ADTS, STACKS AND QUEUES

Data Type

- **Definition: "A *data type* defines a set of values and a set of operations that can be applied on those values."**
- **Most of the programming languages provide a set of basic data types also called as *atomic data types* or *primitive data types.***

Data Type

- Definition: “A *data type* defines a set of values and a set of operations that can be applied on those values.”
- Most of the programming languages provide a set of basic data types also called as *atomic data types* or *primitive data types.*
- Example 1: Integers: (*int* in C language)

  values : ……-2,-1,0,1,2,…..

  operations : *, +, /, - , %....
- Data type has a particular representation: 1’s Complement, 2’s Complement or Sign magnitude.
- However the representation is abstract to the user (user need not worry about the representation!!!).

Data Type

- **Example 2: Character: (*char* in C language)**
  values : \0,….'A', 'B', 'a', 'b'……
  operations : -, +,…….
- **Representation may be: ASCII, Unicode, or UTF-8….**

Data Type

- **Example 2: Character: (*char* in C language)**
  values : \0,….'A', 'B', 'a', 'b'……
  operations : -, +,…….
- **Representation may be: ASCII, Unicode, or UTF-8….**
- **The opposite of atomic data is** *composite data type*, **which is made up of primitive types.**
- Example: **we may define** point **as a data type which is made up of x, y coordinates where x and y are floating points.**

Abstract Data Type (ADT)

- An abstract data type is a data declaration packaged together with the operations that are meaningful on the data type (composite types).
- In other words, we encapsulate the data and the operation on data and we hide them from the user.

Abstract Data Type (ADT)

- An abstract data type is a data declaration packaged together with the operations that are meaningful on the data type (composite types).
- In other words, we encapsulate the data and the operation on data and we hide them from the user.
- ADT has
  1. Declaration of data (set of values on which it operates)
  2. Declaration of operation(set of functions)

  and hides the representation and implementation details

Arrays vs. Lists

- Array as a Data structure: **It is an ordered set which consist of** ***fixed*** *number of Objects.* ***Operations which can be performed on arrays are:***
  - ***Create an array of some fixed size,***
  - ***Store elements, retrieve elements, destroy an array***
  - ***No insertion or deletion possible (fixed size)!!!!***
- List: **Ordered set consisting of variable number of objects.**

Arrays vs. Lists

- ***Operations which can be performed on Lists are:***
  - ***Create a List***
  - ***Insert elements, delete elements***
  - ***destroy a List***
  - ***Size is Not fixed size!!!!***

Array as Abstract Data Type

- ADT Array
- objects: A set of pairs <*index, value*> where for each value of *index* there is a value from the set *item*. *Index* is a finite ordered set of one or more dimensions, for example, {0, … , n-1} for one dimension, {(0,0),(0,1),….,….,(2,1),(2,2)} for two dimensions,
  etc.
- Functions: for all A ∈ Array, i ∈ index, x ∈ item, j, size ∈ integer

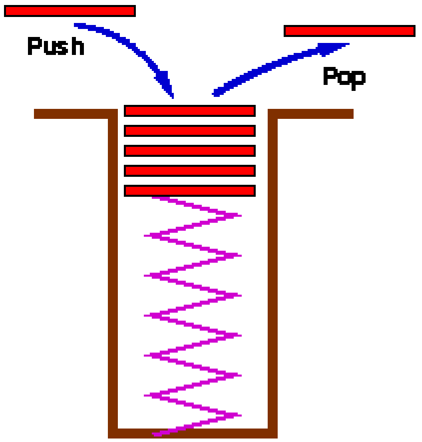
  Array Create(j, list) ::= return an array of j dimension where list is a
  j-tuple whose ith element is the size of ith dimension

  Item Retrieve(A, i) ::= if i ∈ index retrieve the element from array A indexed
  by i

  Array Store(A, i, x) ::= if i ∈ index store the value x at ith index in the array A
- End Array

Stacks

- Definition: **A Stack is an ordered list in which insertions and deletions are made at one end called the top.**
- **Insertion is called as PUSH**
- **Deletion of an element is called as POP**
- **Stack is also called as Last In First Out (LIFO) list.**



stack: a Last-In-First-Out (LIFO) list

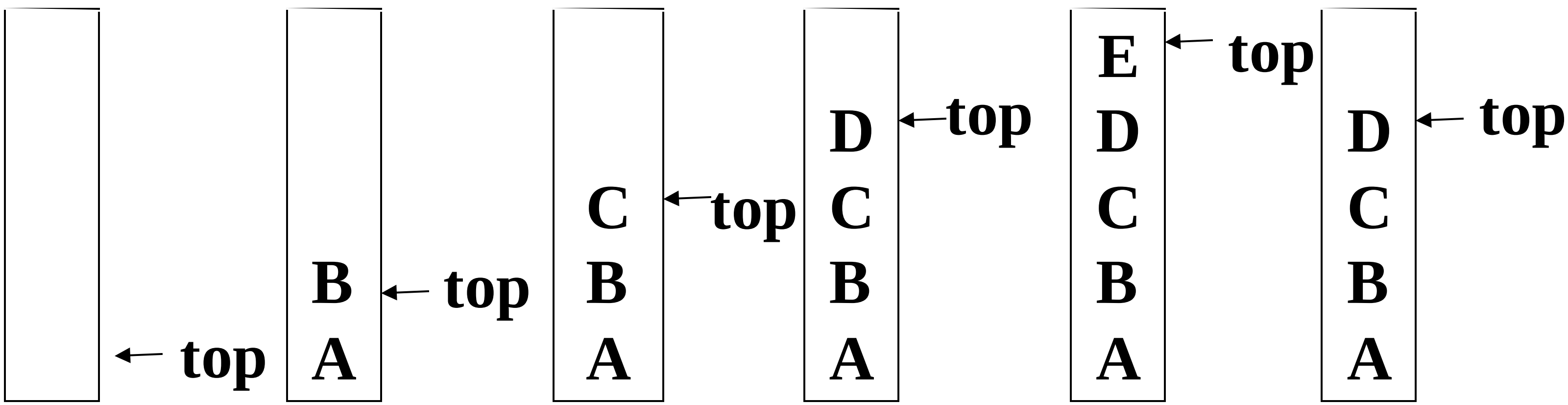


*Figure 3.1: **Inserting and deleting elements in a stack (p.102**

abstract data type for stack

ADT ***Stack* is**

objects: **a finite ordered list with zero or more elements.**

functions:

for all *stack* $\in$ *Stack*, *item* $\in$ *element*, *max_stack_size* $\in$ positive integer

abstract data type for stack

ADT ***Stack*** **is**

objects: **a finite ordered list with zero or more elements.**

functions:

for all *stack* ∈ *Stack*, *item* ∈ *element*, *max_stack_size* ∈ positive integer

