Data Structure in C

Unit 1

Introduction

Definition of Data Structure

Data may be organised in many different ways; the logical or mathematical model of a particular organization is called a data structure. The choice of a particular data model depends on two considerations. First, it must be rich enough in structure to mirror the actual relationships of the data in real world. On the other hand, the structure should be simple enough that one can effectively process the data when necessary.

Classification:

Data structures are generally classified into two types; Primitive and Non-Primitive.

Primitive: Integer, Real, Character, Boolean.

Non-Primitive:

Linear:

- Arrays
- Linked list
- Stacks
- Queues

Non-Linear:

- Trees
- Graphs

Data Structure operations

The data appearing in our Data Structures are processed by means of certain operations. In fact, the particular data structure that one chooses for a given situation depends largely on the frequency with which specific operations are performed. The following operations play a major role in the.

1. Traversing: Accessing each record exactly once so that certain items in the record may be processed (This accessing and processing is sometimes called "Visiting" the record).
2. Searching: Finding the location of the record with a given key value, or finding the locations of all records which satisfy one or more conditions.

3. Inserting: Adding new record to the structure.
4. Deleting: Removing a record from the structure.

Sometimes two or more of the operations may be used in a given situation; e.g., we may want to delete the record with a given key, which may mean we first need to search for the location of the record. The following two operations, which are used in special situations, will also be considered:

1. Sorting: Arranging the records in some logical order
2. Merging: Combining the records in two different sorted files into a single sorted file.

Algorithms: Complexity, Time-Space trade off

An Algorithm is a well-defined list of steps for solving a particular problem. The time and space it uses are the two major measures of the efficiency of an algorithm. The complexity of an algorithm is the function which gives the running time and/or space in terms of input size. Each of our algorithms will involve a particular data structure. Accordingly, we may not always be able to use the most efficient algorithm, since the choice of data structure depends on many things, including the type of data and frequency with which various data operations are applied. Sometimes the choice of data structure involves a time-space trade off: By increasing the amount of space for sorting the data, one may be able to reduce the time needed for processing the data, or vice versa.

Searching Algorithms:

Consider a membership file in which each record contains, among other data, the name and telephone number of its members. Suppose we are given the name of a member and we want to find his or her telephone number. One way to do this is to linearly search through the file, i.e., to apply the following.

Linear Search:

Search each record of the file, one at a time, until finding the given name and hence the corresponding telephone number.

First of all, it is clear that the time required to execute the algorithm is proportional to the number comparisons. Also, assuming that each name in the field is equally likely to be picked, it is intuitively clear that the average number of comparisons for a file with ***n*** records is equal to ***n/2;*** i.e., the complexity of the linear search algorithm is given by $C(n) = n/2$.

Binary Search:

Compare the given name with the name in the middle of the list; this tells which half of the list contains the name. Then compare Name with the name in the middle of the correct half to determine which quarter of the list contains name. Continue the process until finding Name in the list. One can show the complexity of the binary search is given by Thus, for example, one will not require not more than 15 comparisons to find a given name in a list containing 25,000 names. Although this algorithm is very efficient it has some drawbacks.

Specifically, the algorithm assumes that one has direct access to the middle name in the list or a sub-list. This means that the list must be stored in some type of array. Unfortunately, inserting an element in an array requires elements to be moved up the list.

Time space trade-off:
Suppose a file of records contains names, social security number and much additional information among its fields. Sorting the file alphabetically and running a binary search is a very efficient way to find the record for a given name. On the other hand, suppose we are given only the social security number of the person. Then we would have to do a linear search for the record, which is extremely time-consuming for a very large number of records. One way to solve this is to have another file which is sorted numerically according to social number. This however, would double the space required for sorting the data. Another way is to have the main file sorted numerically by social security number and to have an auxiliary array with only two columns; list of names and pointer which give the locations of the corresponding records in the main file. Thus, the trade of space for minimal time.

Complexity of Algorithm:

The analysis of algorithm is a major task in computer science. In order to compare algorithms, we must have some criteria to measure the efficiency of algorithms. Suppose **M** is an algorithm, and suppose **N** is the size of the input data. The time and space used by the algorithm M are two main measures for the efficiency of **M**. The time is measured by counting the number of key operations-in sorting and searching algorithms. For example, the number of comparisons. That is because key operations are so defined that the time for the other operations is much less than or at most proportional to the time for the key operation. The space is measured by counting the maximum of memory needed by the algorithm.

The above discussion leads us to the question of finding the complexity function $f(n)$for certain cases. The two cases one usually investigates in complexity theory are as follows:

1. ***Worst Case:*** The maximum of $f(n)$ for any positive input.
2. ***Average Case:*** The expected value of $f(n)$

Sometimes we also consider the minimum possible value of $f(n)$, called the "Best Case". The analysis of average case assumes certain probabilistic distribution for the input data; one such assumption might be that all possible permutations of an input data set are equally likely. The average case also uses the following concept in probability theory.

1. Linear search: $0(n)$
2. Binary Search: $0(log\ n)$
3. Bubble sort: $0(n^2)$
4. Merge-sort: $0(n\ log\ n)$

Asymptotic Notation for complexity of algorithms:

The "big O" notation defines an upper bound function $g(n) for\ f(n)$ which represents the time/space complexity of the algorithm on an input characteristic n. There are other asymptotic notations such as Ω, θ, o which serve to provide bounds for the function $f(n)$.

Omega Notation (Ω):

It is used when the function $g(n)$ defines lower bound for the function $f(n)$.

Definition:

F(n) = $\Omega\big(\mathrm{g(n)}\big)$, if and only if(iff) there exist a positive integer n_0 and a positive number M such that $|f(n)| \geq |M|g(n)|$ for all $n \geq n_0$.

For $f(n) = 18 + 9, f(n) > 18n$ for all n, hence $f(n) = \Omega(n)$. Also, for $f(n) = 90n^2 + 18n + 6$, for $f(n) > 90n^2$ for $n \geq 0$ therefore, $f(n) = \Omega(n^2)$.

For $f(n) = \Omega\big(g(n)\big), g(n)$ is a lower bound function and there may be several such functions, but it is appropriate that the function which is almost as large of n as possible such that the definition Ω is satisfied, is chosen as $g(n)$. Thus, for example, $f(n) = 5n + 1$ leads to both $f(n) = \Omega\ and\ f(n) = \Omega(1)$. However, we never consider the latter to be correct, since $f(n) = \Omega(n)$ represents the largest possible function of n satisfying the definition of Ω and hence, is more informative.

Theta Notation (Θ):

It is used when the function $g(n)$ is bounded both from above and below by the function $f(n)$.

Definition:

F(n) = $\Theta\big(\mathrm{g(n)}\big)$, if and only if(iff) there exist two positive constants c_1 and c_2 and a positive integer n such that $|c1|\ g(n)| \leq c2|g(n)|\ \forall\ n \geq n_0$.

From the definition it implies that the function $g(n)$ are both an upper bound and a lower bound for the function $f(n)$ for all values of $n, n \geq n_0$. In other words, $f(n)$ is such that, $f(n) = O\big(g(n)\big)$ and $f(n) = \Omega\big(g(n)\big)$.

For $f(n) = 18n + 9$, since $f(n) > 18n$ and $f(n) \leq 27n$ for $n \geq 1$, we have $f(n) = \Omega(n)$ and $f(n) = O(n)$ respectively, for $n \geq 1$. Hence, $f(n) = \Theta(n)$. Again. $16n^2 + 30n - 90 = \Theta(n^2)$ and $7.2^n + 30n = \Theta(n^2)$.

Little Oh Notation(o)

Definition: $f(n) = o\big(g(n)\big)$ iff $f(n) = O\big(g(n)\big)\ and\ f(n) \neq \Omega\big(g(n)\big)$.

For $f(n) = 18n + 9$, we have $f(n) = O(n^2)$ but $f(n) \neq \Omega(n^2)$. Hence, $f(n) = o(n^2)$. However, $f(n) \neq o(n)$.

Sub Algorithms(Functions format):

A sub algorithm is a complete and independently defined algorithmic module which is used (or invoked or called) by some main algorithm or by some other sub algorithms. A sub algorithm receives value, called *arguments,* from an originating (calling) algorithm; performs computation; and then sends back the result to the calling algorithm. The sub algorithm is defined independently so that it may be called by many different algorithms or called at different times in the same algorithm.

The relationship between an algorithm and a sub-algorithm is similar to the relationship between a main program and a subprogram of a programming language.

```
#include <stdio.h>


float MEAN(int, int, int);

Int main()

{

        int A,B,C;

        printf("Enter the values of A, B, C: ");

        scanf("%d %d %d", &A, &B, &C);


        printf("The average of %d, %d and %d is: %.2f", A, B, C, MEAN(A, B, C));


}

float MEAN(int a1, int b2, int c3)
```

```
{
        Float AVG;
        AVG = (a1, b2, c3)/3;
        return(AVG);
}
```

Variables, Data types

Each variable in any of our algorithms or programs has a datatype which determines the code that is used for storing its value. Four such types follows:

1. Character: Here data are coded using some character code as EBCDIC or ASCII.
2. Real (or floating point): Here, numeric data are coded using the exponential form of data.
3. Integer (or fixed point): Here positive integers are coded using binary representation, and negative integers by some binary variations usch as 2's compliment.
4. Logical: Here, the variable can have only the value "True or False"; hence, it may be coded using only one bit, 1 for true and 0 for false.

Unit 2

Arrays

Introduction:

Data structures are classified as either linear or nonlinear. A data structure is said to be linear if its elements form a sequence, or, in other words, a linear list. There are two basic ways of representing such linear structures in memory. One way is to have the linear relationship between the elements represented by means of sequential memory locations. These linear structures are called: arrays. The other way is to have the linear relationship between the elements represented by means of pointers or links. These linear structures are called linked lists.

The operations one normally performs on any linear structure, whether it be an array or a linked list, include the following:

(a) **Traversal.** Processing each element in the list.

(b) **Search.** Finding the location of the element with a given value or the record with a given key.

(c) **Insertion.** Adding a new element to the list.

(d)**Deletion**. Removing an element from the list. (e} Sorting. Arranging the elements in some type of order.

(f) **Merging**. Combining two lists into a single list. The particular linear structure that one chooses for a given situation depends on the relative frequency with which one performs these different operations on the structure. This chapter discusses a very common linear structure called an array. Since arrays are usually easy to traverse, search and sort, they are frequently used to store relatively permanent collections of data. On the other hand, if the size of the structure and the data in the structure are constantly changing, then the array may not be as useful a structure as the linked list.

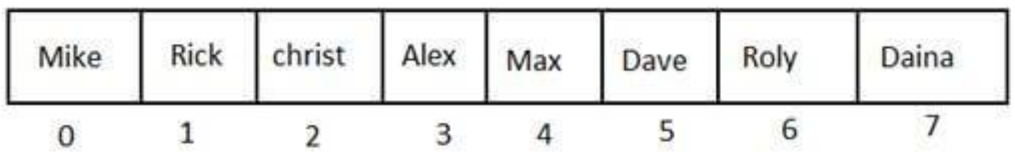
The particular linear structure that one chooses for a given situation depends on the relative frequency with which one performs these different operations on the structure.

Linear Array:

A linear array, is a list of finite numbers of elements stored in the memory. In a linear array, we can store only homogeneous data elements. Elements of the array form a sequence or linear list, that can have the same type of data.

Each element of the array, is referred by an index set. And, the total number of elements in the array list, is the length of an array. We can access these elements with the help of the index set.

For example, consider an array of employee names with the index set from 0 to 7, which contains 8 elements as shown in fig:



Now, if we want to retrieve the name 'Alex' from the given array, we can do so by calling "employees [3]". It gives the element stored in that location, that is 'Alex', and so on.

We can calculate the length, or a total number of elements, of a linear array (LA), by the given formula:

Length (LA)=UB-LB+1

Here, UB refers to the upper bound, the largest index set of the given array list. LB refers to the lower bound, smallest index set of the given array list.

For example, we have an array of employees, in which lower bound is 153 and the upper bound is 234, so we can calculate the total number of elements in the list, by giving the formula.

234-153+1=82

We can perform various operations on a linear array:

- **Traversing-** processing each element of the array list.
- **Inserting-** adding new elements in the array list.
- **Deleting-** removing an element from the array list.
- **Sorting-** arranging the elements of the list in some sorting order.
- **Merging-** combining the elements of two array lists in a single array list.

Representation of Linear arrays in memory:

Let LA be a linear array in the memory of the computer. Recall that the memory of the computer is simply a sequence of addressed locations.

LOC(LA[K]) = address of the clement LAIK] of the array LA

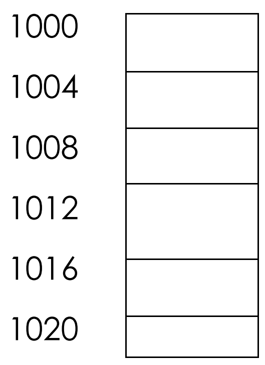
As previously noted, the elements of LA are stored in successive memory cells. Accordingly, the computer does not need to keep track of the address of every element of LA, but needs to keep track only of the address of the first element of LA, denoted by

*base (*LA*)*

and called the base address of LA. Using this address Base (LA), the computer calculates the base address of any clement of LA oy the following formula:

LOC(LAJ[K]) = Base(LA) + WK - lower bound)

where w is the whilst of words per memory cell for the array LA. Observe that the time to calculate LOC(LA[K}) is essentially the same for any value of K. Furthermore, given any subscript K, one can locate and access the content of LA(K] without scanning any other element of LA.



Address Calculation using Row and Column Major ordering

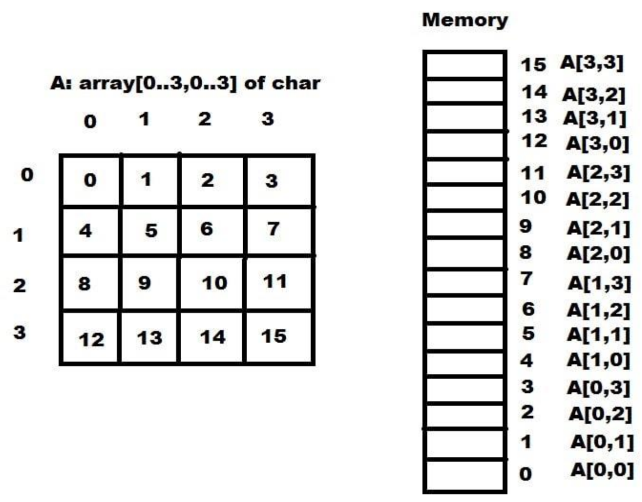
Row Major Order Formula

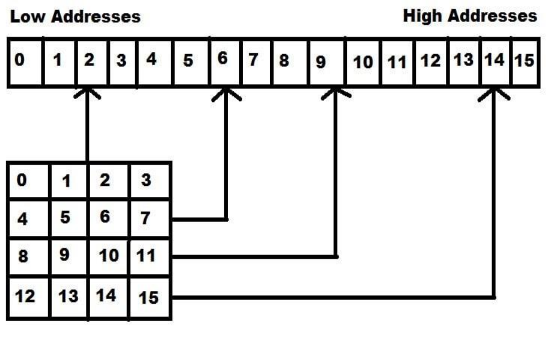
Row Major Ordering in an array:

Row major ordering assigns successive elements, moving across the rows and then down the columns, to successive memory locations.

