**Data Structure Topics**

|  |  |
| --- | --- |
| **Data Structure & Algorithms (DSA)**   * Categories of Algorithm * Characteristics of Algorithm * Data Type * Basic operations in Data Structure     **Algorithm Complexity**   * Time Complexity * Space Complexity * Execution Time Cases   + Worst Case   + Average Case   + Best Case * Big O Notation   **Divide & Conquer**    **Dynamic Programming**  **Leaner Programming**  **Parallel Programming**  **Array**  **Linked List**   * Singly Linked List * Doubly Linked List * Circular Linked List | **Stack & Queue**   * Push * Pop * Enqueue * Dequeue   **Searching**   * Linear Search * Binary Search   **Sorting**   * Selection Sort * Bubble Sort * Insertion Sort * Quick Sort * Merge Sort * Heap Sort   **Tree**   * Binary Tree * Tree Traversal * AVL Tree   **Graph**   * Shortest Path * Most Direct Route * Greedy Algorithms * Breadth First Search (BFS) * Depth First Search (DFS) * Dijkstra's Algorithm |

**Data Structure & Algorithms (DSA)**

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. An algorithm can be implemented in more than one programming language.

**Categories of Algorithm or Basic operations in Data Structure:**

From the data structure point of view, following are some important categories of algorithms –

* **Searching −** Algorithm to search an item in a data structure.
* **Sorting −** Algorithm to sort items in a certain order.
* **Insertion −** Algorithm to insert item in a data structure.
* **Update −** Algorithm to update an existing item in a data structure.
* **Deletion −** Algorithm to delete an existing item from a data structure.
* **Traversing**
* **Merging**

**Characteristics of Algorithm:**

Not all procedures can be called an algorithm. An algorithm should have the following characteristics:

* **Unambiguous −** Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
* **Input −** An algorithm should have 0 or more well-defined inputs.
* **Output −** An algorithm should have 1 or more well-defined outputs, and should match the desired output.
* **Finiteness −** Algorithms must terminate after a finite number of steps.
* **Feasibility −** Should be feasible with the available resources.
* **Independent −** An algorithm should have step-by-step directions, which should be independent of any programming code.

**Data Type**

Data type is a way to classify various types of data such as integer, string, etc. which determines the values that can be used with the corresponding type of data, the type of operations that can be performed on the corresponding type of data. There are two data types −

* Built-in Data Type
* Derived Data Type

**Built-in Data Type**

Those data types for which a language has built-in support are known as Built-in Data types. For example, most of the languages provide the following built-in data types.

* Integers
* Boolean (true, false)
* Floating (Decimal numbers)
* Character and Strings

**Derived Data Type**

Those data types which are implementation independent as they can be implemented in one or the other way are known as derived data types. These data types are normally built by the combination of primary or built-in data types and associated operations on them. For example −

* List
* Array
* Stack
* Queue

**Algorithm Complexity**

Suppose X is an algorithm and n is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

**Time Factor −** Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.

**Space Factor −** Space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm f(n) gives the running time and/or the storage space required by the algorithm in terms of n as the size of input data.

**Time Complexity**

Running time or the execution time of operations of data structure must be as small as possible. Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n-bit integers takes n steps. Consequently, the total computational time is T(n) = c ∗ n, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.

**Space Complexity**

Memory usage of a data structure operation should be as little as possible. Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components −

* A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
* A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I.

Following is a simple example that tries to explain the concept −

**Algorithm: SUM (A, B)**

Step 1 - START

Step 2 - C ← A + B + 10

Step 3 - Stop

Here we have three variables A, B, and C and one constant. Hence S(P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

**Execution Time Cases**

**Worst Case (**Maximum time required for program execution**) –**

This is the scenario where a particular data structure operation takes maximum time it can take. If an operation's worst-case time is ƒ(n) then this operation will not take more than ƒ(n) time where ƒ(n) represents function of n.

**Average Case (**Average time required for program execution**) –**

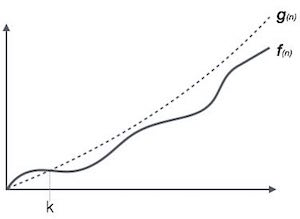
This is the scenario depicting the average execution time of an operation of a data structure. If an operation takes ƒ(n) time in execution, then m operations will take mƒ(n) time.

**Best Case (**Minimum time required for program execution) **–**

This is the scenario depicting the least possible execution time of an operation of a data structure. If an operation takes ƒ(n) time in execution, then the actual operation may take time as the random number which would be maximum as ƒ(n).

**Big O Notation**

The notation Ο(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst-case time complexity or the longest amount of time an algorithm can possibly take to complete.



For example, for a function f(n)

Ο(*f*(n)) = { *g*(n) : there exists c > 0 and n0 such that *f*(n) ≤ c.*g*(n) for all n > n0. }

**Understanding of Big O Notation:**

## O(1)

O(1) describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.

bool IsFirstElementNull(IList<string> elements)

{

return elements[0] == null;

}

## O(N)

O(N) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set. The example below also demonstrates how Big O favours the worst-case performance scenario; a matching string could be found during any iteration of the for loop and the function would return early, but Big O notation will always assume the upper limit where the algorithm will perform the maximum number of iterations.

bool ContainsValue(IList<string> elements, string value)

{

foreach (var element in elements)

{

if (element == value) return true;

}

return false;

}

## O(N2)

O(N2) represents an algorithm whose performance is directly proportional to the square of the size of the input data set. This is common with algorithms that involve nested iterations over the data set.

Deeper nested iterations will result in O(N3), O(N4) etc.

bool ContainsDuplicates(IList<string> elements)

{

for (var outer = 0; outer < elements.Count; outer++)

{

for (var inner = 0; inner < elements.Count; inner++)

{

// Don't compare with self

if (outer == inner) continue;

if (elements[outer] == elements[inner]) return true;

}

}

return false;

}

## O(2N)

O(2N) denotes an algorithm whose growth doubles with each addition to the input data set. The growth curve of an O(2N) function is exponential - starting off very shallow, then rising meteorically.