Stack **CreateS(*max_stack_size*) ::=**

create an empty stack whose maximum size is *max_stack_size*

abstract data type for stack

ADT ***Stack*** **is**

objects: **a finite ordered list with zero or more elements.**

functions:

for all *stack* ∈ *Stack*, *item* ∈ *element*, *max_stack_size* ∈ positive integer

Stack **CreateS(*max_stack_size*) ::=**

create an empty stack whose maximum size is *max_stack_size*

Boolean **IsFull(*stack, max_stack_size*) ::=**

if **(number of elements in *stack == max_stack_size*)**

return **TRUE**

else return **FALSE**

abstract data type for stack

ADT ***Stack*** **is**

objects: **a finite ordered list with zero or more elements.**

functions:

for all *stack* ∈ *Stack*, *item* ∈ *element*, *max_stack_size* ∈ positive integer

Stack **CreateS(*max_stack_size*) ::=**

create an empty stack whose maximum size is *max_stack_size*

Boolean **IsFull(*stack, max_stack_size*) ::=**

if **(number of elements in *stack* == *max_stack_size*)**

return **TRUE**

else return **FALSE**

Stack **Push(*stack, item*) ::=**

if **(IsFull(*stack*)) *stack_full***

else **insert *item* into top of *stack* and** return

Boolean **IsEmpty(*stack*) ::=**

if**(*stack* == CreateS(*max_stack_size*))**

return **TRUE**

else return **FALSE**

***Boolean* IsEmpty(*stack*) ::=**

if**(*stack* == CreateS(*max_stack_size*))**

return **TRUE**

else return **FALSE**

***Element* Pop(*stack*) ::=**

if**(IsEmpty(*stack*))** return

else **remove and return the *item* on the top of the stack.**

Implementation: **using array**

Stack CreateS(max_stack_size) **::=**

```
#define MAX_STACK_SIZE 100 /* maximum stack size */
typedef struct {
        int key;
        /* other fields */
        } element;
element stack[MAX_STACK_SIZE];
int top = -1;
```

Implementation: **using array**

```
Stack CreateS(max_stack_size) ::=
  #define MAX_STACK_SIZE 100 /* maximum stack size */
  typedef struct {
          int key;
          /* other fields */
          } element;
  element stack[MAX_STACK_SIZE];
  int top = -1;

  Boolean IsEmpty(Stack) ::= top< 0;

  Boolean IsFull(Stack) ::= top >= MAX_STACK_SIZE-1;
```

Add to a stack

```
void push(int top, element item)
{
 /* add an item to the global stack */
    if (top >= MAX_STACK_SIZE-1) {
        stack_full( );
    }
    stack[++top] = item;
}
```

Add to a stack

```
void push(int top, element item)
{
 /* add an item to the global stack */
    if (top >= MAX_STACK_SIZE-1) {
          stack_full( );
    }
    stack[++top] = item;
}

Void stack_full()
{
             fprintf(stderr, “Stack is full, cannot add element”);
             exit(EXIT_FAILURE);
}
```

Delete from a stack

```
element pop(int *top)
{
 /* return the top element from the stack */
    if (top == -1)
       return stack_empty( );  /* returns and error key */
    return stack[(top)--];
 }
```

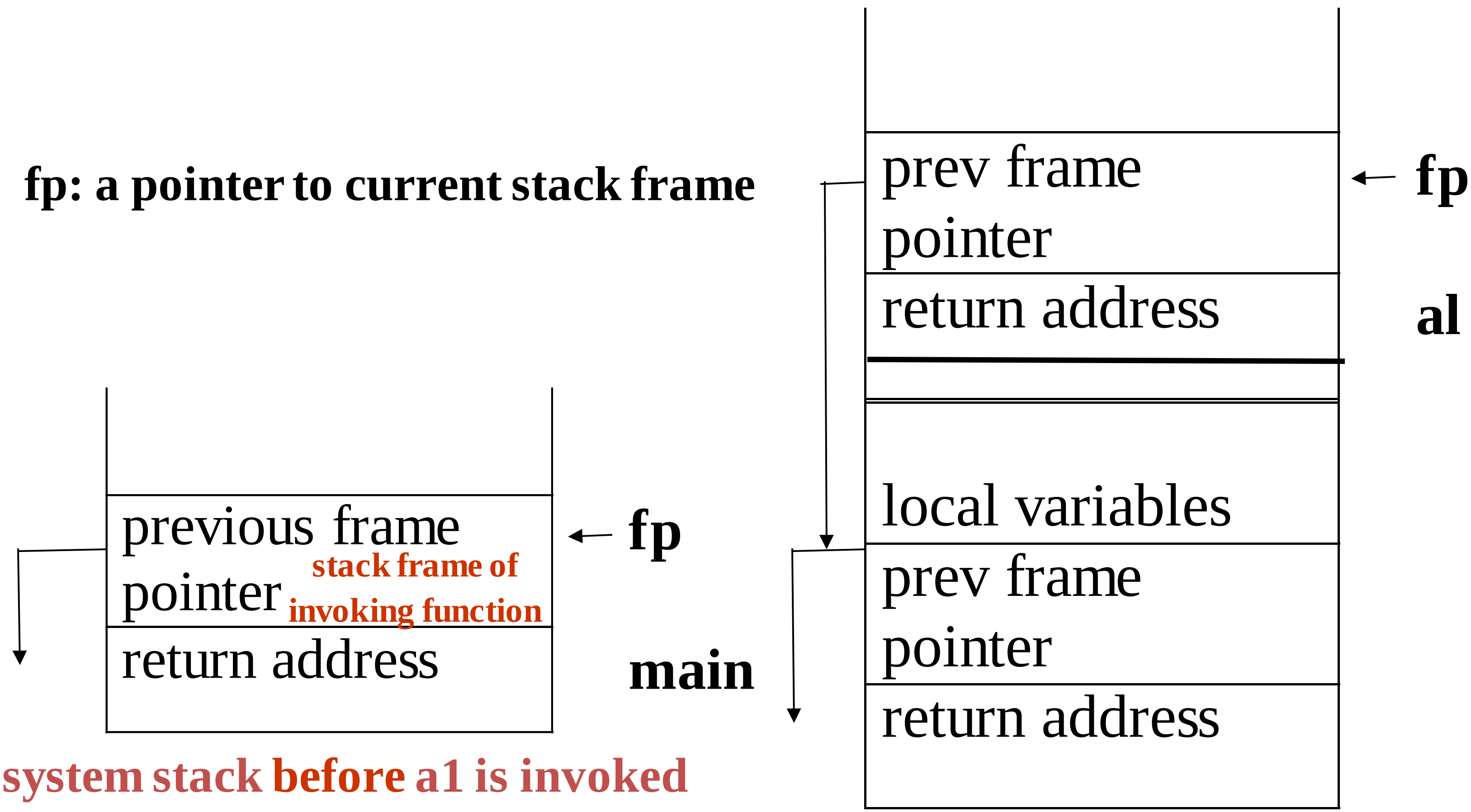
Stack - Relavance

- **Stacks appear in computer programs**
  - **Key to call / return in functions & procedures**
  - **Stack frame allows recursive calls**
  - **Call: push stack frame**
  - **Return: pop stack frame**

