ADTS, STACKS AND ITS APPLICATIONS

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)!!!!***

Arrays vs. Lists

- List: **Ordered set consisting of variable number of objects.**
- ***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**

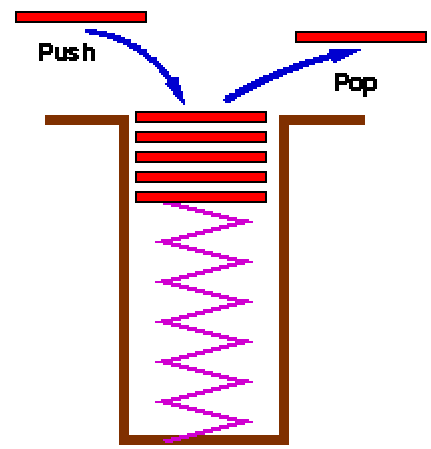
  Array Create(j, list) ::= return an array of j dimension where list is a j-tuple whose ith element is the size of ith dimension

  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

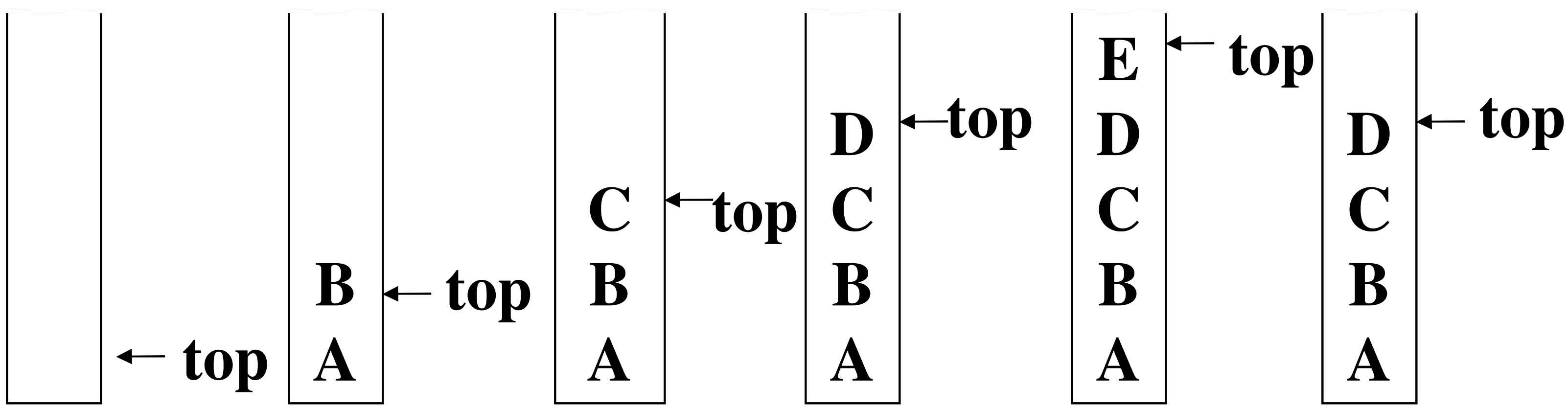


*Figure 3.1: **Inserting and deleting elements in a stack (p.102**

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(element item)
{
 /* add an item to the global stack */
    if (top >= MAX_STACK_SIZE-1)  {
         stack_full( );
return;
    }
    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

Use of Stacks in Function call – System Stack

- Whenever a function is invoked program creates a structure called activation record or a stack frame and places it on top of system stack.
- Initially, the activation record for the invoked functions contains only a pointer to the previous frame and return address.
- The previous stack frame pointer points to the stack frame of invoking function.

