DATA STRUCTURES LAB FILE

NAME: YASH SHARMA
ROLL NO.: DTU/2K16/IT/127

Aim: To implement Array Data Structure and its operations (Creation, Insertion, Deletion, Searching(Binary), Sorting(Insertion,Merge)).

Description: Array is a collection of contiguous memory location under one name. All the memory blocks are of same data-type. Certain operations can be performed on array like insertion, deletion, rotation, searching etc.

Algorithm:

Creation:

```
1. Start
2. Set J = N
3. Set N = N+1
4. Repeat steps 5 and 6 while J >= K
5. Set LA[J+1] = LA[J]
6. Set J = J-1
7. Set LA[K] = ITEM
8. Stop
```

Deletion:

```
1. Start
2. Set J = K, I=0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is not equal ITEM THEN GOTO STEP 5
5. Set LA[I] = LA[J] with increment by 1 in I
6. Set J = J+1
7. Set N = N-1
8. Stop
```

Search:

```
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop
```

Binary Search:

```
1. Start
2. Set $L$ to 0 and $R$ to $n – 1$.
3. If $L > R$, the search terminates as unsuccessful.
```

```
4. Set m(the position of the middle element) to the floor (the largest previous integer) of $(L + R)/2$.
5. If A[m] < T, set L to m + 1 and go to step 2.
6. If A[m] > T, set R to m − 1 and go to step 2.
7. Now A[m] = T, the search is done; return m.
8. Stop
```

Insertion Sort:

```
1. Start
2. i ← 1
3. while i < length(A)
4. x ← A[i]
5. j ← i – 1
6. while j >= 0 and A[j] > x
7. A[j+1] ← A[j]
8. j ← j - 1
9. end while
10. A[j+1] ← x
11. i ← i + 1
12. end while
13. Stop
```

Merge Sort:

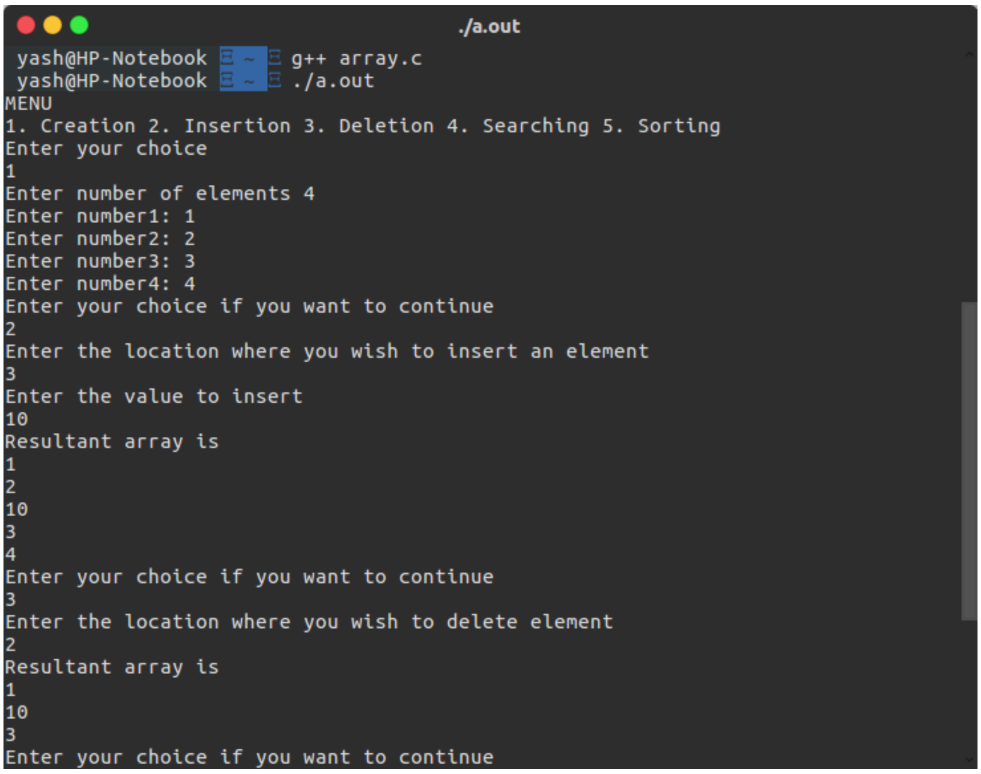
```
If r > l
1. Find the middle point to divide the array into two halves:
        middle m = (l+r)/2
2. Call mergeSort for first half:
        Call mergeSort(arr, l, m)
3. Call mergeSort for second half:
        Call mergeSort(arr, m+1, r)
4. Merge the two halves sorted in step 2 and 3:
        Call merge(arr, l, m, r)
```