In simple language, if the elements of an array are being stored in Row-Wise fashion. This mapping is demonstrated in the below figure:

The formula to compute the Address (offset) for a two-dimension row-major ordered array as:





Address of A[I][J] = Base Address + W * (C * I + j)

Where Base Address is the address of the first element in an array.

W= is the weight (size) of a data type.

C= is Total No of Columns.

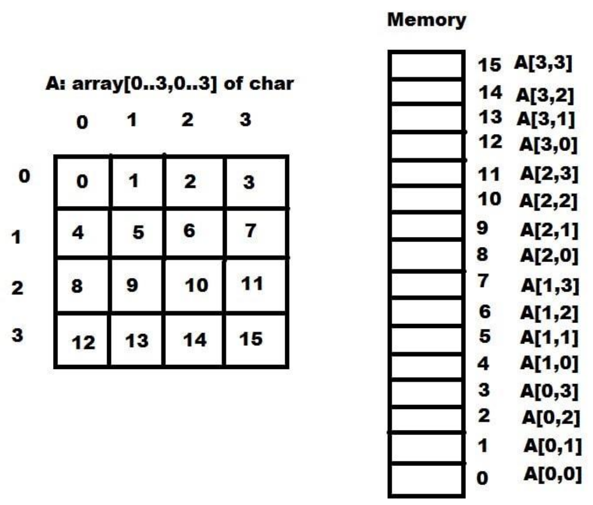
I= is the Row Number

J= is Column Number of an element whose address is to find out.

Column Major Ordering

If the element of an array is being stored in Column Wise fashion then it is called column-major ordering in an array. In row-major ordering, the rightmost index increased the fastest as you moved through consecutive memory locations. In column-major ordering, the leftmost index increases the fastest.

Pictorially, a column-major ordered array is organized as shown below



The formulae for computing the address of an array element when using column-major ordering is very similar to that for row-major ordering. You simply reverse the indexes and sizes in the computation:

For a two-dimensional column-major array:

Address of A[I][J] = Base Address + W * (R * J + I)

Where Base Address is the address of the first element in an array.

W= is the weight (size) of a data type.

R= is Total No of Rows.

I= is the Row Number

J= is Column Number of an element whose address is to find out.

Address Calculation in single-dimensional Array: In 1-D array, there is no Row Major and no Column major concept, all the elements are stored in contiguous memory locations. We can calculate the address of the 1-D array by using the following formula:

Address of A[I] = Base Address + W * I.

Where I[is a location(Indexing) of element] whose address is to be found out. W (is the size of data type).

Traversing, Insertion and Deletion

1. **Traverse** – print all the array elements one by one

```c
#include <stdio.h>
int main() {
  int LA[] = {1,3,5,7,8};
  int item = 10, k = 3, n = 5;
  int i = 0, j = n;
  printf("The original array elements are :\n");
  for(i = 0; i<n; i++) {
    printf("LA[%d] = %d \n", i, LA[i]);
  }
}
```

Output:

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
```

2. **Insert:**
   (Adds element at the given index)

```
#include <stdio.h>
int main() {
  int LA[] = {1,3,5,7,8};
  int item = 10, k = 3, n = 5;
  int i = 0, j = n;

  printf("The original array elements are :\n");
  for(i = 0; i<n; i++) {
    printf("LA[%d] = %d \n", i, LA[i]);
  }

  n = n + 1;

    while( j >= k) {
       LA[j+1] = LA[j];
       j = j - 1;
    }

    LA[k] = item;

    printf("The array elements after insertion :\n");

    for(i = 0; i<n; i++) {
       printf("LA[%d] = %d \n", i, LA[i]);
    }
}
```

Output:

```
The original array elements are:

LA[0] = 1

LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after insertion:
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 10
LA[4] = 7
LA[5] = 8
```

3. Delete

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

```
#include <stdio.h>

void main() {
   int LA[] = {1,3,5,7,8};
   int k = 3, n = 5;
   int i, j;

   printf("The original array elements are :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
   j = k;
   while( j < n) {
      LA[j-1] = LA[j];
      j = j + 1;
   }

   n = n -1;
   printf("The array elements after deletion :\n");

   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after deletion :
LA[0] = 1
LA[1] = 3
LA[2] = 7
LA[3] = 8
```

MULTIDIMENSIONAL ARRAY

Representation of 2-Dimensional array in a memory

2D array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns.

However, 2D arrays are created to implement a relational database look alike data structure. It provides ease of holding bulk of data at once which can be passed to any number of functions wherever required.

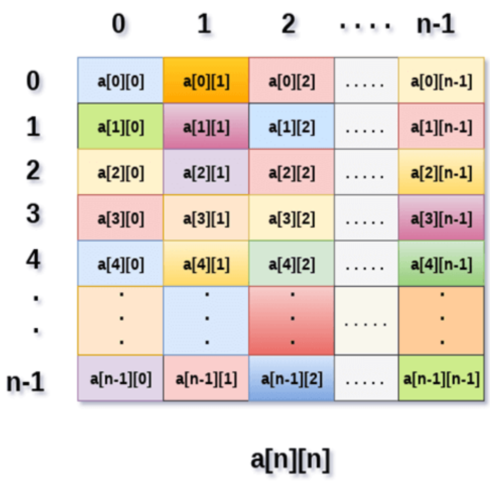
How to declare 2D Array

The syntax of declaring 2 dimensional array is very much similar to that of a one dimensional array, given as follows.

```
int arr[max_rows][max_columns];
```

however, it produces the data structure which looks like following.

Above image shows the two dimensional array, the elements are organized in the form of rows and columns. First element of the first row is represented by a[0][0]



where the number shown in the first index is the number of that row while the number shown in the second index is the number of the column.

Pointers

A pointer in Data Structure is nothing more than a variable that holds addresses of some other variable when pointed to.

This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture. However, in 32-bit architecture the size of a pointer is 2-byte.

Pointers Arrays:

Assuming you have some understanding of pointers in C, let us start: An array name is a constant pointer to the first element of the array. Therefore, in the declaration –

```
double balance[50];
```

balance is a pointer to &balance[0], which is the address of the first element of the array balance. Thus, the following program fragment assigns p as the address of the first element of balance –

```
double *p;
double balance[10];

p = balance;
```

It is legal to use array names as constant pointers, and vice versa. Therefore, *(balance + 4) is a legitimate way of accessing the data at balance[4].

Once you store the address of the first element in 'p', you can access the array elements using *p, *(p+1), *(p+2) and so on. Given below is the example to show all the concepts discussed above –

```c
#include <stdio.h>

int main () {

   /* an array with 5 elements */
   double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
   double *p;
   int i;

   p = balance;

   /* output each array element's value */
   printf( "Array values using pointer\n");

   for ( i = 0; i < 5; i++ ) {
      printf("*(p + %d) : %f\n",  i, *(p + i) );
   }

   printf( "Array values using balance as address\n");

   for ( i = 0; i < 5; i++ ) {
      printf("*(balance + %d) : %f\n",  i, *(balance + i) );
   }

   return 0;
}
```

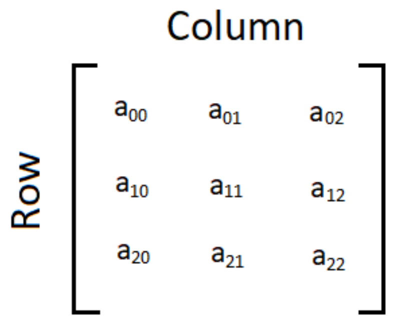
When the above code is compiled and executed, it produces the following result −

```
Array values using pointer
*(p + 0): 1000.000000
*(p + 1): 2.000000
*(p + 2): 3.400000
*(p + 3): 17.000000
*(p + 4): 50.000000
Array values using balance as address
*(balance + 0): 1000.000000
*(balance + 1): 2.000000
*(balance + 2): 3.400000
*(balance + 3): 17.000000
*(balance + 4): 50.000000
```

In the above example, p is a pointer to double, which means it can store the address of a variable of double type. Once we have the address in p, *p will give us the value available at the address stored in p, as we have shown in the above example.

Matrices

Matrix is a two-dimensional array. And to represent the two-dimensional array there should be two loops, where outer loops represent rows of the matrix and the inner loop represents the column of the matrix. Now, let us start with the addition of two matrices.


Column
Row
a_{00} a_{01} a_{02}
a_{10} a_{11} a_{12}
a_{20} a_{21} a_{22}

In this program, we will perform all operations through the 3×3 matrix, and the input for the matrix is given explicitly but you can ask for the input from the end-user.

```
#include <stdio.h>
#define N 4

// This function adds A[][] and B[][], and stores
// the result in C[][]
void add(int A[][N], int B[][N], int C[][N])
{
    int i, j;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    int A[N][N] = { {1, 1, 1, 1},
                    {2, 2, 2, 2},
                    {3, 3, 3, 3},
                    {4, 4, 4, 4}};

    int B[N][N] = { {1, 1, 1, 1},
                    {2, 2, 2, 2},
                    {3, 3, 3, 3},
                    {4, 4, 4, 4}};

    int C[N][N]; // To store result
    int i, j;
    add(A, B, C);

    printf("Result matrix is \n");
    for (i = 0; i < N; i++)
    {
        for (j = 0; j < N; j++)
           printf("%d ", C[i][j]);
        printf("\n");
    }

    return 0;
```

Matrix Subtraction:

```
#include <stdio.h>
#define N 4

// This function adds A[][] and B[][], and stores
// the result in C[][]
void sub(int A[][N], int B[][N], int C[][N])
{
    int i, j;
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            C[i][j] = A[i][j] - B[i][j];
}

int main()
{
    int A[N][N] = { {1, 1, 1, 1},
                    {2, 2, 2, 2},
                    {3, 3, 3, 3},
                    {4, 4, 4, 4}};

    int B[N][N] = { {1, 1, 1, 1},
                    {2, 2, 2, 2},
                    {3, 3, 3, 3},
                    {4, 4, 4, 4}};

    int C[N][N]; // To store result
    int i, j;
    add(A, B, C);

    printf("Result matrix is \n");
    for (i = 0; i < N; i++)
    {
        for (j = 0; j < N; j++)
           printf("%d ", C[i][j]);
        printf("\n");
    }

    return 0;
```

Matrix Multiplication

```
#include<stdio.h>

int main(void)
{
  int c, d, p, q, m, n, k, tot = 0;
  int fst[10][10], sec[10][10], mul[10][10];
  printf(" Please insert the number of rows and columns for first
matrix \n ");
  scanf("%d%d", &m, &n);

  printf(" Insert your matrix elements : \n ");
  for (c = 0; c < m; c++)
    for (d = 0; d < n; d++)
      scanf("%d", &fst[c][d]);

  printf(" Please insert the number of rows and columns for second
matrix\n");
  scanf(" %d %d", &p, &q);

  if (n != p)
    printf(" Your given matrices cannot be multiplied with each other.
\n ");
  else
  {
    printf(" Insert your elements for second matrix \n ");

    for (c = 0; c < p; c++)
      for (d = 0; d < q; d++)
        scanf("%d", &sec[c][d] );

    for (c = 0; c < m; c++) {
      for (d = 0; d < q; d++) {
        for (k = 0; k < p; k++) {
          tot = tot + fst[c][k] * sec[k][d];
        }
        mul[c][d] = tot;
        tot = 0;
      }
    }

    printf(" The result of matrix multiplication or product of the
matrices is: \n ");
    for (c = 0; c < m; c++) {
      for (d = 0; d < q; d++)
        printf("%d \t", mul[c][d] );
      printf(" \n ");
    }
  }
  return 0;
}
```

Sparse Matrix:

Sparse Matrix is a matrix that contains a few non-zero elements. Almost all the places are filled with zero. Matrix of m*n dimension refers to a 2-D array with m number of rows and n number of columns. And if the non-zero elements in the matrix are more than zero elements in the matrix then it is called a sparse matrix. And in case the size of the matrix is big, a lot of space is wasted to represent such a small number of non-zero elements. Similarly for scanning the same non-zero will take more time.

$$\begin{pmatrix} 5 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 6 & 0 & 0 \end{pmatrix}$$

Let us illustrate the concept of a sparse matrix with an example. Consider below 2-D matrix with 5 rows and 8 columns.

$$\begin{bmatrix} 0 & 9 & 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 6 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 5 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 6 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

As we can see only 8 elements out of 8 * 5 = 40 elements in the matrix are non-zero. That depicts that we just need to store these 8 elements to store in the memory. Since the rest are zero elements thus can be ignored.

There are 2 ways to represent such elements in the memory:

I. Array representation:

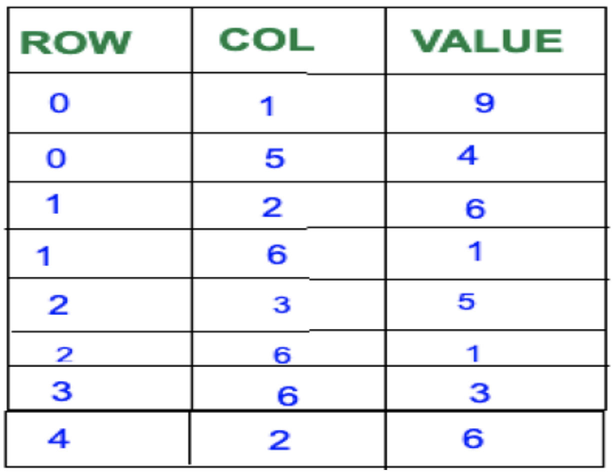
Step 1: List the non-zero elements in the matrix with their row and column index.

Step 2: Create a new array with the following dimensions:

Rows = number of non-zero elements in the matrix.

Columns = 3(Row, Column and value)

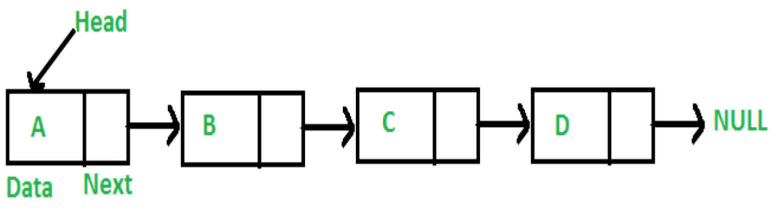
Step 3: Fill the array with the non-zero elements.



ROW	COL	VALUE
0	1	9
0	5	4
1	2	6
1	6	1
2	3	5
2	6	1
3	6	3
4	2	6

Linked List

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



Representation of Linked list in memory

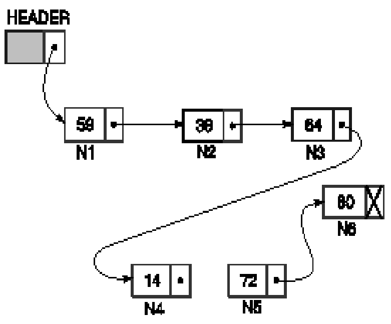
There are two ways to represent a linked list in memory:

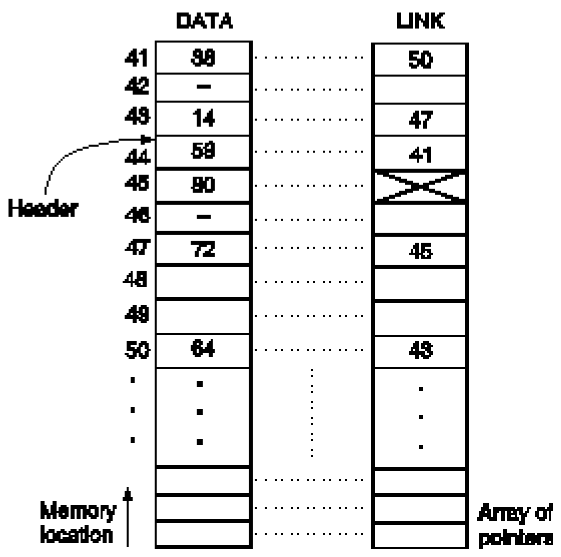
1. Static representation using array

2. Dynamic representation using free pool of storage

Static representation

In static representation of a single linked list, two arrays are maintained: one array for data and the other for links. The static representation of the linked list in Figure 5.2 is shown in Figure 5.3.





Two parallel arrays of equal size are allocated which should be sufficient to store the entire linked list. Nevertheless, this contradicts the idea of the linked list (that is non-contagious location of elements). But in some programming languages, for example, ALGOL, FORTRAN, BASIC, etc. such a representation is the only representation to manage a linked list.

Dynamic representation

The efficient way of representing a linked list is using the free pool of storage. In this method, there is a memory bank (which is nothing but a collection of free memory spaces) and a memory manager (a program, in fact). During the creation of a linked list, whenever a node is required, the request is placed to the memory manager; the memory manager will then search the memory bank for the block requested and, if found, grants the desired block to the caller. Again, there is also another program called the garbage collector; it plays whenever a node is no more in use; it returns the unused node to the memory bank. It may be noted that memory bank is basically a list of memory spaces which is available to a programmer. Such a memory management is known as dynamic memory management. The dynamic representation of linked list uses the dynamic memory management policy.

check

Traversing a lined list

Create a temporary variable for traversing. Assign reference of head node to it, say temp = head.

Repeat below step till temp != NULL.

temp->data contains the current node data. You can print it or can perform some calculation on it.

Once done, move to next node using temp = temp->next;.

Go back to 2nd step.