Stack - Relavance

- Stacks appear in computer programs
  - Key to call / return in functions & procedures
  - Stack frame allows recursive calls
  - Call: push stack frame
  - Return: pop stack frame
- Stack frame
  - Function arguments
  - Return address
  - Local variables

An application of stack: stack frame of function call (activation record)



system stack before a1 is invoked

(a)

(b)

system stack after a1 is invoked

System stack after function call a1

Infix expression

- **In an expression if the binary operator, which performs an operation , is written in between the operands it is called an infix expression. Ex: a+b*c**

Infix prefix and postfix expression

- In an expression if the binary operator, which performs an operation , is written in between the operands it is called an **infix expression**. Ex: a+b*c
- If the operator is written before the operands , it is called **prefix expression** Ex: +a*bc
- If the operator is written after the operands, it is called **postfix expression**. Ex: abc* +

Infix, prefix, and Postfix expression

- **An expression in infix form is dependent of precedence during evaluation**
- Ex: to evaluate a+b*c, sub expression a+b can be evaluated only after evaluating b*c.

- **As soon as we get an operator we cannot perform the operation specified on the operands.**
- So it takes more time for compilers to check precedence to evaluate sub expression.

Infix, prefix, and Postfix expression

- **Both prefix and postfix representations are independent of precedence of operators.**

- In a single scan an entire expression can be evaluated
- **Takes less time to evaluate.**
  - However infix expressions have to be converted to postfix or prefix.

Evaluation of Expressions

X = a / b - c + d * e - a * c

a = 4, b = c = 2, d = e = 3

Interpretation 1:

((4/2)-2)+(3*3)-(4*2)=0 + 8+9=1

Interpretation 2:

(4/(2-2+3))*(3-4)*2=(4/3)*(-1)*2=-2.66666…

Evaluation of Expressions

X = a / b - c + d * e - a * c

a = 4, b = c = 2, d = e = 3

Interpretation 1:

((4/2)-2)+(3*3)-(4*2)=0 + 8+9=1

Interpretation 2:

(4/(2-2+3))*(3-4)*2=(4/3)*(-1)*2=-2.66666…

How to generate the machine instructions corresponding to a given expression?

precedence rule + associative rule

Token	Operator	Precedence[1]	Associativity
( ) [ ] -> .	function call array element struct or union member	17	left-to-right
— ++	increment, decrement[2]	16	left-to-right
— ++ ! - - + & * sizeof	decrement, increment[3] logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	mutiplicative	13	Left-to-right

+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
|	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
||	logical or	4	left-to-right

?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^= |=	assignment	2	right-to-left
,	comma	1	left-to-right

user compiler

Infix	Postfix
2+3*4 a*b+5 (1+2)*7 a*b/c (a/(b-c+d))*(e-a)*c a/b-c+d*e-a*c	234*+ ab*5+ 12+7*

*Figure 3.13: **Infix and postfix notation (p.120)**

Postfix: **no parentheses, no precedence** user compiler

Infix	Postfix
2+3*4	234*+
a*b+5	ab*5+
(1+2)*7	12+7*
a*b/c	ab*c/
(a/(b-c+d))*(e-a)*c	abc-d+/ea-*c*
a/b-c+d*e-a*c	ab/c-de*ac*-

Postfix: no parentheses, no precedence

Infix to Postfix Conversion

(Intuitive Algorithm/ Manual method)

(1) Fully parenthesize expression

a / b - c + d * e - a * c -->

((((a / b) - c) + (d * e)) - a * c))

(2) All operators replace their corresponding right parentheses.

((((a / b) - c) + (d * e)) - a * c))

/ - * + * -

(3) Delete all parentheses.

ab/c-de*+ac*-

two passes

Infix to postfix conversion: Sample Exercises

- **Convert the following infix expression to postfix expression**
- **a+b*c+d*e**
- **a*b+5**
- **(a/(b-c+d))*(e-a)*c**
- **a/b-c+d*e-a*c**

Evaluation of Postfix Using Stack

Token	Stack [0]	[1]	[2]	Top
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
-	6/2-3			0
4	6/2-3	4		1
2	6/2-3	4	2	2
*	6/2-3	4*2		1
+	6/2-3+4*2			0

Evaluate postfix expression

Assumptions:

operators: +, -, *, /, %   operands: single digit integer

Evaluate postfix expression

Assumptions:

operators: +, -, *, /, %   operands: single digit integer

```
#define MAX_STACK_SIZE 100 /* maximum stack size */
#define MAX_EXPR_SIZE 100 /* max size of expression */

int stack[MAX_STACK_SIZE]; /* global stack */
char expr[MAX_EXPR_SIZE]; /* input string -- expression */
```

