

Aufgabenblatt 4

letzte Aktualisierung: 26. November, 10:47 Uhr

(75334e8112f762feaced7df8abeb81af5c65cab)

Ausgabe: Mittwoch, 25.11.2015

Abgabe: spätestens Freitag, 04.12.2015, 20:00

Thema: Trees, tree traversal, reverse polish notation

Abgabemodalitäten

- Alle abzugebenden Quelltexte müssen ohne Warnungen und Fehler auf den Rechnern des tubIT/IRB mittels `gcc -std=c99 -Wall` kompilieren.
- Abgaben erfolgen prinzipiell immer in Gruppen à 2 Personen, welche in den Tutorien festgelegt wurden. Einzelabgaben sind explizit als solche gekennzeichnet.
- Die Abgabe erfolgt ausschließlich über SVN. Die finale Abgabe
 - für Gruppenabgaben erfolgt im Unterordner
Tutorien/t<xx>/Gruppen/g<xx>/Abgaben/Blatt<xx>/
 - für Einzelabgaben erfolgt im Unterordner
Tutorien/t<xx>/Studierende/<tuBIT-Login>/Abgaben/Blatt<xx>/

wobei die Ordner von uns erstellt werden.

- Benutze für alle Abgaben von Programmcode das folgende Namensschema: `introprog_blatt0X_aufgabe0Y_Z.c`, wobei X durch die Blattnummer, Y durch die Aufgabe und Z durch die Unteraufgabe zu ersetzen ist.
Beispiel: Aufgabe 1.2 wird zu: `introprog_blatt01_aufgabe01_2.c`
Gib für jede Unteraufgabe maximal eine Quellcodedatei ab, es sei denn, die Aufgabenstellung erfordert explizit die Abgabe mehrerer Dateien pro Aufgabe.
Benenne alle anderen Abgaben (Pseudocode, Textaufgaben) wie oben beschrieben. Die zugelassenen Abgabeformate sind PDF, ODT und Text (txt). Verwende auch hier eine Datei pro Aufgabe, nicht jedoch pro Unteraufgabe.

1. Aufgabe: Implementieren eines Taschenrechners (2 Punkte)

In dieser Aufgabe sollst Du einen einfachen Taschenrechner implementieren, der Addition, Subtraktion und Multiplikation auf Fließkommawerten beherrscht. Um das Einlesen der Eingabe einfach zu gestalten, verwendet dieser anstatt der geläufigen Infix Notation die auch als *Reverse Polish Notation* bekannte Postfix Notation:

Postfix Notation: 38 4 +

Infix Notation: 38 + 4

Diese Notation bietet den Vorteil, dass die Eingabe schrittweise mit Hilfe eines Stacks abgearbeitet werden kann und die Ausdrücke auch ohne Klammern eindeutig sind.

Postfix Ausdruck	equivalenter Infix Ausdruck	Wert
1 2 3 + +	1 + (2 + 3)	= 6
1 2 + 3 +	(1 + 2) + 3	= 6
1 2 3 + *	1 * (2 + 3)	= 5
3.1415 2 3 - *	3.1415 * (2 - 3)	= -3.1415
4.0 0.2 + 3.0 * 5 +	((4.0 + 0.2) * 3.0) + 5	= 17.6

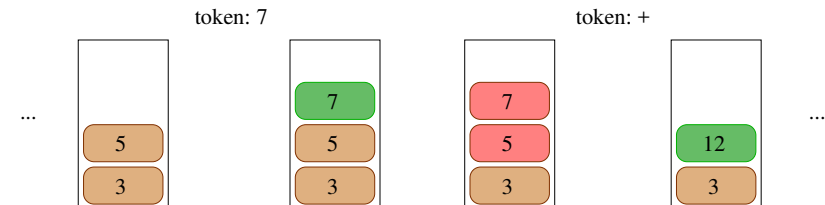


Abbildung 1: Stack während der Berechnung von 3 5 7 + +

Grüne Zahlen werden in dem Schritt auf dem Stack abgelegt und rote Zahlen vom Stack genommen.

