

INTRODUCTION TO DATA STRUCTURES

Evaluating postfix expressions
& Translate infix to postfix in C

HW2_D84099084_賴時雨

1.Evaluating postfix expressions

實作過程

當使用 C 語言編寫程式以利用堆疊 (Stack) 資料結構來評估 postfix notation時，涉及堆疊這一抽象資料型態 (ADT) 以及評估 postfix 表示法的演算法。

使用的資料結構：堆疊 (Stack)

堆疊是一種線性資料結構，遵循 後進先出 (LIFO) 的原則。它 有兩個主要操作：push (將一個項目加到頂部) 和 pop (移除頂部項目)。透過堆疊實作中使用 Add (push) 和 Delete (pop) 函數來模擬這些操作。

程式邏輯與演算法

- 1. 初始化：創建一個空堆疊。
- 2. Tokenization：使用空格作為 token，將表達式分解為標記 (operands 和 operators)。
- 3. 處理 tokens：
 - 如果 tokens 是 operands，將其轉換為整數並推入堆疊。
 - 如果 tokens 是 operators，從堆疊彈出所需數量的 operands，應用操作符，然後將結果推回堆疊。
- 4. 最終結果：處理完所有 tokens 後，表達式的最終結果是堆疊上剩下的單一元素。

測試正確性

簡單測試案例：使用您可以手動驗證結果的已知簡單表達式來運行程式。例如，"1 1 2 * +" 應該得到結果 3。

邊界測試案例：包括測試程式極限的表達式，例如包含大量 tokens 的表達式，或可能導致整數溢出的表達式。

錯誤處理：包含設計上會失敗的測試案例，例如具有太多 operators 的表達式 (導致堆疊下溢) 或太多 operands (可能會在最後留下多個項目在堆疊上)。

Shih Yu Lai

Stack

IsFull()

Add()

IsEmpty()

Delete()

輸入範例
(簡報題目)

輸入測試
(Q1.Testcase)

```
C Q1.c test  05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000
```

測試與測試資料設計

正常案例：正確遵循 postfix notation 的表達式。

邊緣案例：僅有 single number 或 single operator 的表達式，看程式如何處理最小的輸入。

無效輸入：提供具有無效 tokens 或語法的表達式，以確保程式可以正確處理錯誤。

實際測試

編譯並運行帶有一系列測試案例作為輸入的程式。然後將輸出與預期結果進行比較。

main 函數包括一個固定的測試案例陣列，代表了可能的 postfix notation 表達式的良好組合。為了確保堆疊操作本身被正確實現，為函數編寫測試。這些測試通過控制輸入調用函數，並斷言函數輸出或結果堆疊狀態是否如預期。其中包含：

- 1. 向堆疊中添加和刪除元素，在每次操作後檢查 top 元素。
- 2. 將堆疊填充到容量極限，以確保 IsFull 返回 true，並且 Add 不允許推入更多元素。
- 3. 從空堆疊中刪除元素，檢查是否有正確的錯誤處理。

輸出

```
問題 輸出 設置主控台 結構圖 邊境圖 CODE REFERENCE LOG 註解
PS D:\Data Structure> cd "d:\Data Structure\h2\Q1" & if ($?) { gcc Q1.c -o Q1 } & if ($?) { .\Q1 }
Example:
Input: 1, Output: 3
-----
Test:
Input: 2, Output: 14
Input: 16, Output: 613
Input: 1, Output: 21
Input: 120, Output: 100
Input: 256, Output: 896
Input: 100000, Output: -12199
PS D:\Data Structure> h2\Q1
```

2. Translate infix to postfix

實作過程 (1/2)

C 語言程式利用堆疊資料結構將 infix notation (operator 位於 operands 之間的典型數學表示法) 轉換為 postfix notation (operator 跟在 operands 後面)。
使用的資料結構: 堆疊 (Stack)

選擇堆疊作為這項任務的資料結構, 因為它的後進先出 (LIFO) 特性適合處理 operator, 並確保它們以正確的順序被應用。在處理表達式時, 它暫時儲存 operator 和括號。

輸出

```
PS D:\Data Structure> cd "D:\Data Structure\hwo4\"; if ($?) { gcc Q2.c -o Q2 }; if ($?) { .\Q2 }

Example-----
Infix: 1 + 1 + 2
Postfix: 1 1 2 + +

-----Test-----
Infix: 2 + 3 * 4
Expected Postfix: 2 3 4 * +
Actual Postfix: 2 3 4 * +
Result: correct

-----
Infix: 16 * 38 + 5
Expected Postfix: 16 38 * 5 +
Actual Postfix: 16 38 * 5 +
Result: correct

-----
Infix: (1 + 2) * 7
Expected Postfix: 1 2 + * 7
Actual Postfix: 1 2 + * 7
Result: correct

-----
Infix: 120 * 5 / 6
Expected Postfix: 120 5 * 6 /
Actual Postfix: 120 5 * 6 /
Result: correct

-----
Infix: (256 / (10 - 6 + 12)) * (13 - 5) * 7
Expected Postfix: 256 10 - 6 + 12 / 13 - 5 * 7 *
Actual Postfix: 256 10 - 6 + 12 / 13 - 5 * 7 *
Result: correct

-----
Infix: 100000 / 20 - 135 + 26 * 76 - 560 * 34
Expected Postfix: 100000 20 / 135 - 26 76 * + 560 34 * -
Actual Postfix: 100000 20 / 135 - 26 76 * + 560 34 * -
Result: correct

PS D:\Data Structure> hwo4\
```