An example of an O(2N) function is the recursive calculation of Fibonacci numbers:

int Fibonacci(int number)

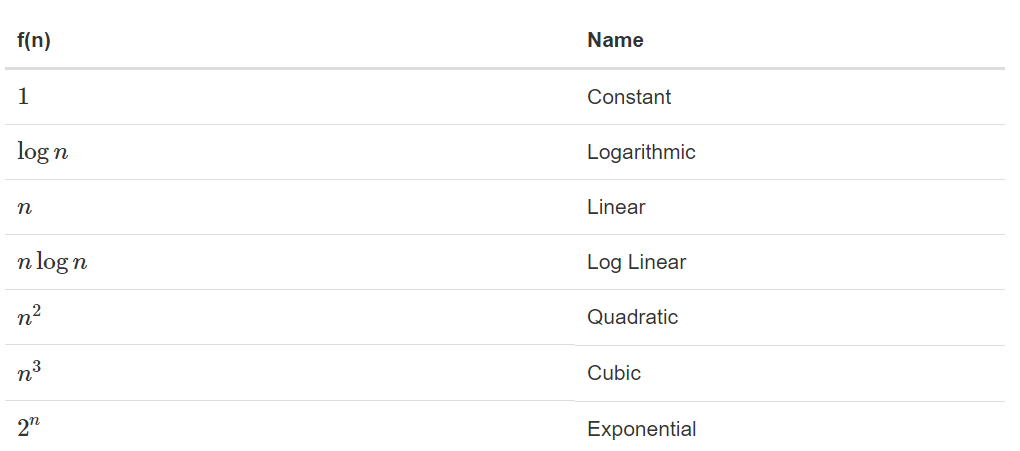
{

if (number <= 1) return number;

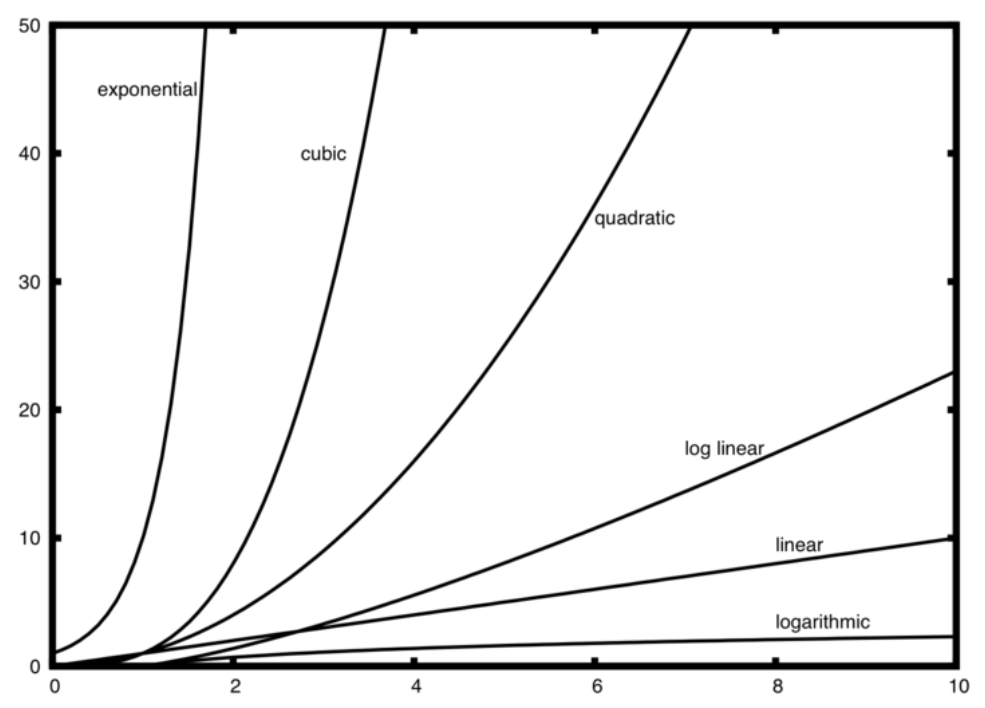
return Fibonacci(number - 2) + Fibonacci(number - 1);

}

**Table 1**



**Fig. 1**



**Example:**

a=5

b=6

c=10

**for** i **in** range(n):

**for** j **in** range(n):

x = i \* i

y = j \* j

z = i \* j

**for** k **in** range(n):

w = a\*k + 45

v = b\*b

d = 33

The number of assignment operations is the sum of four terms.

The first term is the constant 3 i.e. 1+1+1, representing the three assignment statements at the start of the fragment.

The second term is 3n2, since there are three statements that are performed n2 times due to the nested iteration.

The third term is 2n, two statements iterated n times.

Finally, the fourth term is the constant 1, representing the final assignment statement. This gives us

T(n)=3+3n2+2n+1=3n2+2n+4T(n)=3+3n2+2n+1=3n2+2n+4.

By looking at the exponents, we can easily see that the n2n2 term will be dominant and therefore this fragment of code is O(n2).

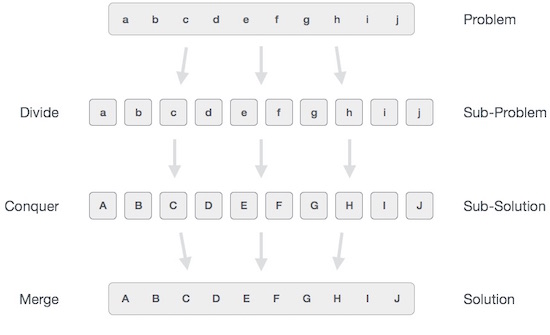
Note that all of the other terms as well as the coefficient on the dominant term can be ignored as n grows larger.

**Complexity of Algorithms:**

1. Time Complexity of binary Search: O (log n)

**Divide & Conquer**

In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. When we keep on dividing the subproblems into even smaller sub-problems, we may eventually reach a stage where no more division is possible. Those "atomic" smallest possible sub-problem (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.



We can understand divide-and-conquer approach in a three-step process.

**Divide/Break**

This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic in nature but still represent some part of the actual problem.

**Conquer/Solve**

This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.

**Merge/Combine**

When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem. This algorithmic approach works recursively and conquer & merge steps works so close that they appear as one.

**Greedy Algorithms**

As name suggest, this algorithm work on greedy manner like short time/immediate solution.

Greedy algorithms try to find a localized optimum solution. Greedy algorithms do not provide globally optimized solutions.

In greedy algorithm approach, decisions are made from the given solution domain. As being greedy, the closest solution that seems to provide an optimum solution is chosen.

**Dynamic Programming**

Dynamic programming approach is similar to divide and conquer in breaking down the problem into smaller and yet smaller possible sub-problems. But unlike, divide and conquer, these sub-problems are not solved independently. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.

Dynamic programming is used where we have problems, which can be divided into similar sub-problems, so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems. The solutions of sub-problems are combined in order to achieve the best solution.

So we can say that −

* The problem should be able to be divided into smaller overlapping sub-problem.
* An optimum solution can be achieved by using an optimum solution of smaller sub-problems.
* Dynamic algorithms use memorization.

**Example of dynamic programming**

The following computer problems can be solved using dynamic programming approach −

* Fibonacci number series
* Knapsack problem
* Tower of Hanoi
* All pair shortest path by Floyd-Warshall
* Shortest path by Dijkstra
* Project scheduling

Dynamic programming can be used in both top-down and bottom-up manner. And of course, most of the times, referring to the previous solution output is cheaper than recomputing in terms of CPU cycles.