```c
#include <stdio.h>
#include <stdlib.h>

/* Structure of a node */
struct node {
    int data;          // Data
    struct node *next; // Address
}*head;


/*
 * Functions to create and display list
 */
void createList(int n);
void traverseList();


int main()
{
    int n;

    printf("Enter the total number of nodes: ");
    scanf("%d", &n);

    createList(n);

    printf("\nData in the list \n");
    traverseList();

    return 0;
}

/*
 * Create a list of n nodes
 */
void createList(int n)
{
    struct node *newNode, *temp;
    int data, i;

    head = (struct node *)malloc(sizeof(struct node));

    // Terminate if memory not allocated
    if(head == NULL)
    {
        printf("Unable to allocate memory.");
        exit(0);
    }


    // Input data of node from the user
    printf("Enter the data of node 1: ");
    scanf("%d", &data);

    head->data = data; // Link data field with data
    head->next = NULL; // Link address field to NULL


    // Create n - 1 nodes and add to list
    temp = head;
    for(i=2; i<=n; i++)
    {
        newNode = (struct node *)malloc(sizeof(struct node));

        /* If memory is not allocated for newNode */
        if(newNode == NULL)
        {
            printf("Unable to allocate memory.");
            break;
        }

        printf("Enter the data of node %d: ", i);
        scanf("%d", &data);

        newNode->data = data; // Link data field of newNode
        newNode->next = NULL; // Make sure new node points to NULL

        temp->next = newNode; // Link previous node with newNode
        temp = temp->next;    // Make current node as previous node
    }
}


/*
 * Display entire list
 */
void traverseList()
{
    struct node *temp;

    // Return if list is empty
    if(head == NULL)
    {
        printf("List is empty.");
        return;
    }

    temp = head;
    while(temp != NULL)
    {
        printf("Data = %d\n", temp->data); // Print data of current node
        temp = temp->next;                 // Move to next node
    }
}
```

Searching:

```c
#include<stdio.h>
#include<stdlib.h>
void create(int);
void search();
struct node
{
    int data;
    struct node *next;
};
struct node *head;
void main ()
{
    int choice,item,loc;
    do
    {
        printf("\n1.Create\n2.Search\n3.Exit\n4.Enter your choice?");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            printf("\nEnter the item\n");
            scanf("%d",&item);
            create(item);
            break;
            case 2:
            search();
            case 3:
            exit(0);
            break;
            default:
            printf("\nPlease enter valid choice\n");
        }

    }while(choice != 3);
}
    void create(int item)
    {
        struct node *ptr = (struct node *)malloc(sizeof(struct node *));
        if(ptr == NULL)
        {
            printf("\nOVERFLOW\n");
        }
        else
        {
            ptr->data = item;
            ptr->next = head;
            head = ptr;
            printf("\nNode inserted\n");
        }

    }
void search()
{
    struct node *ptr;
    int item,i=0,flag;
    ptr = head;
    if(ptr == NULL)
    {
        printf("\nEmpty List\n");
    }
    else
    {
        printf("\nEnter item which you want to search?\n");
        scanf("%d",&item);
        while (ptr!=NULL)
        {
            if(ptr->data == item)
            {
                printf("item found at location %d ",i+1);
                flag=0;
            }
            else
            {
                flag=1;
            }
            i++;
            ptr = ptr -> next;
        }
        if(flag==1)
        {
            printf("Item not found\n");
        }
    }

}
```

Memory Allocation

Garbage Collection

In programming, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language

A major catch of garbage collection is the total CPU cycles. They are involved to find the unused memory and delete it. The user perceives which pointer memory is to be released and not in use. We will not know the time when the destructor is called nor when it is deleted.

Some benefits are that automatic deallocation permits a programmer not to worry about memory management. It rises the write ability of a system. Hence drop development time and costs. Garbage Collection promotes reliability.

Overflow and Underflow

When a memory in an array or a list with a fixed memory is exceeded, there occurs memory overflow, and if it's < 0, there occurs an Underflow.

Linked List Insertion

```
struct Node
{
  int data;
  struct Node *next;
};

Int main()
{
        int inp, I, data;
        printf("Enter data: ");
        scanf("%d", &data);
        struct Node *head = malloc(sizeof(struct Node));
        head->data = data;
        head->next = NULL;
        printf("Entered          Element:          %d",          head->data);
}
```

Deletion:

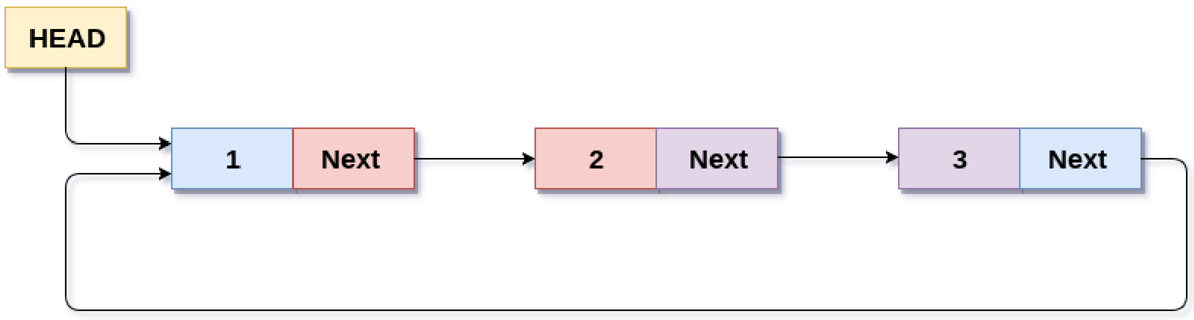
```
struct Node
{
  int data;
  struct Node *next;
};

Int main()
{
        int inp, I, data;
        printf("Enter data: ");
        scanf("%d", &data);
        struct Node *head = malloc(sizeof(struct Node));
        head->data = data;
        head->next = NULL;
        printf("Entered Element: %d", head->data);

        //deletion at the beginning
        struct node *temp = malloc(sizeof(struct node));
        temp = head;
        head = head->link;
        free(temp);
        temp                         =                         NULL;
}
```

Circular Linked List:

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.



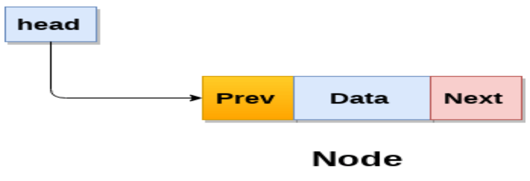
Circular Singly Linked List

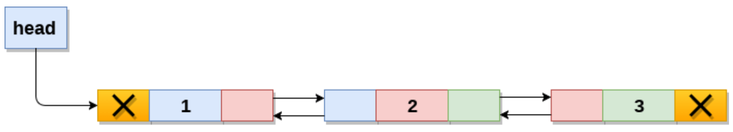
Circular linked lists are mostly used in task maintenance in operating systems. There are many examples where circular linked lists are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

Check (also check applicatons)

Doubly Linked list

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer), pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.



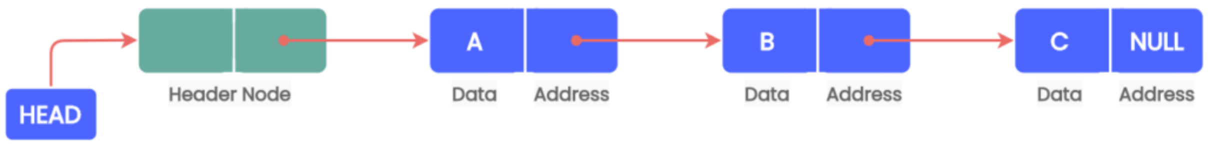


Doubly Linked List

check

Header Linked List

A header linked list is a type of linked list that has a header node at the beginning of the list. In a header linked list, HEAD points to the header node instead of the first node of the list.



The header node does not represent an item in the linked list. This data part of this node is generally used to hold any global information about the entire linked list. The next part of the header node points to the first node in the list.

A header linked list can be divided into two types:

Grounded header linked list that stores NULL in the last node's next field.

Circular header linked list that stores the address of the header node in the next part of the last node of the list.

Stacks and Queues

Definition

Stack is an abstract data type with a bounded(predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the top of the stack and the only element that can be removed is the element that is at the top of the stack, just like a pile of objects.

Array Representation of Stacks:

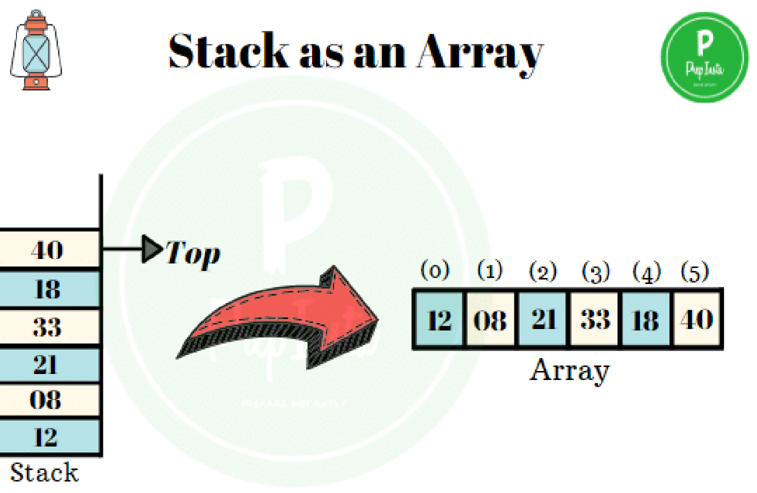
A stack is a data structure that can be represented as an array. Let us learn more about Array representation of a stack –

An array is used to store an ordered list of elements. Using an array for representation of stack is one of the easy techniques to manage the data. But there is a major difference between an array and a stack.

Size of an array is fixed.

While, in a stack, there is no fixed size since the size of stack changed with the number of elements inserted or deleted to and from it.

Despite the difference, an array can be used to represent a stack by taking an array of maximum size; big enough to manage a stack.



For more: Visit here

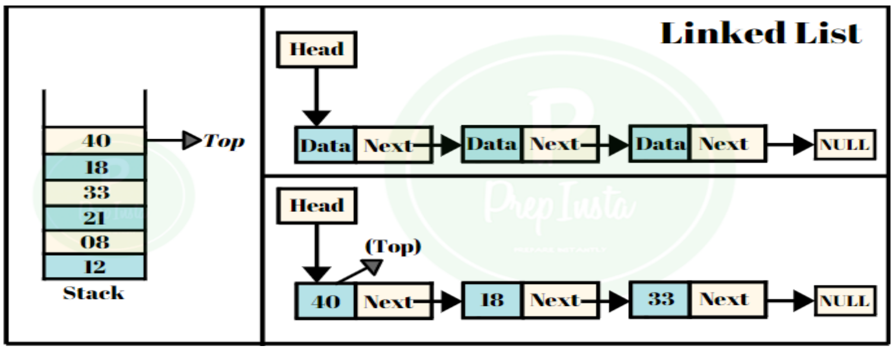
Linked Representation of stacks

This

First, we have to understand what a linked list is, to understand linked list representation of a stack. A Linked List is a Data Structure which consists of two parts:

- The data/value part.
- The pointer which gives the location of the next node in the list.

Stack can also be represented by using a linked list. We know that in the case of arrays we face a limitation i.e., array is a data structure of limited size. Hence, before using an array to represent a stack we will have to consider an enough amount of space to suffice the memory required by the stack.



However, a Linked List is a much more flexible data structure. Both the push and pop operations can be performed on either ends. But We prefer performing the Push and pop operation at the beginning.

The Stack operations occur in constant time. But if we perform the push and pop operations at the end of the linked list it takes time O(n).

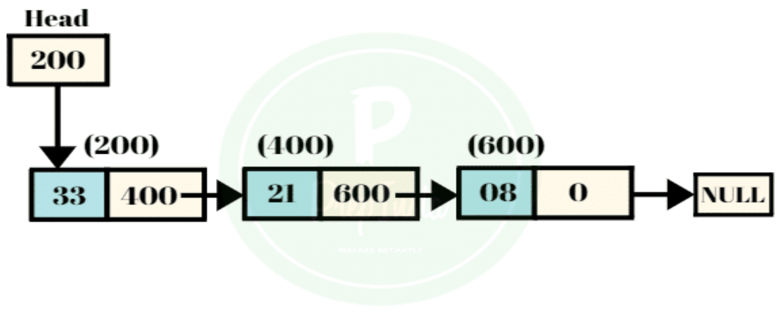
But performing the operations at the beginning occurs in constant time. Let us understand this with the help of an example.

Let us consider a linked list as shown here.

In the given data we can see that we have-

Head = 200.

Top = 33.



Polish Notation:

Polish Notation | Data structures and algorithms

In polish notation, the operator is placed before the operands. it is also known as prefix notation. generally, we use an operator between the two operands like x + y but in polish notation, we use the operators before the operands like +xy.

this notation is given by a mathematician Jan Lukasiewicz in 1924.

Polish Notation
Infix
Prefix
Postfix

Types of Notations.

In general, we have three types of notation; infix, postfix, and prefix.

Infix: Any Normal Expression.

Eg:- A+B+C*(B+C)

Prefix: The operators of an expression shifted to the left based on its precedence (BODMAS)

Following the precedence rule, B+C inside the bracket is added then multiplied by C, thus, *C+BC. Then A+B is added, and Lastly +C (Assuming C holds the Value of C*(B+C).

Therefore, ++AB*C+BC. Or Some may assume +A+B*C+BC.

Postfix: Postfix if the ultimate opposite of prefix, here the operands are shifted to right.

Such that, A+B+C*(B+C) becomes: AB+C*BC++.

The last thing we calculate is usually at the end. For instance, A+B = D is added, B+C = E is added to bring it out of the bracket, then it is multiplied with C. Thus, C*E = F, D **+** F = final result.

Similarly, AB+C*BC++.

Infix of Postfix expression:

Let's assume the following expressions:

a) AB+C*D-

b) AB*C+DB+-

a solution:

AB+C*D-

Here, Following BODMAS rule

B*C

A+B*C

A+B*C-D

Therefore, A+B*C-D.

B solution:

AB*C+DB+-

Here, Following BODMAS rule

A*B

A*B+C

D+B

A*B+C-D+B

Therefore, A*B+C-D+B.

Evaluation of Postfix:

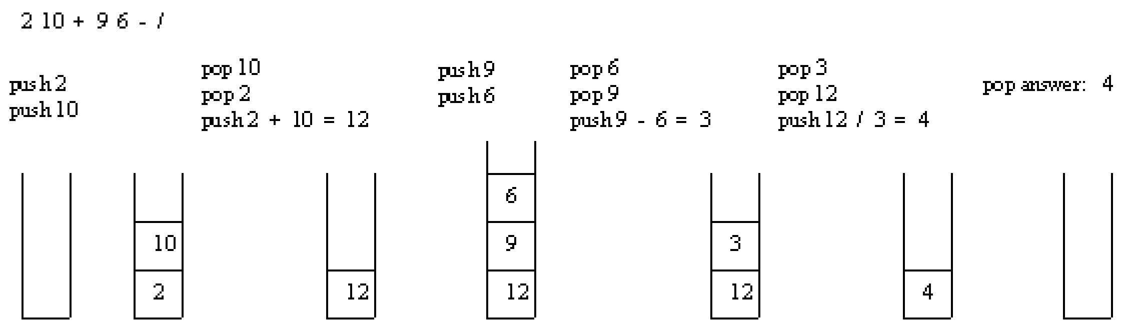
Create an empty stack and start scanning the postfix expression from left to right.

If the element is an operand, push it into the stack.

If the element is an operator O, pop twice and get A and B respectively. Calculate BOA and push it back to the stack.

When the expression is ended, the value in the stack is the final answer.

Evaluation of a postfix expression using a stack is explained in below example:



(pseudocode)

