Course Details

Course Title : Data Structure and Algorithms (3 Cr.)

Course Code : CACS201

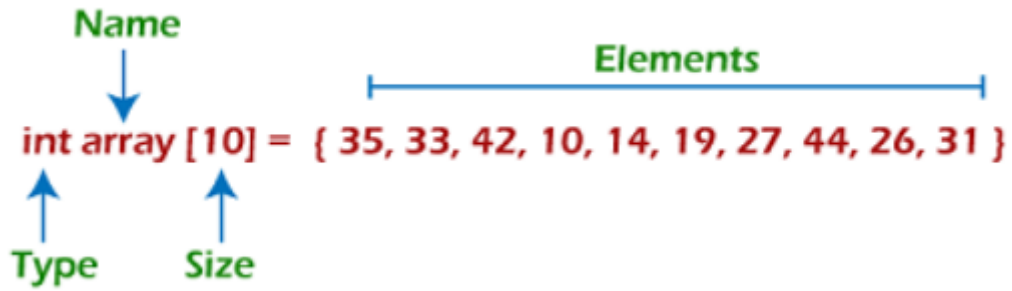
Why should we learn DSA?

1. Data Structures allow us to organize and store data, whereas Algorithms allow us to process that data meaningfully.
2. Learning Data Structures and Algorithms will help us become better Programmers.
3. We will be able to write code that is more effective and reliable.
4. We will also be able to solve problems more quickly and efficiently.

Unit 2

The Stack

Array



- Arrays are defined as the collection of similar types of data items stored at contiguous memory locations.
- It is one of the simplest data structures where each data element can be randomly accessed by using its index number.
- Basic operations supported in an array:
  - Traversal: This operation is used to print the elements of the array.
  - Insertion: It is used to add an element at a particular index.
  - Deletion: It is used to add an element at a particular index.
  - Search: It is used to search an element using the given index or by the value.
  - Update: It updates an element at a particular index.

Array

Algorithm for traversal operation:

```
function traversal(arr[], size):
for i = 0 to size-1:
print "arr[", i, "] = ", arr[i]
```

Algorithm for insert operation:

```
function insert(arr[], size, index, element):
for i = size-1 down to index:
arr[i+1] = arr[i]
arr[index] = element
size = size + 1
print "Element inserted successfully"
traversal(arr, size)
```

Algorithm for update operation:

```
function update(arr[], size, index, element):
arr[index] = element
print "Element updated successfully"
traversal(arr, size)
```

Algorithm for delete operation:

```
function delete(arr[], size, index):
for i = index to size-2:
arr[i] = arr[i+1]
size = size - 1
print "Element deleted successfully"
traversal(arr, size)
```

Algorithm for search operation:

```
function search(arr[], size, element):
for i = 0 to size-1:
if arr[i] == element:
print "Element found at index ", i
return
print "Element not found"
```

Array [Traversal operation]

- This operation is performed to traverse through the array elements. It prints all array elements one after another. We can understand it with the below program -

C

```
#include <stdio.h>

void traversal(int arr[], int size) {
  printf("------------------Traversal Operation------------------ \n");
  for (int i = 0; i < size; i++) {
    printf("arr[%d] = %d \n", i, arr[i]);
  }
}
void main() {
  int arr[5] = { 1, 3, 5, 7, 9 };

  int size = sizeof(arr) / sizeof(arr[0]);
  printf("Size of array is : %d \n", size);
  traversal(arr, size);

}
```

Java

Array [Insert operation]

- This operation is performed to insert one or more elements into the array. As per the requirements, an element can be added at the beginning, end, or at any index of the array.

C

```
#include <stdio.h>

void insert(int arr[], int size, int index, int element) {
  printf("------------------Insert Operation------------------ \n");

  for (int i = size - 1; i >= index; i--) {
    arr[i + 1] = arr[i];
  }

  arr[index] = element;
  size++;

  printf("Element inserted successfully \n");
  printf("Now the new array is : \n");
  traversal(arr, size);
}
```

```
void main() {
  int arr[5] = { 1, 3, 5, 7, 9 };

  int size = sizeof(arr) / sizeof(arr[0]);

  printf("Size of array is : %d \n", size);

  insert(arr, size, 2, 10);
}
```

Array [Delete operation]

- This operation is performed to insert one or more elements into the array. As per the requirements, an element can be added at the beginning, end, or at any index of the array.