Use of Stacks in Function call – System Stack

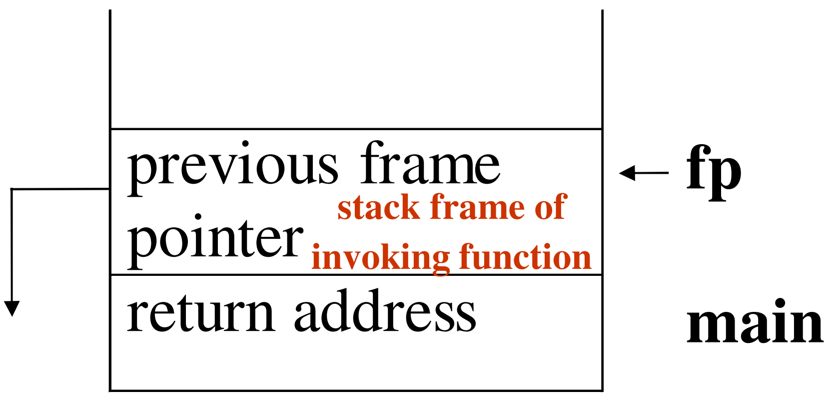
- While return address contains the location of the statement to be executed after the function terminates.
- Only one function executes at given time which is the top of stack.
- If this function invokes another, the non-static local variables, parameters of invoking function are added to stack frame.

Stacks in Function call – System Stack

- Assume that main() invokes function a1.
- It creates stack frame for a1.
- Frame pointer is a pointer to the current stack frame
- Also system maintains separately a stack pointer
- When a function terminates its stack frame is removed
- Processing of invoking which is on top of stack continues

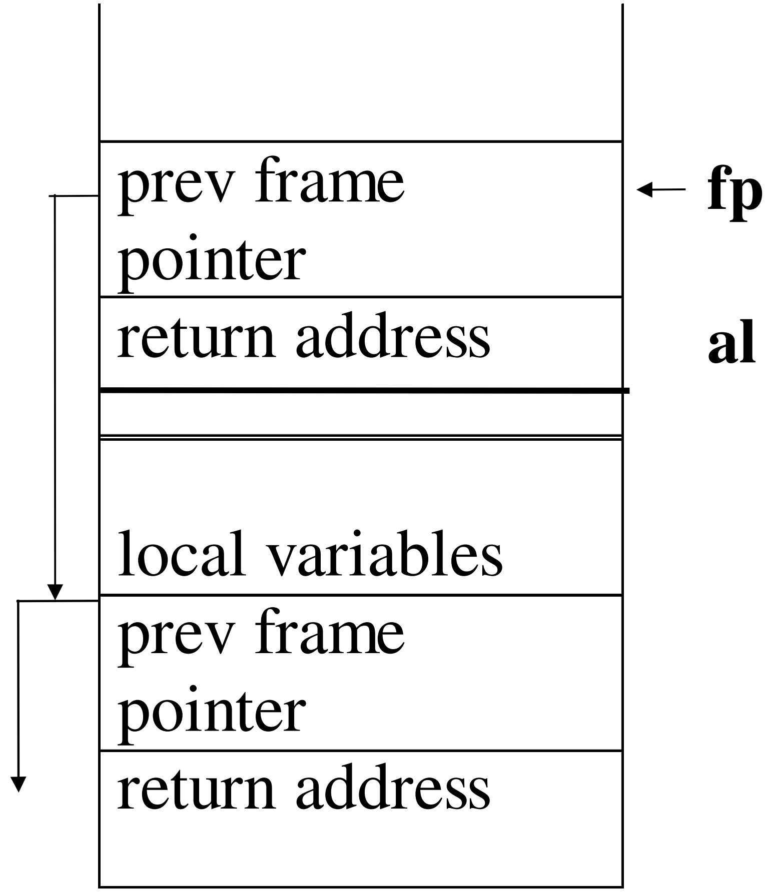
An application of stack: stack frame of function call (activation record)

fp: a pointer to current stack frame



system stack **before** a1 is invoked

(a)



(b)

system stack **after** a1 is invoked

System stack after function call a1

Other Applications:

Decimal to binary

Check for balanced parenthesis

Expression Conversions: Infix – Postfix, Infix – Prefix, etc

Evalation of Expressions: Postfix Evaluation, Prefix Evaluation

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

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

Postfix: **no parentheses, no precedence** 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* ab*c/ abc-d+/ea-*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*-

<u>**two passes**</u>

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	$abc+*_1d$
eos	$*_2$			0	$abc+*_1d*_2$