Shih Yu Lai

```
Data Structure > Hwo2 > C Q2.c > InfixToPostfix(char*, char*)
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <stdbool.h>
4 #include <string.h>
5 #include <ctype.h>
6 #include <limits.h> // Include for INT_MIN
7
8 #define MAX_STACK_SIZE 100
9
10 // Stack structure and function declarations
11 typedef struct {
12     int items[MAX_STACK_SIZE];
13     int top;
14 } Stack;
15
16 Stack* CreateS();
17 bool Add(Stack* stack, int item);
18 int Delete(Stack* stack);
19 bool IsEmpty(Stack* stack);
20 bool IsFull(Stack* stack);
21 int precedence(char operator);
22 void infixToPostfix(char* infix, char* postfix);
23
24 // Implementations of stack functions
25 // Function to create a stack
26 Stack* CreateS() {
27     Stack* S = (Stack*)malloc(sizeof(Stack)); // Allocate memory for stack
28     S->top = -1; // Initialize top to -1
29     return S;
30 }
31
32 // Function to check if stack is full
33 bool IsFull(Stack* stack) {
34     return stack->top == MAX_STACK_SIZE - 1; // Return true if stack is full
35 }
36
37 // Add an item to the stack
38 bool Add(Stack* stack, int item) {
39     if (IsFull(stack)) {
40         return false; // Return false if stack is full
41     }
42     stack->items[++stack->top] = item; // Add item to stack
43     return true; // Return true if stack is not full
44 }
45
46 // Function to check if stack is empty
47 bool IsEmpty(Stack* stack) {
48     return stack->top == -1; // Return true if stack is empty
49 }
50
51 // Function to delete an item from the stack
52 int Delete(Stack* stack) {
53     if (IsEmpty(stack)) {
54         return INT_MIN; // Use INT_MIN to indicate an error or empty stack
55     }
56     return stack->items[stack->top--];
57 }
58
59 // Function to return precedence of operators
60 int precedence(char operator) {
61     switch (operator) {
62         case '+':
63             return 1;
64         case '*':
65         case '/':
66             return 2;
67         default:
68             return 0; // For non-operators
69     }
70 }
71
72 // Function to convert infix expression to postfix
73 void infixToPostfix(char* infix, char* postfix) {
74     int i, j = 0;
75     char token;
76     for (i = 0; infix[i] != '\0'; i++) {
77         token = infix[i];
78         // Skip space in the infix expression
79         if (token == ' ') {
80             continue;
81         }
82         // Handle operands (digits)
83         if (isdigit(token)) {
84             // Add operand to postfix expression
85             postfix[j++] = token;
86             // Handle multi-digit numbers
87             while (isdigit(infix[i + 1])) {
88                 postfix[j++] = infix[i++];
89             }
90         }
91         // Add space after operand
92         postfix[j++] = ' ';
93     }
94     // Push '(' onto stack
95     Add(stack, token);
96     // Also if token == ')' {
97     // Pop everything up to the matching '('
98     while (!IsEmpty(stack) && stack->items[stack->top] != '(') {
99         postfix[j++] = Delete(stack);
100         postfix[j++] = ' ';
101     }
102     // Pop the '('
103     Delete(stack);
104     // Operator
105     // Pop operators with higher or equal precedence
106     while (!IsEmpty(stack) && precedence(token) <= precedence(stack->items[stack->top] && stack->items[stack->top] != '(') {
107         postfix[j++] = Delete(stack);
108         postfix[j++] = ' ';
109     }
110     // Push the current operator onto stack
111     Add(stack, token);
112 }
113
114 // Pop any remaining operators from the stack
115 while (!IsEmpty(stack)) {
116     postfix[j++] = Delete(stack);
117     postfix[j++] = ' ';
118 }
119 // Replace the last space with a null terminator
120 if (j > 0 && postfix[j - 1] == ' ') {
121     postfix[j - 1] = '\0';
122 }
123 // Free the stack
124 free(stack);
125
126 // Main function to read infix expressions from a file and convert them to postfix
127 int main() {
128     printf("=====Example=====\n");
129     char exampleInfix[] = "1 + 1 + 2 * 7"; // Example infix expression
130     char examplePostfix[strlen(exampleInfix) + 1]; // Postfix expression buffer
131     infixToPostfix(exampleInfix, examplePostfix); // Convert infix expression to postfix
132     printf("Infix: %s\nInfix: %s\n", exampleInfix, examplePostfix); // Print the postfix expression
133     printf("=====Test=====\n");
134     // Test
135     printf("=====Test=====\n");
136     FILE* q2file = fopen("Q2.Testcase.txt", "r"); // Open Q2 test file for reading
137     FILE* q1file = fopen("Q1.Testcase.txt", "r"); // Open Q1 test file for reading
138     char infix[MAX_STACK_SIZE]; // Buffer for reading an infix expression
139     char postfix[MAX_STACK_SIZE]; // Buffer for the corresponding postfix expression
140     char expectedPostfix[MAX_STACK_SIZE]; // Buffer for the expected postfix expression from Q1
141     if (q2file != NULL && q1file != NULL) {
142         while (fgets(infix, sizeof(infix), q2file)) {
143             // fgets(expectedPostfix, sizeof(expectedPostfix), q1file) == NULL {
144             // printf("Error reading from Q1 Testcase file.\n");
145             break;
146         }
147         infix[strcspn(infix, "\n")] = '\0'; // Remove newline character
148         expectedPostfix[strcspn(expectedPostfix, "\n")] = '\0'; // Remove newline character
149         infixToPostfix(infix, postfix); // Convert infix to postfix
150         printf("Infix: %s\nExpected Postfix: %s\nActual Postfix: %s\n", infix, expectedPostfix, postfix); // Print the results
151         // Compare actual postfix with expected postfix
152         if (strcmp(postfix, expectedPostfix) == 0) {
153             printf("Result: Correct\n");
154         } else {
155             printf("Result: Incorrect\n");
156         }
157         printf("-----\n");
158         fclose(q2file); // Close Q2 test file
159         fclose(q1file); // Close Q1 test file
160     } else {
161         if (q2file == NULL) perror("Error opening Q2 Testcase file");
162         if (q1file == NULL) perror("Error opening Q1 Testcase file");
163     }
164     return 0; // Return 0 to indicate successful execution
165 }
```

