Data and File Structures

(MCA-102)

Unit – 1

[Array (Searching and Sorting), Linked List, Stack and Queue]

by

Dr. Sunil Pratap Singh
(Assistant Professor, BVICAM, New Delhi)

2021

Introduction

- Data Type: A *data type* is a term which refers to the kinds of data that variables may "hold" in a programming language.
  - For example, a variable of type boolean can assume either the value true or the value false, but no other value.
- Data Structure: A data structure is an arrangement of data in a computer's memory (or sometimes on a disk).
  - In other words, a data structure is meant to be an organization or structuring for a collection of data items. A sorted list of integers stored in an array is an example of such a structuring.
  - Algorithms manipulate the data in these structures in various ways, such as inserting a new data item, searching for a particular item, or sorting the items.

Categories of Data Structures

- **Linear Data Structures**
  - A data structure whose elements form a sequence, and every element in the structure has a unique **predecessor** and unique **successor**.
  - Examples: **Array**, **Stack**, **Queue**, **Linked List**
- **Non-Linear Data Structures**
  - A data structure whose elements do not form a sequence, there is no unique predecessor or unique successor.
  - Examples: **Tree**, **Graph**

Common Operations on Data Structures

- Traversal: accessing or visiting each data item exactly once
- Searching: finding the data item within the data structure which satisfies searching condition
- Insertion: adding a new data element within the data structure
- Deletion: removing a new data element from the data structure
- Sorting: arranging the data in some logical order
- Merging: combining the data elements of two data structures

Array

- An array is a fixed-size sequential collection of elements of same data type.
- An array is simply a grouping of like-type data.
- In its simplest form, an array can be used to represent a list of numbers, or a list of names.
- Some examples where the concept of an array can be used:
  - List of temperatures recorded every hour in a day
  - List of employees in an organization
  - Test scores of a class of students
  - Table of daily rainfall data
  - etc.

One-Dimensional Array

- A list of items can be given one variable name using only one subscript and such a variable is called a single-scripted variable of a one-dimensional array.
- Declaration:

```
data-type variable-name[size];
```

- Declaration Examples:

```
float height[50];
int group[10];
char name[10];
```

One-Dimensional Array

- Initialization at Compile Time:

```
data-type variable-name[size] = {list of values};
```

- Compile Time Initialization Examples

```
int number[3] = {5, 6, 7};

int age[5] = {22, 24, 23};
            --> Remaining two elements will be initialized to 0.

int counter[] = {1, 2, 3, 4, 5};
            --> The array size may be omitted.

char city[5] = {'D', 'E', 'L'};
            --> Remaining two elements will be initialized to NULL.
```

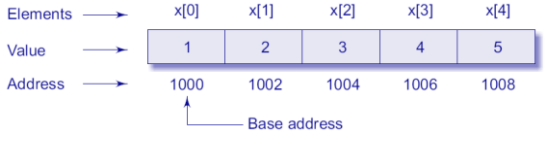

One-Dimensional Array

- Run Time Initialization Examples

```
int counter[10];
for(i=1, i<=10, i++)
{
    counter[i] = i;
}
```

```
Using scanf() function
int counter[10];
for(i=1, i<=10, i++)
{
    scanf("%d", &counter[i]);
}
```


Memory Layout of 1D Array



Calculating Address of Elements in 1D Array

- Let `x[n]` be an one-dimensional array having `n` elements with indices `i` = 0, 1, … …, n-1.
- Then, the address of i^{th} element (`x[i]`) is calculated as follows:

Base Address + (i × Scale Factor of Data Type of Array)

Example: Given an array x[5] of integers with base address = 1000. Calculate the address of element x[3].

Address of x[3] = Base Address + (3 × Scale Factor of Integer)

= 1000 + (3 × 2) = 1006

Two-Dimensional Array

- Declaration:

```
data-type variable-name[row-size] [column-size];
```

- Declaration Examples:

```
float sales[3][3];

int matrix[4][3];
```


Two-Dimensional Array

- Initialization at Compile Time:

```
data-type variable-name[row-size][column-size] = {list of values};
```

- Compile Time Initialization Examples

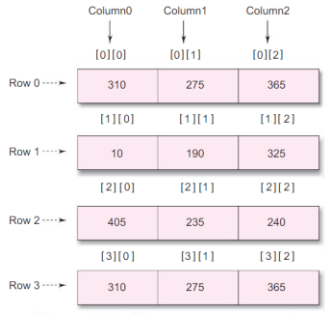
```
int table[2][3] = {1, 1, 1, 2, 2, 2};

int table[2][3] = {{1, 1, 1}, {2, 2, 2}};

--> When array is initialized with all values, explicitly, we need
not specify the size of first dimension.
int table[][3] = {{1, 1, 1}, {2, 2, 2}};

int table[2][3] = {{1, 1}, {2}};
--> It will initialize the first two elements of first row to one,
the first element of second row to two, and all other to zero.
```

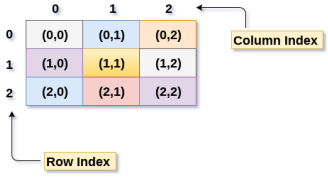
Representation of 2D Array



Memory Layout of 2D Array

- There are two main techniques of storing 2D array elements into memory:
  - Row Major Ordering
    - All the **rows** of the 2D array are stored into the memory contiguously.
  - Column Major Ordering
    - All the **columns** of the 2D array are stored into the memory contiguously.

Memory Layout of 2D Array (contd…)



Row Major Ordering

(0,0)	(0,1)	(0,2)	(1,0)	(1,1)	(1,2)	(2,0)	(2,1)	(2,2)

Column Major Ordering

(0,0)	(1,0)	(2,0)	(0,1)	(1,1)	(2,1)	(0,2)	(1,2)	(2,2)

Calculating Address of Elements in 2D Array

- Let `x[m][n]` be a two-dimensional array having `m` rows and `n` columns with indices i = 0, 1, … …, m; j = 0, 1, … …n.
- Then, the address of an element `x[i][j]` of the array, stored in Row Major, is calculated as:

Base Address + (i × n + j) × Scale Factor of Data Type of Array

Example: Given an array x[5][7] of integers with base address = 900. Calculate the address of element x[4][6].

Address of x[4][6] = 900 + (4 × 7 + 6) × 2 = **968**

Question: Given an array [1…5, 1…7] of integers with base address = 900. Calculate address of element [4, 4].

Calculating Address of Elements in 2D Array

- Let `x[m][n]` be a two-dimensional array having `m` rows and `n` columns with indices i = 0, 1, … …, m; j = 0, 1, … …n.
- Then, the address of an element `x[i][j]` of the array, stored in Column Major, is calculated as:

Base Address + (j × m + i) × Scale Factor of Data Type of Array

Example: Given an array x[5][7] of integers with base address = 900. Calculate the address of element x[4][6].

Address of x[4][5] = 900 + (5 × 5 + 4) × 2 = **958**

Question: Given an array [1…5, 1…6] of integers with base address = 2000. Calculate address of element [4, 4].

Sparse Matrix

- A matrix can be defined as a two-dimensional array having 'm' columns and 'n' rows representing m×n matrix.
- Sparse matrices are those matrices that have the majority of their elements equal to zero.
  - In other words, the sparse matrix is a matrix that has a greater number of zero elements than the non-zero elements.

	0	1	2	3
0	0	4	0	5
1	0	0	3	6
2	0	0	2	0
3	2	3	0	0
4	0	0	0	0

Limitations of Sparse Matrix

- Storage
  - We need to store m×n (all elements) elements of matric even though maximum number of elements of the matrix are zero.
- Computing Time
  - In case of searching (or performing any operation) in a sparse matrix, we need to traverse m×n (all elements) rather than accessing non-zero elements of the sparse matrix.

Sparse Matrix Representation

- The non-zero elements can be stored with triples, i.e., rows, columns, and value.
- The sparse matrix can be represented in the following ways:
  - Array Representation
  - Linked List Representation
  - List of Lists Representation

Sparse Matrix: Triples/Array Representation)

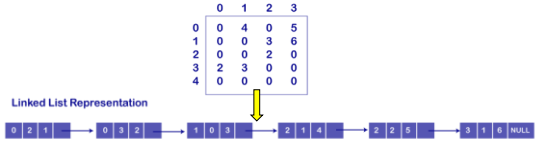
- A 2D array with 3 row or columns is used to represent the sparse matrix:
  - **Row**: It is an index of a row where a non-zero element is located.
  - **Column**: It is an index of the column where a non-zero element is located.
  - **Value**: The value of the non-zero element is located at the index (row, column).

	0	1	2	3
0	0	4	0	5
1	0	0	3	6
2	0	0	2	0
3	2	3	0	0
4	0	0	0	0

Row	Column	Value
0	1	4
0	3	5
1	2	3
1	3	6
2	2	2
3	0	2
3	1	3

Sparse Matrix: Linked List Representation

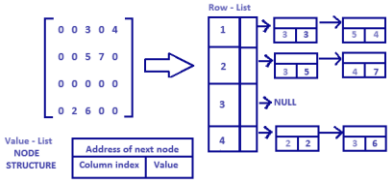
- A linear linked list is used to represent the sparse matrix. Each node of the list consists of four fields:
  - **Row**: Row: An index of row where a non-zero element is located.
  - **Column**: An index of column where a non-zero element is located.
  - **Value**: Value of the non-zero element which is located at the index (row, column).
  - **Next Node**: It stores the address of the next node.



 U1.22

Sparse Matrix: List of List Representation

- One list is used to represent the rows, and each row contains the list of triples:
  - **Column**: An index of column where a non-zero element is located.
  - **Value**: Value of the non-zero element.
  - **Address of Next Node** : It stores the address of the next non-zero element.



 U1.23

Linear Search

- Searching is a process of finding a value in a list of values.
- Linear search is a very simple search algorithm.
- In this type of search, a sequential search is made over all items one by one.
- Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.
- It has a time complexity of O(n), which means the time is linearly dependent on the number of elements, which is not bad, but not that good too.

 U1.24

Linear Search: Step-by-Step Process

- **Step 1:** Read the element to be searched from the user
- **Step 2:** Compare, the element to be searched with the first element in the list.
- **Step 3:** If both are matched, then display "Given element found" and terminate the search process.
- **Step 4:** If both are not matched, then compare search element with the next element in the list.
- **Step 5:** Repeat steps 3 and 4 until the search element is compared with the last element in the list.
- **Step 6:** If the last element in the list is also not matched, then display "Element not found!" and terminate the function.

Linear Search: Working Example

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

search element **12**

Step 1:

search element (12) is compared with first element (65)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Linear Search: Working Example

Step 2:

search element (12) is compared with next element (20)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 3:

search element (12) is compared with next element (10)

	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

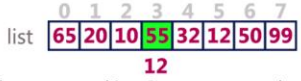
12

Both are not matching. So move to next element

Linear Search: Working Example

Step 4:

search element (12) is compared with next element (55)



Both are not matching. So move to next element

Step 5:

search element (12) is compared with next element (32)

Both are not matching. So move to next element

Linear Search: Working Example

Step 6:

search element (12) is compared with next element (12)

Both are matching. So we stop comparing and display element found at index 5.

Binary Search

- Binary search is a fast search algorithm with run-time complexity of O(log n).
- This search algorithm works on the principle of divide and conquer.
- For this algorithm to work properly, the data collection should be in the sorted form.

Binary Search: Step-by-Step Process

- **Step 1:** Read the element to be searched from the user.
- **Step 2:** Find the middle element in the sorted list.
- **Step 3:** Compare, the search element with the middle element in the sorted list.
- **Step 4:** If both are matched, then display "Given element found!" and terminate the search process.
- **Step 5:** If both are not matched, then check whether the search element is smaller or larger than middle element.
- **Step 6:** If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for the left sub-list of the middle element.
- **Step 7:** If the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for the right sub-list of the middle element.
- **Step 8:** Repeat the same process until we find the search element in the list or until the sub-list contains only one element.
- **Step 9:** If that element also doesn't match with the search element, then display "Element not found in the list!" and terminate the function.

 U1.31

Binary Search: Working Example

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

search element **12**

Step 1:

search element (12) is compared with middle element (50)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

12

Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

 U1.32

Binary Search: Working Example

Step 2:

search element (12) is compared with middle element (12)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

12

Both are matching. So the result is "Element found at index 1"

 U1.33

Program Code for Binary Search

```
low=0;
high=n-1;
while(low<=high) {
        mid=(low+high)/2;
        if(item<a[mid])
                high=mid-1;
        else if(item>a[mid])
                low=mid+1;
        else if(item==a[mid]) {
                printf("Item Found");
                break;
        }
        else {
                printf("Not Found");
        }
}
```


Selection Sort: Step-by-Step Process

- **Step 1:** Select the first element of the list (i.e., element at first position in the list).
- **Step 2:** Compare the selected element with all other elements in the list.
- **Step 3:** For every comparison, if any element is smaller than selected element (for ascending order), then these two are swapped.
- **Step 4:** Repeat the same procedure with next position in the list till the entire list is sorted.

- Complexity: $O(n^2)$

Selection Sort: Working Example

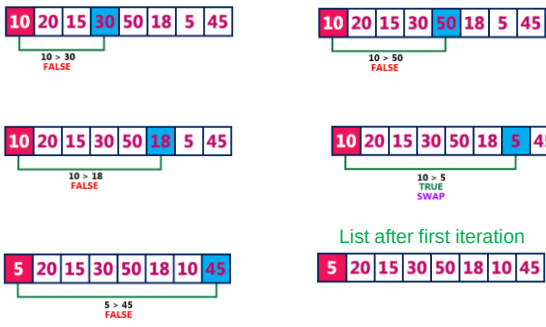
- Consider the following unsorted list of elements:

Iteration 1:

Select the **first** element of the list,

- compare it with all other elements in the list, and
- whenever we found a smaller element than the element at first position then swap those two elements.

Selection Sort: Working Example



List after first iteration

5 20 15 30 50 18 10 45

Selection Sort: Working Example

Iteration 2:

Select the **second** position element of the list,

- compare it with all other elements in the list, and
- whenever we found a smaller element than the element at second position then swap those two elements.

List after second iteration: 5 10 20 30 50 18 15 45

Iteration 3:

List after third iteration: 5 10 15 30 50 20 18 45

Selection Sort: Working Example

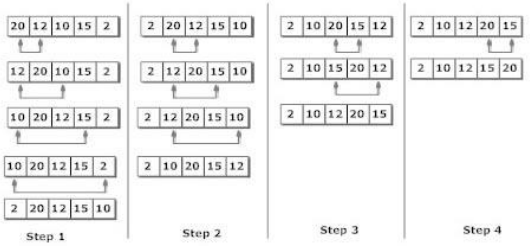
List after fourth iteration: 5 10 15 18 50 30 20 45

List after fifth iteration: 5 10 15 18 20 50 30 45

List after sixth iteration: 5 10 15 18 20 30 50 45

List after seventh iteration: 5 10 15 18 20 30 45 50

Selection Sort: Working Example



Program Code for Selection Sort

```
for(i=0; i<size; i++)
{
    for(j=i+1; j<size; j++)
    {
        if(list[i] > list[j])
        {
            temp=list[i];
            list[i]=list[j];
            list[j]=temp;
        }
    }
}
```


Revised Program Code for Selection Sort

```
for(i=0; i<n-1; i++)
{
    int min = i; //Consider the first element as minimum.
    for(j=i+1; j<n; j++)
    {
        if(a[j] < a[min])
        {
            min = j;
        }
    }
    if(min != i)
    {
        swap(a[i], a[min]);
    }
}
```

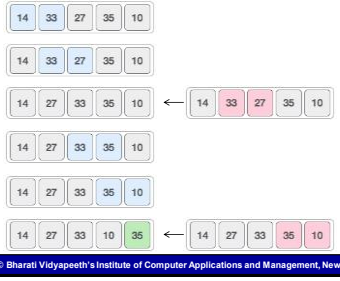
Bubble Sort: Step-by-Step Process

- **Step 1:** Select the first element of the list (i.e., element at first position in the list).
- **Step 2:** Compare the current element with next element of the list.
- **Step 3:** If the current element is greater than the next element (for ascending order), then these two are swapped.
- **Step 4:** If the current element is less than the next element, move to the next element.
- **Step 5:** Repeat from Step 1.
- Complexity: $O(n^2)$

Bubble Sort: Working Example

- Consider the following unsorted list of elements:

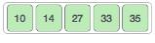
Iteration 1:



After Iteration 2

After Iteration 3

After Iteration 3



Program Code for Bubble Sort

```
for(i=1;i<n;i++) {
      for(j=0;j<n-i;j++) {
            if(a[j]>a[j+1])
            {
                  temp=a[j];
                  a[j]=a[j+1];
                  a[j+1]=temp;
            }
      }
}
```


Insertion Sort: Step-by-Step Process

- **Step 1:** Assume that first element in the list is in sorted portion of the list and remaining all elements are in unsorted portion.
- **Step 2:** Consider first element from the unsorted list and insert that element into the sorted list in order specified.
- **Step 3:** Repeat the above process until all the elements from the unsorted list are moved into the sorted list.
- Complexity: $O(n^2)$

Insertion Sort: Working Example

- Consider the following unsorted list of elements:

15	20	10	30	50	18	5	45

- Assume that the sorted portion of the list is empty and all elements in list are in unsorted portion, as shown below:

Sorted | Unsorted

15	20	10	30	50	18	5	45

- Move the first element **15** from the unsorted portion to sorted portion.

Sorted | Unsorted

15	20	10	30	50	18	5	45

Insertion Sort: Working Example

- To move **20** from unsorted to sorted portion, compare **20** with **15** and insert it at correct position.

Sorted | Unsorted

15	20	10	30	50	18	5	45

- To move element **10** from unsorted portion to sorted portion, compare **10** with **20**, it is smaller so perform swapping. Then, compare **10** with **15**, again it is smaller so perform swapping.

Sorted | Unsorted

10	15	20	30	50	18	5	45

Insertion Sort: Working Example

- Similarly, an element from unsorted portion is retrieved and is compared with element in sorted portion and is inserted accordingly.

Sorted | Unsorted

10	15	20	30	50	18	5	45

Sorted | Unsorted

10	15	20	30	50	18	5	45

Sorted | Unsorted

10	15	18	20	30	50	5	45

 U1.49

Insertion Sort: Working Example

Sorted | Unsorted

5	10	15	18	20	30	50	45

Final Sorted List

5	10	15	18	20	30	45	50

 U1.50

Insertion Sort: Working Example

Step 1	12 **3** 1 5 8	Checking second element of array with element before it and inserting it in proper position. In this case, 3 is inserted in position of 12.
Step 2	3 12 **1** 5 8	Checking third element of array with elements before it and inserting it in proper position. In this case, 1 is inserted in position of 3.
Step 3	1 3 12 **5** 8	Checking fourth element of array with elements before it and inserting it in proper position. In this case, 5 is inserted in position of 12.
Step 4	1 3 5 12 **8**	Checking fifth element of array with elements before it and inserting it in proper position. In this case, 8 is inserted in position of 12.

 U1.51

Program Code for Insertion Sort

```
for(i=1; i<n; i++)
{
    temp = data[i];
    j = i-1;
    while(temp<data[j] && j>=0)
    {
        data[j+1] = data[j];
        j = j-1;
    }
    data[j+1]=temp;
}
```


Shell Sort

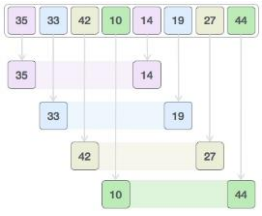
- In Insertion Sort, a large number of swaps/shifts are performed to sort the elements.
- Shell sort is an efficient sorting algorithm and is based on Insertion Sort.
- This algorithm avoids large shifts as in case of Insertion Sort, if the smaller value is to the far right and has to be moved to the far left.
- Shell Sort compares items that lie far apart which allows elements to move faster to the front of the list.

Shell Sort: Algorithm Working

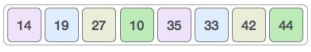
1. Divide the list into sub-lists using interval **Floor(N/2^k)**.
   - Shell Sequence **(Floor(N/2^k))**
2. Short sub-lists using Insertion Sort.
3. Repeat until complete list is sorted.

