Data Structures

Lecture 5: Stack

Dongbo Min

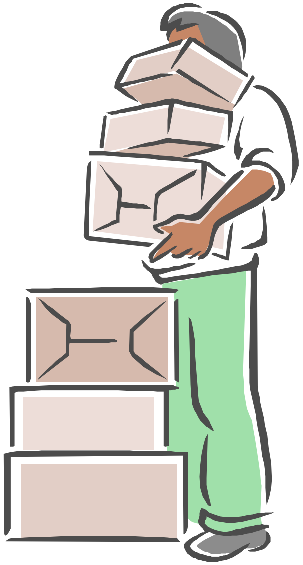
Department of Computer Science and Engineering

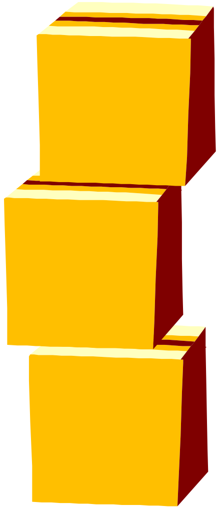
Ewha Womans University, Korea

E-mail: dbmin@ewha.ac.kr

Stack

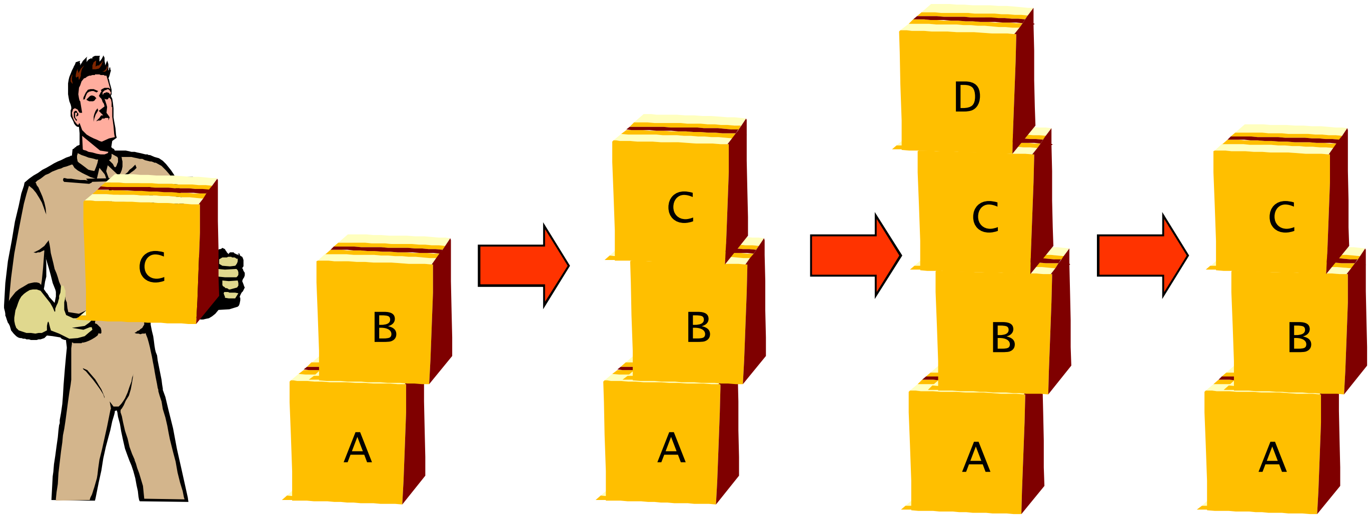
- Stack: a file of stacks



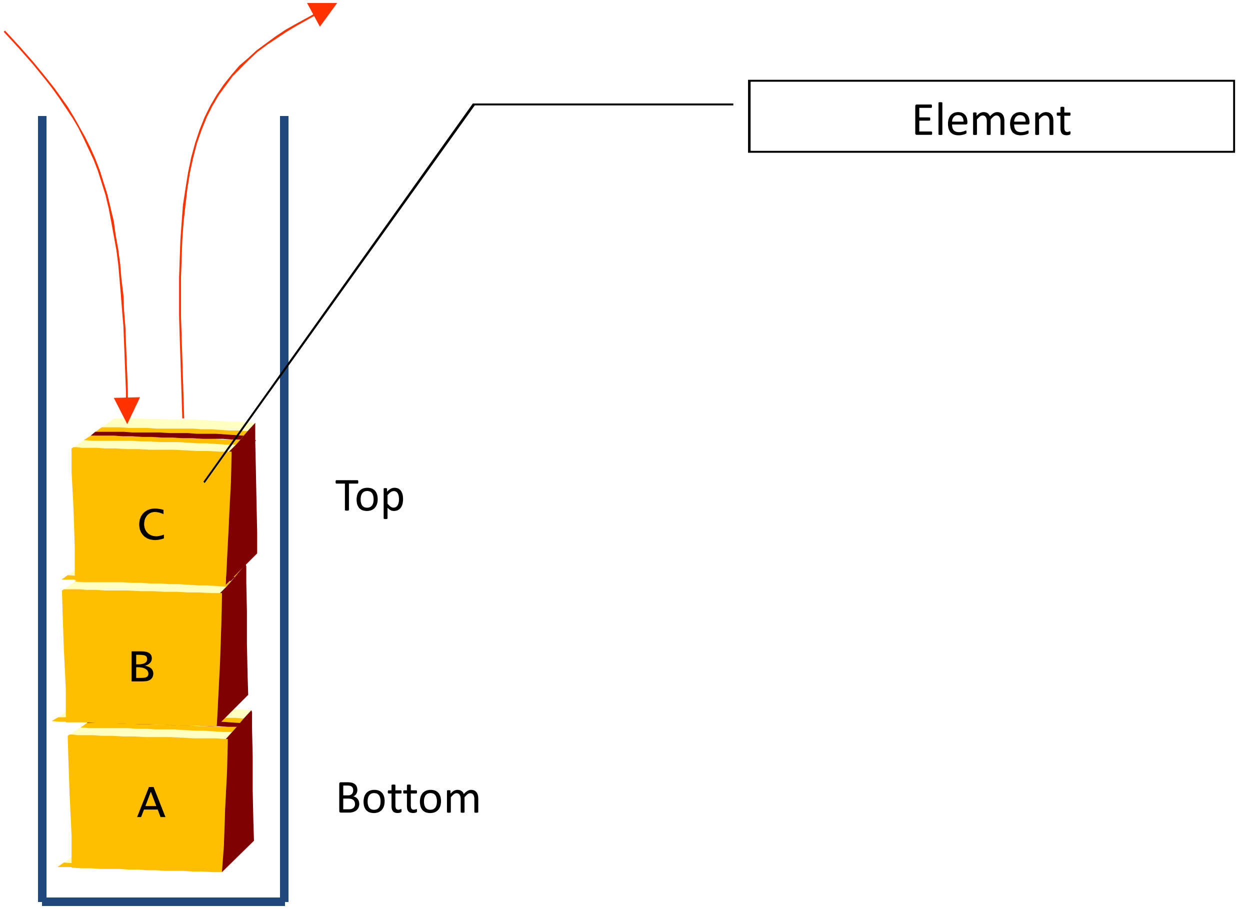


Stack

- Last-In First-Out (LIFI)
  - The most recent data comes first.



