CONFIDENTIAL

C Programming Basic – week 4,5

For Data Structure and Algorithms

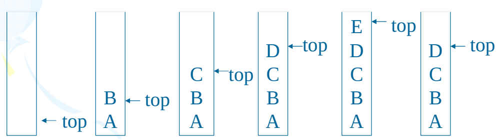
Lecturers :
Cao Tuan Dung
Le Duc Trung
Dept of Software Engineering
Hanoi University of Technology

Topics of this week

- Data structure: Stack
  - Implementation of stack using array
  - Implementation of stack using linked list
- Data structure Queue
  - Implementation of circular queue using array
  - Implementation of queue using linked list
- Exercises on Stack and Queue

Stack

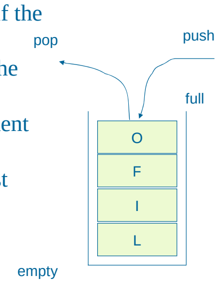
- A stack is a **linear data structure** which can be **accessed only at one of its ends** for storing and retrieving data.
- A LIFO (Last In First Out) structure



Inserting and deleting elements in a stack

Operations on a stack

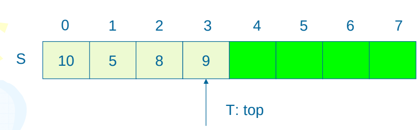
- *initialize(stack)* --- clear the stack
- *empty(stack)* --- check to see if the stack is empty
- *full(stack)* --- check to see if the stack is full
- *push(el,stack)* --- put the element *el* on the top of the stack
- *pop(stack)* --- take the topmost element from the stack
- How to implement a stack?



Separate implementation from specification

- INTERFACE: specify the allowed operations
- IMPLEMENTATION: provide code for operations
- CLIENT: code that uses them.
- Could use either array or linked list to implement stack
- Client can work at higher level of abstraction

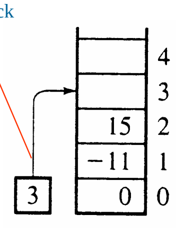
Implementation using array



- Each element is stored as an array's element.
- stack is empty: top= 0
- stack is full: top = Max_Element

Stack specification (stack.h)

Top of stack



(a)

```
#define Max 50
typedef int Eltype;
typedef Eltype StackType[Max];
int top;

void Initialize(StackType stack);
int empty(StackType stack);
int full(StackType stack);
void push(Eltype el, StackType stack);
Eltype pop(StackType stack);
```

array implementation of stack (stack.c)

```
Initialize(StackType stack)
{
   top = 0;
}
empty(StackType stack)
{
   return top == 0;
}
full(StackType stack)
{
   return top == Max;
}
```

```
push(Eltype el, StackType stack)
{
   if (full(*stack))
      printf("stack overflow");
   else stack[top++] = el;
}
Eltype pop(StackType stack)
{
   if (empty(stack))
      printf("stack underflow");
   else return stack[--top];
}
```

stack implementation using structure

- Implementation (c): stack is declared as *a structure* with two fields: one for storage, one for keeping track of the topmost position

```
#define Max 50
typedef int Eltype;
typedef struct StackRec    {
   Eltype storage[Max];
   int top;
};
typedef struct StackRec StackType;
```

stack implementation using structure

```
Initialize(StackType *stack)
{
   (*stack).top=0;
}
empty(StackType stack)

{
   return stack.top == 1;
}
full(StackType stack)
{
   return stack.top == Max;

}
```

```
push(Eltype el, StackType *stack)
{
   if (full(*stack))
      printf("stack overflow");
   else (*stack).storage[
         (*stack).top++]=el;
}
Eltype pop(StackType *stack)
{
 if (empty(*stack))
    printf("stack underflow");
    else return
    (*stack).storage[--(*stack).top];;
}
```

Compile file with library

You'got stack.h, stack.c and test.c

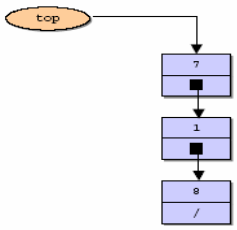
You need to insert this line:
#include "stack.h"
into stack.c and test.c

gcc – c stack.c
gcc –c test.c
gcc – o test.out test.o stack.o

Implementation using linked list

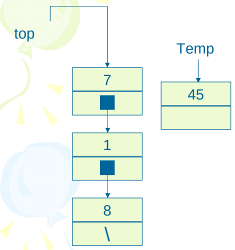
- Implementation of stacks using linked lists are very simple
- The difference between a normal linked list and a stack using a linked list is that some of the linked list operations are not available for stacks
- Being a stack we have only one insert operation called push().
  – In many ways push is the same as insert in the front
- We have also one delete operation called pop()
  – This operation is the same as the operation delete from the front