Shell Sort: Working

- Let the list of elements be: **35**, **33**, **42**, **10**, **14**, **19**, **27**, **44**
- Gap/Interval = Floor($8/2^1$) = **4**
- Sub-lists: {35, 14}, {33, 19}, (42, 27}, and {10, 44}
- Short sub-lists using Insertion Sort.



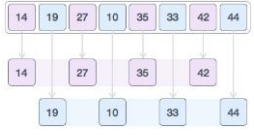
After this step, array becomes



 U1.55

Shell Sort: Working

- In next phase, the interval becomes Floor($8/2^2$) = **2**
- Then, we take interval of 2 and this gap generates two sub-lists {14, 27, 35, 42} and {19, 10, 33, 44}.
- Short sub-lists using Insertion Sort.



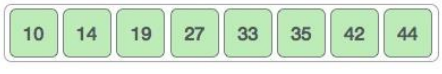
After this step, array becomes

14, 10, 27, 19, 35, 33, 42, 44

 U1.56

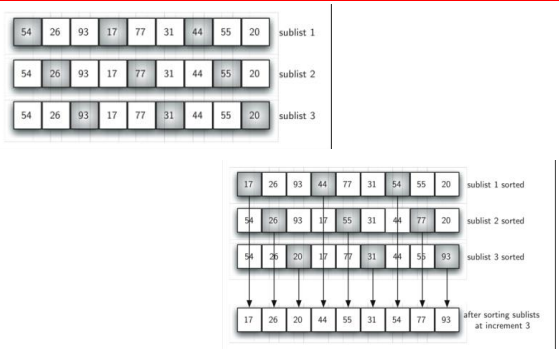
Shell Sort: Working

- In next phase, the interval becomes Floor($8/2^3$) = **1**
- Finally, sort (using Insertion Sort) the rest of the array using interval of value 1.



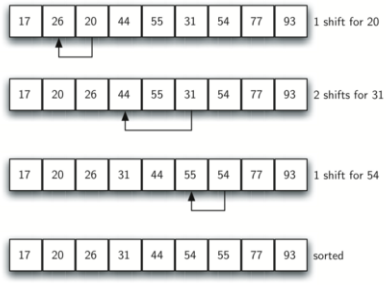
 U1.57

Shell Sort: Example



Shell Sort: Example

Sorting by using interval of value 1 (using Insertion Sort)



Program Code for Shell Sort

```
for(gap = n/2; gap >= 1; gap = gap/2) {
	for(j = gap; j < n ; j++) {
		for(i = j-gap; i >= 0; i = i - gap) {
			if(a[i+gap] > a[i]) {
				break;
			}
			else {
				swap(a[i+gap], a[i])
			}
		}
	}
}
```

Radix Sort

- A list of numbers is sorted based on the digits of individual numbers.
- Sorting is performed from least significant digit to the most significant digit.
- The number of passes required are equal to the number of digits present in the largest number of the list.
  - Example: If the largest number has 3 digits, then the list will be sorted in 3 passes.

 U1.61

Radix Sort: Algorithm

1. Define 10 queues, each representing a bucket for each digit from 0 to 9.
2. Consider the least significant digit of each number in the list which is to be sorted.
3. Insert each number into their respective queue based on the least significant digit.
4. Group all the numbers from queue 0 to queue 9 in the order they have inserted into their respective queues.
5. Repeat from step 2 until all the numbers are grouped based on the most significant digit.

 U1.62

Radix Sort: Example

Consider the following list of unsorted integer numbers

82, 901, 100, 12, 150, 77, 55 & 23

Step 1 - Define 10 queues each represents a bucket for digits from 0 to 9.

 U1.63

Radix Sort: Example

Step 2 - Insert all the numbers of the list into respective queue based on the Least significant digit (once placed digit) of every number.

82, 901, 100, 12, 150, 77, 55 & 23

Pass - 1

Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

100, 150, 901, 82, 12, 23, 55 & 77

 U1.64

Radix Sort: Example

Step 3 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Tens placed digit) of every number.

100, 150, 901, 82, 12, 23, 55 & 77

Pass - 2

Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

100, 901, 12, 23, 150, 55, 77 & 82

 U1.65

Radix Sort: Example

Step 4 - Insert all the numbers of the list into respective queue based on the next Least significant digit (Hundres placed digit) of every number.

100, 901, 12, 23, 150, 55, 77 & 82

Pass - 3

Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

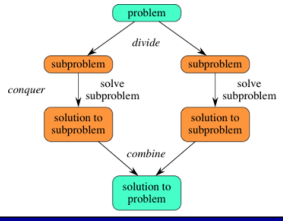
12, 23, 55, 77, 82, 100, 150, 901

List got sorted in the incresing order.

 U1.66

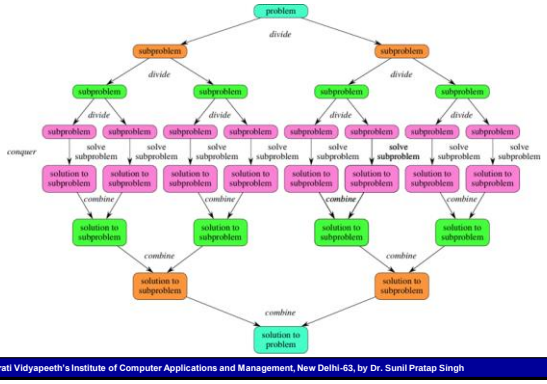
Divide and Conquer

1. **Divide** the problem into multiple small problems.
2. **Conquer** the sub-problems by solving them. The idea is to break down the problem into atomic sub-problems, where they are actually solved.
3. **Combine** the solutions of the sub-problems to find the solution of the actual problem.



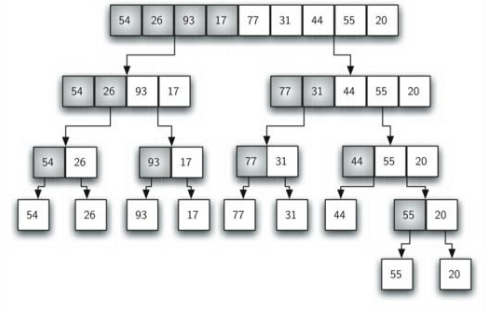
 U1.67

Divide and Conquer



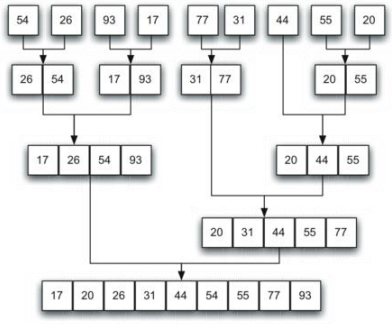
 U1.68

Merge Sort: Working



 U1.69

Merge Sort: Working



Algorithm/Code for Merge Sort

```
mergeSort(list, lower, upper)
{
    if(lower < upper)
    {
        mid = (lower + upper)/2;
        mergeSort(list, lower, mid)
        mergeSort(list, mid+1, upper)
        merge(list, lower, mid, upper)
    }
}
```


Code for Merging in Merge Sort

```
merge(a, lower, mid, upper)
{
    int i, j, k, b[n];
    i = lower;
    j = mid+1;
    k = lower;
    while (i <= mid && j <= upper)
    {
        if (a[i] <= a[j])
        {
            b[k] = a[i];
            i++; k++;
        }
        else
        {
            b[k] = a[j];
            j++; k++;
        }
    }
    //continued … … …
```

```
//If any element is left in sub-lists
    if (i > mid)
    {
        while (j <= upper)
        {
            b[k] = a[j];
            j++;
            k++;
        }
    else
    {
        while (i <= mid)
        {
            b[k] = a[i];
            i++;
            k++;
        }
    }
//We may copy the items to main array
}
```

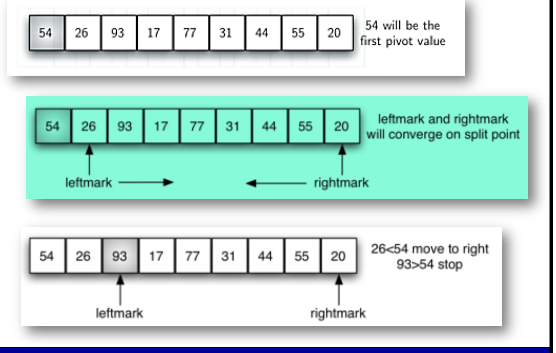
Quick Sort: Procedure/Process

- The quick sort uses **divide and conquer** to gain the same advantages as the merge sort, while not using additional storage.
- A quick sort first selects a value, which is called the pivot value. The **actual position** where the pivot value belongs in the final sorted list, commonly called the **split point**, is used to divide the list for subsequent calls to the quick sort.
- Partitioning begins by locating two position markers—let's call them leftmark and rightmark — **at the beginning** and **end of the remaining items in the list**.

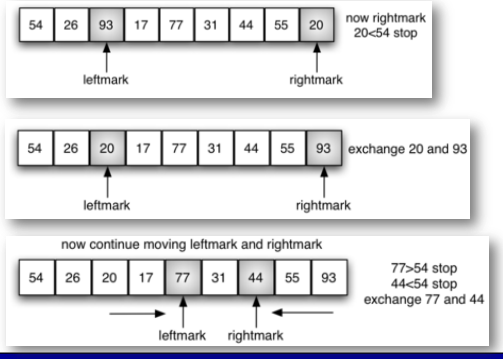
Quick Sort: Procedure/Process

- Begin by incrementing leftmark until we locate a value that is greater than the pivot value.
- Then decrement rightmark until we find a value that is less than the pivot value.
- At the point where rightmark becomes less than leftmark, we stop.
  - The position of rightmark is now the split point.
  - The pivot value can be exchanged with the contents of the split point and the pivot value is now in place.

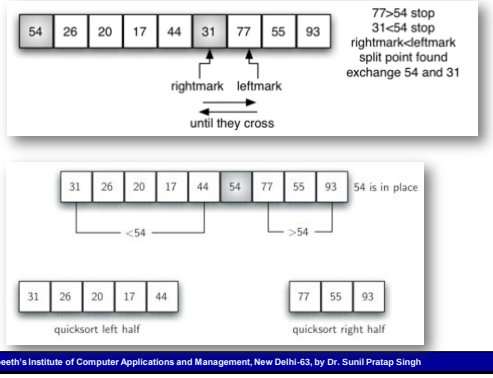
Quick Sort: Working



Quick Sort: Working



Quick Sort: Working



Algorithm/Code for Quick Sort

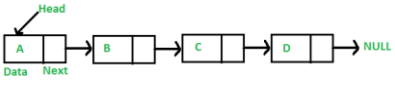
```
quickSort(list, low, high)
{
    int pivot;
    if ( high > low ) //Termination Condition
    {
        pivot = partition(a, low, high);
        quickSort(a, low, pivot-1);
        quickSort(a, pivot+1, high);
    }
}
```

Linked List

- Linked List is a linear collection of data elements, called *nodes*.
- The linear order is given by pointers.
- Each node is divided into two or more parts.
- A Linked List can be of following types:
  - Linear Linked List (One-Way List)
  - Doubly Linked List (Two-Way List)
  - Circular Linked List

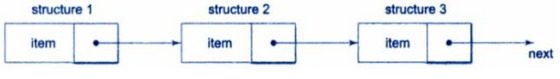
Linear Linked List

- **Linked List** is a linear data structure which consists of a series of nodes.
- Unlike arrays, linked list elements are not stored at contiguous location; the elements are linked using pointers.



- **Advantages:**
  - **Dynamic data structure**: can grow or shrink dynamically
  - **Ease of insertion/deletion**: insertion and deletion are efficient
  - **Implementation of other complex data structures**
- **Drawbacks:**
  - **No random access**: access to an arbitrary data item is time-consuming
  - **Requires more memory**: extra space is required for pointer

Implementation of Linear Linked List




```
//Structure Representation for Node of a Linear Linked List
struct node
{
    int item;
    struct node *next;
};
```

Insertion of a Node in Linear Linked List

```
//Insertion of a Node in the Beginning of a Linear Linked List
void insertBegin(int item) {
 NODE *node;
 node=(NODE*)malloc(sizeof(NODE));
 node->data=item;
 if(start==NULL) {
   node->next=NULL;
 }
 else {
   node->next=start;
 }
 start=node;
}
```


Insertion of a Node in Linear Linked List

```
//Insertion of a Node in the End of a Linear Linked List
void insertEnd(int item) {
  NODE *node,*pos;
  node=(NODE*)malloc(sizeof(NODE));
  node->data=item;
  node->next=NULL;
  if(start==NULL) {
    start=node;
  }
  else {
    pos=start;
    while(pos->next!=NULL) {
      pos=pos->next;
    }
    pos->next=node;
  }
}
```


Insertion of a Node in Linear Linked List

```
//Insertion of a Node at Specific Position of a Linear Linked List
void insertPosition(int item,int p) {
  NODE *node,*pos;
  int count=1;
  pos=start;
  while(count<p)
    if(count==(p-1)) {
      node=(NODE*)malloc(sizeof(NODE));
      node->data=item;
      node->next=pos->next;
      pos->next=node;
      break;
    }
    else {
      pos=pos->next;
      count++;
    }
}
```

Deletion of a Node from Linear Linked List

```
//Deletion of a Node from the Beginning of a Linear Linked List
void deleteBegin() {
   NODE *node;
   if(start==NULL) {
       printf("\nUNDERFLOW");
       return;
   }
   else {
       node=start;
       start=start->next;
       printf("NODE DELETED %d ", node->data);
       free(node);
   }
}
```


Deletion of a Node from Linear Linked List

```
//Deletion of a Node from the End of a Linear Linked List
void deleteEnd() {
    NODE *node,*pos;
    if(start==NULL) {
         printf("\nUNDERFLOW");
         return; }
    else if(start->next==NULL) {
        node=start;
        start=NULL;
        printf("\nNODE DELETED %d", node->data);
        free(node); }
    else {
        pos=start;
        node=start->next;
        while(node->next!=NULL) {
            pos=node;
            node=node->next; }
   pos->next=NULL;
   printf("\nNODE DELETED %d", node->data);
   free(node);
 }
}
```


Deletion of a Node from Linear Linked List

```
//Deletion of a Node from Specific Position of a Linear Linked List
void deletePosition(int p) {
    NODE *node,*pos;
    pos=start;
    int count=0;
    while(count<p) {
        if(count==(p-1)) {
            node=pos->next;
            pos->next=node->next;
            free(node);
            break;
        }
        else {
            pos=pos->next;
            count++;
        }
    }
}
```

Traversal of a Linear Linked List

```
//Traversal of a Linear Linked List
void travers() {
    NODE *pos;
    pos=start;
    if(pos==NULL) {
        printf("\nLIST IS EMPTY");
    }
    else {
        printf("\nLIST ELEMENTS: ");
        while(pos!=NULL) {
            printf("%d  ",pos->data);
           pos=pos->next;
        }
    }
}
```


Polynomials Addition using Linear Linked List

```
void addPoly(NODE **start, NODE *p, NODE *q) {
  NODE *node = (NODE *)malloc(sizeof(NODE));
  node->next = NULL;
  *start = node;
  while(p && q) { //LOOP WHILE BOTH LISTS HAVE VALUES
    if(p->pow > q->pow) {
      node->pow = p->pow;    node->coe = p->coe;    p = p->next;
    }
    else if(p->pow < q->pow) {
      node->pow = q->pow;    node->coe = q->coe;    q = q->next;
    }
    else {
      node->pow = p->pow;    node->coe = p->coe + q->coe;    p = p->next;    q = q->next;
    }
    if(p && q) { //GROW THE LINKED LIST ON CONDITION
       node->next = (NODE *)malloc(sizeof(NODE));
       node = node->next;
       node->next = NULL;
    }
  }
```


Polynomials Addition using Linear Linked List

```
  //continued…
  while(p || q) {
    NODE *newNode = (NODE *)malloc(sizeof(NODE));
    node->next = newNode;
    node = newNode;
    node->next = NULL;
    if(p) {
      node->pow = p->pow;    node->coe = p->coe;    p = p->next;
    }
    if(q) {
      node->pow = q->pow;    node->coe = q->coe;    q = q->next;
    }
  }
}
```

Stack

- A Stack is a linear data structure.
- It is a list in which insertion of new data item and deletion of existing data item is done from one end, known as **Top** of Stack.
- Stack is also called **LIFO** (Last-in-First-out) type of list.
  - The last inserted element will be the first to be deleted from Stack.
- Example:
  - Some of you may eat biscuits (or poppins). If you assume only one side of the cover is torn and biscuits are taken out one by one. This is called **poping**. If you want to preserve some biscuits for some time later, you will put them back into the pack through the same torn end. This is called **pushing**.

 U1.91

Operations on Stack

- Push
  - The process of inserting a new element to the top of stack is called Push operation.
  - In case the list is full, no new element can be accommodated, it is called Stack Overflow condition.
- Pop
  - The process of deleting an element from top of stack is called Pop operation.
  - If there is no any element in the Stack and Pop is performed then this will result in Stack Underflow condition.

 U1.92

Implementation of Stack

- Static Implementation
  - It is achieved using Array
- Dynamic Implementation
  - It is achieved using Linked List

 U1.93

Implementation of Stack using Array

- Push Operation

```
int stack[10],top = -1;
void push(int x)
{
    top = top+1;
    stack[top] = x;
}
```

- Pop Operation

```
int pop()
{
    int temp;
    temp = stack[top];
    top = top-1;
    return temp;
}
```


Implementation of Stack using Linked List

- Structure Definition

```
struct stack
{
    int data;
    struct node *next;
};
typedef struct stack STACK;
STACK *top;
```

- Required Functions

```
void create();
int isempty();
int isfull()
void push(int);
int pop();
void display();
```


Some Applications of Stack

- Reverse of String/Number
- Recursion (Recursive Function)
- Expression Conversion
- Expression Evaluation
- Syntax Parsing
- Undo-mechanism in an Editor
- etc.

Expressions and their Types

- An expression is defined as a number of operands or data items combined using several operators.
- The way to write arithmetic expression is known as a notation.
- An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression.
- These notations are:
  - Infix Notation
  - Prefix Notation
  - Postfix Notation

 U1.97

Infix Notation

- **Infix Notation** is what we come across in our general mathematics.
- In **Infix Notation**, operators are written *in-between* the operands.
- Example:
  - Expression to add two numbers A and B is written as:

A + B

- **Infix Notation** needs precedence of the operators and we sometimes use bracket () to override these rules.

 U1.98

Prefix Notation

- In **Prefix Notation**, operators are written *before* the operands.
- This is also known as *polish notation* in the honor of the Polar mathematician (Jan Lukasiewicz) who developed this notation.
- Example:
  - Expression to add two numbers A and B is written as:

+ A B

 U1.99

Postfix Notation

- In **Postfix Notation**, operators are written *after* the operands.
- This is also known as *reverse polish notation*.
- Example:
  - Expression to add two numbers A and B is written as:

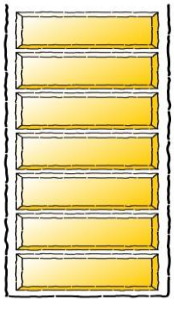
    A B +
- It is most suitable for computer to calculate any expression as there is no need for operator precedence and other rules.
- It is the universally accepted notation for designing ALU of the CPU, **therefore important for us to study**.

Conversion from Infix to Postfix Notation

- While there are tokens to be read from expression, read the token.
- If the token is an operand, then insert it to output.
- If the token is an operator and if the Top of Stack is not any operator then push the operator to stack.
- If the token is an operator O1:
  - While there is an operator, O2 at top of stack (O2 is Top), and
  - If precedence of O1 > O2
    - ✓ Push O1 on to Stack (now O1 is Top)
  - Else if precedence of O1 <= O2
    - ✓ Pop O2 to the output and Push O1 onto Stack
- If the token is a left parenthesis, the Push it onto the Stack.
- If the token is a right parenthesis:
  - Until the token at Top is a left parenthesis, Pop operators off the Stack onto the output
  - Pop the left parenthesis from Stack
- If the token at Top is an operator, Pop and insert it onto output.