Evaluate postfix expression

Assumptions:
operators: +, -, *, /, %   operands: single digit integer

```
#define MAX_STACK_SIZE 100 /* maximum stack size */
#define MAX_EXPR_SIZE 100 /* max size of expression */

typedef enum{lparan, rparen, plus, minus, times, divide,
             mod, eos, operand} precedence;

int stack[MAX_STACK_SIZE]; /* global stack */
char expr[MAX_EXPR_SIZE]; /* input string -- expression */
```

```
int eval(void)
{
/* evaluate a postfix expression, expr, maintained as a
   global variable,
  '\0' is the the end of the expression.

   The stack and top of the stack are global variables.




*/
}
```

```
int eval(void)
{
/* evaluate a postfix expression, expr, maintained as a
   global variable,
  '\0' is the the end of the expression.

   The stack and top of the stack are global variables.

   get_token() is used to return the token type and
   the character symbol.

  Operands are assumed to be single character digits */
}
```

```
precedence get_token(char *symbol, int *n)
{
/* get the next token,

   symbol is the character representation, which is
   returned,

   the token is  represented by its enumerated value, which
   is returned in the function name */
}
```

```
int eval(void)
{
  precedence token;
  char symbol;
  int op1, op2;
  int n = 0;  /* counter for the expression string */
  int top = -1;

 // Scan left to right
          //If operand push (single digit)
          // If operator pop 2 operands; push the op result
//If End of Expression, pop the result
```

```
int eval(void)
{
  precedence token;
  char symbol;
  int op1, op2;
  int n = 0;  /* counter for the expression string */
  int top = -1;

 // Scan left to right
  token = get_token(&symbol, &n);
  while (token != eos)  {
     if (token == operand)
         push(&top, symbol-'0');   /* push operand */
```

```
else {
        /* operator:  remove two operands*/
  op2 = pop(&top);
  op1 = pop(&top);
  switch(token) {    /* perform operation; result to stack */
      case plus:          push(&top, op1+op2); break;
      case minus:         push(&top, op1-op2); break;
      case times:         push(&top, op1*op2); break;
      case divide:        push(&top, op1/op2); break;
      case mod:           push(&top, op1%op2);
  }
}
token = get_token (&symbol, &n);
} /* End of Expression*/
return pop(&top); /* return result  from the stack */
}
```

```
precedence get_token(char *symbol, int *n)
{
 *symbol =expr[(*n)++];

 switch (*symbol)  {
   case '(' : return lparen;
   case ')' : return rparen;
   case '+': return plus;
   case '-' : return minus;
```

```
    case '/' :  return divide;
    case '*' : return times;
    case '%' : return mod;
    case '\0' : return eos;
    default  : return operand;
               /* no error checking, default is operand */
    }
}
```

Infix to Postfix Conversion (Using Stack)

Token	Stack			Top	Output
	[0]	[1]	[2]		
a				-1	a
+	+			0	a
b	+			0	ab
*	+	*		1	ab
c	+	*		1	abc
eos				-1	abc*+

*Figure 3.15: **Translation of a+b*c to postfix (p.124)**

The orders of operands in infix and postfix are the same.

a + b * c, * > +

$a *_1 (b + c) *_2 d$

Token	Stack [0]	Stack [1]	Stack [2]	Top	Output
a				-1	a
$*_1$	$*_1$			0	a
(	$*_1$	(		1	a
b	$*_1$	(		1	ab
+	$*_1$	(	+	2	ab
c	$*_1$	(	+	2	abc
)	$*_1$	**match)**		0	abc+
$*_2$	$*_2$	$*_1 = *_2$		0	abc+$*_1$
d	$*_2$			0	abc+$*_1$d
eos	$*_2$			0	abc+$*_1$d$*_2$

* Figure 3.16: **Translation of a*(b+c)*d to postfix (p.124)**

Rules

(1) Operators are taken out of the stack as long as their in-stack precedence is higher than or equal to the incoming precedence of the new operator.

(2) (has low in-stack precedence, and high incoming precedence.

	(	)	+	-	*	/	%	eos
isp	0	19	12	12	13	13	13	0
icp	20	19	12	12	13	13	13	0

```
precedence stack[MAX_STACK_SIZE];
/* isp and icp arrays -- index is value of precedence
lparen, rparen, plus, minus, times, divide, mod, eos */
static int isp [ ] = {0, 19, 12, 12, 13, 13, 13, 0};
static int icp [ ] = {20, 19, 12, 12, 13, 13, 13, 0};
```

isp: in-stack precedence
icp: incoming precedence

```
void postfix(void)
{
/* output the postfix of the expression. The expression
   string, the stack, and top are global */
  char symbol;
  precedence token;
  int n = 0;
  int top = 0; /* place eos on stack */
  stack[0] = eos;

   // Scan left to right
       //If operand print.
          else
          //If rpar, unstack tokens and print until lpar;
                         discard lpar.
            else remove and print symbols if isp>=icp  else push()
  //Unstack remaining symbols and print
```

```
void postfix(void)
{
/* output the postfix of the expression. The expression
    string, the stack, and top are global */
   char symbol;
   precedence token;
   int n = 0;
   int top = 0; /* place eos on stack */
   stack[0] = eos;

   for (token = get _token(&symbol, &n); token != eos;
                 token = get_token(&symbol, &n)) {
     if (token == operand)
       printf ("%c", symbol);
```

```
else if (token == rparen ){
  /*unstack tokens until left parenthesis */
   while (stack[top] != lparen)
     print_token(pop(&top));
   pop(&top); /*discard the left parenthesis */
  }
```

```
else if (token == rparen ){
  /*unstack tokens until left parenthesis */
   while (stack[top] != lparen)
      print_token(pop(&top));
   pop(&top); /*discard the left parenthesis */
  }
  else{
   /* remove and print symbols whose isp is greater
      than or equal to the current token's icp */
   while(isp[stack[top]] >= icp[token] )
      print_token(pop(&top));

   push(&top, token);
  }
 }
```

```
 while ((token = pop(&top)) != eos)
    print_token(token);
 print("\n");
}
```

*Program 3.11: **Function to convert from infix to postfix (p.126)**

Infix to postfix conversion using stack

- Convert the following infix expression to postfix expression using a stack . Show the instances of stack during conversion.
- **a+b*c+d*e**
- **a*b+5**
- **((a/(b-c+d))*(e-a)*c**
- **a/b-c+d*e-a*c**

Infix	Prefix
a*b/c a/b-c+d*e-a*c a*(b+c)/d-g	/*abc -+-/abc*de*ac -/*a+bcdg

(1) evaluation
(2) transformation

*Figure 3.17: **Infix and postfix expressions (p.127)**

Prefix to Postfix

Read the Prefix expression in reverse order

(from right to left)

If the symbol is an operand,

then push it onto the Stack

Prefix to Postfix

Read the Prefix expression in reverse order

(from right to left)

If the symbol is an operand,

then push it onto the Stack

If the symbol is an operator,

then pop two operands from the Stack

Create a string by concatenating the two operands

and the operator after them.

string = operand1 + operand2 + operator

And push the resultant string back to Stack

Prefix to Postfix

Read the Prefix expression in reverse order

(from right to left)

If the symbol is an operand,

then push it onto the Stack

If the symbol is an operator,

then pop two operands from the Stack

Create a string by concatenating the two operands

and the operator after them.

string = operand1 + operand2 + operator

And push the resultant string back to Stack

Repeat the above steps

until end of Prefix expression.

