Chapter 3 - CAI PACKAGES FOR SELECTED DATA STRUCTURES

I. Introduction.

In general, designing a program to solve a given problem needs careful decision making in deciding the following:

- How data elements are arranged in relation to each other?
- How to store data elements in memory?
- What operations can be performed on these data elements?

Thus we are concerned with the use and implementation of data objects which are part of Data Structures. Data Structures are mainly classified as primitive and non-primitive. Efficiency of a program is depending on the algorithm as well as the data structures. Thus in order to get good results, suitable data structure is to be selected. Knowledge of data structures is essential for designing and developing effective systems. However, some data structures are not linearly ordered. They need to be understood logically with good imaginative skills. We have attempted to present the intrinsic concepts of some of the data structures using graphics. We present here, the CAI packages that we are able to design and develop for improving the teaching-learning process of data structures.
II. CAI for understanding the data structure Stack.

Stack is a linear list structure, in which, insertions and deletions are performed at only one end called top of the stack [10]. Inserting an element into the stack is known as PUSH operation. Deleting an element from a stack is known as POP operation. In this package we demonstrate the process of PUSH and POP operations using animation and audio-visual effects. Main aim of this CAI package is to make the learner to understand the process of performing insertions and deletions from only one end. We allow the user to input a number that is intended to push into the stack. System presents the process of PUSHing this element into the stack using visual effects. Also changes to the stack pointer are shown through graphics so that it points to the top of the stack. On same lines the operation POP is also presented. Besides understanding these two operations, student can identify the OVERFLOW and UNDERFLOW conditions using this package. This package helps in effective understanding of stack operations and the stack data structure more effectively at the quickest possible time. We present the following preliminaries as part of the package in the help menu.

1. Definitions and Preliminaries.

1.1 Stack:

Stack is a linear structure where insertions and deletions are performed at only one end. This is also known as FIFO (FIRST IN First OUT) structure
because the element which is inserted first will be the element to come out last. One example may be the stack of plates. The end, where insertions and deletions are performed is known as top which is also known as stack pointer.

1.2 Stack Operations:
Two operations are possible on a stack. One operation is inserting an element into a stack which is known as PUSH. Another operation is deleting an element from a stack which is known as POP. These two operations affect the stack pointer top. Whenever an element is inserted, top is increased by one. Whenever an element is deleted from the stack, top is decreased by one. There are two exceptional cases one may encounter while performing push and pop operations on stack. They are OVERFLOW and UNDERFLOW. OVERFLOW occurs whenever top is equal to the size of the stack. UNDERFLOW occurs whenever top becomes zero.

1.3 Stack Applications:
Applications of stack include stack addressing, converting infix expression to reverse polish expression, evaluating reverse polish expression, recursion etc.

2. Software Development Methodology.
To make the learner comfortable in using this package, we have divided the package into three components namely help on stack, simulation of push operation and simulation of pop operation.
2.1 Help on Stack:
We have explained about stack through help window. Learner can go through the whole information using PgDn/Pgup keys.

2.2 Simulation of PUSH:
To simulate the PUSH operation, screen is divided into different number of windows as shown in Fig. 3.1. These are algorithm window, stack window, and instruction window. In algorithm window, the steps that are required to push an element into the stack are displayed. Instructions to the learner about the package are explained through instruction window. Stack window shows the stack and its contents. There is another small window which is present on the screen shows the value of stack pointer top.

2.3 Simulation of POP:
To simulate POP operation, screen is divided into different windows as explained in 2.2. Fig. 3.3 gives the sample screen for POP operation. Here algorithm window will be containing the instructions required to pop an element from the stack.

Learner is provided an option with a message to select any one of three choices namely PUSH, POP and Exit. If PUSH is selected, then system asks the learner to input a number to push into the stack. System accepts the number supplied by the learner and shows the process of PUSH operation by moving an arrow head which points to the steps displayed in algorithm window (Fig. 3.1). Through
this process, learner can visualize the process of pushing number into the stack, change in the value of stack pointer (i.e. top) and the contents of stack. Sound effects are also provided to make the learner attentive. Learner can repeat this process any number of times. After some time OVERFLOW message will be displayed when top becomes equal to the size of stack (Fig. 3.2). If the option selected is POP, process of deleting an element (i.e. POP) is shown on the same lines of PUSH. Only difference is whenever an element is deleted, top is decremented by 1 and UNDERFLOW message will be displayed when top becomes zero (Fig. 3.3 & Fig. 3.4).