Die Berechnung eines in Postfix Notation geschriebenen Ausdruck erfolgt wie folgt. Zunächst wird der als Zeichenkette bestehende Ausdruck (z.B. "0.3 -2 +") in die von Leerzeichen getrennte Abschnitte unterteilt ("0.3", "-2" und "+"). Jede dieser Einheiten, die selbst wieder eine Zeichenkette ist, bezeichnen wir als ein *Token*. Die Token werden dabei nacheinander, von links nach rechts, verarbeitet:

- Wenn es sich bei dem Token um eine, als Zeichenkette repräsentierte, Zahl handelt, dann konvertiere die Zahl in ein Zahlenformat (siehe `atof`) und platziere sie auf dem Stack (engl. `push`).
- Wenn es sich bei dem Token um einen Operator (+, - und *) handelt, dann nimm die obersten zwei Elemente vom Stack (engl. `pop`), führe die dem Operator entsprechende Operation aus und lege das Resultat der Operation wieder auf den Stack (engl. `push`).
 - Wenn der Stack zur Zeit der Abfrage, kein Elemente enthält, dann gibt er ein NAN ("not-a-number") zurück. Dieses Symbol ergibt bei jeder Berechnung ebenfalls ein NAN und ermöglicht es damit Fehler schnell zu finden.
- Wenn es sich bei dem Token um eine andere (als die genannten) Zeichenkette handelt, dann ignoriere das Token und fahre mit dem nächsten Token fort.

Das Programm endet wenn alle Token abgearbeitet wurden.

Um das Programm zu vereinfachen muss die Vorgabe verwendet werden, zusätzlich muss die Bibliothek `input_blatt04.c` und `input_blatt04.h` wie angegeben eingebunden werden. (Da die Ausgabe zu groß ist wurde sie weggelassen.)

Listing 1: Programmbeispiel

```
1 > gcc -std=c99 -Wall introprog_blatt04_aufgabe01.c input_blatt04.c \  
2 -o introprog_blatt04_aufgabe01  
3 > ./introprog_blatt04_aufgabe01
```

Zusätzlich benötigst du in dieser Aufgabe den Befehl `double` `atof(const char *string)`. Die Funktion `atof()` wandelt die Zeichenkette, auf die `string` zeigt, in eine Zahl des Format `double` um.

Listing 2: Beispiel für `atof`

```
1 char *string = "-1.2";  
2 float number = atof(string);  
3 printf("String:_%s_Zahl:_%d\n", string, number);
```

Dein Programm soll die folgenden Bedingungen erfüllen:

- **Funktionen:**

Programmiere die folgenden vier Funktionen und verändere nicht die anderen Funktionen in der Vorgabe:

- `stack_push(stack *astack, float value)`
Füge Element mit dem Wert `value` am Anfang des Stack ein.
- `stack_pop(stack *astack)`
Nehme das zuletzt eingefügte Element vom Stack und gebe den enthaltenen Wert zurück. Gebe `NAN` zurück, wenn der Stack leer ist.
- `stack* stack_erstellen()`
Erstelle ein Stack (dynamische Speicherreservierung & Initialisierung) und gebe einen Pointer auf den Stack zurück.
- `void process(stack *astack, char* token)`
Verarbeite die Token wie oben beschrieben.

- **Datenstrukturen:**

Mache dabei Gebrauch von den bestehenden Datenstrukturen:

Listing 3: Vorgabe `introprog_blatt04_aufgabe01_vorgabe.c`

```
1 typedef struct _stack stack;  
2 typedef struct _stack_element stack_element;  
3  
4 struct _stack {  
5     stack_element* top;  
6 };  
7  
8 struct _stack_element {  
9     stack_element* next;  
10     float value;  
11 };
```

- **Speicherverwaltung:**

Der Speicher soll zu Ende des Programm vollständig freigegeben sein.