```
Begin

  for each character ch in the postfix expression, do
    if ch is an operator (+,-,/,*) , then
      a: pop first element from stack
      b: pop second element from the stack
      res: b ⊙ a
      push res into the stack
    else if ch is an operand, then
      add ch into the stack
  done
  return element of stack top
End
```

Queues

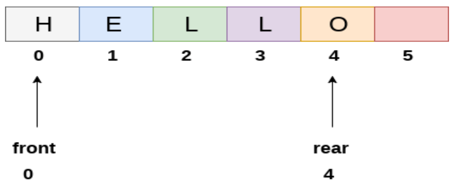
Definition

A queue is a foundational data structure used in programming applications. It is an abstract data type or a linear data structure that stores the elements sequentially. It uses the FIFO approach (First In First Out) for accessing elements.

Queues are typically used to manage threads in multithreading and implementing priority queuing systems.

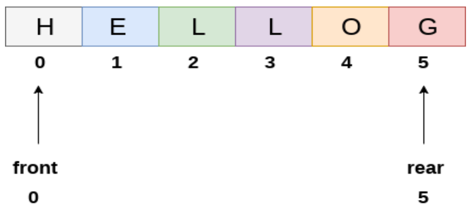
Array Representation

A queue can be represented using linear arrays. Two variables, namely the front and end are implemented in the case of arrays. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and queue is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.



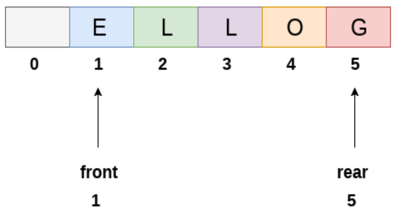
Queue

The above figure shows the queue of characters forming the English word "HELLO". Since no deletion is performed in the queue so far, the value of front remains -1. However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.



Queue after inserting an element

After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



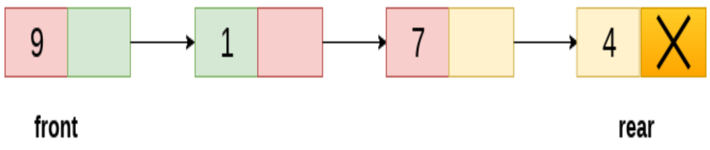
Queue after deleting an element

Linked Representation of Queue:

In the linked queue, there are two pointers maintained in the memory i.e., front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.

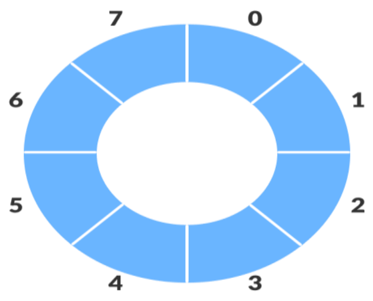
The linked representation of queue is shown in the following figure.



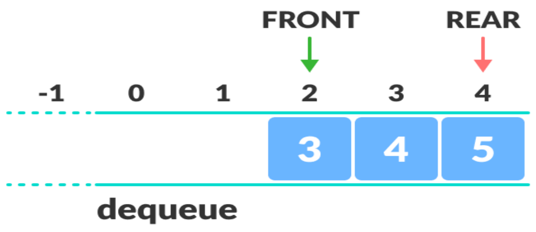
Linked Queue

Circular Queue

A circular queue is the extended version of a regular queue where the last element is connected to the first element. Thus, forming a circle-like structure.



The circular queue solves the major limitation of the normal queue. In a normal queue, after a bit of insertion and deletion, there will be non-usable empty space.



Priority Queue

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

Characteristics of a Priority queue

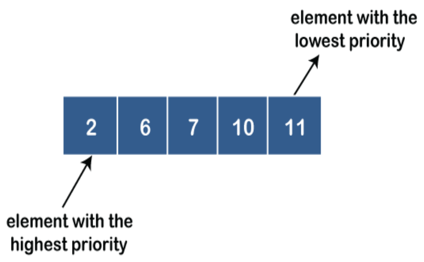
A priority queue is an extension of a queue that contains the following characteristics:

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

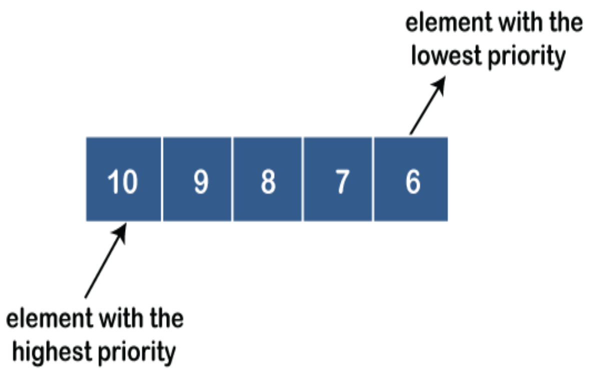
Types of Priority Queue

There are two types of priority queue:

Ascending order priority queue: In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.


element with the lowest priority
2
6
7
10
11
element with the highest priority

Descending order priority queue: In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue.


element with the lowest priority
10
9
8
7
6
element with the highest priority

Unit V

Sorting

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios –

Telephone Directory – The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.

Dictionary – The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

Bubble sort:

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Let's assume an array [4,5,7,2,1,6]

Now, in bubble sort, elements are compared in a two-by-two order.

Total iteration = size of array (6) – 1.

Iteration 1:

[<u>4,5,</u>7,2,1,6]

Swap if 4>5:

Else no swapping

Iteration 2:

[4,$\underline{5,7}$,2,1,6]

Swap if 5>7:

Else no swapping.

Iteration 3:

[4,5,$\underline{7,2}$,1,6]

Swap if 7>2: True.

[4,5,2,7,1,6]

Iteration 4:

[4,5,2,$\underline{7,1}$,6]

Swap if 7>1: True.

[4,5,2,1,7,6]

Iteration 5:

[4,5,2,1,$\underline{7,6}$]

Swap if 7>6: True.

[4,5,2,1, 6, 7]

This will continue on until the index position is at i-1. Why "-1"?

Because the largest number in bubble sort will got to the end during the first J^{th} iteration.

Code:

```
#include <stdio.h>
int main(){
   int arr[50], num, x, y, temp;
   printf("Please Enter the Number of Elements you want in the array: ");
   scanf("%d", &num);
   printf("Please Enter the Value of Elements: ");
   for(x = 0; x < num; x++)
      scanf("%d", &arr[x]);
   for(x = 0; x < num - 1; x++){
      for(y = 0; y < num - x - 1; y++){
         if(arr[y] > arr[y + 1]){
            temp = arr[y];
            arr[y] = arr[y + 1];
            arr[y + 1] = temp;
         }
      }
   }
   printf("Array after implementing bubble sort: ");
   for(x = 0; x < num; x++){
      printf("%d  ", arr[x]);
   }
   return 0;
}
```

Insertion Sort:

this

Insertion sort algorithm picks elements one by one and places it to the right position where it belongs in the sorted list of elements. In the following C program, we have implemented the same logic.

Before going through the program, let's see the steps of insertion sort with the help of an example.

Input elements: **89** 17 8 12 0

Step 1: 89 **17** 8 12 0 (the bold elements are sorted list and non-bold unsorted

list)
Step 2: 17 89 **8** 12 0 (each element will be removed from unsorted list and placed at the right position in the sorted list)
Step 3: 8 17 89 **12** 0
Step 4: 8 12 17 89 **0**
Step 5: 0 8 12 17 89

Code:

```
#include<stdio.h>
int main(){
  int i, j, count, temp, number[25];
  printf("How many numbers u are going to enter?: ");
  scanf("%d",&count);

  printf("Enter %d elements: ", count);
  // This loop would store the input numbers in array
  for(i=0;i<count;i++)
    scanf("%d",&number[i]);

  // Implementation of insertion sort algorithm
  for(i=1;i<count;i++){
    temp=number[i];
    j=i-1;
    while((temp<number[j])&&(j>=0)){
      number[j+1]=number[j];
      j=j-1;
    }
    number[j+1]=temp;
  }

  printf("Order of Sorted elements: ");
  for(i=0;i<count;i++)
    printf(" %d",number[i]);

