `[Ошибка] ::Последовательность не может быть пустой::`

6. Переполнение стека

   На входе: попытка добавить элемент в полный стек

   На выходе: `[Ошибка] ::Стек переполнен (Stack Overflow)::`

7. Удаление элемента из пустого стека

   На входе: попытка удалить элемент из стека, ни содержащего ни одного элемента

   На выходе: `[Ошибка] ::В стеке нет ни одного элемента (Stack Underflow)::`

Структуры данных

Для хранения элементов стека использую псевдоним `stacktype_t` типа данных `char`

```
typedef char stacktype_t;
```

Для хранения стека на основе вектора использую структуру `VStack`

```
typedef struct
{
    stacktype_t *ptr;   // указатель на последний элемент
    size_t size;        // размер стека
    size_t capacity;    // текущая емкость вектора
    stacktype_t *buf;   // данные
} VStack;
```

Для хранения стека на основе вектора использую структуру `LLStack`

```
typedef struct {
    Node *ptr;              // указатель на последний элемент
    size_t size;            // размер стека
    size_t curr_size;       // текущий размер стека
    Vector *freelist_ptr;   // вектор высвобожденных адресов
} LLStack;
```

Для хранения узла использую структуру `Node`

```
typedef struct Node Node;
struct Node {
    stacktype_t data;        // данные текущего узла
    Node *next;             // указатель на следующий узел
};
```

Для хранения высвобожденных адресов использую `Vector`

```
typedef void* vtype_t;
typedef struct {
    vtype_t *data;          // данные
    size_t length;          // текущая длина вектора
    size_t capacity;        // емкость вектора
} Vector;
```

Описание основных функций

- `VStack *vstack_create(size_t size)`
- `LLStack *llstack_create(size_t size)`
    - принимает размер стека
    - создает стек указанного размера над вектором или односвязным списком соответственно
    - возвращает созданный стек
- `int vstack_push(VStack *vstack, stacktype_t data)`
- `int llstack_push(LLStack *llstack, stacktype_t data)`
    - принимает значение элемента
    - добавляет элемент с указанным значением в стек над вектором или односвязным списком соответственно
- `stacktype_t vstack_peek(VStack *vstack, int *rc)`
- `stacktype_t llstack_peek(LLStack *llstack , int *rc)`
    - принмимает стек
    - просматривает верхний элемент в стеке над вектором или односвязным списком соответственно
    - возвращает значение элемента
- `stacktype_t vstack_pop(VStack *vstack, int *rc)`
- `stacktype_t llstack_pop(LLStack *llstack, int *rc)`
    - принмает стек
    - извлекает верхний элемент в стеке над вектором или односвязным списком соответственно
    - возвращает значение элемента
- `int vstack_display(VStack *vstack)`
- `int llstack_display(LLStack *llstack)`
    - принимает стек

    - печатает содержимое стека над вектором или односвязным списком соответственно

- `void vstack_delete(VStack *vstack)`
- `void llstack_delete(LLStack *llstack)void llstack_delete(LLStack *llstack)void llstack_delete(LLStack *llstack)`
    - принимает стек
    - удаляет стек над вектором или односвязным списком соответственно

Алгоритм

1. На экран пользователю выводится меню
2. Пользователь вводит номер команды
3. Выполняется действие согласно номеру команды

Стек-вектор

- Добавить элемент в стек
    - если вектор заполнен полностью, он расширяется вдвое
    - новый элемент помещается в конец вектора
- Извлечь элемент из стека
    - если вектор пустой, то выводится сообщение об ошибке
    - если в векторе есть элементы, то
        - выводится последний элемент стека
        - в дескрипторе длина вектора уменьшается на единицу
- Просмотреть содержимое стека
    - проходимся по всем элементам вектора и печатаем их

