Lecture Notes on
Data Structures

M1522.000900

Seoul National University

Fall 2022

Part II

Stack and Queue

Stack ADT

- Data: a finite ordered list with zero or more elements.
- Operations:
  1. *CreateStack* : $n \to Stack$, (Create an empty stack whose (max) size is n.)
  2. *IsEmpty* : $Stack \to Boolean$,
  3. *IsFull* : $Stack \to Boolean$,
  4. *Push* : $Stack, x \to Stack$, (If the stack is full, return OVERFLOW. Otherwise, add a new element x at the top of the stack.)
  5. *Pop* : $Stack \to x$, (If the stack is empty, return EMPTY. Otherwise, remove and return the top element.)
  6. *Top* : $Stack \to x$. (If the stack is empty, return EMPTY. Otherwise, return the top element without removing it.)

Example 1

Reverse a list $a_1, a_2, \ldots, a_n$.

It is trivial if the elements are stored in an array.

What if they are stored in a (singly) linked list?

```
S = CreateStack(n);
while(the list is not empty) {
   read an element x;
   Push(S,x);
}
while(S is not empty)
   print Pop(S);
```

Example 2

Evaluate a postfix expression. For example, a postfix expression `1 2 + 3 4 - * 5 + 6 *` is equivalent to an infix expression `((((1 + 2) * (3 - 4)) + 5) * 6)`.

```
S = CreateStack(n);
while(input is not empty) {
   read a token x;
   if (x is a number) Push(S,x);
   else if (x is an operator) {
      a = Pop(S);
      b = Pop(S);
      Push(S, result of (b x a));
   }
}
print Pop(S);
```


Problem 1

Convert a fully-parenthesized infix expression into a postfix expression.

Problem 2

In Example 2, how should the value of n be determined so that memory usage can be minimized without causing stack overflow?

Implementations of the Stack ADT

From the fact that the ADT specification defines data as a finite ordered list, we can think of using either an array or a linked list as a baseline data structure for the stack ADT.

Array implementation of Stack

	Array implementation
CreateStack(n)	`int stack[n];` `int top=-1; int size=n;`
IsEmpty(S)	`return (top==-1);`
IsFull(S)	`return (top==size-1);`
Push(S,x)	`if IsFull(S) return OVERFLOW;` `stack[++top] = x;`
Pop(S)	`if IsEmpty(S) return EMPTY;` `return stack[top--];`

Linked List implementation of Stack

	Linked list implementation
CreateStack(n)	`Link top=null;` `int numelt=0; int size=n;`
IsEmpty(S)	`return (top==null);`
IsFull(S)	`return (numelt==size);`

Push(S,x)	`if IsFull(S) return OVERFLOW;`
	`L = new Link node;`
	`L.key = x; L.next = top;`
	`top = L; numelt++;`

Pop(S)	`if IsEmpty(S) return EMPTY;`
	`y = top.key; tmp = top;`
	`top = top.next; free tmp;`
	`numelt--; return y;`

