Data Structures and Lab

Stack Data Structure

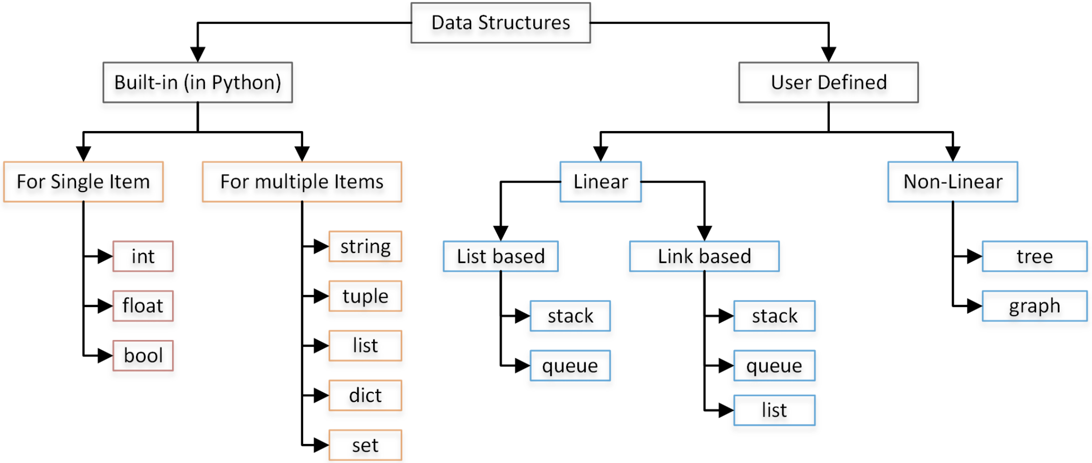
Muhammad Tariq Mahmood

tariq@koreatech.ac.kr
School of Computer Science and Engineering

Note: These notes are prepared from the following resources.

- Data Structures and Algorithms Using Python by Rance D. Necaise
- Data Structures Using Python by Y.K. Choi and I.G. Chan (Korean)
- https://www.geeksforgeeks.org/data-structures/
- https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/Python1a_OOP.html

Data Structures



Contents

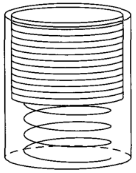
Stack Data Structure

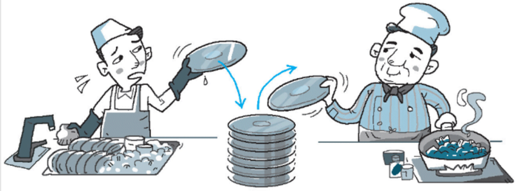
Stack Data Structure

- A stack data structure a type of list data structure with some restrictions.
- The items can only be accessed at one position i.e.
  - Add new items at the top
  - Remove an item at the top
- The last element to go into the stack is the first to come out
- Stack is also called LIFO (Last In, First Out) data structure
- Adding an item to a stack is referred to as pushing that item onto the stack
- Removing an item from the stack is referred to as popping the stack.

Stack Data Structure (cont...)

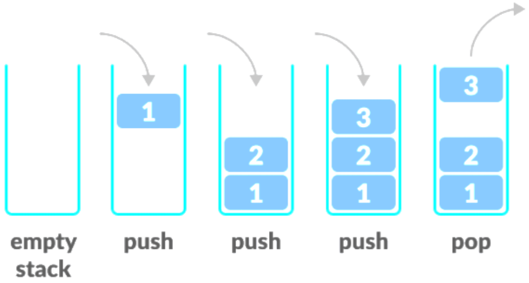
- One common explanation for this terminology is the operation of a spring-loaded stack of plates in a cafeteria:
- Object : Spring-Storage (cleaned dishes)
- Operations
  - Put the cleaned plate in the storage box - > Insert new item into the stack
  - Take one plate out of the storage box - > Take out an item.
  - Check if there are any plates in the container - > Check stack is empty
  - Find out what is on the top plate without taking out a plate.
  - Check whether the storage box is full.
  - Indicate the number of plates in the storage compartment.
  - Print out the dishes on the monitor.





Stack Data Structure (cont...)

