Data Type Stack & Queue

Stack and Queue

Both stack and queue are important data types used in computing. They are essentially lists of data with restricted entry and exit orderings.

Use of Stack

1. Most modern computer architecture supports hardware stack to implement recursive programming, exception handling, system call implementation.
2. Compiler uses stack for syntax checking and semantic action.

Basic Operations on a Stack

init() $\longrightarrow$ s:Stack, empty stack.

isEmpty(s) $\longrightarrow$ b:Boolean

top(s) $\longrightarrow$ v:Data, if s is not empty
error, otherwise

push(s, v) $\longrightarrow$ t:Stack

pop(s) $\longrightarrow$ t:Stack, if s is not empty
error, otherwise

A Few Axioms of the Stack Operations

$$\begin{aligned}
\text{isEmpty}(\text{init}()) &= \mathit{true} \\
\text{isEmpty}(\text{push}(s, v)) &= \mathit{false} \\
\text{pop}(\text{init}()) &= \mathit{error} \\
\text{pop}(\text{push}(s, v)) &= s \\
\text{top}(\text{init}()) &= \mathit{error} \\
\text{top}(\text{push}(s, v)) &= v
\end{aligned}$$

The axioms define the operations.

Stack: an Example

$$
\begin{aligned}
\mathrm{init}() &\rightarrow [],\ \text{empty stack} \\
\mathrm{push}([], 5) &\rightarrow [5] \\
\mathrm{push}([5], 7) &\rightarrow [5\ 7] \\
\mathrm{push}([5\ 7], 3) &\rightarrow [5\ 7\ 3] \\
\mathrm{pop}([5\ 7\ 3]) &\rightarrow [5\ 7] \\
\mathrm{push}([5\ 7], 10) &\rightarrow [5\ 7\ 10] \\
\mathrm{top}([5\ 7\ 10]) &\rightarrow 10
\end{aligned}
$$

LIFO list.

Basic Operations on a Queue

$$
\begin{aligned}
\text{init}() &\rightarrow q\text{:Queue, an empty queue} \\
\text{isEmpty}(q) &\rightarrow b\text{:Boolean} \\
\text{front}(q) &\rightarrow d\text{:Data, if q is not empty} \\
&\rightarrow \text{error, otherwise} \\
\text{add}(q, d) &\rightarrow p\text{:Queue} \\
\text{delete}(q) &\rightarrow p\text{:Queue, if q is notempty} \\
&\rightarrow \text{error, otherwise}
\end{aligned}
$$

Note

The operation add is also called insert, enqueue; similarly the operation delete is also called dequeue.

A Few Axioms of the Queue Operations

$$
\begin{aligned}
\text{isEmpty}(\text{init}()) &= \textit{true} \\
\text{isEmpty}(\text{add}(q, d)) &= \textit{false} \\
\text{delete}(\text{init}()) &= \textit{error} \\
\text{delete}(\text{add}(q, d)) &= q, \text{ if } q \text{ is empty} \\
&= \text{add}(\text{delete}(q), d), \text{ otherwise} \\
\text{front}(\text{init}()) &= \textit{error} \\
\text{front}(\text{add}(q, v)) &= v, \text{ if } q \text{ is empty} \\
&= \text{front(q), otherwise}
\end{aligned}
$$

Queue: an Example

$$
\begin{aligned}
\text{init}() &\rightarrow [\,],\ \text{empty queue} \\
\text{add}([\,], 5) &\rightarrow [5] \\
\text{add}([5], 7) &\rightarrow [5\ 7] \\
\text{add}([5\ 7], 3) &\rightarrow [5\ 7\ 3] \\
\text{delete}([5\ 7\ 3]) &\rightarrow [7\ 3] \\
\text{add}([7\ 3], 10) &\rightarrow [7\ 3\ 10] \\
\text{front}([7\ 3\ 10]) &\rightarrow 7
\end{aligned}
$$