Step-by-Step Example: Infix to Postfix Conversion

Stack



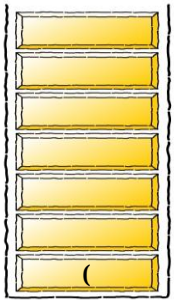
Infix

(a + b - c) * d – (e + f)

Postfix

Step-by-Step Example: Infix to Postfix Conversion

Stack



Infix

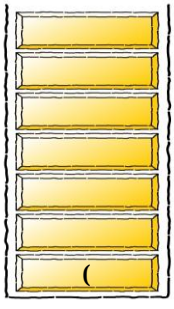
a + b - c) * d – (e + f)

Postfix

 U1.103

Step-by-Step Example: Infix to Postfix Conversion

Stack



Infix

+ b - c) * d – (e + f)

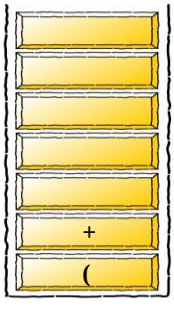
Postfix

a

 U1.104

Step-by-Step Example: Infix to Postfix Conversion

Stack



Infix

b - c) * d – (e + f)

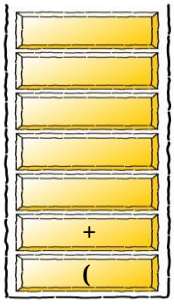
Postfix

a

 U1.105

Step-by-Step Example: Infix to Postfix Conversion

Stack



Infix

- c) * d – (e + f)

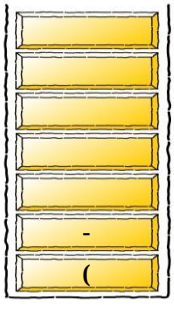
Postfix

a b

 U1.106

Step-by-Step Example: Infix to Postfix Conversion

Stack



Infix

c) * d – (e + f)

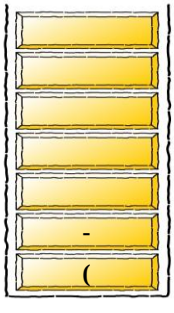
Postfix

a b +

 U1.107

Step-by-Step Example: Infix to Postfix Conversion

Stack



Infix

) * d – (e + f)

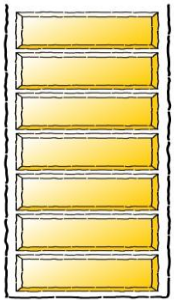
Postfix

a b + c

 U1.108

Step-by-Step Example: Infix to Postfix Conversion

Stack



Infix

* d – (e + f)

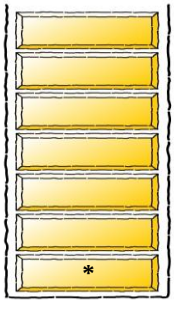
Postfix

a b + c -

 U1.109

Step-by-Step Example: Infix to Postfix Conversion

Stack



Infix

d – (e + f)

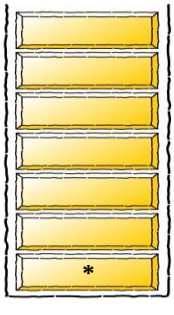
Postfix

a b + c -

 U1.110

Step-by-Step Example: Infix to Postfix Conversion

Stack



Infix

– (e + f)

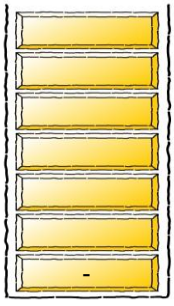
Postfix

a b + c - d

 U1.111

Step-by-Step Example: Infix to Postfix Conversion

Stack



Infix

(e + f)

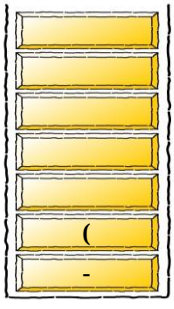
Postfix

a b + c – d *

 U1.112

Step-by-Step Example: Infix to Postfix Conversion

Stack



Infix

e + f)

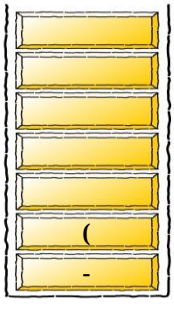
Postfix

a b + c – d *

 U1.113

Step-by-Step Example: Infix to Postfix Conversion

Stack



Infix

+ f)

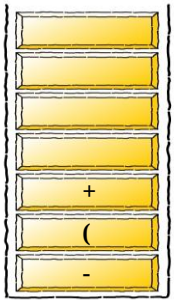
Postfix

a b + c – d * e

 U1.114

Step-by-Step Example: Infix to Postfix Conversion

Stack



Infix

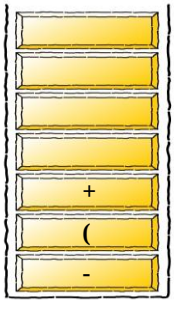
f)

Postfix

a b + c – d * e

Step-by-Step Example: Infix to Postfix Conversion

Stack



Infix

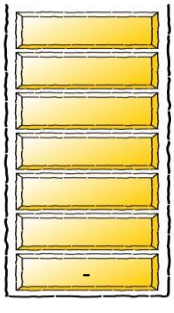
)

Postfix

a b + c – d * e f

Step-by-Step Example: Infix to Postfix Conversion

Stack

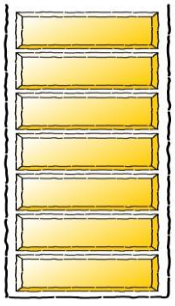


Infix

Postfix

a b + c – d * e f +

Step-by-Step Example: Infix to Postfix Conversion



Infix

Postfix

a b + c – d * e f + -

Infix to Postfix Conversion: Example

Convert ((A – (B + C)) * D) ^(E + F) to Postfix form.

SYMBOL	POSTFIX STRING	STACK	REMARKS
(		(	
(		((	
A	A	((	
-	A	((-	
(	A	((-(	
B	AB	((-(	
+	AB	((-(+	
C	ABC	((-(+	
)	ABC+	((-	
)	ABC+-	(	
*	ABC+-	(*	
D	ABC+-D	(*	
)	ABC+-D*		
↑	ABC+-D*	↑	
(	ABC+-D*	↑(	
E	ABC+-D*E	↑(	
+	ABC+-D*E	↑(+	
F	ABC+-D*EF	↑(+	
)	ABC+-D*EF+	↑	
End of string	ABC+-D*EF+↑	The input is now empty. Pop the output symbols from the stack until it is empty.	

Conversion from Infix to Prefix Notation

- The conversion process is almost same according to Postfix notation.
  - The only change from Postfix form is that **traverse the expression from right to left** and **the operator is placed before the operand** rather than after them.
- Convert the expression A * B + C / D into Prefix notation.
  - Answer: + * A B/ C D

Conversion from Postfix to Infix Notation

1. Scan the postfix expression from left to right.
2. If the scanned symbol is an operand, then push it onto the stack.
3. If the scanned symbol is an operator, pop two symbols from the stack and create it as a string by placing the operator in between the operands and push it onto the stack.
4. Repeat steps 2 and 3 till the end of the expression.

Postfix to Infix Conversion: Example

Convert the expression A B C * D E F ^ / G * - H + * to Infix notation.

Symbol	Stack	Remarks
A	A	Push A
B	A B	Push B
C	A B C	Push C
*	A (B*C)	Pop two operands and place the operator in between the operands and push the string.
D	A (B*C) D	Push D
E	A (B*C) D E	Push E
F	A (B*C) D E F	Push F
^	A (B*C) D (E^F)	Pop two operands and place the operator in between the operands and push the string.
/	A (B*C) (D/(E^F))	Pop two operands and place the operator in between the operands and push the string.
G	A (B*C) (D/(E^F)) G	Push G
*	A (B*C) ((D/(E^F))*G)	Pop two operands and place the operator in between the operands and push the string.
-	A ((B*C) - ((D/(E^F))*G))	Pop two operands and place the operator in between the operands and push the string.
H	A ((B*C) - ((D/(E^F))*G)) H	Push H
*	A (((B*C) - ((D/(E^F))*G)) * H)	Pop two operands and place the operator in between the operands and push the string.
+	(A + (((B*C) - ((D/(E^F))*G)) * H))	
End of string	The input is now empty. The string formed is infix.	

Infix to Postfix Conversion: Questions

- A * B + C
- A + B * C
- A * (B + C)
- A – B + C
- A * B ^ C + D
- A * (B + C * D) + E
- (A + B) * C / D + E ^ F /G → A B + C * D / E F ^ G / + (Answer)
- A + (B * C – (D / E ^ F) * G) * H → A B C * D E F ^ / G * - H * + (Answer)
- A - B / (C * D ^ E) → A B C D E ^ * / - (Answer)

Evaluation of Postfix Expression

Expression: 4 5 6 * +

Step	Input	Operation	Stack	Calculation
1	4	Push	4	
2	5	Push	4 5	
3	6	Push	4 5 6	
4	*	Pop 2 Elements and Evaluate	4	6 * 5 = 30
5		Push Result (30)	4 30	
6	+	Pop 2 Elements and Evaluate	Empty	4 + 30 = 34
7		Push Result (34)	34	
8		No More Elements (Pop)	Empty	34

Double Stack/Multistack

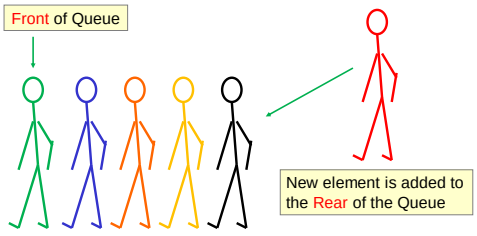
- Double stack means two stacks which are implemented using a single array.
- To prevent memory wastage, the two stacks are grown in opposite direction.
- The pointer Top1 and Top2 points to top-most element of Stack1 and Stack 2 respectively.
- Initially, Top1 is initialized to -1 and Top2 is initialized the size of array.
- As the elements are pushed into Stack1, Top1 is incremented.
- Similarly, as the elements are pushed into Stack2, Top2 is decremented.
- The array is full when Top1=Top2-1.
- Multistack means more than 2 stacks which are implemented using a single array.

Queue

- A Queue is a linear data structure.
- It is a list in which insertion of new data items is done from one end, called **Rear** end, and deletion of existing data item is done from other end, known as **Front** end of Queue.
- Queue is also called **FIFO** (First-in-First-out) type of list.
  - The first inserted element will be the first to be deleted from Queue.

Conceptual View of a Queue

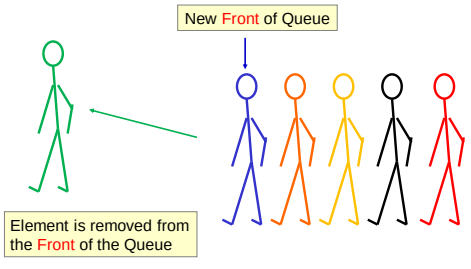
- **Inserting/Adding an Element in Queue**



 U1.127

Conceptual View of a Queue

- **Deleting/Removing an Element from Queue**



 U1.128

Applications of Queue

- Real World Examples
  - People on an Escalator or Waiting in a Line
  - Cars at a Gas Station
- Computer Science Examples
  - Print Queue
  - Keyboard Input Buffer
  - Queue of Network Data Packets
  - Queue of Processes
- Applications in Simulation Studies

 U1.129

Working of Queue

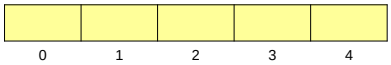
- Enqueue
  - The process of inserting a new element at the Back of queue is called Enqueue operation.
  - In case the list is full, no new element can be accommodated, it is called Queue Overflow condition.
- Dequeue
  - The process of deleting an element from Front of queue is called Dequeue operation.
  - If there is no any element in the queue and Dequeue is performed then this will result in Queue Underflow condition.

 U1.130

Working of Queue

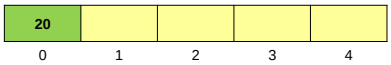
- Empty Queue

F = -1
R = -1



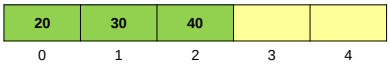
- Queue after inserting 1 elements

F = 0
R = 0



- Queue after inserting 2 more elements

F = 0
R = 2

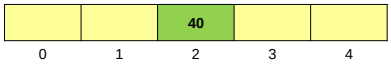


 U1.131

Working of Queue

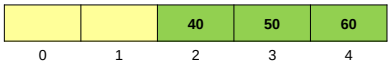
- Queue after deleting 2 elements

F = 2
R = 2



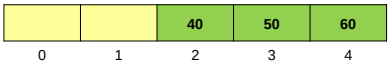
- Queue after inserting 2 elements

F = 2
R = 4



- What if we want to insert 1 more element?

F = 2
R = 4



  - Insertion not possible because **R = 4**.

 U1.132

Implementation of Queue

- Static Implementation
  - It is achieved using Array
- Dynamic Implementation
  - It is achieved using Linked List

Implementation of Queue using Array

- Insertion (Enqueue)

```
#define Max 5
#define Nil -1
int queue[Max];
int front, rear;

void enqueue(int x) {
    if(front == Nil) {
        front = rear = 0;
    }
    else {
        rear = rear + 1;
    }
    queue[rear] = a;
}
```


Implementation of Queue using Array

- Deletion (Dequeue)

```
int dequeue(int x)
{
    int temp = queue[front];
    if(rear == front)
    {
        front = rear = Nil;
    }
    else
    {
        front = front + 1;
    }
    return temp;
}
```

Implementation of Queue using Array

- Traversal (Display/Print Elements)

```
void display()
{
    int i;
    for(i = front; i <= rear; i++)
    {
        printf("%d  ",queue[i]);
    }
}
```


Implementation of Queue using Linked List

- Structure Definition

```
struct queue
{
    int data;
    struct node *next;
};
typedef struct queue QUEUE;
QUEUE *start;
```

- Required Functions

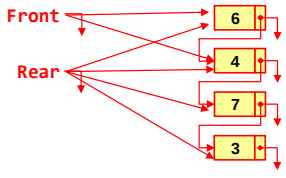
```
void create();
int isempty();
int isfull()
void enqueue(int);
int dequeue();
void display();
```


Implementation of Queue using Linked List

- In Queue, insertion takes place at Rear end.
  - This is similar to inserting an element at the **end** of a Linked List.
- In Queue, deletion takes place at Front end.
  - This is similar to deleting an element from the **front** of a Linked List.
- Therefore, Linked List has application in implementing a Queue.

Queue using Linked List

```
Enqueue(6);
Enqueue(4);
Enqueue(7);
Enqueue(3);
Dequeue();
```



Structure for a Queue using Linked List

```
struct queue
{
    int data;
    struct node *next;
};
typedef struct queue QUEUE;
QUEUE *start;
```


Insertion in a Queue using Linked List

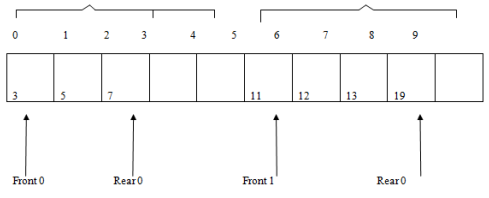
```
void enqueue(int a)
{
    QUEUE *node,*pos;
    node = (QUEUE*)malloc(sizeof(QUEUE));
    node->data = a;
    node->next = NULL;
    if(start == NULL)
        start = node;
    else
    {
        pos = start;
        while(pos->next != NULL)
            pos = pos->next;
        pos->next = node;
    }
}
```

Deletion from a Queue using Linked List

```
int dequeue()
{
    QUEUE *node;
    int item;
    if(start != NULL)
    {
        node = start;
        item = node->data;
        start = start->next;
        free(node);
        return (item);
    }
    else
        return 0;
}
```


Multiqueue

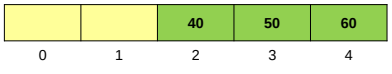
- Maintaining two or more queues in the same array refers to Multiqueue.
- Double Queue:



Limitations of Linear Queue (With Array)

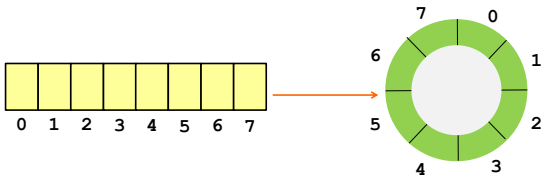
- Consider the following representation of Queue:

F = 2
R = 4



- Even after having 2 unoccupied cells, we are unable to insert data elements because insertion is done at Rear end, and Rear is pointing to last position of the Queue.
- Solution?
  - **Circular Queue**

Circular Queue



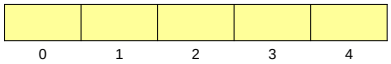
Circular Queue

- Circular Queue is a linear data structure
- The operations are performed based on FIFO (First In, First Out) principle
- The last position is connected back to the first position to make a circle

Working of Circular Queue

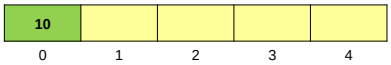
- Empty Queue

F = -1
R = -1



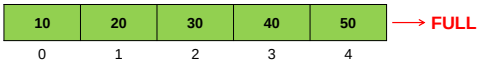
- Queue after inserting 1 elements

F = 0
R = 0



- Queue after inserting 4 more elements

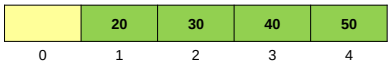
F = 0
R = 4



Working of Circular Queue

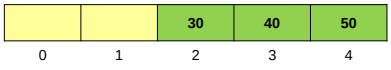
- Queue after deleting 1 elements

F = 1
R = 4



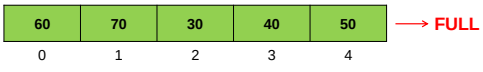
- Queue after deleting 1 more elements

F = 2
R = 4



- Queue after inserting 2 more element

F = 2
R = 1



⟶ FULL

 U1.148

Implementing Circular Queue using Array

```
//Method 1 to Check Queue Overflow
int isFull() {
    if((front == rear + 1) || (front == 0 && rear == SIZE-1))
        return 1;
    else
        return 0;
}
```

```
//Method 2 to Check Queue Overflow
int isFull() {
    if((rear+1) % SIZE == front)
        return 1;
    else
        return 0;
}
```

```
int isEmpty() {
    if(front == -1)
        return 1;
    else
        return 0;
}
```

 U1.149

Insertion in Circular Queue using Array

```
void insert(int item)
{
    if(isFull())
        printf("OVERFLOW!");
    else
    {
        if(front == -1)
        {
            front = 0;
        }
        rear = (rear + 1) % SIZE;
        queue[rear] = item;
    }
}
```

 U1.150

Deletion in Circular Queue using Array

```
int delete() {
    int item;
    if(isEmpty()) {
        printf("UNDERFLOW!");
        return(-1);
    }
    else {
        item = queue[front];
        if(front == rear) {
            front = -1;
            rear = -1;
        }
        else {
            front = (front + 1) % SIZE;
        }
        printf("ITEM DELETD %d", item);
        return (element);
    }
}
```


Traversal of a Circular Queue using Array

```
int travers()
{
    int i;
    if(isEmpty())
        printf("UNDERFLOW!");
    else
    {
         printf("ITEMS: ");
        for(i = front; i!=rear; i=(i+1)%SIZE) {
            printf("%d ",queue[i]);
        }
        printf("%d ",queue[i]);
    }
}
```


Deque (Double Ended Queues)

- Insertion and Deletion are performed from both the ends, i.e.,
  - we can insert/delete elements from the **REAR** end or from the **FRONT** end
- Four operations are performed:
  - Insertion of an element at the **REAR** end of Queue.
  - Deletion of an element from the **FRONT** end of Queue.
  - Insertion of an element at the **FRONT** end of Queue.
  - Deletion of an element from the **REAR** end of Queue.
- There are two types of Deques:
  - **Input-restricted Deque**: Deletion can be performed from both ends (**FRONT** and **REAR**) while Insertion can be done at one end (**REAR**)
  - **Output-restricted Deque**: Deletion can be performed from one end (FRONT) while Insertion can be done at both ends (**REAR** and **FRONT**)

Implementation of Deque using Array

- Methods to be implemented for Deque

```
int isEmpty()
int isFull()
void insertFront(int x)
void insertRear(int x)
int deleteFront()
int deleteRear()
```


Bibliography

- E. Horowitz and S. Sahani, "Fundamentals of Data Structures in C"
- Mark Allen Weiss, "Data Structures and Algorithm Analysis in C"
- R. S. Salaria, "Data Structure & Algorithms Using C"
- Schaum's Outline Series, "Data Structure"
- http://www.btechsmartclass.com/ (Online)

Data and File Structures

(MCA-102)

Unit – 2

[Tree]

by

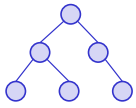
Dr. Sunil Pratap Singh
(Assistant Professor, BVICAM, New Delhi)

2021

 U2.1

Tree

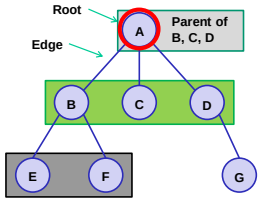
- A Tree is a non-linear data structure which organizes data in a hierarchical structure.
- In Tree, every individual element is called a *node* which stores the data value.
- Each node is connected by an *edge* to another node.
- Example:
  - Tree with 6 nodes.