- LIFO Principle of Stack



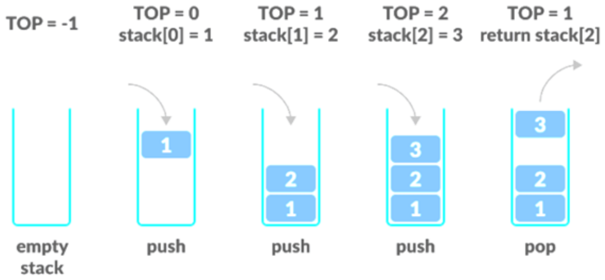
- In programming terms, putting an item on top of the stack is called "push" and removing an item is called "pop".
- In the above image, although item 3 was kept last, it was removed first - so it follows the Last In First Out(LIFO) principle.

Stack Data Structure (cont...)

- Stack abstract data type ADT
- Object: Collection of data items such that last-in first out (LIFO) mechanism is maintained
- Operations
  - push(x): adds an element x on the top of the stack
  - pop(): removes the top element of the stack. The next element will become the top element
  - isEmpty(): It returns true if the stack is empty, otherwise false
  - peek(): It returns the top element without removing it from the Stack
  - size(): It returns the number of items in the stack
  - display(): It displays all the elements stored in the stack

Stack Data Structure (cont...)

- Working of Stack Data Structure
  - A pointer called TOP is used to keep track of the top element in the stack.
  - When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing TOP == -1.
  - On pushing an element, we increase the value of TOP and place the new element in the position pointed to by TOP.
  - On popping an element, we return the element pointed to by TOP and reduce its value.
  - Before pushing, we check if the stack is already full (if static array is used)
  - Before popping, we check if the stack is already empty



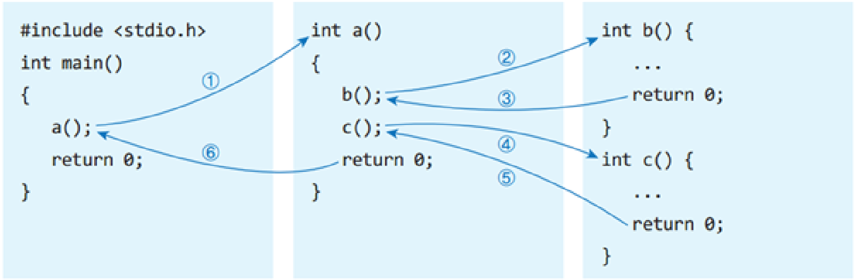
Applications of Stack Data Structure

Applications of Stack Data Structure

- There are many practical applications of stack data structure.
- Following are some of the important applications of a Stack data structure:
  - Stacks can be used to reverse a word and strings
  - Stacks can be used to check parenthesis matching in an expression.
  - Stacks can be used in Function Calls.
  - Stacks can be used for expression evaluation.
  - Stacks can be used for Conversion from one form of expression to another.
  - Stacks can be used for Memory Management.
  - Stack data structures are used in backtracking problems.

Use of Stacks in Function Calls

Use of Stacks in Function Calls



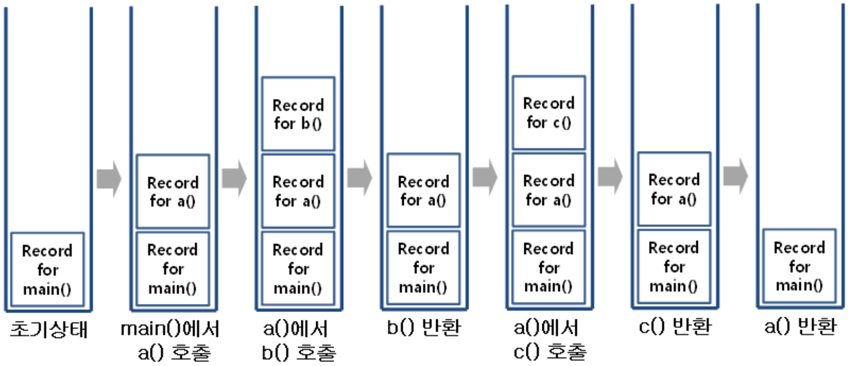
```
#include <stdio.h>
int main()
{
    a();
    return 0;
}
```

```
int a()
{
    b();
    c();
    return 0;
}
```

```
int b() {
    ...
    return 0;
}
int c() {
    ...
    return 0;
}
```

- Whenever a function begins execution (i.e., is activated), an activation record (or stack frame) is created to store the current environment for that function.
- Current environment includes values of its parameters, contents of registers, the function's return value, local variables, and the address of the instruction to which execution is to return when the function finishes execution.

