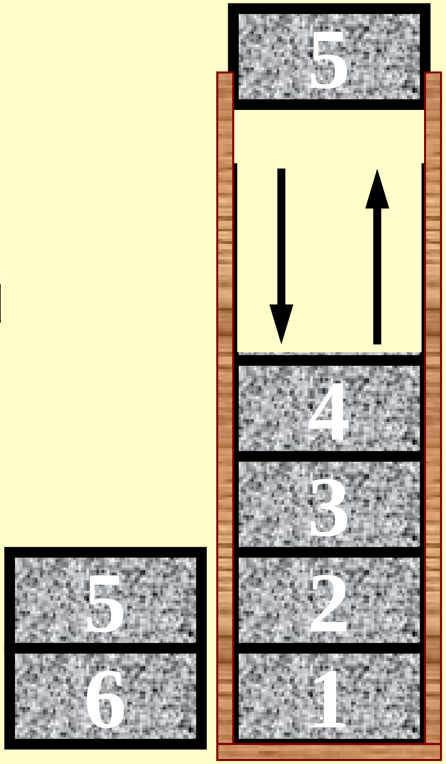
§ 3 The Stack ADT

1. ADT

A stack is a Last-In-First-Out (LIFO) list, that is, an ordered list in which insertions and deletions are made at the top only.



Objects: A finite ordered list with zero or more elements.

Operations:

- ☞ Int IsEmpty(Stack S);
- ☞ Stack CreateStack( );
- ☞ DisposeStack(Stack S);
- ☞ MakeEmpty(Stack S);
- ☞ Push(ElementType X, Stack S);
- ☞ ElementType Top(Stack S);
- ☞ Pop(Stack S);

Note: A Pop (or Top) on an empty stack is an error in the stack ADT.
Push on a full stack is an implementation error but not an ADT error.

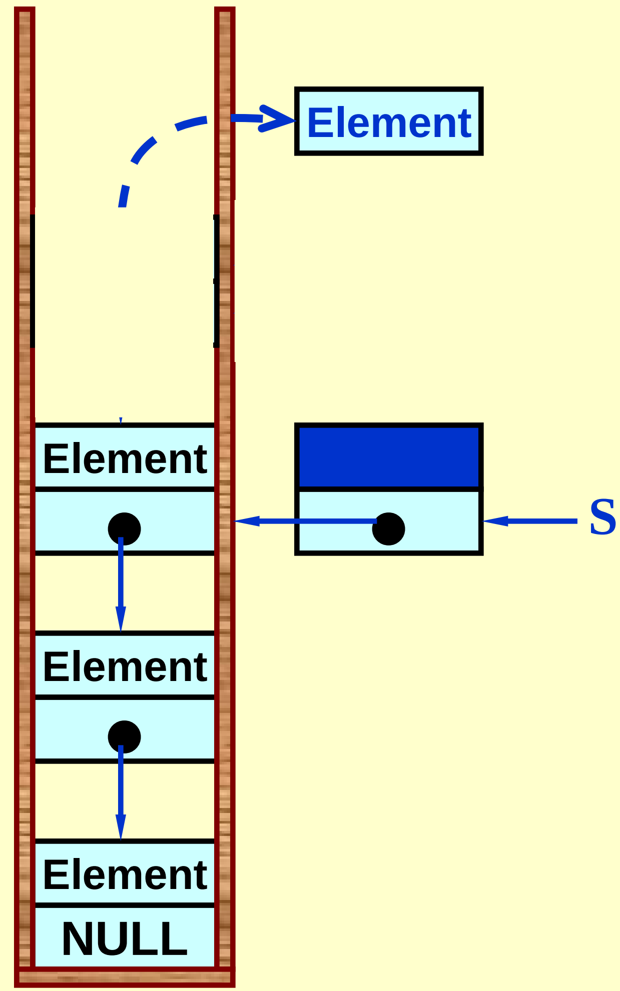
2. Implementations

➢ **Linked List Implementation (with a header node)**

Push:
① TmpCell->Next = S->Next
② S->Next = TmpCell

Top: return S->Next->Element

Pop:
① FirstCell = S->Next
② S->Next = S->Next->Next
③ free (FirstCell)



Array Implementation

```
struct  StackRecord {
        int     Capacity ;              /* size of stack */
        int     TopOfStack;             /* the top pointer */
        /* ++ for push, -- for pop, -1 for empty stack */
        ElementType  *Array;    /* array for stack elements */
} ;
```

Note: ① The stack model must be well encapsulated. That is, no part of your code, except for the stack routines, can attempt to access the Array or TopOfStack variable.
② Error check must be done before Push or Pop (Top).

