What is the types of Data Structure for implementing Stack?

Data Structure: Stack

1. Understanding Data-Structure ADT and Basic concepts
2. Understand Operations of Data-Structure(or Algorithms)
3. Application of Data-Structure(or Algorithms)
4. Performance of Stack Operation
5. Solve Example problems

Convert infix-expr to postfix-expr.

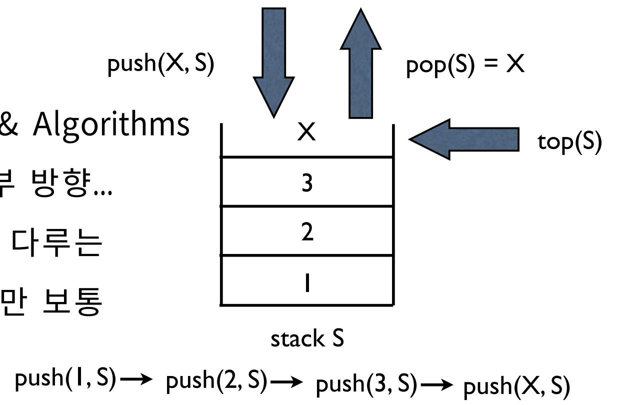
Analysis using Asymptotic Notation

Data Structure & Algorithms Class의 이론 공부 방향... 구체적인 내용은 다루는 주제마다 다르지만 보통 이 흐름대로..

Stack ADT

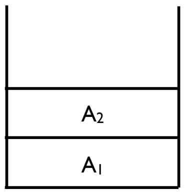
1. Last in First out 구조.
2. Stack이 지원하는 모든 연산은 Stack의 끝(Top_of_Stack)에서 이루어짐

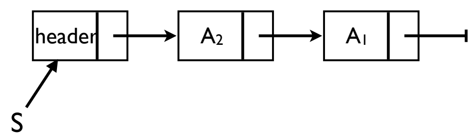
- a list that insertions and deletions can be performed at the end of the list
- operations
  - push(X, S): insert X in the list S
  - pop(S): deletes the most recently inserted element from S



이게 스택의 전부 이 두가지 개념을 잡으면 어렵지않게 Stack ADT와 연산의 진행 방법 이해 가능

Stack ADT: linked list implementation

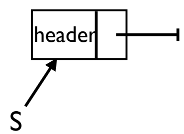





```
struct Node;
typedef struct Node *PtrToNode;
typedef PtrToNode Stack;

struct Node{
   ElementType Element;
   PtrToNode Next;
   };
```

Stack ADT: linked list implementation

```
Stack CreateStack (){

      Stack S;
      S = malloc(sizeof (struct Node));

      if (S==NULL)
          FatalError("Out of space !!!");

      S -> Next = NULL;
      return S;
}
```

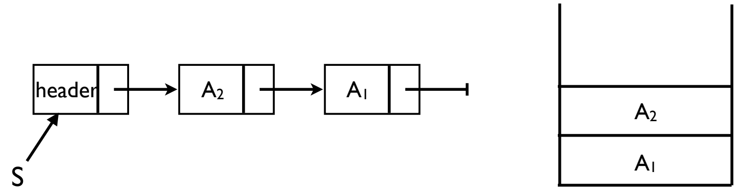


Stack ADT: linked list implementation

```
void MakeEmpty(Stack S)  {

      if  (S == NULL)
            Error (“No stack exists”);
      else
            while( !IsEmpty(S))
               Pop(S);
}
```

