Stack and Queue

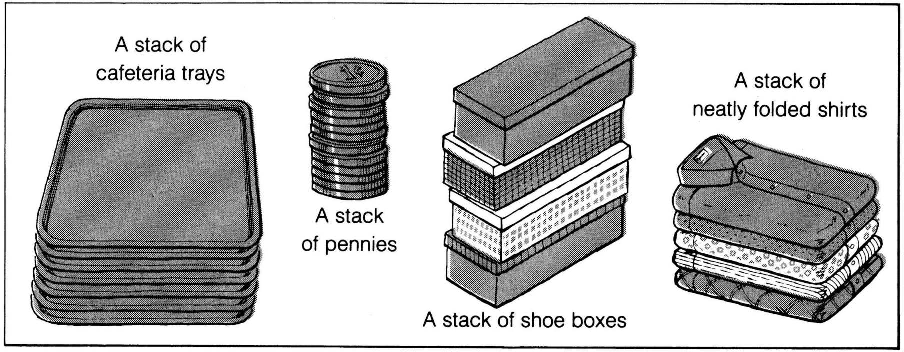
Data Structures CS2001

Overview

- Introduction to Stack Data structure
- Static vs Dynamic stack
- Implementation details of stack
- Applications of stack
- Introduction to Queue Data structure
- Implementation details of Queue
- Applications of Queue

Introduction to the Stack

- Stack: a LIFO (last in, first out) data structure
- Examples:
  - plates in a cafeteria serving area
  - return addresses for function calls



Stack Basics

- Stack is usually implemented as a list, with additions and removals taking place at one end of the list
- The active end of the list implementing the stack is the top of the stack
- Stack types:
  - Static – fixed size, often implemented using an array
  - Dynamic – size varies as needed, often implemented using a linked list

Stack Operations and Functions

Operations:

- push: add a value at the top of the stack
- pop: remove a value from the top of the stack

Functions:

- isEmpty: true if the stack currently contains no elements
- isFull: true if the stack is full; only useful for static stacks

Push Operation

- *Function*: Adds newItem to the top of the stack.
- *Preconditions*: Stack has been initialized and is not full.
- *Postconditions*: newItem is at the top of the stack.

Pop Operation

- *Function*: Removes topItem from stack and returns it in item.
- *Preconditions*: Stack has been initialized and is not empty.
- *Postconditions*: Top element has been removed from stack and item is a copy of the removed element.

Stack Overflow and Underflow

Stack overflow

- The condition resulting from trying to push an element onto a full stack.

```
if(!stack.IsFull())
   stack.Push(item);
```

Stack underflow

- The condition resulting from trying to pop an empty stack.

```
if(!stack.IsEmpty())
   stack.Pop(item);
```

Static Stack Implementation

- Uses an array of a fixed size
- Bottom of stack is at index 0. A variable called top tracks the current top of the stack

```
const int STACK_SIZE = 3;
char s[STACK_SIZE];
    int top = 0;
```

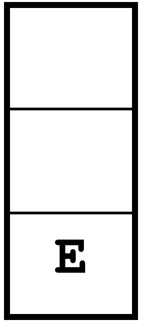
top is where the next item will be added

Array Implementation Example

This stack has max capacity 3, initially top = 0 and stack is empty.

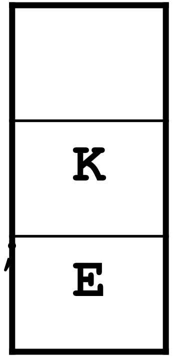
`push('E');`

top is 1



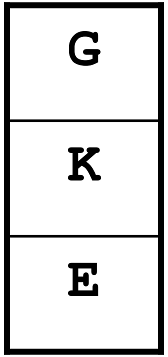
`push('K');`

top is 2



`push('G');`

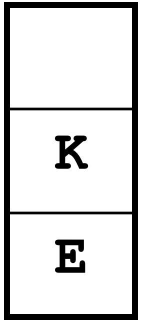
top is 3



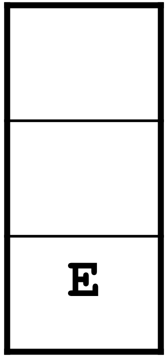
Stack Operations Example

After three pops, **`top == 0`** and the stack is empty

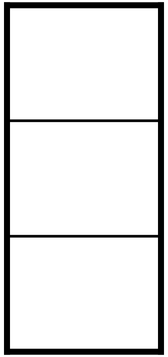
`pop();`
(remove **`G`**)



`pop();`
(remove **`K`**)



