Abstract Data Types

Definition

- An abstract data type (ADT) is a specification of a set of data and the set of operations that can be performed on the data.
- Such data type is abstract in the sense that it is independent of various concrete implementations.
- Some examples follow.

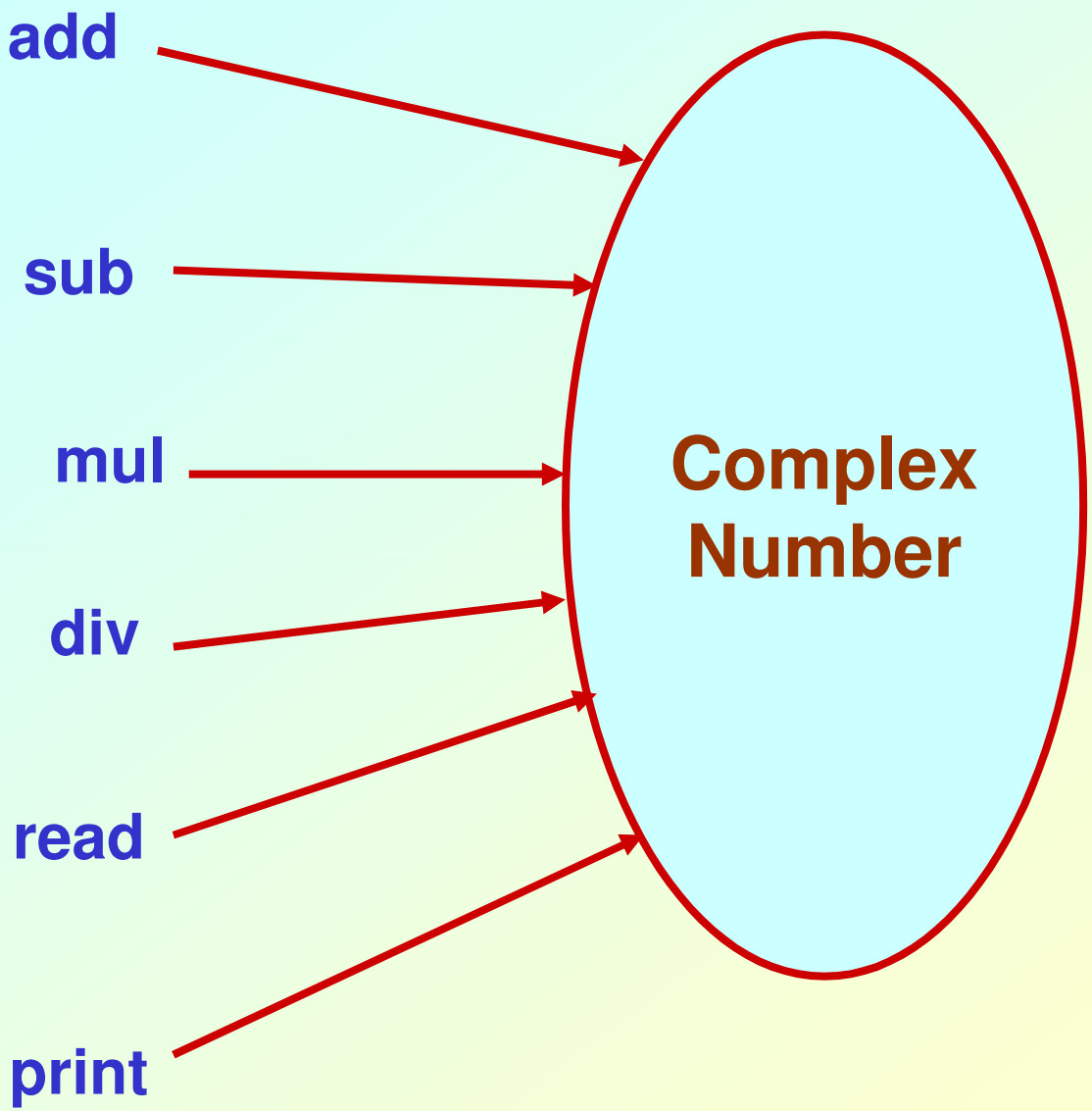
Example 1 :: Complex numbers

```
struct cplx {
              float  re;
              float  im;
            }
typedef struct cplx complex;
```