C

```
#include <stdio.h>

void delete (int arr[], int size, int index) {
  printf("------------------Delete Operation------------------ \n");

  for (int i = index; i < size - 1; i++) {
    arr[i] = arr[i + 1];
  }

  size--;

  printf("Element deleted successfully \n");
  printf("Now the new array is : \n");
  traversal(arr, size);
}
```

```
void main() {
  int arr[5] = { 1, 3, 5, 7, 9 };

  int size = sizeof(arr) / sizeof(arr[0]);

  printf("Size of array is : %d \n", size);

  delete(arr, size, 2);
}
```

Array [Search operation]

- This operation is performed to insert one or more elements into the array. As per the requirements, an element can be added at the beginning, end, or at any index of the array.

C

```
#include <stdio.h>

void search(int arr[], int size, int element) {
  printf("------------------Search Operation------------------ \n");

  for (int i = 0; i < size; i++) {
    if (arr[i] == element) {
      printf("Element found at index %d \n", i);
      return;
    }
  }

  printf("Element not found \n");
}
```

```
void main() {
  int arr[5] = { 1, 3, 5, 7, 9 };

  int size = sizeof(arr) / sizeof(arr[0]);

  printf("Size of array is : %d \n", size);

  search(arr, size, 7);

}
```

Array [Update operation]

- This operation is performed to insert one or more elements into the array. As per the requirements, an element can be added at the beginning, end, or at any index of the array.

C

```
#include <stdio.h>

void update(int arr[], int size, int index, int element) {
  printf("------------------Update Operation------------------ \n");

  arr[index] = element;

  printf("Element updated successfully \n");
  printf("Now the new array is : \n");
  traversal(arr, size);
}

void main() {
  int arr[5] = { 1, 3, 5, 7, 9 };

  int size = sizeof(arr) / sizeof(arr[0]);

  printf("Size of array is : %d \n", size);

  update(arr, size, 2, 10);

}
```

Complexity of Array operations

- Time and space complexity of various array operations are described in the following table.

Time Complexity

Operation	Best Case	Worst Case
Access	Ω(1)	O(1)
Search	Ω(1)	O(n)
Insertion	Ω(1)	O(n)
Deletion	Ω(1)	O(n)

Space Complexity

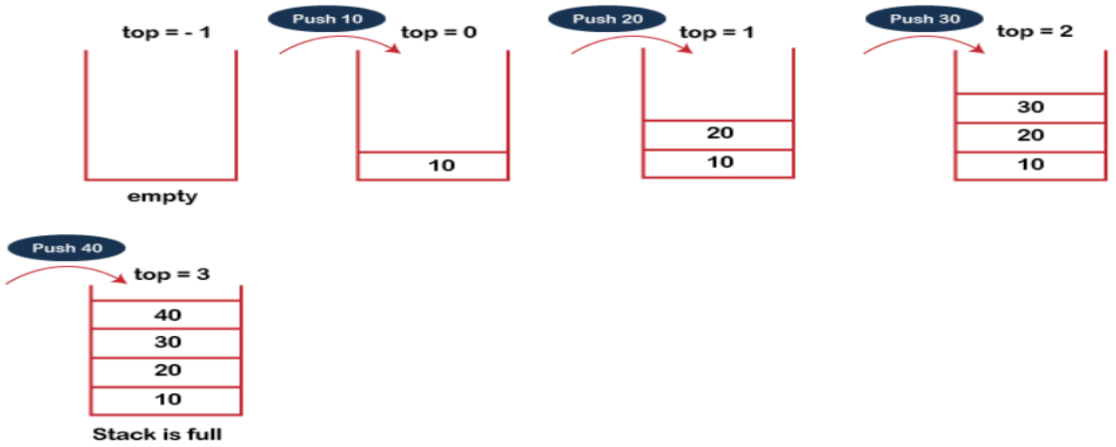
In array, space complexity for worst case is **O(n)**.