Stack



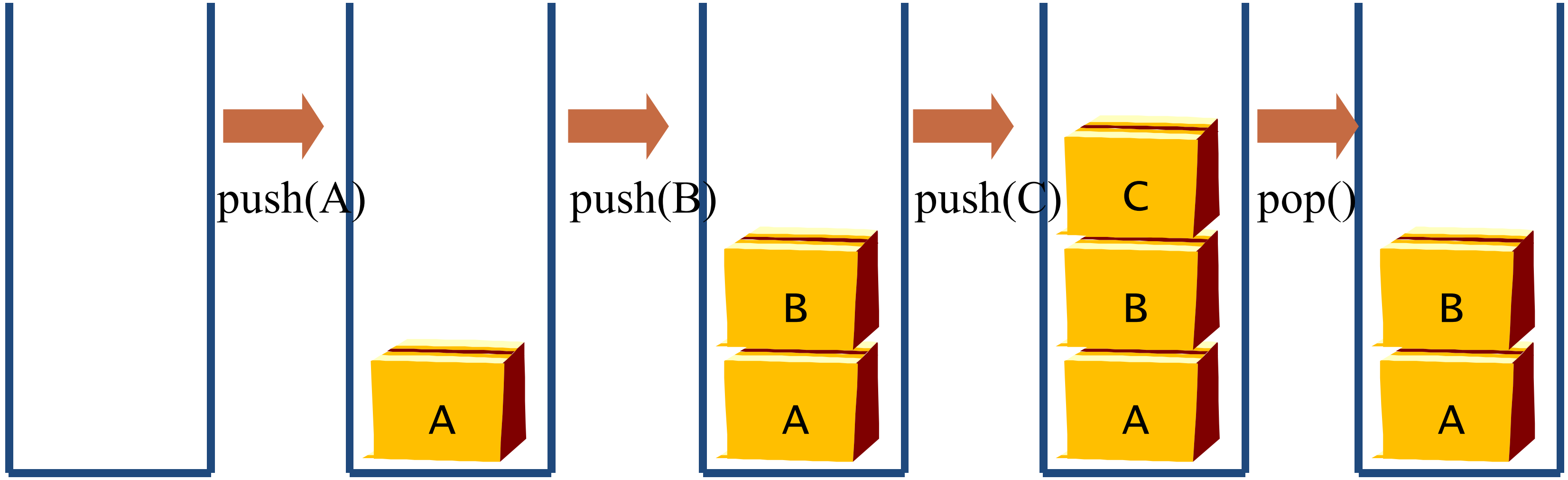
Abstract Data Type (ADT) of Stack

- Object: a linear list of n elements

- Operation
  - create() :: = Create a stack.
  - is_empty(s) :: = Checks if the stack ‘s’ is empty.
  - is_full(s) :: = Checks if the stack is full.
  - push(s, e) :: = Add element ‘e’ to the top of the stack.
  - pop(s) :: = Return element at the top of the stack and then deletes it.
  - peek(s) :: = Returns the element at the top of the stack without deleting it.

Stack Operation

- push(): add data to the stack
- pop(): delete data from the stack



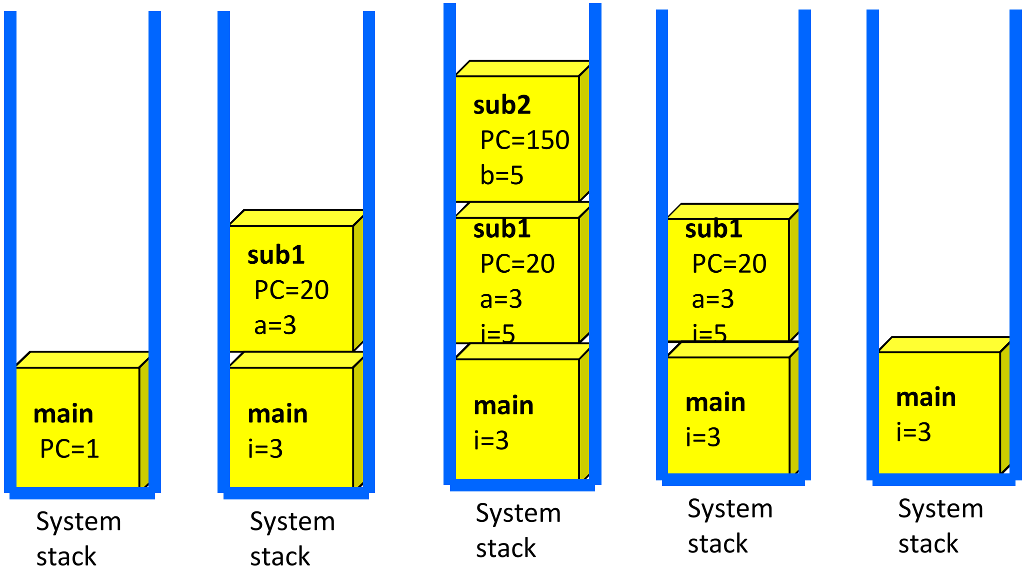
Stack Application

- Return output in reverse order to an input

  Ex) Undo function in editor

  Remember return address from function call

```
1       int main()
        {
            int i=3;
20           sub1(i);
             ...
        }

100     int sub1(int a)
        {
            int j=5;
150          sub2(j);
             ...
        }

200     void sub2(int b)
        {
              ...
        }
```

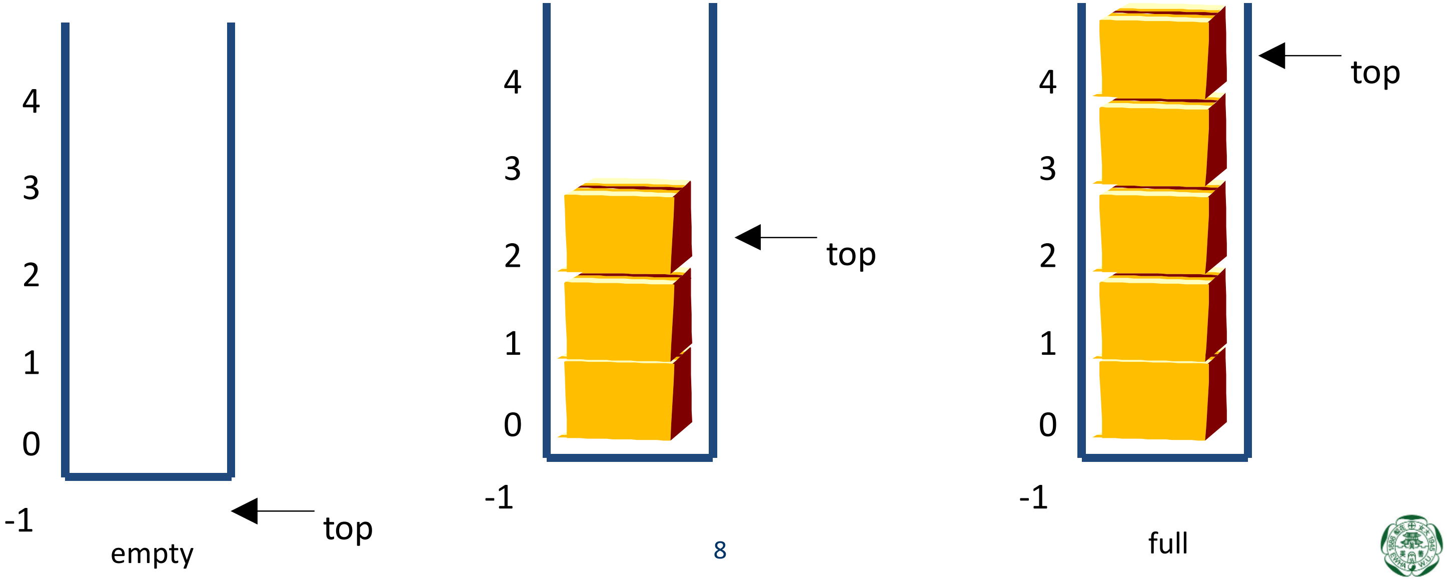


Stack Implementation using Arrays

- A stack that is implemented using arrays

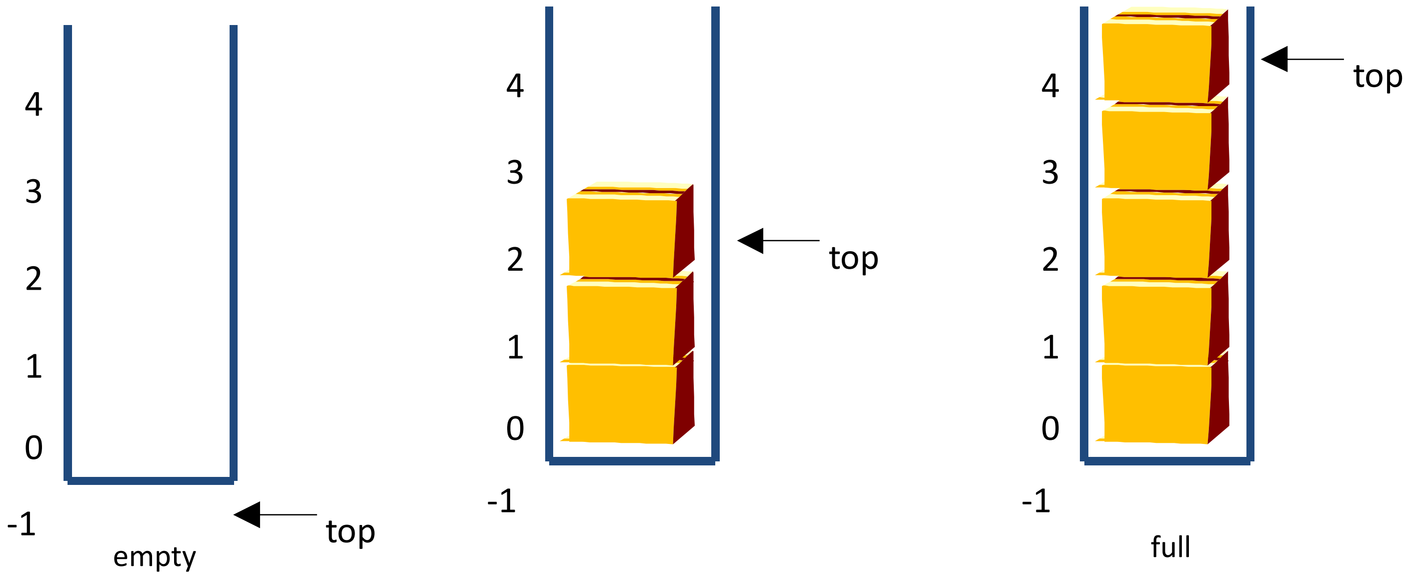
- Pros
  - The implementation is simple.
  - Insertion or deletion operations are fast.