Code:

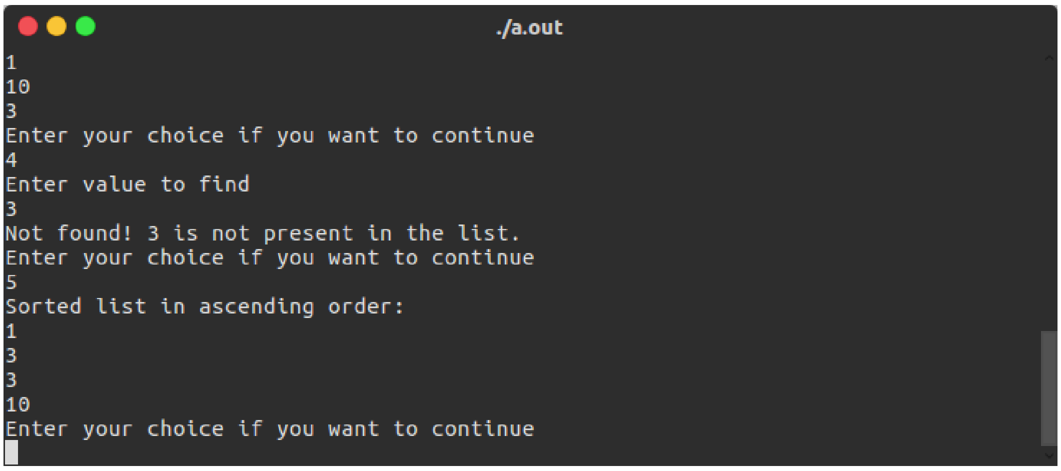
```
#include <stdio.h>
#include <stdlib.h>
 static int n,l,m,r;
 static int a[100];
void create()
{
  int i;
 printf("Enter number of elements ");
     scanf("%d", &n);
     for( i=0; i<n; ++i)
     {
          printf("Enter number%d: ",i+1);
          scanf("%d", &a[i]);
     }
}
void delete()
{
    int position,c;
     printf("Enter the location where you wish to delete element\n");
   scanf("%d", &position);

   if ( position >= n+1 )
      printf("Deletion not possible.\n");
   else
   {
      for ( c = position - 1 ; c < n - 1 ; c++ )
         a[c] = a[c+1];

      printf("Resultant array is\n");

      for( c = 0 ; c < n - 1 ; c++ )
         printf("%d\n", a[c]);
   }
}
void insert()
{
     int position,value,c;
      printf("Enter the location where you wish to insert an element\n");
   scanf("%d", &position);

   printf("Enter the value to insert\n");
   scanf("%d", &value);

   for (c = n - 1; c >= position - 1; c--)
      a[c+1] = a[c];

   a[position-1] = value;

   printf("Resultant array is\n");

   for (c = 0; c <= n; c++)
      printf("%d\n", a[c]);
}
void search()
{
    int  first, last, middle, n, search;
   printf("Enter value to find\n");
   scanf("%d", &search);
   first = 0;
   last = n - 1;
   middle = (first+last)/2;

   while (first <= last) {
      if (a[middle] < search)
         first = middle + 1;
      else if (a[middle] == search) {
         printf("%d found at location %d.\n", search, middle+1);
         break;
      }
      else
         last = middle - 1;

      middle = (first + last)/2;
   }
   if (first > last)
      printf("Not found! %d is not present in the list.\n", search);

}
void insertionsort()
{
    int c,d,t=0;
    for (c = 1 ; c <= n - 1; c++) {
    d = c;

    while ( d > 0 && a[d] < a[d-1]) {
      t          = a[d];
      a[d]   = a[d-1];
      a[d-1] = t;

      d--;
    }
  }

  printf("Sorted list in ascending order:\n");

  for (c = 0; c <= n - 1; c++) {
    printf("%d\n", a[c]);
  }
}

int main()
{
    int x;
    printf("MENU");
    printf("1. Creation 2. Insertion 3. Deletion 4. Searching 5. Sorting \n");
    printf("Enter your choice \n");
```

```
      scanf("%d",&x);
      while(x>0 && x<6)
      {
      switch (x)
      {
      case 1:
        create();
        break;
      case 2:
        insert();
        break;
      case 3:
        delete();
        break;
      case 4:
        search();
        break;
      case 5:
        insertionsort();
        break;
      default:
        printf("wrong Input");
      }
printf("Enter your choice if you want to continue \n");
scanf("%d",&x);
      }
}
```

