PDS LAB – 10 (Section-5) Date: 10th April 2017
Stacks, Queues & Files
Tutorial Problems

1. A stack is a data structure used to maintain data in "Last In First Out (ILFO)" principle. Define the suitable structure to represent the stack using array and linked list. Write the following basic functions involved with stack to carry out the following operations:
   (i) Create the stack
   (ii) Check whether stack is empty or full
   (iii) Placing the element into the stack (PUSH)
   (iv) Removing the element from stack (POP)
   (v) Accessing the top element on the stack
   (vi) Print the elements in the stack

2. A queue is a data structure to maintain data in "First In First Out (FIFO)" principle. Define the suitable structure to represent the queue using array and linked list. Write the following basic functions involved with queue to carry out the following operations:
   (i) Create the queue
   (ii) Check whether queue is empty or full
   (iii) Placing the element into the queue (ENQUEUE)
   (iv) Removing the element from queue (DEQUEUE)
   (v) Accessing the front element from the queue
   (vi) Print the elements in the queue

3. Write a C program to create a file, write some data items into the file and read the contents of the file and print them.

Assignment Problems (For All Students)

1. Write a C program to check whether the given expression is correctly parenthesized using stack.
   Input: [a+(b-c)*{d/e}+[f+h]-i]
   Output: The given expression is correctly parenthesized
   Input: [a+(b-c)*{d/e}+[f+h)-i]
   Output: The given expression is incorrectly parenthesized

2. Write a C program to check whether the given string is palindrome or not using stack.

3. Write a C program to enter the data items (say integers) in a queue. After the entry of N items, fetch the items and feed to 2 new queues such that the positive numbers will be in

one queue and negative numbers will be in another queue. At the end show the contents of 2 queues which holds the positive and negative numbers.

4. Write a C program to create a file and enter some N data items in first attempt. Next time open the same file and append say M data items to it. Later, fetch the contents of the file and copy them to 2 new files such that even data items will be copied to a file named "evenfile", and odd items will be copied into "oddfile". After the operation print the contents of original file (initial one) as well as "evenfile" and "oddfile".

5. Write a C program to create a file which contains the records of the employees of an organization. The fields of the record are (i) emp_name, (ii) emp_no, (iii) emp_age, (iv) emp_sal. Enter 4-5 employee records into the file. Access each employee's salary and compute his/her annual income and place it in a new file as fifth field in each of the record. Display the contents of the original file as well as new file.

Assignment Problems (For those who completed 5 assignment problems)

1. Write C program to convert infix expression to postfix expression using stack.

2. Write a C program to implement the Queue using Stacks.

3. Write a C program to create a file and enter student records which contains roll_no, name, age, marks. After entering some N records, access the records and place them in 2 files ("passfile" and "failfile") based on their performance. If the marks are greater than or equal to 40, then the student has passed otherwise he/she has failed.

STACK: Last-In-First-Out (LIFO)

- `void push (stack *s, int element);`

  /* Insert an element in the stack */
- `int pop (stack *s);`

  /* Remove and return the top element */
- `void create (stack  *s);`

  /* Create a new stack */
- `int isempty (stack *s);`

  /* Check if stack is empty */
- `int isfull (stack *s);`

  /* Check if stack is full */

Assumption: stack contains integer elements!

Declaration

```
#define MAXSIZE 100

struct lifo
{
   int st[MAXSIZE];
   int  top;
};
typedef struct lifo
                  stack;
stack s;
```

ARRAY

```
struct lifo
{
   int value;
   struct lifo *next;
};
typedef struct lifo
                   stack;

stack *top;
```

LINKED LIST

Stack Creation

```
void create (stack *s)
{
   s->top = -1;

   /* s->top points to
      last element
      pushed in;
      initially -1 */
}
```

ARRAY

```
void create (stack **top)
{
   *top = NULL;

   /* top points to NULL,
      indicating empty
      stack              */
}
```

LINKED LIST

Pushing an element into stack

```
void push (stack *s, int element)
  {
   if (s->top == (MAXSIZE-1))
   {
     printf (“\n Stack overflow”);
         exit(-1);
     }
     else
     {
         s->top++;
         s->st[s->top] = element;
     }
  }
```