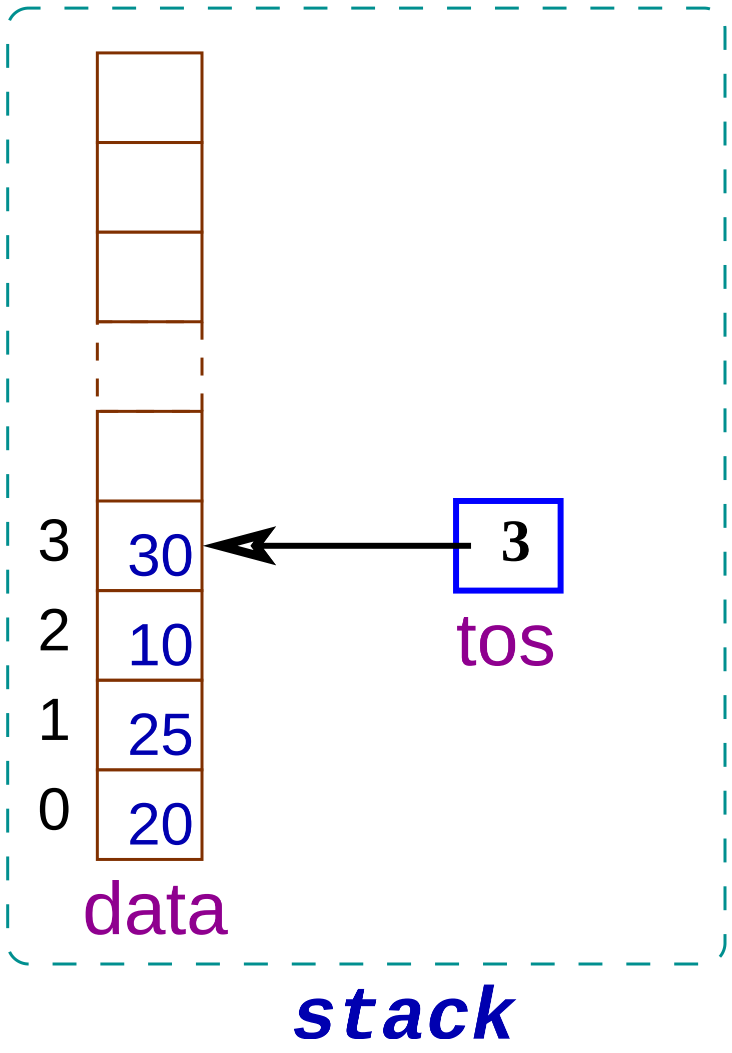
FIFO list.

Implementation of Stack & Queue

A stack (queue) may grow to any arbitrary size. So an ideal stack (ideal queue) cannot be implemented in a real machine (finite capacity). We can implement an approximation of a stack (queue).

Stack on an Array: data representation

```
typedef struct {
        int data[SIZE];
        int tos; // top of stack
} stack;
```


3
2
1
0
30
10
25
20
data
3
tos
stack

Stack on an Array: operations

```
void init(stack *) ;
int push(stack * , int) ;
int pop(stack *) ;
int top(stack *, int *) ;
int isEmpty(stack *) ;
int isFull(stack *) ;  // For finite size
```

Note

The data type stack is a big structure and we shall always pass a pointer to it to avoid large volume of data copy in parameter passing (call-by value).

stack.h

```
#ifndef _STACK_H
#define _STACK_H
#define SIZE 200
#define ERROR 1
#define OK 0
typedef struct {
        int data[SIZE];
        int tos;
} stack ;

void init(stack *) ;
int push(stack * , int) ;
```

```
int pop(stack *) ;
int top(stack *, int *) ;
int isEmpty(stack *) ;
int isFull(stack *) ;  // For finite size
#endif
```