Structure definition

```
complex *add (complex a, complex b);
complex *sub (complex a, complex b);
complex *mul (complex a, complex b);
complex *div (complex a, complex b);
complex *read();
void print (complex a);
```

Function prototypes


add
sub
mul
div
read
print
Complex
Number

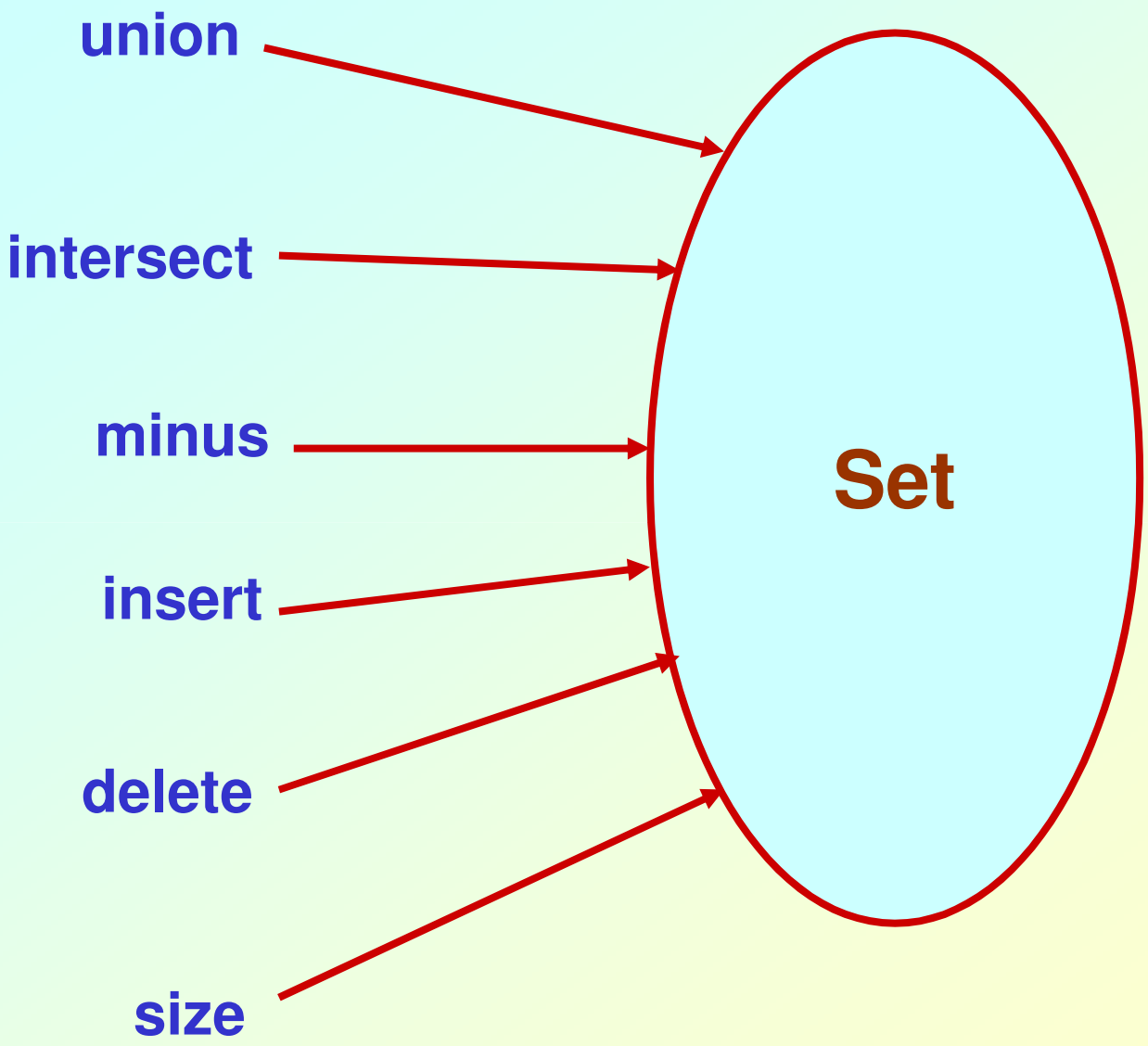
Example 2 :: Set manipulation

```
struct node {
              int element;
              struct node *next;
            }
typedef struct node set;
```