**Comparison with Divide and Conquer and Greedy Algorithm**

* In contrast to greedy algorithms, where local optimization is addressed, dynamic algorithms are motivated for an overall optimization of the problem.
* In contrast to divide and conquer algorithms, where solutions are combined to achieve an overall solution, dynamic algorithms use the output of a smaller sub-problem and then try to optimize a bigger sub-problem. Dynamic algorithms use memorization to remember the output of already solved sub-problems.

**Array**

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

* **Element −** Each item stored in an array is called an element.
* **Index −** Each location of an element in an array has a numerical index, which is used to identify the element.

Arrays can be declared in various ways in different languages.



**Basic Operations on an Array -**

Following are the basic operations supported by an array.

* **Traverse −** print all the array elements one by one.
* **Insertion −** Adds an element at the given index.
* **Deletion −** Deletes an element at the given index.
* **Search −** Searches an element using the given index or by the value.
* **Update −** Updates an element at the given index.

**Linked List**

A linked list is a sequence of data structures, which are connected together via links. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

* **Link** − Each link of a linked list can store a data called an element.
* **Next** − Each link of a linked list contains a link to the next link called Next.
* **LinkedList** − A Linked List contains the connection link to the first link called First.

**Types of Linked List**

Following are the various types of linked list.

* **Simple Linked List** − Item navigation is forward only.
* **Doubly Linked List** − Items can be navigated forward and backward.
* **Circular Linked List** − Last item contains link of the first element as next and the first element has a link to the last element as previous.

**Basic Operations on Linked List**

Following are the basic operations supported by a list.

* **Insertion**
* **Deletion**
* **Display**
* **Searching**
* **Traversing**

**Singly Linked List**

In singly linked list, each node has the address of its next node. If is node don’t have the address of its next node then it means this node is the last node of the list.



As per the above illustration, following are the important points to be considered.

* Linked List contains a link element called first.
* Each link carries a data field(s) and a link field called next.
* Each link is linked with its next link using its next link.
* Last link carries a link as null to mark the end of the list.

**Insertion operation in singly linked list:**

**Fig. 1**



**Fig. 2**



**Fig. 3**



**Fig. 4**



**Insertion into Singly Linked List**

**Deletion from Singly Linked List**

**Doubly Linked List**

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.

**Link** − Each link of a linked list can store a data called an element.

**Next** − Each link of a linked list contains a link to the next link called Next.

**Prev** − Each link of a linked list contains a link to the previous link called Prev.

**LinkedList** − A Linked List contains the connection link to the first link called First and to the last link called Last.



As per the above illustration, following are the important points to be considered.

* Doubly Linked List contains a link element called first and last.
* Each link carries a data field(s) and two link fields called next and prev.
* Each link is linked with its next link using its next link.
* Each link is linked with its previous link using its previous link.
* The last link carries a link as null to mark the end of the list.

**Insertion into doubly linked list**

//insert link at the first location

void insertFirst(int key, int data) {

//create a link

struct node \*link = (struct node\*) malloc(sizeof(struct node));

link->key = key;

link->data = data;

if(isEmpty()) {

//make it the last link

last = link;

} else {

//update first prev link

head->prev = link;

}

//point it to old first link

link->next = head;

//point first to new first link

head = link;

}

**Deletion from doubly linked list**

//delete first item

struct node\* deleteFirst() {

//save reference to first link

struct node \*tempLink = head;

//if only one link

if(head->next == NULL) {

last = NULL;

} else {

head->next->prev = NULL;

}

head = head->next;

//return the deleted link

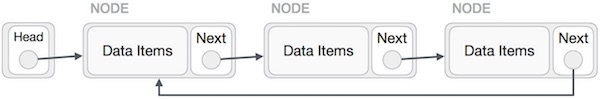
return tempLink;

}

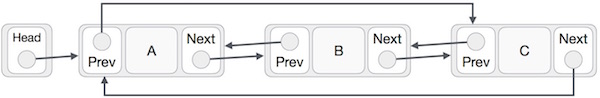
**Circular Linked List**

Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

**Singly Linked List as Circular**



**Doubly Linked List as Circular**



**Insertion into Circular Linked List**

//insert link at the first location

void insertFirst(int key, int data) {

//create a link

struct node \*link = (struct node\*) malloc(sizeof(struct node));

link->key = key;

link->data= data;

if (isEmpty()) {

head = link;

head->next = head;

} else {

//point it to old first node

link->next = head;

//point first to new first node

head = link;

}

}

**Deletion from Circular Linked List**

//delete first item

struct node \* deleteFirst() {

//save reference to first link

struct node \*tempLink = head;

if(head->next == head) {

head = NULL;

return tempLink;

}

//mark next to first link as first

head = head->next;

//return the deleted link

return tempLink;

}

**Display from Circular Linked List**

//display the list

void printList() {

struct node \*ptr = head;

printf("\n[ ");

//start from the beginning

if(head != NULL) {

while(ptr->next != ptr) {

printf("(%d,%d) ",ptr->key,ptr->data);

ptr = ptr->next;

}

}

printf(" ]");

}

**Stack**

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called PUSH operation and removal operation is called POP operation.



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

**Basic functions on Stack**

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations –

* **push()** − Pushing (storing) an element on the stack.
* **pop()** − Removing (accessing) an element from the stack.
* **peek()** − get the top data element of the stack, without removing it.
* **isFull()** − check if stack is full.
* **isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

**Algorithm of peek() function**

begin procedure peek

return stack[top]

end procedure

**Implementation of peek() function in C programming language**

int peek() {

return stack[top];

}

**Algorithm of isfull() function –**

begin procedure isfull

if top equals to MAXSIZE

return true

else

return false

endif

end procedure

**Implementation of isfull() function in C programming language −**

bool isfull() {

if(top == MAXSIZE)

return true;

else

return false;

}

**Algorithm of isempty() function –**

begin procedure isempty

if top less than 1

return true

else

return false

endif

end procedure

**Implementation of isempty() function in C programming language**

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code −

bool isempty() {

if(top == -1)

return true;

else

return false;

}

## Push Operation on Stack

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps –

* **Step 1** − Checks if the stack is full.
* **Step 2** − If the stack is full, produces an error and exit.
* **Step 3** − If the stack is not full, increments **top** to point next empty space.
* **Step 4** − Adds data element to the stack location, where top is pointing.
* **Step 5** − Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

### Algorithm for PUSH Operation

begin procedure push: stack, data

if stack is full

return null

endif

top ← top + 1

stack[top] ← data

end procedure

**Implementation of this algorithm in C, is very easy**

void push(int data) {

if(!isFull()) {

top = top + 1;

stack[top] = data;

} else {

printf("Could not insert data, Stack is full.\n");

}

}

## Pop Operation on Stack

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps −

* **Step 1** − Checks if the stack is empty.
* **Step 2** − If the stack is empty, produces an error and exit.
* **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
* **Step 4** − Decreases the value of top by 1.
* **Step 5** − Returns success.