* 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

Infix and prefix expressions

Infix to Prefix : Using the algorithm for infix-to-postfix with few modifications (highlighted in red)

```
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 [ ] = {19, 0,12, 12, 13, 13, 13, 0};
static int icp [ ] = {19, 20, 12, 12, 13, 13, 13, 0};
```

isp: in-stack precedence
icp: incoming precedence

```
void infix_prefix(void)
{
/* output the postfix of the expression. The expression
   string, the stack, and top are global */
  char symbol;
  precedence token;
  char  prefix[100];
  int j = 0; //index for prefix
  int n = 0;
  int top = 0; /* place eos on stack */
  stack[0] = eos;
  strrev(expr); //reverse the infix expression
  for (token = get _token(&symbol, &n); token != eos;
                 token = get_token(&symbol, &n)) {
    if (token == operand)
      prefix[j++]= symbol;
    else if (token == lparen ){
```

```
    /*unstack tokens until right parenthesis */
    while (stack[top] != rparen)
       prefix[j++]= delete(&top);
    delete(&top); /*discard the right 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] )   //only > to achieve left
        prefix[j++]= delete(&top);                          //associativity
     add(&top, token);
   }
 }
//pop and add to expression until eos is reached
 while ((symbol = delete(&top)) != eos)
     prefix[j++]= symbol;
 print("\n");
 strrev(prefix); print(prefix);
}
```

Function to convert from infix to prefix

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.

CHAPTER 3

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.

Multiple stacks

Two stacks

m[0], m[1], …, m[n-2], m[n-1]

bottommost stack 1 → ← **bottommost stack 2**

Multiple stacks

m[0], m[1], …, m[n-2], m[n-1]

More than two stacks (n)

memory is divided into n equal segments

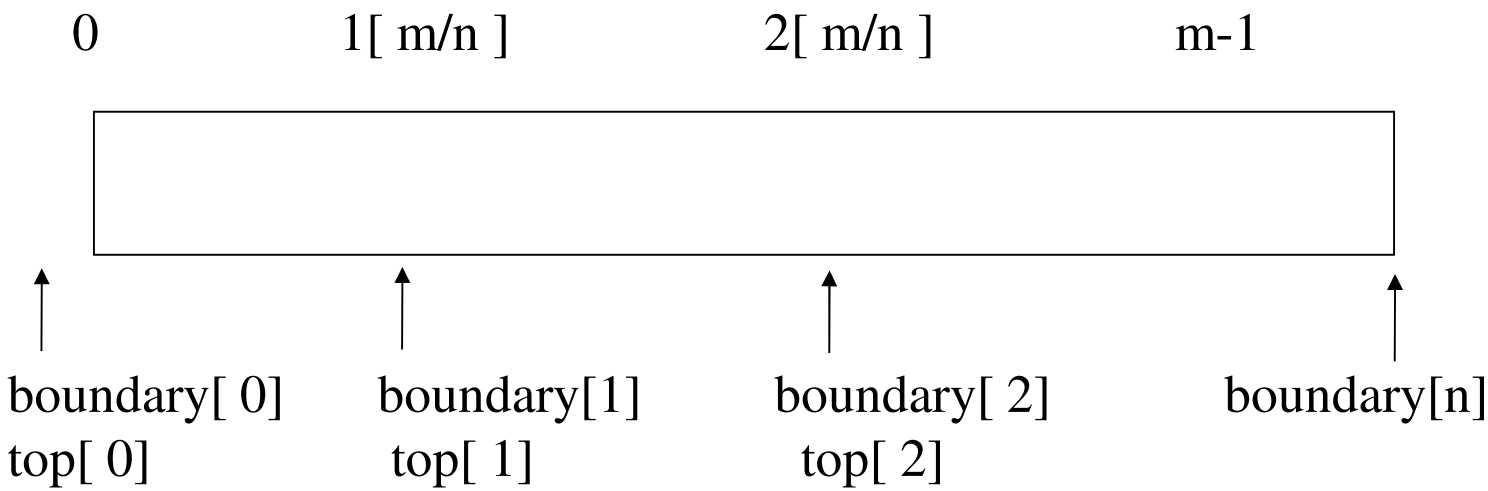
boundary[stack_no]

0 ≤ stack_no < MAX_STACKS

top[stack_no]

0 ≤ stack_no < MAX_STACKS

Initially, boundary[i]=top[i].



All stacks are empty and divided into roughly equal segments.

