ADT

An abstract data type is an abstraction of a data structure that provides only the interface to which the data structure must adhere. The interface does not give any specific details about something should be implemented or in what programming language.

In other words, we can say that abstract data types are the entities that are definitions of data and operations but do not have implementation details. In this case, we know the data that we are storing and the operations that can be performed on the data, but we don't know about the implementation details. The reason for not having implementation details is that every programming language has a different implementation strategy for example; a C data structure is implemented using structures while a C++ data structure is implemented using objects and classes.

For example, a List is an abstract data type that is implemented using a dynamic array and linked list. A queue is implemented using linked list-based queue, array-based queue, and stack-based queue. A Map is implemented using Tree map, hash map, or hash table.

IN SIMPLE WORDS………

ADT stands for Abstract Data Type, and it refers to a high-level description of a data structure that defines its properties, operations, and behavior. In computer science and programming, ADTs are used to specify the interface of a data structure, which can be implemented in any programming language or framework.

In the context of data structure algorithms (DSA), ADTs serve as a formal specification of the data structures that are used to store and manipulate data in an algorithm. The key benefit of using ADTs in DSA is that they provide a way to separate the conceptual aspects of a data structure from its implementation details. This enables DSA developers to focus on solving a problem using the right data structures, while leaving the details of implementation to the programming language or framework used.

Some of the commonly used ADTs in DSA include lists, arrays, stacks, queues, trees, graphs, and hash tables. These ADTs define the properties and operations of different types of data structures, and they help DSA developers to choose the right structure for a specific problem.

Example

An ADT for a list of integers might specify the following operations:

1. Insert a new integer at a particular position in the list.
2. Return True if the list is empty.
3. Reinitialize the list.
4. Return the number of integers currently in the list.
5. Retrieve the integer at a particular position in the list.
6. Delete the integer at a particular position in the list.

From this description, the input and output of each operation should be clear, but the implementation for lists has not been specified.

One application that makes use of some ADT might use particular member functions of that ADT more than a second application, or the two applications might have different time requirements for the various operations. These differences in the requirements of applications are the reason why a given ADT might be supported by more than one implementation.

Features of ADT:

Abstract data types (ADTs) are a way of encapsulating data and operations on that data into a single unit. Some of the key features of ADTs include:

Abstraction: The user does not need to know the implementation of the data structure only essentials are provided.

Better Conceptualization: ADT gives us a better conceptualization of the real world.

Robust: The program is robust and has the ability to catch errors.

Encapsulation: ADTs hide the internal details of the data and provide a public interface for users to interact with the data. This allows for easier maintenance and modification of the data structure.

Data Abstraction: ADTs provide a level of abstraction from the implementation details of the data. Users only need to know the operations that can be performed on the data, not how those operations are implemented.

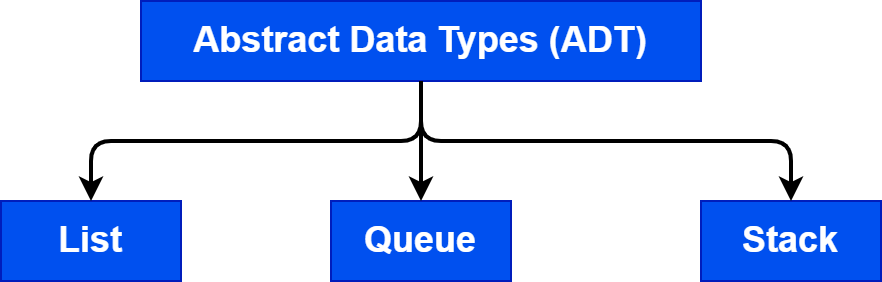
Data Structure Independence: ADTs can be implemented using different data structures, such as arrays or linked lists, without affecting the functionality of the ADT.

Information Hiding: ADTs can protect the integrity of the data by allowing access only to authorized users and operations. This helps prevent errors and misuse of the data.

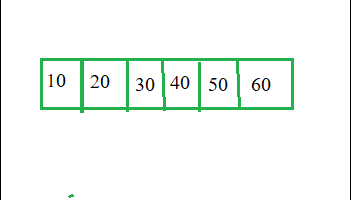
Modularity: ADTs can be combined with other ADTs to form larger, more complex data structures. This allows for greater flexibility and modularity in programming.

Overall, ADTs provide a powerful tool for organizing and manipulating data in a structured and efficient manner.