Evaluation of Prefix expression

Hint: Scan the expression from right to left

e.g., +*abc

Algorithm for Postfix to Prefix:

1. Scan the Postfix expression from left to right.
2. If the symbol is an operand, then push it onto the Stack
3. If the symbol is an operator, then
   a. pop two operands from the Stack in the following order:

      operand2 = Pop()

      operand1 = Pop()

   b. Create a string by concatenating the two operands and the operator before them.

      string = operator + operand1 + operand2

   c. push the resultant string back to Stack
4. Repeat the above steps until end of Postfix expression.
5. Pop the string representing the Prefix expression on stack and return.

Algorithm for Prefix to Postfix:

1. Scan the Prefix expression in reverse order (from right to left)
2. If the symbol is an operand, then push it onto the Stack
3. If the symbol is an operator, then
   a. pop two operands from the Stack in the following order:

      operand1 = Pop()

      operand2 = Pop()

   b. Create a string by concatenating the two operands and the operator after them.

      string = operand1 + operand2 + operator

   c. Push the resultant string back to Stack
4. Repeat the above steps until end of Prefix expression.
5. Pop the string representing the Postfix expression on stack and return.

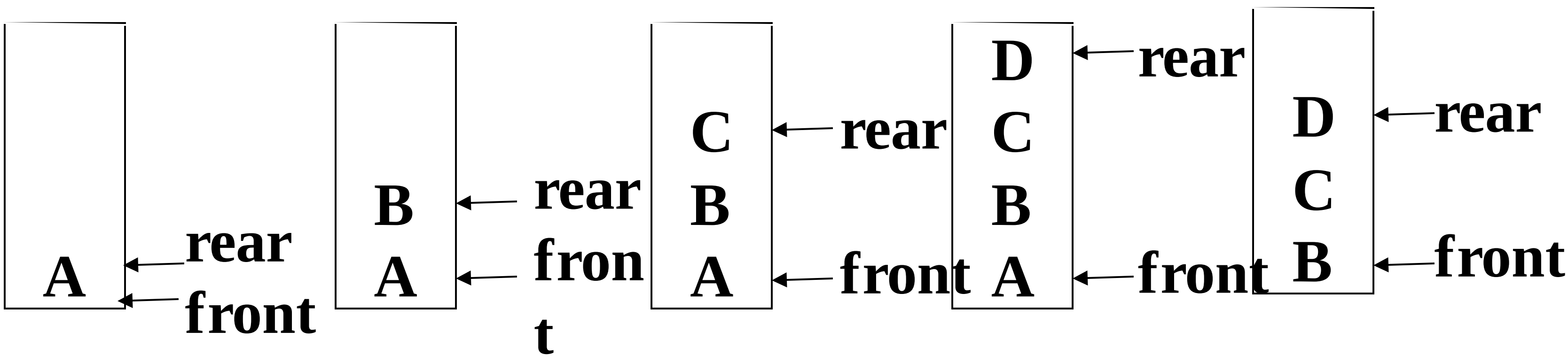
Queues

- Definition: A queue is an ordered list in which insertions and deletions takes place at different ends.

- **New elements are added to rear end**
- **Old elements are deleted from front end**

- **Since first element inserted is the first element deleted queues is also known as First-In-First-Out (FIFO) lists.**

Queue: **a First-In-First-Out (FIFO) list**



<u>*Figure 3.4: **Inserting and deleting elements in a queue (p.106**</u>

Queues - Application

- **Used by operating system (OS) to create job queues.**
- **If OS does not use priorities then the jobs are processed in the order they enter the system**

Application: **Job scheduling**

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
–1	–1					queue is empty
–1	0	J1				Job 1 is added
–1	1	J1	J2			Job 2 is added
–1	2	J1	J2	J3		Job 3 is added
0	2		J2	J3		Job 1 is deleted
1	2			J3		Job 2 is deleted

*Figure 3.5: **Insertion and deletion from a sequential queue (p.108)**

Abstract data type of queue

structure ***Queue*** **is**

objects: **a finite ordered list with zero or more elements.**

functions:

for all *queue* ∈ *Queue*, *item* ∈ *element*,

max_ queue_ size **∈ positive integer**

Queue **CreateQ(*max_queue_size*) ::=**

create an empty queue whose maximum size is *max_queue_size*

Implementation 1: **using array**

```
Queue CreateQ(max_queue_size) ::=
# define MAX_QUEUE_SIZE 100/* Maximum queue size */
typedef struct {
        int key;
        /* other fields */
        } element;

element queue[MAX_QUEUE_SIZE];
int rear = -1;
int front = -1;
```

Abstract data type of queue

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

***Queue* AddQ(*queue, item*) ::=**
if **(IsFullQ(*queue*)) *queue_full***
else **insert *item* at rear of *queue* and return *queue***

Implementation 1: **using array**

Boolean IsFullQ(queue) ::= rear == MAX_QUEUE_SIZE-1

Add to a queue

```
void addq(int *rear, element item)
{
/* add an item to the queue */
   if (*rear == MAX_QUEUE_SIZE-1) {
      queue_full( );
      return;
   }
  queue [++*rear] = item;
}
```

*Program 3.3: **Add to a queue (p.108)**

***Boolean* IsEmptyQ(*queue*) ::=**

if **(*queue* ==CreateQ(*max_queue_size*))**

return ***TRUE***

else return ***FALSE***

***Element* DeleteQ(*queue*) ::=**

if **(IsEmptyQ(*queue*))** return

else **remove and return the *item* at front of queue.**

*Structure 3.2: **Abstract data type *Queue (p.107)***

Implementation 1: **using array**

Boolean IsEmpty(queue) ::= front == rear

Delete from a queue

```
element deleteq(int *front, int rear)
{
/* remove element at the front of the queue */
   if ( *front == rear)
      return queue_empty( );    /* return an error key */
   return queue [++ *front];
}
```