### Algorithm for Pop Operation

begin procedure pop: stack

if stack is empty

return null

endif

data ← stack[top]

top ← top - 1

return data

end procedure

**Implementation of this algorithm in C, is as follows −**

int pop(int data) {

if(!isempty()) {

data = stack[top];

top = top - 1;

return data;

} else {

printf("Could not retrieve data, Stack is empty.\n");

}

}

**Complete implementation of Stack**

#include <stdio.h>

int MAXSIZE = 8;

int stack[8];

int top = -1;

int isempty() {

if(top == -1)

return 1;

else

return 0;

}

int isfull() {

if(top == MAXSIZE)

return 1;

else

return 0;

}

int peek() {

return stack[top];

}

int pop() {

int data;

if(!isempty()) {

data = stack[top];

top = top - 1;

return data;

} else {

printf("Could not retrieve data, Stack is empty.\n");

}

}

int push(int data) {

if(!isfull()) {

top = top + 1;

stack[top] = data;

} else {

printf("Could not insert data, Stack is full.\n");

}

}

int main() {

// push items on to the stack

push(3);

push(5);

push(9);

push(1);

push(12);

push(15);

printf("Element at top of the stack: %d\n" ,peek());

printf("Elements: \n");

// print stack data

while(!isempty()) {

int data = pop();

printf("%d\n",data);

}

printf("Stack full: %s\n" , isfull()?"true":"false");

printf("Stack empty: %s\n" , isempty()?"true":"false");

return 0;

}

**Queue**

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.



**Basic functions on Queue**

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues –

* **enqueue()** − add (store) an item to the queue.
* **dequeue()** − remove (access) an item from the queue.
* **peek()** − Gets the element at the front of the queue without removing it.
* **isfull()** − Checks if the queue is full.
* **isempty()** − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueuing (or storing) data in the queue we take help of **rear** pointer.

**The algorithm of peek() function is as follows −**

**Algorithm of isfull() function −**

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ

**Algorithm of isempty() function −**

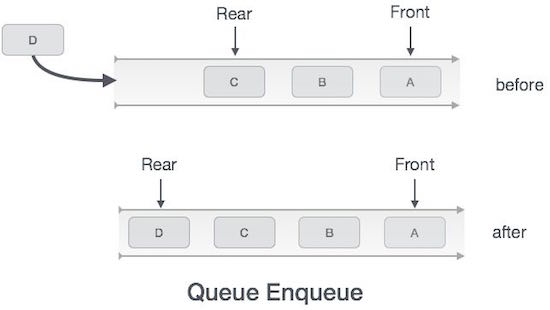
If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code −

## Enqueue Operation on Queue

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks. The following steps should be taken to enqueue (insert) data into a queue –

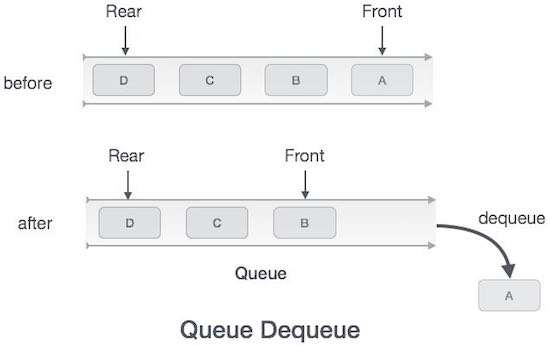
* **Step 1** − Check if the queue is full.
* **Step 2** − If the queue is full, produce overflow error and exit.
* **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
* **Step 4** − Add data element to the queue location, where the rear is pointing.
* **Step 5** − return success.



**Dequeue Operation on queue**

Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation –

* **Step 1** − Check if the queue is empty.
* **Step 2** − If the queue is empty, produce underflow error and exit.
* **Step 3** − If the queue is not empty, access the data where **front** is pointing.
* **Step 4** − Increment **front** pointer to point to the next available data element.
* **Step 5** − Return success.



**Complete implementation of Queue**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX 6

int intArray[MAX];

int front = 0;

int rear = -1;

int itemCount = 0;

int peek() {

return intArray[front];

}

bool isEmpty() {

return itemCount == 0;

}

bool isFull() {

return itemCount == MAX;

}

int size() {

return itemCount;

}

void insert(int data) {

if(!isFull()) {

if(rear == MAX-1) {

rear = -1;

}

intArray[++rear] = data;

itemCount++;

}

}

int removeData() {

int data = intArray[front++];

if(front == MAX) {

front = 0;

}

itemCount--;

return data;

}

int main() {

/\* insert 5 items \*/

insert(3);

insert(5);

insert(9);

insert(1);

insert(12);

// front : 0

// rear : 4

// ------------------

// index : 0 1 2 3 4

// ------------------

// queue : 3 5 9 1 12

insert(15);

// front : 0

// rear : 5

// ---------------------

// index : 0 1 2 3 4 5

// ---------------------

// queue : 3 5 9 1 12 15

if(isFull()) {

printf("Queue is full!\n");

}

// remove one item

int num = removeData();

printf("Element removed: %d\n",num);

// front : 1

// rear : 5

// -------------------

// index : 1 2 3 4 5

// -------------------

// queue : 5 9 1 12 15

// insert more items

insert(16);

// front : 1

// rear : -1

// ----------------------

// index : 0 1 2 3 4 5

// ----------------------

// queue : 16 5 9 1 12 15

// As queue is full, elements will not be inserted.

insert(17);

insert(18);

// ----------------------

// index : 0 1 2 3 4 5

// ----------------------

// queue : 16 5 9 1 12 15

printf("Element at front: %d\n",peek());

printf("----------------------\n");

printf("index : 5 4 3 2 1 0\n");

printf("----------------------\n");

printf("Queue: ");

while(!isEmpty()) {

int n = removeData();

printf("%d ",n);

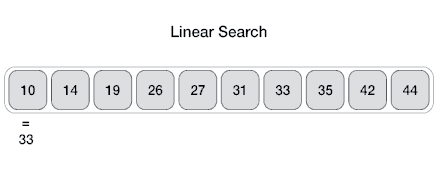
}

}

**Searching**

**Linear Search**

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.



