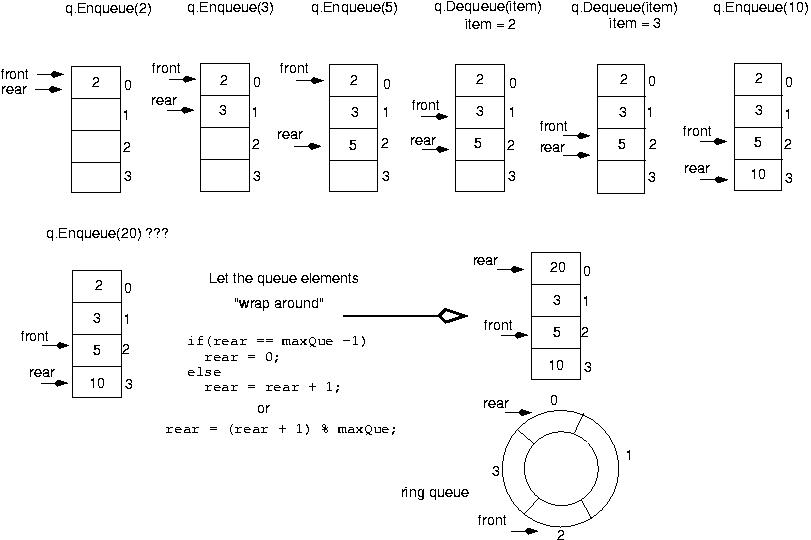
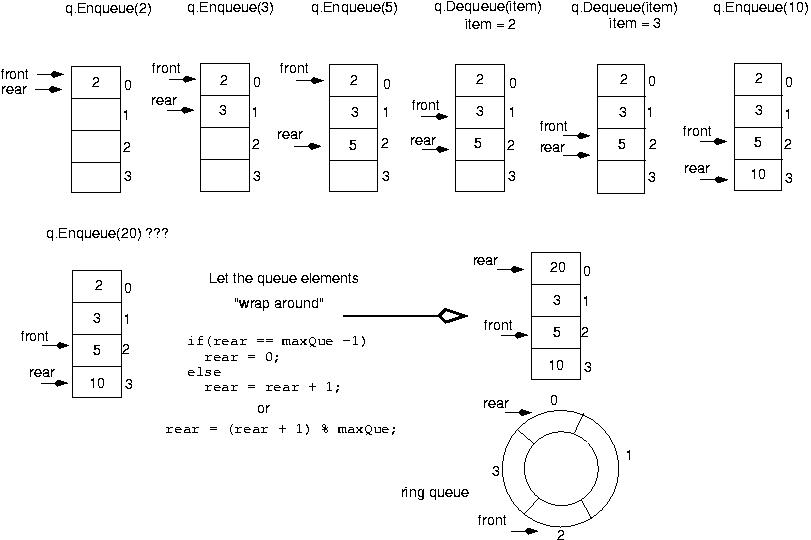
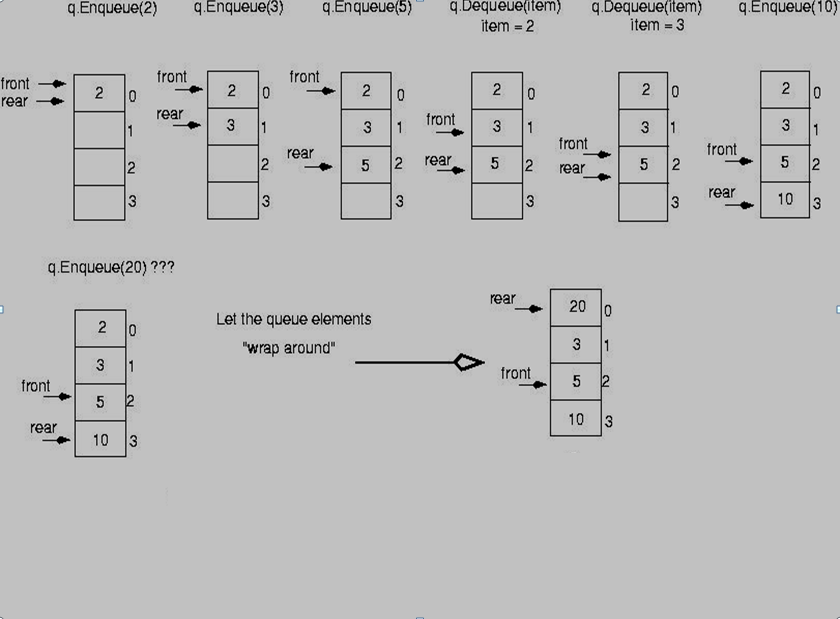
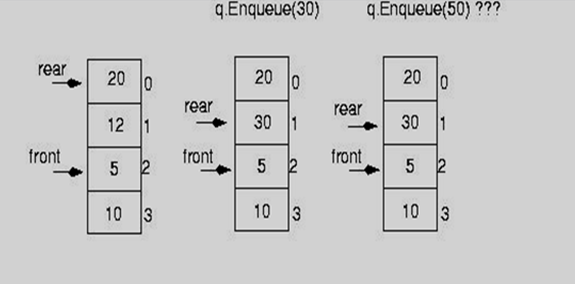
**Circular list**