Structure definition

```
set  *union (set a, set b);
set  *intersect (set a, set b);
set  *minus (set a, set b);
void insert (set a, int x);
void delete (set a, int x);
int size (set a);
```

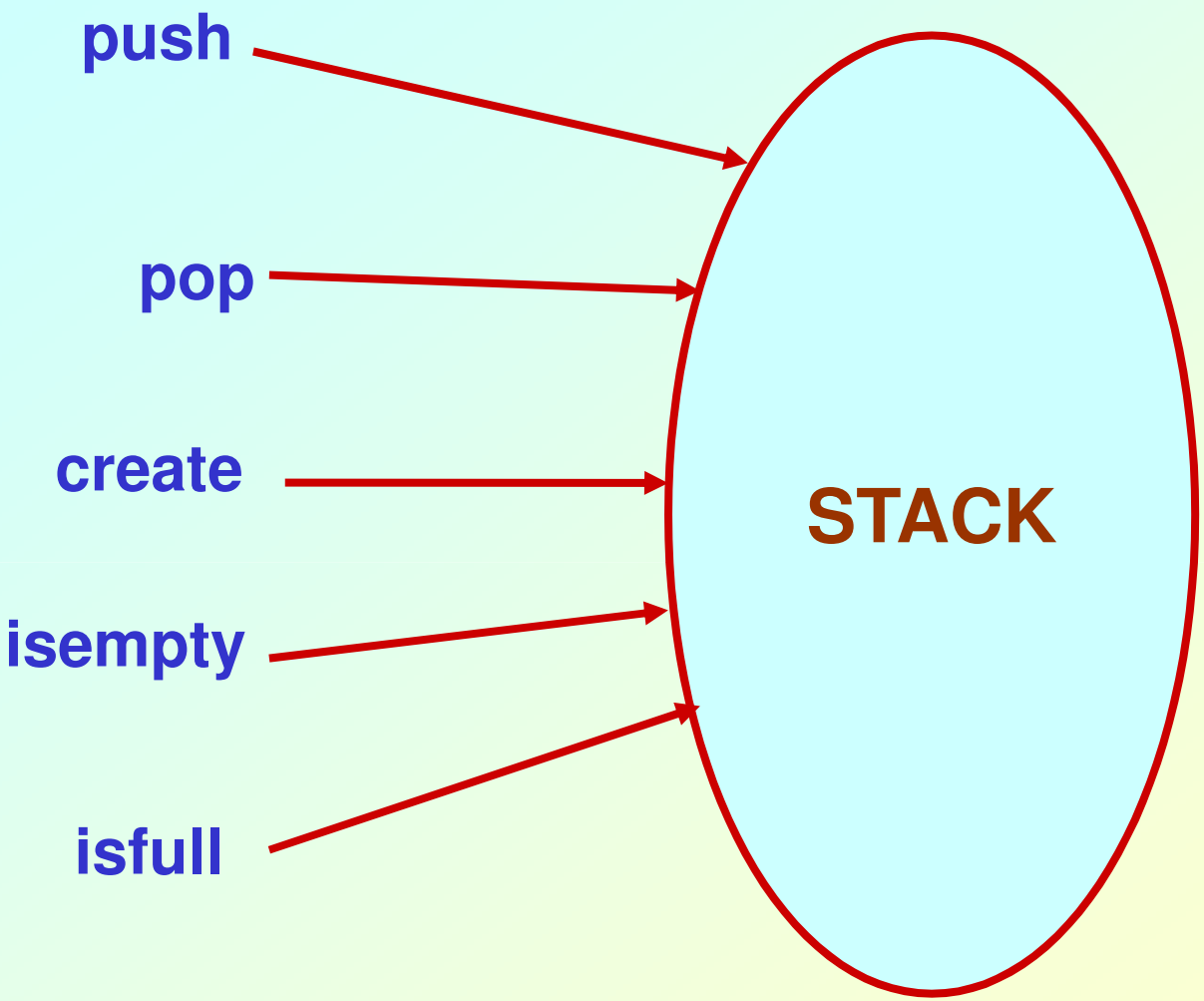