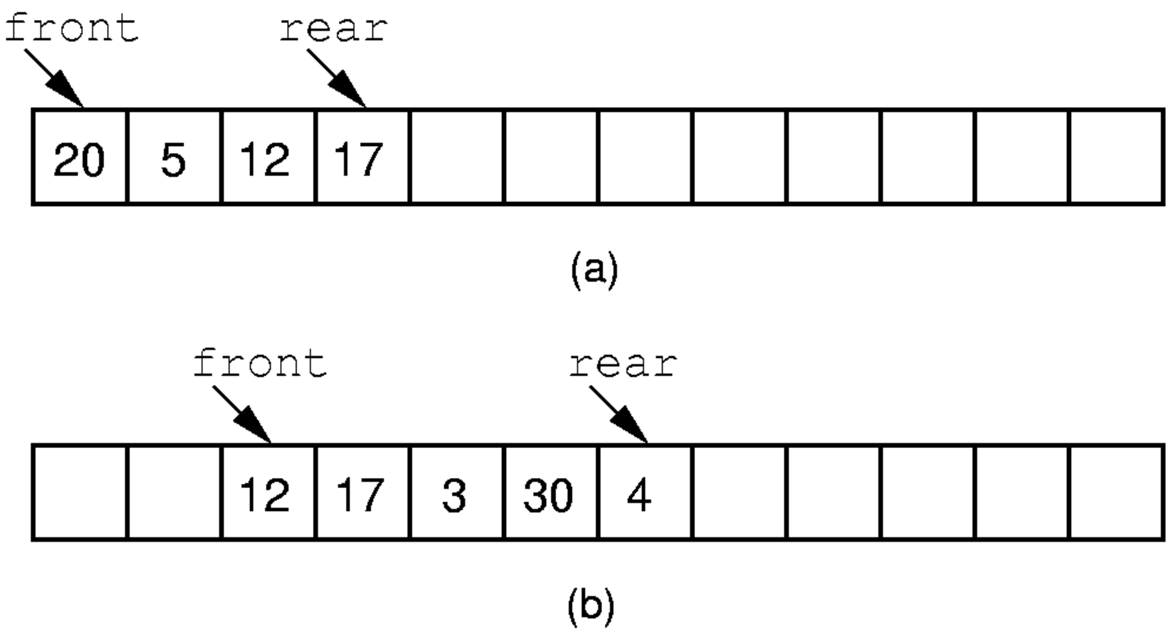
Comparison of the Stack Implementations

- Time
  1. Array : each operation is $\mathcal{O}(1)$.
  2. Linked list : each operation is $\mathcal{O}(1)$.
- Space
  1. Array: statically allocated; $n \times E$, where E is the size of space for a stack element.
  2. Linked list: dynamically allocated; $numelt \times (E + P)$, where P is the size of space for a pointer.
  3. Break-even point: $numelt = \frac{E}{E+P} \times n$.

Queue ADT

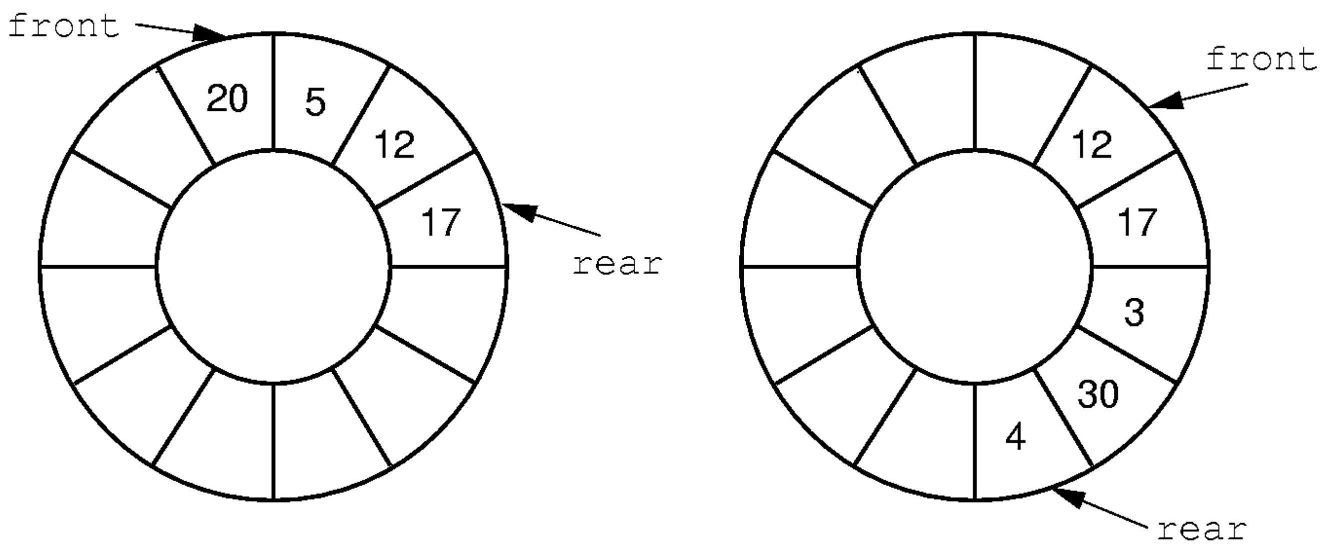
- Data: a finite ordered list with zero or more elements.
- Operations:
  1. *CreateQueue* : $n \rightarrow$ *Queue*, (Create an empty queue whose (max) size is n.)
  2. *IsEmpty* : *Queue* $\rightarrow$ *Boolean*,
  3. *IsFull* : *Queue* $\rightarrow$ *Boolean*,
  4. *Enqueue* : *Queue*, $x \rightarrow$ *Queue*, (If the queue is full, return OVERFLOW. Otherwise, add a new element x at the rear of the queue.)
  5. *Dequeue* : *Queue* $\rightarrow x$, (If the queue is empty, return EMPTY. Otherwise, remove and return the element at the front of the queue.)