`pop();`
(remove **`E`**)



Array Implementation

```
char s[STACK_SIZE];
int top=0;
```

To check if stack is empty:

```
bool isEmpty()
{
  if (top == 0)
 return true;
 else return false;
}
```

```
char s[STACK_SIZE];
int top=0;
```

To check if stack is full:

```
bool isFull()
{
   if (top == STACK_SIZE)
     return true;
   else return false;
}
```

To add an item to the stack

```
void push(char x)
 {
  if (isFull())
     {error(); exit(1);}
  // or could throw an exception
  s[top] = x;
  top++;
 }
```

To remove an item from the stack

```
void pop(char &x)
 {
  if (isEmpty())
     {error(); exit(1);}
  // or could throw an exception
 top--;
  x = s[top];
}
```

Dynamic Stacks

- Implemented as a linked list
- Can grow and shrink as necessary
- Can't ever be full as long as memory is available

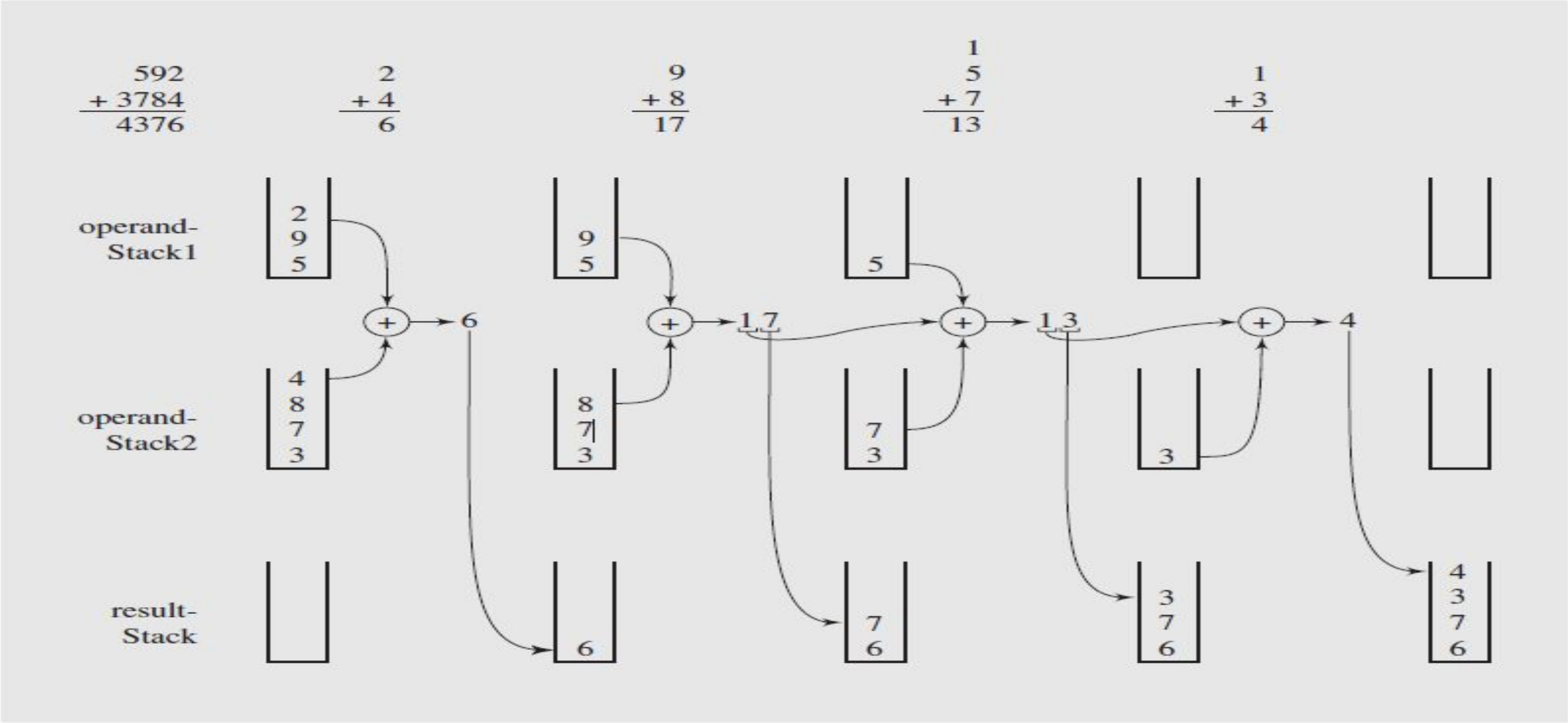
Discussion

- How we will implement a dynamic stack using linked list?