Стек-список

- Добавить элемент в стек
    - создается новый узел списка, указывающий на "голову" списка
    - указатель на "голову" списка перенаправляется на новый узел
- Вытыщить элемент из стека
    - если указатель на "голову" является нулевым указателем, то выводится сообщение об ошибке
    - если указатель на "голову" не является нулевым указателем, то
        - выводится значение "головы"
        - сохраняется адрес "головы"
        - указатель на "голову" перенаправляется на следующий (второй) элемент списка
        - ранее сохраненный адрес памяти высвобождается
- Просмотреть содержимое стека
    - проходимся по всем элементам вектора и печатаем их и их адреса в памяти

Алгоритм проведения сравнительного анализа стеков

- Перебираются различные длины стека
- Для каждого измерения создаются новые стеки
- Фиксируется время, за которое стеки полностью заполнятся
- Фиксируется время, за которое стеки полностью освободятся
- Выводится таблица с численными и объёмными характеристиками

Алгоритм для решения задачи

- Счиывается строка, которую необходимо проверить.
- Производится анализ строки
    - Если встречается открывающая скобка, она добавляется в стек
    - Если встречается закрывающая скобка и стек не пустой, то извлекается последний элемент стека (скобка), а если пустой - `Скобки расставлены не верно`.
    - Если скобка, которую извлекли из стека совпадает по типу с исходной, анализ продолжается, а иначе - `Скобки расставлены не верно`.
- Если после заверешения анализа строки стек пуст, то `Скобки расставлены верно`, а иначе `Скобки расставлены не верно`.

Тесты

Меню

```
======:::МЕНЮ:::======
:::Работа со стеком:::
::Стек на односвязном списке::
        1. Добавить элемент
        2. Просмотреть последний элемент
        3. Извлечь последний элемент
        4. Удалить последний элемент (без вывода)
        5. Просмотреть сожержимое стека
        6. Изменить размер стека
::Стек на векторе::
        7. Добавить элемент
        8. Просмотреть последний элемент
        9. Извлечь последний элемент
        10. Удалить последний элемент (без вывода)
        11. Просмотреть содержимое стека
        12. Изменить размер стека
:::Задача на правильную скобочную последовательность:::
        13. Проверить расстановку скобок в строке (Стек-Список)
        14. Проверить расстановку скобок в строке (Стек-Вектор)
:::Дополнительные функции:::
        15. Просмотреть список высвобожденных адресов
        16. Провести сравнительный анализ (Стек-Ветор и Стек-Список)

        0. Выход

Введите число в диапазоне [0..16] и нажмите <Enter>:
```

# Описание теста	Вввод	Вывод
Неправильный номер команды	-1 `<Enter>`	Сообщение об ошибке
Неправильный тип данных стека	7 `<Enter>` w `<Enter>`	Сообщение об ошибке
Попытка удаления элемента в пустом стеке	3 `<Enter>`	Сообщение об ошибке

# Описание теста	Вввод	Вывод
Переполнение стека	12 <Enter> 1 <Enter> 7 <Enter> q <Enter> 7 <Enter> <Enter>	Сообщение об ошибке
Добавить элемент в стек	1 <Enter> w <Enter> 1 <Enter> q <Enter> 5 <Enter>	-> q(0x55f759db9c90) -> w(0x55f759db9c70)
Извлечь элемент из стека	3 <Enter> 3 <Enter> 15 <Enter>	q w Высвобожденные адреса: -> 0x55f759db9c70 -> 0x55f759db9c90
Проверить расстановку скобок в строке	13 <Enter> ({[{Hello}]}) <Enter>	::Скобки расставлены верно::
Проверить расстановку скобок в строке	14 <Enter>)(<Enter>	::Скобки расставлены не верно::
Провести сравнительный анализ	16 <Enter>	Таблица-результат анализа