Pictorial view of stack



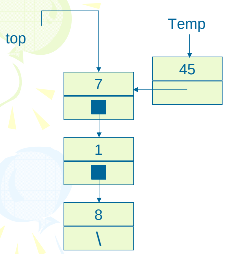

```
struct node {
int data;
struct node *link;
};
```

Push



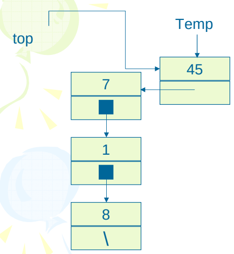

```
struct node *push(struct node *p, int value)
{
  struct node *temp;
  temp=(struct node *)malloc(sizeof(struct
   node));
   if(temp==NULL)  {
    printf("No Memory available Error\n");
    exit(0);
   }
  temp->data = value;
  temp->link = p;
  p = temp;
  return(p);
}
```

Push



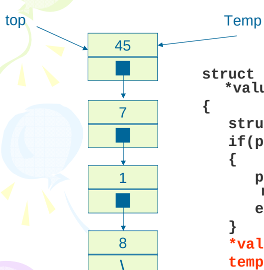

```
struct node *push(struct node *p, int value)
{
  struct node *temp;
  temp=(struct node *)malloc(sizeof(struct
   node));
   if(temp==NULL)  {
    printf("No Memory available Error\n");
    exit(0);
   }
  temp->data = value;
  temp->link = p;
  p = temp;
  return(p);
}
```

Push




```
struct node *push(struct node *p, int value)
{
  struct node *temp;
  temp=(struct node *)malloc(sizeof(struct
   node));
   if(temp==NULL)  {
    printf("No Memory available Error\n");
    exit(0);
   }
  temp->data = value;
  temp->link = p;
  p = temp;
  return(p);
}
```

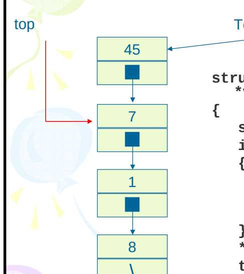
Pop (linked list)




```
struct node *pop(struct node *p, int
  *value)
{
   struct node *temp;
   if(p==NULL)
   {
      printf(" The stack is empty can
       not pop Error\n");
      exit(0);
   }
   *value = p->data;
   temp = p;
   p = p->link;
   free(temp);
   return(p);
}
```

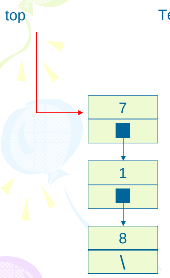
Value at top element need to be save before pop operation

Pop (linked list)




```
struct node *pop(struct node *p, int
  *value)
{
   struct node *temp;
   if(p==NULL)
   {
      printf(" The stack is empty can
       not pop Error\n");
      exit(0);
   }
   *value = p->data;
   temp = p;
   p = p->link;
   free(temp);
   return(p);
}
```

Pop (linked list)




```
struct node *pop(struct node *p, int
   *value)
{
   struct node *temp;
   if(p==NULL)
   {
      printf(" The stack is empty can
       not pop Error\n");
      exit(0);
   }
   *value = p->data;
   temp = p;
   p = p->link;
   free(temp);
   return(p);
}
```

Using stack in program

```
# include <stdio.h>
# include <stdlib.h>
void main()
{
  struct node *top = NULL;
  int n,value;
  do
  {
    do
    {
      printf("Enter the element
      to be pushed\n");
      scanf("%d",&value);
      top = push(top,value);
      printf("Enter 1 to
         continue\n");
      scanf("%d",&n);
    } while(n == 1);

printf("Enter 1 to pop an element\n");
    scanf("%d",&n);
    while( n == 1)
    {
       top = pop(top,&value);
       printf("The value poped is
        %d\n",value);
       printf("Enter 1 to pop an element\n");
       scanf("%d",&n);
    }
    printf("Enter 1 to continue\n");
    scanf("%d",&n);
  } while(n == 1);
}
```

Using stack in program

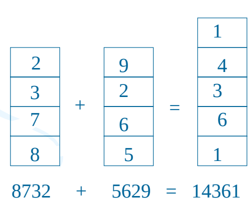
```
printf("Enter 1 to pop an element\n");
    scanf("%d",&n);
    while( n == 1)
    {
       top = pop(top,&value);
     printf("The value poped is %d\n",value);
       printf("Enter 1 to pop an element\n");
       scanf("%d",&n);
    }
    printf("Enter 1 to continue\n");
    scanf("%d",&n);
  } while(n == 1);
}
```

Exercises

- Test the "stack" type that you've defined in a program that read from user a string, then reverse it.

