1. Write a recursive function to compute
    a. Maximum element of the array
    b. The minimum element of the array
    c. The sum of elements of array
    d. The product of elements of array
    e. The average of the elements of the array.

Solution

Function to compute the maximum element of an array

```
max(int * a, int n)
//a is pointer to an array, n is the index of the last element
{
if(n==0)
  return a[0]
else
  m= max(a, n-1)
 if( a[n] > m)
   return a[n]
else
 return m
}
```

Function to compute the minimum element of an array

```
min(int * a, int n)
//a is pointer to an array, n is the index of the last element
{
if(n==0)
  return a[0]
else
  m= min (a, n-1)
 if( a[n] < m)
   return a[n]
else
 return m
}
```

Function to compute the sum of elements of an array

```
sum(int * a, int n)
//a is pointer to an array, n is the index of the last element
{
if(n==0)
  return a[0]
else
  return ( a[n] + sum(a,n-1))
}
```

Function to compute the product of elements of array

```
product(int * a, int n)
//a is pointer to an array, n is the index of the last element
{
if(n==0)
  return a[0]
else
  return ( a[n] * product(a,n-1))
}
```

Function to compute the average of the elements of the array.

```
average(int * a, n, int i)
//a is pointer to an array, i is the index of the element, n is the number of elements
{
if(i==n-1)
  return (a[i]/n)
else
  a=a[i]/n + average(a, i +1,n)
  return a
}
```

2. **Evaluate each of the following using recursive definitions**
    a. **6 !**
    b. **9 !**
    c. **100 * 3**
    d. **6 * 4**
    e. **fib(10)**
    f. **fib(11)**

Solution

a) 6 ! = 6 * 5 !
5 ! = 5 * 4!
4 != 4 * 3!
3 ! = 3 * 2!
2 ! = 2 * 1!
1! = 1;
Therefore

2! = 2 * 1 ! = 2 * 1 = 2
3 ! = 2 * 2! = 3 * 2 = 6
4 != 4 * 3 ! = 4 * 6 = 24
5 ! = 5 * 4 ! = 5 * 24 = 120
6 ! = 6 * 5 != 6 * 120 = 720

b) 9 ! = 9 * 8 !

c) 100 * 3
Using a * b = a if b =1
a * b = a + a * (b - 1) if b > 1

= 100 + 100 * (3-1)
= 100 + 100 *2
100 * 2 = 100 + 100 * (2 -1)
= 100 = 100 * 1
100 * 1 = 100
100 * 2 = 100 + 100*1 = 100 + 100 = 200
100 * 3 = 100 * 100 * 2 = 100 + 200 = 300

d) 6 * 4
= 6 + 6 * 3

e) fib(10)
Using fib(n) = n if n=0 or n=1
fib(n) = fib(n-2) + fib(n-1)
fib(10) = fib(8) + fib(9)

fib(11) = fib(9) + fib(10)

3.Determine what the following recursive C function computes.

```
 int func(int n)
 {
     if(n = =0)
         return 0;
    return ( n + func(n-1));
}
```

Solution : It computes the 0 + 1 + 2 + 3 …+ n

4. Transform each of the following expressions to prefix and postfix

a. **A + B - C**
b. **(A + B)* (C - D) $ E * F**
c. **(A + B)* (C $ (D - E) + F) - G**
d. **A + (((B - C) * (D - E) + F) / G) $ (H - J)**

Solution

a) **A + B - C**
   Postfix = AB + C -
   Prefix = - C + A B
b) **(A + B)* (C - D) $ E * F**
   Postfix = AB * C D - * E $ * F
   Prefix = * $ * * A B - C D E F

c) **(A + B)* (C $ (D - E) + F) - G**
   Postfix = AB+CDE-$F+*G-
   Prefix = - * + AB + $ C - DEFG

d) **A + (((B - C) * (D - E) + F) / G) $ (H - J)**
   e)

5. A deque is an ordered set of items from which items may be deleted at either end and into which items may be inserted at either end. Call the two ends of a deque left and right. How can a deque be represented as a LInked List. Write for routines remvleft, remvright, insrtleft, insrtright.

```
void qinsert_left(int x,struct dequeue *dq)
  {
    struct node *temp;
```

```
  temp=(struct node*)malloc(sizeof(struct node));
  temp->data=x;
  temp->prev=temp->next=NULL;

  if(dq->front==NULL)
     dq->front=dq->rear=temp;
   else
    {
      temp->next=dq->front;
      dq->front->prev=temp;
      temp->prev=NULL;
      dq->front=temp;
    }
}


void qinsert_right(int x,struct dequeue* dq)
{
  struct node *temp;

  temp=(struct node*)malloc(sizeof(struct node));
  temp->data=x;
  temp->prev=temp->next=NULL;

  if(dq->front==NULL)
     dq->front=dq->rear=temp;
   else
    {
      dq->rear->next=temp;
      temp->prev=dq->rear;
      dq->rear=temp;
    }
}

int qremv_left(struct dequeue* dq)
{
  struct node *q;
  int x;
  if(dq->front==NULL)
    return -1;
```

```
  q=dq->front;
   x=q->data;
  if(dq->front==dq->rear)//only one node
    dq->front=dq->rear=NULL;
  else
    {
      dq->front=dq->front->next;
      dq->front->prev=NULL;
    }
    free(q);
   return x;
 }

int  qremv_right(struct dequeue* dq)
{
  struct node *q;
   int x;
   if(dq->front==NULL)
     return -1;

  q=dq->rear;
   x=q->data;
  if(dq->front==dq->rear)//only one node
    dq->front=dq->rear=NULL;
  else
   {
      dq->rear=dq->rear->prev;
      dq->rear->next=NULL;
    }
   free(q);
    return x;
 }

  void qdisplay(struct dequeue dq)
  {
     struct node *p;
    if(dq.front==NULL)
      printf("Empty Queue...\n");
    else
```

```
        {
         for(p=dq.front;p!=dq.rear;p=p->next)
         {
            printf("%d ",p->data);
         }
         printf("%d ",p->data);
        }
      }
```