- Cons
  - The stack size is limited.



Stack Implementation using Arrays

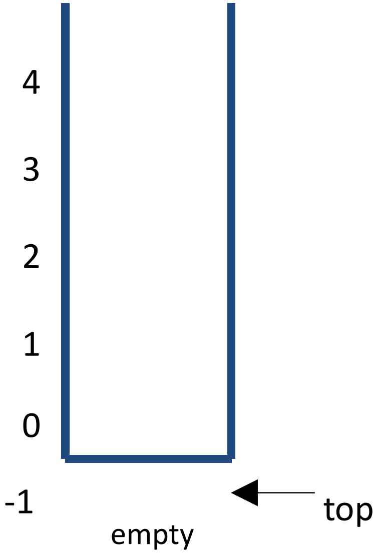
- 1-D array stack []
  - ‘top’: points to the most recently typed data in the stack
  - ‘stack[0]’: The first element
  - ‘stack[top]’: the last element
  - If the stack is empty, top = -1

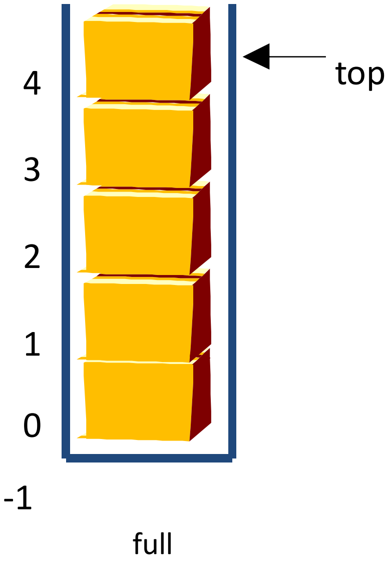


Stack Implementation using Arrays

```
is_empty(S)

if top = -1
         then return TRUE
else return FALSE
```

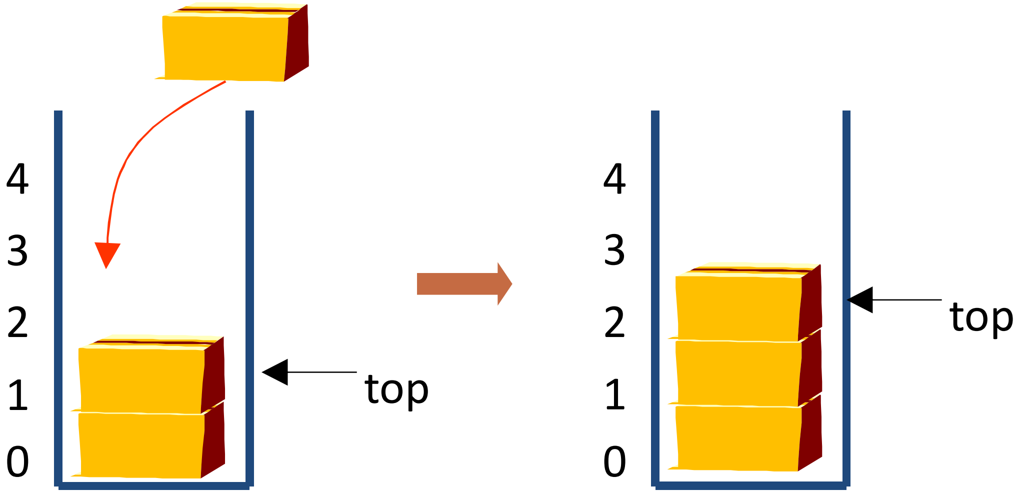
```
is_full(S)

if top = (MAX_STACK_SIZE-1)
         then return TRUE
else return FALSE
```





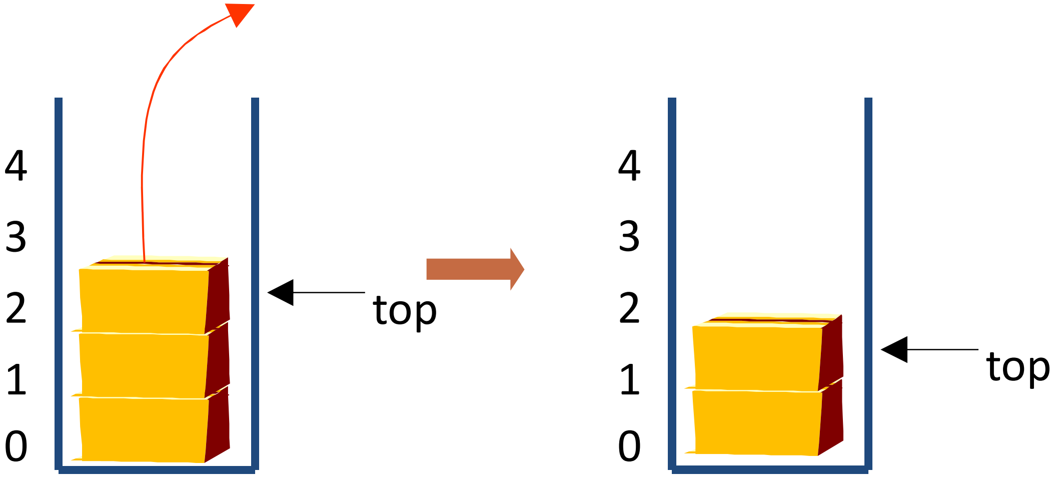
Stack Implementation using Arrays

```
push(S, x)

if is_full(S)
        then error "overflow"
else
        top ← top+1
        stack[top] ← x
```



Stack Implementation using Arrays

```
pop(S, x)

if is_empty(S)
         then error "underflow"
Else
         e ← stack[top]
         top ← top-1
         return e
```



Stack Implementation using Arrays

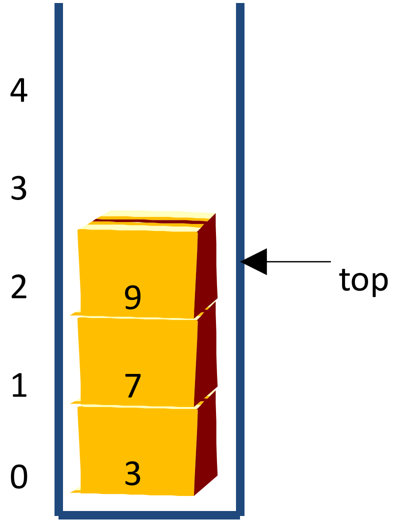
```c
typedef int element;
typedef struct {
        element stack[MAX_STACK_SIZE];
        int top;
} StackType;

// Stack initialization
void init(StackType *s)
{
        s->top = -1;
}
int is_empty(StackType *s)
{
        return (s->top == -1);
}
int is_full(StackType *s)
{
        return (s->top == (MAX_STACK_SIZE - 1));
}
```

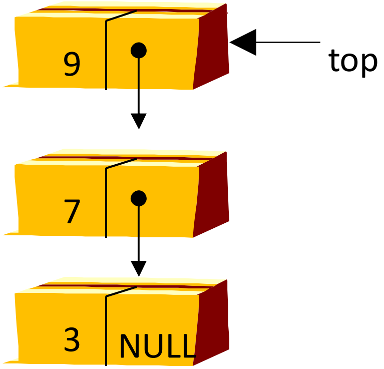
Stack Implementation using Arrays

```c
void push(StackType *s, element item)
{
        if (is_full(s)) {
                fprintf(stderr, "Stack is full\n");
                return;
        }
        else s->stack[++(s->top)] = item;
}

element pop(StackType *s)
{
        if (is_empty(s)) {
                fprintf(stderr, "Stack is empty\n");
                exit(1);
        }
        else return s->stack[(s->top)--];
}

element peek(StackType *s)
{
        if (is_empty(s)) {
                fprintf(stderr, "Stack is empty\n");
                exit(1);
        }
        else return s->stack[s->top];
}
```

Linked Stack

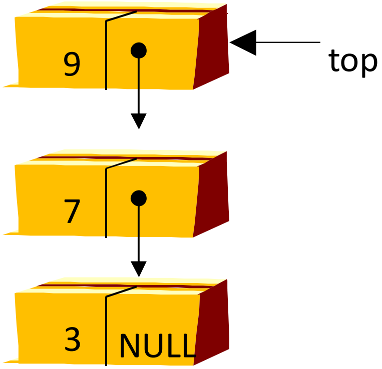
- A stack that is implemented using a linked list
- Pros
  - The stack size is not limited.
- Cons
  - The implementation is complex.
  - It takes a long time to insert or delete.