There are mainly three types of ADTs:



List ADT



Vies of list

The data is generally stored in key sequence in a list which has a head structure consisting of count, pointers and address of compare function needed to compare the data in the list.

The data node contains the pointer to a data structure and a self-referential pointer which points to the next node in the list.

The List ADT Functions is given below:

get() – Return an element from the list at any given position.

insert() – Insert an element at any position of the list.

remove() – Remove the first occurrence of any element from a non-empty list.

removeAt() – Remove the element at a specified location from a non-empty list.

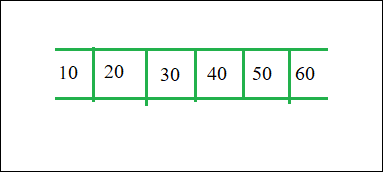
replace() – Replace an element at any position by another element.

size() – Return the number of elements in the list.

isEmpty() – Return true if the list is empty, otherwise return false.

isFull() – Return true if the list is full, otherwise return false.

2.Queue ADT



View of Queue

The queue abstract data type (ADT) follows the basic design of the stack abstract data type.

Each node contains a void pointer to the data and the link pointer to the next element in the queue. The program’s responsibility is to allocate memory for storing the data.

enqueue() – Insert an element at the end of the queue.

dequeue() – Remove and return the first element of the queue, if the queue is not empty.

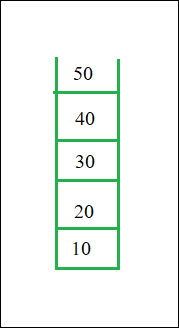
peek() – Return the element of the queue without removing it, if the queue is not empty.

size() – Return the number of elements in the queue.

isEmpty() – Return true if the queue is empty, otherwise return false.

isFull() – Return true if the queue is full, otherwise return false.

3.Stack ADT



View of stack

In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored.

The program allocates memory for the data and address is passed to the stack ADT.

The head node and the data nodes are encapsulated in the ADT. The calling function can only see the pointer to the stack.

The stack head structure also contains a pointer to top and count of number of entries currently in stack.

push() – Insert an element at one end of the stack called top.

pop() – Remove and return the element at the top of the stack, if it is not empty.

peek() – Return the element at the top of the stack without removing it, if the stack is not empty.

size() – Return the number of elements in the stack.

isEmpty() – Return true if the stack is empty, otherwise return false.

isFull() – Return true if the stack is full, otherwise return false.

## Introduction to Stack in Data Structure

A stack is a data structure that allows insertion and deletion operation in a LIFO (last-in-first-out) manner. The memory operations, therefore, are regulated in a particular manner. When an element is added to the stack, it occupies the top position. When it comes to removal operation, the most recent element in terms of being inserted into the stack gets first removed, hence the LIFO characteristic. This is similar to a stack of saucers or tiles, kept one over another. We keep going on placing saucers on over another, and while removing, the most recently added one is removed first.

### Features of Stack

Some of the features are as follows:

* A stack is an ordered list of elements of a similar type. E.g. stack storing numbers or stack of strings.
* It is essentially based on LIFO (Last-In-First-Out) feature or FILO (First-In-Last-Out) feature.
* When a stack is full, it is called to be in Overflow state, and when it is empty, it is called to be in Underflow state.

### How does Stack work in Data Structure?

A stack is a very simple data structure, and there are necessarily two operations associated with stack, which are Push and Pop. The working of the stack as a data structure can be understood through these operations. Let’s understand each of them one by one, as described below. Let’s understand the insertion and deletion (removal) operations in a stack as discussed below.

Standard Stack Operations

The following are some common operations implemented on the stack:

* 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.

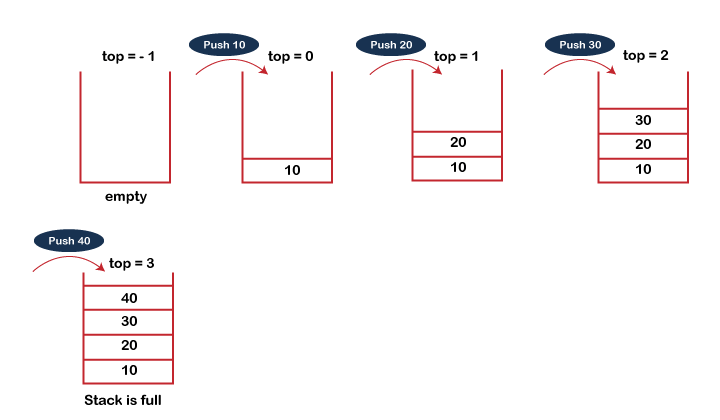
#### 1. Insertion Operation(push)

The operation to insert an element in a data structure is called Push operation. When we insert an element into a stack, it occupies the bottom-most position, as an object pushed into a pit. The next element inserted goes over the top of the previous element, and likewise, the insertion of all elements happens. As each time, the insertion operation resembles pushing an element into the stack, and the operation is termed as “posh operation”. We will understand the working of push operations through the following steps.

