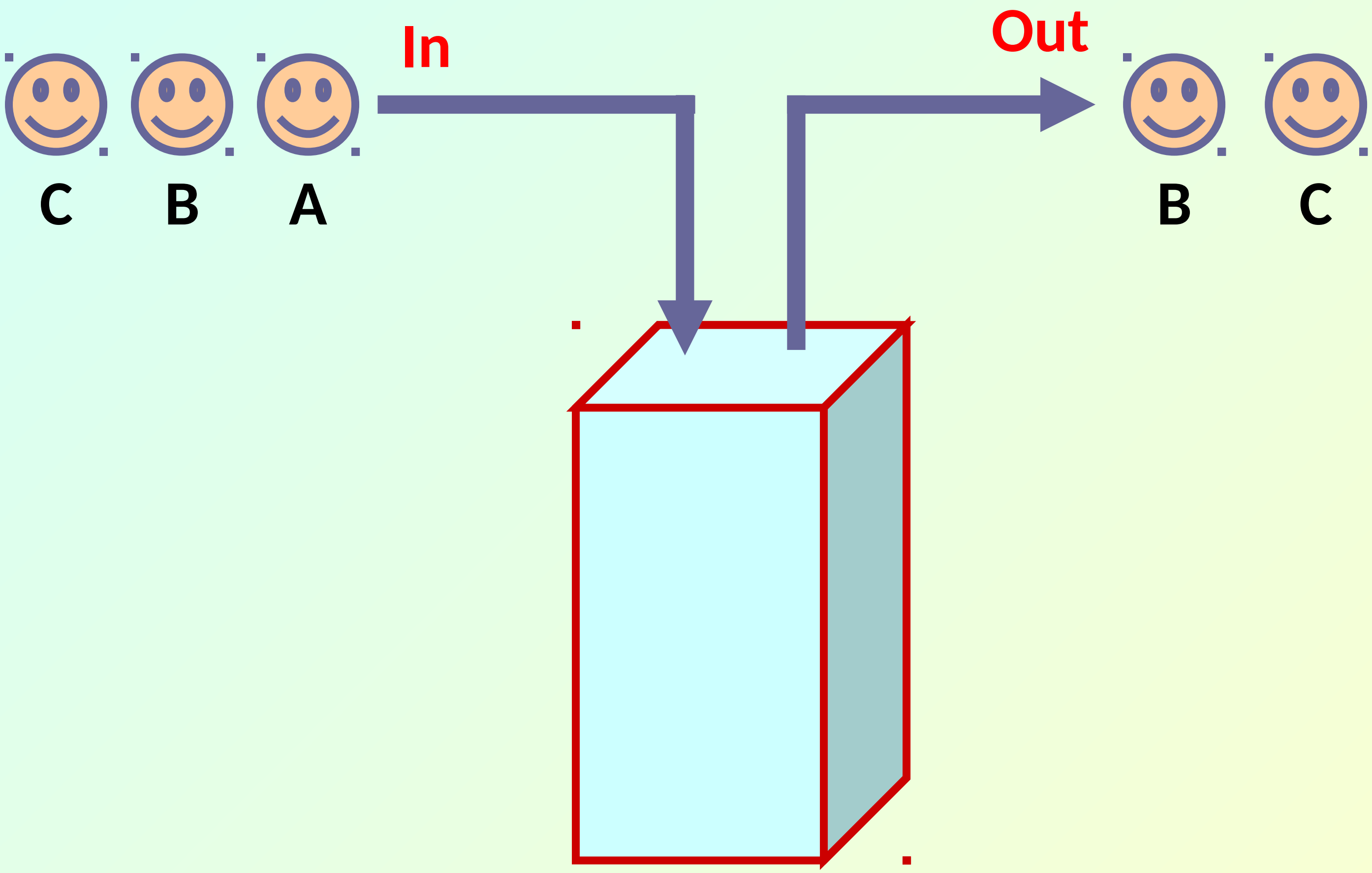
Stacks and Queues: Implementation

Visualization of a Stack (Last In First Out)



Stack Implementation

a) Using arrays
b) Using linked list

Basic Idea

- In the array implementation, we would:
  - Declare an array of fixed size (which determines the maximum size of the stack).
  - Keep a variable *top* which always points to the "top" of the stack.
    - Contains the array index of the "top" element.
- In the linked list implementation, we would:
  - Maintain the stack as a linked list.
  - A pointer variable *top* points to the start of the list.
  - The first element of the linked list is considered as the stack top.