Use of Stacks in Function Calls (cont...)

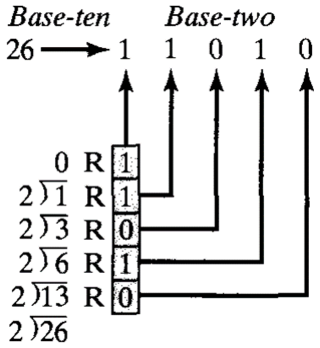


- If execution is interrupted by a call to another (or the same) function, this activation record must be saved
- A stack is the appropriate structure for storing current environment for different function calls,
- Since it is manipulated during execution, it is called the run-time stack.

Base conversion

Base conversion

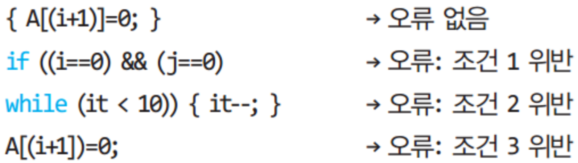
- One algorithm for base conversion from base-ten to base-two uses repeated division by 2, with the successive remainders giving the binary digits in the base-two representation from right to left.
- For example, the base-two representation of 26 is 11010
- First, push all remainders onto the stack
- Then, pop all elements from the stack



Parentheses Matching

Parentheses Matching

- Types of parentheses used in expressions: [,], { , }, (,)
- Given an expression string exp, write a program to examine whether the pairs and the orders of {, }, (,), [,] are correct in expression.
- Balanced parentheses : ( )( ), ((( ))), ((( )) ( ))
- Not Balanced parentheses (( ) (,) ( ) (
- Few Examples



```
{ A[(i+1)]=0; }                  → 오류 없음
if ((i==0) && (j==0)             → 오류: 조건 1 위반
while (it < 10)) { it--; }       → 오류: 조건 2 위반
A[(i+1])=0;                      → 오류: 조건 3 위반
```

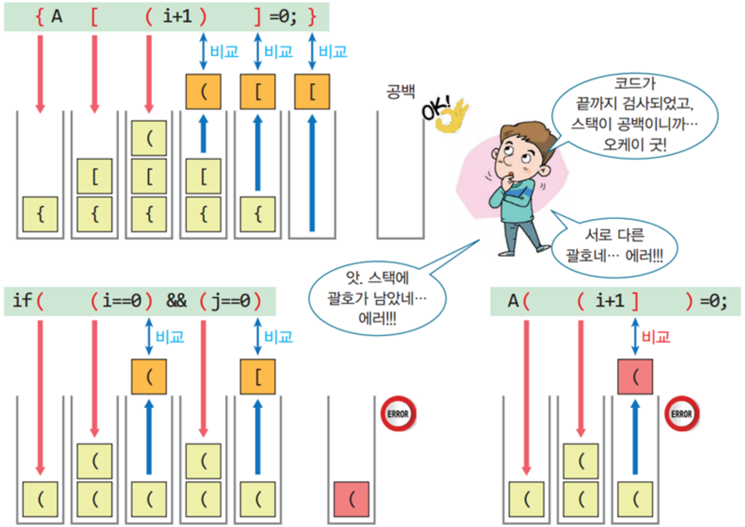
Parentheses Matching (cont...)