**Implementation of Linear Search**

#include <stdio.h>

#define MAX 20

// array of items on which linear search will be conducted.

int intArray[MAX] = {1,2,3,4,6,7,9,11,12,14,15,16,17,19,33,34,43,45,55,66};

void printline(int count) {

int i;

for(i = 0;i <count-1;i++) {

printf("=");

}

printf("=\n");

}

// this method makes a linear search.

int find(int data) {

int comparisons = 0;

int index = -1;

int i;

// navigate through all items

for(i = 0;i<MAX;i++) {

// count the comparisons made

comparisons++;

// if data found, break the loop

if(data == intArray[i]) {

index = i;

break;

}

}

printf("Total comparisons made: %d", comparisons);

return index;

}

void display() {

int i;

printf("[");

// navigate through all items

for(i = 0;i<MAX;i++) {

printf("%d ",intArray[i]);

}

printf("]\n");

}

void main() {

printf("Input Array: ");

display();

printline(50);

//find location of 1

int location = find(55);

// if element was found

if(location != -1)

printf("\nElement found at location: %d" ,(location+1));

else

printf("Element not found.");

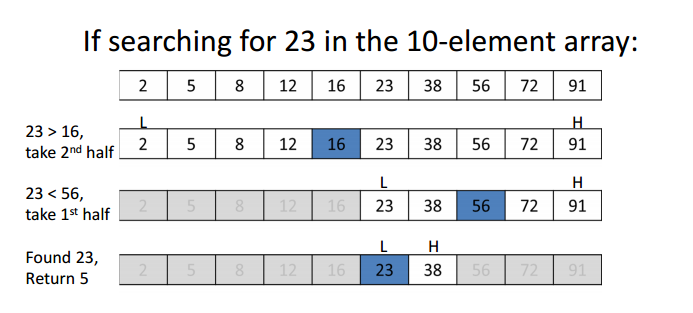
}

**Binary Search**

Binary search is a fast search algorithm with run-time complexity of Ο(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form. Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

**How Binary Search Works?**

For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



Procedure binary\_search

A ← sorted array

n ← size of array

x ← value to be searched

Set lowerBound = 1

Set upperBound = n

while x not found

if upperBound < lowerBound

EXIT: x does not exists.

set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

if A[midPoint] < x

set lowerBound = midPoint + 1

if A[midPoint] > x

set upperBound = midPoint - 1

if A[midPoint] = x

EXIT: x found at location midPoint

end while

end procedure

**Implementation of Binary Search**

|  |
| --- |
| #include "stdafx.h"  #include <iostream>  using namespace std;  void binarySearch();  int binarySearch\_iterativeMethod(int arr[], int l, int r, int x);  int binarySearch\_recursiveMethod(int arr[], int l, int r, int x);  int main()  {  binarySearch();  return 0;  }  #pragma region BinarySearch  void binarySearch()  {  int arr[] = { 2, 3, 4, 10, 40 };  int n = sizeof(arr) / sizeof(arr[0]);  int x = 10;  int result = binarySearch\_iterativeMethod(arr, 0, n - 1, x);  //int result = binarySearch\_recursiveMethod(arr, 0, n - 1, x);  if (result == -1)  {  cout << "Element is not present in the array!\n";  }  else  {  cout << "Element is present at index %d", result;  }  }  int binarySearch\_iterativeMethod(int arr[], int l, int r, int x)  {  while (r >= l)  {  int m = l + (r - l) / 2;  if (arr[m] == x) //if x present at mid.  {  return m;  }  if (arr[m] < x) //if x is greater than mid, ignore left half.  {  l = m + 1;  }  else //if x is less than mid, ignore right half.  {  r = m - 1;  }  }  return -1; //if reach here that means element is not present.  }  int binarySearch\_recursiveMethod(int arr[], int l, int r, int x)  {  if (r >= l)  {  int m = l + (r - l) / 2;  if (arr[m] == x) //if x present at mid.  {  return m;  }  if (arr[m] < x) //if x is greater than mid, ignore left half.  {  l = m + 1;  }  else //if x is less than mid, ignore right half.  {  r = m - 1;  }  return binarySearch\_recursiveMethod(arr, l, r, x);  }  return -1; //if reach here that means element is not present.  }  #pragma endregion |

**Sorting**

**Bubble sort**

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of Ο(n2) where **n** is the number of items.

Bubble sort starts with very first two elements, comparing them to check which one is greater.

Bubble Sort

In this case, value 33 is greater than 14, so no action is needed as it is already in sorted locations. Next, we compare 33 with 27. We find that 27 is smaller than 33 and these two values must be swapped.

Bubble Sort Bubble Sort

Next we compare 33 and 35. 35 is greater than 33 so no action is needed as it is already in sorted location. Now move to the next element and it is 10. Now we compare 35 with 10.

Bubble Sort Bubble Sort

Since 10 is smaller than 35. We need to swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this –

Bubble Sort

Similarly, after second iteration, it will look like:

Bubble Sort

And after final iteration it will looks like:

Bubble Sort

**Algorithm of Bubble Sort**

begin BubbleSort(list)

for all elements of list

if list[i] > list[i+1]

swap(list[i], list[i+1])

end if

end for

return list

end BubbleSort

**Complete implementation of Bubble sort**

#include <stdio.h>

#include <stdbool.h>

#define MAX 10

int list[MAX] = {1,8,4,6,0,3,5,2,7,9};

void display() {

int i;

printf("[");

// navigate through all items

for(i = 0; i < MAX; i++) {

printf("%d ",list[i]);

}

printf("]\n");

}

void bubbleSort() {

int temp;

int i,j;

bool swapped = false;

// loop through all numbers

for(i = 0; i < MAX-1; i++) {

swapped = false;

// loop through numbers falling ahead

for(j = 0; j < MAX-1-i; j++) {

printf(" Items compared: [ %d, %d ] ", list[j],list[j+1]);

// check if next number is lesser than current no

// swap the numbers.

// (Bubble up the highest number)

if(list[j] > list[j+1]) {

temp = list[j];

list[j] = list[j+1];

list[j+1] = temp;

swapped = true;

printf(" => swapped [%d, %d]\n",list[j],list[j+1]);

} else {

printf(" => not swapped\n");

}

}

// if no number was swapped that means

// array is sorted now, break the loop.

if(!swapped) {

break;

}

printf("Iteration %d#: ",(i+1));

display();

}

}

void main() {

printf("Input Array: ");

display();

printf("\n");

bubbleSort();

printf("\nOutput Array: ");

display();

}

**Insertion Sort**

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted.

An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. That why its name is **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array).

This algorithm is not suitable for large data sets as its average and worst case complexity are of Ο(n2), where **n** is the number of items.

Same like bubble sort, Insertion sort compares the first two elements i.e. 14 and 33.

Insertion Sort

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list. Insertion sort moves ahead and compares 33 with 27.

Insertion Sort

And finds that 33 is not in the correct position. It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

Insertion Sort

Next, it compares 33 with 10. These values are not in a sorted order. So we swap them.

Insertion Sort

Insertion Sort

However, swapping makes 27 and 10 unsorted. Hence, we swap them too.