 U2.2

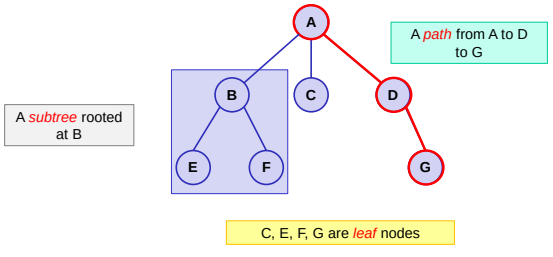
Tree Terminology

- Root
- Rooted Tree
- Degree of a Node
- Degree of a Tree
- Leaf Node / Terminal Node
- Non-Terminal Node / Internal Node
- Siblings
- Level
- Height of a Tree
- Path
- Subtree
- Forest



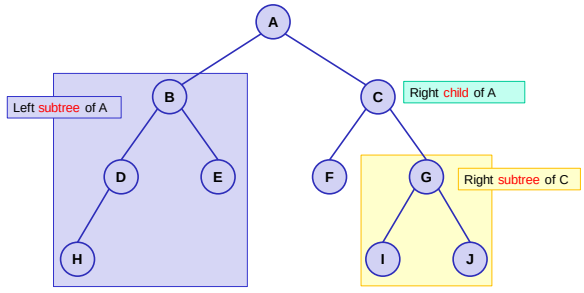
 U2.3

Tree Terminology: Example



 U2.4

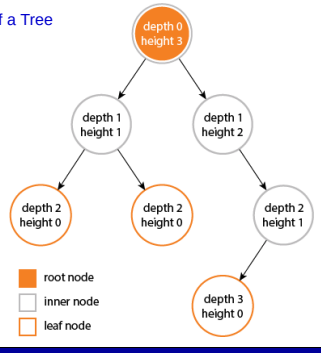
Tree Terminology: Example



 U2.5

Tree Terminology: Example

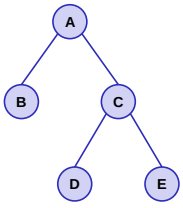
Depth and Height of a Tree



 U2.6

Binary Tree

- In a normal tree, every node can have any number of children.
- A binary tree is tree in which each node can have a maximum of 2 children.
- Example:

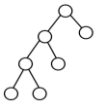


 U2.7

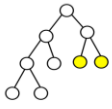
Types of Binary Tree

- Strictly (Full) Binary Tree
  - If every node has either 0 or 2 children, a binary tree is called **Strictly Binary Tree**.
- Complete Binary Tree
  - A **Complete Binary Tree** is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible
- Perfect Binary Tree
  - A binary tree is **Perfect Binary Tree** in which all internal nodes have two children and all leaves are at same level.

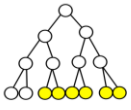
Strictly Binary Tree



Complete Binary Tree



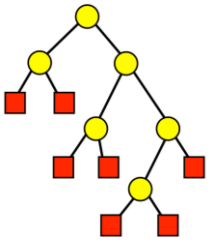
Perfect Binary Tree



 U2.8

Extended Binary Tree

- The full binary tree obtained by adding dummy nodes (external nodes) to a binary tree is called **Extended Binary Tree**.



 U2.9

Properties of Binary Tree

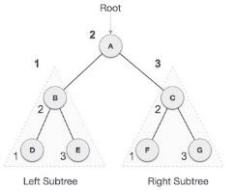
- A binary tree with n internal nodes has exactly $n + 1$ external nodes.
- For any non-empty binary tree with n_0 leaf nodes and n_2 nodes of degree 2, $n_0 = n_2 + 1$
- The maximum number of nodes on level i of a binary tree is 2^i, $i \geq 0$.
- The maximum number of nodes in a binary tree of height k is $2^{k+1} - 1$.
- The height of a binary tree with n nodes is at least $\lceil \log_2(n + 1) - 1 \rceil$ and at most $n - 1$.
- The number of distinct binary trees with n nodes is $\frac{(2n)!}{(n+1)!\, n!}$.

Binary Tree Traversal

- Traversal is a process to visit all the nodes of a tree and may print their values too.
- There are three ways which we use to traverse a tree:
  - In-order Traversal
  - Pre-order Traversal
  - Post-order Traversal

In-order Traversal

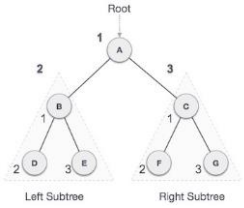
- In this traversal method, the left subtree is visited first, then the root and later the right sub-tree.
- We should always remember that every node may represent a subtree itself.



Output: D – B – E – A – F – C – G

Pre-order Traversal

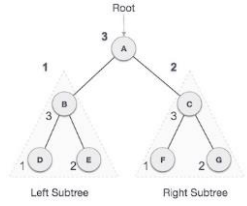
- In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



Output: A – B – D – E – C – F – G

Post-order Traversal

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



Output: D – E – B – F – G – C – A

Inorder Traversal with Stack (Non-Recursive)

1) Create an empty Stack S.

2) Initialize current node as root.

3) **Push** the current node to S and set current = current->left until current is NULL.

4) If current is NULL and Stack is not empty, then

   a. **Pop** the top item from Stack.

   b. Print the popped item, set current = poppedItem->right

   c. Go to step 3.

5) If current is NULL and Stack is empty then we are done.

Preorder Traversal with Stack (Non-Recursive)

1) Create an empty Stack S.

2) Push the root node to S.

3) While the Stack is not empty, then

a. **Pop** the top item from S and print.

b. **Push** the poppedItem->right item to S.

c. **Push** the poppedItem->left item to S.

Postorder Traversal with Stack (Non-Recursive)

1) Create two empty Stacks **S1** and **S2**.

2) Push the root node to **S1**.

3) While the Stack **S1** is not empty, then

a. **Pop** the top item from **S1** and Push it into **S2**.

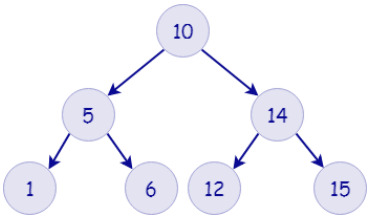
b. **Push** the poppedItem->left item to **S1**.

c. **Push** the poppedItem->right item to **S1**.

4) Pop out all the items from Stack **S2** and Print.

Example

Find **Inorder**, Preorder and Postorder traversal sequence for the given tree.

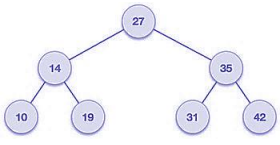


Representation of Binary Tree

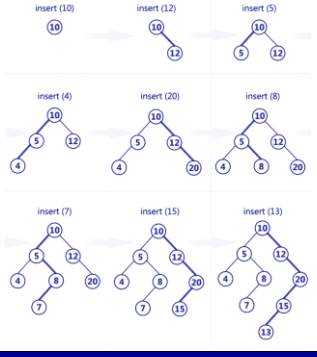
- Using Array
- Using Linked List

Binary Search Tree

- A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties:
  - The left sub-tree of a node has a key less than or equal to its parent node's key.
  - The right sub-tree of a node has a key greater than to its parent node's key.



Insertion in Binary Search Tree



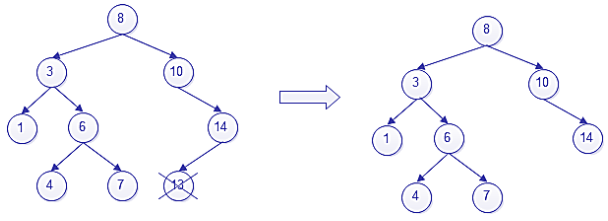
Deletion in Binary Search Tree

- To delete an element from, there are three cases:
  - The node to be deleted has no children.
  - The node to be deleted has 1 child node.
  - The node to be deleted has 2 child nodes.

 U2.22

Deletion in Binary Search Tree

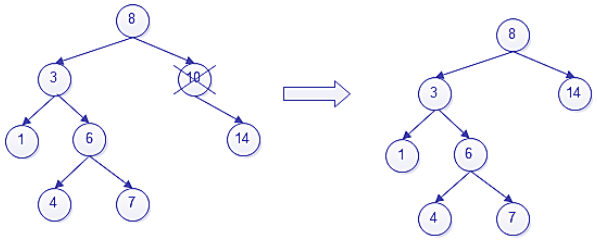
- The node to be deleted has no children.
  - Simply delete the node.



 U2.23

Deletion in Binary Search Tree

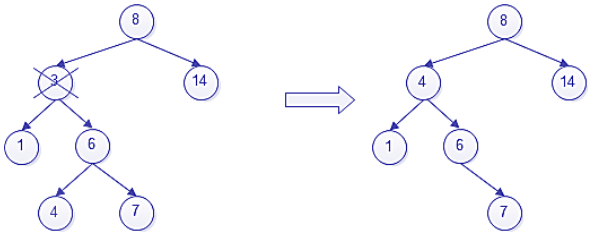
- The node to be deleted has 1 child node.
  - Replace the node with its child node and delete it.



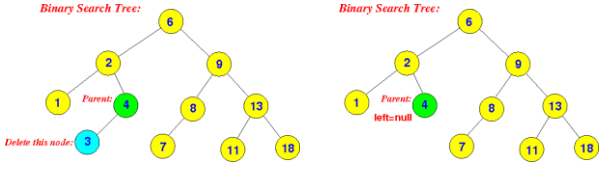
 U2.24

Deletion in Binary Search Tree

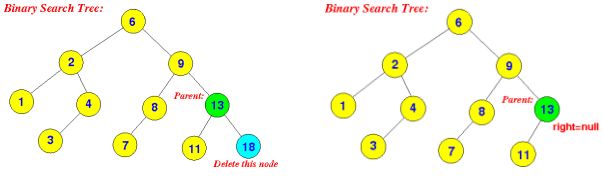
- The node to be deleted has 2 child nodes.
  - Find its in-order successor node, and replace it with in-order successor, then delete it.



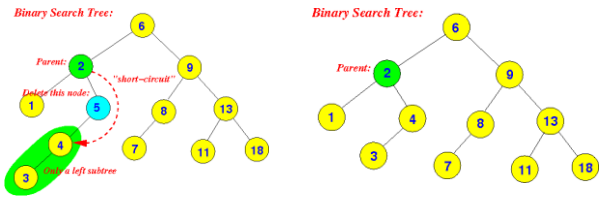
Deletion in Binary Search Tree



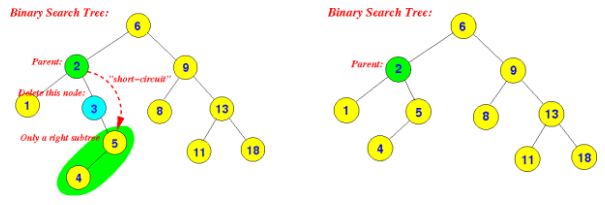
Deletion in Binary Search Tree



Deletion in Binary Search Tree

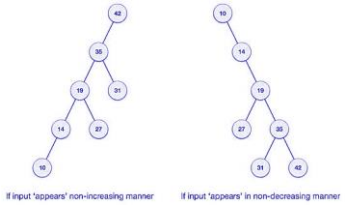


Deletion in Binary Search Tree



AVL Tree

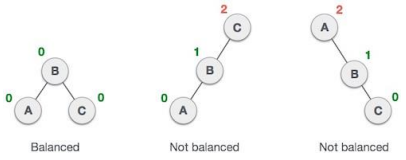
- What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this:



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

AVL Tree

Balance Factor = height(left-subtree) – height(right-subtree)
or
Balance Factor = height(right-subtree) – height(left-subtree)



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

How Rotation Works to Balance the Tree

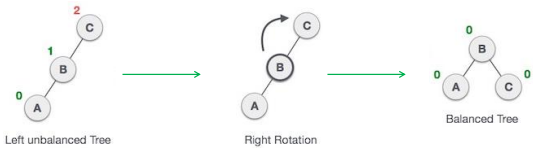
- Let the newly inserted node be **w**:
  1. Perform standard BST insert for **w**.
  2. Starting from **w**, travel up and find the first unbalanced node. Let **z** be the first unbalanced node, **y** be the child of **z** that comes on the path from **w** to **z** and **x** be the grandchild of **z** that comes on the path from **w** to **z**.
  3. Re-balance the tree by performing appropriate rotations on the sub-tree rooted with **z**.
     - There can be 4 possible cases that needs to be handled as **x**, **y** and **z** can be arranged in 4 ways.

Insertion in AVL Tree

- 4 Possible Cases for Unbalanced Node:
  1. Left-Left Case: x is the left child of y and y is the left child of z
  2. Left-Right Case: x is the right child of y and y is the left child of z
  3. Right-Left Case: x is the left child of y and y is the right child of z
  4. Right-Right Case: x is the right child of y and y is the right child of z

Example: Right Rotation (LL Case)

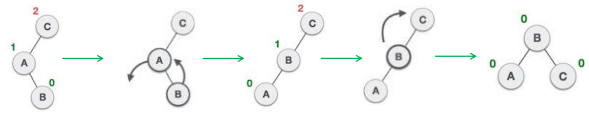
- If a tree becomes unbalanced, when a node is inserted in the left subtree of the left subtree, then we perform a single right rotation.
- Example: Insert C, B and A



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

Example: Left Right Rotation (LR Case)

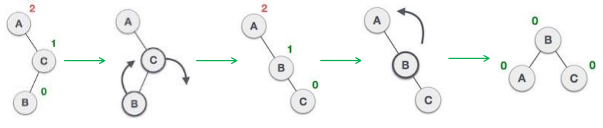
- The LR Rotation is combination of left rotation followed by right rotation.
- Example: Insert C, A and B



- In our example, node C has become unbalanced as B is inserted in the right subtree of C's left subtree.
  - Perform the left rotation on the left subtree of C. This makes A, the left subtree of B.
  - Perform the right-rotation on the tree, making B the new root node of this subtree.

Example: Right Left Rotation (RL Case)

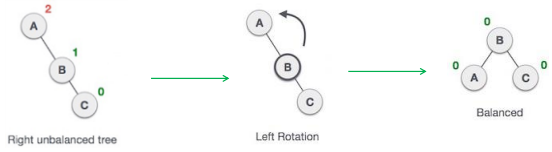
- The RL Rotation is combination of right rotation followed by left rotation.
- Example: Insert A, C and B



- In our example, node A has become unbalanced as B is inserted in the left subtree of A's right subtree.
  - Perform the right rotation along C. This makes C, the right subtree of B.
  - Perform the left rotation on the tree, making B the new root node of this subtree.

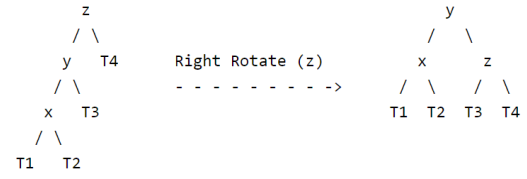
Example: Left Rotation (RR Case)

- If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation.
- Example: Insert A, B and C

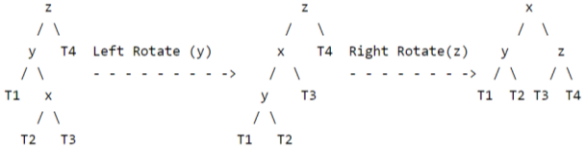


- In our example, node A has become unbalanced as C is inserted in the right subtree of A's right subtree. We performed the left rotation by making A as the left-subtree of B.

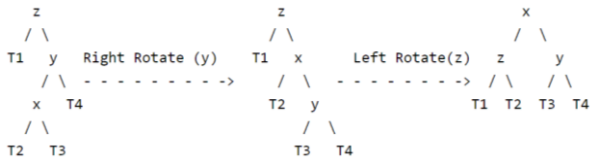
Example: Complex Situation of LL Case



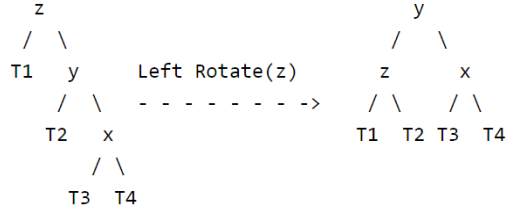
Example: Complex Situation of LR Case



Example: Complex Situation of RL Case



Example: Complex Situation of RR Case

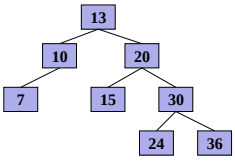


Example: Drawing AVL Tree

- Draw AVL Tree by inserting the values: 15, 20, 24, 10, 13, 7, 30, 36

Example: Drawing AVL Tree

- AVL Tree by inserting the values: 15, 20, 24, 10, 13, 7, 30, 36

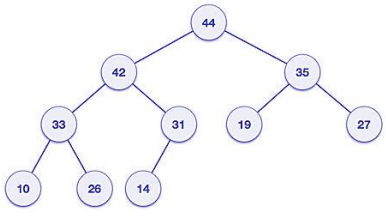


Heap

- Heap is a special balanced binary tree with special characteristics.
- Heap can be defined as a collection of keys (data elements) which satisfies the following characteristics:
  - Ordering: Nodes must be arranged in a order according to values.
  - Structural: All levels in a heap must full, except last level and nodes must be filled from left to right strictly (Complete Binary Tree)
- There are two types of heap:
  - Max Heap
  - Min Heap

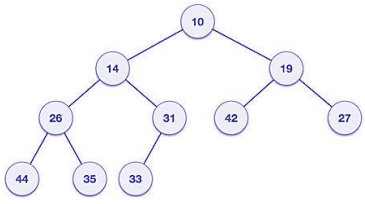
Max Heap

- When the value of the root node is greater than or equal to either of its children, it is called Max Heap.
- Max heap is used for **Heap Sort**



Min Heap

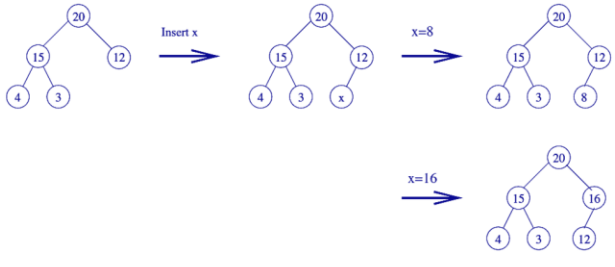
- When the value of the root node is less than or equal to either of its children, it is called Min Heap.
- Min heap is used to implement **Priority Queue**.
- It may also be used to implement **Heap Sort**.