- Algorithm: Steps for Parentheses Matching
  - Declare a character stack s.
  - Now traverse the expression string *expr*.
  - If the current character *ch* is a starting bracket ((or { or [) then push it to stack.
  - If the current character *ch* is a closing bracket () or } or]) then pop from stack and if the popped character is the matching starting bracket then fine else parenthesis are not balanced.
  - After complete traversal, if there is some starting bracket left in stack then not balanced

```
check_matching(expr)
while (입력 expr의 끝이 아니면)
  ch ← expr의 다음 글자
  switch(ch)
    case '(': case '[': case '{':
       ch를 스택에 삽입
       break
    case ')': case ']': case ']':
       if ( 스택이 비어 있으면 )
         then 오류
         else 스택에서 open_ch를 꺼낸다
             if (ch 와 open_ch가 같은 짝이 아니면)
                 then 오류 보고
       break
if( 스택이 비어 있지 않으면 )
  then 오류
```

Parentheses Matching (cont...)

- Example : number of opening brackets and the order both are important



{ A [(i+1)] =0; }
비교
공백
OK!
코드가 끝까지 검사되었고, 스택이 공백이니까… 오케이 굿!
서로 다른 괄호네… 에러!!!
앗. 스택에 괄호가 남았네… 에러!!!
if((i==0) && (j==0)
A((i+1]) =0;
ERROR

Converting Infix to Postfix

Converting Infix to Postfix

- Consider the expression A+B: The operator + is applied to the operands A and B.+ is termed a binary operator: it takes two operands.

- Writing the sum of two operands can be expressed in three ways
  - A+B $\rightarrow$ *infix form*
  - + A B $\rightarrow$ *prefix form*
  - A B + $\rightarrow$ *postfix form*

- Algebraic expressions can be represented using three forms: prefix, infix, postfix

Infix Form	Prefix Form	Postfix Form
2 + 3 * 4	+ 2 * 3 4	2 3 4 *+
a * b + 5	+5 * a b	a b * 5 +
(1 + 2) + 7	+ 7 + 1 2	1 2 + 7+

Converting Infix to Postfix (cont...)

- Our objective is to transform an expression from infix form to postfix form . for example

Infix Form	Postfix Form
A + B	A B +
12 + 60 23	12 60 + 23
(A + B)*(C D)	A B + C D *

- Note that the postfix form an expression does not require parenthesis.
- Consider $4 + 3 * 5$ and $(4 + 3) * 5$. The parenthesis are not needed in the first but they are necessary in the second.
- the postfix forms are
  1. $4 + 3 * 5 \rightarrow 4\,3\,5 * +$
  2. $(4 + 3) * 5 \rightarrow 4\,3\; + 5*$

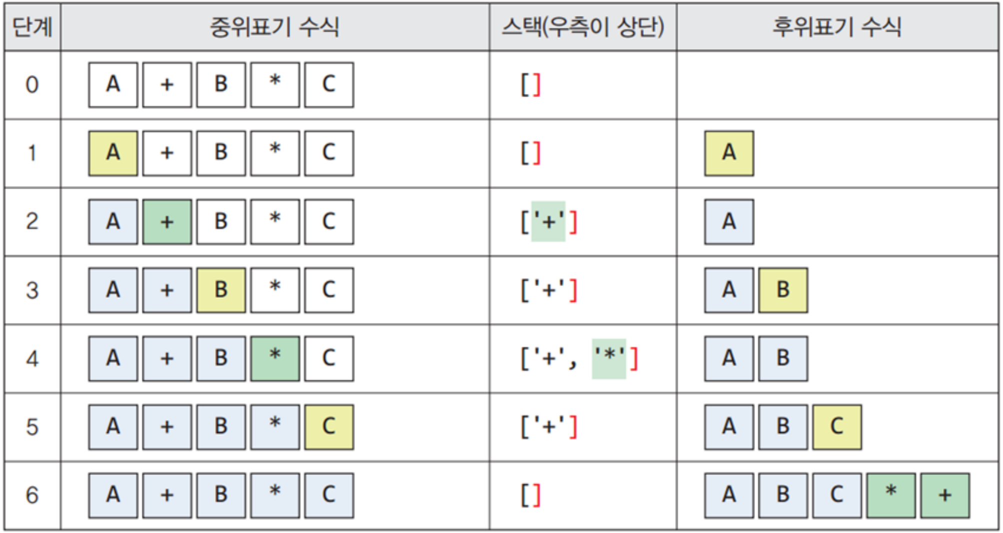
Converting Infix to Postfix (cont...)