Read Figures 3.38-3.52 for detailed implementations of stack operations.

3. Applications

✵ Balancing Symbols

Check if parenthesis ( ), brackets [ ], and braces { } are balanced.

```
Algorithm  {
    Make an empty stack S;
    while (read in a character c) {
        if (c is an opening symbol)
            Push(c, S);
        else if (c is a closing symbol) {
            if (S is empty)  { ERROR; exit; }
            else  {  /* stack is okay */
                if  (Top(S) doesn't match c)  { ERROR, exit; }
                else  Pop(S);
            }  /* end else-stack is okay */
        }  /* end else-if-closing symbol */
    } /* end while-loop */
    if (S is not empty)  ERROR;
}
```

$T(N) = O(N)$ where N is the length of the expression. This is an on-line algorithm.

✵ Postfix Evaluation

〖Example〗 An **infix** expression: $a + b * c - d / e$

A **prefix** expression: $- + a * b\ c / d\ e$

A **postfix** expression: $a\ b\ c * + d\ e / -$

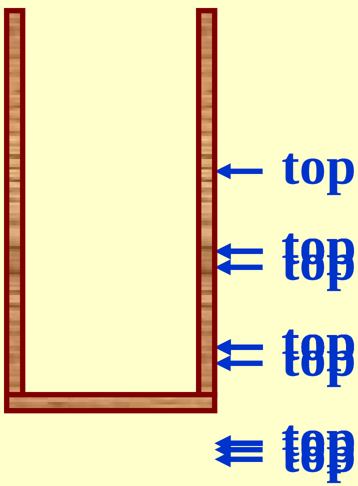
Reverse Polish notation

operand

operator with the highest precedence

operator

〖Example〗 6 2 / 3 – 4 2 * + = 8



Get token: 6 (operand)	Get token: 2 (operand)
Get token: / (operator)	Get token: 3 (operand)
Get token: – (operator)	Get token: 4 (operand)
Get token: 2 (operand)	Get token: * (operator)
Get token: + (operator)	Pop: 8

$T(N) = O(N)$. No need to know precedence rules.

❈ Infix to Postfix Conversion

〖Example〗 $a + b * c - d$ = $a\ b\ c * + d\ -$

Note:

- ➢ The order of operands is the same in infix and postfix.
- ➢ O… higher precedence appear before those …dence.

…t: a …

Get …	…: + (plus)
Get toke…	…t token: * (times)
Get token: c (operand)	Get token: − (minus)
Get token: d (operand)	

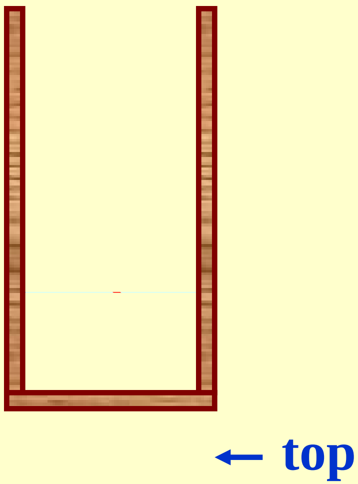
+ ≥ * ?

* + ≥ − ??

…top

〖Example〗 $a * (b + c) / d$ = $a\ b\ c + * \ d\ /$

Output: $a\ b\ c + * \ d\ /$



Get token: *a* (operand)	**Get token: * (times)**
Get token: ((lparen)	**Get token: *b* (operand)**
Get token: + (plus)	**Get token: *c* (operand)**
Get token:) (rparen)	**Get token: / (divide)**
Get token: *d* (operand)	

$* \geq ($?