Declaration

```
#define MAXSIZE 100

struct lifo
{
   int  st[MAXSIZE];
   int  top;
};

typedef struct lifo stack;
```

ARRAY

```
struct lifo
{
   int value;
   struct lifo *next;
};

typedef struct lifo stack;
```

LINKED LIST

Stack Creation

```
void create (stack *s)
{
    (*s).top = -1;

    /* s.top points to
       last element
       pushed in;
       initially -1 */
}
```

ARRAY

```
void create (stack **top)
{
    *top = NULL;

    /* top points to NULL,
       indicating empty
       stack            */
}
```

LINKED LIST

Pushing an element into the stack

```
void push (stack *s, int element)
  {
      if ((*s).top == (MAXSIZE-1))
      {
          printf ("\n Stack overflow");
          exit(-1);
      }
      else
      {
          (*s).top++;
          (*s).st[(*s).top] = element;
      }
  }
```

ARRAY

```c
void push (stack **top, int element)
{
    stack *new;

    new = (stack *) malloc(sizeof(stack));
    if (new == NULL)
    {
       printf ("\n Stack is full");
       exit(-1);
    }

    new->value = element;
    new->next = *top;
    *top = new;
}
```

LINKED LIST

Popping an element from the stack

```
int pop (stack *s)
  {
     if ((*s).top == -1)
     {
        printf ("\n Stack underflow");
        exit(-1);
     }
     else
     {
        return ((*s).st[(*s).top--]);
     }
  }
```

ARRAY

```c
int pop (stack **top)
{
    int t;
    stack *p;

    if (*top == NULL)
    {
        printf ("\n Stack is empty");
        exit(-1);
    }
    else
    {
        t = (*top)->value;
        p = *top;
        *top = (*top)->next;
        free (p);
        return t;
    }
}
```

LINKED LIST

Checking for stack empty

```
int isempty (stack s)
{
    if (s.top == -1)
            return (1);
    else
            return (0);
}
```

ARRAY

```
int isempty (stack *top)
{
    if (top == NULL)
            return (1);
     else
            return (0);
}
```

LINKED LIST

Checking for stack full

```
int isfull (stack s)
{
   if (s.top ==
            (MAXSIZE-1))

        return (1);
   else
        return (0);
}
```

ARRAY

- Not required for linked list implementation.
- In the `push()` function, we can check the return value of `malloc()`.
  - If -1, then memory cannot be allocated.

LINKED LIST

Example main function :: array

```
#include <stdio.h>
#define MAXSIZE 100

struct lifo
{
   int st[MAXSIZE];
   int  top;
};
typedef struct lifo stack;
```

```
main()
{
  stack A, B;
  create(&A);  create(&B);
  push(&A,10);
  push(&A,20);
```

```
  push(&A,30);
  push(&B,100);  push(&B,5);

  printf ("%d %d", pop(&A),
              pop(&B));

  push (&A, pop(&B));

  if (isempty(B))
    printf ("\nB is empty");
}
```

Example main function :: linked list

```
#include <stdio.h>
struct lifo
{
   int value;
   struct lifo *next;
};
typedef struct lifo stack;
```

```
main()
{
  stack *A, *B;
  create(&A); create(&B);
  push(&A,10);
  push(&A,20);
```