4. Advantages.
Learner will be clear about stack structure and stack operations after using this package, because the whole process is visualized. Understanding OVERFLOW and UNDERFLOW becomes concepts become easy once the stack structure and its contents are visualized.

5. Limitations.
5.1 Learner is allowed to use only two digit numbers.
5.2 Stack size is limited to 9 elements due to the screen height that is available.
### Stack Operations

**Fig. 3.1 Process of pushing 42.**

<table>
<thead>
<tr>
<th>STACK</th>
<th>TOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>40</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Enter a number to PUSH: 42

1. If Stack is full then Stack Overflow
2. Top <- Top + 1
3. Assign the element pointed by Top
4. Terminate the algorithm.
## Stack Operations

<table>
<thead>
<tr>
<th>STACK</th>
<th>TOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>54</td>
</tr>
<tr>
<td>4</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>42</td>
</tr>
<tr>
<td>7</td>
<td>100</td>
</tr>
<tr>
<td>8</td>
<td>67</td>
</tr>
<tr>
<td>9</td>
<td>80</td>
</tr>
</tbody>
</table>

Stack Overflow

Press any key to continue.

1. If Stack is full then Stack Overflow
2. Top <- Top + 1
3. Assign the element pointed by Top
4. Terminate the algorithm.
**Stack Operations**

<table>
<thead>
<tr>
<th>Step 1. If Stack is empty then Stack Underflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 2. Remove top most element</td>
</tr>
<tr>
<td>Step 3. Top ( \leftarrow ) Top - 1</td>
</tr>
<tr>
<td>Step 4. Terminate the algorithm.</td>
</tr>
</tbody>
</table>

**Fig. 3.3. Pop operation.**

<table>
<thead>
<tr>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element just popped!</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Fig. 3.4. Stack Underflow.

<table>
<thead>
<tr>
<th>Stack Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Stack Underflow</strong></td>
</tr>
<tr>
<td><strong>Step 1.</strong> If Stack is empty then Stack Underflow</td>
</tr>
<tr>
<td><strong>Step 2.</strong> Remove top most element</td>
</tr>
<tr>
<td><strong>Step 3.</strong> Top &lt;- Top - 1</td>
</tr>
<tr>
<td><strong>Step 4.</strong> Terminate the algorithm</td>
</tr>
</tbody>
</table>

Stack Underflow Press any key to continue.
III. Conversion of Infix Expression to Reverse Polish - A Graphical Presentation.

In this package, we developed an algorithm for accomplishing the following:

a. To accept and display the given infix expression on the screen.

b. Display the algorithm which is used to convert a given Infix notation to Reverse Polish notation in a window.

c. Demonstrate through audio-visual effects and animation the step by step process which converts the given Infix expression to Reverse Polish expression.

This graphical presentation allows to enter any Infix Expression and animates each step on formation of corresponding Reverse Polish expression. In that process, the stack operations are also simulated on the screen.

1. Polish and Reverse Polish notations.

For our most common arithmetic operations, the operator is placed between two operands.

Example : A+B

This is referred as infix notation.

Polish Notation named after the Polish Mathematician Jan Lukasiewicz, refers to the notation in which the operator symbol is placed before two operands.

Example : +AB

This is to add the content of A and the content of B.

Reverse Polish notation refers to the analogous notation in which the operator symbol is placed after its two operands.
Example: AB+
The importance of this postfix notation is that we never need parentheses to determine the order of the operations. Hence it is very popular in semantics. This reverse polish notation is also called as postfix or suffix notation or RP expression.

In this package, we denote the given infix expression by IN and the postfix expression (Reverse Polish) by PN. We refer to the operands, operators and the parentheses as elements.

For converting the infix expression to reverse polish expression, we develop the following algorithm and use it for software implementation on presenting the process through animation using graphics.

2.1 Accept the given infix expression.
2.2 Validate on the specified rules in forming the infix expression. If invalid display the suitable message and branch to step 2.1.
2.3 Display the given infix expression and designate it as IN on the screen. Reserve space for PN on the screen to keep Reverse Polish expression.
2.4 Display the stack structure on the screen. Form another window and display the important steps(algorithm) for converting the given infix to postfix expression.
2.5 PUSH left parenthesis to stack and join right parenthesis at the end of the expression displayed at IN. These two are used as sentinels.
2.6 Display pointers at the suitable step in the algorithm window and PN.