NO?!

$T(N) = O (N)$

$* \geq /$?

Solutions:

① Never pop a (from the stack except when processing a) .

② Observe that when (is not in the stack, its precedence is the highest; but when it is in the stack, its precedence is the lowest. Define in-stack precedence and incoming precedence for symbols, and each time use the corresponding precedence for comparison.

Note: $a - b - c$ will be converted to $a\ b - c\ -$. However, 2^2^3 (2^{2^3}) must be converted to 2 2 3 ^ ^, not 2 2 ^ 3 ^ since exponentiation associates right to left.

❈ Function Calls -- System Stack

f p

nents

ne.

Recursion can always be **completely removed**.
Non recursive programs are generally **faster** than equivalent recursive programs.
However, recursive programs are in general much **simpler and easier to understand**.

```
void  PrintL     List L )
{
   if ( L != NU  L )  {
      PrintEle ment ( L->Element );
      PrintList( L->next );
   }
}  /* a bad use of recursion */
```

```
void  PrintList ( List L )
{
top:  if ( L != NULL )  {
         PrintElement ( L->Element );
         L = L->next;
         goto top; /* do NOT do this */
      }
}  /* compiler removes recursion */
```

§ 4 The Queue ADT

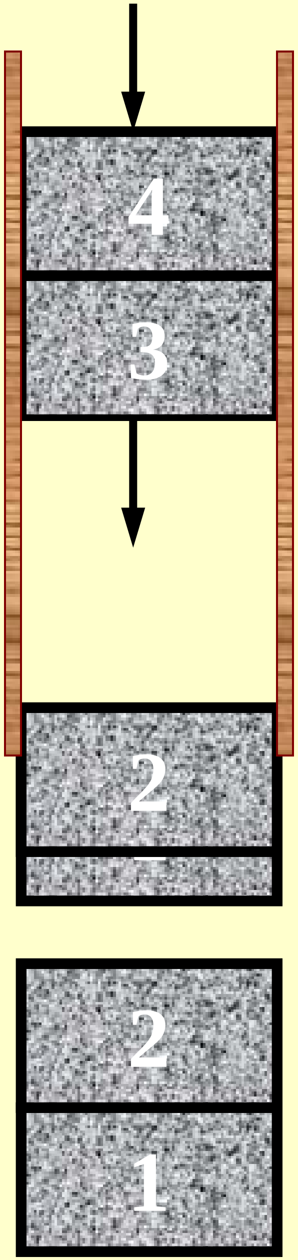
1. ADT

A queue is a First-In-First-Out (FIFO) list, that is, an ordered list in which insertions take place at one end and deletions take place at the opposite end.

Objects: A finite ordered list with zero or more elements.

Operations:

- ☞ **int IsEmpty(Queue Q);**
- ☞ **Queue CreateQueue( );**
- ☞ **DisposeQueue(Queue Q);**
- ☞ **MakeEmpty(Queue Q);**
- ☞ **Enqueue(ElementType X, Queue Q);**
- ☞ **ElementType Front(Queue Q);**
- ☞ **Dequeue(Queue Q);**