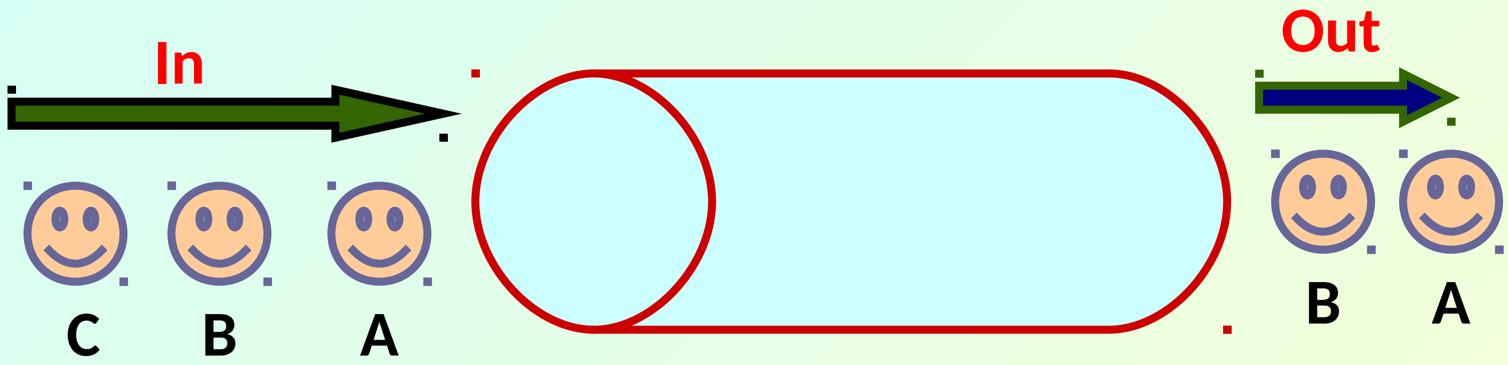
```
  push(&A,30);
  push(&B,100);  push(&B,5);

  printf ("%d %d",
        pop(&A), pop(&B));

  push (&A, pop(&B));

  if (isempty(B))
    printf ("\nB is empty");
}
```

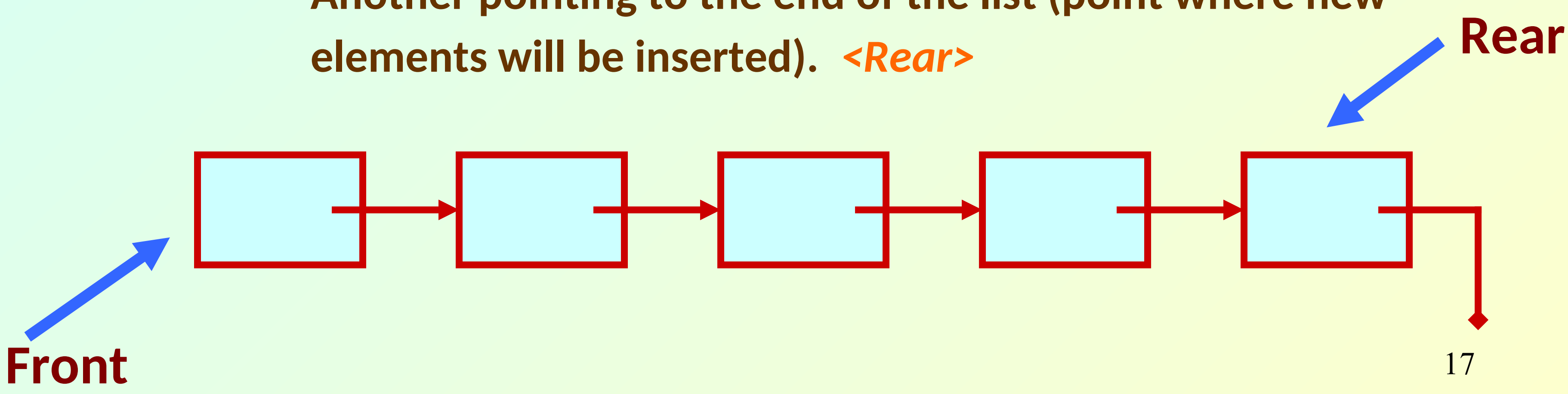
Visualization of a Queue
(First In First Out)



Queue Implementation using Linked List

Basic Idea

- **Basic idea:**
  - Create a linked list to which items would be added to one end and deleted from the other end.
  - Two pointers will be maintained:
    - One pointing to the beginning of the list (point from where elements will be deleted). *<Front>*
    - Another pointing to the end of the list (point where new elements will be inserted). *<Rear>*



Declaration

```
struct fifo {
                int  value;
                struct fifo *next;
            };
typedef struct fifo queue;

queue *front, *rear;
```

Creating a queue

```
void createq (queue **front, queue **rear)
{
    *front =  NULL;
    *rear  =  NULL;
}
```