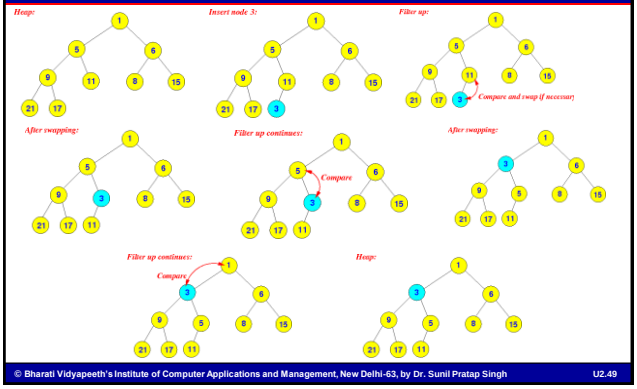
Max Heap Construction: Algorithm

- First increase the heap size by 1, so that it can store the new element.
- Insert the new element at the end of the Heap.
- This newly inserted element may distort the properties of Heap for its parents. So, in order to keep the properties of Heap, heapify this newly inserted element following a bottom-up approach.

Max Heap Insertion: Example



Min Heap Insertion: Example

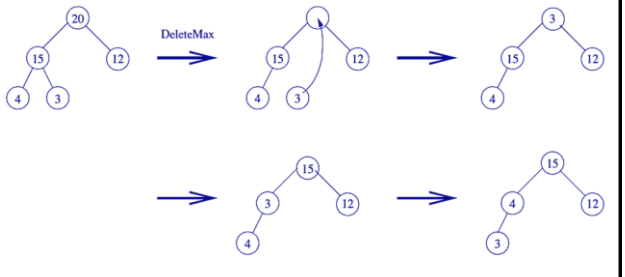


Max Heap Deletion: Algorithm

Deletion of Root Node

- Replace the root or element to be deleted by the last element.
- Delete the last element from the Heap.
- Since, the last element is now placed at the position of the root node. So, it may not follow the heap property. Therefore, heapify the last node placed at the position of root.

Max Heap Deletion (Root): Example

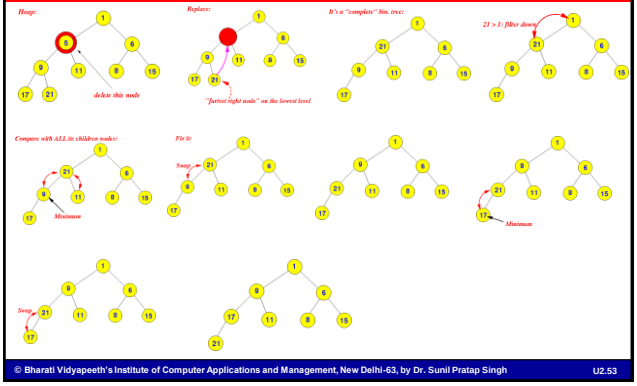


Max Heap Deletion (Specific): Algorithm

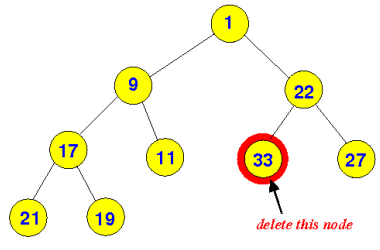
Deletion of a Specific Node

- Delete a node from the array.
- Replace the deletion node with the "farthest right node" on the lowest level of the Binary Tree
- Heapify (fix the heap):
  - If the value in replacement node is greater then its parent node, filter the replacement node UP the binary tree.
  - Else Filter the replacement node DOWN the binary tree

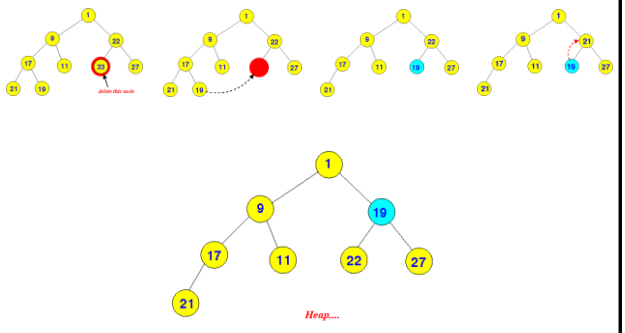
Min Heap Deletion (Specific): Example



Min Heap Deletion: Example



Min Heap Deletion: Example

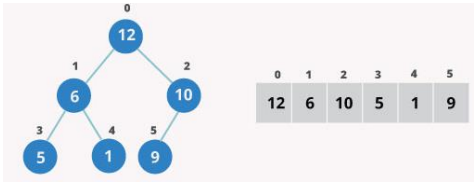


Heap Sort: Step-by-Step Process

- In max-heaps, largest element will always be at the root. Heap Sort uses this property of heap to sort the array.
- Heap sort is an in-place algorithm, i.e., does not use any extra space, like merge sort.
- Complexity: O(n log n)
- Procedure:
  1) Build a max-heap of elements in array.
  2) Swap the root element with last element of array.
  3) Reduce the size of the heap by 1 and heapify the root element so that we have highest element at root.
  4) Repeat the steps 2 and 3, until all the items of the list are sorted.

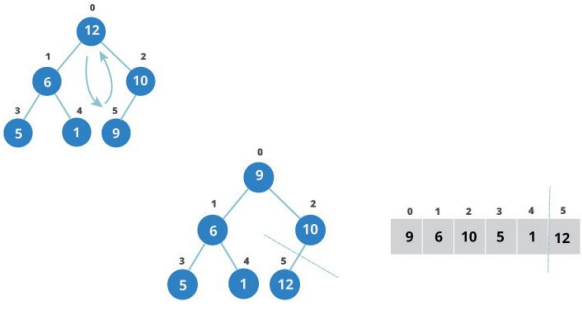
Heap Sort: Working Example

- Let the elements of the array are: 1, 12, 9, 5, 6, 10. For these given elements, the max heap is constructed as follows:



Heap Sort: Working Example

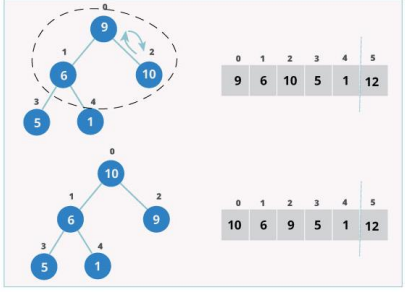
- Swapping the root with last element, and decreasing the size of heap:



 U2.58

Heap Sort: Working Example

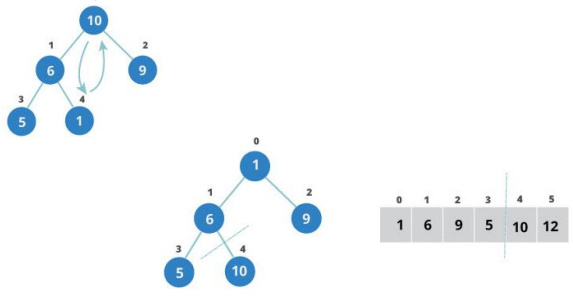
- Heapify the root element:



 U2.59

Heap Sort: Working Example

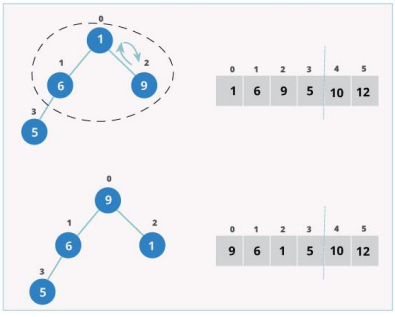
- Swapping the root with last element, and decreasing the size of heap:



 U2.60

Heap Sort: Working Example

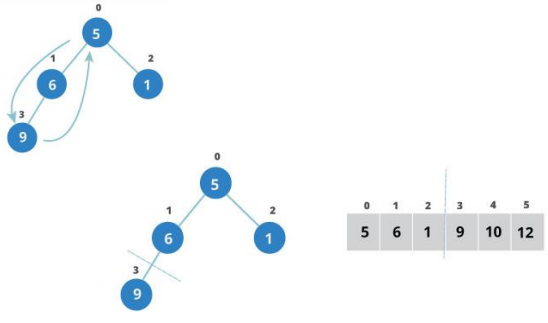
- Heapify the root element:



 U2.61

Heap Sort: Working Example

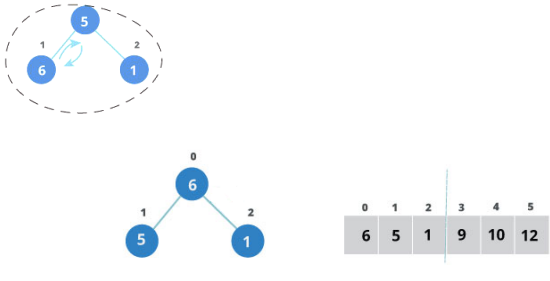
- Swapping the root with last element, and decreasing the size of heap:



 U2.62

Heap Sort: Working Example

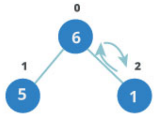
- Heapify the root element:

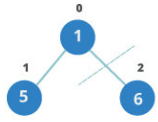


 U2.63

Heap Sort: Working Example

- Swapping the root with last element, and decreasing the size of heap:



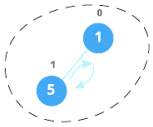


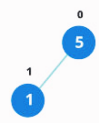
0	1	2	3	4	5
1	5	6	9	10	12

 U2.64

Heap Sort: Working Example

- Heapify the root element:



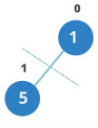


0	1	2	3	4	5
5	1	6	9	10	12

 U2.65

Heap Sort: Working Example

- Swapping the root with last element, and decreasing the size of heap:



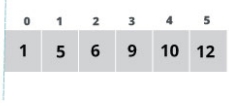
0	1	2	3	4	5
1	5	6	9	10	12

 U2.66

Heap Sort: Working Example

- Heapify the root element:

- Final sorted elements:

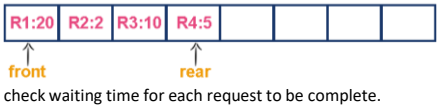


Program Code for Heap Sort

```
for(i=n-1; i>=0; i--)
{
    swap(a[0], a[i])
    heapify(a, 0, i)
}
```


Priority Queue

- Consider a networking application where four requests arrived to the queue in the order of R1 requires 20 units of time, R2 requires 2 units of time, R3 requires 10 units of time and R4 requires 5 units of time. Queue is as follows:



- Now, check waiting time for each request to be complete.
  - R1 : 20 units of time
  - R2 : 22 units of time
  - R3 : 32 units of time
  - R4 : 37 units of time
- **Average waiting time for all requests = (20+22+32+37)/4 ≈ 27 units of time**
- That means, if we use a normal queue data structure to serve these requests the average waiting time for each request is **27 units of time**.

Priority Queue

- Now, consider another way of serving these requests. If we serve according to their required amount of time.
- Then, the waiting time for each request to be complete will be as follows:
  - R2 : 2 units of time
  - R4 : 7 units of time
  - R3 : 17 units of time
  - R1 : 37 units of time
- **Average waiting time for all requests = (2+7+17+37)/4 ≈ 15 units of time**
- **Priority queue is a variant of queue data structure in which insertion is performed in the order of arrival and deletion is performed based on the priority.**

Types of Priority Queue

- Max Priority Queue
- Min Priority Queue

Max Priority Queue

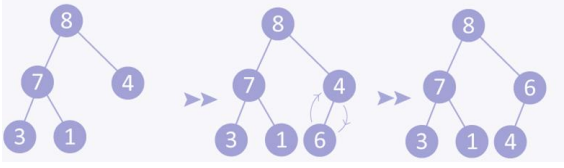
- In max priority queue, elements are inserted in the order in which they arrive the queue and always maximum value is removed first from the queue.
- For example, assume that we insert in order 8, 3, 2, 5 and they are removed in the order 8, 5, 3, 2.

Min Priority Queue

- In min priority queue, elements are inserted in the order in which they arrive the queue and always minimum value is removed first from the queue.
- For example, assume that we insert in order 8, 3, 2, 5 and they are removed in the order 2, 3, 5, 8.

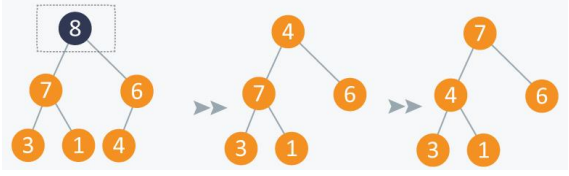
Insertion in Priority Queue

- Initially there are 5 elements in Priority Queue.
- Insert value 6.



Deletion in Priority Queue

- Extract Maximum



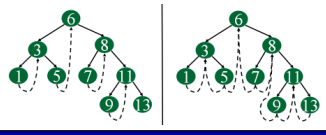
Threaded Binary Tree

- When a binary tree is represented using linked list representation, we use NULL pointer for nodes which do not have children.
- In any binary tree linked list representation, there are more number of NULL pointer than actual pointers.
- A. J. Perlis and C. Thornton proposed new binary tree called "Threaded Binary Tree", which make use of NULL pointer to improve its traversal processes.
- The idea of Threaded Binary Trees is to make in-order traversal faster and do it without recursion.
- To convert Binary Tree into Threaded Binary Tree, first find the in-order traversal of that tree.

 U2.76

Threaded Binary Tree

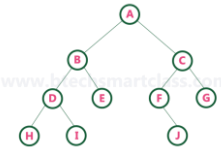
- **One Way Threading**:
  - Each node is threaded towards either the in-order predecessor or successor (left **OR** right) means all right null pointers will point to in-order successor **OR** all left null pointers will point to in-order predecessor.
- **Two Way Threading**:
  - Each node is threaded towards both, in-order predecessor and successor (left **AND** right) which means all right null pointers will point to in-order successor **AND** all left null pointers will point to in-order predecessor.



 U2.77

Threaded Binary Tree: Example

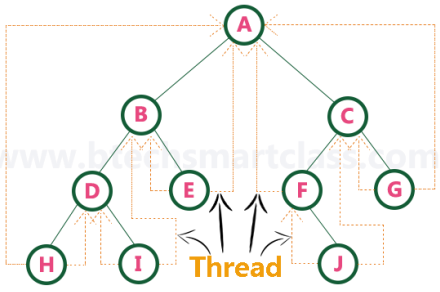
- In-Order : H D I B E A F J C G



- Left child pointers of nodes H, I, E, F, J and G are NULL.
- These NULLs are replaced by address of their in-order predecessor, respectively (I to D, E to B, F to A, J to F and G to C), but here the node H does not have its in-order predecessor, so it points to the root node A.
- Right child pointers of nodes H, I, E, J and G are NULL.
- These NULLs are replaced by address of their in-order successor, respectively (H to D, I to B, E to A, and J to C), but here the node G does not have its in-order successor, so it points to the root node A.

 U2.78

Threaded Binary Tree: Example



 U2.79

m-way Tree

- The concept of two-way search tree (BST) can be extended to create an m-way search tree. The m-way tree has following properties:
  - Each node has any number of children from 2 to M, i.e., all nodes have degree <= M, where $M >= 2$
  - Each node has keys (K_1 to K_n) and pointers to its children (P_0 to P_n), i.e., number of keys is one less than the number of pointers. The keys are ordered, i.e., $K_i < K_{i+1}$ for $1 <= i <n$
  - The subtree pointed by a pointer P_i has key values less than the key value of K_{i+1} for $1 <= i <n$
  - The subtree pointed by a pointer P_n has key values greater than the key value of K_n
  - All subtrees pointed by pointers P_i are m-way trees.

 U2.80

B-Tree

- B-Tree is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time.
- B-Tree was developed in the year of 1972 by Bayer and McCreight with the name **Height Balanced m-way Search Tree**.
- When data volume is large and does not fit in memory, an extension of the binary search tree to disk-based environment is the B-tree.
- Since the B-tree is always balanced (all leaf nodes appear at the same level), it is an extension of the balanced binary search tree.

 U2.81

B-Tree

- The B in B-Tree technically doesn't represent a word. However some common characteristics can be summarized with words that begin with B, which is most likely the origin of the name.
  - **Balanced** – this is a self balancing data structure, which means that performance can be guaranteed when B-Trees are utilized.
  - **Broad** – as opposed to binary search trees, which grow vertically, B-Trees expand horizontally, so saying that they are broad is a suitable description.
  - **Bayer** – lastly the creator of B-Trees was named Bayer Rudolf. In all actuality this is probably the reason why B-Trees got their name.

B-Tree vs. Binary Search Tree

Basis for Comparison	B-Tree	Balanced Binary Search Tree
Essential Constraint	A node can have at max M number of child nodes(where M is the order of the tree).	A node can have at max 2 number of subtrees.
Use	It is used when data is stored on disk.	It is used when data is stored on RAM.
Height of the Tree	$\log_M N$ (where M is the order of the M-way tree)	$\log_2 N$

B-Tree

- When the number of data elements (keys) are more, the data is read from disk in the form of blocks.
- Disk access time is very high compared to main memory access time.
- The main idea of using B-Trees is to **reduce** the number of disk accesses.
  - Most of the tree operations (search, insert, delete, max, min) require **O(h)** disk accesses where **h** is height of the tree.
  - Height of B-Trees is kept low by putting maximum possible keys in a B-Tree node.
  - Since each disk access exchanges a whole block of information between memory and disk rather than a few bytes, a node of the B-tree is expanded to hold more than two child pointers, up to the block capacity
  - Since **h** is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees (like AVL Tree).

B-Tree

- B-Trees are a good example of a data structure for external memory.
- B-Trees are commonly used in databases and files systems.
- Most database management systems have implemented the B-tree or its variants.

 U2.85

Properties of B-Tree

- A height-balanced m-way tree is called as B-Tree.
- A B-Tree of order m, has following properties:
  - The root has at least two children. If the tree contains only a root, i.e., it is a leaf node then it has no children.
  - Each internal node (except the root) has between $\lceil m/2 \rceil$ and m children.
    - Internal nodes stores up to $m - 1$ keys.
  - All paths from the root to leaves have the same length, i.e., all leaves are at the same level making it height balanced.
  - Leaves store between $\lceil m/2 \rceil - 1$ and m-1 data records

 U2.86

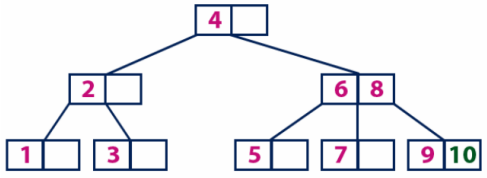
Insertion in B-Tree

- Inserting into a B-tree starts out by "find"ing the leaf in which to insert.
- If there is room in the leaf for another data item, then we're done.
- If the leaf already has m-1 items, then there's no room.
  - Split the overfull node in half and pass the middle (median) value up to the parent for insertion there.
  - If the value passed up to the parent causes the parent to be over-full, then it too splits and passes the middle value up to its parent.

 U2.87

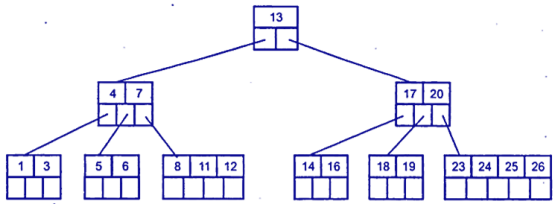
Example of B-Tree

- Construct a B-tree of order 3 by inserting numbers from 1 to 10



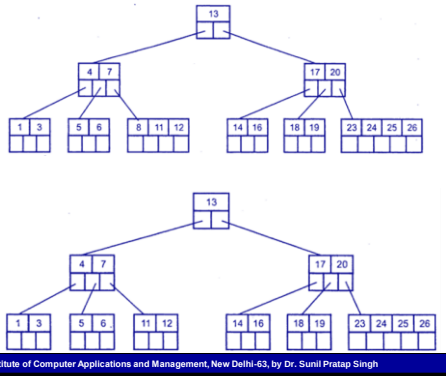
Example of B-Tree

- Construct a B-Tree of order 5 for following numbers:
  3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, 19



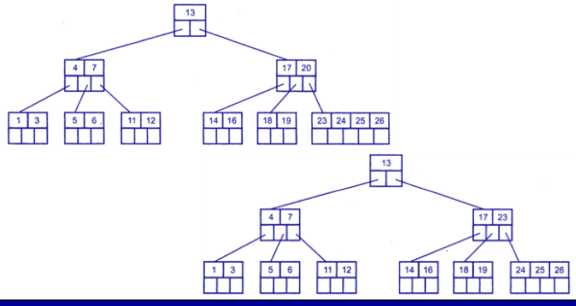
Deletion in B-Tree

- Delete 8



Deletion in B-Tree

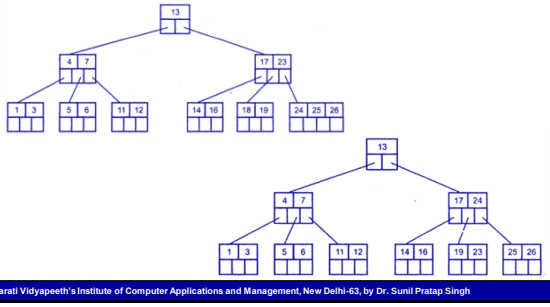
- Delete 20, which is not a leaf node so find its **successor** which is 23. Hence 23 will be moved up to replace 20.