Insertion Sort

Insertion Sort

Again we find 14 and 10 in an unsorted order. We swap them again.

Insertion Sort

Repeat the same kind of steps until getting the sorted list.

**Algorithm of Insertion Sort**

procedure insertionSort( A : array of items )

int holePosition

int valueToInsert

for i = 1 to length(A) inclusive do:

/\* select value to be inserted \*/

valueToInsert = A[i]

holePosition = i

/\*locate hole position for the element to be inserted \*/

while holePosition > 0 and A[holePosition-1] > valueToInsert do:

A[holePosition] = A[holePosition-1]

holePosition = holePosition -1

end while

/\* insert the number at hole position \*/

A[holePosition] = valueToInsert

end for

end procedure

**Complete implementation of Insertion sort**

#include <stdio.h>

#include <stdbool.h>

#define MAX 7

int intArray[MAX] = {4,6,3,2,1,9,7};

void printline(int count) {

int i;

for(i = 0;i < count-1;i++) {

printf("=");

}

printf("=\n");

}

void display() {

int i;

printf("[");

// navigate through all items

for(i = 0;i < MAX;i++) {

printf("%d ",intArray[i]);

}

printf("]\n");

}

void insertionSort() {

int valueToInsert;

int holePosition;

int i;

// loop through all numbers

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

// select a value to be inserted.

valueToInsert = intArray[i];

// select the hole position where number is to be inserted

holePosition = i;

// check if previous no. is larger than value to be inserted

while (holePosition > 0 && intArray[holePosition-1] > valueToInsert) {

intArray[holePosition] = intArray[holePosition-1];

holePosition--;

printf(" item moved : %d\n" , intArray[holePosition]);

}

if(holePosition != i) {

printf(" item inserted : %d, at position : %d\n" , valueToInsert,holePosition);

// insert the number at hole position

intArray[holePosition] = valueToInsert;

}

printf("Iteration %d#:",i);

display();

}

}

void main() {

printf("Input Array: ");

display();

printline(50);

insertionSort();

printf("Output Array: ");

display();

printline(50);

}

**Selection Sort**

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst-case complexities are of Ο(n2), where **n** is the number of items.

Unsorted Array

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

Selection Sort

So, we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

Selection Sort

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner. We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

Selection Sort

Selection Sort

The same process is applied to the rest of the items in the array.

**Algorithm of Selection Sort**

procedure selection sort

list : array of items

n : size of list

for i = 1 to n - 1

/\* set current element as minimum\*/

min = i

/\* check the element to be minimum \*/

for j = i+1 to n

if list[j] < list[min] then

min = j;

end if

end for

/\* swap the minimum element with the current element\*/

if indexMin != i then

swap list[min] and list[i]

end if

end for

end procedure

**Complete implementation of Selection Sort**

#include <stdio.h>

#include <stdbool.h>

#define MAX 7

int intArray[MAX] = {4,6,3,2,1,9,7};

void printline(int count) {

int i;

for(i = 0;i < count-1;i++) {

printf("=");

}

printf("=\n");

}

void display() {

int i;

printf("[");

// navigate through all items

for(i = 0;i < MAX;i++) {

printf("%d ", intArray[i]);

}

printf("]\n");

}

void selectionSort() {

int indexMin,i,j;

// loop through all numbers

for(i = 0; i < MAX-1; i++) {

// set current element as minimum

indexMin = i;

// check the element to be minimum

for(j = i+1;j < MAX;j++) {

if(intArray[j] < intArray[indexMin]) {

indexMin = j;

}

}

if(indexMin != i) {

printf("Items swapped: [ %d, %d ]\n" , intArray[i], intArray[indexMin]);

// swap the numbers

int temp = intArray[indexMin];

intArray[indexMin] = intArray[i];

intArray[i] = temp;

}

printf("Iteration %d#:",(i+1));

display();

}

}

void main() {

printf("Input Array: ");

display();

printline(50);

selectionSort();

printf("Output Array: ");

display();

printline(50);

}

**Merge Sort**

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being Ο(n log n), it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

Unsorted Array

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

Merge Sort Division

This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

Merge Sort Division

We further divide these arrays and we achieve atomic value which can no more be divided.

Merge Sort Division

Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.

Merge Sort Combine

In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

Merge Sort Combine

After the final merging, the list should look like this –

Merge Sort

**Algorithm of Merge Sort**

procedure mergesort( var a as array )

if ( n == 1 ) return a

var l1 as array = a[0] ... a[n/2]

var l2 as array = a[n/2+1] ... a[n]

l1 = mergesort( l1 )

l2 = mergesort( l2 )

return merge( l1, l2 )

end procedure

procedure merge( var a as array, var b as array )

var c as array

while ( a and b have elements )

if ( a[0] > b[0] )

add b[0] to the end of c

remove b[0] from b

else

add a[0] to the end of c

remove a[0] from a

end if

end while

while ( a has elements )

add a[0] to the end of c

remove a[0] from a

end while

while ( b has elements )

add b[0] to the end of c

remove b[0] from b

end while

return c

end procedure

**Complete Implementation of Merge Sort**

|  |
| --- |
| public void UseOfMergeSort()  {  int[] arr = { 2, 3, 1, 5, 6, 4 };  //int[] arr = { 2, 3, 1 };  int[] temp = new int[arr.Length];  int start = 0;  int end = arr.Length - 1;  Console.WriteLine("Before sorting:\n" + string.Join(", ", arr));  MergeSort(ref arr, ref temp, start, end);  Console.WriteLine("\nAfter sorting:\n" + string.Join(", ", arr));  Console.ReadLine();  }  private void MergeSort(ref int[] arr, ref int[] temp, int start, int end)  {  if (start < end)  {  int mid = (start + end) / 2;  MergeSort(ref arr, ref temp, start, mid);  MergeSort(ref arr, ref temp, mid + 1, end);  Merge(ref arr, ref temp, start, end, mid);  }  }  private void Merge(ref int[] arr, ref int[] temp, int leftStart, int rightEnd, int mid)  {  int start = leftStart;  int leftEnd = mid;  int rightStart = leftEnd + 1;  int size = rightEnd - leftStart + 1;    int tempIndex = leftStart;  while(leftStart <= leftEnd && rightStart <= rightEnd)  {  if(arr[leftStart] <= arr[rightStart])  {  temp[tempIndex] = arr[leftStart];  leftStart++;  }  else  {  temp[tempIndex] = arr[rightStart];  rightStart++;  }  tempIndex++;  }  while(leftStart <= leftEnd)  {  temp[tempIndex++] = arr[leftStart++];  }  while (rightStart <= rightEnd)  {  temp[tempIndex++] = arr[rightStart++];  }  //System.Array.Copy(arr, leftStart, temp, tempIndex, leftEnd - leftStart + 1);  //System.Array.Copy(arr, rightStart, temp, tempIndex, rightEnd - rightStart + 1);  System.Array.Copy(temp, start, arr, start, size);  } |

**Quick Sort**

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are of Ο(n2), where **n** is the number of items.