Output:



```
./a.out
 yash@HP-Notebook  ~  g++ array.c
 yash@HP-Notebook  ~  ./a.out
MENU
1. Creation 2. Insertion 3. Deletion 4. Searching 5. Sorting
Enter your choice
1
Enter number of elements 4
Enter number1: 1
Enter number2: 2
Enter number3: 3
Enter number4: 4
Enter your choice if you want to continue
2
Enter the location where you wish to insert an element
3
Enter the value to insert
10
Resultant array is
1
2
10
3
4
Enter your choice if you want to continue
3
Enter the location where you wish to delete element
2
Resultant array is
1
10
3
Enter your choice if you want to continue
```



```
1
10
3
Enter your choice if you want to continue
4
Enter value to find
3
Not found! 3 is not present in the list.
Enter your choice if you want to continue
5
Sorted list in ascending order:
1
3
3
10
Enter your choice if you want to continue
```

Learning Outcome: Array can be manipulated in a no. of ways. Operations like insertion, deletion ,searching take O(1), O(n), O(n) time respectively.In Sorting, insertion sort works in O(n^2), whereas Merge Sort works in O(nlogn). {Here, n is the no of elements}

Aim:

1. To implement a stack using array and perform its operations on it(push,pop,empty,size).
2. To convert an infix expression to Prefix and Postfix.

Description: Stack works on the principle of LIFO, i.e. Last-In First-Out. This means that insertion or deletion in a stack takes place at the same point. Push() operation adds element to the stack whereas Pop() removes the element.

Prefix expression notation requires that all operators precede the two operands that they work on.
Postfix, on the other hand, requires that its operators come after the corresponding operands.

Algorithm:

Infix To PostFix

```
1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
…..3.1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty), push it.
…..3.2 Else, Pop the operator from the stack until the precedence of the scanned operator is less-equal to the precedence of the operator residing on the top of the stack. Push the scanned operator to the stack.
4. If the scanned character is an ‘(‘, push it to the stack.
5. If the scanned character is an ‘)’, pop and output from the stack until an ‘(‘ is encountered.
6. Repeat steps 2-6 until infix expression is scanned.
7. Pop and output from the stack until it is not empty.
```

Infix to Prefix

```
1. Push “)” onto STACK, and add “(“ to end of the A
2. Scan A from right to left and repeat step 3 to 6 for each element of A until the STACK is empty
3. If an operand is encountered add it to B
4. If a right parenthesis is encountered push it onto STACK
5. If an operator is encountered then: a. Repeatedly pop from STACK and add to B each operator (on the top of STACK) which has same or higher precedence than the operator. b. Add operator to STACK
6. If left parenthesis is encontered then
        a. Repeatedly pop from the STACK and add to B (each operator on top of stack until a left parenthesis is encounterd)
        b. Remove the left parenthesis
7. Exit
```