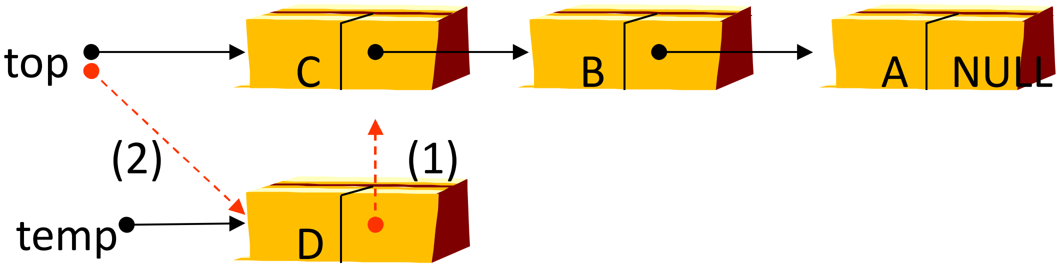


Linked Stack

```
typedef int element;

typedef struct StackNode {
          element item;
          struct StackNode *link;
} StackeNode;

typedef struct {
          StackNode *top;
} LinkedStackType;
```

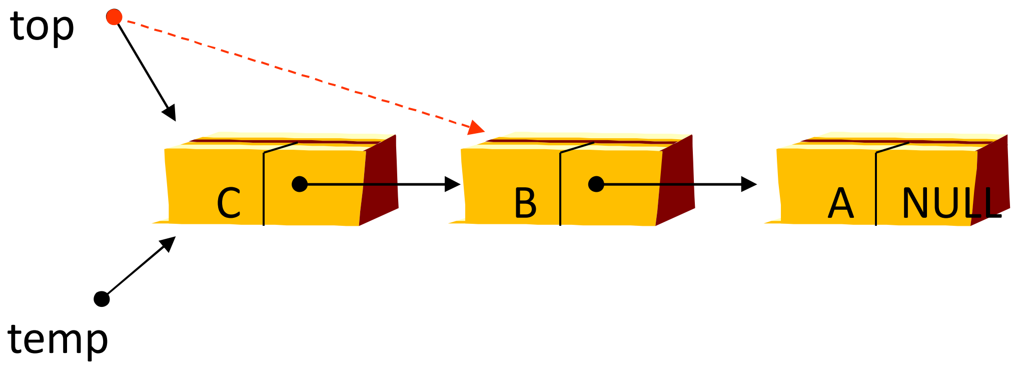


Operations in Linked Stack



```
void push(LinkedStackType *s, element item)
{
         StackNode *temp = (StackNode *)malloc(sizeof(StackNode));
         if (temp == NULL) {
         fprintf(stderr, “Memory allocation error\n");
         return;
         }

         else {
                  temp->item = item;
                  temp->link = s->top;
                  s->top = temp;
         }
}
```

Operations in Linked Stack



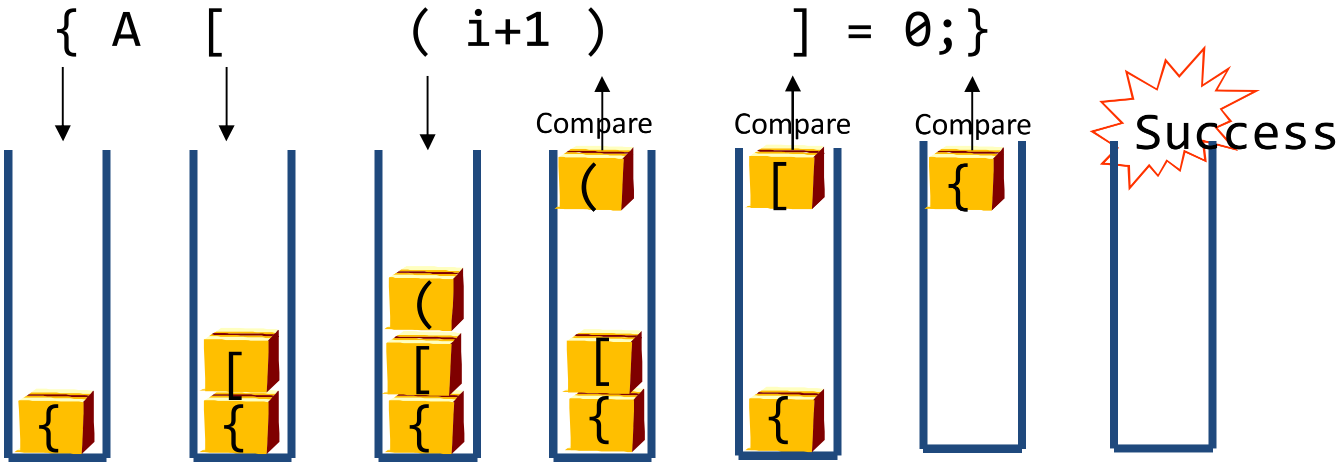
```
element pop(LinkedStackType *s)
{
        if (is_empty(s)) {
                fprintf(stderr, "Stack is empty\n");
                exit(1);
        }
        else {
                StackNode *temp = s->top;
                int item = temp->item;
                s->top = s->top->link;
                free(temp);
                return item;
        }
}
```

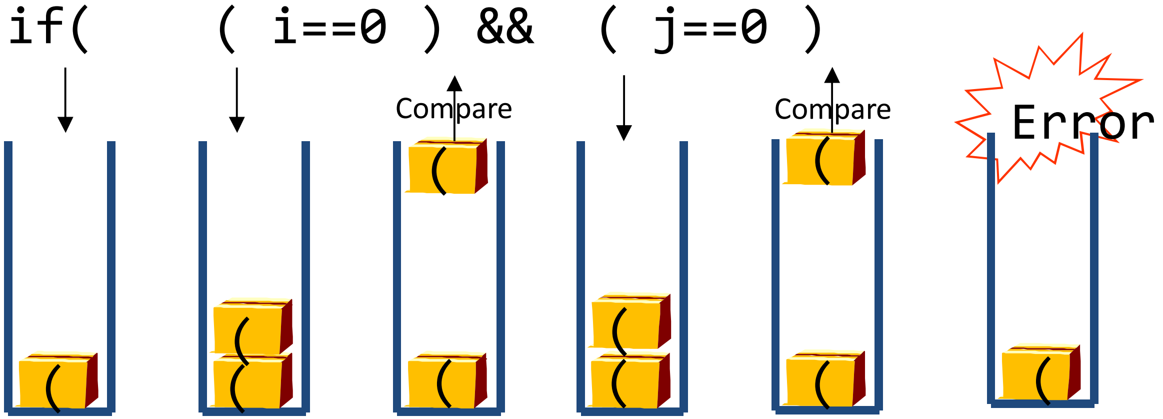
Stack Application: Parenthesis Check

- Types of parentheses
  - brackets ('[', ']')
  - braces ('{', '}')
  - parentheses ('(', ')‘
- Condition
  1. The number of left parentheses and right parentheses must be the same.
  2. The left parenthesis must precede the right parenthesis.
  3. The left and right parentheses of different types should not cross each other.
- Example

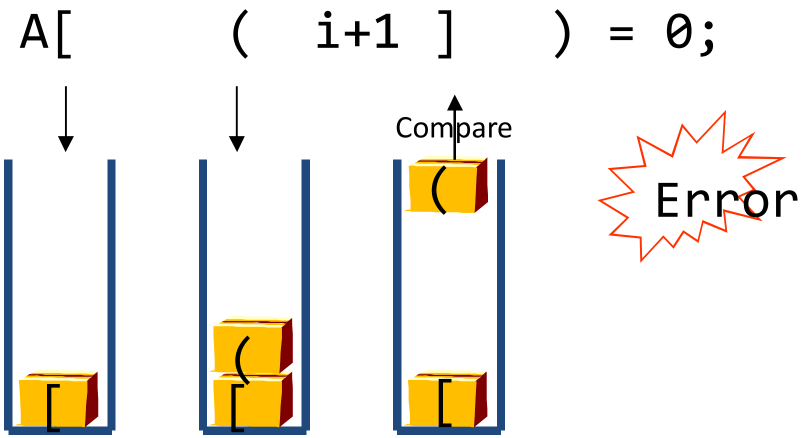
```
{ A[(i+1)] = 0; } → No error
if((i==0) && (j==0) → Condition 1 violation
A[(i+1])=0; → Condition 3 violation
```

Stack Application: Parenthesis Check


```
{ A [ ( i+1 ) ] = 0;}
```


```
if( ( i==0 ) && ( j==0 )
```

Stack Application: Parenthesis Check



Stack Application: Parenthesis Check

- Algorithm overview
  - If we encounter the left parenthesis, inserts it into the stack.
  - If we encounter the right parenthesis, check if the stack is empty. If the stack is empty, it violates ‘Condition 1’ or ‘Condition 2’. Otherwise, we remove the top parenthesis from the stack and check to see if it matches the right parenthesis. If the parentheses are mismatched, it violates Condition 3.
  - If the parentheses remain on the stack after checking the last parentheses, it returns 0 (false) because it violates condition 1, otherwise it returns 1 (true).