PUSH operation

The steps involved in the PUSH operation is given below:

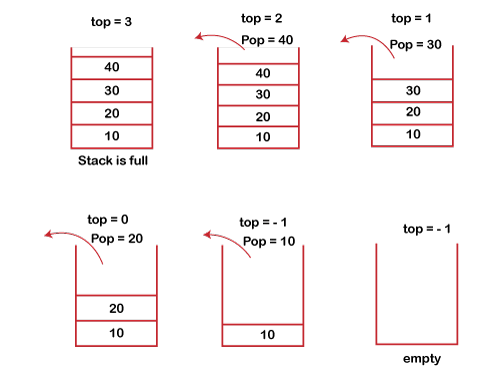
* Before inserting an element in a stack, we check whether the stack is full.
* If we try to insert the element in a stack, and the stack is full, then the overflow condition occurs.
* When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
* When the new element is 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.

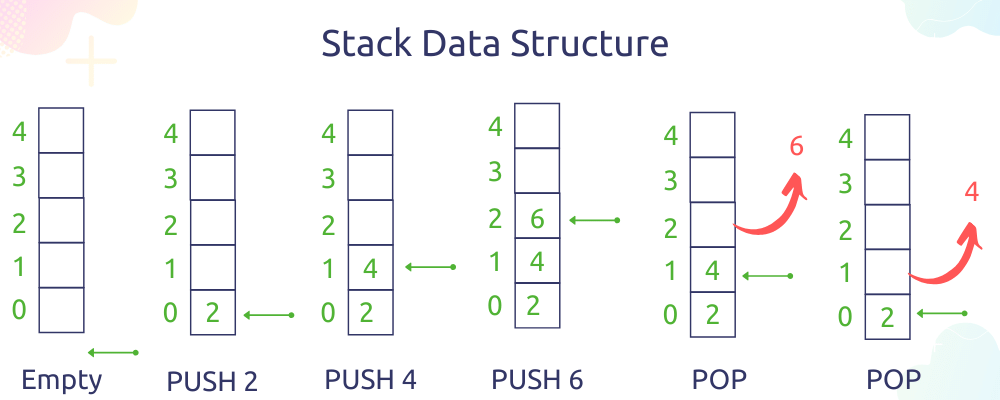


### Array Representation of Stack

This is the general representation of Stack.

We initialize an empty array of size n in which we perform PUSH and POP operations.

The array Stack contains the element from starting index up to the top pointer which represents the last element of the stack.



[Open Image](https://holycoders.com/content/images/wordpress/2020/04/Stack-Data-structure-push-and-pop.png)

Stack using Array

When the topPointer = -1, it means the stack is empty and when the topPointer = n it means the stack is full.

It can be very helpful to check overflow and underflow condition.

Here is the pseudo code for Stack using array arr of size n.

check\_Overflow

if(top==n)

return True

else

return False

check\_UnderFlow

if(top==-1)

return True

else

return False

PUSH\_OPERATION(data)

if(check Overflow)

print("Stack Over flow, No more space left")

return

else

top = top+1

arr[top] = data

POP\_OPERATION

if(check Underflow)

print("Stack Underflow, No more elements left")

return

else

top = top-1

return arr[top+1]

This is the pseudo code to create stack using an array. You can create a class in any language and implement these methods.

It is necessary to implement methods to check overflow and underflow condition because it is a common error.

Program using c lang..