Implementation: stack.c

```
#include "stack.h"
void init(stack *s) // stack.c
{ s->tos = -1;}

int isFull(stack *s)
{   return s->tos == SIZE-1;}

int isEmpty(stack *s)
{   return s->tos == -1;}

int push(stack *s, int n) {
    if(isFull(s)) {
```

```
        printf("The STACK is full\n");
        return ERROR ;
    }
    s->data[++s->tos]=n;
    return OK ;
}

int Pop(stack *s) {
    if(isEmpty(s)) {
        printf("The STACK is empty\n");
        return ERROR ;
    }
    s -> tos-- ;
    return OK ;
```

```
}

int Top(stack *s , int *val) {
    if(isEmpty(s)) {
       printf("The STACK is empty\n") ;
       return ERROR ;
    }
    *val = (s -> data[s -> tos]) ;
    return OK ;
}
```

Compiling the datatype

```
$ cc -Wall -c stack.c
```

We get the object module `stack.o`. We can construct library from it.

User Program: testStack.c

```
#include <stdio.h>
#include "stack.h"
int main() // testStack.c
{
    stack s ;
    int x , err , val ;
    char c ;

    init(&s);

    printf(" 'U' for push (U 15)\n 'O' for pop\n 'T' for to
    printf(" 'E' for exit :\n");
```

```
while((c = getchar()) != 'e' && c != 'E')
     switch(c) {
           case 'u' :
           case 'U' :
              scanf("%d",&x);
              err = push(&s,x);
              break;
           case 'o' :
           case 'O' :
              err = pop(&s);
              break;
           case 't' :
           case 'T' :
              err = top(&s , &val) ;
```

```
                if(!err) printf("%d\n", val);
                break;
            case '\n' :
            case '\t' :
            case ' '  :
                break;
            default :
                printf("Token Unknown\n");
        }
    return 0;
}
```