Quick sort use divide and conquer method to sort the array. In this method, we need to select a pivot item form the array and do the partition around the pivot item. We can choose the pivot item in either of the following way:

* We can pick first item as a pivot.
* We can pick the last item as a pivot.
* We can pick a median item as a pivot
* We can pick a random item as a pivot

**Algorithm of Quick Sort**

procedure quickSort(left, right)

if right-left <= 0

return

else

pivot = A[right]

partition = partitionFunc(left, right, pivot)

quickSort(left,partition-1)

quickSort(partition+1,right)

end if

end procedure

**Complete implementation of Quick Sort**

|  |
| --- |
| public void UseOfQuickSort()  {  int[] arr = { 35, 33, 42, 10, 14, 19, 27, 44, 26, 31 };  int start = 0;  int end = arr.Length - 1;  Console.WriteLine("Before sorting:\n" + string.Join(", ", arr));  QuickSort(ref arr, start, end);  Console.WriteLine("\nAfter sorting:\n" + string.Join(", ", arr));  Console.ReadLine();  }  private void QuickSort(ref int[] arr, int start, int end)  {  if (start < end)  {  int partitionIndex = Partition(ref arr, start, end);  QuickSort(ref arr, start, partitionIndex - 1);  QuickSort(ref arr, partitionIndex + 1, end);  }  }  private int Partition(ref int[] arr, int start, int end)  {  int pivot = arr[end];  int partitionIndex = start;  for (int i = start; i < end; i++)  {  if (arr[i] <= pivot)  {  Common.Instance.Swap<int>(ref arr[i], ref arr[partitionIndex]);  partitionIndex++;  }  }  Common.Instance.Swap<int>(ref arr[partitionIndex], ref arr[end]);  return partitionIndex;  } |

**Tree**

Tree represents the nodes connected by edges. Binary Tree is a special data structure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.



Following are the important terms with respect to tree.

* **Path** − Path refers to the sequence of nodes along the edges of a tree.
* **Root** − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
* **Parent** − Any node except the root node has one edge upward to a node called parent.
* **Child** − The node below a given node connected by its edge downward is called its child node.
* **Leaf** − The node which does not have any child node is called the leaf node.
* **Subtree** − Subtree represents the descendants of a node.
* **Visiting** − Visiting refers to checking the value of a node when control is on the node.
* **Traversing** − Traversing means passing through nodes in a specific order.
* **Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
* **keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.

## Tree Node

The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.

struct node

{

int data;

struct node \*leftChild;

struct node \*rightChild;

};

**Binary Search Tree**

Binary Search tree exhibits a special behaviour. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties –

* The left sub-tree of a node has a key less than or equal to its parent node's key.
* The right sub-tree of a node has a key greater than to its parent node's key.

Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −



We're going to implement tree using node object and connecting them through references.

## Binary Search Tree Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following −

* **Insert** − Inserts an element in a tree/create a tree.
* **Search** − Searches an element in a tree.

**Traversing on the Binary Search Tree**

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot random access a node in a tree. There are three ways which we use to traverse a tree –

* **Preorder Traversal** − Traverses a tree in a pre-order manner.
* **Inorder Traversal** − Traverses a tree in an in-order manner.
* **Postorder Traversal** − Traverses a tree in a post-order manner.

**In-order Traversal**

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of in-order traversal of this tree will be −

***D → B → E → A → F → C → G***

**Pre-order Traversal**

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be –

***A → B → D → E → C → F → G***

**Post-order Traversal**

In this traversal method, the root node is visited last. First, we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

***D → E → B → F → G → C → A***

**Complete implementation of Binary Search Tree**

#include <stdio.h>

#include <stdlib.h>

struct node {

int data;

struct node \*leftChild;

struct node \*rightChild;

};

struct node \*root = NULL;

void insert(int data) {

struct node \*tempNode = (struct node\*) malloc(sizeof(struct node));

struct node \*current;

struct node \*parent;

tempNode->data = data;

tempNode->leftChild = NULL;

tempNode->rightChild = NULL;

//if tree is empty

if(root == NULL) {

root = tempNode;

} else {

current = root;

parent = NULL;

while(1) {

parent = current;

//go to left of the tree

if(data < parent->data) {

current = current->leftChild;

//insert to the left

if(current == NULL) {

parent->leftChild = tempNode;

return;

}

} //go to right of the tree

else {

current = current->rightChild;

//insert to the right

if(current == NULL) {

parent->rightChild = tempNode;

return;

}

}

}

}

}

struct node\* search(int data) {

struct node \*current = root;

printf("Visiting elements: ");

while(current->data != data) {

if(current != NULL)

printf("%d ",current->data);

//go to left tree

if(current->data > data) {

current = current->leftChild;

}

//else go to right tree

else {

current = current->rightChild;

}

//not found

if(current == NULL) {

return NULL;

}

}

return current;

}

void **pre\_order\_traversal**(struct node\* root) {

if(root != NULL) {

printf("%d ",root->data);

pre\_order\_traversal(root->leftChild);

pre\_order\_traversal(root->rightChild);

}

}

void **inorder\_traversal**(struct node\* root) {

if(root != NULL) {

inorder\_traversal(root->leftChild);

printf("%d ",root->data);

inorder\_traversal(root->rightChild);

}

}

void **post\_order\_traversal**(struct node\* root) {

if(root != NULL) {

post\_order\_traversal(root->leftChild);

post\_order\_traversal(root->rightChild);

printf("%d ", root->data);

}

}

int main() {

int i;

int array[7] = { 27, 14, 35, 10, 19, 31, 42 };

for(i = 0; i < 7; i++)

insert(array[i]);

i = 31;

struct node \* temp = search(i);

if(temp != NULL) {

printf("[%d] Element found.", temp->data);

printf("\n");

}else {

printf("[ x ] Element not found (%d).\n", i);

}

i = 15;

temp = search(i);

if(temp != NULL) {

printf("[%d] Element found.", temp->data);

printf("\n");

}else {

printf("[ x ] Element not found (%d).\n", i);

}

printf("\nPreorder traversal: ");

pre\_order\_traversal(root);

printf("\nInorder traversal: ");

inorder\_traversal(root);

printf("\nPost order traversal: ");

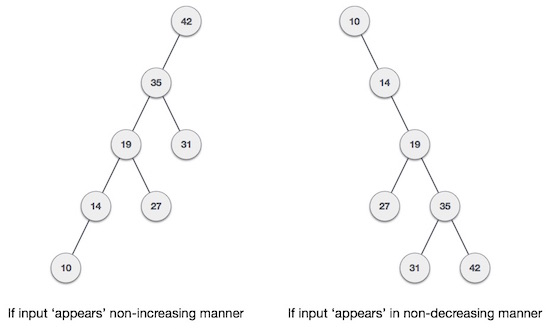
post\_order\_traversal(root);

return 0;

}

**Adelson**, **Velski** & **Landis (AVL) Tree or Balanced Tree**

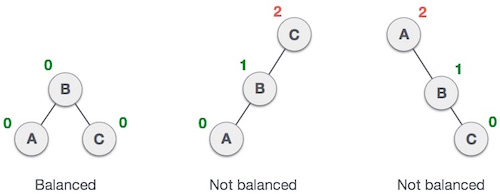
What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this –



it is observed that BST's worst-case performance is closest to linear search algorithms, that is Ο(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor **Adelson**, **Velski** & **Landis**, **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor**.

