Stack and Queue

Yu-Tai Ching
Department of Computer Science
National Chiao Tung University

Template

- Purpose: make classes and functions more reusable.
- Data type as a parameter to a class or a function

template <class T>
void $SelectionSort$ ($T * a$, const n)
{
. . . ;
}
float $farray[100]$;
int $intarray[250]$;
. . . ;
$SelectionSort(farray, 100)$;
$SelectionSort(intarray, 250)$;

- Stack and queue can be considered a “container”
- You can put different object into a container
- implemented using template
- you don’t have to implement different stack or queue for different objects (data type).

Stack and Queus

- Two kinds of data structures frequently used in computer programs.
- Special cases of ordered list.
- Stack: an ordered list in which insertion (push) and deletion (pop) are made at one end called the *top*, *LIFO*, figure 3.1.
- Queue: an ordered list, insertion and deletion take place at different ends, insertion at the *rear* end and deletion at the *front* end, *FIFO*, figure 3.4.

System Stack

- used by a program at runtime to process function calls.
- A function is invoked, program creates a *structure-activation record* or a *stack frame*, and put it on top of the system stack.
- Figure 3.2, main invokes function a.

- To implement a stack
  - Use an array $stack[]$
  - The first (bottom) element stored at $stack[0]$, second $stack[1]$, ...
  - The ith stored at $stack[i-1]$.
  - A variable top points to the top of the stack.
  - initially, top is -1, indicating empty stack.

ADT of stack

private:

$T*$ stack;

int top;

int $capacity$;

template <class T>

{

class $Stack$

public:

$Stack$ (int $StackCapacity = 10$) ;

bool $IsEmpty()$ const;

$T\&\ Top()$ const;

void $Push$(const $T\&\ item$) ;

void $Pop()$;

}

Queue

- We can use an array $queue[]$ to implement a Queue.
- front stored in $queue[0]$, next in $queue[1]$, ..., a $rear$ pointer points to the end of queue.
- Figure 3.5 (a) a 3 elements queue,
- delete A, to make front be at $queue[0]$, move data around (b).
- Dequeue, $\Theta(n)$, inqueue can be done in $\Theta(1)$.
- Try to make both dequeue and inqueue done in $\Theta(1)$ time,
- need another pointer $front$. Figure 3.6.

Queue

- Let $capacity$ be the capacity of the array $queue[]$,
- If $front > 0$ and $rear = capacity - 1$, there are space but we cannot do inqueue.
- So we have to shift elements in the queue to the left, time complexity depends on the number of element in the queue.
- If we are allowed to have a wrap around array, inqueue and dequeue can be done in $\Theta(1)$.

Circular Queue

- $front$ points to one position counterclockwise from the location of the front element in the queue.
- $rear$ points to the last one.
- Figure 3.8.
- Initially, $front == rear$, an insertion: $(rear + 1)\%capacity$ then insert,
- a deletion, move $front$ to the next, $(front + 1)\%capacity$.
- Queue empty condition, $front == rear$.
- In Figure 3.8, after a sequency of insertions, we are going to have a full queue, at that time $front == rear$.

Circular Queue

- We can either don’t use a space in cicular implementation, or
- use another variable to distinguish queue empty or queue full, or
- apply a "dynamic table" approach.
- If a push make the queue full, double the queue size.
- than copy every one into the new and larger queue.

A Maze Problem

$$
\begin{array}{cccccc}
0 & 1 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 1 \\
1 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 & 0 & 1 \\
0 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 1 \\
0 & 1 & 1 & 1 & 1 & 0
\end{array}
$$

- Given a 0/1 matrix, 1 stands for wall, 0 stands for path,
- an entrance and an exit,
- is there a path (consists of all 0s) from entrance to exist?
- Need a stack, at a place (i, j), if there is a neighbor is 0, there is a way out, take that step and push that move to stack.
- reach a place, but there is no way out, back a step (we know how to get back from the stack).
- repeat until the exit is reached or the stack is empty.