*Program 3.4: **Delete from a queue(p.108)**

problem:

there may be available space when IsFullQ is true

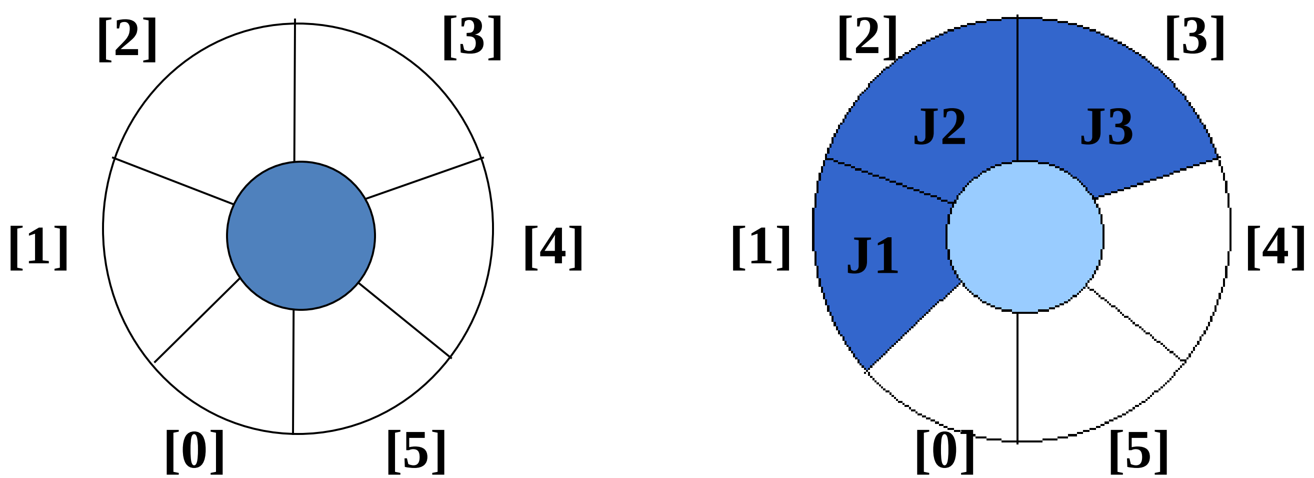
i.e.. movement is required.

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

*Figure 3.6: **Empty and nonempty circular queues (p.109)**

Problem: **one space is left when queue is full**

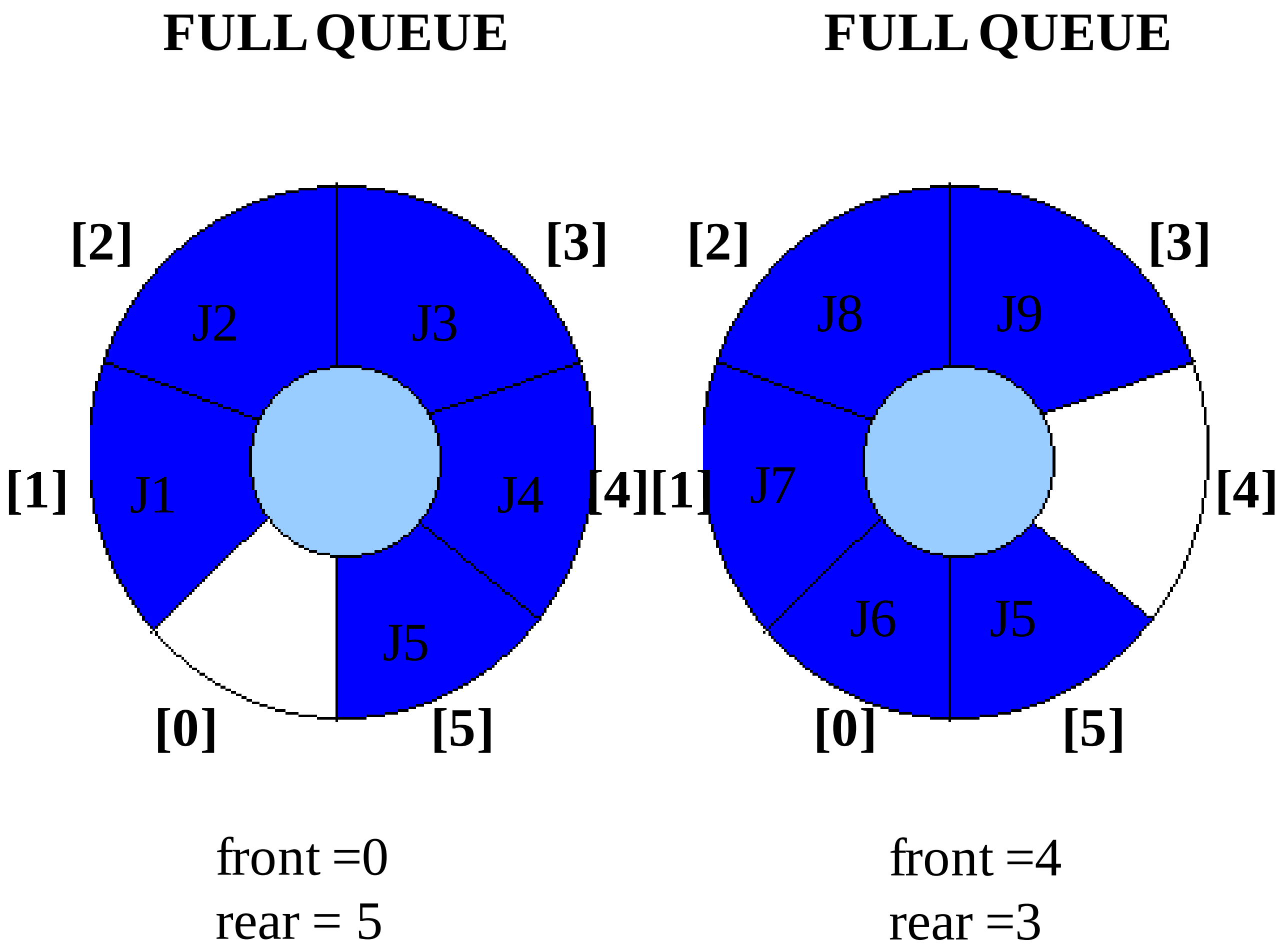


*Figure 3.7: **Full circular queues and then we remove the item (p.110)**

Add to a circular queue

```
void addq(int front, int *rear, element item)
{




}
```

*Program 3.5: **Add to a circular queue (p.110)**

Add to a circular queue

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

    queue[*rear] = item;
}
```

*Program 3.5: **Add to a circular queue (p.110)**

Delete from a circular queue

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

}
```