ARRAY

```
void push (stack **top, int element)
{
    stack *new;

    new = (stack *)malloc (sizeof(stack));
    if (new == NULL)
    {
       printf (“\n Stack is full”);
       exit(-1);
    }

    new->value = element;
    new->next = *top;
    *top = new;
}
```

LINKED LIST

Popping an element from stack

```
int pop (stack *s)
  {
     if (s->top == -1)
     {
        printf (“\n Stack underflow”);
        exit(-1);
     }
     else
     {
        return (s->st[s->top--]);
     }
  }
```

ARRAY

```
int pop (stack **top)
{
   int t;
   stack *p;
   if (*top == NULL)
   {
      printf (“\n Stack is empty”);
      exit(-1);
   }
   else
   {
      t = (*top)->value;
      p = *top;
      *top = (*top)->next;
      free (p);
      return t;
   }
}
```

LINKED LIST

Checking for stack empty

```
int isempty (stack *s)
{
   if (s->top == -1)
          return 1;
   else
          return (0);
}
```

ARRAY

```
int isempty (stack *top)
{
   if (top == NULL)
        return (1);
    else
        return (0);
}
```

LINKED LIST

Checking for Stack Full

```
int isempty (stack *s)
{
   if (s->top == -1)
          return 1;
   else
          return (0);
}
```

ARRAY

```
int isempty (stack *top)
{
   if (top == NULL)
        return (1);
    else
        return (0);
}
```

LINKED LIST

Example: A Stack using an Array

```
#include <stdio.h>
#define MAXSIZE 100

struct lifo
{
   int st[MAXSIZE];
   int  top;
};
typedef struct lifo stack;

main() {
  stack A, B;
  create(&A);
  create(&B);
  push(&A,10);
  push(&A,20);
  push(&A,30);
  push(&B,100);
  push(&B,5);

  printf ("%d %d", pop(&A), pop(&B));

  push (&A, pop(&B));

  if (isempty(&B))
    printf ("\n B is empty");
  return;
}
```

Infix to postfix conversion

- Use a stack for processing operators (push and pop operations).
- Scan the sequence of operators and operands from left to right and perform one of the following:
  - output the operand,
  - push an operator of higher precedence,
  - pop an operator and output, till the stack top contains operator of a lower precedence and push the present operator.

The algorithm steps

1. Print operands as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
3. If the incoming symbol is a left parenthesis, push it on the stack.
4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.
7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.
8. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

Infix to Postfix Conversion

Requires operator precedence information

Operands:

Add to postfix expression.

Close parenthesis:

pop stack symbols until an open parenthesis appears.

Operators:

Pop all stack symbols until a symbol of lower precedence appears. Then push the operator.

End of input:

Pop all remaining stack symbols and add to the expression.

Infix to Postfix Rules

Expression:

A * (B + C * D) + E

becomes

A B C D * + * E +

Postfix notation is also called as Reverse Polish Notation (RPN)

	Current symbol	Operator Stack	Postfix string
1	A		A
2	*	*	A
3	(	* (	A
4	B	* (	A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9	)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

QUEUE: First-In-First-Out (LIFO)

```
void enqueue (queue *q, int element);
                    /* Insert an element in the queue */
int dequeue (queue *q);
                    /* Remove an element from the queue */
queue *create();
                    /* Create a new queue */
int isempty (queue *q);
                    /* Check if queue is empty */
int size (queue *q);
                    /* Return the no. of elements in queue */
```

Assumption: queue contains integer elements!

Example :Queue using Linked List

```
struct qnode
{
  int val;
   struct qnode *next;
};

struct queue
{
   struct qnode *qfront, *qrear;
};
typedef struct queue QUEUE;
```

```
void enqueue (QUEUE *q,int element)
{
   struct qnode *q1;
   q1=(struct qnode *)malloc(sizeof(struct qnode));
   q1->val= element;
   q1->next=q->qfront;
   q->qfront=q1;
}
```

Example :Queue using Linked List

```
int size (queue *q)
{
   queue *q1;
   int count=0;
   q1=q;
   while(q1!=NULL)
   {
     q1=q1->next;
     count++;
   }
return count;
}
```

```
int peek (queue *q)
{
   queue *q1;
   q1=q;
   while(q1->next!=NULL)
      q1=q1->next;
return (q1->val);
}
```

```
int dequeue (queue *q)
{
   int val;
   queue *q1,*prev;
   q1=q;
   while(q1->next!=NULL)
   {
      prev=q1;
      q1=q1->next;
   }
   val=q1->val;
   prev->next=NULL;
   free(q1);
return (val);
}
```

File Handling Commands

- Include header file `<stdio.h>` to access all file handling utilities.
- A data type namely FILE is there to create a pointer to a file.