Оценка эффективности

Для каждого измерения взято среднее значение по времени для 10000 измерений

Размер стека	Время заполнения стека-вектора (μs)	Время заполнения стека-списка (μs)	Время освобождения стека-вектора (μs)	Время освобождения стека-списка (μs)	Объем памяти, занимаемый заполненным стеком вектором (байт)	Объём памяти, занимаемый заполненным стеком-списком (байт)
32	0.960	1.452	0.868	1.343	72	544
33	1.124	1.416	0.877	1.365	104	560
64	1.437	2.234	1.057	1.937	104	1056
65	1.417	2.443	1.094	2.020	168	1072
128	1.743	4.487	1.493	3.297	168	2080
129	1.809	3.931	1.493	4.643	296	2096
256	2.857	7.543	2.274	6.722	296	4128
257	3.501	7.301	2.295	6.963	552	4144

Размер стека	Время заполнения стека-вектора (μs)	Время заполнения стека-списка (μs)	Время освобождения стека-вектора (μs)	Время освобождения стека-списка (μs)	Объем памяти, занимаемый заполненным стеком вектором (байт)	Объём памяти, занимаемый заполненным стеком-списком (байт)
512	4.521	14.882	3.922	12.065	552	8224
513	5.398	14.108	3.927	12.268	1064	8240
1024	8.588	26.194	7.167	22.477	1064	16416
1025	8.177	25.805	7.166	23.478	2088	16432
2048	15.203	48.669	13.669	42.026	2088	32800
2049	15.841	53.119	13.949	45.905	4136	32816
4096	29.348	94.020	26.629	81.330	4136	65568
4097	29.962	93.771	26.639	81.646	8232	65584
8192	58.524	184.529	52.727	158.125	8232	131104
8193	57.806	181.386	52.420	216.698	16424	131120
16384	114.804	365.253	104.527	368.764	16424	262176
16385	114.652	365.450	104.262	316.774	32808	262192
32768	231.492	749.096	210.918	768.051	32808	524320
32769	228.238	727.924	208.460	631.832	65576	524336
65536	454.046	1448.870	417.007	1254.242	65576	1048608
65537	453.441	1452.072	416.128	1794.711	131112	1048624
131072	908.015	2919.404	834.009	2543.430	131112	2097184
131073	903.787	2876.231	830.330	3726.210	262184	2097200
262144	1807.436	5787.805	1663.029	6192.459	262184	4194336
262145	1804.051	5759.440	1661.574	7466.025	524328	4194352

Заметим, что стек, построенный на векторе, работает гораздо быстрее стека, построенного на односвязном списке: в среднем заполнение стека, построенного на векторе, происходит `в 2,87 раза (~ на 187%)` быстрее, освобождение - `в 2,91 раза (~ на 191%)` быстрее.

Это объясняется тем, что для добавления каждого элемента в стек, основанный на списке, необходимо выделить память под новый узел, соответственно, при каждом добавлении элемента в стек-список просиходит обращение к оперативной памяти и формирование нового узла, а в случае стека-вектора - память выделяется только после добавления `2^n` элементов, когда вектор заполняется и происходит его расширение в `2 раза`

В то же время, вектор из-за подобного принципа работы нерационально использует память. Заметим, что при длине стека-вектора, равной `2^n` элементов, вектор полностью заполнен и вся выделенная память задействована, но при длине вектора - `2^n + 1` как раз произойдет расширение вектора и он станет

заполнен только наполовину, а вторая половина использоваться не будет. Для наглядности в таблице приведены именно длины стека `2^n` и `2^n + 1`.

В моем случае этот недостаток вектора нивелируется отсутствием необходимости хранить указатель на следующий элемент (как это происходит в случае односвязного списка). Это можно увидеть, проанализировав таблицу: да, в случае размера стека, равного `2^n + 1`, вектор занимает больше памяти, чем при `2^n` элементов, однако, несмотря на это, он более эффективен по памяти.