*Program 3.6: **Delete from a circular queue (p.111)**

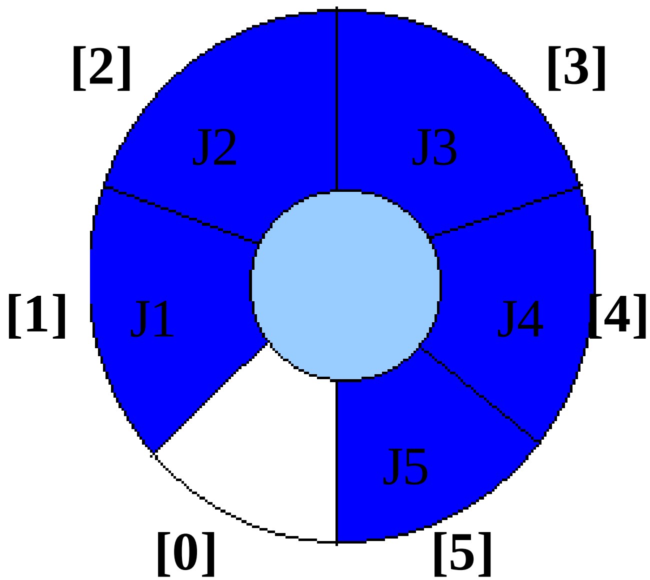
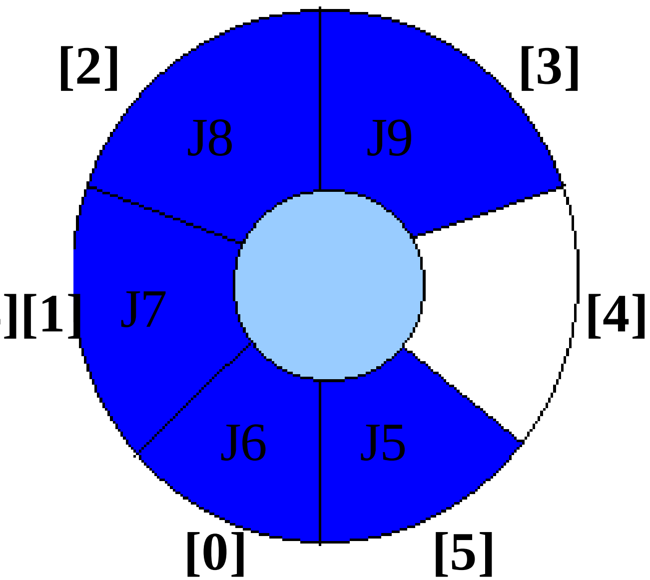
Delete from a circular queue

```
element deleteq(int* front, int rear)
{
  element item;
  /* remove front element from the queue and put it in item
*/
     if (*front == rear)
       return queue_empty( );
               /* queue_empty returns an error key */
    *front = (*front+1) % MAX_QUEUE_SIZE;

    return queue[*front];
}
```

*Program 3.6: **Delete from a circular queue (p.111)**

Circular Queue FULL: front = = rear condition

FULL QUEUE	FULL QUEUE
	
front =0 rear = 5	front =4 rear =3

Priority Queues

The priority queue is a data structure in which the intrinsic ordering of the elements does determine the results of its basic operations.
Two types of priority queues:

an ascending priority queue

a descending priority queue.

An ascending priority queue - collection of items into which items can be inserted arbitrarily and from which only the smallest item can be removed. If apq is an ascending priority queue. the operation pqinsert(apq,x) inserts element x into apq and pqmindelete(apq) removes the minimum element from apq and returns its value.

A descending priority queue - allows deletion of only the largest item. The operations applicable to a descending priority queue dpq, are pqinsert(dpq.x) and pqmaxdelete(dpq), pqinsert(dpq.x) inserts element x into dpq and is logically identical to pqinsert for an ascending priority queue. pqmaxdelete(dpq) removes the maximum element from dpq and returns its value.

The operation empty(pq) applies to both types of priority queue and determines whether a priority queue is empty.

Elements of a priority queue –

- **Need not be numbers or characters that can be compared directly.**
- **May be complex structures that are ordered on one or several fields.**
- **The *field on* which the elements of a priority queue is ordered need not be part of the elements themselves - it may be a special external value used specifically for the purpose of ordering the priority queue.**

- **Example:**
  - **Stack may be viewed as a descending priority queue whose elements are ordered by time of insertion.**
  - **Queue may be viewed as an ascending priority queue whose elements are ordered by time of insertion.**

Array Implementation of a Priority Queue

Suppose that the n elements of a priority queue pq are maintained in positions 0 to n-1 of an an-array *pq.items* of size maxpq and suppose that *pq.rear* equals to first empty array position, n.

pqinsert(pq,x)

```
if  (pq. rear ›= maxpq){
  prints ("priority queue overflow");
  exit(1);
) /* end if */
pq.items[pq.rear] = x;
pg.rear++;
```

Note: Under this insertion method the elements of the priority queue are not kept ordered in the array.

pqmindelete(pq) on an **ascending priority queue**

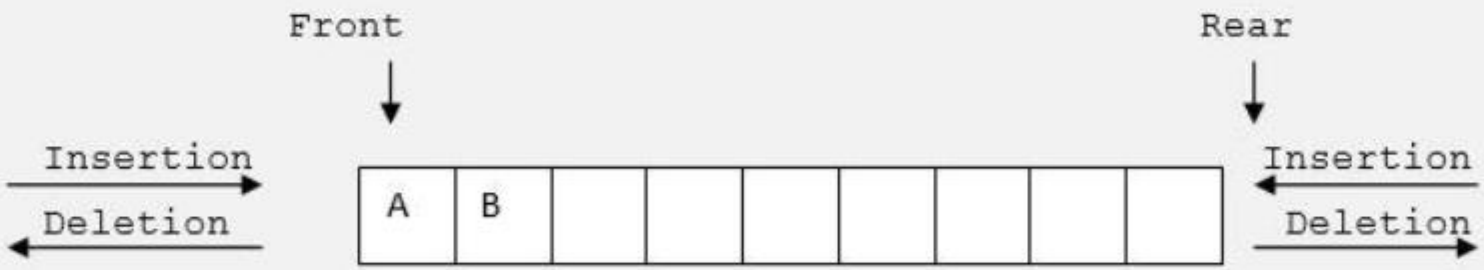
1. **Locate the smallest element - every element of the array from** *pq.items[0]* **through** ***pir.items[pq.rear-1] must*** **be examined. Requires accessing every element of the priority queue.**
2. **Deleting an element in the middle of the array. Solutions:**
   i. **Special "empty" indicator – an invalid value could be placed**

   ii. **Special "empty" indicator is used. New item is inserted in the first empty position**
   iii. **Each deletion, compact the array**
   iv. **Maintain the pq as an ordered circular array.**

Double Ended Queue (dequeuer)

Description:

A queue that supports insertion and deletion at both the front and rear is called **double-ended queue or Dequeue.** A Dequeue is a linear list in which elements can be added or removed at either end but not in the middle.