Compiling the User Program

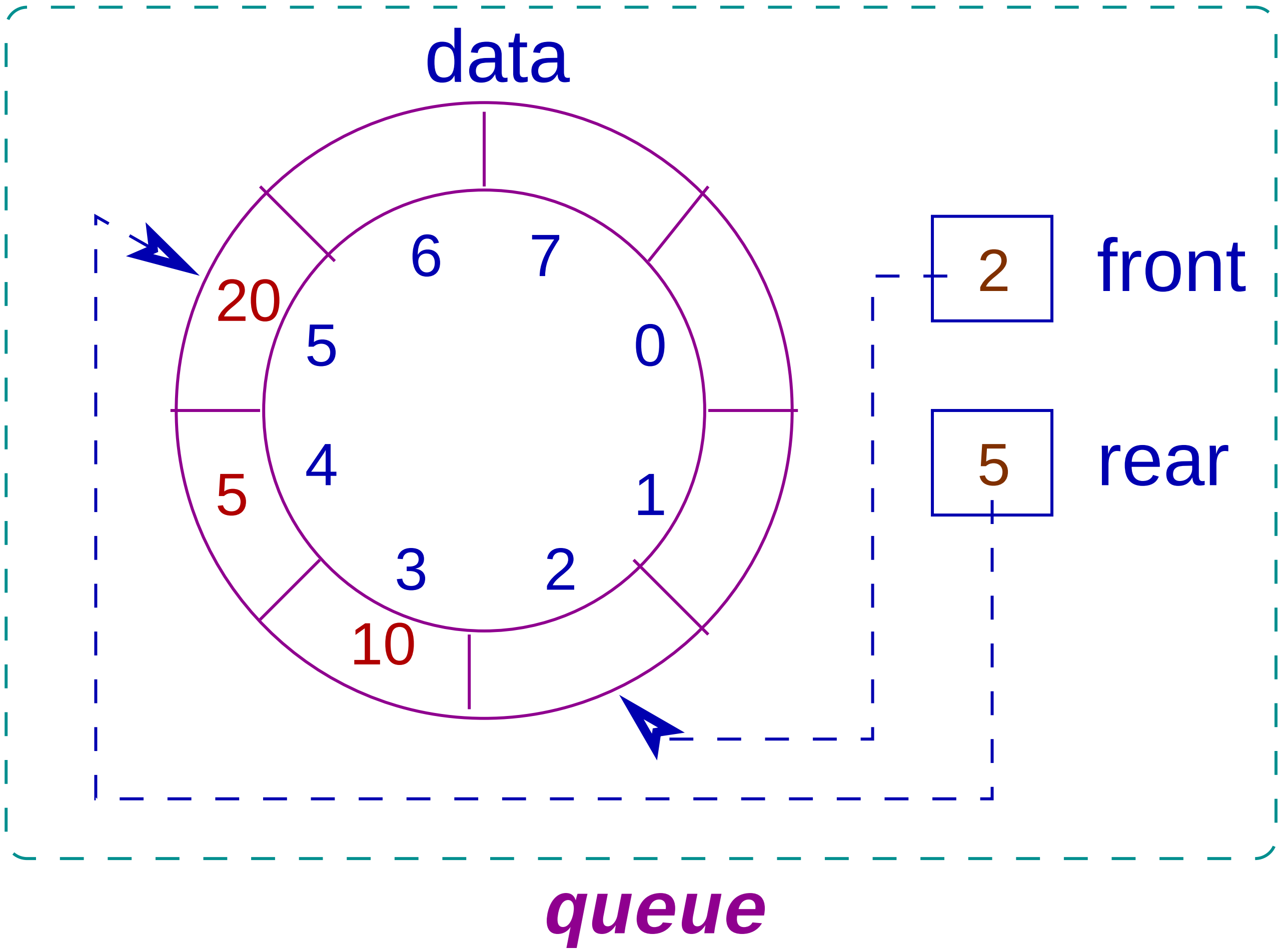
```
$ cc -Wall testStack.c stack.o
```

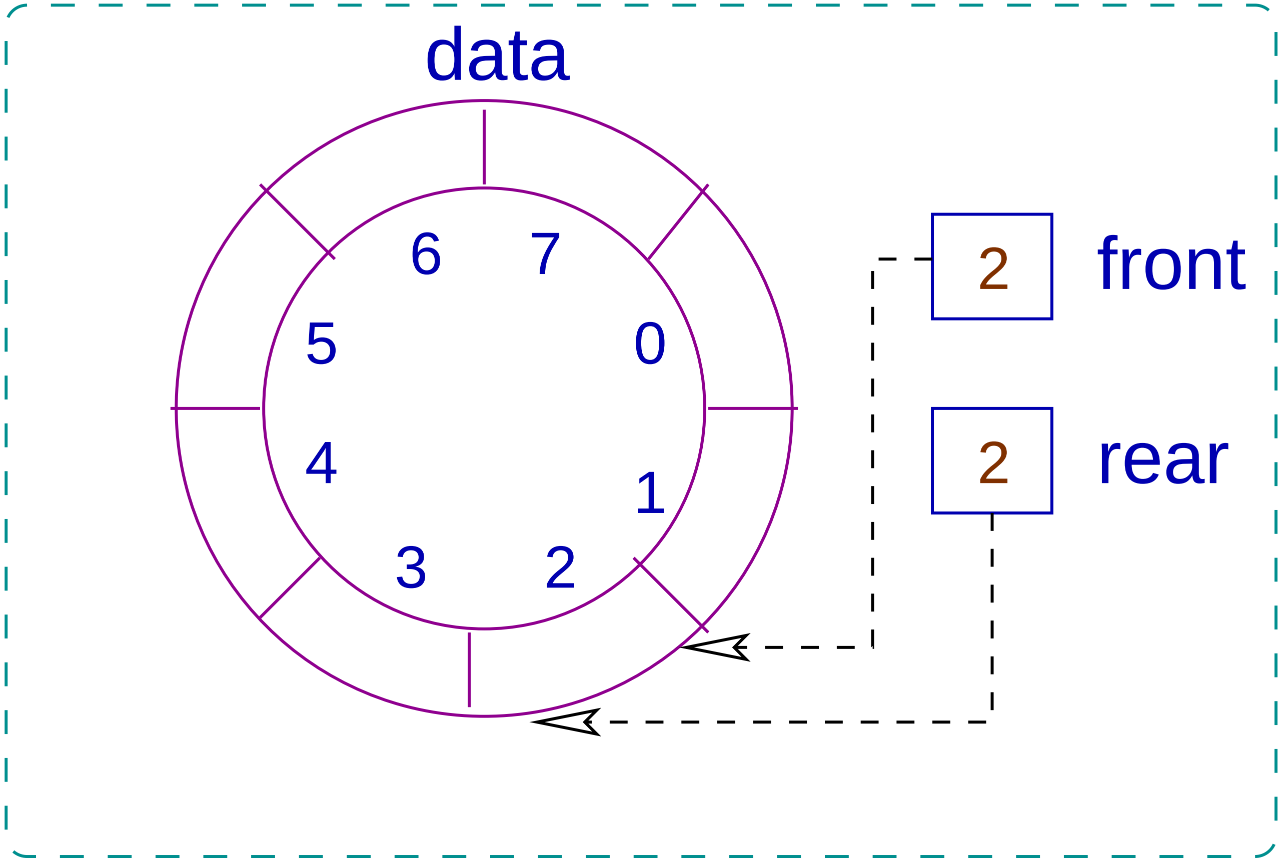
We get the executable module `a.out`.