Evaluation of Expression

- in high level programming language, given an arithmetic expression, generate machine-language instructions and evaluate the expression.
- The expression $X = A/B - C + D * E - A * C$; this could have many meaning, depending on the order of the operators performed.
- Expression is made up of operands, operators, and delimiters, Above expression, A, B, C, D, and E are operands., /, +, * are operators.
- Priority defined with operators, parentheses override the priority.
- $X = ((A/(B - C + D)) * (E - A) * C$

Operators

priority	operator
1	uniary operator, !
2	*, /, %
3	+, -
4	$<$, $<=$, $>=$, $>$
5	==, !=
6	&&
7	||

- Parentheses override the priority, innermost parenthesized expression first.
- same priority, evaluate expression from left to right, tie broken rule.
- The above expression should be $X = (((A/B) - C) + (D * E)) - (A * C)$.

Postfix Notation

- *infix expression* in your C++ codes, convert it into *postfix expression*
- infix expression: the operators come in-between the operands.
- Postfix expression: the operators appear after its operands,
- infix $A * B/C$, postfix $AB * C/$.

Evaluate a Postfix expression

- infix: $A/B - C + D * E - A * C$,
- postfix: $AB/C - DE * +AC * -$

operation	postfix
	$AB/C - DE * +AC * -$
$T_1 = A/B$	$T_1C - DE * +AC * -$
$T_2 = T_1 - C$	$T_2DE * +AC * -$
$T_3 = D * E$	$T_2T_3 + AC * -$
$T_4 = T_2 + T_3$	$T_4AC * -$
$T_5 = A * C$	T_4T_5-
$T_6 = T_4 - T_5$	T_6

Implementation

- Need a stack
- Input is a postfix expression,
- scan from left to right
- if we have an operand, push the opernad,
- if we have an operator, pop out two operands from the stack, perform the operation, push the result back to the stack.
- At the end of expression, pop out the result from the stack.

Infix to Postfix

- To design an algorithm to convert infix to postfix, observe that the order of the operands in both forms is the same.
  1. Fully parenthesize the expression.
  2. Move all operators so that they replace their corresponding right parentheses.
  3. Delete all parentheses.
- For $A/B - C + D * E - A * C$
- $((((A/B) - C) + (D * E)) - (A * C))$
- then move the operator and remove the parentheses.
- Note, not a computer algorithm.

Infix to Postfix

- Since order of operands in infix exp and postfix exp is the same, when we scan the infix exp from left to right, if the token is an operand, we output the token to the output.
- We only have to process the operators.
- Rule, “stack the operator having higher priority” (priority higher than the operator on the top of the stack).
- Rule, “Pop out the operator on the top of the stack when the incoming one has equal to or less than priority”.
- Rule, dump the stack at the end of string.

$$A + B * C$$

- We should get $ABC*+$

next token	stack	output
none	empty	none
A	empty	A
+	+	A
B	+	AB
*	+*	AB
C	+*	ABC

- End of string, dumpt the stack,
- we get $ABC*+$

- $A * (B + C) * D$
- We should get $ABC + *D*$
- Rule, there are in-coming priority and in-stack priority for "(".
  - in-coming priority for "(" is high (to make sure that "(" can be pushed).
  - once "(" is in the stack, the priority is low (so that operators in the parentheses can be pushed).
- A token ")" comes, pop out the stack until "(" is popped.

$$A * (B + C) * D$$

next token	stack	output
none	empty	none
A	empty	A
*	*	A
(	*(	A
B	*(	AB
+	*(+	AB
C	*(+	ABC
)	*	$ABC+$
*	*	$ABC+*$
D	*	$ABC+*D$
done	empty	$ABC+*D*$

Queue in Computer Science

- Need to define an ordering that who can have the CPU.
- a job queue
- The priority may not be identical between processes, the problem can be solved by using the heap structure (priority queue).
- We consider only one priority.
- Can you implement a queue using two stacks?

- Queue
  - Inqueue
  - Dequeue
- Stack
  - Empty
  - Push
  - Pop

To implement a queue Q by using two stacks S1 and S2

- Q.Inqueue
  - S1.Push
- Q.Dequeue
  - if (S2.Empty) while (not S1.Empty) S2.Push(S1.Pop); S2.Pop;
  - else S2.Pop;