- Algorithm : steps involved in converting from infix to postfix form.
  1. Read in the tokens one at a time
  2. If a token is an integer, write it into the output
  3. If a token is an operator, push it to the stack, if the stack is empty. If the stack is not empty, you pop entries with higher or equal priority and only then you push that token to the stack.
  4. If a token is a left parentheses '(', push it to the stack
  5. If a token is a right parentheses ')', you pop entries until you meet '('.
  6. When you finish reading the string, you pop up all tokens which are left there.
  7. Arithmetic precedence is in increasing order: '+', '-', '*', '/';
- Example. Suppose we have an infix expression:2+(4+3*2+1)/3.

```
'2' - send to the output.
'+' - push on the stack.
'(' - push on the stack.
'4' - send to the output.
'+' - push on the stack.
'3' - send to the output.
'*' - push on the stack.
'2' - send to the output.
```

Converting Infix to Postfix (cont...)

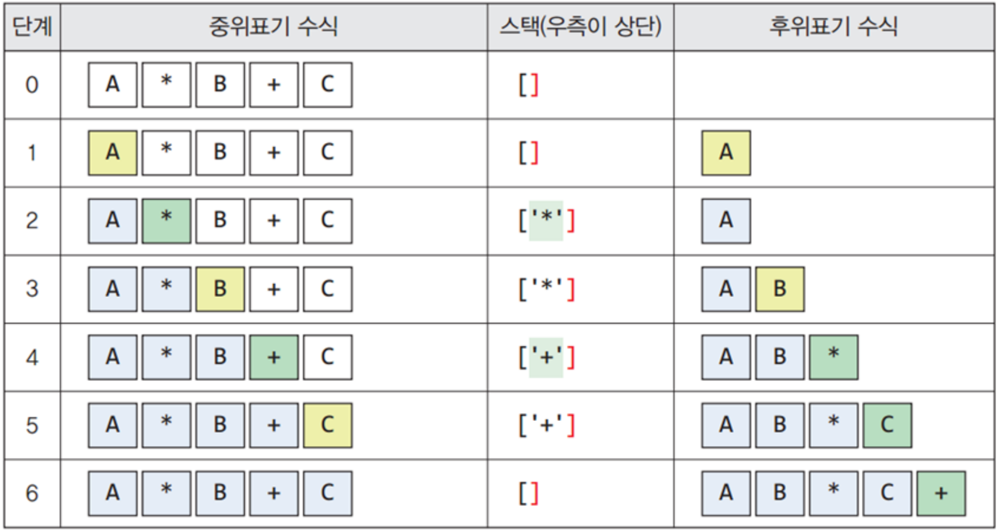
- Example: Procedure for conversion to postfix expression: A+B*C



단계	중위표기 수식	스택(우측이 상단)	후위표기 수식
0	A + B * C	[]	
1	A + B * C	[]	A
2	A + B * C	['+']	A
3	A + B * C	['+']	A B
4	A + B * C	['+', '*']	A B
5	A + B * C	['+']	A B C
6	A + B * C	[]	A B C * +

Converting Infix to Postfix (cont...)

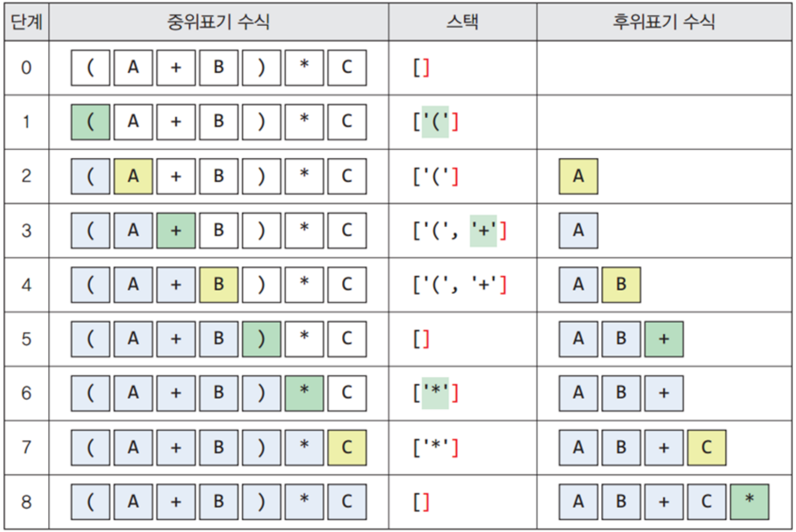
- Example: Procedure for conversion to postfix expression: A*B+C