Queue on Circular Array: Representation

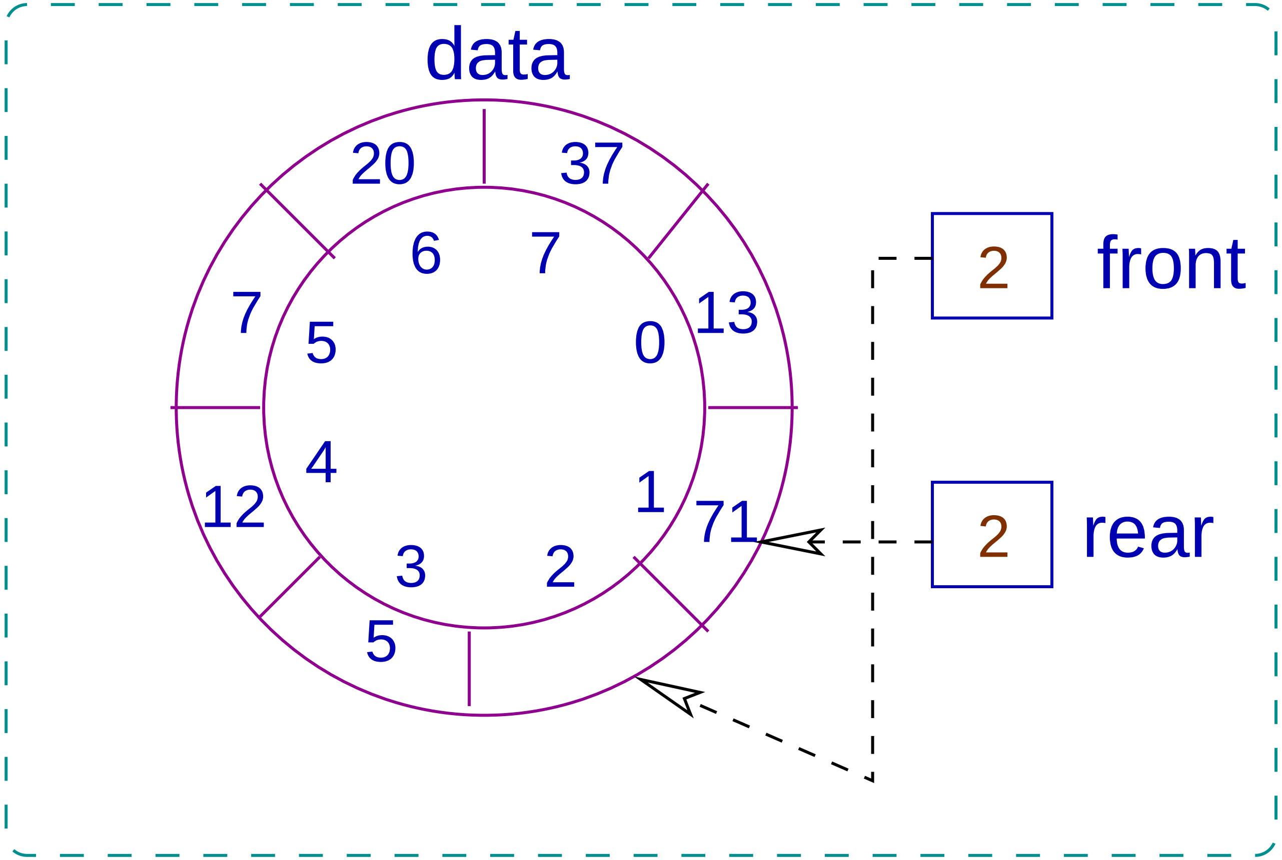
```
#define MAX 200
typedef struct {
        int data[MAX] ;
        int front , rear ;
} queue;
```