2.7 Show the animation on scanning the expression at IN from left to right in repeating the following steps until the stack is empty.

2.7.1 If an operand is found, join it to PN.

2.7.2 If left parenthesis is encountered, PUSH it on to stack.

2.7.3 If an operator (say OP) is encountered then

2.7.3.1 Repeatedly POP each operator from the stack and join it to PN while it has same or higher precedence as OP.

2.7.3.2 Push OP to stack.

2.7.4 If right parenthesis is encountered then

2.7.4.1 Repeatedly POP each operator from the stack and join it to PN until left parenthesis is encountered.

2.7.4.2 Remove the left parenthesis from the stack.

2.8 Exit.

3. Example of an Infix Expression and the steps in converting the same to Postfix expression using the algorithm in 2.

We join right parenthesis to the given Infix expression and present as follows.

IN : A+B-C+D*Q^-(R-S)-T/(E-F/G))

We tabulate the elements of STACK in the following table and PN at each step of scanning an element of the above Infix notation.

31
<table>
<thead>
<tr>
<th>Element scanned</th>
<th>Elements in Stack</th>
<th>Postfix notation (PN)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nil</td>
<td>(</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>(</td>
<td>A</td>
</tr>
<tr>
<td>+</td>
<td>(,+</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>(,+</td>
<td>AB</td>
</tr>
<tr>
<td>-</td>
<td>(,-</td>
<td>AB+</td>
</tr>
<tr>
<td>C</td>
<td>(,-</td>
<td>AB+C</td>
</tr>
<tr>
<td>+</td>
<td>(,+</td>
<td>AB+C-</td>
</tr>
<tr>
<td>D</td>
<td>(,+</td>
<td>AB+C-D</td>
</tr>
<tr>
<td>*</td>
<td>(,+,*</td>
<td>AB+C-D</td>
</tr>
<tr>
<td>Q</td>
<td>(,+,*</td>
<td>AB+C-DQ</td>
</tr>
<tr>
<td>&quot;</td>
<td>(,+,*,&quot;</td>
<td>AB+C-DQ</td>
</tr>
<tr>
<td>(</td>
<td>(,+,*,&quot;,(</td>
<td>AB+C-DQ</td>
</tr>
<tr>
<td>R</td>
<td>(,+,*,&quot;,(</td>
<td>AB+C-DQR</td>
</tr>
<tr>
<td>-</td>
<td>(,+,*,&quot;,(,-</td>
<td>AB+C-DQR</td>
</tr>
<tr>
<td>S</td>
<td>(,+,*,&quot;,(,-</td>
<td>AB+C-DQRS</td>
</tr>
<tr>
<td>)</td>
<td>(,+,*,&quot;</td>
<td>AB+C-DQRS-</td>
</tr>
<tr>
<td>-</td>
<td>(,+,*,&quot;,-</td>
<td>AB+C-DQRS--**+</td>
</tr>
<tr>
<td>T</td>
<td>(,-</td>
<td>AB+C-DQRS--**+T</td>
</tr>
<tr>
<td>/</td>
<td>(,-,/</td>
<td>AB+C-DQRS--**+T</td>
</tr>
<tr>
<td>(</td>
<td>(,-,/,(</td>
<td>AB+C-DQRS--**+T</td>
</tr>
<tr>
<td>E</td>
<td>(,-,/,(</td>
<td>AB+C-DQRS--**+TE</td>
</tr>
<tr>
<td>-</td>
<td>(,-,/,(,-</td>
<td>AB+C-DQRS--**+TE</td>
</tr>
<tr>
<td>F</td>
<td>(,-,/,(,-</td>
<td>AB+C-DQRS--**+TEF</td>
</tr>
<tr>
<td>/</td>
<td>(,-,/,(,-,/</td>
<td>AB+C-DQRS--**+TEF</td>
</tr>
<tr>
<td>G</td>
<td>(,+,/,(,-,/</td>
<td>AB+C-DQRS--**+TEFG</td>
</tr>
<tr>
<td>)</td>
<td>(,-,/</td>
<td>AB+C-DQRS--**+TEFG/</td>
</tr>
<tr>
<td>)</td>
<td>Nil</td>
<td>AB+C-DQRS--**+TEFG/-</td>
</tr>
</tbody>
</table>
4. Advantages.
Algorithm to convert a given infix expression to its equivalent postfix expression is very difficult to understand in one go and it is tedious to repeat the algorithm with many examples. This package provides good understanding about the whole process by showing each step. Also helps in repeating the same with different types of examples. Here we provide a facility to validate the infix expressions and learner will be knowing reasons for the invalidity of the infix expression.