  return 0;
}
```

Quick Sort

Like merge sort in C, quick sorting in C also follows the principle of decrease and conquer — or, as it is often called, divide and conquer. The quicksort algorithm is a sorting algorithm that works by selecting a pivot point, and thereafter partitioning the number set, or array, around the pivot point.

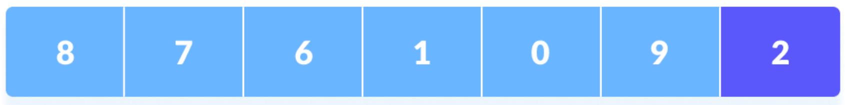
The main process in a quicksort algorithm is partitioning. If x is the pivot in an array, then the main intent of the sorting process is to put x at the right position in a sorted array, such that smaller elements precede x and greater elements follow it.

Once the pivot element has been selected, the elements smaller than the pivot element are placed before it, and the greater ones after.

Working of Quicksort Algorithm

1. Select the Pivot Element

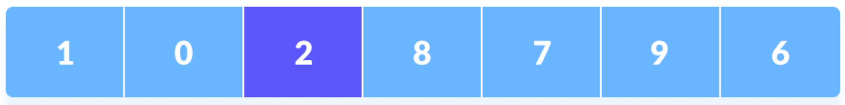
There are different variations of quicksort where the pivot element is selected from different positions. Here, we will be selecting the rightmost element of the array as the pivot element.



Select a pivot element

2. Rearrange the Array

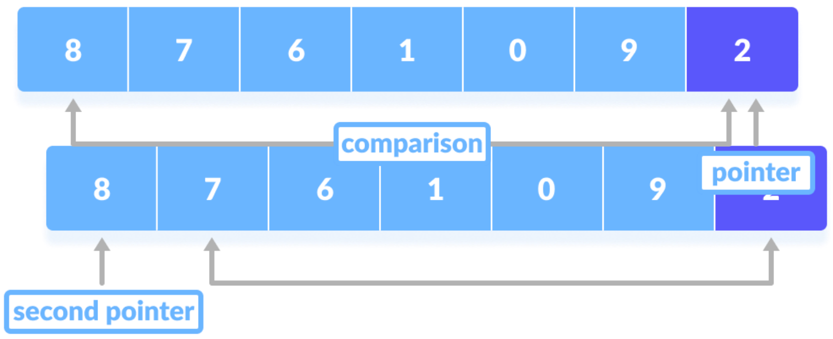
Now the elements of the array are rearranged so that elements that are smaller than the pivot are put on the left and the elements greater than the pivot are put on the right.



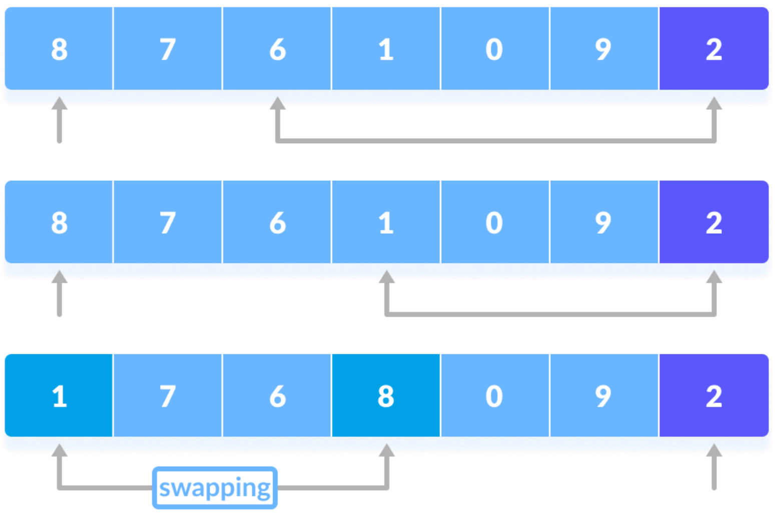
Put all the smaller elements on the left and greater on the right of pivot element

Here's how we rearrange the array:

A pointer is fixed at the pivot element. The pivot element is compared with the elements beginning from the first index.Comparison of pivot element with element beginning from the first index

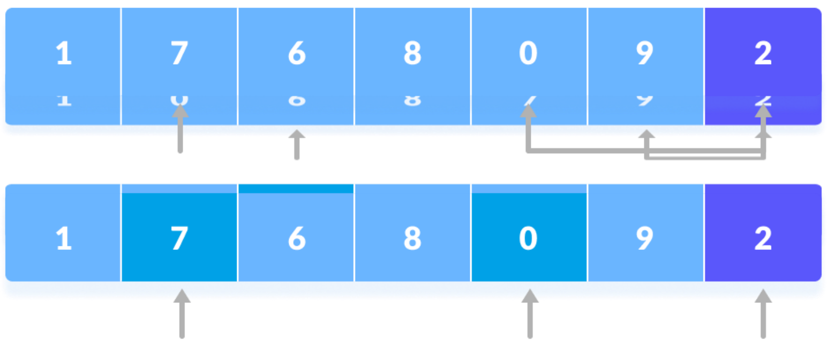


If the element is greater than the pivot element, a second pointer is set for that element.If the element is greater than the pivot element, a second pointer is set for that element.



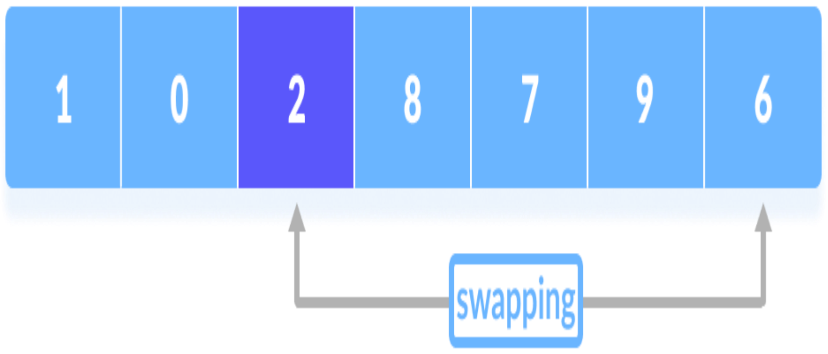
Now, pivot is compared with other elements. If an element smaller than the pivot element is reached, the smaller element is swapped with the greater element found earlier.Pivot is compared with other elements.

Again, the process is repeated to set the next greater element as the second pointer. And, swap it with another smaller element.The process is repeated to set the next greater element as the second pointer.



The process goes on until the second last element is reached.The process goes on until the second last element is reached.

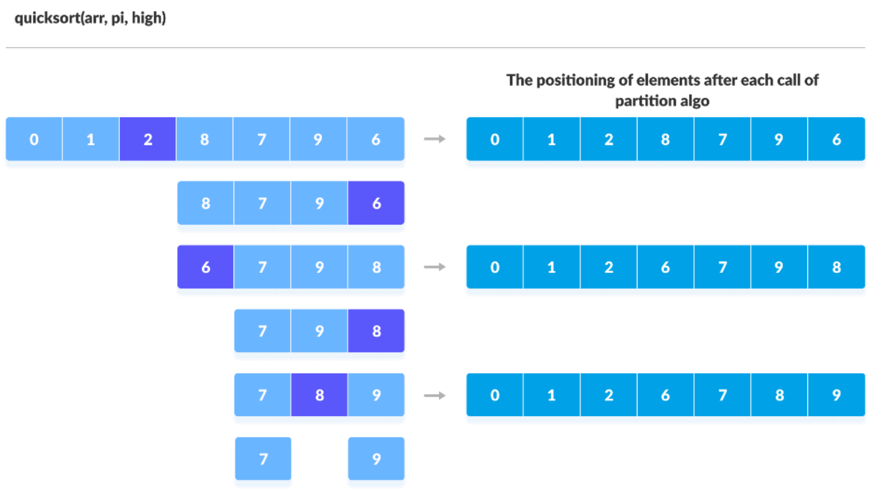
Finally, the pivot element is swapped with the second pointer.Finally, the pivot



element is swapped with the second pointer.

3. Divide Subarrays

Pivot elements are again chosen for the left and the right sub-parts separately. And, step 2 is repeated.



Select pivot element of in each half and put at correct place using recursion

The subarrays are divided until each subarray is formed of a single element. At this point, the array is already sorted.

Code

```
#include<stdio.h>
#define MAX 100
int arr[MAX];
void quicksort(int first,int last){

  int i, j, pivot, temp;

  if(first<last)
  {
    pivot=first;
    i=first;
    j=last;

    while(i<j)
    {
      while(arr[i]<=arr[pivot]&&i<last)
        i++;
      while(arr[j]>arr[pivot])
        j--;
      if(i<j)
      {
        temp=arr[i];
        arr[i]=arr[j];
        arr[j]=temp;
      }
    }
```

```
            temp=arr[pivot];
            arr[pivot]=arr[j];
            arr[j]=temp;
            quicksort(first,j-1);
            quicksort(j+1,last);
        }
}

int main()
{
    int i, size;
    printf("Enter size for the array: ");
    scanf("%d",&size);

    printf("Enter %d elements: ", size);
    for(i=0;i<size;i++)
        scanf("%d",&arr[i]);

    quicksort(0,size-1);

    printf("The Sorted Order is: ");
    for(i=0;i<size;i++)
        printf(" %d",arr[i]);

    return 0;

}
```

Selection Sort

Selection sort is another algorithm that is used for sorting. This sorting algorithm, iterates through the array and finds the smallest number in the array and swaps it with the first element if it is smaller than the first element. Next, it goes on to the second element and so on until all elements are sorted.

Example of Selection Sort

Consider the array:

[10,5,2,1]

The first element is 10. The next part we must find the smallest number from the remaining array. The smallest number from 5 2 and 1 is 1. So, we replace 10 by 1.

The new array is [1,5,2,10] Again, this process is repeated.

Finally, we get the sorted array as [1,2,5,10].

Code

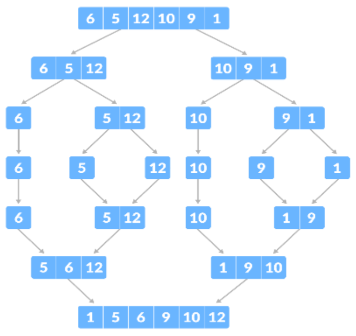
```
#include <stdio.h>
int main()
{
        int a[100], n, i, j, position, swap;
        printf("Enter number of elementsn");
        scanf("%d", &n);
        printf("Enter %d Numbersn", n);
        for (i = 0; i < n; i++)
                scanf("%d", &a[i]);
        for(i = 0; i < n - 1; i++)
        {
                position=i;
                for(j = i + 1; j < n; j++)
                {
                        if(a[position] > a[j])
                                position=j;
                }
                if(position != i)
                {
                        swap=a[i];
                        a[i]=a[position];
                        a[position=swap;
                }
        }
        printf("Sorted Array:n");
        for(i = 0; i < n; i++)
                printf("%dn", a[i]);
        return 0;
}
```

Merging Sort

(detailed**)**

Merge Sort is one of the most popular sorting algorithms that is based on the principle of Divide and Conquer Algorithm.

Here, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are combined to form the final solution.



Using the Divide and Conquer technique, we divide a problem into subproblems. When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem.

Suppose we had to sort an array A. A subproblem would be to sort a sub-section of this array starting at index p and ending at index r, denoted as A[p..r].

Divide

If q is the half-way point between p and r, then we can split the subarray A[p..r] into two arrays A[p..q] and A[q+1, r].

Conquer

In the conquer step, we try to sort both the subarrays A[p..q] and A[q+1, r]. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

Combine

When the conquer step reaches the base step and we get two sorted subarrays A[p..q] and A[q+1, r] for array A[p..r], we combine the results by creating a sorted array A[p..r] from two sorted subarrays A[p..q] and A[q+1, r].

```
// Merge sort in C

#include <stdio.h>

// Merge two subarrays L and M into arr
void merge(int arr[], int p, int q, int r) {

  // Create L ← A[p..q] and M ← A[q+1..r]
  int n1 = q - p + 1;
  int n2 = r - q;

  int L[n1], M[n2];

  for (int i = 0; i < n1; i++)
    L[i] = arr[p + i];
  for (int j = 0; j < n2; j++)
    M[j] = arr[q + 1 + j];

  // Maintain current index of sub-arrays and main array
  int i, j, k;
  i = 0;
  j = 0;
  k = p;

  // Until we reach either end of either L or M, pick larger among
  // elements L and M and place them in the correct position at A[p..r]
  while (i < n1 && j < n2) {
    if (L[i] <= M[j]) {
      arr[k] = L[i];
      i++;
    } else {
      arr[k] = M[j];
      j++;
    }
    k++;
  }

  // When we run out of elements in either L or M,
  // pick up the remaining elements and put in A[p..r]
  while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
  }

  while (j < n2) {
    arr[k] = M[j];
    j++;
    k++;
  }
}

// Divide the array into two subarrays, sort them and merge them
void mergeSort(int arr[], int l, int r) {
  if (l < r) {

    // m is the point where the array is divided into two subarrays
    int m = l + (r - l) / 2;

    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);

    // Merge the sorted subarrays
    merge(arr, l, m, r);
  }
}

// Print the array
void printArray(int arr[], int size) {
  for (int i = 0; i < size; i++)
    printf("%d ", arr[i]);
  printf("\n");
}

// Driver program
int main() {
  int arr[] = {6, 5, 12, 10, 9, 1};
  int size = sizeof(arr) / sizeof(arr[0]);

  mergeSort(arr, 0, size - 1);

  printf("Sorted array: \n");
  printArray(arr, size);
}
```

Searching

Search algorithms are a fundamental computer science concept that you should understand as a developer. They work by using a step-by-step method to locate specific data among a collection of data.

Types of Search Algorithms

In this post, we are going to discuss two important types of search algorithms:

Linear or Sequential Search

Binary Search

Linear or Sequential Search

This algorithm works by sequentially iterating through the whole array or list from one end until the target element is found. If the element is found, it returns its index, else -1.

Now let's look at an example and try to understand how it works:

arr = [2, 12, 15, 11, 7, 19, 45]

Suppose the target element we want to search is 7.

Approach for Linear or Sequential Search

Start with index 0 and compare each element with the target

If the target is found to be equal to the element, return its index

If the target is not found, return -1

Code Implementation

```
//Linear search
#include<stdio.h>
#define MAX 100

int main()
{
    int arr[MAX];
    int i, size, to_find, place, count;

    printf("Enter the array size: \n");
    scanf("%d", &size);

    printf("Enter elements into array: \n");
    for(i=0; i<size; i++)
        scanf("%d", &arr[i]);

    printf("Enter value to find: \n");
    scanf("%d", &to_find);