Here we see that the first tree is balanced and the next two trees are not balanced –



In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

***BalanceFactor*** = height(left-subtree) − height(right-subtree)

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

## AVL Rotations

To balance itself, an AVL tree may perform the following four kinds of rotations −

* Left rotation
* Right rotation
* Left-Right rotation
* Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

### Left Rotation

If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation –



In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A**the left-subtree of B.

## Right Rotation

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

### Left-Right Rotation

Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

.

|  |  |
| --- | --- |
| **State** | **Action** |
| Right Rotation | A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation. |
| Left Rotation | We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**. |
| Left Rotation | Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree. |
| Right Rotation | We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree. |
| Balanced Avl Tree | The tree is now balanced. |

**Right-Left Rotation**

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

|  |  |
| --- | --- |
| **State** | **Action** |
| Left Subtree of Right Subtree | A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2. |
| Subtree Right Rotation | First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**. |
| Right Unbalanced Tree | Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation. |
| Left Rotation | A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**. |
| Balanced AVL Tree | The tree is now balanced. |

**Graph**

**Depth First Search (DFS)**

Depth First Search (DFS) algorithm traverses a graph in a depth ward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

* **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
* **Rule 2** − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
* **Rule 3** − Repeat Rule 1 and Rule 2 until the stack is empty.

|  |  |  |
| --- | --- | --- |
| **Step** | **Traversal** | **Description** |
| 1 | Depth First Search Step One | Initialize the stack. |
| 2 | Depth First Search Step Two | Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order. |
| 3 | Depth First Search Step Three | Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S**and **D** are adjacent to **A** but we are concerned for unvisited nodes only. |
| 4 | Depth First Search Step Four | Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order. |
| 5 | Depth First Search Step Five | We choose **B**, mark it as visited and put onto the stack. Here **B**does not have any unvisited adjacent node. So, we pop **B**from the stack. |
| 6 | Depth First Search Step Six | We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack. |
| 7 | Depth First Search Step Seven | Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack. |

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX 5

struct Vertex {

char label;

bool visited;

};

//stack variables

int stack[MAX];

int top = -1;

//graph variables

//array of vertices

struct Vertex\* lstVertices[MAX];

//adjacency matrix

int adjMatrix[MAX][MAX];

//vertex count

int vertexCount = 0;

//stack functions

void push(int item) {

stack[++top] = item;

}

int pop() {

return stack[top--];

}

int peek() {

return stack[top];

}

bool isStackEmpty() {

return top == -1;

}

//graph functions

//add vertex to the vertex list

void addVertex(char label) {

struct Vertex\* vertex = (struct Vertex\*) malloc(sizeof(struct Vertex));

vertex->label = label;

vertex->visited = false;

lstVertices[vertexCount++] = vertex;

}

//add edge to edge array

void addEdge(int start,int end) {

adjMatrix[start][end] = 1;

adjMatrix[end][start] = 1;

}

//display the vertex

void displayVertex(int vertexIndex) {

printf("%c ",lstVertices[vertexIndex]->label);

}

//get the adjacent unvisited vertex

int getAdjUnvisitedVertex(int vertexIndex) {

int i;

for(i = 0; i < vertexCount; i++) {

if(adjMatrix[vertexIndex][i] == 1 && lstVertices[i]->visited == false) {

return i;

}

}

return -1;

}

void depthFirstSearch() {

int i;

//mark first node as visited

lstVertices[0]->visited = true;

//display the vertex

displayVertex(0);

//push vertex index in stack

push(0);

while(!isStackEmpty()) {

//get the unvisited vertex of vertex which is at top of the stack

int unvisitedVertex = getAdjUnvisitedVertex(peek());

//no adjacent vertex found

if(unvisitedVertex == -1) {

pop();

} else {

lstVertices[unvisitedVertex]->visited = true;

displayVertex(unvisitedVertex);

push(unvisitedVertex);

}

}

//stack is empty, search is complete, reset the visited flag

for(i = 0;i < vertexCount;i++) {

lstVertices[i]->visited = false;

}

}

int main() {

int i, j;

for(i = 0; i < MAX; i++) // set adjacency {

for(j = 0; j < MAX; j++) // matrix to 0

adjMatrix[i][j] = 0;

}

addVertex('S'); // 0

addVertex('A'); // 1

addVertex('B'); // 2

addVertex('C'); // 3

addVertex('D'); // 4

addEdge(0, 1); // S - A

addEdge(0, 2); // S - B

addEdge(0, 3); // S - C

addEdge(1, 4); // A - D

addEdge(2, 4); // B - D

addEdge(3, 4); // C - D

printf("Depth First Search: ")

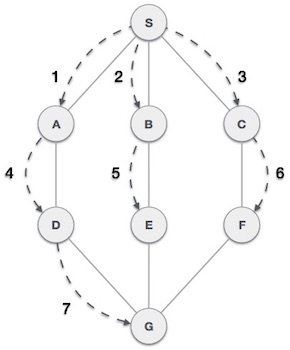
depthFirstSearch();

return 0;

}

**Breadth First Search (DFS)**

Breadth First Search (BFS) algorithm traverses a graph in a breadth ward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

* **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
* **Rule 2** − If no adjacent vertex is found, remove the first vertex from the queue.
* **Rule 3** − Repeat Rule 1 and Rule 2 until the queue is empty.

|  |  |  |
| --- | --- | --- |
| **Step** | **Traversal** | **Description** |
| 1 | Breadth First Search Step One | Initialize the queue. |
| 2 | Breadth First Search Step Two | We start from visiting **S** (starting node), and mark it as visited. |
| 3 | Breadth First Search Step Three | We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it. |
| 4 | Breadth First Search Step Four | Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it. |
| 5 | Breadth First Search Step Five | Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it. |
| 6 | Breadth First Search Step Six | Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**. |
| 7 | Breadth First Search Step Seven | From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it. |

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.