5. Assumptions and Limitations.
5.1 We assume the elements of the given infix expression as follows.
Operands are limited to only one character. This character can be any alphabet or numeric digit. However, letters may be represented in either capital or small letters.
5.2 Operators are restricted to the following:
+ for addition
- for subtraction.
* for multiplication
/ for division
~ for exponentiation
5.3 The hierarchical levels of operators used in the given infix expression are assumed to be as follows.
Level 1 : Parenthesis - Highest priority.
Level 2 : Exponentiation.
Level 3 : Multiplication and Division.
Level 4: Addition and Subtraction - Lowest priority.

5.4 Maximum number of elements in the infix expression are limited to 30 elements for the reasons of accommodating on the limited screen.

5.5 Positions in the stack are limited to 10 for the reasons of adjusting on the limited screen.

5.6 Rules on validating the given Infix expression:
   5.6.1 The number of left parentheses must be equal to the number of right parentheses.
   5.6.2 The number of elements in the infix expression must be odd.
   5.6.3 No operand must precede a left parenthesis or a right parenthesis.
   5.6.4 No two operators or no two operands occur simultaneously.

5.7 We have implemented the complete process using Pascal. The screen layout at one of the stages of the process is enclosed (Fig.3.5 & Fig.3.6) herewith. While using animation for describing various stages of the processes in the package, we have used several techniques using graphics. These points are not explained here.
IN: \( A \div B - C + D \times Q \wedge ( R - S ) \)
PN: \( A B + C - D Q \)

1. PUSH "(" to STACK and join ")" to the end of IN.

2. Scan IN from left to right in repeating 2.1 to 2.4 until STACK is empty
   2.1 If an operand found, join it to PN
   2.2 If ")" found PUSH it on to STACK
   2.3 If an OP is encountered then ..
       2.3.1 Repeatedly POP OP from the STACK and join it to PN while it has SAME or HIGHER precedence than OP.
       2.3.2 PUSH OP to STACK
   2.4 IF ")" is encountered then ..
       2.4.1 Repeatedly POP OP from STACK and join it to PN until "(" is found.
       2.4.2 Remove the "(".

3. EXIT.
IN: A + B - C + D * Q ^ ( R - S )

PN: A B + C - D Q R S - ^ x +

1. PUSH "(" to STACK and join ")" to the end of IN.

2. Scan IN from left to right in repeating 2.1 to 2.4
   until STACK is empty
   2.1 If an operand found, join it to PN
   2.2 If "(" found PUSH it on to STACK
   2.3 If an OP is encountered then ..
      2.3.1 Repeatedly POP OP from the STACK and
           join it to PN while it has SAME or
           HIGHER precedence than OP.
      2.3.2 PUSH OP to STACK
   2.4 If ")" is encountered then ..
      2.4.1 Repeatedly POP OP from STACK and
           join it to PN until "(" is found.
      2.4.2 Remove the "(".

3. EXIT.
IV. Evaluation of Postfix Expression - A Simulation.

The process of evaluating a given postfix expression uses a well defined algorithm and a stack. However this process is logically presented during teaching process in class rooms and it is not visible to naked eye to students. In this package we have simulated the process of evaluation of postfix expression on a graphics screen. We present the postfix expression, algorithm and stack using graphics and provide audio & visual signals on each step of the process. This software also provides interactions on entering and editing the postfix expression.

1. Definitions and Preliminaries.

1.1 Algorithm for evaluating the given RP expression:

1.1.1 Put the character # at the end of the expression. This symbol would be used as sentinel.

1.1.2 Scan the expression on the following lines while each element is not equal to the symbol #.

1.1.2.1 If an element is found to be an operand, PUSH it to stack.

1.1.2.2 If an element is found to be an operator then
    a. POP two elements from the stack.
    b. Do arithmetic on these two elements using the operator and get the result.
    c. PUSH the result to the stack.

