CS11001/CS11002
Programming and Data Structures (PDS)
(Theory: 3-1-0)

The queue ADT

- A queue is like a "natural" queue of elements.
- It is an ordered list in which all insertions occur at one end called the **back** or **rear** of the queue.
- All deletions occur at the other end called the **front** or **head** of the queue.
- In the popular terminology, insertion and deletion in a queue are respectively called the **enqueue** and the **dequeue** operations.
- The element dequeued from a queue is always the first to have been enqueued among the elements currently present in the queue.
  - In view of this, a queue is often called a **First-In-First-Out** or a **FIFO** list.

Initializing the queue

```
queue init ()
{
queue Q;
Q.front = 0;
Q.back = MAXLEN - 1;
return Q;
}
```

Operations on the queue ADT

- Q = init();
  - Initialize the queue Q to the empty queue.
- isEmpty(Q);
  - Returns "true" if and only if the queue Q is empty.
- isFull(Q);
  - Returns "true" if and only if the queue Q is full, provided that we impose a limit on the maximum size of the queue.
- front(Q);
  - Returns the element at the front of the queue Q or error if the queue is empty.

Operations on the queue ADT

- Q = enqueue(Q,ch);
  - Inserts the element ch at the back of the queue Q. Insertion request in a full queue should lead to failure together with some appropriate error messages.
- Q = dequeue(Q);
  - Delete one element from the front of the queue Q. A dequeue attempt from an empty queue should lead to failure and appropriate error messages.
- print(Q);
  - Print the elements of the queue Q from front to back.

Implementations of the queue ADT

- We maintain two indices to represent the front and the back of the queue.
- During an enqueue operation, the back index is incremented and the new element is written in this location.
- For a dequeue operation, on the other hand, the front is simply advanced by one position.
- It then follows that the entire queue now moves down the array and the back index may hit the right end of the array, even when the size of the queue is smaller than the capacity of the array.

Reducing wastage in the queue

- In order to avoid waste of space, we allow our queue to *wrap* at the end.
- This means that after the back pointer reaches the end of the array and needs to proceed further down the line, it comes back to the zeroth index, provided that there is space at the beginning of the array to accommodate new elements.
- Thus, the array is now treated as a circular one with index MAXLEN treated as 0, MAXLEN + 1 as 1, and so on.
- That is, index calculation is done modulo MAXLEN.
- We still don't have to maintain the total queue size.
- As soon as the back index attempts to collide with the front index modulo MAXLEN, the array is considered to be full.

Tackling Isempty and Isfull

- There is just one more problem to solve.
- A little thought reveals that under this wrap-around technology, there is no difference between a full queue and an empty queue with respect to arithmetic modulo MAXLEN.
- This problem can be tackled if we allow the queue to grow to a maximum size of MAXLEN - 1.
  - This means we are going to lose one available space, but that loss is inconsequential.
  - Now the condition for full array is that the front index is two locations ahead of the back modulo MAXLEN, whereas the empty array is characterized by that the front index is just one position ahead of the back again modulo MAXLEN.

Isempty and Isfull

```
int isEmpty ( queue Q )
{
return (Q.front == (Q.back + 1) % MAXLEN);
 }
```

```
int isFull ( queue Q )
{
return (Q.front == (Q.back + 2) % MAXLEN);
}
```

Enqueue

```
queue enqueue ( queue Q , char ch )
{
  if (isFull(Q))
{
  fprintf(stderr,"enqueue: Queue is full\n");
  return Q;
}
++Q.back;
if (Q.back == MAXLEN)
  Q.back = 0;
  Q.element[Q.back] = ch;
 return Q;
}
```

Dequeue

```
queue dequeue ( queue Q )
{
  if (isEmpty(Q))
    {
 fprintf(stderr,"dequeue: Queue is empty\n"); return Q;
    }
   ++Q.front;
 if (Q.front == MAXLEN)
   Q.front = 0;
  return Q;
  }
```

The main program

```
int main ()
{
   queue Q;

   Q = init(); printf("Current queue : "); print(Q); printf("\n");
   Q = enqueue(Q,'h'); printf("Current queue : "); print(Q); printf("\n");
   Q = enqueue(Q,'w'); printf("Current queue : "); print(Q); printf("\n");
   Q = enqueue(Q,'r'); printf("Current queue : "); print(Q); printf("\n");
   Q = dequeue(Q); printf("Current queue : "); print(Q); printf("\n");
   Q = dequeue(Q); printf("Current queue : "); print(Q); printf("\n");
   Q = enqueue(Q,'c'); printf("Current queue : "); print(Q); printf("\n");
   Q = dequeue(Q); printf("Current queue : "); print(Q); printf("\n");
   Q = dequeue(Q); printf("Current queue : "); print(Q); printf("\n");
   Q = dequeue(Q); printf("Current queue : "); print(Q); printf("\n");
}
```

Output

```
Current queue :
Current queue : h
Current queue : hw
Current queue : hwr
Current queue : wr
Current queue : r
Current queue : rc
Current queue : c
Current queue :
dequeue: Queue is empty
Current queue :
```