Syntax

```
FILE * fptr;        // fptr is a pointer to file
```

- To open a file, use `fopen()` function

Syntax

```
FILE * fopen(char *filename, char *mode)
```

- To close a file, use `fclose()` function

Syntax

```
int fclose(FILE *fptr);
```

`fopen()` function

- The first argument is a string to characters indicating the name of the file to be opened.
  - The convention of file name should follow the convention of giving file name in the operating system.

Examples:

```
xyz12.c      student.data      File PDS.txt

myFile
```

`fopen()` function

- The second argument is to specify the mode of file opening. There are five file opening modes in C
  - "r" : Opens a file for reading
  - "w" : Creates a file for writing (overwrite, if it contains data)
  - "a" : Opens a file for appending - writing on the end of the file
  - “rb” : Read a binary file (read as bytes)
  - “wb” : Write into a binary file (overwrite, if it contains data)
- It returns the special value NULL to indicate that it couldn't open the file.

`fopen()` function

- If a file that does not exist is opened for writing or appending, it is created as a new.
- Opening an existing file for writing causes the old contents to be discarded.
- Opening an existing file for appending preserves the old contents, and new contents will be added at the end.
- File opening error
  - Trying to read a file that does not exist.
  - Trying to read a file that doesn't have permission.
  - If there is an error, fopen() returns NULL.

Example: fopen( )

```
#include <stdio.h>
void main()
{
   FILE *fptr;                  // Declare a pointer to a file
   char filename[]= "file2.dat";
   fptr = fopen(filename,"w");
//Also, alternatively
//    fptr = fopen ("file2.dat","w");
   if (fptr == NULL) {
      printf ("Error in creating file");
      exit(-1);                 // Quit the function
   }
   else /* code for doing something */
}
```

Reading from a File

- Following functions in C (defined in `stdio.h`) are usually used for reading simple data from a file
  - `fgetc(…)`
  - `fscanf(…)`
  - `fgets(…)`
  - `getc(…)`
  - `ungetc(…)`

Reading from a File: `fgetc()`

Syntax for `fgetc(…)`

```
int fgetc(FILE *fptr)
```

- The `fgetc()` function returns the next character in the stream `fptr` as an `unsigned char` (converted to `int`).
- It returns `EOF` if end of file or error occurs.

```
FILE *fptr;
int c;
/* Open file and check it is open */
while ((c = fgetc(fptr)) != NULL)
   {
     printf ("%c",c);
   }
```

Reading from a File: `fscanf()`

Syntax for `fscanf(...)`

```
int fscanf(FILE *fptr, char *format, ...);
```

- `fscanf` reads from the stream `fptr` under control of format and assigns converted values through subsequent assignments, each of which must be a pointer.
  - It returns when format is exhausted.
- `fscanf` returns `EOF` if end of file or an error occurs before any conversion.
- it returns the number of input items converted and assigned.

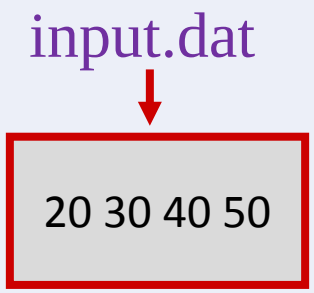
Example: Using `fscanf(…)`

```
FILE *fptr;

fptr= fopen (“input.dat”,“r”);
int n;
/* Check it's open */
if (fptr == NULL)
   {
    printf(“Error in opening file \n”);
   }


  n = fscanf(fptr,“%d %d”,&x,&y);

  ...
```



x = 20
x = 30

Reading from a File: `fgets(…)`

Syntax for `fgets(…)`

```
char *fgets(char *s, int n, FILE *fptr)
```

`s`	The array where the characters that are read will be stored.
`n`	The size of *s*.
`fptr`	The stream to read.

- `fgets()` reads at most `n-1` characters into the array `s`, stopping if a newline is encountered.
  - The newline is included in the array, which is terminated by '\0'.
- The `fgets()` function returns `s` or NULL if EOF or error occurs.