Inserting an element in queue

```
void enqueue (queue **front, queue **rear, int x)
 {
     queue *ptr;
     ptr = (queue *) malloc(sizeof(queue));

     if (*rear == NULL)     /* Queue is empty */
     {
         *front = ptr;
         *rear  = ptr;
         ptr->value = x;
         ptr->next = NULL;
     }
     else                   /* Queue is not empty */
     {
         (*rear)->next = ptr;
         *rear = ptr;
         ptr->value = x;
         ptr->next = NULL;
     }
 }
```

Deleting an element from queue

```
int dequeue (queue **front, queue **rear)
{
    queue *old;   int k;

    if  (*front == NULL)                  /* Queue is empty */
        printf  ("\n Queue is empty");
    else if (*front == *rear)              /* Single element */
        {
            k = (*front)->value;
            free (*front);    front = rear = NULL;
            return (k);
        }
        else
        {
            k = (*front)->value;   old = *front;

            *front = (*front)->next;
            free (old);
            return (k);
        }
  }
```

Checking if empty

```
int isempty (queue *front)
{
    if (front == NULL)
        return (1);
    else
        return (0);
}
```

Example main function

```
#include <stdio.h>
struct fifo
{
    int value;
    struct fifo *next;
};
typedef struct fifo queue;
```

```
main()
{
  queue *Af, *Ar;
  createq (&Af, &Ar);
  enqueue (&Af,&Ar,10);
  enqueue (&Af,&Ar,20);
```

```
  enqueue(&Af,&Ar,30);

  printf ("%d %d",
        dequeue (&Af,&Ar),
        dequeue(&Af,&Ar));

  if (isempty(Af))
    printf ("\n Q is empty");
}
```

Some Applications of Stack

Applications of Stack

- Evaluation of expressions
  - Polish postfix and prefix notations
- Convert infix to postfix
- Parenthesis matching

Arithmetic Expressions
Polish Notation

What is Polish Notation?

- Conventionally, we use the operator symbol between its two operands in an arithmetic expression.

  `A+B        C–D*E        A*(B+C)`

  - We can use parentheses to change the precedence of the operators.
  - Operator precedence is pre-defined.
- This notation is called *INFIX notation*.
  - Parentheses can change the precedence of evaluation.
  - Multiple passes required for evaluation.

- **Polish notation**
  - Named after Polish mathematician Jan Lukasiewicz.
  - Polish POSTFIX notation
    - Refers to the notation in which the operator symbol is placed after its two operands.

      `AB+     CD*      AB*CD+/`
  - Polish PREFIX notation
    - Refers to the notation in which the operator symbol is placed before its two operands.

      `+AB     *CD      /*AB-CD`

How to convert an infix expression to Polish form?

- Write down the expression in fully parenthesized form. Then convert stepwise.
- Example:

```
A+(B*C)/D-(E*F)-G


(((A+((B*C)/D))-(E*F))-G)
```

- Polish Postfix form:

```
A B C * D / + E F * - G -
```

- Polish Prefix form:
  - Try it out ….

- **Advantages:**
  - No concept of operator priority.
    - Simplifies the expression evaluation rule.
  - No need of any parenthesis.
    - Hence no ambiguity in the order of evaluation.
  - Evaluation can be carried out using a single scan over the expression string.
    - Using stack.

Evaluation of a Polish Expression

- Can be done very conveniently using a stack.
  - We would use the Polish postfix notation as illustration.
    - Requires a single pass through the expression string from left to right.
    - Polish prefix evaluation would be similar, but the string needs to be scanned from right to left.

```
while (not end of string) do
 {
   a = get_next_token();
   if (a is an operand)
      push (a);
   if (a is an operator)
   {
      y = pop();  x = pop();
      push (x 'a' y);
   }
 }
 return (pop());
```

Evaluate: 10 6 3 - * 7 4 + -

Scan string from left to right:

```
10: push (10)        Stack: 10
6:  push (6)         Stack: 10 6
3:  push (3)         Stack: 10 6 3
-:  y = pop() = 3    Stack: 10 6
x = pop() = 6        Stack: 10
push (x-y)  Stack: 10 3
*:  y = pop() = 3    Stack: 10
x = pop() = 10       Stack: EMPTY
push (x*y)  Stack: 30
7:  push (7)         Stack: 30 7
4:  push (4)         Stack: 30 7 4
+:  y = pop() = 4    Stack: 30 7
x = pop() = 7        Stack: 30
push (x+y)  Stack: 30 11
-:  y = pop() = 11   Stack: 30
x = pop() = 30       Stack: EMPTY
push (x-y)  Stack: 19
```