 U2.91

Deletion in B-Tree

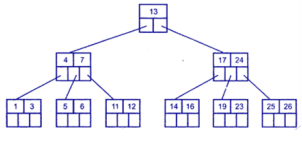
- Delete 18, which causes the node with **only one key**. The sibling node to immediate right has an extra key. In such case borrow a key from parent and move spare key of sibling to up.



 U2.92

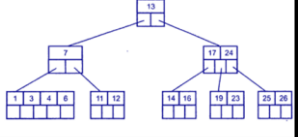
Deletion in B-Tree

- Delete 5. This node has no extra keys nor siblings to left or right. In such a situation, we can combine this node with one of the siblings. That means remove 5 and combine 6 with node 1, 3. To make tree balanced, we have to move parent's key down. Hence move 4 down.



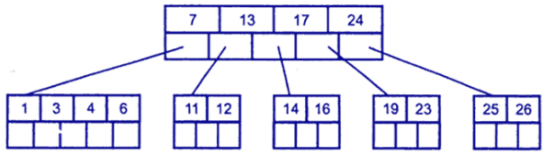
Hence, we need to combine 7 with 13 and 17, 24.

But, again internal node of 7 contains only 1 key, which is not allowed in B-Tree. We will then borrow a key from siblings but sibling has no spare key.



 U2.93

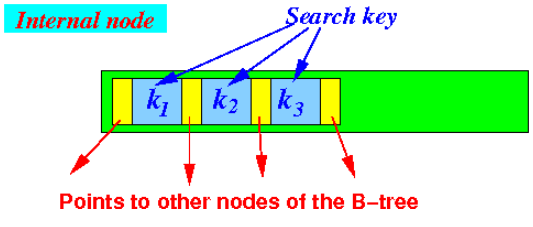
Deletion in B-Tree



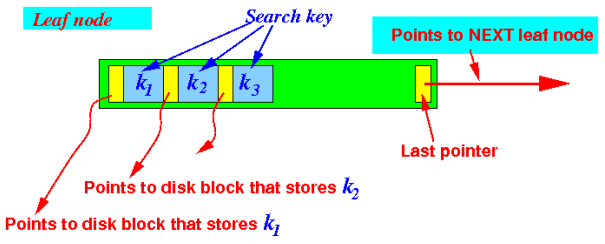
B+ Tree

- B+ Tree is a variation of basic **B-Tree**.
- Leaves are connected to form a Linked List of keys in sequential order.
- B+ Tree has two parts:
  - ***Index Set*** that constitutes **Internal Nodes**
  - ***Sequence Set*** that constitutes **Leaves**
- **The linked Leaves can be accessed sequentially in addition to accessing them directly.**
  - This allows for extremely efficient range queries.

B+ Tree: Structure of Internal Node

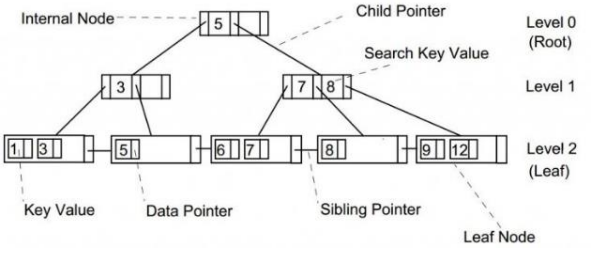


B+ Tree: Structure of Leaf Node



 U2.97

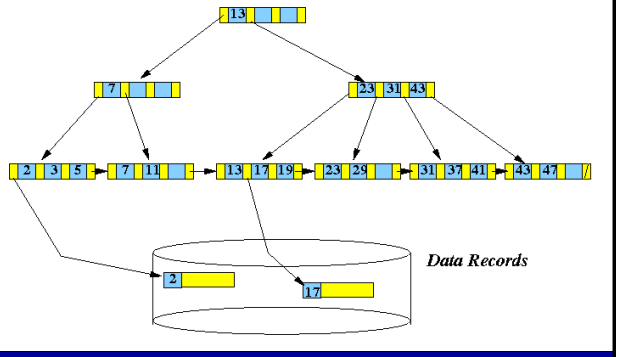
Structure of B+ Tree



B+ trees don't store data pointer in interior nodes, they are ONLY stored in leaf nodes.

 U2.98

Example of B+ Tree



 U2.99

B Tree vs. B^+ Tree

- The leaf nodes in a B-Tree are not linked.
- B^+ Trees do not store data pointer in interior nodes.
- In B Tree, Internal Nodes and Leaves, both, store the search keys.
- B^+ Tree is efficient due to traversal performed with sibling pointers.

 U2.100

B* Tree

- B*-tree is a variant of a B-tree that requires each internal node to be at least 2/3 full, rather than at least half full.

 U2.101

Bibliography

- E. Horowitz and S. Sahani, "Fundamentals of Data Structures in C"
- Mark Allen Weiss, "Data Structures and Algorithm Analysis in C"
- R. S. Salaria, "Data Structure & Algorithms Using C"
- Schaum's Outline Series, "Data Structure"

 U2.102

Data and File Structures

(MCA-102)

Unit – 3

[Graph]

by

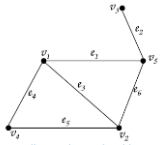
Dr. Sunil Pratap Singh
(Assistant Professor, BVICAM, New Delhi)

2021

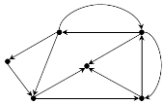
 U3.1

Graph

- A graph G = (V,E) consists of a finite set of vertices V = {v_1, v_2, . . . , v_n} and a finite set E of edges E = {e_1, e_2, . . . , e_m}.
- To each edge *e,* there corresponds a pair of vertices (*u, v*) where *e* is said to be *incident* on.
- A graph is said to be a directed graph (or digraph for short) if the vertex pair (u, v) associated with each edge e (also called arc) is an ordered pair.



Undirected graph with 5 vertices and 6 edges



Digraph with 6 vertices and 11 edges

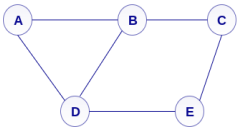
 U3.2

Representation of Graphs in Computer

- Array-based Representation
  - Using Adjacency Matrix
- Linked Representation
  - Using Adjacency List

 U3.3

Array-based (2-D Array) Representation



Undirected Graph

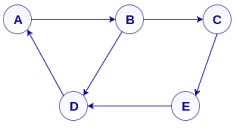
	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Adjacency Matrix

Matrix Representation of an Undirected Graph

 U3.4

Array-based (2-D Array) Representation



Directed Graph

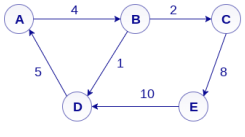
	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0

Adjacency Matrix

Matrix Representation of a Direct Graph

 U3.5

Array-based (2-D Array) Representation



Weighted Directed Graph

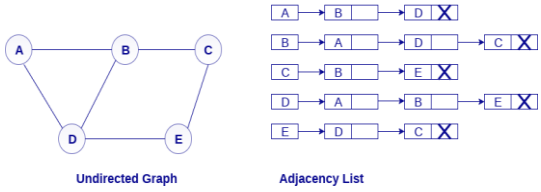
	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

Adjacency Matrix

Matrix Representation of a Weighted Graph

 U3.6

Adjacency List Representation

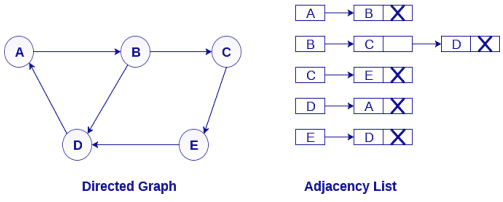


Undirected Graph Adjacency List

Adjacency List Representation of an Undirected Graph

Note: Adjacency List implementation needs Array and Linked List

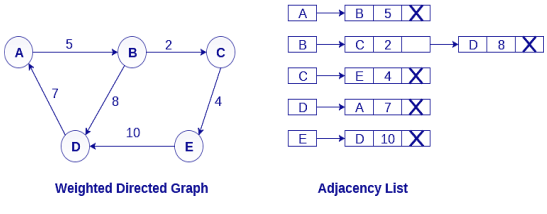
Adjacency List Representation



Directed Graph Adjacency List

Adjacency List Representation of a Directed Graph

Adjacency List Representation



Weighted Directed Graph Adjacency List

Adjacency List Representation of a Weighted Graph

Operations of Graph

- Insertion
  - There are two major components of a graph – Vertex and Edge. Therefore, a node or an edge or both can be inserted into an existing graph.
- Deletion
  - Similarly, a node or an edge or both can be deleted from an existing graph.
- Traversal
  - A graph may be traversed for many purposes – to search a path, to search a shortest path b/w two given nodes, etc.

 U3.10

Insertion Operation

- Insertion of a vertex and its associated edge with other vertices in an adjacency matrix involves:
  - Add a row for the new vertex
  - Add a column for the new vertex
  - Make appropriate entries into the rows and columns of the matrix.

 U3.11

Algorithm: addVertex() – Undirected Graph

```
Algorithm addVertex()
{
    lastRow = lastRow + 1;
    lastCol = lastCol + 1;
    adjMat[lastRow][0] = verTex;
    adjMat[0][lastCol] = verTex;
    Set all elements of last row = 0;
    Set all elements of last col = 0;
}
```

 U3.12

Algorithm: addEdge() – Undirected Graph

```
// A new edge (v1, v2) is added to matrix with entry 1.
Algorithm addEdge()
{
    Find row corresponding to v1, i.e., rowV1;
    Find col corresponding to v2, i.e., colV2;
    adjMat[rowV1][colV2] = 1;
    adjMat[colV2][rowV1] = 1;
}
```

 U3.13

Algorithm: addVertex() – Directed Graph

```
Algorithm addVertex()
{
    lastRow = lastRow + 1;
    lastCol = lastCol + 1;
    adjMat[lastRow][0] = verTex;
    adjMat[0][lastCol] = verTex;
    Set all elements of last row = 0;
    Set all elements of last col = 0;
}
```

 U3.14

Algorithm: addEdge() – Directed Graph

```
// A new edge (v1, v2) is added to matrix with entry 1.
Algorithm addEdge()
{
    Find row corresponding to v1, i.e., rowV1;
    Find col corresponding to v2, i.e., colV2;
    adjMat[rowV1][colV2] = 1;
}
```

 U3.15

Delete Operation

- Deletion of a vertex and its associated edge involves:
  - Delete the row corresponding to the vertex
  - Delete the col corresponding to the vertex
  - **Mark 0 in relation with other vertices (if other vertices are adjacent to the deleted vertex).**

 U3.16

Algorithm: delVertex() – Undirected Graph

```
Algorithm delVertex()
{
  Find row corresponding to verTex and set all ist elements = 0;
  Find col corresponding to verTex and set all ist elements = 0;
}
```

 U3.17

Algorithm: delEdge() – Undirected Graph

```
Algorithm delEdge()
{
    Find row corresponding to v1, i.e., rowV1;
    Find row corresponding to v2, i.e., colV2;
    adjMat[rowV1][colV2] = 0;
    adjMat[colV2][rowV1] = 0;
}
```

 U3.18

Graph Traversal

- Depth First Search
- Breadth First Search
- Spanning Tree
  - Minimum Cost Spanning Tree
    - Prim's Algorithm
    - Kruskal's Algorithm

Graph Traversal: BFS

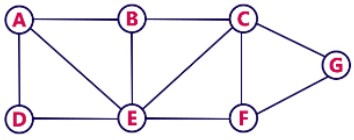
- Breadth First Search (BFS) is an algorithm for traversing or searching graph data structures.
  - It starts at the root (selecting some arbitrary node as the root) and explores the neighbor nodes first, before moving to the next level neighbors.

Breadth First Search (BFS)

- **Step 1:** Define a Queue of size total number of vertices in the graph.
- **Step 2:** Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
- **Step 3:** Visit all the **adjacent** vertices of the vertex which is at front of the Queue which is not visited and insert them into the Queue.
- **Step 4:** When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.
- **Step 5:** Repeat step 3 and 4 until queue becomes empty.
- **Step 6:** When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph

BFS with Queue: Example

Consider the following example graph to perform BFS traversal

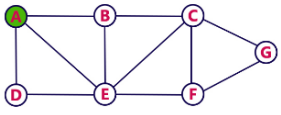


 U3.22

BFS with Queue: Example

Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.



Queue

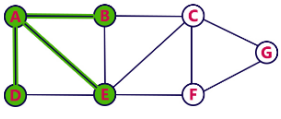
A						

 U3.23

BFS with Queue: Example

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..



Queue

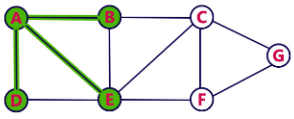
	D	E	B			

 U3.24

BFS with Queue: Example

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.



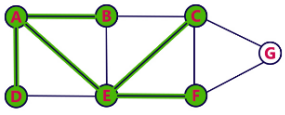
Queue

		E	B			

BFS with Queue: Example

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.



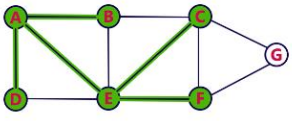
Queue

			B	C	F	

BFS with Queue: Example

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.



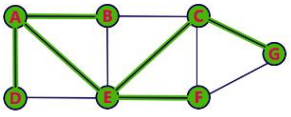
Queue

				C	F	

BFS with Queue: Example

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.



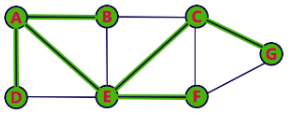
Queue

					F	G

BFS with Queue: Example

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.



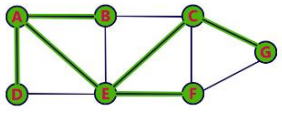
Queue

						G

BFS with Queue: Example

Step 8:

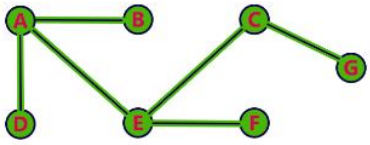
- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue

BFS with Queue: Example

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



 U3.31

Graph Traversal: DFS

- Depth First Search (DFS) is an algorithm for traversing or searching graph data structures.
  - It starts at the root (selecting some arbitrary node as the root) and explores as far as possible along each branch before backtracking.

 U3.32

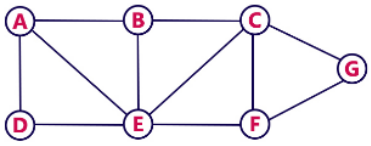
Depth First Search using Stack

- **Step 1:** Define a Stack of size total number of vertices in the graph.
- **Step 2:** Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
- **Step 3:** Visit any one of the **adjacent** vertex of the vertex which is at top of the stack which is not visited and push it on to the stack.
- **Step 4:** Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.
- **Step 5:** When there is no new vertex to be visit then use **backtracking** and pop one vertex from the stack.
- **Step 6:** Repeat steps 3, 4 and 5 until stack becomes Empty.
- **Step 7:** When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

 U3.33

Example of DFS

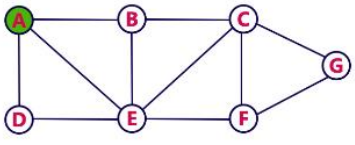
Consider the following example graph to perform DFS traversal



DFS with Stack: Example

Step 1:

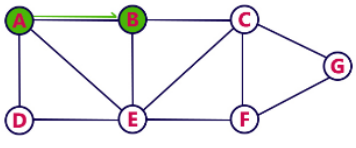
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



DFS with Stack: Example

Step 2:

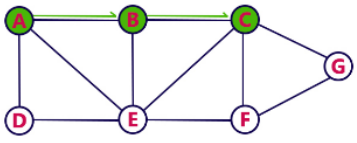
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



DFS with Stack: Example

Step 3:

- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

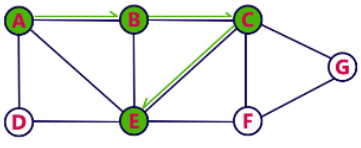


 U3.37

DFS with Stack: Example

Step 4:

- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

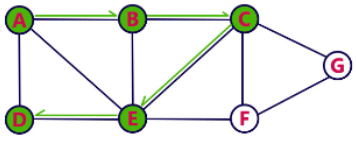


 U3.38

DFS with Stack: Example

Step 5:

- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

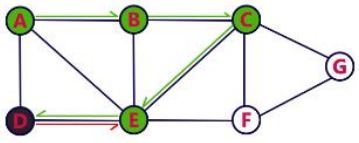


 U3.39

DFS with Stack: Example

Step 6:

- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

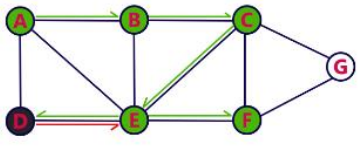


 U3.40

DFS with Stack: Example

Step 7:

- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

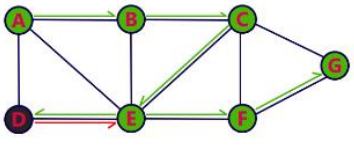


 U3.41

DFS with Stack: Example

Step 8:

- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

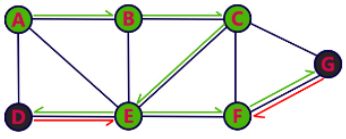


 U3.42

DFS with Stack: Example

Step 9:

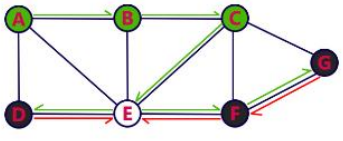
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.



DFS with Stack: Example

Step 10:

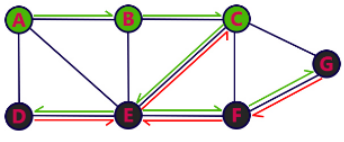
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.



DFS with Stack: Example

Step 11:

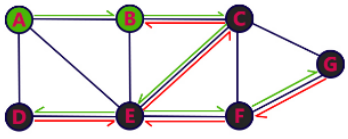
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.



DFS with Stack: Example

Step 12:

- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

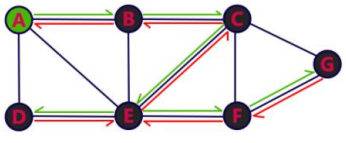


 U3.46

DFS with Stack: Example

Step 13:

- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

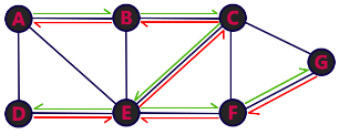


 U3.47

DFS with Stack: Example

Step 14:

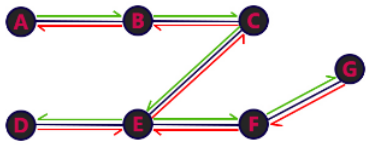
- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.



 U3.48

DFS with Stack: Example

- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



BFS and DFS in Directed Graphs

- Similar to undirected graphs, the same processes work for directed graphs.
- The only difference is that when exploring a vertex v, we only want to look at edges (v,w) going out of v; we ignore the other edges coming into v.
- BFS finds shortest (link-distance) paths from a single source vertex to all other vertices.