#include <stdio.h>

#define SIZE 10

int S[SIZE+1];

int top = 0;

int is\_empty() {

if(top == 0)

return 1;

return 0;

}

void push(int x) {

top = top+1;

if(top > SIZE) {

printf("Stack Overflow\n");

}

else {

S[top] = x;

}

}

int pop() {

if(is\_empty()) {

printf("Stack Underflow\n");

return -1000;

}

else {

top = top-1;

return S[top+1];

}

}

int main() {

pop();

push(10);

push(20);

push(30);

push(40);

push(50);

push(60);

push(70);

push(80);

push(90);

push(100);

push(110);

int i;

for(i=1; i<=SIZE; i++) {

printf("%d\n",S[i]);

}

return 0;

}

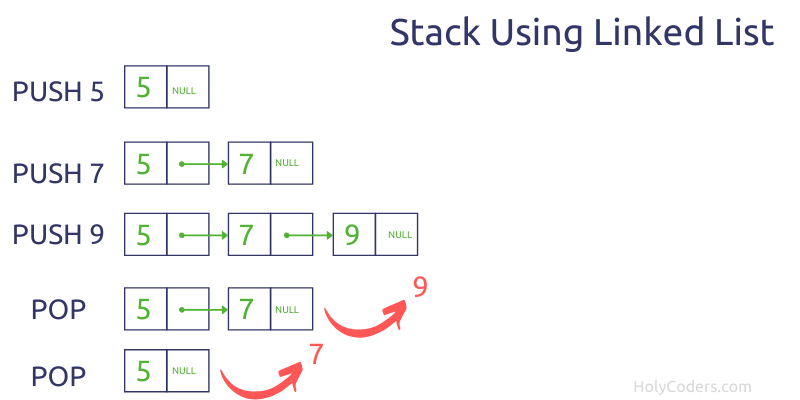
### Linked List Representation of Stack

Before moving to this part you must know about linked list and the difference between array and linked list.

* [Linked List Complete Guide.](https://holycoders.com/data-structures-linked-list/)

A linked list has some advantages and disadvantages over an array which make it more suitable for some specific tasks.

They have dynamic length, it means there is no limit on the number of elements we can add. It is convenient to use a linked list in place of an array when the number of elements is unknown.



[Open Image](https://holycoders.com/content/images/wordpress/2020/04/stack-using-linked-list.png)

Stack using Linked list

Here are some quick facts about Stack using linked lists which will be helpful to make a decision.

PUSH and POP operations in the linked list take O(1) constant time because a pointer is at the end of the linked list which manage insertion and deletion.

We only need to check the underflow condition here. If top pointer refers to null then the Stack is already empty.

Program using c lang…..

#include <stdio.h>

#include <stdlib.h>

typedef struct node {

int data;

struct node \*next;

}node;

typedef struct linked\_list {

struct node \*head;

struct node \*top;

}stack;

//to make new node

node\* new\_node(int data) {

node \*z;

z = malloc(sizeof(struct node));

z->data = data;

z->next = NULL;

return z;

}

//to make a new stack

stack\* new\_stack() {

stack \*s = malloc(sizeof(stack));

s->head = NULL;

s->top = NULL;

return s;

}

void traversal(stack \*s) {

node \*temp = s->head; //temporary pointer to point to head

while(temp != NULL) { //iterating over stack

printf("%d\t", temp->data);

temp = temp->next;

}

printf("\n");

}

int is\_empty(stack \*s) {

if(s->top == NULL)

return 1;

return 0;

}

void push(stack \*s, node \*n) {

if(is\_empty(s)) { //empty

s->head = n;

s->top = n;

}

else {

s->top->next = n;

s->top = n;

}

}

//function to delete

int pop(stack \*s) {

if(is\_empty(s)) {

printf("Stack Underflow\n");

return -1000;

}

else {

int x = s->top->data;

if(s->top == s->head) {// only one node

free(s->top);

s->top = NULL;

s->head = NULL;

}

else {

node \*temp = s->head;

while(temp->next != s->top) // iterating to the last element

temp = temp->next;

temp->next = NULL;

free(s->top);

s->top = temp;

}

return x;

}

}

int main() {

stack \*s = new\_stack();

node \*a, \*b, \*c;

a = new\_node(10);

b = new\_node(20);

c = new\_node(30);

pop(s);

push(s, a);

push(s, b);

push(s, c);

traversal(s);

pop(s);

traversal(s);

return 0;

}

## Applications of Stack Data Structure

As we already see that insertion and deletion take constant time which is performance efficient.

Here are some common application of Stack data structures.

* Back buttons in browsers
* Undo mechanism
* Evaluate prefix and postfix notation
* Backtracking
* Recursion
* Track Nested Function calls

## Recursion in data structures

* Recursion is one of the most powerful tools in a programming language, but one of the most threatening topics-as most of the beginners and not surprising to even experienced students feel.
* When function is called within the same function, it is known as recursion in C. The function which calls the same function, is known as recursive function.
* Recursion is defined as defining anything in terms of itself. Recursion is used to solve problems involving iterations, in reverse order.