6. Implement an ascending priority queue and its operations pqinsert, pqmindelete and empty using an array

```
struct pqueue pqmindelete(struct pqueue *pq,int *count)
 {
   struct pqueue key;
   int i;
   if(*count==0)
   {
     key.data=0;
     key.pty=-1;
    }
   else
   {
     key=pq[0];
     for(i=1;i<=*count-1;i++)
        pq[i-1]=pq[i];
   }
   (*count)--;
    return key;
 }
```

```
 void pqinsert(int x,int pty,struct pqueue* pq,int *count)
 {
   int j;
   struct pqueue key;
   key.data=x;
   key.pty=pty;
```

```
   j=*count-1;

   while((j>=0)&&(pq[j].pty < key.pty))
   {
         pq[j+1]=pq[j];
         j--;
   }
   pq[j+1]=key;
   (*count)++;
 }


 void  qdisplay(struct pqueue *q, int count)
 {
   int i;
   if(count==0)
    printf("Empty Queue");
  else
  {
   for(i=0;i<count;i++)
   {
     printf("data = %d",q[i].data);
     printf("  pty=%d",q[i].pty);
     printf("\n");
   }
  }
 }
```

7. Implement the routines empty, push,pop using the dynamic storage implementation of a linked stack.

```
struct node
{
  int data;
  struct node *next;
 };
 void push(int,struct node **);
 int pop(struct node **);
 void display(struct node *);
```

```
int empty
{
   if(top==NULL)
     return 1
  return 0
}

void push(int x, struct node **p)
    {
          struct node *temp;
          temp=(struct node*)malloc(sizeof(struct node));
          temp->data=x;
          temp->next=*p;
          *p=temp;
     }

   int pop(struct node **p)
   {
        int x;
        struct node *q;
        if(*p==NULL)
        {
           printf("Empty Stack\n");
            return -1;
         }
         else
          {
             q=*p;
              x=q->data;
              *p=q->next;
              free(q);
              return(x);
           }
      }
      void display(struct node *p)
     {
         if(p==NULL)
           printf("Empty Stack\n");
```

```
        else
        {
          while(p!=NULL)
          {
            printf("%d->",p->data);
            p=p->next;
          }
        }
    }
```

8. Implement the routines empty, insert ,remove using the dynamic storage implementation of a linked queue.

```
struct node
{
  int data;`
  struct node *next;
};
void insert(int, struct node **,struct node **);
int remove(struct node **,struct node **);
void display(struct node*,struct node*);

void insert(int x, struct node **f, struct node **r)
   {
     struct node *temp;

     temp=(struct node*)malloc(sizeof(struct node));
     temp->data=x;
     temp->next=NULL;

      //if this is the first node
     if(*f==NULL)
       *f=*r=temp;
     else //insert at the end
        {
          (*r)->next=temp;
           *r=temp;
        }
    }
```

```
int remove(struct node **f, struct node **r)
  {
       struct node *q;
       int x;
       q=*f;
       if(q==NULL)
        {
         printf("Empty queue\n");
         return -1;
        }
        else
        {
          x=q->data;
          if(*f==*r) //only one node
            *f=*r=NULL;
          else
          {
          *f=q->next;
          return x;
         }
        free(q);
       }
 }

  void qdisplay(struct node *f, struct node *r)
   {
      if(f==NULL)
        printf("Queue Empty\n");
    else
     {
      while(f!=r)
      {
        printf("%d-> ",f->data);
        f=f->next;
      }
    printf("%d-> ",f->data);
    }
  }
```

```
 int empty(struct node *f)
{
  if(f==NULL)
    return 1
return 0
}
```

9. Implement the routines empty, pqinsert and pqmindelete using the dynamic storage implementation of a linked priority queue.

```
struct node
{
        int pty;
        int data;
        struct node *next;
};

struct node *front,*rear;

void pqinsert(int x, int y)
{
   struct node *p, *q, *prev;
   p=(struct node*)malloc(sizeof(struct node));
   p->pty=y;
   p->data=x;
   p->next=NULL;

   q=front;
   prev=NULL;

   while((q!=NULL)&&(q->pty<=y))
   {
      prev=q;
      q=q->next;
   }

   if(q==NULL)
   {
```

```
        if(prev==NULL)//empty list
            front=p;//insert as first node of the list
        else
          prev->next=p;//insert as last node
    }
    else
    {
                if(prev==NULL)//insert as first node
                {
                        p->next=q;
                        front=p; //change the value of first
                 }

                else //insert in middle
                 {
                        prev->next=p;
                        p->next=q;
                }
    }
}

int pqmindelete()
{
        int x;
        struct node *q;
        q=front;

        if(q==NULL)
        {
          printf("Empty queue\n");
          return -1;
        }
       else
       {
         x=q->data;
         front=q->next;
         free(q);
         return(x);
     }
}
```

```
void qdisplay()
{
    struct node *q;
    if(front==NULL)
          printf("Empty list\n");
      else
    {
        q=front;
        while(q!=NULL)
        {
             printf("%d %d ->", q->pty, q->data);
            q=q->next;
        }
    }
}
```