Function prototypes


union
intersect
minus
insert
delete
size
Set

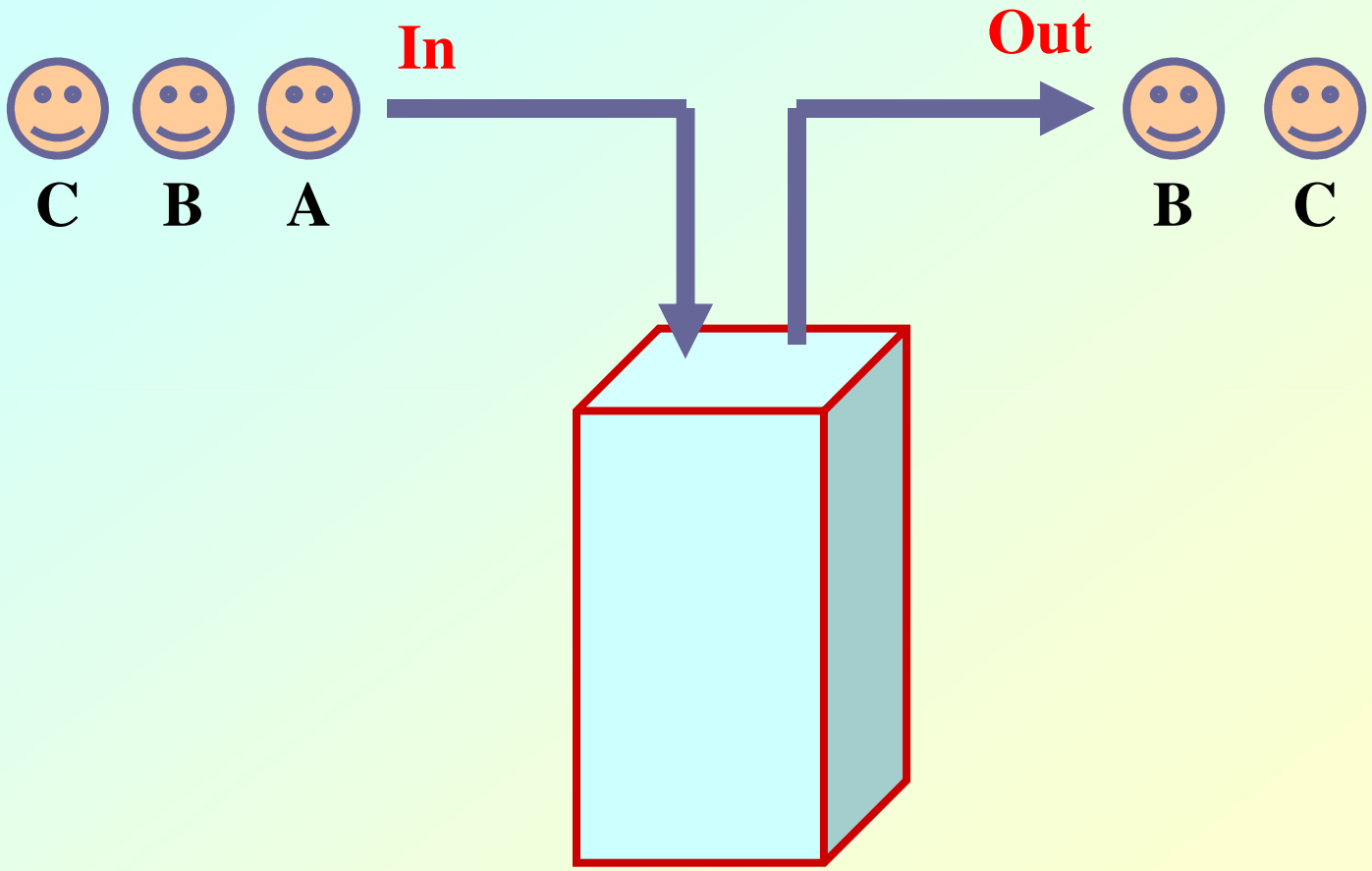
Example 3 :: Last-In-First-Out STACK

Assume:: stack contains integer elements

```
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 */
```