輸入

- Q2.Testcase是infix, Q1.Testcase是postfix
- 用Q2.Testcase為輸入, Q1.Testcase作為正解檢查Q2.Testcase的output

輸入範例
(簡報題目)

輸入測試
(Q2.Testcase)

測試

3

2. Translate infix to postfix

實作過程(2/2)

程式邏輯與演算法

1. **創建堆疊**: 在轉換過程中用來儲存 operator。
2. **對infix notation進行tokenize**: 逐字元處理 infix notation。
3. **處理每個字元**:
 - 如果是數字, 它是 operands的一部分, 因此直接加入到 postfix字符串。
 - 如果是開括號 (, 將其壓入堆疊。
 - 如果是閉括號), 從堆疊中彈出直到彈出一個開括號, 並將彈出的 operator 加到postfix字符串。
 - 如果是operator, 彈出堆疊中優先級更高或相等的 operator, 並將它們加入 postfix字符串, 然後將當前 operator 壓入堆疊。
4. **清空堆疊**: 在表達式的結尾, 將堆疊中剩餘的 operator 彈出至 postfix字符串。
5. **處理空白**: 空白不視為 postfix notation的一部分, 將被跳過。
轉換所用的演算法是 Shunting Yard algorithm, 該演算法專為解析 infix記號法指定的數學表達式而設計。它可用於 產生Reverse Polish notation (postfix)和abstract syntax trees。

測試正確性

為了驗證程式的正確性, 使用了兩個檔案: Q1.Testcase.txt 包含 postfix notation, 和 Q2.Testcase.txt 包含對應的 infix notation。

1. **運行示例案例**: 首先運行一個例子來演示功能。
2. **從 Q2.Testcase.txt 讀取測試案例**: 程式讀取每個 infix notation。
3. **轉換為 postfix**: 然後使用實作的函數將它們轉換為 postfix notation。
4. **從 Q1.Testcase.txt 讀取預期結果**: 程式從第一個文件讀取預期的 postfix結果。
5. **與實際輸出比較**: 將轉換結果與預期結果進行比較。
6. **Print結果**: 打印每個測試案例的結果, 指出轉換是否與預期的 postfix notation匹配。

測試與測試資料設計

簡單案例: 單個數字和無括號的簡單表達式。

複雜案例: 具有多位數字、多個操作和括號的表達式。

邊緣案例: 測試優先規則的表達式, 例如 "3 + 4 * 2" 應該結果為 "3 4 2 * +".

通過將程式輸出與預期的 postfix notation進行比較, 我們確保堆疊操作和 infix轉 postfix的轉換邏輯正常運作。

實際測試

運行程式時, 確保兩個測試檔案 (Q1.Testcase.txt 和 Q2.Testcase.txt) 位於可執行檔的正確位置, 並且它們的內容根據 infix和 postfix表示規則正確格式化。主函數中的比較將確認轉換演算法是否按預期工作。