Defining a node in the queue

```
#include <stdio.h>
#include <malloc.h>

typedef struct _node {
   char element;
   struct _node *next;
} node;

typedef struct {
   node *front;
   node *back;
} queue;
```

Initialization of the queue

```
queue init ()
{
   queue Q;

   /* Create the dummy node */
   Q.front = (node *)malloc(sizeof(node));
   Q.front -> element = ' ';
   Q.front -> next = NULL;
   Q.back = Q.front;
   return Q;
}
```

isEmpty, isFull, front

```
int isEmpty ( queue Q )
{
   return (Q.front ==
    Q.back);
}

int isFull ( queue Q )
{
   return 0;
}
```

```
char front ( queue Q )
{
   if (isEmpty(Q)) {
      fprintf(stderr,"front:
    Queue is empty\n");
      return '\0';
   }
   return Q.front -> element;
}
```

Enqueue

```
queue enqueue ( queue Q
    char ch )
{
   node *C;
   if (isFull(Q)) {
      fprintf(stderr,"enqueue:
    Queue is full\n");
      return Q;
   }
```

```
   /* Create new node */
   C = (node *)malloc(sizeof(node));
   C -> element = ch;
   C -> next = NULL;

   /* Adjust the back of queue */
   Q.back -> next = C;
   Q.back = C;

   return Q;
}
```

Dequeue

```
queue dequeue ( queue Q )
{
   if (isEmpty(Q)) {
      fprintf(stderr,"dequeue: Queue is empty\n");
      return Q;
   }

   /* Make the front of the queue the new dummy node */
   Q.front = Q.front -> next;
   Q.front -> element = '\0';

   return Q;
}
```

Printing the queue

```
void print ( queue Q )
{
   node *G;

   G = Q.front -> next;
   while (G != NULL) {
      printf("%c", G -> element);
      G = G -> next;
   }
}
```

The main program

```
int main ()
{
   queue Q;

   Q = init(); printf("Current queue : "); print(Q); printf("\n");
   Q = enqueue(Q,'h'); printf("Current queue : "); print(Q); printf("\n");
   Q = enqueue(Q,'w'); printf("Current queue : "); print(Q); printf("\n");
   Q = enqueue(Q,'r'); printf("Current queue : "); print(Q); printf("\n");
   Q = dequeue(Q); printf("Current queue : "); print(Q); printf("\n");
   Q = dequeue(Q); printf("Current queue : "); print(Q); printf("\n");
   Q = enqueue(Q,'c'); printf("Current queue : "); print(Q); printf("\n");
   Q = dequeue(Q); printf("Current queue : "); print(Q); printf("\n");
   Q = dequeue(Q); printf("Current queue : "); print(Q); printf("\n");
   Q = dequeue(Q); printf("Current queue : "); print(Q); printf("\n");
}
```

Output

- Current queue :
- Current queue : h
- Current queue : hw
- Current queue : hwr
- Current queue : wr
- Current queue : r
- Current queue : rc
- Current queue : c
- Current queue :
- dequeue: Queue is empty
- Current queue :

Two Examples

Testing whether a line is straight (incomplete program)

```
#include <stdio.h>
typedef struct {
  int x;
  int y;
}points;
typedef points line [3];
main()
{
 line l1;
 int i;
 float  m1, m2;
 for(i=0;i<3;i++){
   printf("Enter Point %d\n",i+1);
   scanf("%d",&l1[i].x);
   scanf("%d",&l1[i].y);
 }
```

```
if((!l1[0].x-l1[0].x)&&(!l1[2].x-
l1[1].x))
{
 m1=(float)(l1[1].y-
l1[0].y)/(l1[1].x-l1[0].x);
 m2=(float)(l1[2].y-
l1[1].y)/(l1[2].x-l1[1].x);
}
```

```
 if(m1==m2)
  printf("Line is straight…\n");
}
```

Reversing a linked list

```
void reverse(olist L)
{
 node *temp;
 node *temp1;
 node *temp2;
 temp=L;temp1=temp->next;temp2=temp1->next;
 temp->next->next=NULL;

 while(temp2!=NULL)
 {
  temp=temp1;
  temp1=temp2;
  temp2=temp1->next;
  temp1->next=temp;
 }
L->next=temp1;
}
```

Recursion using stacks for n>=2

```
void rec(int n, stack S)
{
 int tp, res=1;
 int cnt=n;
   printf("Current stack : "); print(S); printf(" with top = %d.\n", top(S));
 while(!isEmpty(S)){
    tp=top(S);
     if(tp==2)
     {
      res=1;
      while(cnt>1)
       {res*=top(S); S=pop(S);cnt--;}
     }
   else{
     S=push(S,n-1);
     n=n-1;
   }
 }
 printf("Factorial = %d\n",res);

}
```

```
int main ()
{
   stack S;
   int n;
   S = init();
   S = push(S,n);
   printf("Current stack : ");
print(S); printf(" with top =
%d.\n", top(S));


   rec(n,S);
}
```