push
pop
create
isempty
isfull
STACK

Visualization of a Stack



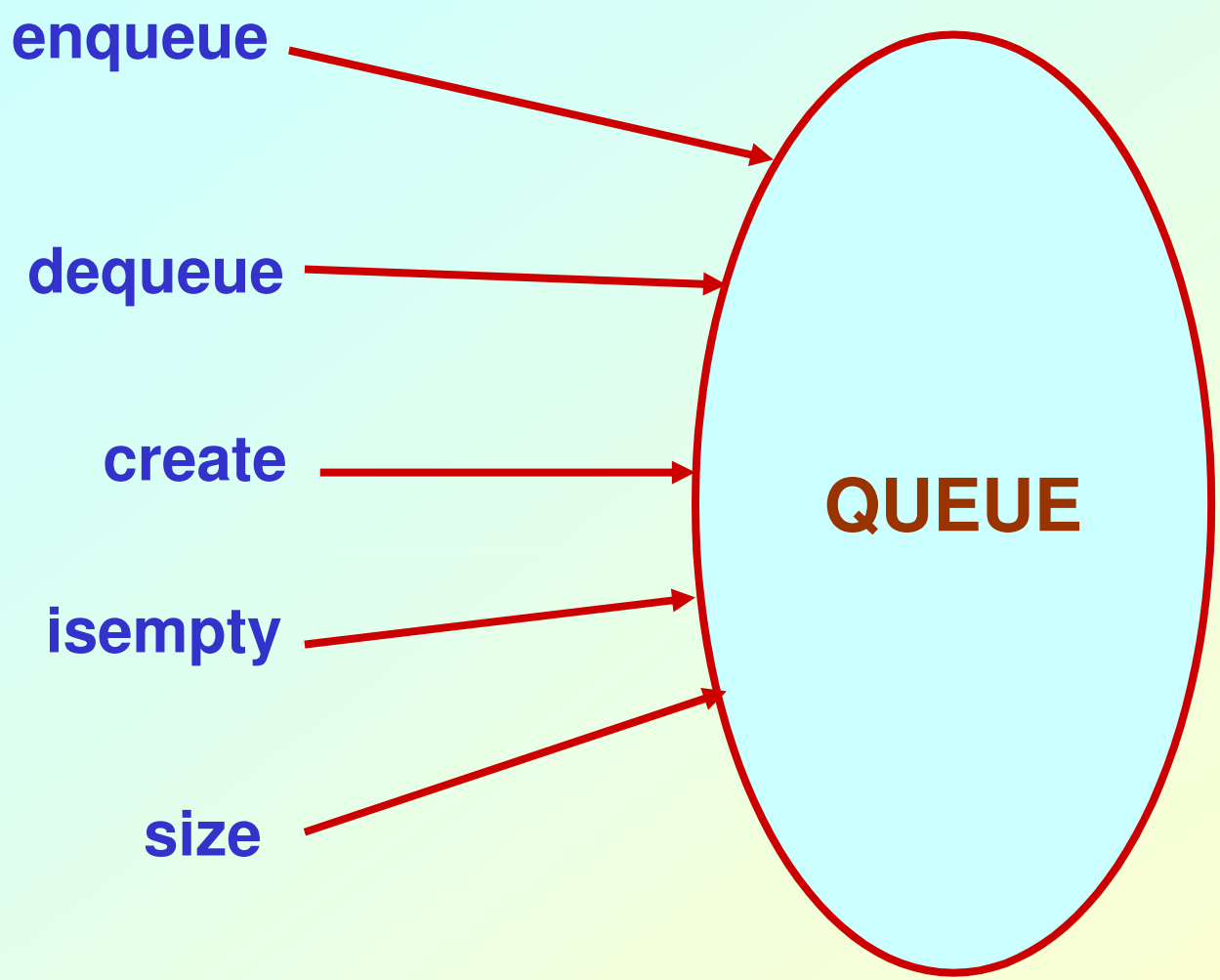
Contd.