Тем не менее, вектор не всегда будет выгоднее списка по памяти. Чтобы понять, когда именно список окажется выгоднее, проанализируем объем занимаемой памяти более подробно. Для этого отдельно проведем замеры памяти для следующих типов данных элементов стека:

```
typedef char stacktype_t;       // 1 байт (меньше указателя на следующий элемент)
```

```
typedef long stacktype_t;       // 8 байт (совпадает с указателем на следующий элемент)
```

```
typedef char stacktype_t[256];  // 256 байт (больше указателя на следующий элемент)
```

Полученные результаты приведены на таблице ниже.

Длина стека	**Вектор** (1 байт)	**Список** (1 байт)	**Вектор** (8 байт)	**Список** (8 байт)	**Вектор** (256 байт)	**Список** (256 байт)
64	104	608	552	1056	16440	16928
65	168	617	1064	1072	32824	17192
256	296	2336	2088	4128	65592	67616
257	552	2345	4136	4144	131128	67880
1024	1064	9248	8232	16416	262200	270368
1025	2088	9257	16424	16432	524344	270632
4096	4136	36896	32808	65568	1048632	1081376
4097	8232	36905	65576	65584	2097208	1081640
16384	16424	147488	131112	262176	4194360	4325408
16385	32808	147497	262184	262192	8388664	4325672
65536	65576	589856	524328	1048608	16777272	17301536
65537	131112	589865	1048616	1048624	33554488	17301800

Из таблицы видно, что с увеличением размера, отведенного под значение элемента, стек, построенный на векторе,ьначинает проигрывать по памяти:

Размер данных стека	Эффективность
1 байт	`Вектор эффективнее ~ на 640%`
8 байт	`Вектор эффективнее ~ на 50%`
256 байт	`Список эффектинвее ~ на 60%`

Более того, в случае размера `2^n + 1` при 8 байтах объемы становятся **сравнимы**, а при 256 байтах - стек-список эффективнее примерно `в 2 раза` !

Контрольные 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).
    - При работе со стеком доступен только его верхний элемент, причем классическая реализация стека предполагает, что просмотреть содержимое стека без извлечения (удаления) его элементов невозможно.
2. Каким образом и сколько памяти выделяется под хранение стека при различной его ализации?
    - Для каждого элемента стека, реализованного списком, выделяется память для хранения указателя и содержания элемента.
    - Для каждого элемента стека, реализованного массивом, выделяется память только для хранения содержания элемента.
3. Что происходит с элементами стека при его просмотре?
    - Все элементы стека удаляются, так как каждый раз достается верхний элемент стека.

Вывод

1. Стек-вектор в среднем заполняется и освобождается быстрее стек-списка примерно `в 3 раза`
2. Стек-вектор в большинстве случаев эффективнее стек-списка по памяти.
3. Стек-список оказывается эффективнее стек-вектора по памяти в случае, если размер данных много больше размера указателя на следующий узел *`(например: для 256 байт - в 1.5 раза)`*, либо неизвестна необходимая длина стека.

Следовательно:

- В большинстве случаев целесообразнее использовать стек, построенный на векторе. Особенно, если акцентируется внимание на **скорость** работы стека.
- Использование стека, построенного на основе списка, целесообразно, если
    - акцентируется внимание, в первую очередь, на эффективность программы по **памяти**
    - при этом размер указателя на следующий узел **много меньше**, чем размер содержимого узла,
    - либо необходимый размер стека неизвестен.

Фрагментация памяти при использовании списка

Первое заполенение стека:

Добавляю 'a', 'b', 'c'

```
-> c(0x55d592600fb0) -> b(0x55d592600f90) -> a(0x55d592600f70)
```

Высвобожденные адреса:

```
-> 0x55d592600f70  -> 0x55d592600f90  -> 0x55d592600fb0
```

Повторное заполенение стека

Добавляю 'f', 'g', 'h'

```
-> h(0x55d592600fb0) -> g(0x55d592600f90) -> f(0x55d592600f70)
```

Поскольку при повторном заполнении стека на основе списка использовались те же ячейки памяти, что и при его заполнении в первый раз, можно сделать вывод, что фрагментации памяти не происходит.