1.1.3 POP the element from the stack and output that as the value of the expression.

1.1.4 Exit
1.2 Example of RP expression and the steps in evaluating the same using the algorithm given in 1.1.

RP expression : 341+*5-26+/#

<table>
<thead>
<tr>
<th>Element Scanned</th>
<th>Elements in Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3, 4</td>
</tr>
<tr>
<td>1</td>
<td>3, 4, 1</td>
</tr>
<tr>
<td>+</td>
<td>3, 5</td>
</tr>
<tr>
<td>*</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>15, 5</td>
</tr>
<tr>
<td>-</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>10, 2</td>
</tr>
<tr>
<td>6</td>
<td>10, 2, 6</td>
</tr>
<tr>
<td>+</td>
<td>10, 8</td>
</tr>
<tr>
<td>/</td>
<td>1.25</td>
</tr>
<tr>
<td>#</td>
<td></td>
</tr>
</tbody>
</table>

On sensing the end character namely # in this case, we pop the element from the stack and output that element as the result. The stack is found to be empty. The above RP expression is the following infix notation that we are usually familiar with.

\[
(3*(4+1)-5)/(2+6)
\]

2. Software Implementation.

In this package, we present the process involved in evaluating any given RP expression (on the lines of the algorithm given in 1.1) with audio-visual effects. The software developed for evaluating the RP expression simulates stack and the evaluation process on graphics screen.

User is prompted to enter the RP expression. After the successful entry of expression, we display the graphics screen with partitions, showing the following:
a. RP expression
b. Graphical representation of stack and stack pointer
c. List of steps in the algorithm given in 1.1 for evaluating the RP expression.
d. Calculation area.

We start evaluation process by indicating with a pointer (an arrow head) to the first step of the algorithm. Suitable sounds are produced to attract the attention to that step. Simultaneously, the operand that is considered in the RP expression is pointed out by a separate pointer (an arrowhead). The process of picking up the operand and pushing the same to the stack is clearly shown using animation. The whole evaluation process with animated pointers would be displayed on each step of the algorithm, on similar lines. For example, it will be pleasure to see the popping operations in animation through which two elements of the stack get popped out to the calculation area. The result thus obtained is pushed to stack using animation. Figures 3.7, 3.8 and 3.9 are graphical screen printouts that are taken at one of the stages of the evaluation process.

3. Limitations and Assumptions.

3.1 An expression in reverse polish notation contains operands in the form of numbers and operators in the form of arithmetic operators.

3.1.1 Length of each number (operand) is restricted to maximum 1 digit (any digit from 0 to 9).
## Evaluation of Postfix Notation

### Algorithm Window

1. **Put # at the end of expression**
2. **Scan the expression on the following lines while each element () #**
   - 2.1 If an element is found to be an operand, push it to stack
   - 2.2 If an element is found to be an operator (say @) then
     - 2.2.1 POP two elements of stack (say A and B)
     - 2.2.2 Use the operator on both elements (evaluate A @ B)
     - 2.2.3 PUSH the result to stack
3. **Output the value of the expression by popping the top element of stack**
4. **Exit**

### STACK

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### CALC. AREA

Enter Postfix notation : \[ 32 \times 76 \times - \]
### Evaluation of Postfix Notation

<table>
<thead>
<tr>
<th>STACK</th>
<th>CALC. AREA</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>+</td>
</tr>
<tr>
<td>2</td>
<td>----------</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Enter Postfix notation : 32+76*-#

1. Put # at the end of expression
2. Scan the expression on the following lines while each element <> #
   2.1 If an element is found to be an operand, push it to stack
   2.2 If an element is found to be an operator (say @) then
       2.2.1 POP two elements of stack (say A and B)
       2.2.2 Use the operator on both elements (evaluate A @ B)
       2.2.3 PUSH the result to stack
3. Output the value of the expression by popping the top element of stack
4. Exit
### Evaluation of Postfix Notation

Enter Postfix notation: \(32+76*-#\)

1. Put \# at the end of expression
2. Scan the expression on the following lines while each element \('<>'\) #
   2.1 If an element is found to be an operand, push it to stack
   2.2 If an element is found to be an operator (say @) then
      2.2.1 POP two elements of stack (say A and B)
      2.2.2 Use the operator on both elements (evaluate A @ B)
      2.2.3 PUSH the result to stack
3. Output the value of the expression by popping the top element of stack
4. Exit
3.1.2 Each of the operator may be one of the following arithmetic operators:

+ for Addition - for Subtraction
* for multiplication / for division

3.2 Total number of elements in the form of operand and operators in RP expression should not be more than eleven. The restriction is made because of the limited space is available on the screen for keeping RP expression.