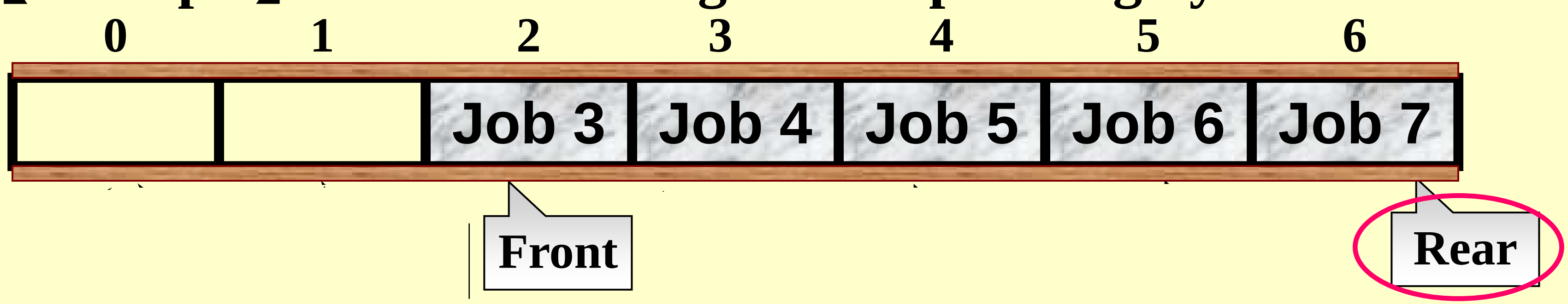


2. Array Implementation of Queues

(Linked list implementation is trivial)

```
struct  QueueRecord {
        int     Capacity ;   /* max size of queue */
        int     Front;       /* the front pointer */
        int     Rear;        /* the rear pointer */
        int     Size;  /* Optional - the current size of queue */
        ElementType  *Array;    /* array for queue elements */
 } ;
```

〖Example〗 Job Scheduling in an Operating System

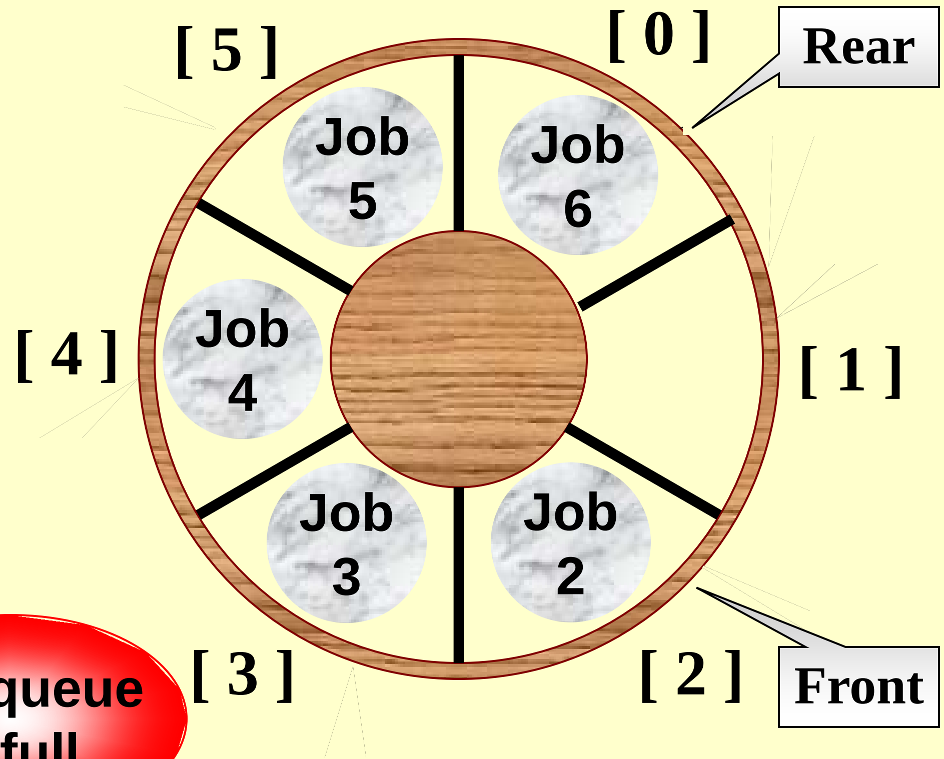


Enqueue Job 1	Enqueue Job 2	Enqueue Job 3	Dequeue Job 1
Enqueue Job 4	Enqueue Job 5	Enqueue Job 6	Dequeue Job 2
Enqueue Job 7	Enqueue Job 8		

Circular Queue:

Question: Why is the queue announced full while there is still a free space left?

The queue is full



Note: Adding a **Size** field can avoid wasting one empty space to distinguish "full" from "empty". Do you have any other ideas?

Bonus Problem 1

LRU-K

(2 points)

Due: Monday, June 17th, 2024 at 10:00pm

The problem can be found and submitted at
https://pintia.cn/