Final result in stack

Converting an INFIX expression to POSTFIX

Basic Idea

- Let Q denote an infix expression.
  - May contain left and right parentheses.
  - Operators are:
    - Highest priority: ^ (exponentiation)
    - Then: * (multiplication), / (division)
    - Then: + (addition), – (subtraction)
  - Operators at the same level are evaluated from left to right.
- In the algorithm to be presented:
  - We begin by pushing a '(' in the stack.
  - Also add a ')' at the end of Q.

The Algorithm (Q:: given infix expression, P:: output postfix expression)

```
push (‘(’);
Add “)” to the end of Q;
while (not end of string in Q do)
{
  a = get_next_token();
  if (a is an operand) add it to P;
  if (a is ‘(’)  push(a);
  if (a is an operator)
  {
      Repeatedly pop from stack and add to P each
      operator (on top of the stack) which has the
      same or higher precedence than “a”;
      push(a);
  }
```

```
   if (a is ‘)’)
   {
        Repeatedly pop from stack and add to P each
        operator (on the top of the stack) until a
        left parenthesis is encountered;

        Remove the left parenthesis;
   }
}
```

Q: A + (B * C – (D / E ^ F) * G) * H)

Q	STACK	Output Postfix String P
A	(	A
+	(+	A
(	(+ (	A
B	(+ (	A B
*	(+ (*	A B
C	(+ (*	A B C
-	(+ (−	A B C *
(	(+ (− (	A B C *
D	(+ (− (	A B C * D
/	(+ (− (/	A B C * D
E	(+ (− (/	A B C * D E
^	(+ (− (/ ^	A B C * D E
F	(+ (− (/ ^	A B C * D E F
)	(+ (−	A B C * D E F ^ /

Q	STACK	Output Postfix String P
*	(+ (- *	A B C * D E F ^ /
G	(+ (- *	A B C * D E F ^ / G
)	(+	A B C * D E F ^ / G * -
*	(+ *	A B C * D E F ^ / G * -
H	(+ *	A B C * D E F ^ / G * - H
)		A B C * D E F ^ / G * - H * +

Parenthesis Matching

The Basic Problem

- **Given a parenthesized expression, to test whether the expression is properly parenthesized.**
  - **Whenever a left parenthesis is encountered, it is pushed in the stack.**
  - **Whenever a right parenthesis is encountered, pop from stack and check if the parentheses match.**
  - **Works for multiple types of parentheses**

    ( ), { }, [ ]

```
while (not end of string) do
 {
    a = get_next_token();
   if (a is '(' or '{' or '[')
        push (a);
    if (a is ')' or '}' or ']')
    {
       if (isempty()) {
         printf ("Not well formed");
         exit();
       }
        x = pop();
        if (a and x do not match) {
           printf ("Not well formed");
           exit();
       }
    }
 }
 if (not isempty())
   printf ("Not well formed");
```

```
Given expression: (a + (b – c) * (d + e))
Search string for parenthesis from left to right:
    (:  push ('(')      Stack: (
    (:  push ('(')      Stack: ( (
    ):  x = pop() = (   Stack: (        MATCH
    (:  push ('(')      Stack: ( (
    ):  x = pop() = (   Stack: (        MATCH
    ):  x = pop() = (   Stack: EMPTY    MATCH

Given expression: (a + (b – c)) * d)
Search string for parenthesis from left to right:
    (:  push ('(')      Stack: (
    (:  push ('(')      Stack: ( (
    ):  x = pop() = (   Stack: (        MATCH
    ):  x = pop() = (   Stack: EMPTY    MATCH
    ):  x = pop() = (   Stack: ?        MISMATCH
```

Some Other Applications

Other applications

- Reversing a string of characters.
- Generating 3-address code from Polish postfix (or prefix) expressions.
- Handling function calls and return
- Handling recursion