Stack

- A Stack is a linear data structure that follows the LIFO (Last-In-First-Out) principle.
- Stack has one end, whereas the Queue has two ends (front and rear).
- It contains only one pointer top pointer pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack.
- In other words, a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.
- **Stack Operations:**
  - **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
  - **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
  - **isEmpty():** It determines whether the stack is empty or not.
  - **isFull():** It determines whether the stack is full or not.'
  - **peek():** It returns the element at the given position.
  - **count():** It returns the total number of elements available in a stack.
  - **change():** It changes the element at the given position.
  - **display():** It prints all the elements available in the stack.

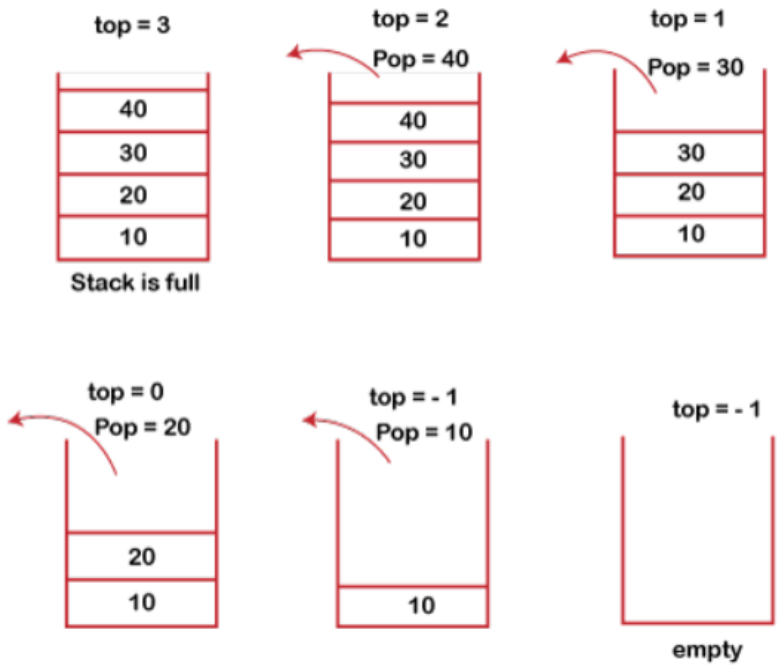
PUSH Operation

- The steps involved in the PUSH operation is given below:
  - Before inserting an element in a stack, we check whether the stack is full.
  - If we try to insert the element in a stack, and the stack is full, then the overflow condition occurs.
  - When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
  - When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., top=top+1, and the element will be placed at the new position of the top.
  - The elements will be inserted until we reach the max size of the stack.



POP Operation

- The steps involved in the POP operation is given below:
  - Before deleting the element from the stack, we check whether the stack is empty.
  - If we try to delete the element from the empty stack, then the underflow condition occurs.
  - If the stack is not empty, we first access the element which is pointed by the top
  - Once the pop operation is performed, the top is decremented by 1, i.e., top=top-1.



Applications of Stack

- **String reversal:** Stack is also used for reversing a string. For example, we want to reverse a “Hello World" string, so we can achieve this with the help of a stack.
  First, we push all the characters of the string in a stack until we reach the null character.
  After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.
- **UNDO/REDO:** It can also be used for performing UNDO/REDO operations. For example, we have an editor in which we write 'a', then 'b', and then 'c'; therefore, the text written in an editor is abc. So, there are three states, a, ab, and abc, which are stored in a stack. There would be two stacks in which one stack shows UNDO state, and the other shows REDO state.
  If we want to perform UNDO operation, and want to achieve 'ab' state, then we implement pop operation.
- **Recursion:** The recursion means that the function is calling itself again. To maintain the previous states, the compiler creates a system stack in which all the previous records of the function are maintained.
- **DFS(Depth First Search):** This search is implemented on a Graph, and Graph uses the stack data structure.