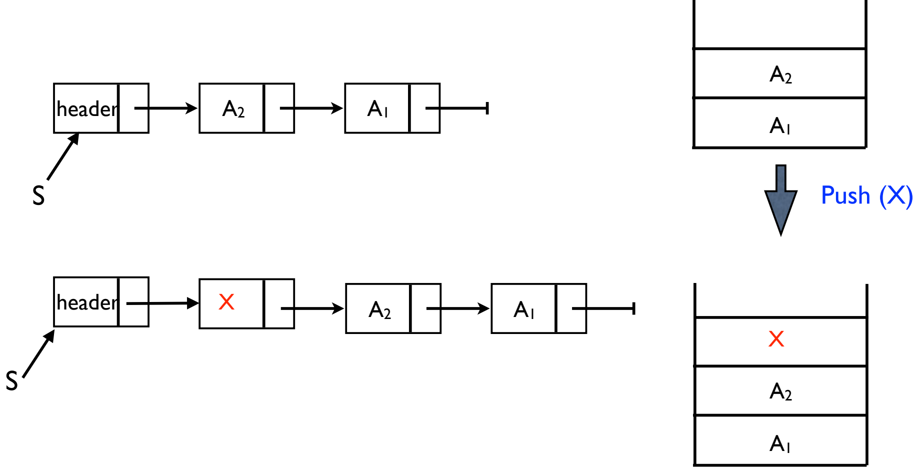
Stack ADT: linked list implementation

```
ElementType Top (Stack S)  {

      if (!IsEmpty(S))
            return    S->Next->Element;

      Error (“Empty stack”);
      return 0;
}
```

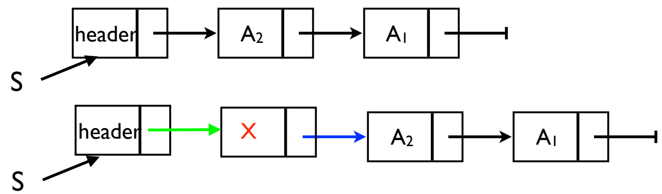


Stack ADT: linked list implementation

Push (ElementType X,   Stack S)



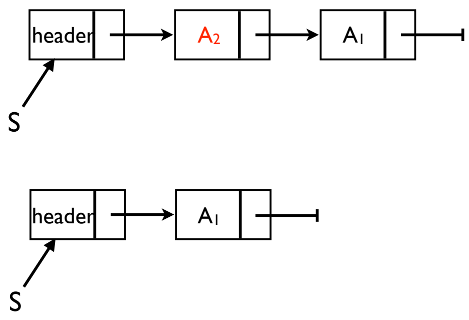
Stack ADT: linked list implementation

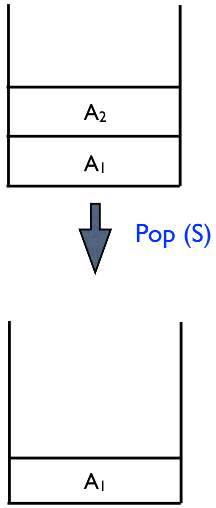



```
void Push (ElementType  X, Stack S)  {

         PtrToNode TmpCell;
         TmpCell = malloc (sizeof (struct Node));

         if (TmpCell ==NULL)  {
               FatalError(“Out of space !!!”);
         } else {
               TmpCell -> Element = X;
               TmpCell -> Next = S -> Next;
               S -> Next = TmpCell;
         }
}
```

Stack ADT: linked list implementation

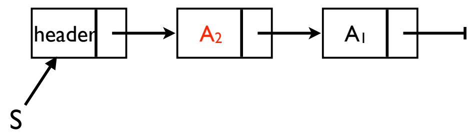
Pop (Stack S)





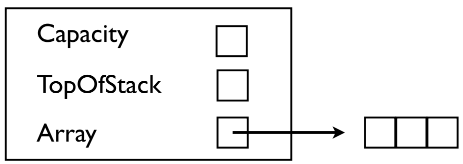
Stack ADT: linked list implementation

```
void Pop (Stack S) {

    PtrToNode FirstCell;

    if (IsEmpty(S))
        Error("Empty stack");
    else{
        FirstCell = S->Next;
        S->Next = S->Next->Next;
        free(FirstCell);
    }
}
```