Adding very large numbers

- Treat these numbers as strings of numerals, store the numbers corresponding to these numerals on two stacks, and then perform addition by popping numbers from the stacks



Adding very large numbers: detail algorithm

Read the numerals of the first number and store the numbers corresponding to them on one stack;
Read the numerals of the second number and store the numbers corresponding to them on another stack;
result=*0;*
while at least one stack is not empty
  pop a number from each non-empty stack and add them;
  push the sum (minus 10 if necessary) on the result stack;
  store carry in **result**;
push carry on the result stack if it is not zero;
pop numbers from the result stack and display them;

Exercise 4.1 Stack using array

- We assume that you make a mobile phone's address book.
- Declare a structure "Address" that can hold at least "name", "telephone number" and "e-mail address".
- Write a program that copies data of an address book from a file to another file using a stack. First, read data of the address book from the file and push them on a stack. Then pop data from the stack and write them to the file in the order of popped. In other words, data read first should be read out last and data read last should be read out first.

Exercise 4-2: Conversion to Reverse Polish Notation Using Stacks

- Write a program that converts an expression in the infix notation to an expression in the reverse polish notation. An expression consists of single-digit positive numbers (from 1 to 9) and four operators (+, -, *, /). Read an expression in the infix notation from the standard input, convert it to the reverse polish notation, and output an expression to the standard output. Refer to the textbook for more details about the Reverse Polish Notation.
- For example,

3+5*4
is input, the following will be output.

3 5 4 * +

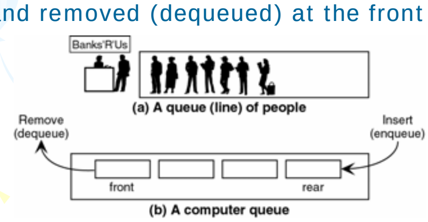
Postfix expression evaluation

- Write a program that reads any postfix expression involving multiplication and addition of interger.
- For example

- ./posteval 5 4 + 6 * => 54

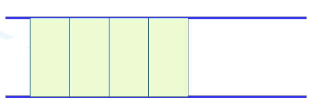
Queue

- A queue is a waiting line
- Both ends are used: one for adding elements and one for removing them.
- Data is inserted (enqueued) at the rear, and removed (dequeued) at the front



Data structure FIFO

- Queue items are removed in exactly the same order as they were added to the queue
  - FIFO structure: First in, First out



front rear

Operations on queue

- *Queue* CreateQ(*max_queue_size*) ::=
  create an empty queue whose maximum size is *max_queue_size*

- *Boolean* IsFullQ(*queue, max_queue_size*) ::=
  if(number of elements in *queue* == *max_queue_size*)
  return *TRUE*
  else return *FALSE*

Operations on queue

- *Queue* EnQ(*queue*, *item*) ::=
  if (IsFullQ(*queue*)) *queue_full*
  else insert *item* at rear of *queue* and return *queue*
- *Boolean* IsEmptyQ(*queue*) ::=
  if (*queue* ==CreateQ(*max_queue_size*))
  return *TRUE*
  else return *FALSE*
- *Element* DeQ(*queue*) ::=
  if (IsEmptyQ(*queue*)) **return**
  else remove and return the *item* at front of queue.

Implementation using array and structure

```
#define MaxLength 100
typedef ... ElementType;
typedef struct {
ElementType Elements[MaxLength];
//Store the elements
int Front, Rear;
} Queue;
```

Initialize and check the status

```
void MakeNull_Queue(Queue *Q){
  Q->Front=-1;
  Q->Rear=-1;
}
int Empty_Queue(Queue Q){
 return Q.Front==-1;
}
int Full_Queue(Queue Q){
return (Q.Rear-Q.Front+1)==MaxLength;
}
```

Enqueue

```
void EnQueue(ElementType X,Queue *Q){
if (!Full_Queue(*Q)){
  if (Empty_Queue(*Q)) Q->Front=0;
  Q->Rear=Q->Rear+1;
  Q->Element[Q->Rear]=X;
}
else printf("Queue is full!");
}
```

Dequeue

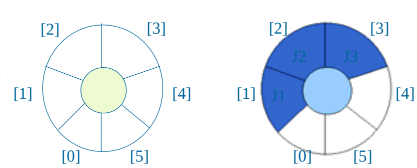
```
void DeQueue(Queue *Q){
  if (!Empty_Queue(*Q)){
     Q->Front=Q->Front+1;
     if (Q->Front > Q->Rear)
     MakeNull_Queue(Q);
     // Queue become empty

}
else printf("Queue is empty!");
}
```