    printf("Given elements:");
    for(i=0;i<size;i++)
    {
        if(arr[i]==to_find)
        {
            count = i;
            place = arr[i];
        }
        printf("%d\t", arr[i]);
    }
    if(to_find == place)
    {
        printf("\nFound! Your element: %d\n", to_find);
        printf("\nFound at location %d", count);
    }else{
        printf("Not found!\n");
    }
}
```

Binary Search

This type of searching algorithm is used to find the position of a specific value contained in a sorted array. The binary search algorithm works on the principle of divide and conquer and it is considered the best searching algorithm because it's faster to run.

Now let's take a sorted array as an example and try to understand how it works:

arr = [2, 12, 15, 17, 27, 29, 45]

Suppose the target element to be searched is 17.

Approach for Binary Search

Compare the target element with the middle element of the array.

If the target element is greater than the middle element, then the search continues in the right half.

Else if the target element is less than the middle value, the search continues in the left half.

This process is repeated until the middle element is equal to the target element, or the target element is not in the array

If the target element is found, its index is returned, else -1 is returned.

Code Implementation

Indexed Search

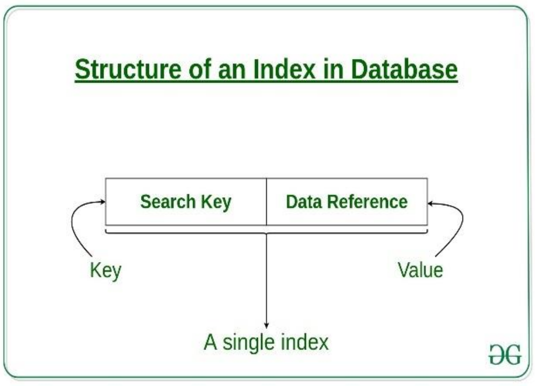
(detailed)

Indexing is a way to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed. It is a data structure technique which is used to quickly locate and access the data in a database.

Indexes are created using a few database columns.

- The first column is the Search key that contains a copy of the primary key or candidate key of the table. These values are stored in sorted order so that the corresponding data can be accessed quickly. Note: The data may or may not be stored in sorted order.

- The second column is the Data Reference or Pointer which contains a set of pointers holding the address of the disk block where that particular key value can be found.



The indexing has various attributes:

- **Access Types:** This refers to the type of access such as value-based search, range access, etc.
- **Access Time:** It refers to the time needed to find particular data element or set of elements.
- **Insertion Time:** It refers to the time taken to find the appropriate space and insert a new data.
- **Deletion Time:** Time taken to find an item and delete it as well as update the index structure.
- **Space Overhead:** It refers to the additional space required by the index.

Types of Indexing:

- **Primary:**
    - **Sparse**
    - **Dense**
- **Secondary**
- **Clustered**

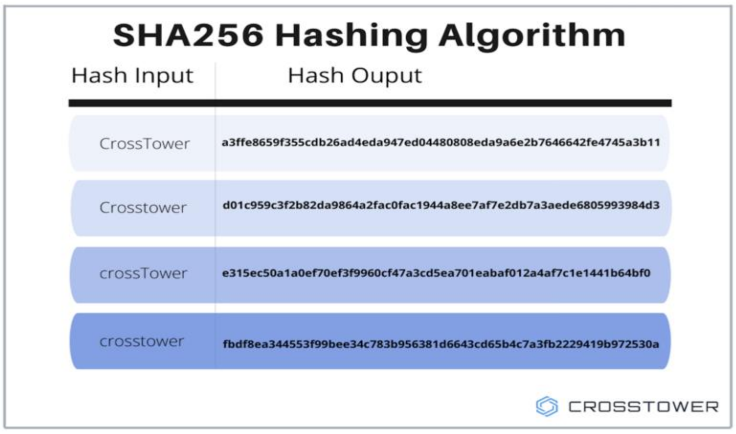
Hashing

Introduction

Hash algorithms were a breakthrough in the cryptographic computing world. This special type of programming function is used to store data of arbitrary size to data of a fixed size. Hash functions were created to compress data to reduce the amount of memory required for storing large files. The hashes they create can be stored in a special data structure called hash tables, which enables quicker data lookups.

Hashing, in the context of cryptocurrency, is the process of computing a "hash value" from plain text in order to protect against interference.

The following are 32-byte hash values produced by a SHA-256 hash calculator:



SHA256 Hashing Algorithm

Hash Input	Hash Ouput
CrossTower	a3ffe8659f355cdb26ad4eda947ed04480808eda9a6e2b7646642fe4745a3b11
Crosstower	d01c959c3f2b82da9864a2fac0fac1944a8ee7af7e2db7a3aede6805993984d3
crossTower	e315ec50a1a0ef70ef3f9960cf47a3cd5ea701eabaf012a4af7c1e1441b64bf0
crosstower	fbdf8ea344553f99bee34c783b956381d6643cd65b4c7a3fb2229419b972530a

CROSSTOWER

What is a Hashing Function?

Hash functions differ by type; however, there are several characteristics that persist between them.

Deterministic: The hash value remains the same. No matter how many times you input a message into the hashing function you need to receive the same output. The deterministic nature is key to creating order within the system utilizing the hash function.

Quick Computation: For a hash function to be used for real-world applications there needs to be efficient computation for any given message. The hashing function should quickly return a hash value for any potential given message.

Irreversible: There is no reverse engineering; messages cannot be re-traced from the hash output. It is impossible for an input to be regenerated from its hash value. The hash algorithm is designed to be a one-way function so if the hash function can be reversed then it is deemed compromised and no longer viable for storing sensitive data.

Popular Hashing Algorithms

Numerous hashing algorithms have been developed throughout the course of digital forensics, of which some of the most prominent include:

Message Digest 5 (MD5)

No longer actively used, MD5 was one of the most common hashing algorithms in early cryptography. Because of its several vulnerabilities, including the frequency of collisions, no cryptocurrencies make use of the 128-bit outputs.

RSA

Named after its designers (Rivest-Shamir-Adleman), RSA is a cryptosystem that originated in the late twentieth century. RSA uses a simple method of distribution: Person A uses Person B's public key to encrypt a message and Person B uses a private key, which remains secret to the user, to uncover its meaning. No active cryptocurrencies use the RSA framework.

Secure Hash Algorithm (SHA)

Secure Hash Algorithm (SHA) is a family of cryptographic hash functions that are used by most cryptocurrencies. This family of cryptographic hash functions were developed by the National Institute of Standards and Technology. Each hashing algorithm released under the SHA family builds upon the last version and since 2000 there has not been a new SHA algorithm released. SHA-384 is used to protect NSA information up to TOP SECRET. Consider this one of the most secure hashing algorithms.

Scrypt

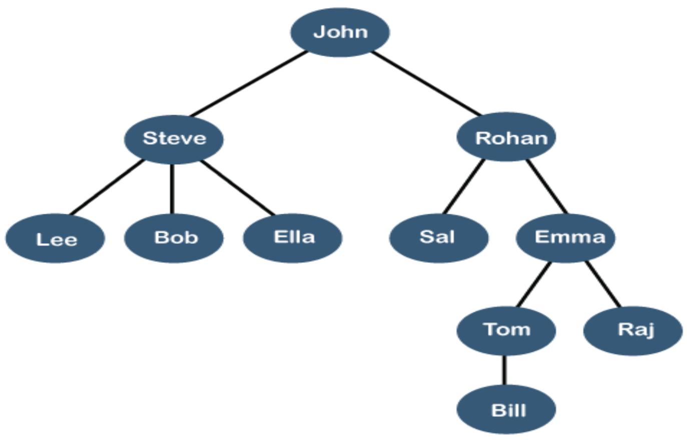
This hash function is computationally intensive which by design takes relatively longer time to compute. Due to the time complexity of the hash algorithm and the big memory volume required, Scrypt hash algorithm is very secure. Litecoin is the most popular cryptocurrency that uses Scrypt to secure its blockchain.

Ethash

Ethash is a proof-of-work mining algorithm created and implemented by the Ethereum network. This hash algorithm was developed to meet three main concerns in the cryptocurrency community: ASIC-resistance, light client verifiability and handling full chain storage. Vitalik Buterin is credited with helping create this hash algorithm.

Unit 3

A tree is also one of the data structures that represent hierarchical data. Suppose we want to show the employees and their positions in the hierarchical form then it can be represented as shown below:



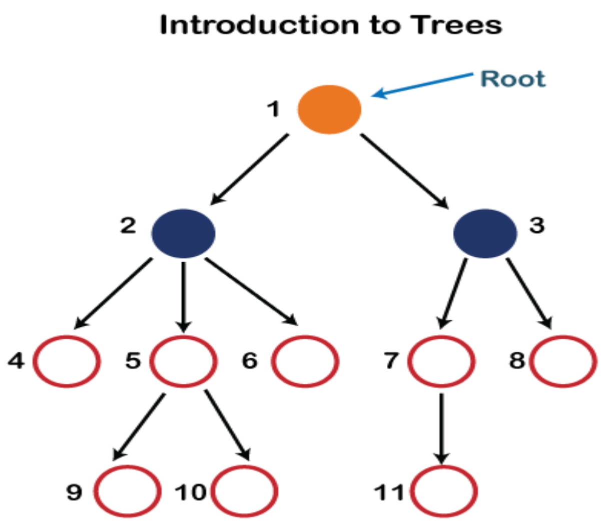
The above tree shows the organization hierarchy of some company. In the above structure, john is the CEO of the company, and John has two direct reports named as Steve and Rohan. Steve has three direct reports named Lee, Bob, Ella where Steve is a manager. Bob has two direct reports named Sal and Emma. Emma has two direct reports named Tom and Raj. Tom has one direct report named Bill. This particular logical structure is known as a Tree. Its structure is similar to the real tree, so it is named a Tree. In this structure, the root is at the top, and its branches are moving in a downward direction. Therefore, we can say that the Tree data structure is an efficient way of storing the data in a hierarchical way.

Let's understand some key points of the Tree data structure.

- A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.
- A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in a Tree are arranged in multiple levels.
- In the Tree data structure, the topmost node is known as a root node. Each node contains some data, and data can be of any type. In the above tree structure, the node contains the name of the employee, so the type of data would be a string.
- Each node contains some data and the link or reference of other nodes that can be called children.
- Some basic terms used in Tree data structure.

TREE terminology:

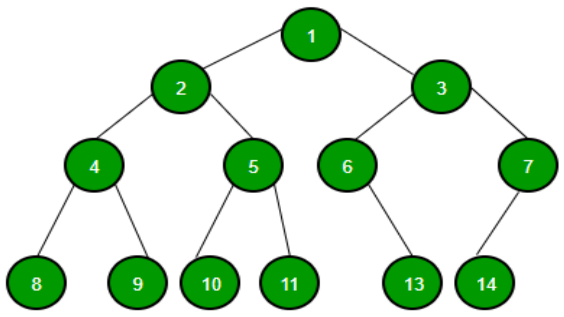
In the above structure, each node is labelled with some number. Each arrow shown in the above figure is known as a link between the two nodes.



- **Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is the root node of the tree. If a node is directly linked to some other node, it would be called a parent-child relationship.
- **Child node:** If the node is a descendant of any node, then the node is known as a child node.
- Parent: If the node contains any sub-node, then that node is said to be the parent of that sub-node.
- Sibling: The nodes that have the same parent are known as siblings.
- **Leaf Node:** The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.
- Internal nodes: A node has at least one child node known as an internal
- **Ancestor node:** An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.
- **Descendant:** The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

Binary Tree

A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



A Binary Tree node contains following parts.

- Data
- Pointer to left child
- Pointer to right child

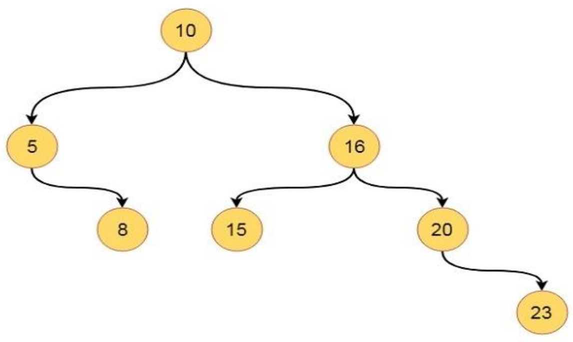
Refer this for types of Binary Trees

Binary Tree Representation in memory

As in array:

Here we will see how to represent a binary tree in computers memory. There are two different methods for representing. These are using array and using linked list.

Suppose we have one tree like this –



The array representation stores the tree data by scanning elements using level order fashion. So it stores nodes level by level. If some element is missing, it left blank spaces for it. The representation of the above tree is like below −

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
10	5	16	-	8	15	20	-	-	-	-	-	-	-	23

The index 1 is holding the root, it has two children 5 and 16, they are placed at location 2 and 3. Some children are missing, so their place is left as blank.

In this representation we can easily get the position of two children of one node by using this formula −

child1=2∗parentchild1=2∗parent

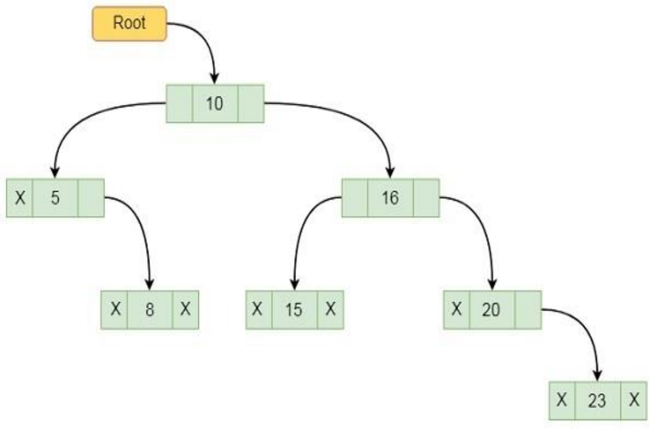
child2=(2∗parent)+1child2=(2∗parent)+1

To get parent index from child we have to follow this formula −

parent=[child2]parent=[child2]

This approach is good, and easily we can find the index of parent and child, but it is not memory efficient. It will occupy many spaces that has no use. This representation is good for complete binary tree or full binary tree.

Another approach is by using linked lists. We create node for each element. This will be look like below −



Traversing Binary Tree

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways.

Traversing a tree means visiting and outputting the value of each node in a particular order. In this tutorial, we will use the In-order, Pre-order, and Post order tree traversal methods.

The major importance of tree traversal is that there are multiple ways of carrying out traversal operations unlike linear data structures like arrays, bitmaps, matrices where traversal is done in a linear order.

Each of these methods of traversing a tree have a particular order they follow:

For **In order**, you traverse from the left subtree to the root then to the right subtree.

For **Pre order,** you traverse from the root to the left subtree then to the right subtree.

For **Post order,** you traverse from the left subtree to the right subtree then to the root.

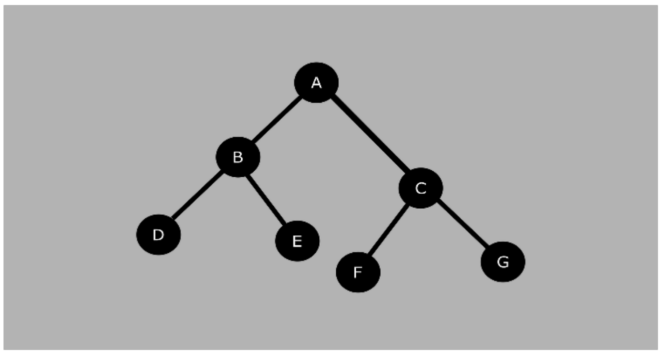
In-order:

Left->Root->Right

We are going to create a tree similar to the one in the last section, but this time the node keys will be numbers instead of letters.

Remember that the values of the nodes on the left subtree are always smaller than the value of the root node. Also, the values of the nodes on the right subtree are larger than the value of the root node.

Here is the diagram we will be working with:

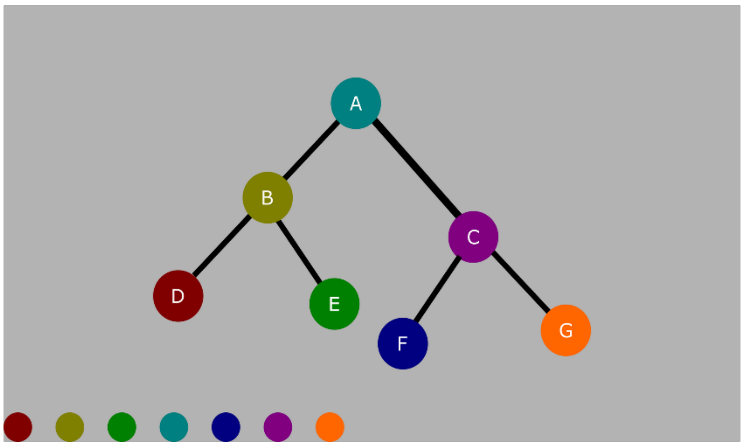


Recall that the order for in-order traversal is Left, Root, right.

This is the result we get after using in-order traversal:

D, B, E, A, F, C, G

If that seems a bit complex for you to understand, then follow the order of the colours in the picture below



- First, visit all the nodes in the left subtree
- Then the root node
- Visit all the nodes in the right subtree

```
inorder(root->left)
display(root->data)
inorder(root->right)
```

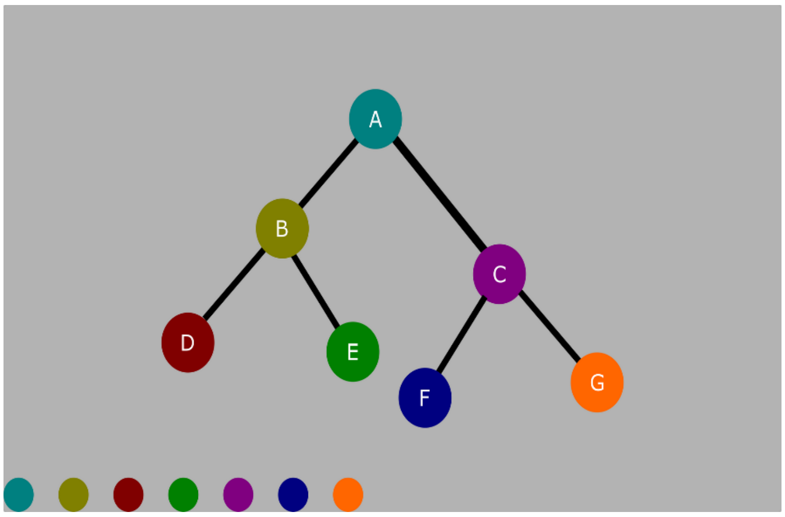
Pre-Order method

The order here is Root, Left and then Right.

Using the same diagram above, we have:

A, B, D, E, C, F, G

Here is the same diagram with different colours used as a guide:



- Visit root node
- Visit all the nodes in the left subtree
- Visit all the nodes in the right subtree