Spanning Tree

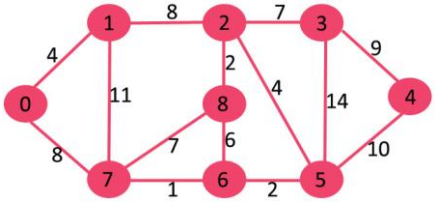
- Given a connected and undirected graph, a **spanning tree** of that graph is a sub-graph that is a tree and connects all the vertices together.
- A single graph can have many different spanning trees.
- A **minimum spanning tree** (MST) for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree.
  - Prim's Algorithm
  - Kruskal's Algorithm

Steps for finding MST using Kruskal's Algo.

Let G be a graph with V vertices:

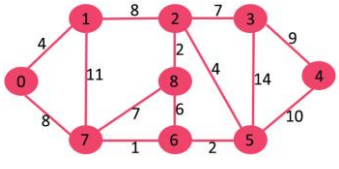
1) Sort all the edges in increasing order of their weight.
2) Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far.
   a) If cycle is not formed, include this edge.
   b) Else, discard it.
3) Repeat step 2 until there are (V-1) edges in the spanning tree.

MST using Kruskal's Algo.: Example



The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

MST using Kruskal's Algo.: Example 1 (contd...)



After sorting:

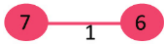
Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

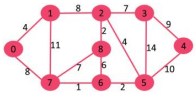
MST using Kruskal's Algo.: Example 1 (contd...)

After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

1. Pick edge 7-6: No cycle is formed, include it..



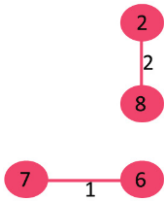


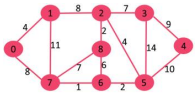
MST using Kruskal's Algo.: Example 1 (contd...)

After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

2. Pick edge 8-2: No cycle is formed, include it.



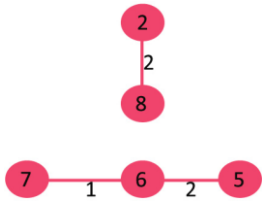


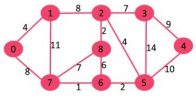
MST using Kruskal's Algo.: Example 1 (contd...)

After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

3. Pick edge 6-5: No cycle is formed, include it.



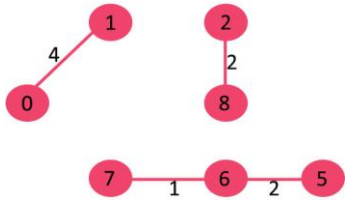


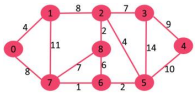
MST using Kruskal's Algo.: Example 1 (contd...)

After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

4. Pick edge 0-1: No cycle is formed, include it.



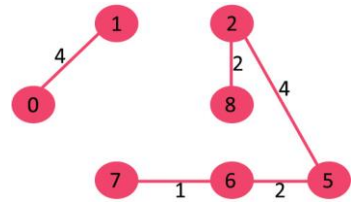


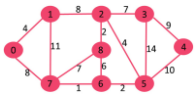
MST using Kruskal's Algo.: Example 1 (contd...)

After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

5. Pick edge 2-5: No cycle is formed, include it.



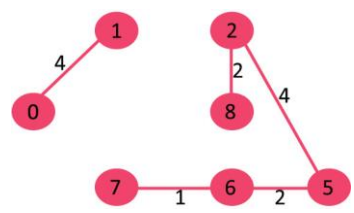


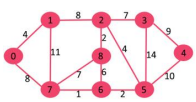
MST using Kruskal's Algo.: Example 1 (contd...)

After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

6. Pick edge 8-6: Since including this edge results in cycle, discard it.



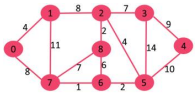


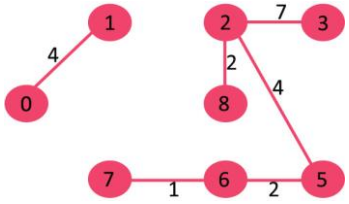
MST using Kruskal's Algo.: Example 1 (contd...)

After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

7. Pick edge 2-3: No cycle is formed, include it



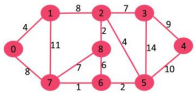


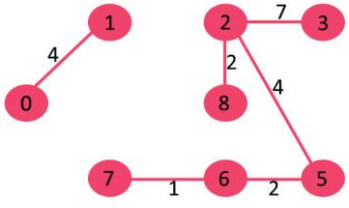
MST using Kruskal's Algo.: Example 1 (contd...)

After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

8. Pick edge 7-8: Since including this edge results in cycle, discard it.



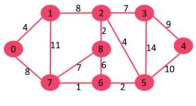


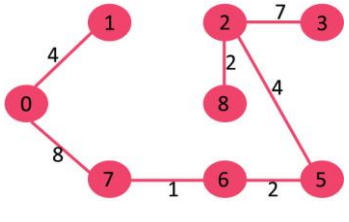
MST using Kruskal's Algo.: Example 1 (contd...)

After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

9. Pick edge 0-7: No cycle is formed, include it.



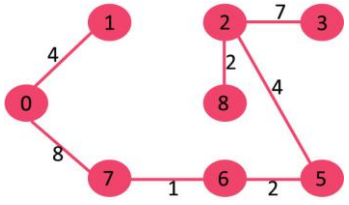


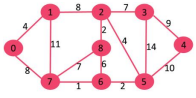
MST using Kruskal's Algo.: Example 1 (contd...)

After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

10. Pick edge 1-2: Since including this edge results in cycle, discard it.



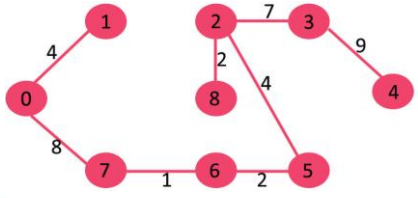


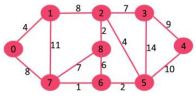
MST using Kruskal's Algo.: Example 1 (contd...)

After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

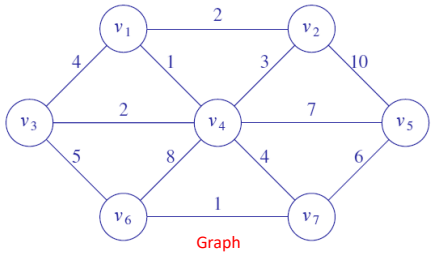
11. Pick edge 3-4: No cycle is formed, include it.





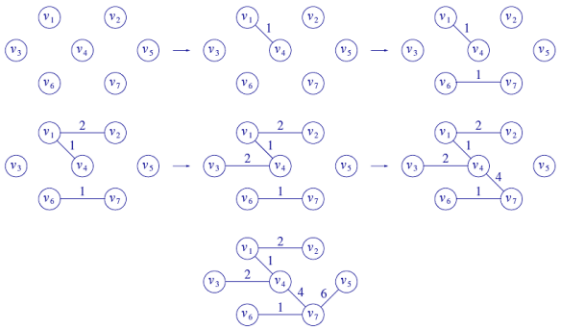
Since the number of edges included equals (V – 1), the algorithm stops here.

MST using Kruskal's Algo.: Example 2



Graph

MST using Kruskal's Algo.: Example 2 (contd...)



Minimum Cost Spanning Tree

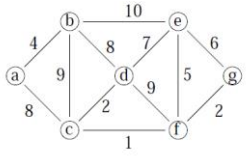
Steps for finding MST using Prim's Algo.

Step 0: Choose any element r; set $S = \{r\}$ and $A = \emptyset$. (Take r as the root of our spanning tree.)

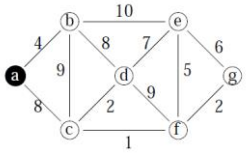
Step 1: Find a lightest edge such that one endpoint is in S and the other is in $V \setminus S$. Add this edge to A and its (other) endpoint to S.

Step 2: If $V \setminus S = \emptyset$, then stop & output (minimum) spanning tree (S, A). Otherwise go to Step 1.

MST using Prim's Algo.: Example 1



Connected graph



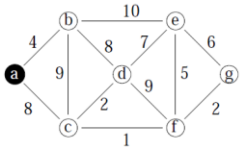
Step 0

S={a}

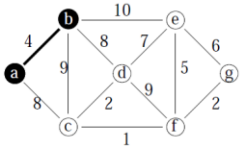
V \ S = {b,c,d,e,f,g}

lightest edge = {a,b}

MST using Prim's Algo.: Example 1 (contd...)

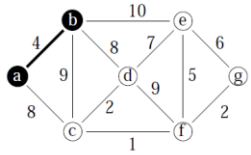


Step 1.1 before
S={a}
V \ S = {b,c,d,e,f,g}
A={}
lightest edge = {a,b}

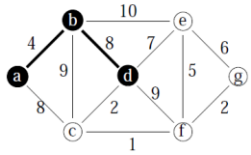


Step 1.1 after
S={a,b}
V \ S = {c,d,e,f,g}
A={{a,b}}
lightest edge = {b,d}, {a,c}

MST using Prim's Algo.: Example 1 (contd...)

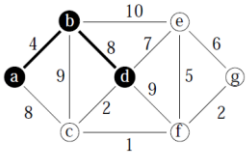


Step 1.2 before
S={a,b}
V \ S = {c,d,e,f,g}
A={{a,b}}
lightest edge = {b,d}, {a,c}

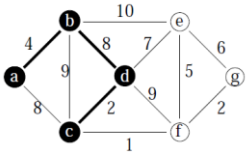


Step 1.2 after
S={a,b,d}
V \ S = {c,e,f,g}
A={{a,b},{b,d}}
lightest edge = {d,c}

MST using Prim's Algo.: Example 1 (contd...)

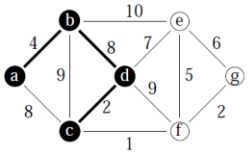


Step 1.3 before
S={a,b,d}
V \ S = {c,e,f,g}
A={{a,b},{b,d}}
lightest edge = {d,c}



Step 1.3 after
S={a,b,c,d}
V \ S = {e,f,g}
A={{a,b},{b,d},{c,d}}
lightest edge = {c,f}

MST using Prim's Algo.: Example 1 (contd...)



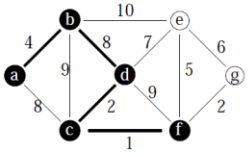
Step 1.4 before

S={a,b,c,d}

V \ S = {e,f,g}

A={{a,b},{b,d},{c,d}}

lightest edge = {c,f}



Step 1.4 after

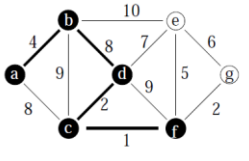
S={a,b,c,d,f}

V \ S = {e,g}

A={{a,b},{b,d},{c,d},{c,f}}

lightest edge = {f,g}

MST using Prim's Algo.: Example 1 (contd...)



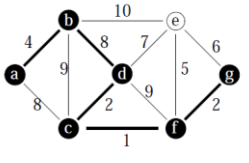
Step 1.5 before

S={a,b,c,d,f}

V \ S = {e,g}

A={{a,b},{b,d},{c,d},{c,f}}

lightest edge = {f,g}



Step 1.5 after

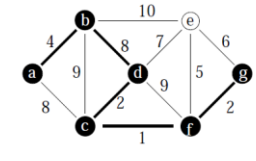
S={a,b,c,d,f,g}

V \ S = {e}

A={{a,b},{b,d},{c,d},{c,f},
{f,g}}

lightest edge = {f,e}

MST using Prim's Algo.: Example 1 (contd...)



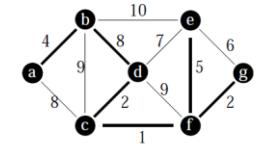
Step 1.6 before

S={a,b,c,d,f,g}

V \ S = {e}

A={{a,b},{b,d},{c,d},{c,f},
{f,g}}

lightest edge = {f,e}



Step 1.6 after

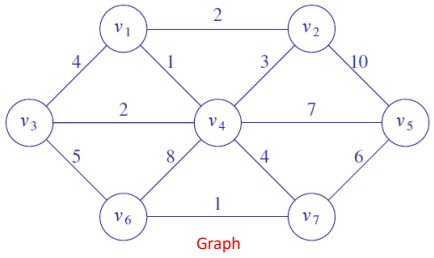
S={a,b,c,d,e,f,g}

V \ S = {}

A={{a,b},{b,d},{c,d},{c,f},
{f,g},{f,e}}

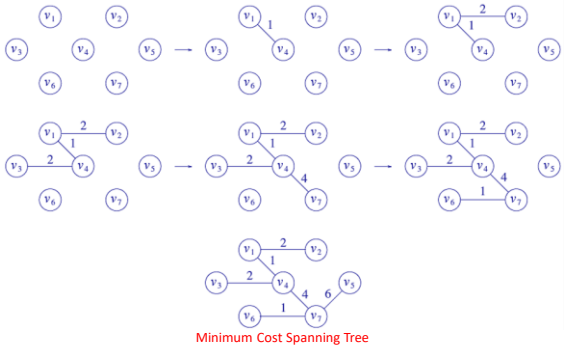
MST completed

MST using Prim's Algo.: Example 2



Graph

MST using Prim's Algo.: Example 2 (contd...)



Minimum Cost Spanning Tree

Prim's Algorithm (Programming) for MST

Let G be a graph with **V** vertices:

1) Create an array **Parent[]** of size V and initialize it with **NIL**.
2) Create a Min Heap of size V. Let the Min Heap be **H**.
3) Insert all vertices to **H** such that the key value of starting vertex is 0 and key value of other vertices is infinite.
4) While **H** is not empty
   a) u = extractMin(H).
   b) For every adjcent **v** of **u**,
      if v is in H
      (i) Update key value of **v** in **H** if weight of edge **u - v** is smaller than current key value of **v**.
      (ii) Parent[v] = u

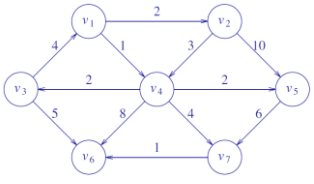
Shortest-Path Problems for Graphs

- **Single-Source Shortest-Path Problem:**
  - Given a (di)graph and a distinguished source vertex, s ϵ V, **determine the shortest path from the source vertex s to every other vertex in the graph.**
- **All-Pairs Shortest-Path Problem**
  - Given a directed graph, **determine the shortest path between all pairs of vertices in the weighted digraph.**

Single-Source Shortest-Path Problem

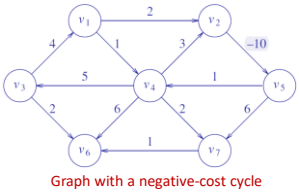
Given a (di)graph G = (V, E) and a distinguished **source vertex**, s ϵ V, **determine the shortest path from the source vertex s to every other vertex in the graph.**

- The shortest weighted path from v1 to v6 has a cost of 6 and goes from **v1** to **v4** to **v7** to **v6**.
- The shortest unweighted path from v1 to v6 has a cost of 2 and goes from **v1** to **v4** to **v6**.
- There is no path from **v6** to **v1**.



Single-Source Shortest-Path Problem (contd…)

- The path from v5 to v4 has cost **1**, but a shorter path exists by following the loop **v5**, **v4**, **v2**, **v5**, **v4**, which has cost **–5**.
- This path is still not the shortest, because we could stay in the loop arbitrarily long.

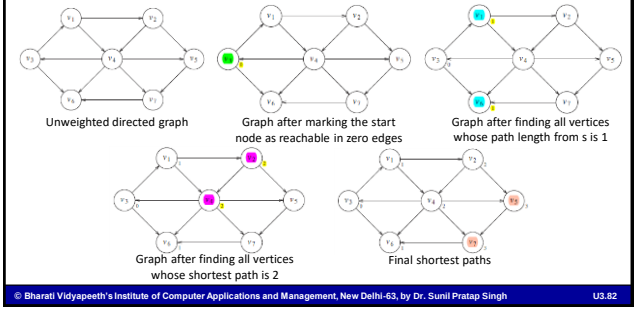


Graph with a negative-cost cycle

- Thus, the shortest path between these two points (v5 to v4) is undefined due to the loop.
- This loop is known as a **negative-cost cycle**; when one is present in the graph, the shortest paths are not defined.

Unweighted Shortest Paths

- In an unweighted (di)graph, the shortest paths from a single source vertex to all other vertices can be determined by following the procedure of **BFS**, which processes the vertices in increasing order of their distance from the source vertex.



Unweighted directed graph | Graph after marking the start node as reachable in zero edges | Graph after finding all vertices whose path length from s is 1

Graph after finding all vertices whose shortest path is 2 | Final shortest paths

Dijkstra's Algorithm

- **Purpose and Use Cases**
  - Find the shortest path from a node (called the "source node") to all other nodes in the graph.
  - This algorithm is used in GPS devices to find the shortest path between the current location and the destination.
  - It has broad applications in industry, specially in domains that require modeling networks.
- **History**
  - In 1959, the algorithm was published by Dr. Edsger W. Dijkstra, a brilliant Dutch Computer Scientist and Software Engineer.

Dijkstra's Algorithm (contd…)

- **Basics of the Algorithm**
  - The algorithm basically starts at the node that we choose (the source node) and it analyzes the graph to find the shortest path between that node and all the other nodes in the graph.
  - The algorithm keeps track of the currently known shortest distance from each node to the source node and it updates these values if it finds a shorter path.
  - Once the algorithm has found the shortest path between the source node and another node, that node is marked as "visited" and added to the path.
  - The process continues until all the nodes in the graph have been added to the path.

NOTE: This algorithm works for both directed and undirected graphs. It works only for connected graphs. The graph should not contain negative edge weights.

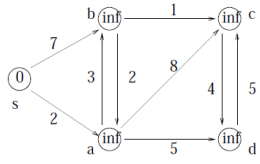
Dijkstra's Algorithm (Pseudocode)

```
Dijkstra(G,w,s)
{                                          % Initialize
   for (each u ∈ V)
   {
      d[u] = ∞;
      color[u] =white;
   }
   d[s] = 0;
   pred[s] = NIL;
   Q = (queue with all vertices);

   while (Non-Empty(Q))                    % Process all vertices
   {
      u = Extract-Min(Q);                  % Find new vertex
      for (each v ∈ Adj[u])
         if (d[u] + w(u,v) < d[v])         % If estimate improves
         {
            d[v] = d[u] + w(u,v);          relax
            Decrease-Key(Q,v,d[v]);
            pred[v] = u;
         }
      color[u] =black;
   }
}
```

 U3.85

Shortest-Path using Dijkstra's Algo: Example



Step 0: Initialization.

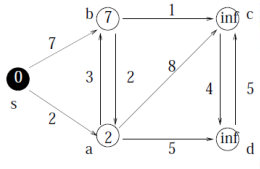
v	s	a	b	c	d
$d[v]$	0	∞	∞	∞	∞
$pred[v]$	nil	nil	nil	nil	nil
$color[v]$	W	W	W	W	W

Priority Queue:

v	s	a	b	c	d
$d[v]$	0	∞	∞	∞	∞

 U3.86

Shortest-Path using Dijkstra's Algo: Example



Step 1: As $Adj[s] = \{a, b\}$, work on a and b and update information.