*Figure 3.18: **Initial configuration for *n* stacks in memory [m]. (p.129)***

```
#define MEMORY_SIZE 100    /* size of memory */
#define MAX_STACKS 10
            /* max number of stacks plus 1 */
/* global memory declaration */
element memory[MEMORY_SIZE];
int top[MAX_STACKS];
int boundary[MAX_STACKS];
int n; /* number of stacks entered by the user */
```

```
top[0] = boundary[0] = -1;
for (i = 1; i < n; i++)
    top[i] =boundary[i] =(MEMORY_SIZE/n)*i;
boundary[n] = MEMORY_SIZE-1;
```

```
void add(int i, element item)
{
   /* add an item to the ith stack */
   if (top[i] == boundary [i+1])
      stack_full(i);     may have unused storage
      memory[++top[i]] = item;
}
```

*Program 3.12:**Add an item to the stack *stack-no* (p.129)**

```
element delete(int i)
{
   /* remove top element from the ith stack */
   if (top[i] == boundary[i])
     return stack_empty(i);
   return memory[top[i]--];
}
```

*Program 3.13:**Delete an *item* from the stack *stack-no* (p.130)**

Find j, stack_no $<$ j $<$ n

such that top[j] $<$ boundary[j+1]

or, $0 \leq$ j $<$ stack_no

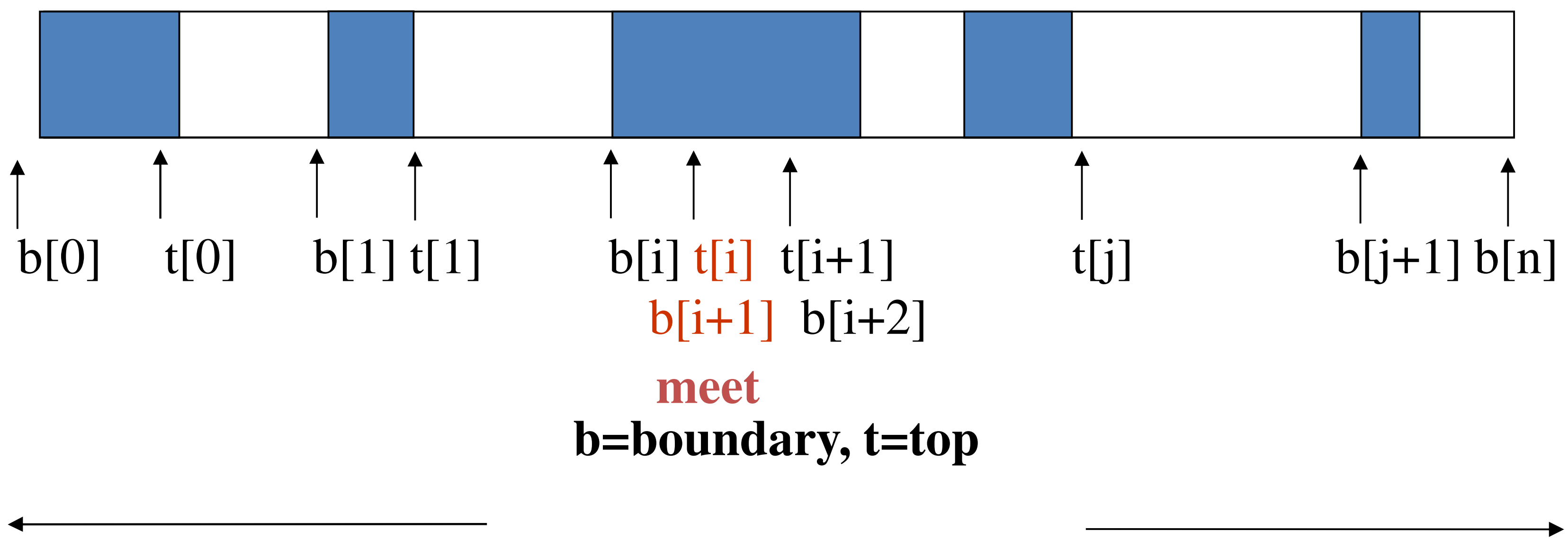


*Figure 3.19: **Configuration when stack i meets stack i+1, but the memory is not full (p.130)**

Stack using Dynamic Arrays

```
element *stack= (element *) malloc(sizeof(element));
int capacity=1;    //replace MAX_ITEM_SIZE

Void StackFull()
{
stack = (element *) realloc(2*capacity, sizeof(element));
capacity *=2;
}
```