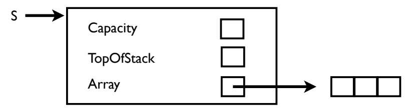


Stack ADT: array implementation

```
typedef struct StackRecord *Stack;

struct StackRecord
{
    int Capacity;
    int TopOfStack;
    ElementType *Array;
};
```



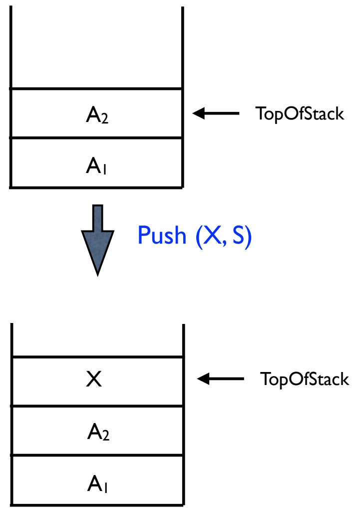
Stack ADT: array implementation




```
#define EmptyTOS ( -1 )

Stack  CreateStack( int MaxElements )
{
   Stack S;

   S = malloc( sizeof( struct StackRecord ) );
   if( S == NULL )
       FatalError( "Out of space!!!" );

   S->Array = malloc( sizeof( ElementType ) * MaxElements );
   if( S->Array == NULL )
       FatalError( "Out of space!!!" );

   S->Capacity = MaxElements;
   S->TopOfStack = EmptyTOS;

   return S;
}
```

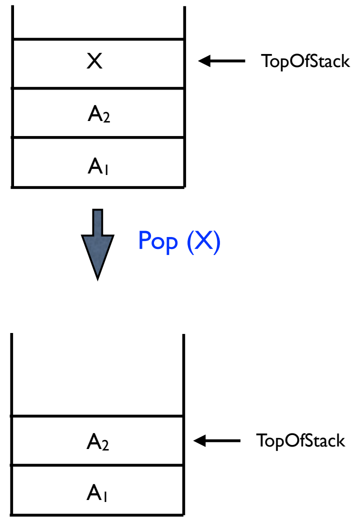
Stack ADT: array implementation

```
void  Push( ElementType X, Stack S )
{
    if( IsFull( S ) )
        Error( “Full stack” );
    else
        S->Array[ ++S->TopOfStack ] = X;
}
```



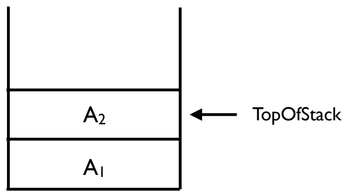
Stack ADT: array implementation

```
void  Pop( Stack S )
{
    if( IsEmpty( S ) )
        Error( “Empty stack” );
    else
        S->TopOfStack--;
}
```



Stack ADT: array implementation

```
ElementType  Top( Stack S)
{
    if( !IsEmpty( S ) )
        return S->Array[ S->TopOfStack ];

    Error( “Empty stack” );
    return 0;
}
```