Applications of Stack

- **Backtracking:** Suppose we have to create a path to solve a maze problem. If we are moving in a particular path, and we realize that we come on the wrong way. In order to come at the beginning of the path to create a new path, we have to use the stack data structure.
- **Expression conversion:** Stack can also be used for expression conversion. This is one of the most important applications of stack. The list of the expression conversion is given below:
  - Infix to prefix
  - Infix to postfix
  - Prefix to infix
  - Prefix to postfix
  - Postfix to infix
- **Memory management:** The stack manages the memory. The memory is assigned in the contiguous memory blocks. The memory is known as stack memory as all the variables are assigned in a function call stack memory. The memory size assigned to the program is known to the compiler. When the function is created, all its variables are assigned in the stack memory. When the function completed its execution, all the variables assigned in the stack are released.

Insertion: push()

- push() is an operation that inserts elements into the stack. The following is an algorithm that describes the push() operation in a simpler way.

- **Algorithm**

  1 − Checks if the stack is full.

  2 − If the stack is full, produces an error and exit.

  3 − If the stack is not full, increments top to point next empty space.

  4 − Adds data element to the stack location, where top is pointing.

  5 − Returns success.

Deletion: pop()

- The pop() is a data manipulation operation which removes elements from the stack. The following pseudo code describes the pop() operation in a simpler way.
- **Algorithm**

  1 − Checks if the stack is empty.

  2 − If the stack is empty, produces an error and exit.

  3 − If the stack is not empty, accesses the data element at which top is pointing.

  4 − Decreases the value of top by 1.

  5 − Returns success.

peek()

- The peek() is an operation retrieves the topmost element within the stack, without deleting it. This operation is used to check the status of the stack with the help of the top pointer.
- **Algorithm**
  1. START
  2. return the element at the top of the stack
  3. END

isFull()

- isFull() operation checks whether the stack is full. This operation is used to check the status of the stack with the help of top pointer.
- **Algorithm**

  1. START
  2. If the size of the stack is equal to the top position of the stack, the stack is full. Return 1.
  3. Otherwise, return 0.
  4. END

isEmpty()

- The isEmpty() operation verifies whether the stack is empty. This operation is used to check the status of the stack with the help of top pointer.
- **Algorithm**

  1. START
  2. If the top value is -1, the stack is empty. Return 1.
  3. Otherwise, return 0.
  4. END

Complexity of Stack operations with Array implementation

- Time and space complexity of various array operations are described in the following table.

Operation	Best Case Time Complexity	Worst Case Time Complexity	Space Complexity
Push	Ω(1)	O(1)	O(1)
Pop	Ω(1)	O(1)	O(1)
Peek	Ω(1)	O(1)	O(1)
Search	Ω(n)	O(n)	O(1)

Introduction of Infix, Postfix and Prefix

Infix Notation

- Infix notation is the commonly used way of writing mathematical and logical expressions.
- Operators are placed between the operands in infix notation.
- Example: 3 + 4 is an infix expression where the "+" operator is placed between the operands 3 and 4.

Postfix Notation

- Postfix notation, also known as Reverse Polish Notation (RPN), is an alternative way of writing mathematical and logical expressions.
- Operators are placed after their operands in postfix notation.
- Example: 3 4 + is a postfix expression where the "+" operator follows the operands 3 and 4.

Prefix Notation

- Prefix notation, also known as Polish Notation, is another alternative way of writing mathematical and logical expressions.
- Operators are placed before their operands in prefix notation.
- Example: + 3 4 is a prefix expression where the "+" operator precedes the operands 3 and 4.