The operations that can be performed on Dequeue are

1. Insert to the beginning
2. Insert at the end
3. Delete from the beginning
4. Delete from end

```
#define MAX 30
typedef struct{
    int data[MAX];
    int front, rear;
}dequeue;
void initialise(deque*P){
    P->front=-1;
    P->rear=-1;
}
int empty(deque*P){
    if(P->rear==-1)
        return 1;
    return 0;
}
int full(deque*P){
    if((P->rear+1)%MAX==P->front)
        return 1;
    return 0;
}
```

```
void enqueueR(deque* P, int x){
    if (full(P)){
        // queue full
    }
    if (empty(P)){
        P->rear=0;
        P->front=0;
        P->data[0]=x;
    }
    else{
        P->rear = (P->rear +1)% MAX;
        P->data[P->rear]=x;
    }
}
```

```
void enqueF(deque* P, int x) {
    :
    :
    else {
        P->front = (P->front - 1 + MAX) % MAX;
        P->data[P->front] = x;
    }
}
```

```
int dequeF (deque* P){
    // check for empty queue
    int x;
    x = P->data[P->front];
    if (P->rear == P->front)
        initialise (P);
    else P->front = (P->fr +1)% MAX;
    return x;
}
```

```
int dequeR(deque *P){

- - x=P->data[P->rear];
- - //same as dequeF
  else
    P->rear=(P->rear-1+MAX)%MAX;
  return x;
}
```

Double Ended Queue (Deque)

There are:

Input-restricted Deque.

Output-restricted Deque.

Bellow show a figure a empty Deque Q[5] which can accommodate five elements.

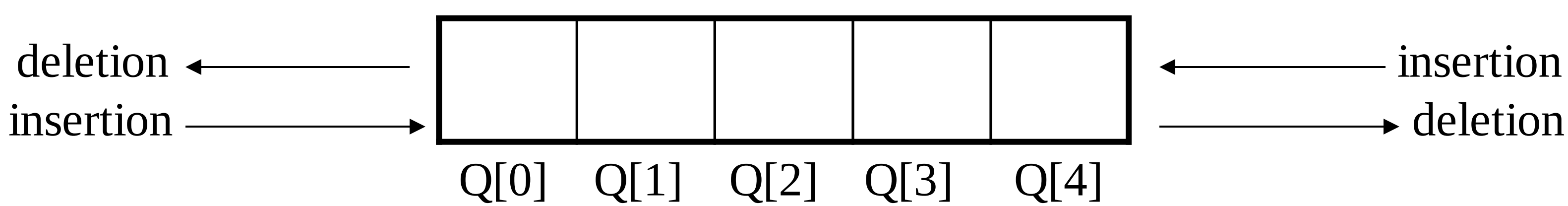


Fig: A Deque

Double Ended Queue (Deque)

- There are:
  - Input-restricted Deque: An input restricted Deque restricts the insertion of the elements at one end only, the deletion of elements can be done at both the end of a queue.

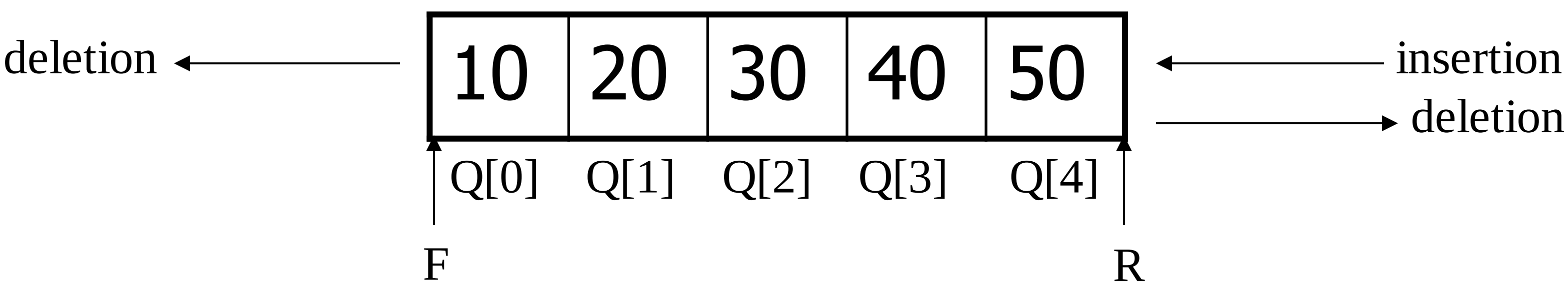


Fig: A representation of an input-restricted Deque

Double Ended Queue (Deque)

- There are:
  - Output-restricted Deque: on the contrary, an Output-restricted Deque, restricts the deletion of elements at one end only, and allows insertion to be done at both the ends of a Deque.

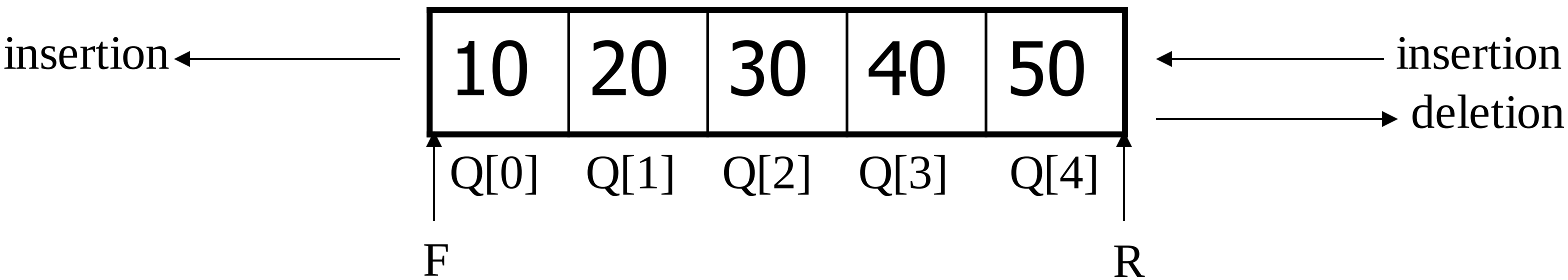


Fig: A representation of an Output-restricted Deque

Double Ended Queue (Deque)

- The programs for input-restricted Deque and output-restricted Deque would be similar to the previous program of Deque except for a small difference.
- The program for the input-restricted Deque would not contain the function addqatbeg().
- Similarly the program for the output-restricted Deque would not contain the function delatbeg().

Applications of Queues

Round robin technique for processor scheduling is implemented using queues.

All types of customer service (like railway ticket reservation) center software's are designed using queues to store customers information.

Printer server routines are designed using queues. A number of users share a printer using printer server the printer server then spools all the jobs from all the users, to the server's hard disk in a queue. From here jobs are printed one-by-one according to their number in the queue.