Verwende folgende Vorgabe und füge Deinen Code an den entsprechenden Stellen ein:

Listing 4: Vorgabe `introprog_blatt04_aufgabe01_vorgabe.c`

```
1 #include <stdio.h>  
2 #include <stdlib.h>  
3 #include <string.h>  
4 #include <math.h> //definiert den speziellen Wert NaN für floats  
5 #include "input_blatt04.h"  
6  
7 typedef struct _stack stack;  
8 typedef struct _stack_element stack_element;  
9  
10 struct _stack {  
11     stack_element* top;  
12 };  
13  
14 struct _stack_element {  
15     stack_element* next;  
16     float value;  
17 };  
18  
19 /* Füge Element am Anfang des Stack ein  
20 *  
21 * astack - Ein Pointer auf den Stack.  
22 * value - Zahl die als neues Element auf den Stack gelegt werden soll.  
23 */  
24 void stack_push(stack* astack, float value)  
25 {  
26     /* HIER implementieren */  
27 }  
28  
29 /* Nehme das letzte eingefügte Element vom Anfang des Stack  
30 * Gebe NaN zurück, wenn keine Element vorhanden ist.  
31 *  
32 * astack - Ein Pointer auf den Stack  
33 *  
34 * Gebe die im Element enthaltenen Zahl zurück  
35 */  
36 float stack_pop(stack* astack)  
37 {  
38     /* HIER implementieren */  
39 }  
40  
41 /*  
42 * Führt abhängig von dem Token eine entsprechende Operation auf dem Stacks  
43 ↳ aus.  
44 * Wenn es sich bei dem Token um  
45 * -> eine Zahl handelt, dann case die Zahl mithilfe von atof zu einem float  
46 * und lege sie auf den Stack.  
47 * -> einen Operator handelt, dann nehme zwei Zahlen vom Stack, führe die  
48 * Operation aus und lege das Resultat auf den Stack.  
49 * -> eine nichterkennbare Zeichenkette handelt, dann tue nichts.  
50 *  
51 * astack - Ein Pointer auf den Stack
```

```

51  * token      - Eine Zeichenkette
52  */
53  void process(stack* astack, char* token)
54  {
55      /* HIER implementieren */
56      printf("\n<Logik_fehlt!>\n");
57      return;
58      /* Du kannst zur Erkennung der Token folgende Hilfsfunktionen benutzen:
59      *
60      * Funktion          / Rückgabewert von 1 bedeutet
61      * -----/-----
62      * is_add(token)     / Token ist ein Pluszeichen
63      * is_sub(token)     / Token ist ein Minuszeichen
64      * is_mult(token)    / Token ist ein
65      *                   / Multiplikationszeichen
66      * is_number(token) / Token ist eine Zahl
67      */
68  }
69
70  /* Debugausgabe des Stack
71  * Diese Funktion kannst du zum debugging des Stack verwenden.
72  *
73  * astack      - Ein Pointer auf den Stack
74  */
75  void print_stack(stack *astack) {
76      int counter = 0;
77      printf("\n_|xxxxx|xxxxxxxxxxxx|xxxxxxxxxxxx|xxxxxxxxx|\n");
78      printf("_|_|Nr_|_|Adresse_|_|Next_|_|Wert_|_||\n");
79      printf("_|_|-----|-----|-----|-----|\n");
80      for (stack_element* elem=astack->top; elem != NULL; elem = elem->next) {
81          printf("_|_|%3d_|_|%9p_|_|%9p_|_|%7.3f_|_|\\n", counter, elem, elem->next,
82              ↪ elem->value);
83          counter++;
84      }
85      printf("_|_|xxxxx|xxxxxxxxxxxx|xxxxxxxxxxxx|xxxxxxxxx|\n");
86  }
87  /* Kreiere einen Stack mit dynamischen Speicher.
88  * Initialisiere die enthaltenen Variablen.
89  *
90  * Gebe einen Pointer auf den Stack zurück.
91  */
92  stack* stack_erstellen() {
93      /* HIER implementieren */
94  }
95
96  int main(int argc, char** args)
97  {
98      stack* astack = stack_erstellen();
99      char zeile[MAX_STR];
100     char* token;
101
102     intro();
103     while (taschenrechner_input(zeile) == 0) {