Stack Application: Parenthesis Check

- Pseudo code

```
check_matching(expr)

while (if not end of input expr)
         ch ← The next character in expr
         switch (ch)
                  case '(': case '[': case '{'
                           insert ch into the stack
                           break
                  case ')': case ']': case ‘}':
                           if (the stack is empty)
                                    then error
                           else take out open_ch from the stack
                                    if (ch and open_ch are not the same pair)
                                             then error
                           break

         if (the stack is not empty)
                  then error
```

```
int check_matching(char *in)
{
           StackType s;
           char ch, open_ch;
           int i, n = strlen(in);
           init(&s);

           for (i = 0; i < n; i++) {
                      ch = in[i];
                      switch (ch) {
                      case '(':   case '[':    case '{':
                                 push(&s, ch);
                                 break;
                      case ')':   case ']':    case '}':
                                 if(is_empty(&s))  return FALSE;
                                 else {
                                            open_ch = pop(&s);
                                            if ((open_ch == '(' && ch != ')') ||
                                                (open_ch == '[' && ch != ']') ||
                                                (open_ch == '{' && ch != '}')) {
                                                       return FALSE;
                                            }
                                            break;
                                 }
                      }
           }
           if(!is_empty(&s)) return FALSE;
           return TRUE;
}
int main()
{
           if( check_matching("{ A[(i+1)]=0; }") == TRUE )
                      printf(“Success\n");
           else
                      printf(“Fail\n");
}
```

Stack Application: Calculation of Formulas

- Formula expression

Infix	Prefix	Postfix
*2+3*4*	*+2*34*	*234*+*
*a*b+5*	*+5*ab*	*ab*5+*
(1+2)+7	*+7+12*	*12+7+*

- Calculation of formula on a computer
  - Infix expression -> Postfix expression -> Calculation
    - 2 + 3 * 4 -> 234 * + -> 14

Stack Application: Calculation of Formulas

- Calculation of postfix notation
  - Scan formulas from left to right

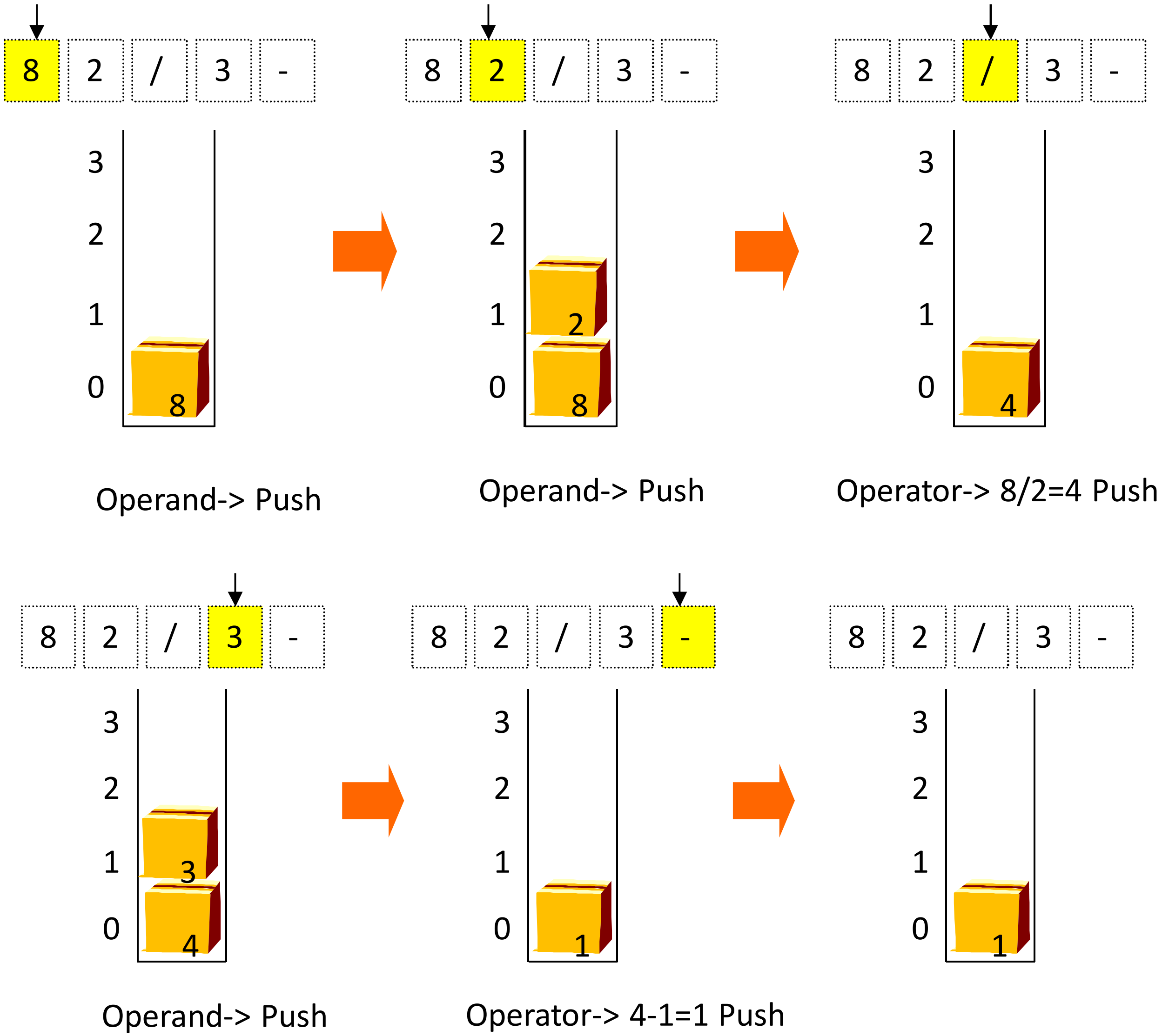
1. If it is an operand, store it in the stack.
2. If it is an operator, fetch the required number of operands from the stack.
3. Stores the operation result back into the stack.

8/2 – 3 + 3*2

→ 82/3–32*+

Token	Stack						
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
8	8						
2	8	2					
/	4						
3	4	3					
-	1						
3	1	3					
2	1	3	2				
*	1	6					
+	7						

Stack Application: Calculation of Formulas



Operand-> Push

Operand-> Push

Operator-> 8/2=4 Push

Operand-> Push

Operator-> 4-1=1 Push

Stack Application: Calculation of Formulas

- Algorithm overview

```
Create and initialize stack s.

for entry in postfix expression
        if (the item is an operand)
                push(s, item)
        if (item is operator op)
                then second ← pop(s)
                first ← pop(s)
                result ← first op second // op: + - * /
                push(s, result)

final_result ← pop(s);
```

```c
int eval(char exp[])
{
        int op1, op2, value, i = 0;
        int len = strlen(exp);
        char ch;
        StackType s;

        init(&s);
        for (i = 0; i<len; i++) {
                ch = exp[i];
                if (ch != '+' && ch != '-' && ch != '*' && ch != '/') {
                        value = ch - '0';       // Operand
                        push(&s, value);
                }
                else { //Operator
                        op2 = pop(&s);
                        op1 = pop(&s);
                        switch (ch) {
                        case '+': push(&s, op1 + op2); break;
                        case '-': push(&s, op1 - op2); break;
                        case '*': push(&s, op1*op2); break;
                        case '/': push(&s, op1 / op2); break;
                        }
                }
        }
        return pop(&s);
}

void main()
{
        int result;
        printf("Postfix expression : 8 2 / 3 – 3 2 * +\n");
        result = eval("82/3-32*+");
        printf("Result: %d\n", result);
}
```

Stack Application: Calculation of Formulas

- Infix to Postfix expression
  - The order of the operands is the same
  - The order of operators is different (priority order)