Code for stack implementation:

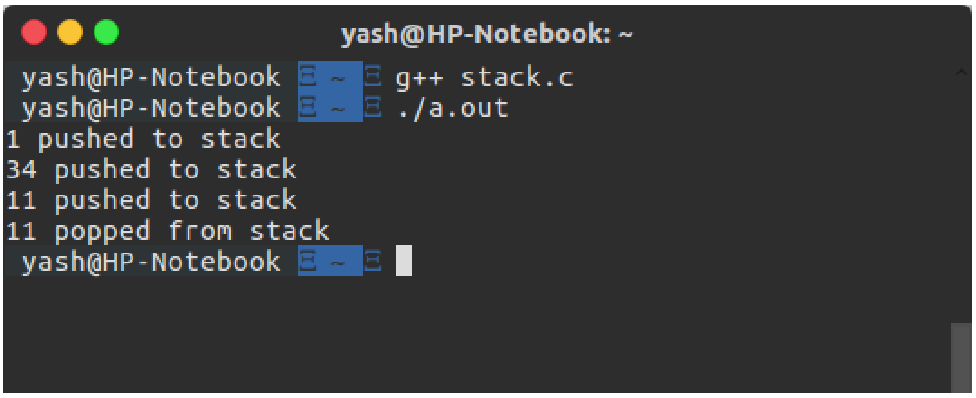
```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

struct Stack
{
   int top;
   unsigned capacity;
   int* array;
};

struct Stack* createStack(unsigned capacity)
{
   struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));
   stack->capacity = capacity;
   stack->top = -1;
   stack->array = (int*) malloc(stack->capacity * sizeof(int));
   return stack;
}

int isFull(struct Stack* stack)
{   return stack->top == stack->capacity - 1; }

int isEmpty(struct Stack* stack)
{   return stack->top == -1;  }

void push(struct Stack* stack, int item)
{
   if (isFull(stack))
      return;
   stack->array[++stack->top] = item;
   printf("%d pushed to stack\n", item);
}

int pop(struct Stack* stack)
{
   if (isEmpty(stack))
      return INT_MIN;
   return stack->array[stack->top--];
}
int main()
{
   struct Stack* stack = createStack(100);
   push(stack, 1);
   push(stack, 34);
   push(stack, 11);

   printf("%d popped from stack\n", pop(stack));

   return 0;
}
```

Code for infix to postfix:

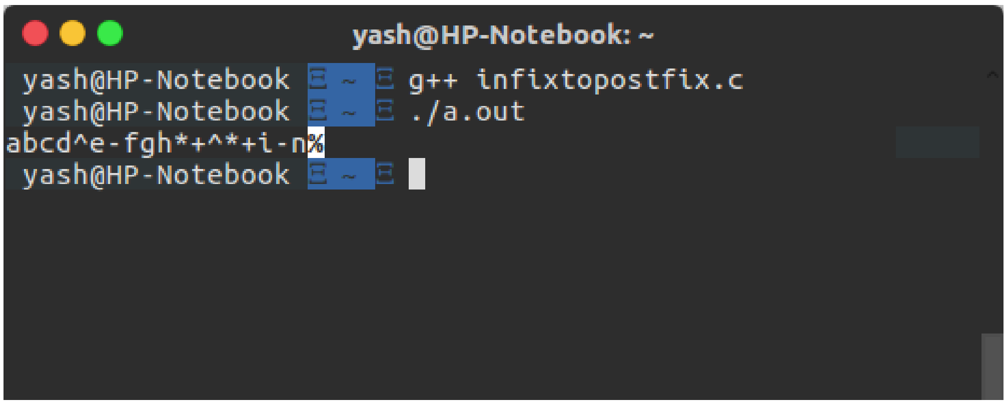
```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

struct Stack
{
   int top;
   unsigned capacity;
   int* array;
};
struct Stack* createStack( unsigned capacity )
{
   struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack));

   if (!stack)
      return NULL;

   stack->top = -1;
   stack->capacity = capacity;

   stack->array = (int*) malloc(stack->capacity * sizeof(int));

   if (!stack->array)
      return NULL;
   return stack;
}
int isEmpty(struct Stack* stack)
{
   return stack->top == -1 ;
}
char peek(struct Stack* stack)
{
   return stack->array[stack->top];
}
char pop(struct Stack* stack)
```

```
{
   if (!isEmpty(stack))
      return stack->array[stack->top--] ;
   return '$';
}
void push(struct Stack* stack, char op)
{
   stack->array[++stack->top] = op;
}

int isOperand(char ch)
{
   return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z');
}
int Prec(char ch)
{
   switch (ch)
   {
   case '+':
   case '-':
      return 1;

   case '*':
   case '/':
      return 2;

   case '^':
      return 3;
   }
   return -1;
}

int infixToPostfix(char* exp)
{
   int i, k;
   struct Stack* stack = createStack(strlen(exp));
   if(!stack)
      return -1 ;

   for (i = 0, k = -1; exp[i]; ++i)
   {
      if (isOperand(exp[i]))
         exp[++k] = exp[i];

      else if (exp[i] == '(')
            push(stack, exp[i]);


        else if (exp[i] == ')')
        {
            while (!isEmpty(stack) && peek(stack) != '(')
                exp[++k] = pop(stack);
            if (!isEmpty(stack) && peek(stack) != '(')
                return -1;
            else
                pop(stack);
        }
        else
        {
            while (!isEmpty(stack) && Prec(exp[i]) <= Prec(peek(stack)))
                exp[++k] = pop(stack);
            push(stack, exp[i]);
        }

    }
    while (!isEmpty(stack))
        exp[++k] = pop(stack );

    exp[++k] = '\0';
    printf( "%sn", exp );
}

int main()
{
    char exp[] = "a+b*(c^d-e)^(f+g*h)-i";
    infixToPostfix(exp);
    return 0;
}
```

Output:



```
 yash@HP-Notebook  ~  g++ stack.c
 yash@HP-Notebook  ~  ./a.out
1 pushed to stack
34 pushed to stack
11 pushed to stack
11 popped from stack
 yash@HP-Notebook  ~ 
```



```
yash@HP-Notebook  ~  g++ infixtopostfix.c
yash@HP-Notebook  ~  ./a.out
abcd^e-fgh*+^*+i-n%
yash@HP-Notebook  ~  
```

<u>Learning outcome</u>: All operations on the stack, work effectively with O(1) time complexity. Problems like ‘STOCK SPAN’ can be solved efficiently using stack.

Aim: To implement a queue using array.

Description: Queue is a data structure which works on FIFO principle, i.e. First-In First-Out, means, insertion takes place at a rear point and insertion takes place at the front.

Algorithm:

Insertion:

```
1. If (REAR == N) Then
2. Print: Overflow
3. Else
4. If (FRONT and REAR == 0) Then
....(a) Set FRONT = 1
....(b) Set REAR = 1
5. Else
6. Set REAR = REAR + 1[End of Step 4 If]
7. QUEUE[REAR] = ITEM
8. Print: ITEM inserted [End of Step 1 If]
9. Exit
```

Deletion:

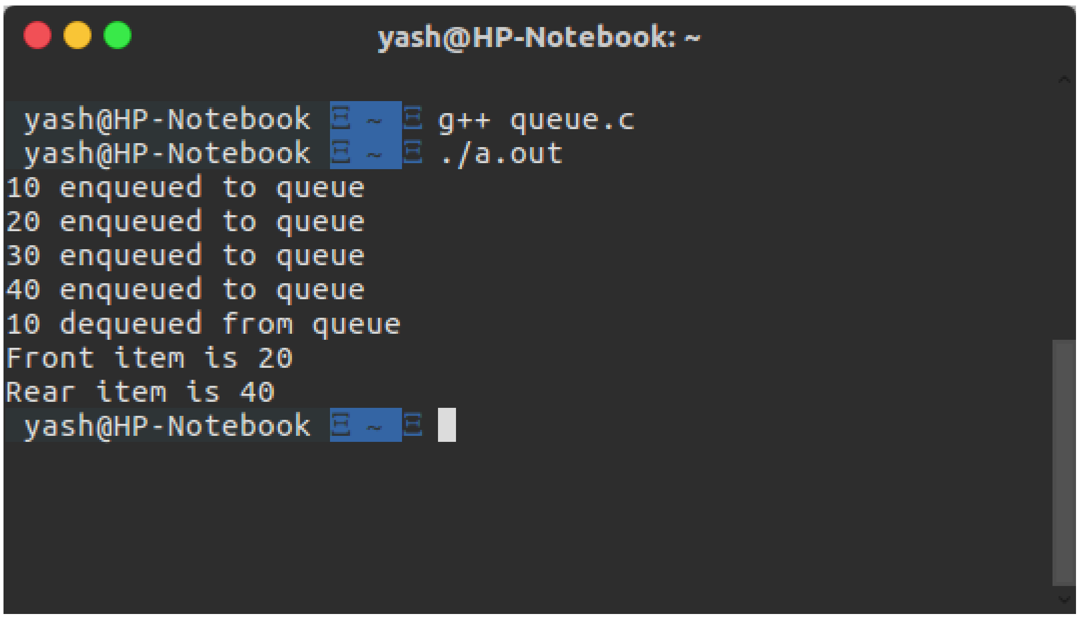
```
1. If (FRONT == 0) Then
2. Print: Underflow
3. Else
4. ITEM = QUEUE[FRONT]
5. If (FRONT == REAR) Then
....(a) Set FRONT = 0
.....(b) Set REAR = 0
6. Else
7. Set FRONT = FRONT + 1 [End of Step 5 If]
8. Print: ITEM deleted [End of Step 1 If]
9. Exit
```

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

struct Queue
{
    int front, rear, size;
    unsigned capacity;
    int* array;
};

struct Queue* createQueue(unsigned capacity)
{
    struct Queue* queue = (struct Queue*) malloc(sizeof(struct Queue));
    queue->capacity = capacity;
    queue->front = queue->size = 0;
    queue->rear = capacity - 1;
    queue->array = (int*) malloc(queue->capacity * sizeof(int));
    return queue;
}

int isFull(struct Queue* queue)
{  return (queue->size == queue->capacity);  }

int isEmpty(struct Queue* queue)
{  return (queue->size == 0); }

void enqueue(struct Queue* queue, int item)
{
    if (isFull(queue))
        return;
    queue->rear = (queue->rear + 1)%queue->capacity;
    queue->array[queue->rear] = item;
    queue->size = queue->size + 1;
    printf("%d enqueued to queue\n", item);
}
int dequeue(struct Queue* queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    int item = queue->array[queue->front];
    queue->front = (queue->front + 1)%queue->capacity;
    queue->size = queue->size - 1;
    return item;
}

int front(struct Queue* queue)
{
    if (isEmpty(queue))
        return INT_MIN;
    return queue->array[queue->front];
}
```

```
int rear(struct Queue* queue)
{
   if (isEmpty(queue))
      return INT_MIN;
   return queue->array[queue->rear];
}

int main()
{
   struct Queue* queue = createQueue(1000);

   enqueue(queue, 10);
   enqueue(queue, 20);
   enqueue(queue, 30);
   enqueue(queue, 40);

   printf("%d dequeued from queue\n", dequeue(queue));

   printf("Front item is %d\n", front(queue));
   printf("Rear item is %d\n", rear(queue));

   return 0;
}
```

Output:



```
 yash@HP-Notebook  ~  g++ queue.c
 yash@HP-Notebook  ~  ./a.out
10 enqueued to queue
20 enqueued to queue
30 enqueued to queue
40 enqueued to queue
10 dequeued from queue
Front item is 20
Rear item is 40
 yash@HP-Notebook  ~ 
```

Learning Outcome: All operations on a queue work in O(1) time complexity. Problems like “Level Wise Print” of elements of a Binary Tree can be solved using Queue.