Example: Using `fgets(…)`

```
FILE *fptr;
char line [1000];
/* Open file and check it is open */

while (fgets(line,1000,fptr) != NULL)
   {
   printf ("Read line %s\n",line);
   }
```

Reading a File: `getc(…)`

Syntax for `getc(…)`

```
int getc(FILE *fptr)
```

- `getc(…)` is equivalent to `fgetc(…)` except that it is a macro.

Example: Using `getc(…)`

C program to read a text file and then print the content on the screen.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int ch, fileName[25];
   FILE *fp;
   printf("Enter the name of file you wish to read\n");
   gets(fileName);
   fp = fopen(fileName,"r"); // read mode

  if( fp == NULL )
   {
      printf("Error while opening the file.\n");
      exit(-1);
   }

   printf("The contents of %s f
   while( ( ch = getc(fp) ) !=
      printf("%c",ch);

   fclose(fp);
   return 0;
}
```

OUTPUT

```
Enter the name of file you wish to read
test.txt
The contents of test.txt file are :
C programming is fun.
```

Writing into a File

- Following functions in C (defined in `stdio.h`) are usually used for writing simple data into a file
  - `fputc(…)`
  - `fprintf(…)`
  - `fputs(…)`
  - `putc(…)`

Writing into a File: `fputc(…)`

Syntax for `fputc(…)`

```
int fputc(int c, FILE *fptr)
```

- The `fputc()` function writes the character `c` to file `fptr` and returns the character written, or EOF if an error occurs.

```
#include  <stdio.h>

   filecopy(File *fpIn, FILE *fpOut)
   {
       int c;
       while ((c = fgetc(fpIn) != EOF)
           fputc(c, fpOut);
   }
```

Writing into a File: `fprintf(…)`

Syntax for `fprintf(…)`

```
int fprintf(FILE *fptr, char *format,...)
```

- `fprintf()` converts and writes output to the steam `fptr` under the control of `format`.
- The function is similar to `printf()` function except the first argument which is a file pointer that specifies the file to be written.
- The `fprintf()` returns the number of characters written, or negative if an error occur.

Writing into a File: `fprintf(…)`

```
#include  <stdio.h>

   void main()
   {
      FILE *fptr;
      fptr = fopen(“test.txt”, “w”);

      fprintf(fptr, “Programming in C is really a fun!\n”);
      fprintf(fptr, “Let’s enjoy it\n”);

      fclose(fptr);

      return;
   }
```

Writing into a File: `fputs()`

Syntax for `fputs`:

```
int fputs(char *s, FILE *fptr)
```

- The `fputs()` function writes a string (which need not contain a newline) to a file.
- It returns non-negative, or EOF if an error occurs.

Example: `fputs(…)`

```
#include  <stdio.h>

   void main()
   {
      FILE *fptr;
      fptr = fopen(“test.txt”, “w”);

      fputs(“Programming in C is really a fun!”, fptr);
      fputs(“\n”, fptr);
      fputs(“Let’s enjoy it \n”, fptr);

      fclose(fptr);

      return;
   }
```

Writing into a File: putc(…)

Syntax for putc(…)

```
int putc(FILE *fptr)
```

- The putc() function is same as the putc(…).

```
#include  <stdio.h>

   filecopy(File *fpIn, FILE *fpOut)
   {
       int c;
       while ((c = getc(fpIn) != EOF)
           putc(c, fpOut);
   }
```

Writing into a File: Example

- A sample C program to write some text reading from the keyboard and writing them into a file and then print the content from the file on the screen.

```
#include  <stdio.h>

   main()
   {
       FILE *f1;
       char c;
       printf("Data Input\n\n");
         /* Open the file INPUT */

       f1 = fopen("INPUT", "w");
```

Contd…

Writing into a File

```
    while((c=getchar()) != EOF) /* Get a character from keyboard*/
    putc(c,f1); /* Write a character to IN

    fclose(f1); /* Close the file INPUT*/
    printf("\nData Output\n\n")

    f1 = fopen("INPUT","r"); /* R

    while((c=getc(f1)) != EOF)
    printf("%c",c); /* Display a chara

    fclose(f1); /* Close the file INPUT */

}
```

OUTPUT

```
Data Input
   This is a program to test
the file handling features on
this system

Data Output
   This is a program to test
the file handling features on
this system
```