infix, prefix, and postfix notation

infix

$3 + 4 * 6 \xrightarrow{?} (3 + 4) * 6$

$3 + 4 * 6 \xrightarrow{?} 3 + (4 * 6)$

prefix

$(3 + 4) * 6 \longrightarrow * + 3\ 4\ 6$

$3 + (4 * 6) \longrightarrow + 3 * 4\ 6$

postfix

$(3 + 4) * 6 \longrightarrow 3\ 4 + 6 *$

$3 + (4 * 6) \longrightarrow 3\ 4\ 6 * +$

postfix evaluation

$\underbrace{7\ 2\ 3\ *}_{2*3=6}\ -\ 4\ \uparrow\ 9\ 3\ /\ +$

$\underbrace{7\ 6\ -}_{7-6=1}\ 4\ \uparrow\ 9\ 3\ /\ +$

$\underbrace{1\ 4\ \uparrow}_{1^4=1}\ 9\ 3\ /\ +$

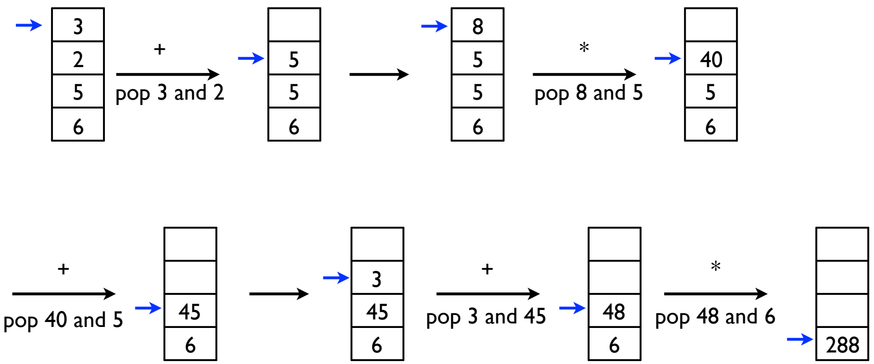
$1\ \underbrace{9\ 3\ /}_{9/3=3}\ +$

$\underbrace{1\ 3\ +}_{1+3=4}$

Stack ADT: postfix evaluation

6 5 2 3 + 8 * + 3 + *

→ TopOfStack



Stack ADT: postfix evaluation

- scan left-to-right
- place the operands on a stack until an operator is found
- perform operations by popping two elements in the stack when an operator is found

계산법 자체는 어렵지 않음. 그러나 Post-fix계산법이 중요한 이유는 뒤에 설명되어있는 Stack을 이용한 infix-to-postfix 변환과 같이 응용되어서 나오기 때문에 반드시 숙지 해야함

Stack ADT: postfix evaluation

10 2 8 * + 3 -

Stack ADT: translation of infix to postfix

3 + 4 * 6 ⟶ 3 4 6 * +

(3 + 4) * 6 ⟶ 3 4 + 6 *

3 + (4 * 6) ⟶ 3 4 6 * +

Stack ADT: translation of infix to postfix

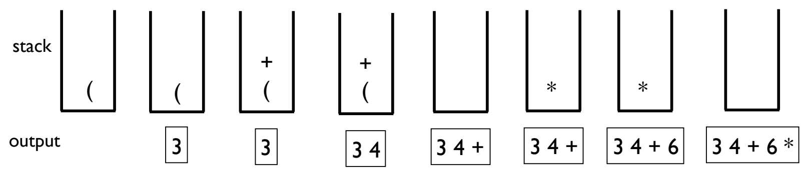
- When you meet an operand, print it.
- When you meet an operator, push it as long as the precedence of the operator at the top of the stack is less than the precedence of the incoming operator.
- When you meet an operator whose precedence is equal to or less than the precedence of the top of the stack, pop the top element and print it.
- When you meet the right parenthesis, pop all the operators until we reach the corresponding left parenthesis.
- When you meet the left parenthesis, push it in the stack.
- When you reach the end of expression, pop all the operators from the stack.

눈 감고도 남에게 설명해 줄 수 있을만큼 완벽하게 이해 하고 넘어가야함.

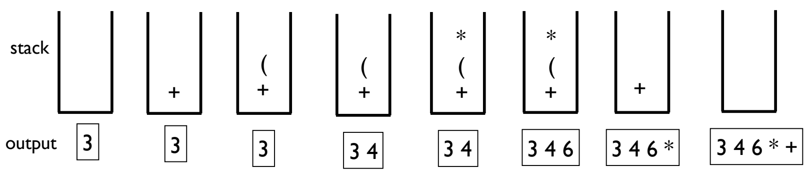
다음 페이지에 수록된 그림이랑 같이 보면서 완벽하게 이해.