Implementation 2: regard an array as a circular queue

front: one position counterclockwise from the first element

rear: current end

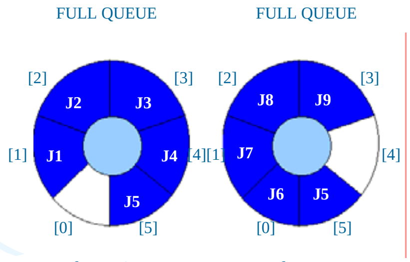
EMPTY QUEUE



front = 0
rear = 0

front = 0
rear = 3

Problem: one space is left when queue is full



front =0
rear = 5

front =4
rear =3

Queue is full or not?

```
int Full_Queue(Queue Q){
 return (Q.Rear-Q.Front+1) %
  MaxLength==0;
}
```

Dequeue

```
void DeQueue(Queue *Q){
if (!Empty_Queue(*Q)){
//if queue contain only one element
  if (Q->Front==Q->Rear) MakeNull_Queue(Q);
  else Q->Front=(Q->Front+1) % MaxLength;
  }
else printf("Queue is empty!");
}
```

Enqueue

```
void EnQueue(ElementType X,Queue *Q){
if (!Full_Queue(*Q)){
  if (Empty_Queue(*Q)) Q->Front=0;
  Q->Rear=(Q->Rear+1) % MaxLength;
  Q->Elements[Q->Rear]=X;
} else printf("Queue is full!");
}
```

Implementation using a List

- Exercise: A Queue, is a list specific. Implement operations on queue by reusing implemented operations of list.

Implementation using a List

```
typedef ... ElementType;
typedef struct Node{
  ElementType Element;
  Node* Next; //pointer to next element
 };
 typedef Node* Position;
 typedef struct{
  Position Front, Rear;
  } Queue;
```

Initialize an empty queue

```
void MakeNullQueue(Queue *Q){
  Position Header;
  Header=(Node*)malloc(sizeof(Node));
  //Allocation    Header
  Header->Next=NULL;
  Q->Front=Header;
  Q->Rear=Header;
}
```

Is-Empty

```
int EmptyQueue(Queue Q){
return (Q.Front==Q.Rear);
}
```

EnQueue

```
void EnQueue(ElementType X, Queue *Q){
  Q->Rear->Next=
  (Node*)malloc(sizeof(Node));
  Q->Rear=Q->Rear->Next;
  Q->Rear->Element=X;
  Q->Rear->Next=NULL;
}
```

DeqQueue

```
void DeQueue(Queue *Q){
if (!Empty_Queue(Q)){
  Position T;
  T=Q->Front;
  Q->Front=Q->Front->Next;
  free(T);
}
else printf(”Error: Queue is empty.”);
}
```

Exercise 4-3: Queues Using Lists

- We assume that you write a mobile phone's address book.
- Declare a structure "Address" that can hold at least "name", "telephone number" and "e-mail address".
- Write a program that copies data of an address book from the file to other file using a queue. First, read data of the address book from the file and add them to the queue. Then retrieve data from the queue and write them to the file in the order of retrieved. In other words, data read in first should be read out first and data read in last should be read out last.

Exercises

- Make a queue that holds integers. The size of the queue is fixed to 10.
- Read integers separated by spaces from the standard input, and add them to the queue. When the program reads the 11th integer, the queue is already full. So the program removes the first integer and adds the 11th integer. Print the removed integer to the standard output.
- Process all the integers in this way.

Exercise:To Do List

- By using a queue, write a To Do List management program with a menu for adding, deleting, modifying elements in the list.
- A work has the following fields:
  - Time
  - Place
  - People
  - Description.
- The time field can be the system time at the moment of input.

Another implementation using array

- Queue CreateQ(*max_queue_size*) ::=

```
# define MAX_QUEUE_SIZE 100
typedef struct {
                int key;    /* other fields */
                } element;
element queue[MAX_QUEUE_SIZE];
int rear = -1;
int front = -1;
Boolean IsEmpty(queue) ::= front == rear
Boolean IsFullQ(queue) ::= rear ==
MAX_QUEUE_SIZE-1
```

Enqueue

- void enq(int *rear, element item)

```
{
/* add an item to the queue */
    if (*rear == MAX_QUEUE_SIZE_1) {
        queue_full( );
        return;
    }
    queue [++*rear] = item;
}
```

Dequeue

- element deq(int *front, int rear)

```
{
    if ( *front == rear)
        return queue_empty( );
            /* return an error key */
    return queue [++ *front];
}
```

Enqueue

```
void addq(int front, int *rear, element item)
{
    *rear = (*rear +1) % MAX_QUEUE_SIZE;
    if (front == *rear) /* reset rear and print
                                error */
    return;
  }
    queue[*rear] = item;
}
```

Dequeue

```
element deleteq(int* front, int rear)
{
    element item;

    if (*front == rear)
        return queue_empty( );
       /* queue_empty returns an error key */

    *front = (*front+1) % MAX_QUEUE_SIZE;
    return queue[*front];
}
```