- We shall later look into two different ways of implementing stack:
  - Using arrays
  - Using linked list

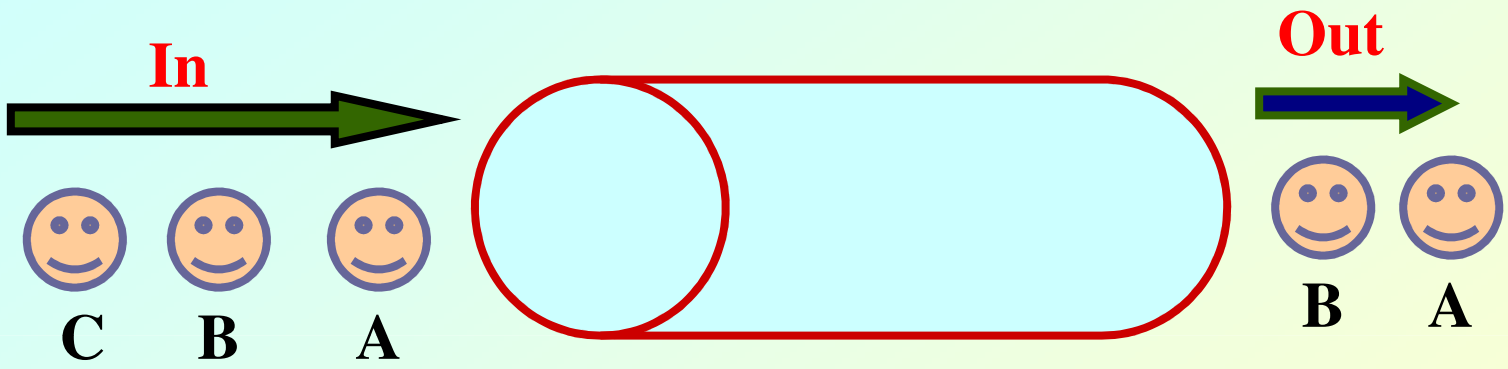
Example 4 :: First-In-First-Out QUEUE

Assume:: queue contains integer elements

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


enqueue
dequeue
create
isempty
size
QUEUE

Visualization of a Queue



Stack Implementation

a) Using arrays
b) Using linked list

Basic Idea

- **In the array implementation, we would:**
  - Declare an array of fixed size (which determines the maximum size of the stack).
  - Keep a variable `top` which always points to the "top" of the stack.
    - Contains the array index of the "top" element.
- **In the linked list implementation, we would:**
  - Maintain the stack as a linked list.
  - A pointer variable `top` points to the start of the list.
  - The first element of the linked list is considered as the stack top.

Declaration

```
#define MAXSIZE 100

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

typedef struct lifo stack;
```

ARRAY

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

typedef struct lifo stack;
```

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 the 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 the 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 isfull (stack s)
{
   if (s.top ==
            (MAXSIZE-1))
        return (1);
   else
        return (0);
}
```

ARRAY

- Not required for linked list implementation.
- In the `push()` function, we can check the return value of `malloc()`.
  - If -1, then memory cannot be allocated.