단계	중위표기 수식	스택(우측이 상단)	후위표기 수식
0	A * B + C	[]	
1	A * B + C	[]	A
2	A * B + C	['*']	A
3	A * B + C	['*']	A B
4	A * B + C	['+']	A B *
5	A * B + C	['+']	A B * C
6	A * B + C	[]	A B * C +

Converting Infix to Postfix (cont...)

- Example: Procedure for conversion to postfix expression: (A+B)*C



단계	중위표기 수식	스택	후위표기 수식
0	(A + B) * C	[]	
1	(A + B) * C	['(']	
2	(A + B) * C	['(']	A
3	(A + B) * C	['(', '+']	A
4	(A + B) * C	['(', '+']	A B
5	(A + B) * C	[]	A B +
6	(A + B) * C	['*']	A B +
7	(A + B) * C	['*']	A B + C
8	(A + B) * C	[]	A B + C *

Evaluating Postfix Expression

Evaluating Postfix Expression

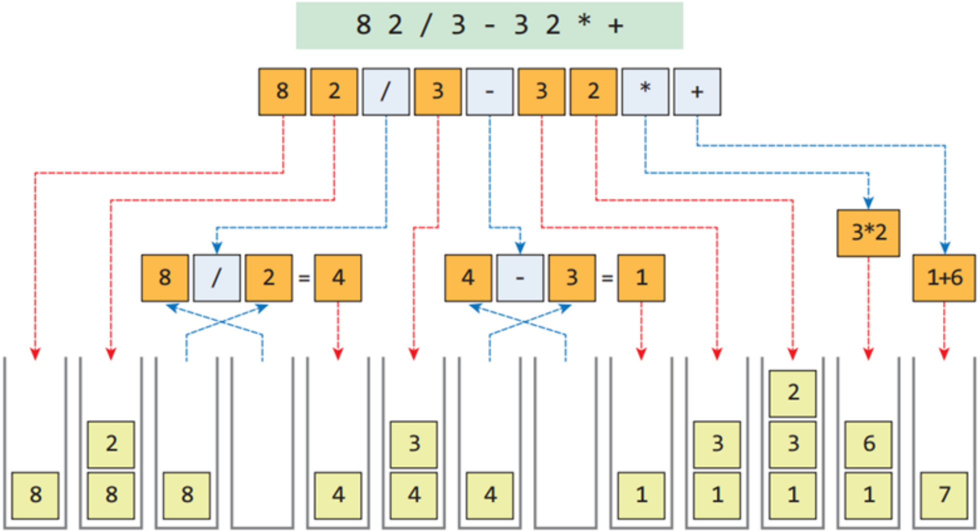
- Stack can also be used in evaluating postfix expressions
- The Postfix notation is used to represent algebraic expressions. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in postfix.
- Algorithm: Procedure for evaluation postfix expressions
  - Create a stack to store operands (or values).
  - Scan the given expression and do following for every scanned element.
  - If the element is a number, push it into the stack
  - If the element is a operator, pop operands for the operator from stack. Evaluate the operator and push the result back to the stack
  - When the expression is ended, the number in the stack is the final answer

```
calcPostfixExpr(expr)

스택 객체 s를 생성하고 초기화한다.
for 항목 in expr
  do if (항목이 피연산자이면)
        s.push(item);
     if (항목이 연산자 op이면)
       then second ← s.pop();
            first ← s.pop();
            temp ← first op second;
            s.push(temp);
result ← s.pop();
```

Evaluating Postfix Expression (cont...)

- Example: Procedure for evaluation postfix expressions



8 2 / 3 - 3 2 * +
8
2
/
3
-
3
2
*
+
8 / 2 = 4
4 - 3 = 1
3*2
1+6
8
2 8
8
4
3 4
4
1
3 1
2 3 1
6 1
7