3.3 Stack is limited to keep 8 operands.

4. Control Checks.

Following checks are made on Reverse Polish expression to ensure that it is a valid RP expression.

4.1 Operands and Operators are checked for their validity.

4.2 A check is made to ensure that the total number of elements in an RP expression is odd in number.

4.3 No operator should be present in the first two elements.

4.4 Before demonstrating graphics presentation, we evaluate internally the RP expression on the lines of algorithm 1.1 by scanning from first element to the last element. While evaluating, we ensure that an occurrence of an operator is supplemented by the presence of at least two operands for evaluation. At the end of the evaluation, it is expected to have only one number which is the result of the RP expression.

On any of the mismatches we give the facility to the user for re-entering the expression.
V. CAI for understanding IEEE representation of Floating point numbers.

1. Introduction.
One of the primitive data structures provided by any programming language is floating point number (or real number). There are various methods available to store floating point numbers in the memory. Among these methods IEEE (Institute of Electrical and Electronics Engineers) single format and double format representations are popular and are used by designers of different languages. This CAI package is to teach the following, using animation and providing audio-visual effects.
- Conversion of floating point number into its equivalent IEEE single or double format.
- Converting a given IEEE single or double format to its equivalent floating point number.
We made an attempt in this package to make the learner to understand excess-n notation, hidden bit principle, normalization of a binary number. This package allows the user to use input of a variety of choices.

2. Preliminaries.
2.1 Floating Point representation:
To perform arithmetic operations involving very large and very small real numbers, a convenient notation containing mantissa and exponent to represent a real number is being used[58]. In this notation, real number $24.56$ is expressed as $0.2456 \times 10^2$, where $0.2456$ is
mantissa and 2 is the exponent. This notation is known as floating point notation and is based on the relation $P^z = a \times r$, where $z$ is the real number, $a$ is mantissa, $p$ is exponent and $r$ is the base of the number system. For example, 10 is the base for decimal number system and 2 is the base for the binary number system. Expressing real numbers in this format allows us to separate range from precision. The range is determined by the number of digits in the exponent and precision is determined by the number of digits in the mantissa (or fractional part). Floating point numbers are represented in computers memory in binary format. Both mantissa and exponent are stored in binary format. Several different formats are available to represent floating point numbers in binary format. IEEE floating point formats are popular floating point representations and are being used by many designers of different languages. The popular IEEE floating point formats include single format and double format. Single format needs 4 bytes of memory and double format needs 8 bytes of memory. Double format provides more precision over single format. IEEE formats contain three components namely a sign bit, fractional part (or mantissa) and an exponent as shown below.

<table>
<thead>
<tr>
<th>Sign</th>
<th>Exponent</th>
<th>Fractional Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>8 bits</td>
<td>23 bits</td>
</tr>
</tbody>
</table>

IEEE single format

45
2.1.1 Sign Bit:
Sign bit is used to represent both positive and negative floating point numbers. Zero in sign bit represents a positive number and one in sign bit represents a negative number.

2.1.2 Fractional part / Mantissa:
In IEEE single format, fractional part occupy 23 bits and 52 bits in the double format. Fractional part in IEEE formats is stored in normalized form. The process of moving the binary point to a position just to the left of the most significant nonzero binary digit (i.e. binary digit 1) is called normalization. This is carried out to increase the number of trailing digits in the fractional part which ensures good precision[36]. After normalization, fractional part in binary format (F) lies in the range 0.5 <= F < 1. This means, normalization ensures that always a binary digit 1 is present immediately after the binary point. IEEE formats follow hidden bit principle for storing fractional part.

Hidden Bit principle:
There will be a binary digit 1 always after decimal point in the normalized form of fractional part. Since there is always 1, it can be assumed without storing[36].
This extra bit is referred as hidden bit. This hidden
bit will be taken into consideration whenever 
computations are carried out.

2.1.3 Exponent :

To store exponent, 8 bits are used in IEEE single format 
and 11 bits are used in IEEE double format. Exponent is 
an integer which can be either positive or negative.