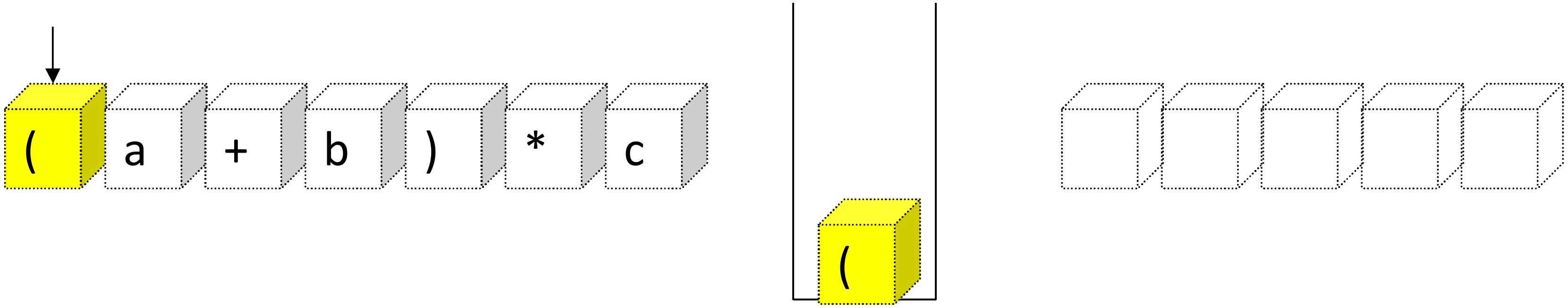
  → Operators can be stored on the stack and output them.

  2+3*4 → 234*+

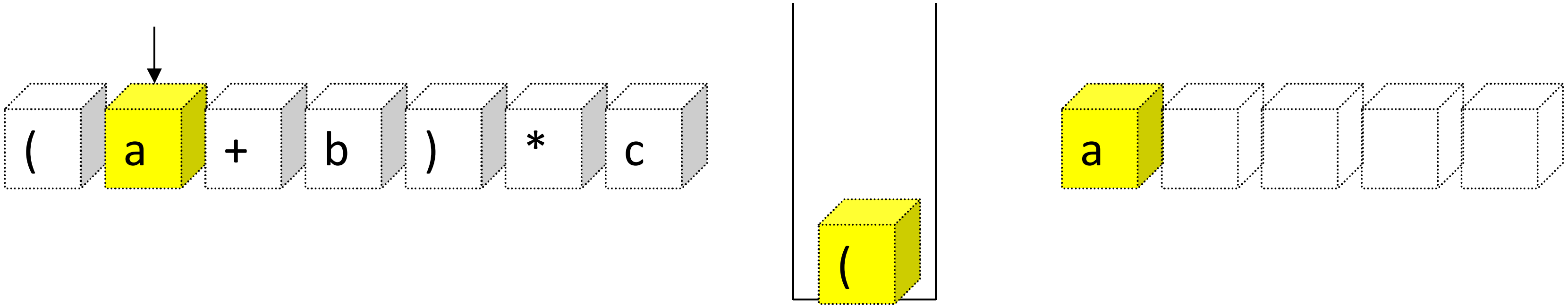
- Algorithm overview
  - Output when the operand is encountered
  - When the operator is encountered, it is stored on the stack.
  - Operator's priority
    - If operator in the stack has higher priority than or equal to current operator, output operators on the stack.

    2*3+4 → 234+*   2-3+4 → 234-+
  - Parenthesis
    - The left parenthesis is treated as the operator with the lowest priority
    - If the right parenthesis appears, output all the operators stacked on the left parenthesis