Infix to Postfix

Algorithm

1. Initialize an empty stack and an empty output list to store the postfix expression.
2. Scan the infix expression from left to right.
3. If the current token is an operand (operand can be a number or a variable), append it to the output list.
4. If the current token is an opening parenthesis '(', push it onto the stack.
5. If the current token is a closing parenthesis ')', pop elements from the stack and append them to the output list until an opening parenthesis is encountered. Discard the opening parenthesis.
6. If the current token is an operator (+, -, *, /, etc.), compare its precedence with the operator at the top of the stack.
   a. If the stack is empty or contains an opening parenthesis, push the current operator onto the stack.
   b. If the precedence of the current operator is higher than the operator at the top of the stack, push the current operator onto the stack.
   c. If the precedence of the current operator is lower than or equal to the operator at the top of the stack, pop operators from the stack and append them to the output list until a lower precedence operator is encountered or the stack becomes empty. Then, push the current operator onto the stack.
7. Repeat steps 3-6 until all tokens in the infix expression have been processed.
8. Pop any remaining operators from the stack and append them to the output list.
9. The output list will contain the postfix expression.

Infix to Postfix

Infix to Postfix

Expression : A + B * C / D - F + A ^ E

Scanned Symbol	Stack	Output	Reason
A		A	Step 2
+	+	A	Step 3.1
B	+	AB	Step 2
*	+*	AB	Step 3.1
C	+*	ABC	Step 2
/	+/	ABC*	Step 3.2
D	+/	ABC*D	Step 2
-	-	ABC*D/+	Step 3.2
F	-	ABC*D/+F	Step 2
+	+	ABC*D/+F-	Step 3.2
A	+	ABC*D/+F-A	Step 2
^	+^	ABC*D/+F-A	Step 2
E	+^	ABC*D/+F-AE	Step 2
(Empty)		ABC*D/+F-A^+	Step 8

Infix to Postfix

A * (B * C + D * E) + F

	Current Token	Operator Stack	Postfix String
1	A		A
2	*	*	A
3	(	*(	A
4	B	*(	AB
5	*	*(*	AB
6	C	*(*	ABC
7	+	*(+	ABC*
8	D	*(+	ABC*D
9	*	*(+*	ABC*D
10	E	*(+*	ABC*DE
11	)	*	ABC*DE*+
12	+	+	ABC*DE*+*
13	F	+	ABC*DE*+*F
14			ABC*DE*+*F+

Infix to Prefix

Algorithm

1. Create an empty stack.
2. Read the infix expression from right to left.
3. For each character in the infix expression, do the following:
    1. If the character is an operand (an alphanumeric character), add it to the output prefix expression.
    2. If the character is a closing parenthesis ')', push it onto the stack.
    3. If the character is an operator or an opening parenthesis '(':
        1. If the stack is empty or the top of the stack is a closing parenthesis ')', push the character onto the stack.
        2. If the character has higher precedence than the top of the stack, push the character onto the stack.
        3. If the character has lower or equal precedence than the top of the stack, pop operators from the stack and add them to the output prefix expression until a lower precedence operator is encountered or the stack becomes empty. Then push the character onto the stack.
4. After processing all the characters, pop any remaining operators from the stack and add them to the output prefix expression.
5. Reverse the output prefix expression to get the final prefix expression.

Infix to Prefix

Infix to Prefix Conversion

Infix Expression: (P + (Q * R) / (S - T))

Note: - Read the infix string in revers

Symbol Scanned	Stack	Output
)	)	-
)	))	-
T	))	T
-	))-	T
S	))-	ST
(	)	-ST
/	)/	-ST
)	)/)	-ST
R	)/)	R-ST
*	)/)*	R-ST
Q	)/)*	QR-ST
(	)/	*QR-ST
+	)+	/*QR-ST
P	)+	P/*QR-ST
(	Empty	+P/*QR-ST

Prefix Expression: +P/*QR-ST

Infix to Postfix and Prefix

Examples of infix to prefix and post fix

Infix	PostFix	Prefix
A+B	AB+	+AB
(A+B) * (C + D)	AB+CD+*	*+AB+CD
A-B/(C*D^E)	ABCDE^*/-	-A/B*C^DE

Infix to Postfix and Prefix Implementation

For code visit my GitHub repository: https://github.com/sujan-poudel-03/DSA

Thank You