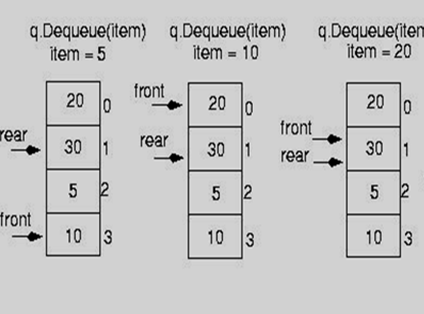
**Deleting and inserting (Have to worry about out of bounds)**

1. Rear and Front start at -1 (Array is empty)
2. Inserts are done at the rear(rear++), deletes are done at the front(front++)
   1. If front or rear is outside the bounds, it has to be set to 0
3. If inserting, first check if array is full (rear+1==front).
   1. What is full? If rear +1 == front, then there is no space in the middle.
      1. There has to be at least one space between rear and front.
   2. If okay to insert, increment rear (resolve out of bounds) and then insert.
4. When deleting
   1. If front and rear are the same and not -1, delete, set rear and front to -1.
   2. If front is -1, nothing can be deleted. The array is empty
   3. If front and rear are not the same, then delete, increment front (resolve out of bounds)
5. Note that when inserting, first increment rear and then insert. When deleting, first delete and then increment front. If inserting for the first time, increment both rear and front.









//When inserting, increment rear and also front (if it is -1) and then insert

//When removing, first remove and then adjust front and also rear (if front is equal to rear)

#define SIZE 3;

int queue[SIZE+1], front=-1, rear=-1;

void insert() {

//It is full if the distance between rear and Front is by 1.

//resolve out of bounds with SIZE?0:rear+1

if (front == (rear+1==SIZE?0:rear+1) )

printf("\n\full.");

else {

printf("\n\nEnter ITEM: ");

scanf("%d", &item);

//We start with -1 to indicate that the list is empty.

//We have to decide where we want to insert.

//We need to increment rear.

//If rear is at the end, then we make it 0. Otherwise, increment it

//If front is at -1, move it because we have something to delete as well.

if(rear == SIZE-1)

rear = 0; //increment which goes all the way back to the beginning

else {

//If the array is empty, at this point we have decided to insert

//which means it is no longer empty and so we should be able to

//increment front

if (front==-1)

front++;

rear++;

}

queue[rear] = item;

}

}

void remove() {

if(front == -1)

cout<<”empty.\n";

else {

cout<< queue[front]; //item to be removed

if(front == rear) {

front = -1;

rear = -1;

}else if(front == SIZE-1)

front = 0;

else

front++;

}

}

int main(){

insert():

insert():

remove():

remove():

insert():

remove():

insert():

insert():

}

**Inserting and deleting using MOD operation (Ignore out of bounds)**

* The mod operator (%) is used to calculate remainders:
  + 0%5=0, 1%5=1, 2%5=2, 5%5=0, 8%5 = 3
* mod can be used to calculate the front and back positions in a circular array, therefore avoiding comparisons to the array size
  + Insert in the following rear position:
    - rear=(rear+1)%SIZE;
  + Remove from following front position:

front=(front+1)%SIZE;

//When inserting, increment rear and count

//When removing, increment front, decrement count

//count is the true count. Zero means no elements.