To store integers sign and magnitude, 1's complement, 
and 2's complement techniques are available. In IEEE 
formats to store exponent none of these methods are 
used. A new method called excess-n notation is used to 
store the exponent. In excess-n notation exponent is 
stored after adding a constant which is called as bias 
or excess factor. In excess-127 notation, exponent 3 is 
stored as 130. If exponent is -25 it is stored as 102 
because -25+127=102. Excess factor is depending on the 
number of bits that are used to store the exponent. 
If 8 bits are used to store the exponent the range 
of positive numbers that can be stored are 0 to 255. By taking excess factor as 127 this range can be 
converted to -127 to 128. This means, using excess -127 
notation, integers in the range -127 to 128 can be 
represented. Following one-to-one map is used by 
excess-127 notation to store the exponent.

<table>
<thead>
<tr>
<th>Actual Exponent</th>
<th>Stored Exponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>-127</td>
<td>0</td>
</tr>
<tr>
<td>-126</td>
<td>1</td>
</tr>
<tr>
<td>-127</td>
<td>0</td>
</tr>
</tbody>
</table>
|    --> (Actual Exponent+127) --> |    .....
| 127             | 254             |
| 128             | 255             |


2.2 Steps required to convert a given floating point number to its equivalent IEEE single format.

Step 1: Convert the given real number (or floating point number) to its equivalent binary number.

Step 2: Normalize the binary number. After normalization put the binary number in 1.f format. At this stage the given floating point number will be in the format 1.f*2^e, where e is the exponent and f is the fractional part.

Step 3: Find the exponent (E) that is to be stored. i.e. E=e+127. Express E in binary format.

Step 4: Find sign bit. Sign bit is 0 if the given real number is positive. Sign bit is 1 if the given real number is negative.

Step 5: Put sign bit, E in binary form and f together. This gives IEEE binary format.

Step 6: Find equivalent IEEE hex format.

Example: Consider real number 12.75.

12.75 = 1100.11 in binary format.

\[1100.11 = .110011\times2^3\text{ after normalization.}
\]

\[= 1.10011\times2^3\]

Here e=3 and f=10011 as the bit 1 is hidden.

E=e+127=130=10000010

Sign bit is 0 since the number is positive

IEEE single in binary format is

01000001010011000000000000000000

IEEE single in hex format is 414C.
2.3 Steps required to convert a given IEEE single format to its equivalent real number:

Step 1: Convert the given IEEE single format in hex to its equivalent binary.

Step 2: Find sign bit, exponent E and the fractional part.

Step 3: Use the formula \((-1)^s \times 2^{E-127} \times 1.f\) to get the equivalent real number.

Example: Consider the IEEE single format in hex format 414C.

Equivalent binary format is

\[01000001010011000000000000000000\]

Sign bit is 0. E is 10000010 which is equal to 130 and f is 100110......0.

Real number is equal to

\[0 \times 130-127 \times 1.100110=12.75.\]

2.4 Special cases in IEEE format:

IEEE format to represent zero becomes a special case because in this case fractional part itself is zero. It is not possible to normalize the fractional part. So zero is represented as 00000000 (positive zero) and 80000000 (negative zero). Infinity is represented by all ones in E section and all zeros in F section. Again there are two representations for infinity namely positive infinity and negative infinity. Results of invalid operations are signalled by making E all zeros, S can be anything and F as nonzero.

To make learner to understand more about IEEE floating point representations, the package is divided into the following components and learner is allowed to select any one of these components at a given time.

- Help on IEEE formats.
- Conversion of real number to IEEE format.
- Conversion of IEEE format to real number.

3.1 Help on IEEE formats:

All the above discussed details are provided in help that is included in the package on F1 key. User is allowed to invoke this help facility and can go through with the help of PgDn/PgUp keys. Help on different steps also provided while the conversion is being carried out. This provides more understanding about each step in the conversion process.

3.2 Conversion of real number to IEEE format:

To show this process, screen is divided into mainly two windows. In one window which we call as WINDOW FOR INTERACTIONS, stepwise instructions are provided to the user. Another window shows different values in the conversion process.

Initially when this option is selected, learner is asked to input a non-zero real number. Validation is being done on this number. Since zero comes under special case, zero is avoided at this juncture. The accepted real number is displayed on the screen with a message 'Given real number:'. After this step, user has to press a key to continue.
This is to provide the user with sufficient time to understand the execution of a particular step. After each step user has to press a key to execute another step. Next step is converting the given real number to its equivalent binary number. Equivalent binary number is displayed with a message without showing the conversion process. After conversion, the resultant binary will be normalized. Normalization is shown slowly with audio-visual and sound effects. Here shifting of decimal point either to the left side or to the right side is shown and accordingly the exponent is either increased or decreased. After normalization, number is changed to 1.0 format by decrementing the exponent which is also shown very effectively. Once number is converted into 1.0 format, finding sign bit, exponent to be stored, IEEE binary format and finding IEEE hex formats are shown again with audio-visual and sound effects. Learner is allowed to repeat the whole exercise with many examples. Figures 3.10 and 3.11 gives sample screen layouts taken during the conversion process.