The queue may contain `MAX - 1` data.


data
6
7
20
5
0
4
5
1
3
2
10
2
front
5
rear
queue



empty queue



full queue

Queue on Circular Array: Operations

```
void init(queue *) ;
int add(queue *, int) ;
int delete(queue *);
int front(queue *, int *) ;
int isEmpty(queue *) ;
int isFull(queue *) ;
```

Interface File: queue.h

```
#include <stdio.h>
#ifndef _QUEUE_H
#define _QUEUE_H

#define MAX 200
#define ERROR 1
#define OK 0

typedef struct { // queue.h
```

```
        int data[MAX];
        int front, rear;
} queue;

/* Queue may contain MAX-1 data.*/

 void init(queue *);
 int add(queue *, int);
 int delete(queue *);
 int front(queue *, int *);
 int isEmpty(queue *);
```

```
 int isFull(queue *);
#endif
```

Implementation File: queue.c

```
#include "queue.h"
void init(queue *q)  // queue.c
{ q->front=q->rear=0; }

int isEmpty(queue *q)
{   return q->rear == q->front; }

int isFull(queue *q)
{   return (q->rear+1)%MAX == q->front; }
```

```
int add(queue *q, int n) {
    if(isFull(q)) return ERROR;
    q->rear=(q->rear+1)%MAX;
    q->data[q->rear]=n;
    return OK ;
}

int delete(queue *q) {
    if(isEmpty(q)) return ERROR ;
    q->front=(q->front+1)%MAX ;
```

```
    return OK ;
}

int front(queue *q , int *v) {
    if(isEmpty(q)) return ERROR ;
    *v=q->data[(q->front+1)%MAX] ;
    return OK ;
}
```

User Program: testQueue.c

```
#include "queue.h"
int main()  // testQueue.c
{
    queue q ;
    int x , err , val ;
    char c;

    init(&q);
    printf(" 'A' for add (A 15)\n") ;
    printf(" 'D' for delete\n 'F' for front\n 'E' for exit
    while((c = getchar()) != 'e' && c != 'E')
        switch(c) {
```

```
case 'a' :
case 'A' :
  scanf("%d",&x);
  err = add(&q,x);
  if(err) printf("The Queue is full\n") ;
  break;
case 'd' :
case 'D' :
  err = delete(&q);
  if(err) printf("The Queue is empty\n") ;
  break;
case 'f' :
case 'F' :
  err = front(&q , &val) ;
```

```
            if(err) printf("The Queue is empty\n") ;
            else printf("%d\n",val);
            break;
          case '\n' :
          case '\t' :
          case ' '  :
            break;
          default :
            printf("Token Unknown\n");
    }
    return 0 ;
}
```

Time Complexity of Operations

Both in stack and queue the running time of each operation is $O(1)$. Similarly the space complexity of each operation is also $O(1)$.

Queue on Circular Array: A Variation

We may use a counter and and keep data in all the array locations. In practice the array locations may be a structure and a counter is just a variable of type `int`.

```
#define MAX 200
typedef struct {
        int data[MAX] ;
        int front, rear, count ;
} queue;
```