Array implementation of Queue



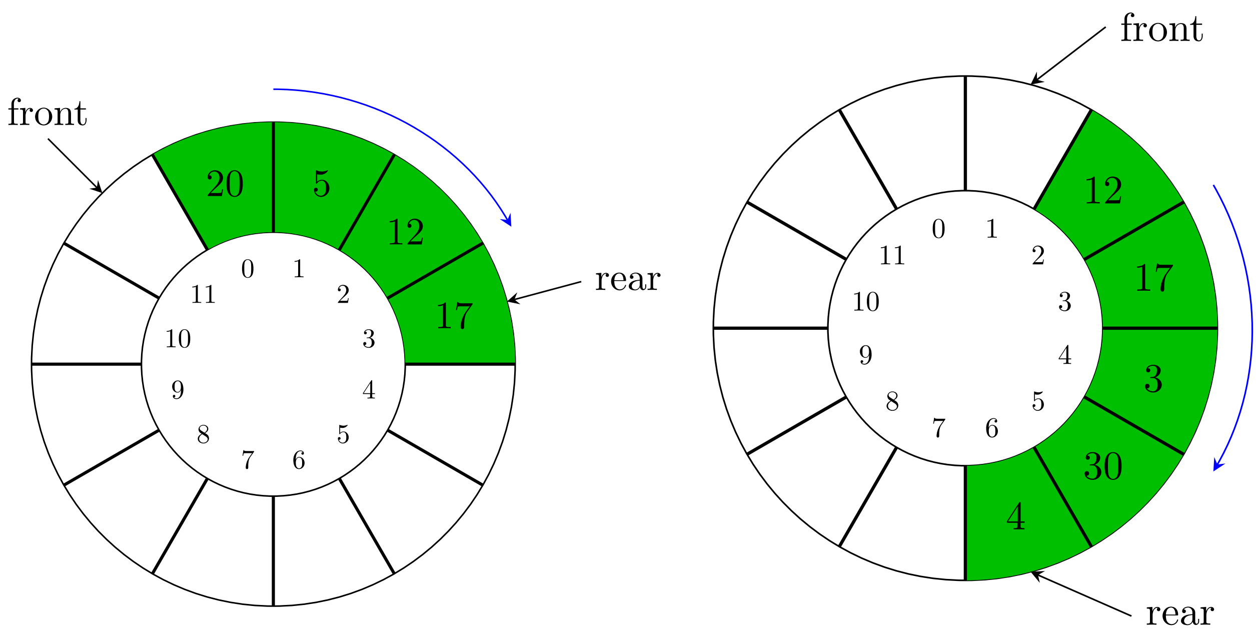
Comments on the array implementation of queue:

- An array, which is a linear data structure, is used as if it is a circular data structure by wrapping around the front and rear pointer variables.



(a) (b)

Alternatively,



More comments on the array implementation of queue:

- An array of $n+1$ elements is used for a queue that can store up to n elements. The reason is that we should be able to distinguish an empty queue from a full queue, as we discussed in class. Another way of looking at this issue is how we represent different states of a queue, more specifically, how we represent the number of elements in a queue consistently with the variables *front* and *rear*. Think about these:
  1. The number of different states of a queue is $n+1$, because a queue can store $0, 1, 2, 3, \ldots,$ or n elements.
  2. If we use an array of n elements instead, the *relative difference* between *front* and *rear*, $(rear - front)\%size$, can be one of $0, 1, 2, 3, \ldots, n-1$, which implies we can represent only n different states a queue can be in.

	Array implementation
CreateQueue(n)	`int queue[n+1]; int size=n+1;` `int front=0, rear=0;`
IsEmpty(Q)	`return (front==rear);`
IsFull(Q)	`return ((rear+1)%size==front);`

Enqueue(Q,x)	`if IsFull(Q) return OVERFLOW;`
	`rear = (rear+1)%size;`
	`queue[rear] = x;`

Dequeue(Q)	`if IsEmpty(Q) return EMPTY;`
	`front = (front+1)%size;`
	`return queue[front];`

Problem 3

The array implementation of Queue does not keep track of the number of elements explicitly. How would you determine the number of elements stored in a Queue?

Linked List implementation of Queue

	Linked list implementation
CreateQueue(n)	`Link front=null, rear=null;` `int numelt=0; int size=n;`
IsEmpty(Q)	`return (front==null);`
IsFull(Q)	`return (numelt==size);`

Enqueue(Q,x)	`if IsFull(Q) return OVERFLOW;`
	`L = new Link node;`
	`L.key = x; numelt++;`
	`if (front == null) {`
	`    front = rear = L;`
	`} else {`
	`rear.next = L; rear = L; }`

Dequeue(Q)	`if IsEmpty(Q) return EMPTY;`
	`y = front.key; numelt--;`
	`tmp = front;`
	`front = front.next; free tmp;`
	`if (front==null) rear=null;`
	`return y;`