3.3 Conversion of IEEE format to a real number:
Upon selecting the second option, the whole conversion process is shown through different windows on the screen. Learner is asked to supply a IEEE hex format for the conversion. Here IEEE hex format is validated to ensure that it will not come under a special case. After accepting the IEEE hex format, it is converted into binary format and shown to the user. From this binary format sign bit,
exponent and fractional part are extracted and shown to the user with audio-visual and sound effects. Real number is found using the formula given in 2.3 and it is shown to the user. During this process, learner is asked to press any key to go from one step to another step. With a few important steps, learner is provided with an option to see help on that step. This option allows the learner is allowed to repeat the whole exercise with many examples. Figure 3.12 shows the screen when the conversion is carried out.

3.4 Special cases in IEEE format:
Special cases are shown one by one with some explanation in the instruction window.

4. Advantages.
One of the main advantages of this package is that it provides good understanding to the learner about normalization in general and the process of normalizing a binary number in particular. Hidden bit principle also becomes clearer to the user. We clear the complexities that we imagine after normalization, on shifting the binary point to the left side, in order to get the normalized number in 1.f format. Learner can work with more examples, specially with some good numbers to know about the importance of normalization. For example, if a number 0.0000000075 is converted to binary, there are more than 20 zeros after the binary point. If we normalize this number instead of these zeros we get ones which are significant and increase the precision of the number. Such type of conversions are
tedious to do manually and consume more time. This package also provides good understanding about special cases.

5. Limitations:
Very small numbers and very big numbers are avoided for conversion due to the limited space available on the screen.
Given Real Number : 12.75
IEEE format selected : Single
Equivalent Binary Number : 1100.11

Normalization is going on!
Normalize the binary number. For this binary point is to be shifted either to the left hand side or to the right hand side.

1100.11 = 11.0011*2^-2

Fig. 3.10 Normalization of Binary Number.
### Real Number to IEEE

<table>
<thead>
<tr>
<th>Given Real Number</th>
<th>: 12.75</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE format selected</td>
<td>: Single</td>
</tr>
<tr>
<td>Equivalent Binary Number</td>
<td>: 1100.11</td>
</tr>
<tr>
<td>After Normalization</td>
<td>: .110011*2^-4</td>
</tr>
<tr>
<td></td>
<td>= 1.10011*2^-3</td>
</tr>
<tr>
<td>Sign bit</td>
<td>: 0</td>
</tr>
<tr>
<td>Exponent</td>
<td>: 130 = 10000010</td>
</tr>
<tr>
<td>Fractional part</td>
<td>: 10011</td>
</tr>
<tr>
<td>IEEE Single format in Binary</td>
<td>: 0100001010011000000000000000000000</td>
</tr>
<tr>
<td>IEEE Single format in Hexadecimal</td>
<td>: 414C0000</td>
</tr>
</tbody>
</table>

Press any key to continue.

*Fig. 3.11. Interactive Screen on representing given Real Number in IEEE single format.*
### IEEE to Real Number

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selected IEEE format is</td>
<td>Single</td>
</tr>
<tr>
<td>Given IEEE format in Hex format</td>
<td>414C0000</td>
</tr>
<tr>
<td>Given IEEE format in binary format</td>
<td>0100000101001100000000000000000000000000</td>
</tr>
<tr>
<td>Sign bit is</td>
<td>0</td>
</tr>
<tr>
<td>Exponent in binary format</td>
<td>10000010</td>
</tr>
<tr>
<td>Exponent in decimal</td>
<td>130</td>
</tr>
<tr>
<td>Fractional part in binary format</td>
<td>100110000000000000000000000000000000000000</td>
</tr>
<tr>
<td>Fractional part in decimal</td>
<td>1.593750</td>
</tr>
<tr>
<td>Real Number is</td>
<td>12.750000</td>
</tr>
</tbody>
</table>

Fig. 3.12 Interactive Screen on evaluating IEEE single format.