```
display(root->data)
preorder(root->left)
preorder(root->right)
```

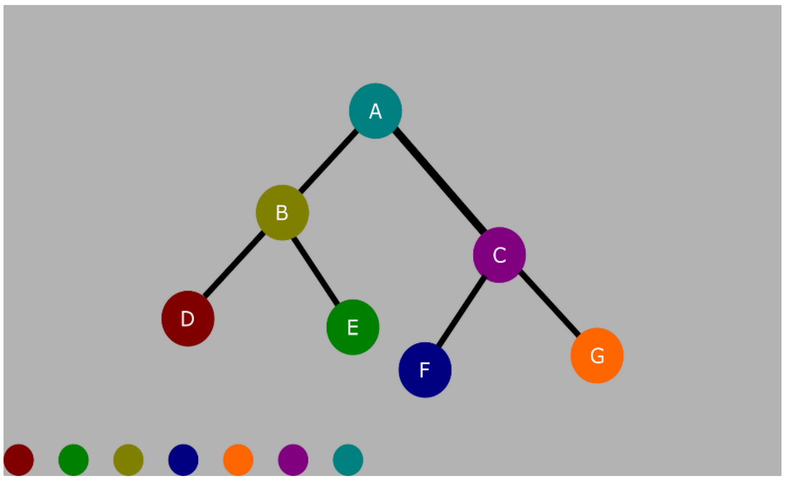
Post-Order:

The order for post order traversal is Left, Right, Root.

Here is the output:

D, E, B, F, G, C, A

If you can't figure out how we arrived at that result, then use the colours in the picture below as a guide:



Visit all the nodes in the left subtree

Visit all the nodes in the right subtree

Visit the root node

```
postorder(root->left)
postorder(root->right)
display(root->data)
```

```c
// Tree traversal in C

#include <stdio.h>
#include <stdlib.h>

struct node {
  int item;
  struct node* left;
  struct node* right;
};

// Inorder traversal
void inorderTraversal(struct node* root) {
  if (root == NULL) return;
  inorderTraversal(root->left);
  printf("%d ->", root->item);
  inorderTraversal(root->right);
}

// preorderTraversal traversal
void preorderTraversal(struct node* root) {
  if (root == NULL) return;
  printf("%d ->", root->item);
  preorderTraversal(root->left);
  preorderTraversal(root->right);
}

// postorderTraversal traversal
void postorderTraversal(struct node* root) {
  if (root == NULL) return;
  postorderTraversal(root->left);
  postorderTraversal(root->right);
  printf("%d ->", root->item);
}
```

```c
// Create a new Node
struct node* createNode(value) {
  struct node* newNode = malloc(sizeof(struct node));
  newNode->item = value;
  newNode->left = NULL;
  newNode->right = NULL;

  return newNode;
}

// Insert on the left of the node
struct node* insertLeft(struct node* root, int value) {
  root->left = createNode(value);
  return root->left;
}

// Insert on the right of the node
struct node* insertRight(struct node* root, int value) {
  root->right = createNode(value);
  return root->right;
}

int main() {
  struct node* root = createNode(1);
  insertLeft(root, 12);
  insertRight(root, 9);

  insertLeft(root->left, 5);
  insertRight(root->left, 6);

  printf("Inorder traversal \n");
  inorderTraversal(root);

  printf("\nPreorder traversal \n");
  preorderTraversal(root);

  printf("\nPostorder traversal \n");
  postorderTraversal(root);
}
```

Traversal Algorithm using stacks

(Algorithm only)

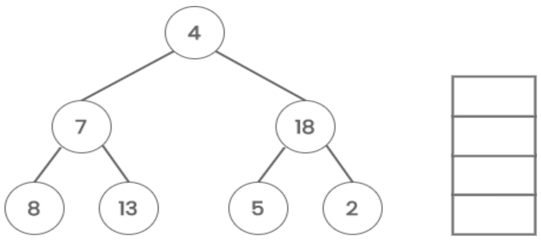
The following operations are performed to traverse a binary tree in pre-order using a stack:

Start with root node and push onto stack.

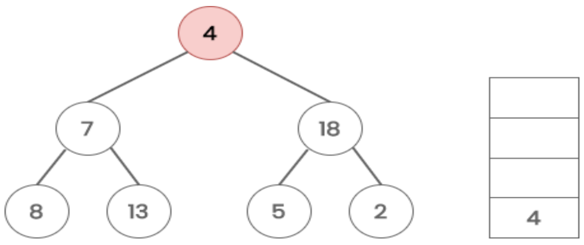
Repeat while the stack is not empty

- POP the top element (PTR) from the stack and process the node.
- PUSH the right child of PTR onto to stack.
- PUSH the left child of PTR onto to stack.

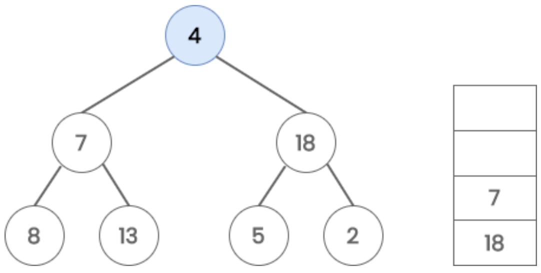
Consider the following tree.



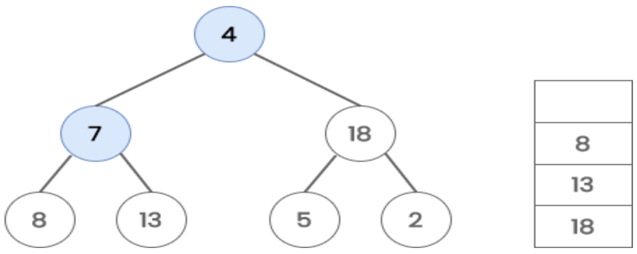
Start with node 4 and push it onto the stack.



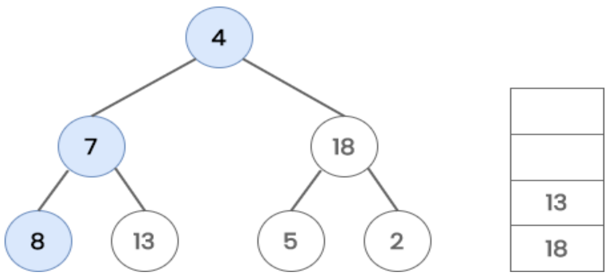
Since the stack is not empty, POP 4 from the stack, process it and PUSH its left(7) and right(18) child onto the stack.



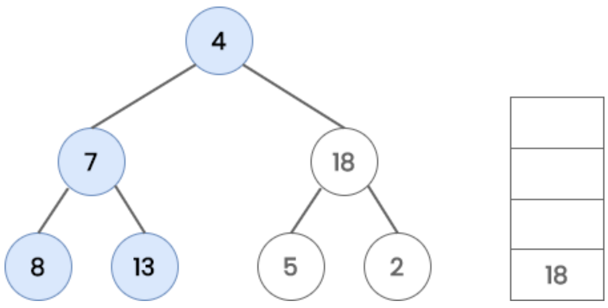
Repeat the same process since the stack is not empty. POP 7 from the stack, process it and PUSH its left(8) and right (13) child onto the stack.



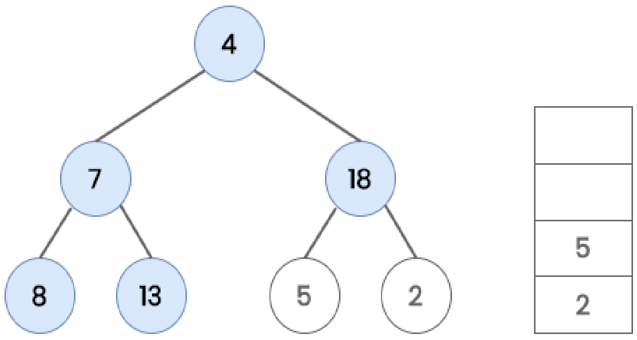
Again, POP 8 from the stack and process it. Since it has no right or left child we don't have to PUSH anything to the stack.



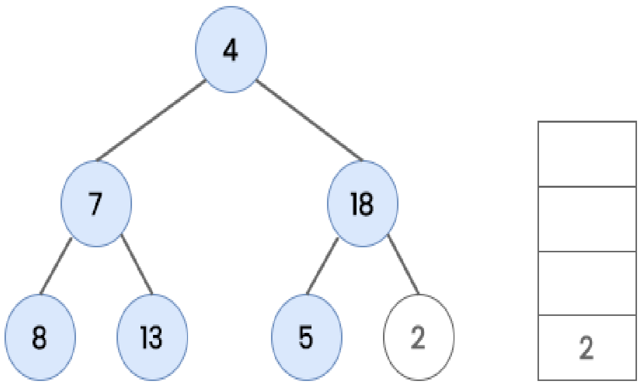
Now POP 13 from the stack and process it. We don't have to PUSH anything to the stack because it also doesn't have any subtrees.

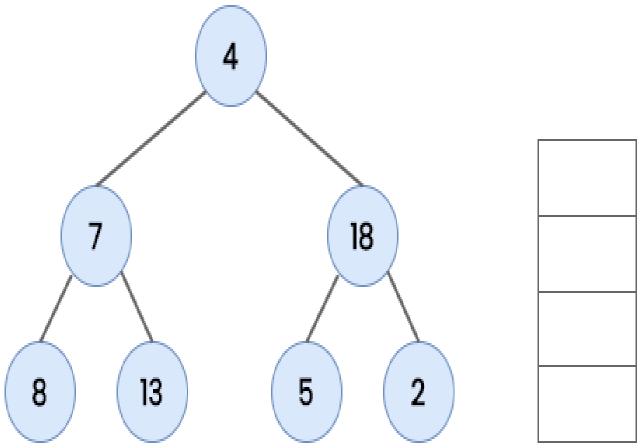


POP 18 from the stack, process it and PUSH its left(5) and right(2) child to the stack.



Similarly, POP 5 and 2 from the stack one after another. Since both these nodes don't have any child, we don't have to PUSH anything onto the stack.





The nodes are processed in the order

```
[ 4, 7, 8, 3, 18, 5, 2 ]
```

This is the required pre-order traversal of the given tree.

The formal algorithm is given below:

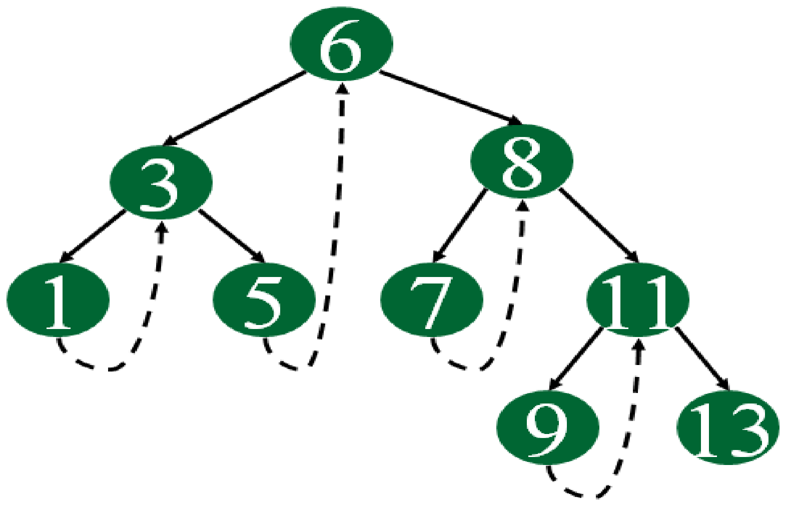
```
1. Set TOP = 1.STACK[1] = NULL and PTR = ROOT.
2. Repeat Steps 3 to 5 while PTR ≠ NULL:
3. Apply PROCESS to PTR->DATA.
4. If PTR->RIGHT ≠ NULL, then:
 Set TOP = TOP + 1, and STACK[TOP] = PTR->RIGHT.
 [End of If]
5. If PTR->LEFT ≠ NULL, then:
 Set PTR = PTR -> LEFT.
 Else:
 Set PTR = STACK[TOP] and TOP = TOP - 1.
 [End of If]
 [End of Step 2 loop.]
6. Exit.
```

Headed Nodes

Threads

(Definition only)

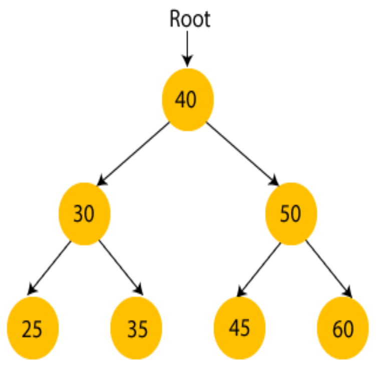
The idea of threaded binary trees is to make In-order traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the In-order successor of the node (if it exists).
There are two types of threaded binary trees.
Single Threaded: Where a NULL-right pointers is made to point to the In-order successor (if successor exists)
Double Threaded: Where both left and right NULL pointers are made to point to in-order predecessor and in-order successor respectively. The predecessor threads are useful for reverse in-order traversal and post-order traversal.
The threads are also useful for fast accessing ancestors of a node.
Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.



Binary Search Tree

A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.

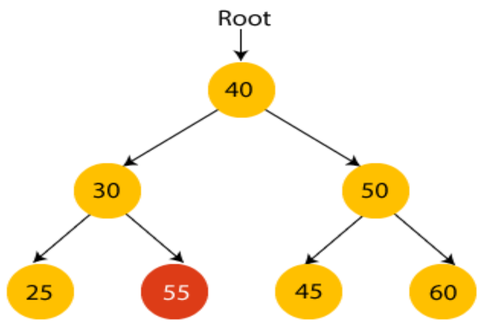
Let's understand the concept of Binary search tree with an example.



In the above figure, we can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node.

Similarly, we can see the left child of root node is greater than its left child and smaller than its right child. So, it also satisfies the property of binary search tree. Therefore, we can say that the tree in the above image is a binary search tree.

Suppose if we change the value of node 35 to 55 in the above tree, check whether the tree will be binary search tree or not.



In the above tree, the value of root node is 40, which is greater than its left child 30 but smaller than right child of 30, i.e., 55. So, the above tree does not satisfy the property of Binary search tree. Therefore, the above tree is not a binary search tree.

Search Operation

The algorithm depends on the property of BST that if each left subtree has values below root and each right subtree has values above the root.

If the value is below the root, we can say for sure that the value is not in the right subtree; we need to only search in the left subtree and if the value is above the root, we can say for sure that the value is not in the left subtree; we need to only search in the right subtree.

Algorithm:

```
If root == NULL
    return NULL;
If number == root->data
    return root->data;
If number < root->data
    return search(root->left)
If number > root->data
    return search(root->right)
```

Insert Operation

(ALL)

Inserting a value in the correct position is similar to searching because we try to maintain the rule that the left subtree is lesser than root and the right subtree is larger than root.

We keep going to either right subtree or left subtree depending on the value and when we reach a point left or right subtree is null, we put the new node there.

Algorithm:

```
If node == NULL
    return createNode(data)
if (data < node->data)
    node->left  = insert(node->left, data);
else if (data > node->data)
    node->right = insert(node->right, data);
return node;
```

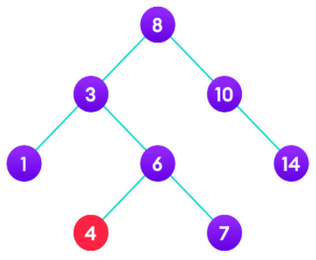
The algorithm isn't as simple as it looks. Let's try to visualize how we add a number to an existing BST.

Deletion Operation

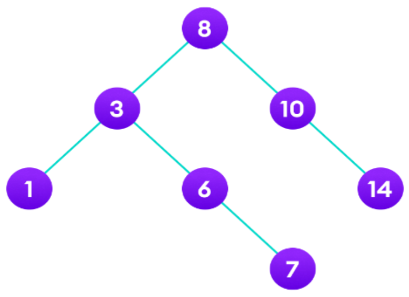
There are three cases for deleting a node from a binary search tree.

Case I

In the first case, the node to be deleted is the leaf node. In such a case, simply delete the node from the tree.



4 is to be deleted



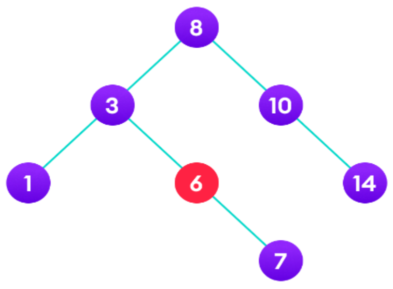
Delete the node

Case II

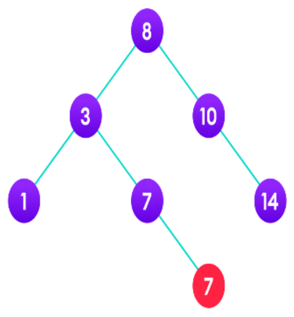
In the second case, the node to be deleted lies has a single child node. In such a case follow the steps below:

Replace that node with its child node.

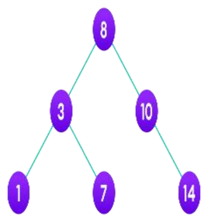
Remove the child node from its original position.



6 is to be deleted



copy the value of its child to the node and delete the child

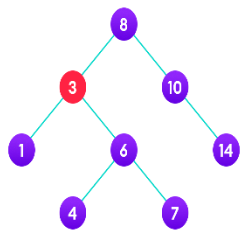


Final tree

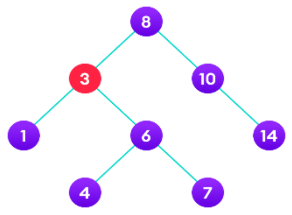
Case III

In the third case, the node to be deleted has two children. In such a case follow the steps below:

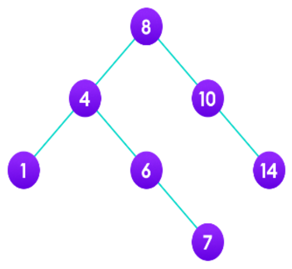
- Get the in-order successor of that node.
- Replace the node with the in-order successor.
- Remove the in-order successor from its original position.



3 is to be deleted



Copy the value of the in-order successor (4) to the node



Delete the In-order successor

AVL-Tree

AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

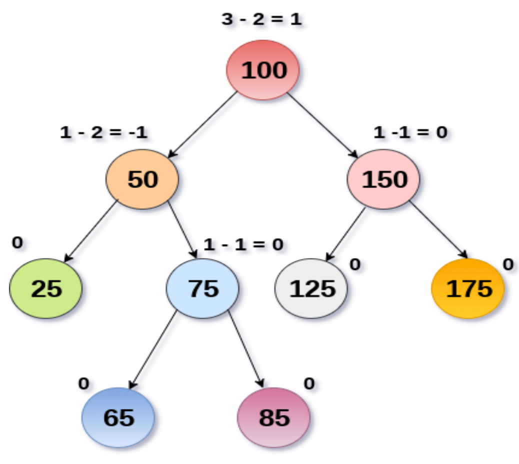
Balance Factor (k) = height (left(k)) - height (right(k))

If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.

If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.

If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.



AVL Tree

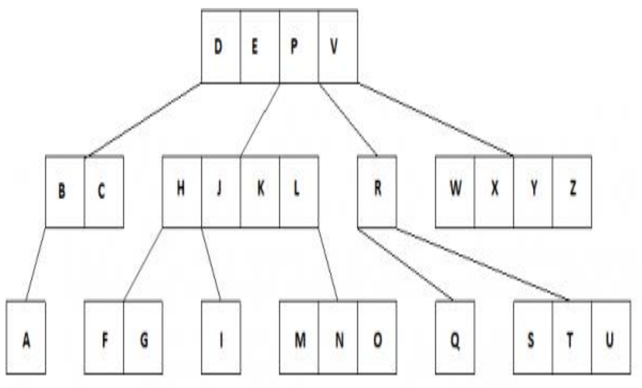
M-Way Tree

(Better understanding)

A multiway tree is defined as a tree that can have more than two children. If a multiway tree can have maximum m children, then this tree is called as multiway tree of order m (or an m-way tree).

As with the other trees that have been studied, the nodes in an m-way tree will be made up of m-1 key fields and pointers to children.

multiway tree of order 5



To make the processing of m-way trees easier some type of constraint or order will be imposed on the keys within each node, resulting in a multiway search tree of order m (or an m-way search tree). By definition an m-way search tree is a m-way tree in which following condition should be satisfied –

- Each node is associated with m children and m-1 key fields
- The keys in each node are arranged in ascending order.
- The keys in the first j children are less than the j-th key.
- The keys in the last m-j children are higher than the j-th key.

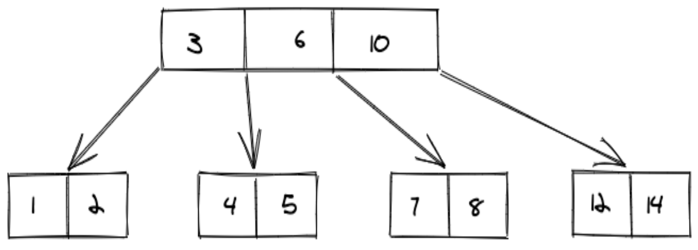
In computer science, M-trees are tree data structures that are similar to R-trees and B-trees. It is constructed using a metric and relies on the triangle inequality for efficient range and k-nearest neighbor (k-NN) queries.While M-trees can perform well in many conditions, the tree can also have large overlap and there is no clear strategy on how to best avoid overlap. In addition, it can only be used for distance functions that satisfy the triangle inequality, while many advanced dissimilarity functions used in information retrieval do not satisfy this.

B Trees

A B tree is an extension of an M-way search tree. This type of tree will be used when the data to be accessed/stored is located on secondary storage devices because they allow for large amounts of data to be stored in a node. Besides having all the properties of an M-way search tree, it has some properties of its own, these mainly are:

- All the leaf nodes in a B tree are at the same level.
- All internal nodes must have M/2 children.
- If the root node is a non-leaf node, then it must have at least two children.
- All nodes except the root node, must have at least [M/2]-1 keys and at most M-1 keys.

Consider the pictorial representation of a B tree shown below:



B tree

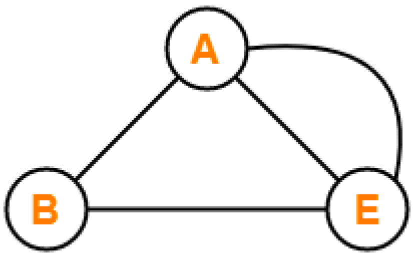
Unit-4

Graphs

A graph can be defined as group of vertices and edges that are used to connect these vertices. A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.

Multi-Graph

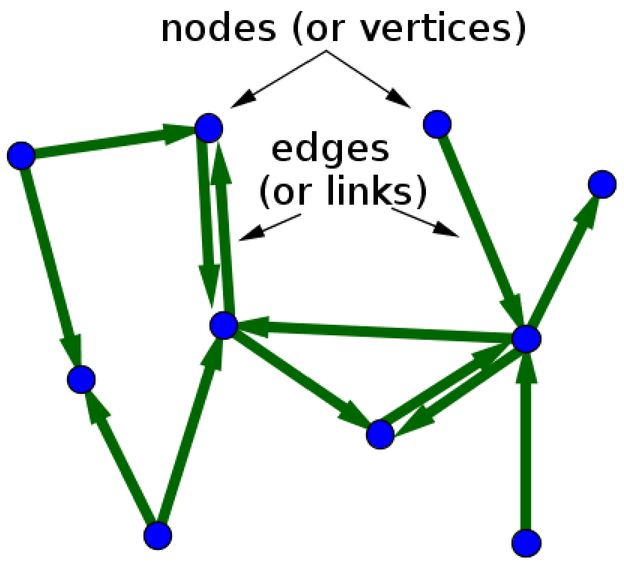
A graph is said to be a multigraph if the graph doesn't consist of any self-loops, but parallel edges are present in the graph. If there is more than one edge present between two vertices, then that pair of vertices is said to be having parallel edges.



We have three vertices and three edges for the graph that is shown in the above image. There are no self-loops, but two edges connect these two vertices between vertex A and vertex E of the graph. In other words, we can say that if two vertices of a graph are connected with more than one edge in a graph, then it is said to be having parallel edges, thus making it a multigraph.

Directed Graph

A directed graph is graph, i.e., a set of objects (called vertices or nodes) that are connected together, where all the edges are directed from one vertex to another. A directed graph is sometimes called a digraph or a directed network. In contrast, a graph where the edges are bidirectional is called an undirected graph.



When drawing a directed graph, the edges are typically drawn as arrows indicating the direction, as illustrated in the following figure.

A directed graph with 10 vertices (or nodes) and 13 edges.

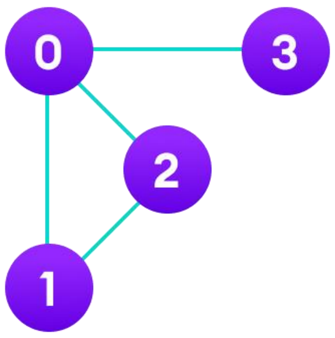
One can formally define a directed graph as G=(N,E)G=(N,E), consisting of the set NN of nodes and the set EE of edges, which are ordered pairs of elements of NN.

Sequential Representation of Graphs

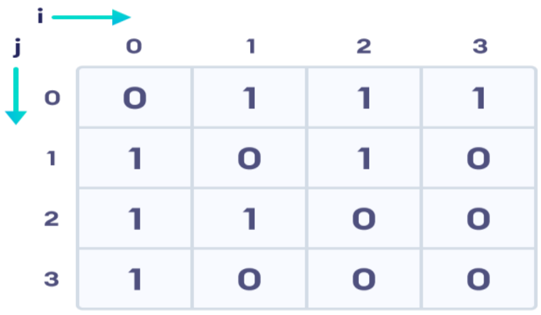
Adjacent Matrix:

An adjacency matrix is a way of representing a graph as a matrix of booleans (0's and 1's). A finite graph can be represented in the form of a square matrix on a computer, where the boolean value of the matrix indicates if there is a direct path between two vertices.

For example, we have a graph below.



We can represent this graph in matrix form like below.



	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

Matrix representation of the graph

Each cell in the above table/matrix is represented as Aij, where i and j are vertices. The value of Aij is either 1 or 0 depending on whether there is an edge from vertex i to vertex j.

If there is a path from i to j, then the value of Aij is 1 otherwise its 0. For instance, there is a path from vertex 1 to vertex 2, so A12 is 1 and there is no path from vertex 1 to 3, so A13 is 0.

In case of undirected graphs, the matrix is symmetric about the diagonal because of every edge (i,j), there is also an edge (j,i).

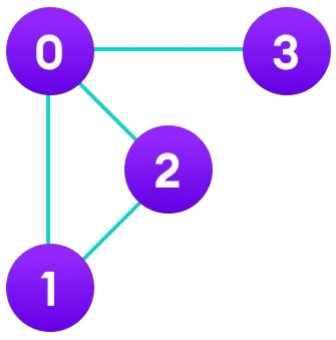
Path Matrix

(*here*)

Linked Representation of Graph/Adjacency List

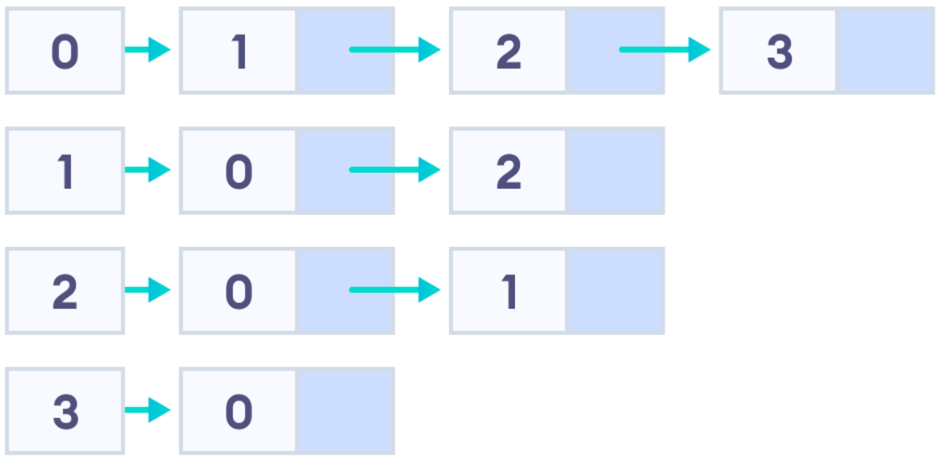
An adjacency list represents a graph as an array of linked lists. The index of the array represents a vertex and each element in its linked list represents the other vertices that form an edge with the vertex.

For example, we have a graph below.



An undirected graph

We can represent this graph in the form of a linked list on a computer as shown below.



Linked list representation of the graph

Here, 0, 1, 2, 3 are the vertices and each of them forms a linked list with all of its adjacent vertices. For instance, vertex 1 has two adjacent vertices 0 and 2. Therefore, 1 is linked with 0 and 2 in the figure above.

(this)

Graph Operations

Add/Remove Vertex – Add or remove a vertex in a graph.

Add/Remove Edge – Add or remove an edge between two vertices.

Check if the graph contains a given value.

Find the path from one vertex to another vertex.

Depth First Search

Refer this

Breadth First Search

Refer this

Spanning Tree

Refer this