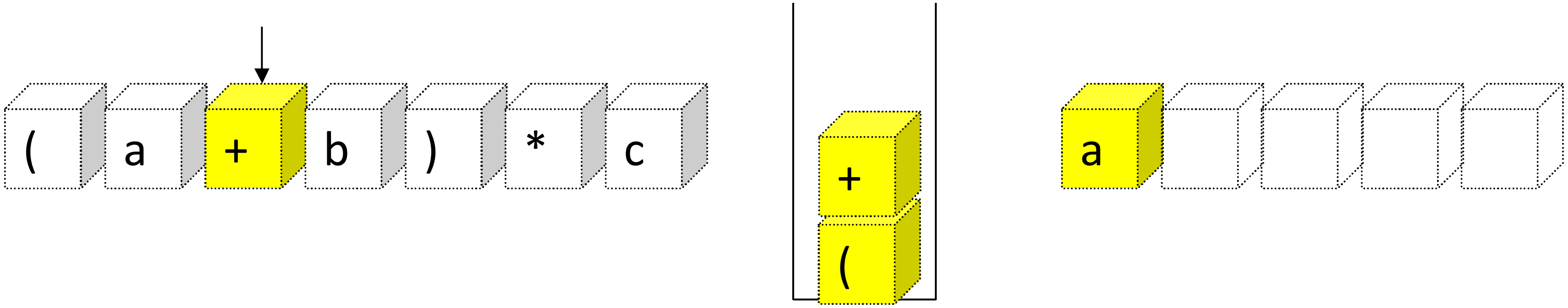
(a + b) * c



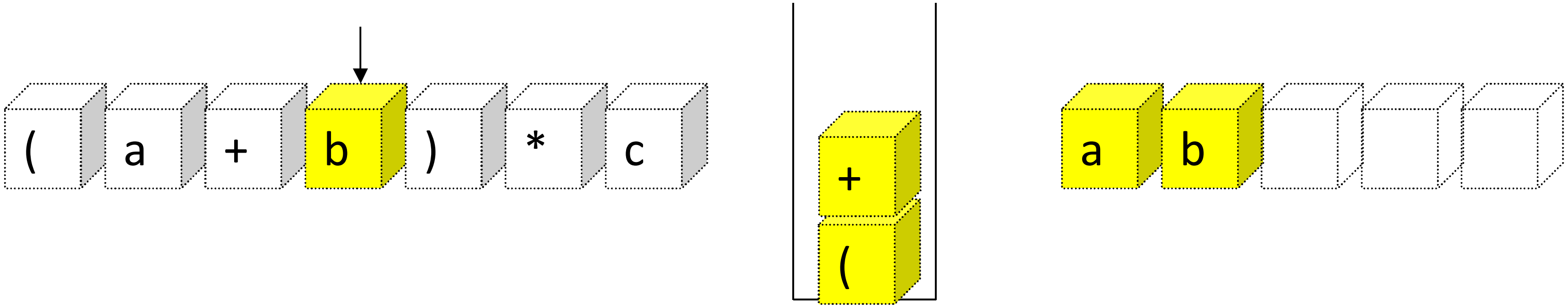
(a + b) * c



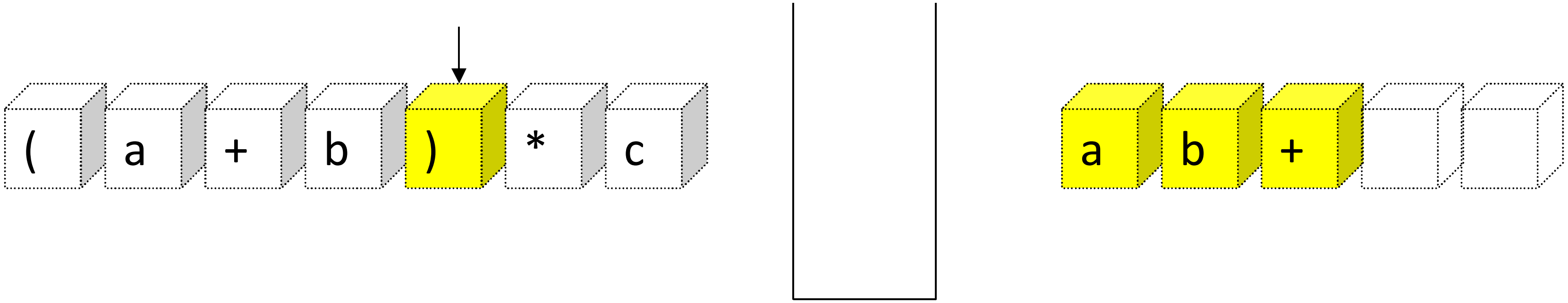
(a + b) * c



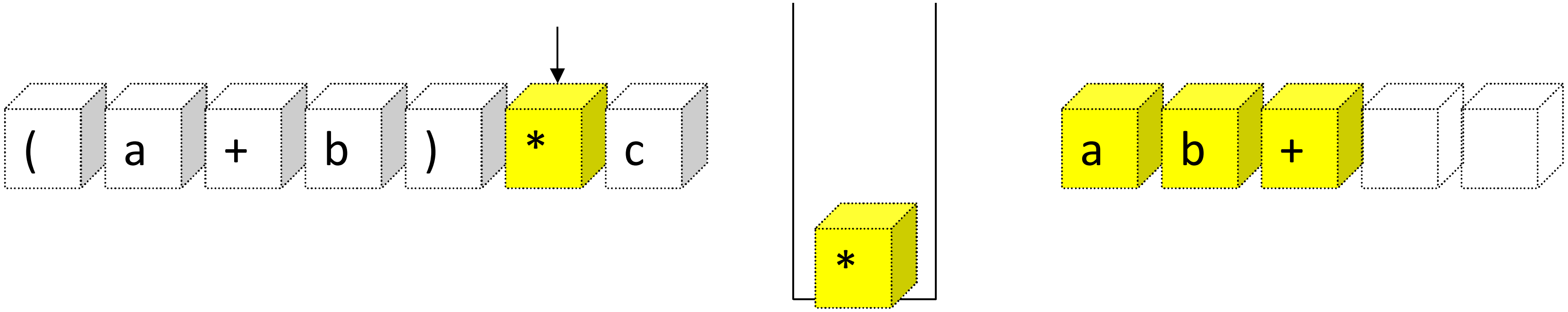
(a + b) * c



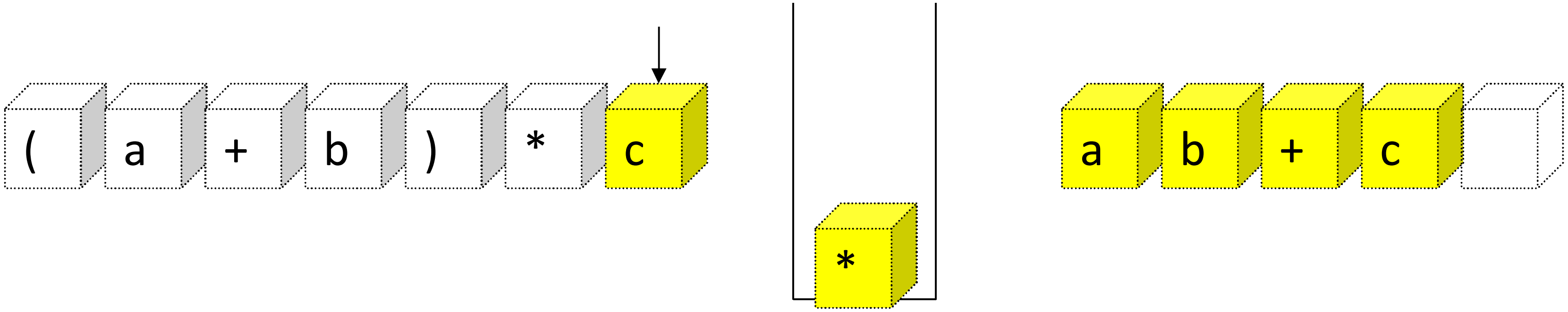
(a + b) * c



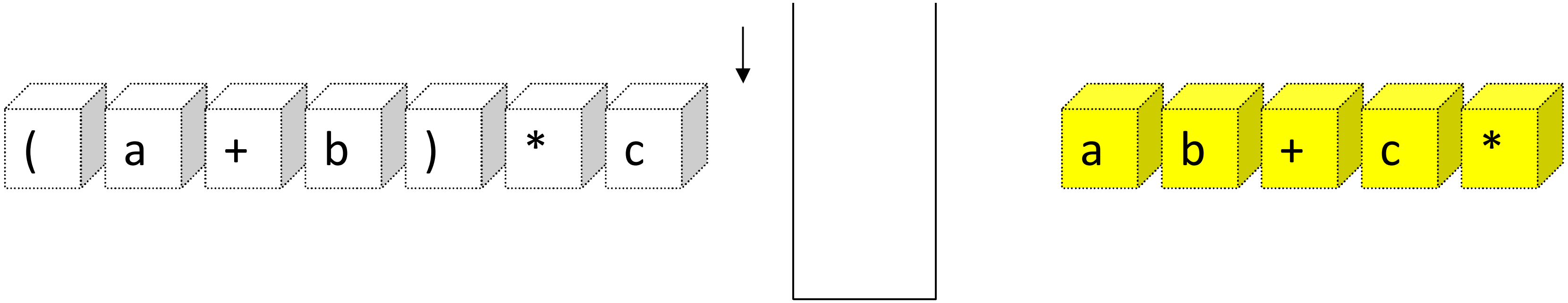
(a + b) * c



(a + b) * c



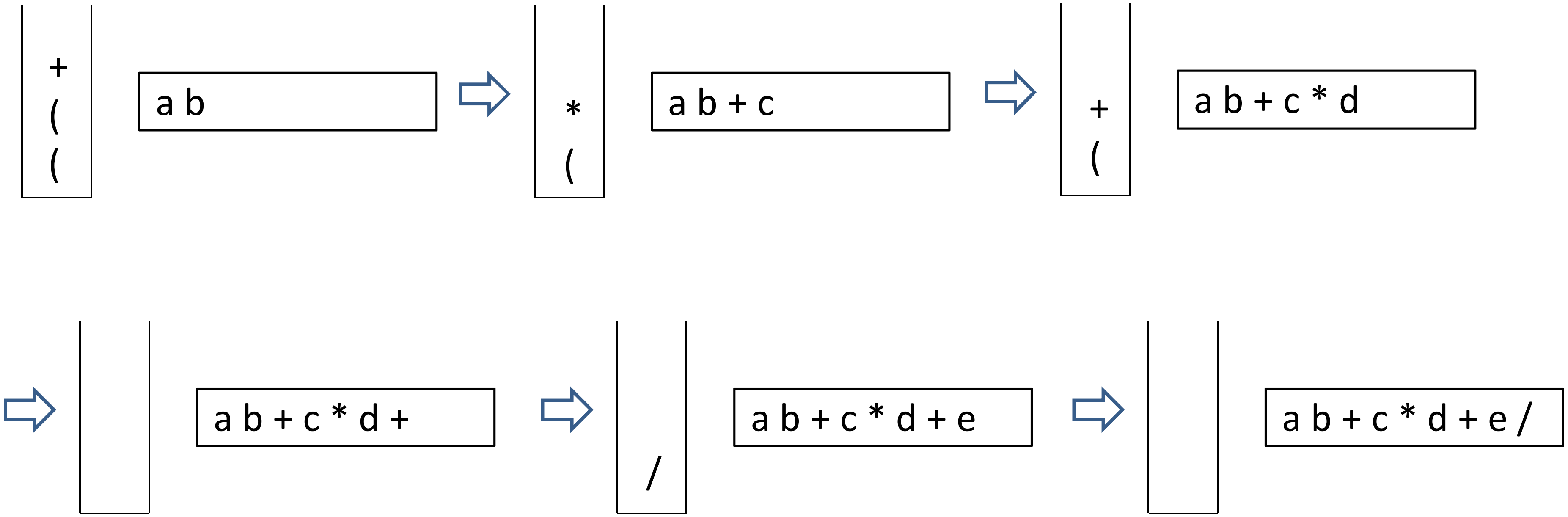
(a + b) * c



a + b * c

a * b + c

((a + b)*c + d) / e



+
(
(

a b

*
(

a b + c

+
(

a b + c * d

a b + c * d +

/

a b + c * d + e

a b + c * d + e /

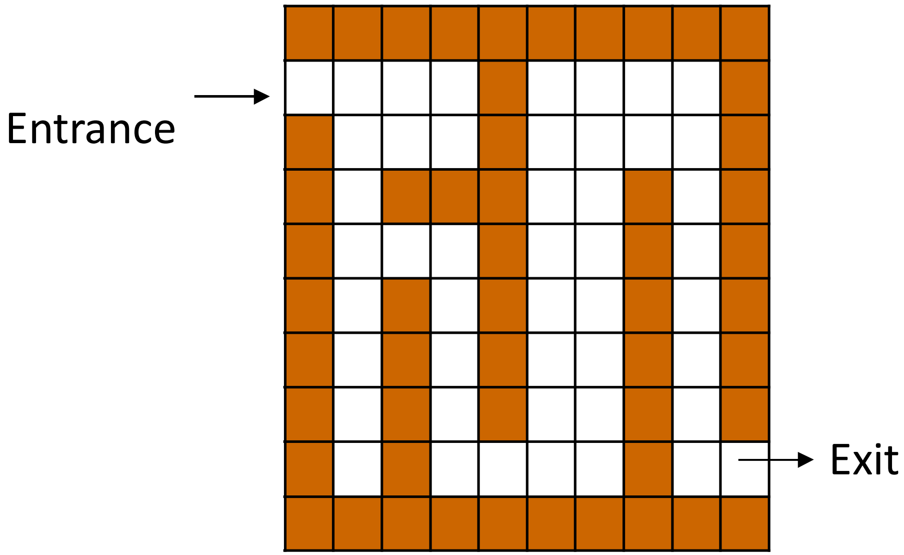
```
infix_to_postfix(exp)

Create and initialize stack s
while (exp has characters to process)
        ch ← Character to be processed next
        switch (ch)
        case operator:
                while (peek(s) priority ≥ ch priority)
                        e ← pop(s)
                        Output e
                push(s, ch);
                break;

        case Left parenthesis :
                push(s, ch);
                break;

        case Right parenthesis :
                e ← pop(s);
                while (e ≠ left parenthesis)
                        output e
                        e ← pop(s)
                break;

        case operand:
                Output ch
                break;

while (not is_empty(s))
        do e ← pop(s)
        Output e
```

```
void infix_to_postfix(char exp[])
{
    int i = 0;
    char ch, top_op;
    int len = strlen(exp);
    StackType s;

    init(&s);// Stack initialization
    for (i = 0; i<len; i++) {
        ch = exp[i];

        switch (ch) {
        case '+': case '-': case '*': case '/': // Operator
        // If the operator priority on the stack is greater than or equal to current operator
            while (!is_empty(&s) && (prec(ch) <= prec(peek(&s))))
                printf("%c", pop(&s));
            push(&s, ch);
            break;
        case '('://  Left parenthesis
            push(&s, ch);
            break;
        case ')'://  Right parenthesis
            top_op = pop(&s);
            // Output until the left parenthesis is encountered
            while (top_op != '(') {
                printf("%c", top_op);
                top_op = pop(&s);
            }
            break;
        default:// Operand
            printf("%c", ch);
            break;
        }
    }
    while (!is_empty(&s))// Output operators stored on the stack
        printf("%c", pop(&s));
}
//
void main(){
    infix_to_postfix("(2+3)*4+9"); }
```

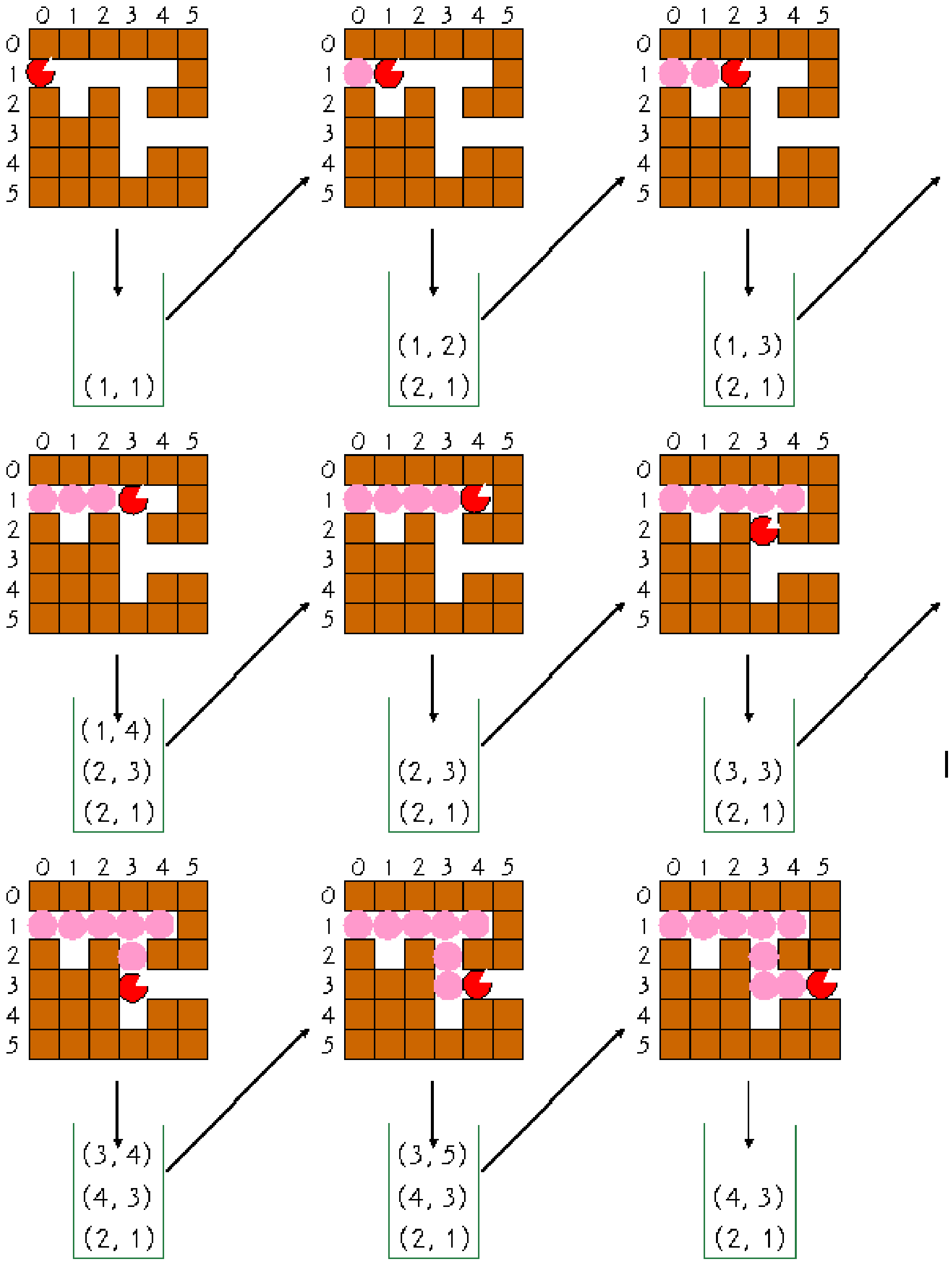
```
int prec(char op)
{