Stack ADT: translation of infix to postfix

(3 + 4) * 6 ⟶ 3 4 + 6 *



3 + (4 * 6) ⟶ 3 4 6 * +

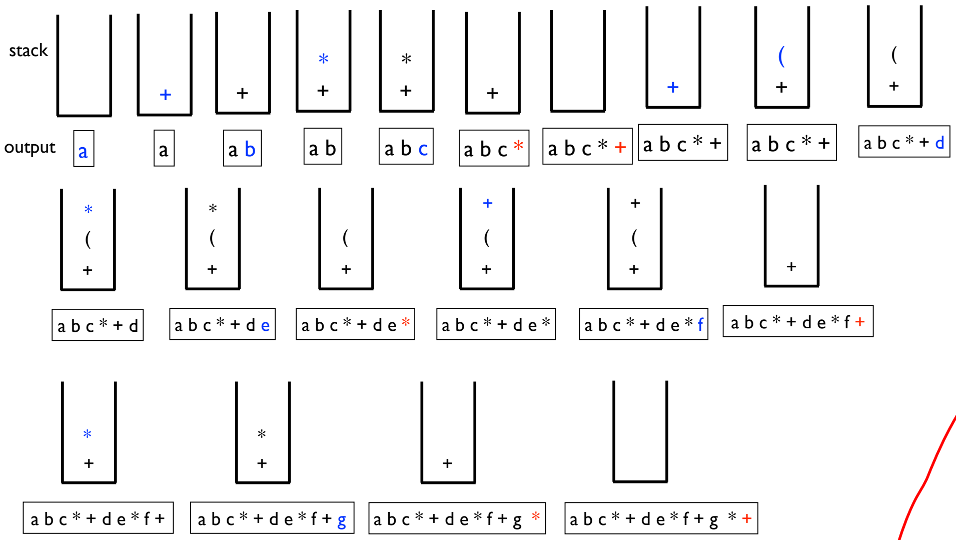


Stack ADT: translation of infix to postfix

a/b-c+d*e-a*c

Stack ADT: translation of infix to postfix

a + b * c + (d * e + f) * g → a b c * + d e * f + g * +



꼭... 다음 스터디 모임전까지
학교 교재 혹은 인터넷 찾아보며 혼자서 고민해보고
풀어볼것..
다른사람 통해서 듣는건 의미 없음
항상 자료구조 수업을 들을때 Best case, Worst Case
Performance Analysis에 관해서 생각해봐야함.
물론 Asymptotic Analysis(점근 표기법)에 대해서 더 심화되고 깊은
이론은 2학기때 알고리즘 과목에서 배우지만.
자료구조 시험문제 출제율 100%. 점근표기법에대한 완벽한
정의를 이용한 수학적 접근은 2학기때 열심히하고 .
최소한 PPT에 주어진 Source Code 단위의 Time complexit와
Space Complexity는 무조건 계산하고 이해할줄 알아야함

Stack ADT: translation of infix to postfix

infix	postfix
2 + 3 * 4	2 3 4 * +
a * b + 5	a b * 5 +
(1 + 2) * 7	1 2 + 7 *
a * b / c	a b * c /
((a / (b - c + d)) * (e - a) * c	a b c - d + / e a - * c *
a/b-c+d*e-a*c	a b / c - d e * + a c * -

Quiz for... 이 pdf를 보는 사람들

Q. Performance Analysis of Stack DS

we implement STACK using Array implementation, Linked list implementation, respectively.

Now, we assume the size of input data is N and the ADT of Stack is written in this pdf slide. Write your answers for each question below.

(Should answer your solution using Asymptotic Analysis)

Q1. in case of using array implementation, What is the running time of Push and Pop Operation? and What is the worst-case of POP operation?

1) Push :

2) Pop :

3) Worst-case :

Q2. in case of using Linked List implementation, What is the running time of Push and Pop Operation?

1) Push :

2) Pop :

3) Worst-case :

Q3. What is the space-complexity of Array implementation and Linked List?