LINKED LIST

Example main function :: 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 ("\nB is empty");
}
```

Example main function :: linked list

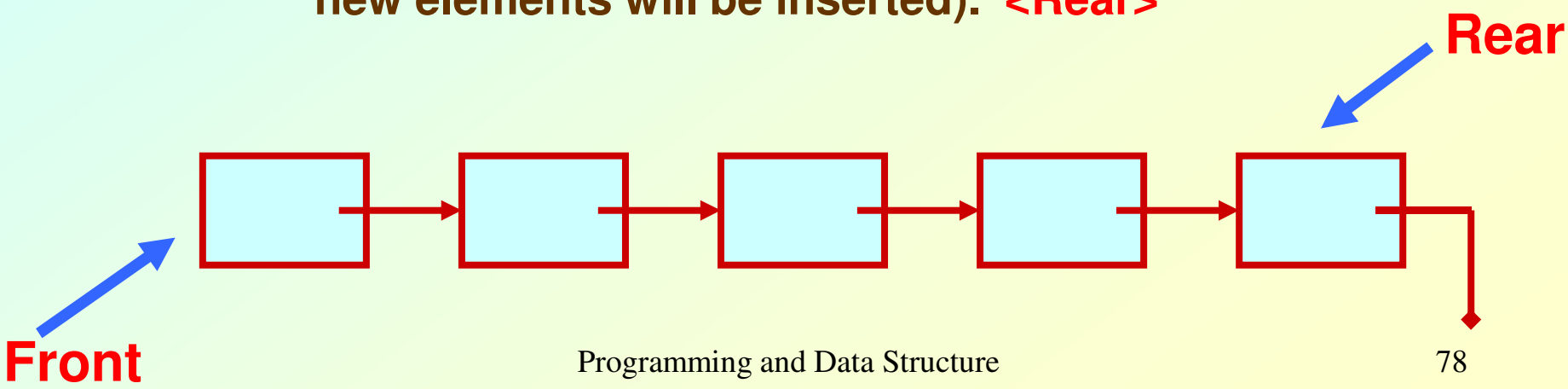
```
#include <stdio.h>
struct lifo
{
    int value;
    struct lifo *next;
};
typedef struct lifo stack;
```

```
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 ("\nB is empty");
}
```

Queue Implementation using Linked List

Basic Idea

- **Basic idea:**
  - Create a linked list to which items would be added to one end and deleted from the other end.
  - Two pointers will be maintained:
    - One pointing to the beginning of the list (point from where elements will be deleted). <Front>
    - Another pointing to the end of the list (point where new elements will be inserted). <Rear>



Declaration

```
struct fifo {
                int  value;
                struct fifo *next;
            };
typedef struct fifo queue;

queue *front, *rear;
```

Creating a queue

```
void createq (queue **front, queue **rear)
{
    *front =  NULL;
    *rear  =  NULL;
}
```

Inserting an element in queue

```
void enqueue (queue **front, queue **rear, int x)
 {
    queue *ptr;
    ptr = (queue *) malloc(sizeof(queue));

    if (*rear == NULL)     /* Queue is empty */
    {
       *front = ptr;
       *rear  = ptr;
       ptr->value = x;
       ptr->next = NULL;
    }
    else                   /* Queue is not empty */
    {
       (*rear)->next = ptr;
       *rear = ptr;
       ptr->value = x;
       ptr->next = NULL;
    }
 }
```

Deleting an element from queue

```
int dequeue (queue **front, queue **rear)
{
   queue *old;   int k;

   if  (*front == NULL)                    /* Queue is empty */
      printf  ("\n Queue is empty");
   else if (*front == *rear)               /* Single element */
        {
            k = (*front)->value;
            free (*front);    front = rear = NULL;
            return (k);
        }
        else
        {
            k = (*front)->value;   old = *front;
            *front = (*front)->next;
            free (old);
            return (k);
        }
  }
```

Checking if empty

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

Example main function

```
#include <stdio.h>
struct fifo
{
   int value;
   struct fifo *next;
};
typedef struct fifo queue;
```

```
main()
{
  queue *Af, *Ar;
  createq (&Af, &Ar);
  enqueue (&Af,&Ar,10);
  enqueue (&Af,&Ar,20);
  enqueue(&Af,&Ar,30);

  printf ("%d %d",
        dequeue (&Af,&Ar),
        dequeue(&Af,&Ar));

  if (isempty(Af))
    printf ("\n Q is empty");
}
```

Some Applications of Stack

Applications ….

- Handling function calls and return
- Handling recursion
- Parenthesis matching
- Evaluation of expressions
  - Polish *postfix* and *prefix* notations