Application of Stacks

- Adding large numbers
- Infix to Postfix Conversion
- Evaluating Postfix notations
- Bracket Matching
- Tower of Hanoi
- Identifying Palindromes
- Searching in Graphs (Depth First Search)
- Undo Redo
- Backtracking

Adding large numbers



Infix to Postfix conversion

Examples of infix to prefix and post fix

Infix	Postfix	Prefix
A+B	AB+	+AB
(A+B) * (C + D)	AB+CD+*	*+AB+CD
A-B/(C*D^E)	ABCDE^*/-	-A/B*C^DE

Rules of Operator Precedence

Operator(s)	Precedence & Associativity
( )	Evaluated first. If **nested (embedded)**, innermost first. If on same level, left to right.
* / %	Evaluated second. If there are several, evaluated left to right
+ -	Evaluated third. If there are several, evaluated left to right.
=	Evaluated last, right to left.

Infix to postfix conversion

- **Rules to follow:**
  - Print operands as they arrive.
  - If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
  - If the incoming symbol is a left parenthesis, push it on the stack.
  - If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.
  - If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
  - If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.
  - If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.
  - At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

Example Expression

Suppose we want to convert 2*3/(2-1)+5*3 into Postfix form,

Expression	Stack	Output
2	Empty	2
*	*	2
3	*	23
/	/	23*
(	/(	23*
2	/(	23*2
-	/(-	23*2
1	/(-	23*21
)	/	23*21-
+	+	23*21-/
5	+	23*21-/5
*	+*	23*21-/53
3	+*	23*21-/53
	Empty	23*21-/53*+

Postfix Evaluation

- Rules to follow
  - 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

Postfix Evaluation

```
             23*21-/53*+


Read         Stack
2            2
3            23
*            6
2            62
1            621
-            61
/            6
5            65
3            653
*            615
+            21

             21
```

Implementation Details

```
int prec(char c)
{
    if(c == '^')
    return 3;
    else if(c == '*' || c == '/')
    return 2;
    else if(c == '+' || c == '-')
    return 1;
    else
    return -1;
}
```

If the scanned character is operand

```
if((s[i] >= 'a' && s[i] <= 'z')||(s[i] >= 'A' && s[i] <= 'Z'))
        ns+=s[i];
```

If the scanned character is '('

```
 if(s[i] == '(')
        st.push('(');
```

If the scanned character is Operator

```
If(s[i] == Operator) then
        while(!st.IsEmpty && prec(s[i]) <= prec(st.top()))  do
            char c = st.top();
            st.pop();
            ns += c;
        end while
        st.push(s[i]);
End if
```

If the scanned character is ‘)’

```
if(s[i] == ')') then
      while(st.top() != '(') do
              char c = st.top();
              st.pop();
             ns += c;
          end while
          if(st.top() == '(') then
              char c = st.top();
              st.pop();
          end if
End if
```

Postfix Evaluation Algorithm

```
Algorithm postfix (String S)
Input: An expression
Output: Result of an expression
for each character ch in the postfix expression, do
    if ch is an operator ⊙ , then
      a := pop first element from stack
      b := pop second element from the stack
        res := b ⊙ a
      push res into the stack
    else if ch is an operand, then
      add ch into the stack
    end if
return element of stack top
End Procedure
```

Tasks to be implemented

Infix to prefix conversion

- Expression = **(A+B^C)*D+E^5**
- Step 1. Reverse the infix expression.
  5^E+D*)C^B+A(
  Step 2. Make Every ‘(‘ as ‘)’ and every ‘)’ as ‘(‘
  5^E+D*(C^B+A)
  Step 3. Convert expression to postfix form.
  5E^DCB^A+*+
  Step 4. Reverse the expression.
  +*+A^BCD^E5

Question:

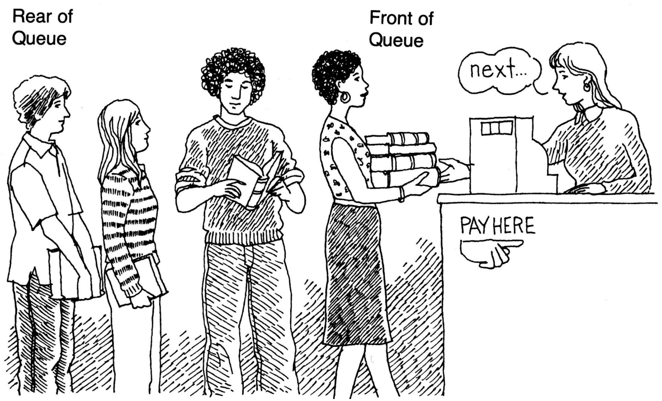
a/b-(c+d)-e --/ab+cde

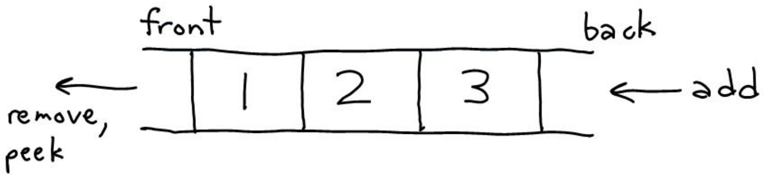
Adding Large numbers using two additional stacks

$$\begin{array}{r} 644 \\ +238 \end{array}$$

Queue

- It is an ordered group of homogeneous items of elements.
- Queues have two ends:
  - Elements are added at one end.
  - Elements are removed from the other end.
- The element added first is also removed first (**FIFO**: First In, First Out).





Queue Locations and Operations

- rear: position where elements are added
- front: position from which elements are removed
- enqueue: add an element to the rear of the queue
- dequeue: remove an element from the front of a queue

Enqueue Operation

- *Function*: Adds newItem to the rear of the queue.
- *Preconditions*: Queue has been initialized and is not full.
- *Postconditions*: newItem is at rear of queue.

Dequeue Operation

- *Function*: Removes front item from queue and returns it in item.
- *Preconditions*: Queue has been initialized and is not empty.
- *Postconditions*: Front element has been removed from queue and item is a copy of removed element.

Queue overflow

- The condition resulting from trying to add an element onto a full queue.

```
if(!q.IsFull())
  q.Enqueue(item);
```

Queue underflow

- The condition resulting from trying to remove an element from an empty queue.

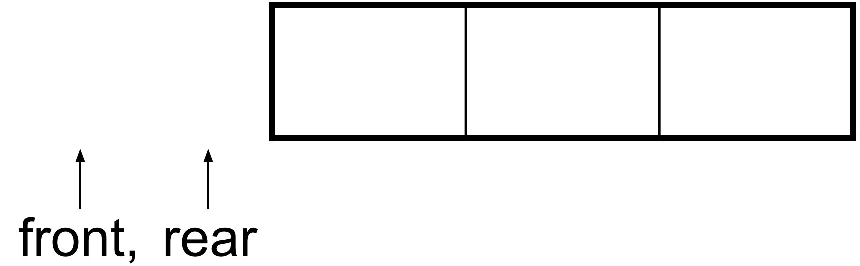
```
if(!q.IsEmpty())
  q.Dequeue(item);
```

Queue Applications

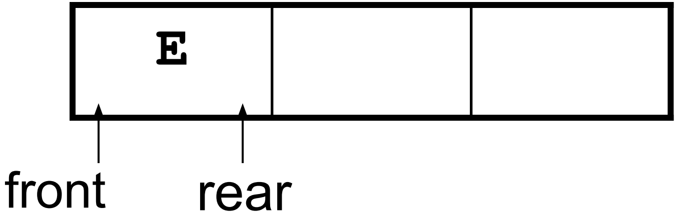
- Operating systems:
  - queue of print jobs to send to the printer
  - queue of programs / processes to be run
  - queue of network data packets to send
- Programming:
  - modeling a line of customers or clients
  - storing a queue of computations to be performed in order
- Real world examples:
  - people on an escalator or waiting in a line
  - cars at a gas station (or on an assembly line)

Implementation Details

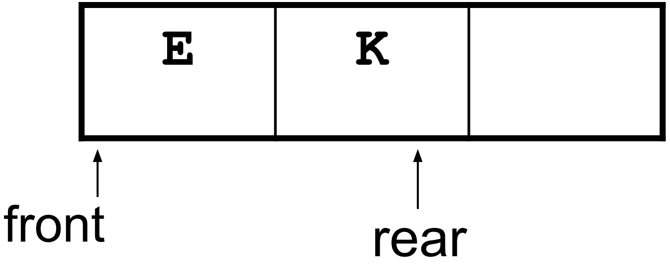
An empty queue that can hold **`char`** values:



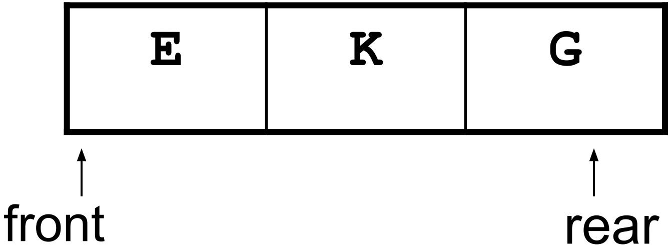

```
enqueue('E');
```



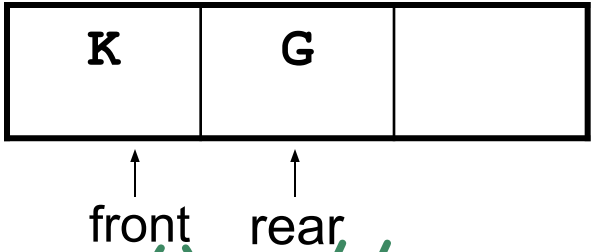
```
enqueue('K');
```



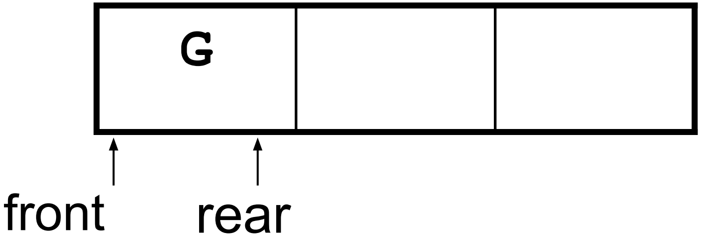

```
enqueue('G');
```



```
dequeue(); // remove E
```




```
dequeue(); // remove K
```



Array Implementation Issues

- In the preceding example, Front never moves.
- Whenever **`dequeue`** is called, all remaining queue entries move up one position. This takes time.
- Alternate approach:
  - Circular array: **`front`** and **`rear`** both move when items are added and removed. Both can 'wrap around' from the end of the array to the front if warranted.
- Other conventions are possible

- Variables needed
  - `const int QSIZE = 100;`
  - `char q[QSIZE];`
  - `int front = -1;`
  - `int rear = -1;`
  - `int number = 0; //how many in queue`

Check if queue is empty

```
bool isEmpty()
{
    if (number > 0)
        return false;
    else
        return true;
}
```

Check if queue is full

```
bool isFull()
{
    if (number < QSIZE)
        return false;
    else
        return true;
}
```

- To enqueue, we need to add an item **`x`** to the rear of the queue

```
if(!isFull)
{ rear = (rear + 1) % QSIZE;
// mod operator for wrap-around
  q[rear] = x;
   number ++;
 }
```

- To dequeue, we need to remove an item **x** from the front of the queue

```
if(!isEmpty)
{   front = (front + 1) % QSIZE;
    x = q[front];
    number--;
}
```

- `enqueue` moves `rear` to the right as it fills positions in the array
- `dequeue` moves `front` to the right as it empties positions in the array
- When `enqueue` gets to the end, it wraps around to the beginning to use those positions that have been emptied
- When `dequeue` gets to the end, it wraps around to the beginning use those positions that have been filled

- Enqueue wraps around by executing

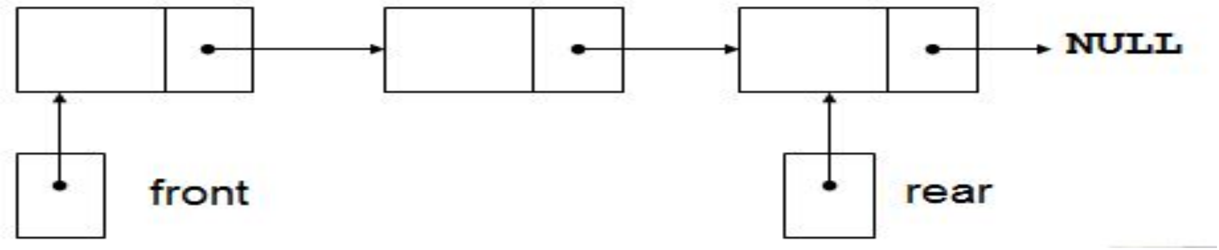
```
rear = (rear + 1) % QSIZE;
```

- Dequeue wraps around by executing

```
front = (front + 1) % QSIZE;
```

Dynamic Queues

- Like a stack, a queue can be implemented using a linked list
- Allows dynamic sizing, avoids issue of wrapping indices



Discussion

- How to implement a queue using linked list??

Summary

- This lecture is summarized as:
  - The concepts of queue and stack are discussed in detail along with their implementation.
  - The underlying implementation of array as well as linked list data structure is covered.
  - Real life applications are also discussed.