switch (op) {
case '(': case ')': return 0;
case '+': case '-': return 1;
case '*': case '/': return 2;
}
}
```

Maze Search Problem

- It stores the possible directions in the current position on the stack, and when it reaches a dead end, it takes the next seek position out of the stack.



```
    1  1  1  1  1  1  1  1  1  1
 →  0  0  0  0  1  0  0  0  0  1
    1  0  0  0  1  0  0  0  0  1
    1  0  1  1  1  0  0  1  0  1
    1  0  0  0  1  0  0  1  0  1
    1  0  1  0  1  0  0  1  0  1
    1  0  1  0  1  0  0  1  0  1
    1  0  1  0  1  m  0  1  0  1
    1  0  1  0  0  0  0  1  0  x  →
    1  1  1  1  1  1  1  1  1  1
```


0 1 2 3 4 5
(1, 1)
(1, 2)
(2, 1)
(1, 3)
(2, 1)
(1, 4)
(2, 3)
(2, 1)
(2, 3)
(2, 1)
(3, 3)
(2, 1)
(3, 4)
(4, 3)
(2, 1)
(3, 5)
(4, 3)
(2, 1)
(4, 3)
(2, 1)

Maze Search Algorithm

```
Initialize stack s, exit position x, and mouse position
while (if the current position is not an exit)
         do mark your current location as visited
         if (the top, bottom, left, and right positions of the current
            location have not yet been visited and can be visited)
                  then push the positions onto the stack
                  if (is_empty (s))
                           then failure
                  else
                           Take one position out of the stack and set as the current
                           location;
success;
```

```c
void push_loc(StackType *s, int r, int c)
{
        if (r < 0 || c < 0) return;
        if (maze[r][c] != '1' && maze[r][c] != '.') {
                element tmp;
                tmp.r = r;
                tmp.c = c;
                push(s, tmp);
        }
}

void main()
{
        int r, c;
        StackType s;

        init(&s);
        here = entry;

        while (maze[here.r][here.c] != 'x') {
                r = here.r;
                c = here.c;
                maze[r][c] = '.';
                push_loc(&s, r - 1, c);
                push_loc(&s, r + 1, c);
                push_loc(&s, r, c - 1);
                push_loc(&s, r, c + 1);
                if (is_empty(&s)) {
                        printf("Fail\n");
                        return;
                }
                else
                        here = pop(&s);
        }
        printf("Success\n");
}
```

```c
element here = { 1,0 }, entry = { 1,0 };

char maze[MAZE_SIZE][MAZE_SIZE] = {
{ '1', '1', '1', '1', '1', '1' },
{ 'e', '0', '1', '0', '0', '1' },
{ '1', '0', '0', '0', '1', '1' },
{ '1', '0', '1', '0', '1', '1' },
{ '1', '0', '1', '0', '0', 'x' },
{ '1', '1', '1', '1', '1', '1' },
};
```