```

```

104     // Erstes Token einlesen
105     token = strtok(zeile, "_");
106
107     while (token != NULL) {
108         printf("Token:_%s\\n", token);
109         // Stackoperationen durchführen
110         process(astack, token);
111         // Nächstes Token einlesen
112         token = strtok(NULL, "_");
113         print_stack(astack);
114     }
115
116     printf("\\nExtrahiere_Resultat\\n");
117     float result = stack_pop(astack);
118     print_stack(astack);
119
120     if (astack->top != NULL) {
121         while (astack->top != NULL) {
122             stack_pop(astack); //Räume Stack auf
123         }
124         printf("\\nDoes_not_Compute:_Stack_nicht_leer!\\n");
125     } else if (result != result) {
126         printf("\\nDoes_not_Compute:_Berechnung_fehlgeschlagen!\\n");
127     } else {
128         printf("\\nDein_Ergebnis:_%t%.3f\\n\\n", result);
129     }
130 }
131     free(astack);
132 }

```

Hinweis: Diese Aufgabe verwendet gerade in der Vorgabe einige Funktionen, die Du noch nicht kennengelernt hast. Falls Du wissen möchtest, was diese tun, so kannst Du das mit dem folgendem Befehl in der Kommandozeile in Erfahrung bringen:

```
> man <Funktionsname>
```

Hinweis: In dieser Aufgabe wird der Wert NAN verwendet. Diese Nicht-Zahl wird in der Bibliothek `math.h` definiert. Für eine beliebige Zahl x gilt: $\text{NAN} + x = \text{NAN}$, $\text{NAN} - x = \text{NAN}$ und $\text{NAN} * x = \text{NAN}$. Diese Eigenschaft machen wir uns in dieser Aufgabe zu Nutze um Fehler zu finden. Allerdings hat NAN noch eine weitere Eigenschaft: $\text{NAN} \neq \text{NAN}$, d.h. der Wert NAN ist nicht nur ungleich jeder Zahl, sondern unterscheidet sich ebenso von sich selbst. Diese Eigenschaft benötigen wir in dieser Aufgabe nicht, aber sie kann, wenn nicht beachtet, zu Problemen führen.

Gib Deinen Quelltext in einer Datei mit folgendem Namen ab:

```
introprog_blatt04_aufgabe01.c
```

2. Aufgabe: Binäre Suchbäume (1 Punkt)

Im Folgenden betrachten wir die in der Vorlesung eingeführten binären Suchbäume.

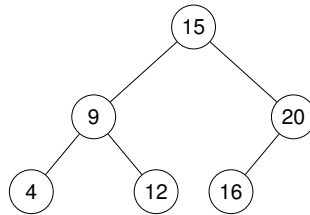


Abbildung 2: Binärbaum

2.1. In-order-tree-walk (0,5 Punkte) Durchlaufe den gegebenen Baum nach dem in der Vorlesung vorgestellten Verfahren. Gib die Reihenfolge an, in der die Knoten bearbeitet (auf den Slides wurden sie ausgegeben) werden.

Gib Deine Lösung der Teilaufgabe in einem Textdokument mit einem der folgenden Namen ab:
`introprog_blatt04_aufgabe02_1.{txt|odt|pdf}`

2.2. Queue (0,5 Punkte) Füge unter der Verwendung der (in der Vorlesung vorgestellten) `enqueue`-Funktion die folgenden Werte in dieser Reihenfolge in eine initial leere FIFO Queue ein: 15, 3, 6, 10, 11. Zeichne die resultierende Queue.

Gib Deine Lösung der Teilaufgabe in einem Textdokument mit einem der folgenden Namen ab:
`introprog_blatt04_aufgabe02_2.{txt|odt|pdf}`