#define SIZE 3

int queue[SIZE]={0}, rear=-1, front=-1, item, count=0;

void insert()

{

if (count<SIZE)

{

rear=(rear+1)%SIZE;

cout<<"insert what? ";

cin>>item;

queue[rear]=item;

count++;

} else

cout<<"full";

}

void remove() {

if (count>0){

front=(front+1)%SIZE;

queue[front]=0;

count--;

} else

cout<<"empty";

}

int main(){

insert ();

insert ();

insert ();

insert ();

remove ();

remove ();

remove ();

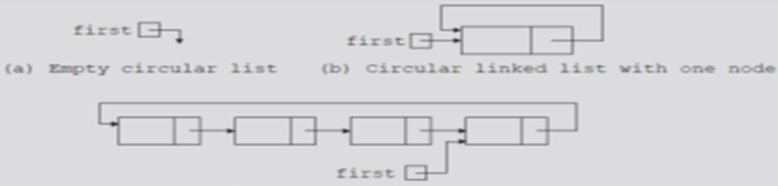
remove ();

}

**Using a linked list**

*• Circular* Instead of setting the .next field of the last node to NULL, set it to point back around to the first node. Instead of needing a fixed head end, any pointer into the list will do.

* + - pointer in the last node points back to the first node



struct node {

int item;

node \*next;

};

//Insert the new node after head and make it the new head

//The new node will point to the beginning

void newHead(int item, struct node \*&head){

struct node \*newNode =(struct node \*) malloc(sizeof (node));

newNode->item=item; //this is the new item

if(!head){

head=newNode;

head->next=head;

}else{

newNode->next=head->next;

head->next=newNode;

head=newNode;

}

}

int main(){

struct node \*head=NULL;

newHead(5,head);

newHead(6,head);

newHead(7,head);

newHead(8,head);

newHead(9,head);

}

**HeadAfter**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Head   |  | | --- | | NULL |   newNode   |  | | --- | | 10 |     Head   |  | | --- | | 10 | | |  | | --- | | 10 | | 1 | | 10 |   newNode->data=1;  Head=newNode  Head->next=head |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Head   |  | | --- | | 10 |   newNode   |  | | --- | | 20 |       Head   |  | | --- | | 20 | | |  | | --- | | 10 | | 1 | | 10/20 |     newNode   |  | | --- | | 20 | | 2 | | 10 |   newNode->data=2;  newnode->next=head->next  Head->next=newnode  Head= newnode |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Head   |  | | --- | | 20 |   newNode   |  | | --- | | 30 |       Head   |  | | --- | | 30 | | |  | | --- | | 10 | | 1 | | 20 |  |  | | --- | | 20 | | 2 | | 10/30 |   newNode   |  | | --- | | 30 | | 3 | | 10 |   newNode->data=3;  newnode->next=head->next  Head->next=newnode  Head= newnode |

1. How will the traverse function work and what will be displayed?

**Challenge**:

The above code places the new node after the current head. How would we insert a new node before the current head?

* *Circular, doubly linked list*
  + - forward pointer of the last node points to the first node and backward pointer of the first node points to the last node

struct node {

int data;

node \*next;

node \*prev;

};

void insertBeginning (int item, struct node \*&head)

{

node \* newNode;

newNode=new node;

newNode->data=item;

if (!head){ //nothing in the list, everything points to self

newNode->prev=newNode;

newNode->next=newNode ;

head=newNode;

}else{ //something in the list. Connect old tail and old head

newNode->prev=head->prev; //newNode->prev=head->prev connect newNode to tail

newNode->next=head; //connect newNode to head

head->prev=newNode; //connect old head to the one before it which is the new node

newNode->prev->next=newNode; //connecting the tail to the new node

head=newNode; //head becomes newNode

}

}