### Types of Recursion

There are two types of Recursion

* Direct recursion
* Indirect recursion
* Tailed recursion
* Non- tailed recursion

### Direct Recursion

When in the body of a method there is a call to the same method, we say that the method is directly recursive.

There are three types of Direct Recursion

* Linear Recursion
* Binary Recursion
* Multiple Recursion

### Linear Recursion

* Linear recursion begins by testing for a set of base cases there should be at least one.

In Linear recursion we follow as under :

* Perform a single recursive call. This recursive step may involve a test that decides which of several possible recursive calls to make, but it should ultimately choose to make just one of these calls each time we perform this step.
* Define each possible recursion call, so that it makes progress towards a base case.

### ​​​​​​​Binary Recursion

* Binary recursion occurs whenever there are two recursive calls for each non base case.​​​​​​​

### Multiple Recursion

* In multiple recursion we make not just one or two but many recursive calls.

//C program for GCD using recursion

#include int

Find\_GCD(int, int);

void main()

{

int n1, n2, gcd;

scanf(“%d %d”,&n1, &n2);

gcd = Find\_GCD(n1, &n2);

printf(“GCD of %d and %d is %d”, n1, n2, gcd);

}

int Find\_GCD(int m, int n)

{

int gcdVal;

if(n>m)

{

gcdVal = Find\_GCD(n,m);

}

else if(n==0)

​​​​​​​ {

  gcdVal = m;

 }

 else

{

  gcdVal = Find\_GCD(n, m%n);

}

return(gcdVal);

}

### Direct Recursion in Data Structure

In the direct recursion, functions call themselves. This kind of operation consists of a single-stage recursive call by the function from within itself. Why don’t we investigate precisely how to carry out direct recursion to determine the square root of a number.

{

  // base case

    if (x == 0)

    {

      return x;

    }

  // recursive case

  else

   {

      return square(x-1) + (2\*x) – 1;

   }

}

int main() {

  // execution of square function

  int input=3;

  cout << input<<“^4 = “<<square(input);

  return 0;

}

The output would be displayed like this:

3^4 = 9

### Indirect Recursion in Data Structure

Indirect recursion happens when functions call other functions to call the initial function. This specific course of action consists of 2 simple steps when developing a recursive call, essentially making functions call functions to generate a recursive call. Mutual recursion could be referred to as indirect recursion.

Let’s examine precisely how to put into action indirect recursion to print or perhaps find out all of the figures from 10 to 20.

using namespace std;

int n=10;

// declaring functions

void foo1(void);

void foo2(void);

// defining recursive functions

void foo1()

{

  if (n <= 20)

  {

    cout<<n<<” “;  // prints n

    n++;           // increments n by 1

    foo2();       // calls foo2()

  }

  else

    return;

}

void foo2()

{

  if (n <= 20)

  {

    cout<<n<<” “;  // prints n

    n++;           // increments n by 1

    foo1();       // calls foo1()

  }

  else

    return;

}

The output would be displayed like this:

10 11 12 13 14 15 16 17 18 19 20

### Tailed Recursion in Data Structure

A recursive function is referred to as tail-recursive if the recursive call is the end execution executed by the function. Let us try to figure out the definition with the help of one example.

int fun(int z)

{

  printf(“%d”,z);

  fun(z-1);

  //Recursive call is last executed statement

}

If you notice this particular program, you can observe that the last line ADI is going to execute for method fun is a recursive call. And due to that, there’s no need to recall any earlier state of the program.

### Non-Tailed Recursion in Data Structure

A recursive function is said to be non-tail recursive in case the recursion call isn’t the final thing performed by the function. After reverting back, there’s another thing still there to assess. Now, look at the example.

int fun(int z)

{

  fun(z-1);

  printf(“%d”,z);

  //Recursive call is not the last executed statement

}

In this particular function, you can notice that there is another operation following the recursive call. Hence the ADI is going to have to remember the preceding state within this specific method block. That’s the reason this program could be regarded as non-tail recursive.

### Difference between recursion and iteration

|  |  |
| --- | --- |
| Iteration | Recursion |
| In iteration,a problem is converted into a train of steps that are finished one at a time, one after another | Recursion is like piling all of those steps on top of each other and then quashing the mall into the solution. |
| With iteration,each step clearly leads on to the next, like stepping stones across a river | In recursion, each step replicates itself at a smaller scale, so that all of them combined together eventually solve the problem. |
| Any iterative problem is solved recursively | Not all recursive problem can solved by iteration |
| It does not use Stack | It uses Stack |