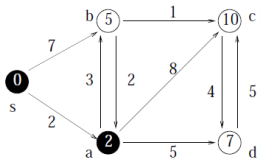
v	s	a	b	c	d
$d[v]$	0	2	7	∞	∞
$pred[v]$	nil	s	s	nil	nil
$color[v]$	B	W	W	W	W

Priority Queue:

v	a	b	c	d
$d[v]$	2	7	∞	∞

 U3.87

Shortest-Path using Dijkstra's Algo: Example



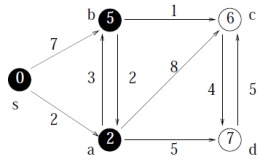
Step 2: After Step 1, a has the minimum key in the priority queue. As $Adj[a] = \{b, c, d\}$, work on b, c, d and update information.

v	s	a	b	c	d
$d[v]$	0	2	5	10	7
$pred[v]$	nil	s	a	a	a
$color[v]$	B	B	W	W	W

Priority Queue:

v	b	c	d
$d[v]$	5	10	7

Shortest-Path using Dijkstra's Algo: Example



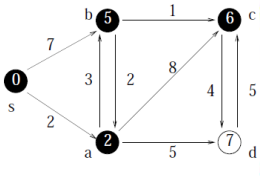
Step 3: After Step 2, b has the minimum key in the priority queue. As $Adj[b] = \{a, c\}$, work on a, c and update information.

v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	nil	s	a	b	a
$color[v]$	B	B	B	W	W

Priority Queue:

v	c	d
$d[v]$	6	7

Shortest-Path using Dijkstra's Algo: Example



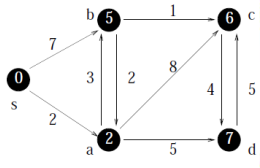
Step 4: After Step 3, c has the minimum key in the priority queue. As $Adj[c] = \{d\}$, work on d and update information.

v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	nil	s	a	b	a
$color[v]$	B	B	B	B	W

Priority Queue:

v	d
$d[v]$	7

Shortest-Path using Dijkstra's Algo: Example



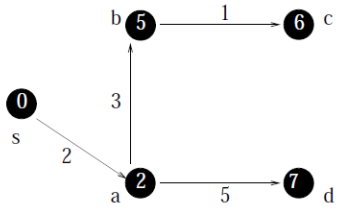
Step 5: After Step 4, d has the minimum key in the priority queue. As $Adj[d] = \{c\}$, work on c and update information.

v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	nil	s	a	b	a
$color[v]$	B	B	B	B	B

Priority Queue: $Q = \emptyset$.

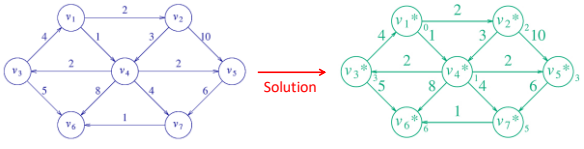
The array pred[v] is used to build the shortest-path tree.

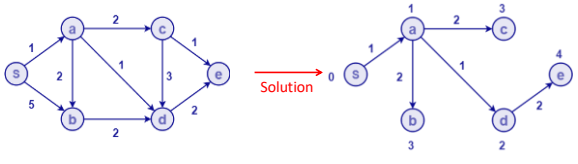
Shortest-Path using Dijkstra's Algo: Example



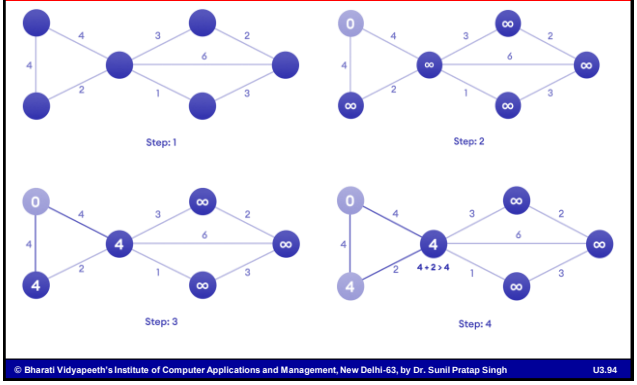
v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	nil	s	a	b	a

More Problems related to Dijkstra's Algo.

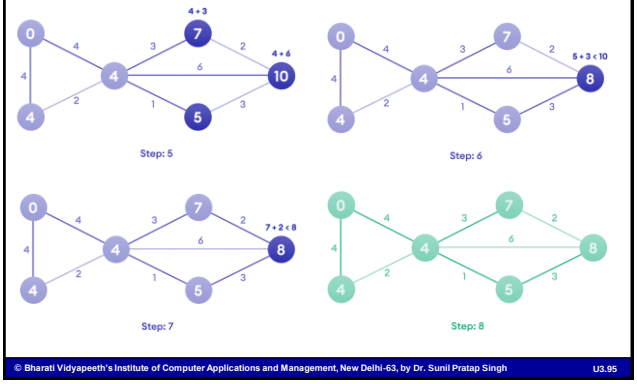




Ex. of Dijkstra's Algo. for Undirected Graph



Ex. of Dijkstra's Algo. for Undirected Graph



All-Pairs Shortest-Path Problem

- Given a weighted digraph G = (V, E) with a weight function w: E → R, where R is the set of real numbers, determine the length of the shortest path (i.e., distance) between all pairs of vertices in G.
- **Solution 1:** Assume no negative edges. Run Dijkstra's algorithm, n times, once with each vertex as source. What's the time complexity?
- **Solution 2:** Floyd-Warshall algorithm (dynamic programming) with time complexity $O(n^3)$, where n is the number of vertices (|V|) in G.

Floyd-Warshall's Algorithm: Background

- Floyd-Warshall's algorithm is a graph analysis algorithm for finding shortest paths in a weighted, directed graph.
- A single execution of the algorithm will find the shortest paths between all pairs of vertices.
- This algorithm compares all possible paths through the graph between each pair of vertices.

Floyd-Warshall's Algorithm: Background

- Let $d_{ij}^{(k)}$ be the length of the shortest path from i to j such that *all* intermediate vertices on the path (if any) are in set $\{1, 2, ..., k\}$.
- $d_{ij}^{(0)}$ is set to be w_{ij}, i.e., no intermediate vertex.
- Let $D^{(k)}$ be the n by n matrix $[d_{ij}^{(k)}]$.
- Claim: $d_{ij}^{(n)}$ is the shortest distance from i to j, with the intermediate vertices set $\{1, 2, ..., n\}$. So our aim is to compute $D^{(n)}$.

Observation: For a shortest path from i to j such that any intermediate vertices on the path are chosen from the set {1, 2, ..., k}, there are two possibilities:

1. k is not a vertex on the path, the shortest such path has length $d_{ij}^{(k-1)}$.
2. k is a vertex on the path, the shortest such path has length $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

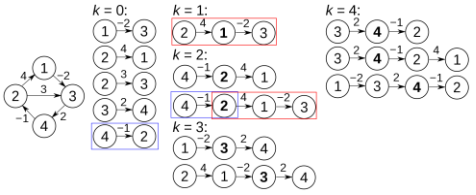
Combining the above two cases we get:

$$d_{ij}^{(k)} = \min \{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$$

Dijkstra's Algorithm (Pseudocode)

```
1 Floyd-Warshall(w, n) // w: weights, n: number of vertices
2 {
3   for i = 1 to n do       // initialize, D(0) = [wij]
4     for j = 1 to n do
5     {
6       d[i, j] = w[i, j];
7     }
8   for k = 1 to n do       // Compute D(k) from D(k-1)
9     for i = 1 to n do
10      for j = 1 to n do
11        if (d[i, k] + d[k, j] < d[i, j])
12        {
13          d[i, j] = d[i, k] + d[k, j];
14        }
15  return d[1..n, 1..n];
16 }
```


Shortest-Path using Floyd–Warshall: Example



k = 0	j			
i	1	2	3	4
1	0	∞	−2	∞
2	4	0	3	∞
3	∞	∞	0	2
4	∞	−1	∞	0

k = 1	j			
i	1	2	3	4
1	0	∞	−2	∞
2	4	0	2	∞
3	∞	∞	0	2
4	∞	−1	∞	0

k = 2	j			
i	1	2	3	4
1	0	∞	−2	∞
2	4	0	2	∞
3	∞	∞	0	2
4	3	−1	1	0

k = 3	j			
i	1	2	3	4
1	0	∞	−2	0
2	4	0	2	4
3	∞	∞	0	2
4	3	−1	1	0

k = 4	j			
i	1	2	3	4
1	0	−1	−2	0
2	4	0	2	4
3	5	1	0	2
4	3	−1	1	0

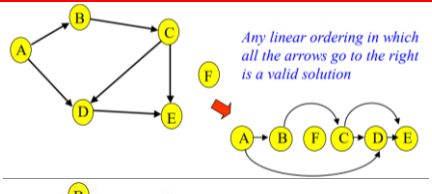
 U3.100

Topological Sort

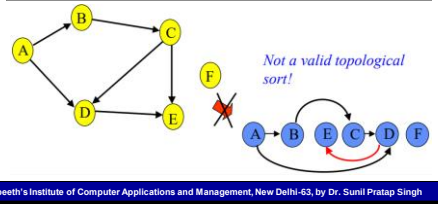
- Directed Acyclic Graph **(DAG)**
  - A directed acyclic graph is a directed graph with no cycles.
  - They are often used to represent dependence constraints of some type.
- Topological Sort of a DAG
  - The topological sort a DAG (V, E) is a total ordering, $v_1 < v_2 \ldots < v_n$ of the vertices in V such that for any edge $(v_i , v_j) \in E$, if $j > i$.
  - Topological Sort is a linear ordering of the vertices in such a way that if there is an edge in the DAG going from vertex 'u' to vertex 'v', then 'u' comes before 'v' in the ordering.
  - There may exist multiple different topological orderings for a given DAG.

 U3.101

Topological Sort (Examples)



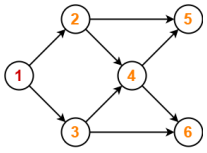
Any linear ordering in which all the arrows go to the right is a valid solution



Not a valid topological sort!

 U3.102

Topological Sort (contd…)

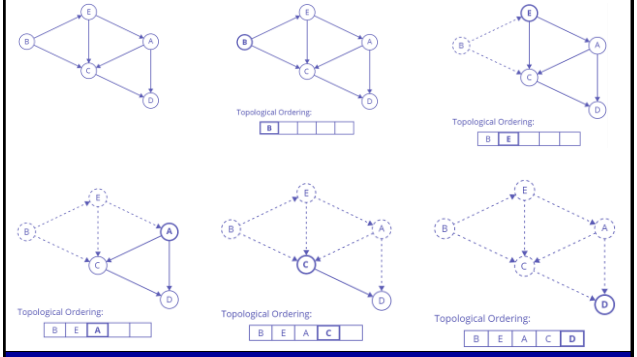


Topo Sorts
1 2 3 4 5 6
1 2 3 4 6 5
1 3 2 4 5 6
1 3 2 4 6 5

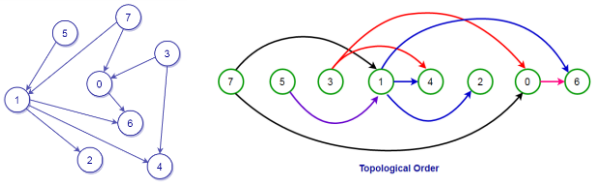
Steps to Find Topological Sort from DAG

1. Identify vertices that have no incoming edge, and select one such vertex.
   - In-degrees of these vertices is zero.
   - If no such edges, graph has cycles (cyclic graph).
2. Delete this vertex of in-degree zero and all its outgoing edges from the graph.
   - Place the deleted vertex in the output.
3. Repeat Steps 1 and Step 2 until graph is empty.

Example: Topological Sort from a DAG



Example: Topological Sort from a DAG



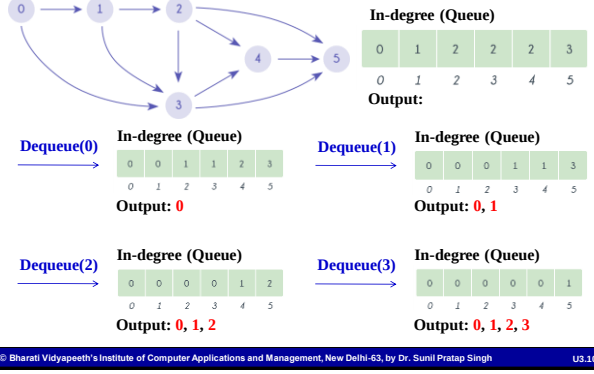
 U3.106

Implementation of Topo Sort using Queue

1. Initialize a queue with each vertex's in-degree.
2. While there are vertices remaining in the queue:
   a) Dequeue and output a vertex.
   b) Reduce in-degree of all vertices adjacent to it by 1.

 U3.107

Ex.: Topo Sort Implementation using Queue



 U3.108

Ex.: Topo Sort Implementation (contd…)

Dequeue(4)

In-degree (Queue)

0	0	0	0	0	0
0	1	2	3	4	5

Output: 0, 1, 2, 3, 4

Dequeue(5)

In-degree (Queue)

0	0	0	0	0	0
0	1	2	3	4	5

Output: 0, 1, 2, 3, 4, 5

Bibliography

- E. Horowitz and S. Sahani, "Fundamentals of Data Structures in C"
- Mark Allen Weiss, "Data Structures and Algorithm Analysis in C"
- R. S. Salaria, "Data Structure & Algorithms Using C"
- Schaum's Outline Series, "Data Structure"

Data and File Structures

Unit – 4

(File Structures)

by
Dr. Sunil Pratap Singh
(Assistant Professor, BVICAM, New Delhi)

2021

 U4.1

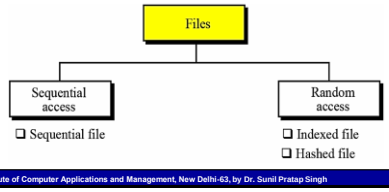
File

- A **file** is a collection of data stored on mass storage (e.g., disk or tape).
  - The data is subdivided into **records** (e.g., student information).
  - Each record contains a number of **fields** (e.g., roll number, name).
  - One (or more) field is the **key** field (e.g., roll number).

 U4.2

File Organizations

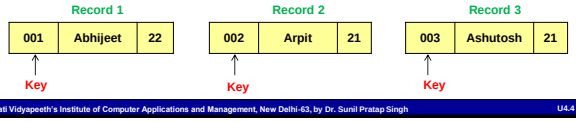
- A **file organization** refers to the way records are arranged on a storage device.
- How best the files be arranged for easy of access?



 U4.3

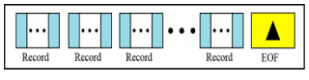
Sequential Files

- A sequential file is one in which records can only be accessed one after another from beginning to end.
- This file organization is the simplest way to store and retrieve records of a file.
- In this file, data records are stored in some specific sequence, e.g., order of arrival, value of key field, etc.



Sequential Files

- The records of a sequential file cannot be accessed at random, i.e., to access the n^{th} record, one must traverse the preceding (n-1) records.



Sequential File Organization

- In sequential file organization, the actual storage of records might or might not be sequential:
  - On a tape, it usually is.
  - On a disk, it might be distributed across sectors and the operating system would use a linked list of sectors to provide the illusion of sequentially.

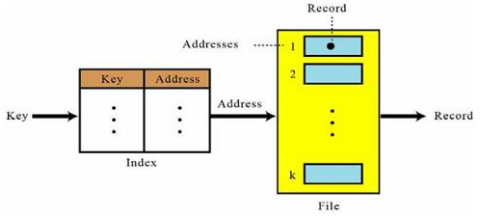
Sequential File Organization

- **Advantages:**
  - Easy to handle
  - Involve no overhead
  - Can be stored on tapes as well as disks
- **Disadvantages:**
  - Records can only be accessed in sequence
  - Time consuming

 U4.7

Indexed File

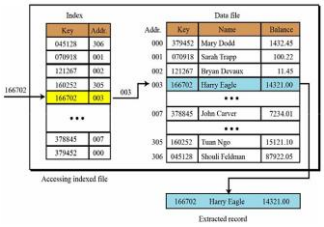
- To access a record in a file randomly, we need to know the address of the record.



 U4.8

Indexed File: Logical View

- An **indexed file** is made of a **data file**, which is a sequential file, and an **index**.
- The index itself is a **very small file** with only two fields: **the key of the sequential file** and **the address of the corresponding record on the disk**.
- The index is sorted based on the key values of the data files.



 U4.9

Indexed File: Accessing a Record

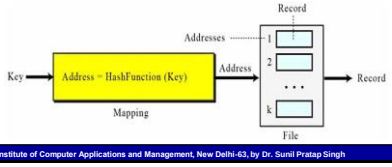
- Accessing a record in the file requires following steps:
  - The entire index file is loaded into main memory (the file is small and uses little memory).
  - The index entries are searched, using an efficient search algorithm such as a binary search, to find the desired key.
  - The address of the record is retrieved.
  - Using the address, the data record is retrieved and passed to the user.

Inverted File

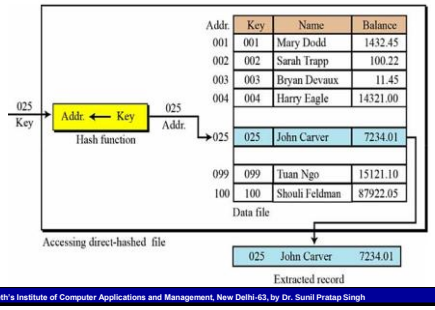
- One of the advantages of indexed files is that we can have **more than one index**, each with a different key.
- This type of indexed file is usually called an **inverted file**.

Direct File (Hashed File)

- A **hashed file** uses a mathematical function to map the key to the address.
- The user gives the key, the function maps the key to the address and passes it to the operating system, and the record is retrieved.



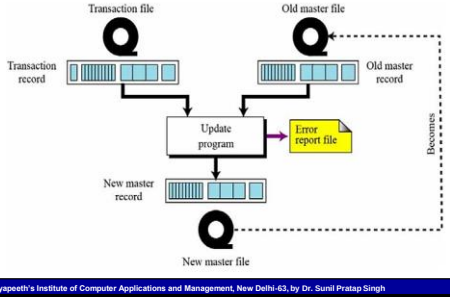
Process of Working in Direct File (Hashed File)



Updating a Sequential File

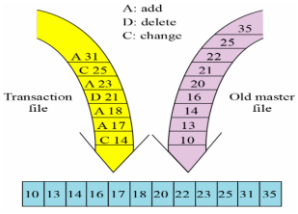
- Information that is kept on files needs to be modified as changes to the information on the file occur.
  - This process is called **UPDATING** and the files that are being update are usually called **MASTER FILES**.
- Updating a file can involve **ADDING, CHANGING or DELETING** records to/from the file.

Process of Updating a Sequential File



Example of Updating a Sequential File

- To make the updating process efficient, all files are sorted on the same key.



Error in Updating a Sequential File

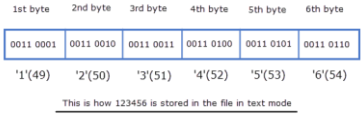
- Several cases may **create an error** and be reported in the error file:
  - If the transaction defines **adding a record that already exists** in the old master file (same key values).
  - If the transaction defines deleting or **changing a record that does not exist** in the old master file.

File Operations in C

- There are two distinct ways to perform file operations in C:
  - Low-level I/O Operations (uses UNIX system calls)
  - High-level I/O Operation (uses functions of C's standard I/O library)
- Data can be stored into files in two ways:
  - Text Mode
  - Binary Mode

Text Mode

- In text mode, data is stored as a line of characters where each character occupies 1 byte.
  - To store 123456 in a text file would take 6 bytes, 1 byte for each character.

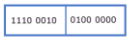


This is how 123456 is stored in the file in text mode

  - In the text mode, what gets stored in the memory is that binary equivalent of the ASCII number of the character.

Binary Mode

- In binary mode, data is stored on a disk in the same way as it is represented in computer memory.
  - Storing 123456 in a binary mode would take only 2 bytes.



This is how 123456 is stored in the file in binary mode

  - Hence by using binary mode, we can save a lot of disk space.

High Level I/O Functions in 'C'

- `fopen()`: Opens an existing/creates a new file for use.
- `fclose()`: Closes a file which has been opened for use.
- `fscanf()`: Reads a set of data values from a file.
- `fprintf()`: Writes a set of data values to a file.
- `getc()`: Reads a character from a file.
- `putc()`: Writes a character to a file.
- `getw()`: Reads an integer from a file.
- `putw()`: Writes an integer to a file.
- `fseek()`: Sets the position to a desired point in the file.
- `rewind()`: Sets the position to the beginning of the file.

Operations on a File in 'C'

```
FILE *fp;
fp = fopen("fileName", "mode");
```

- Mode specifies the purpose of opening of the file. It can be one of the following:
  - `r` opens the file for reading only
  - `r+` opens the existing file for both reading and writing. If file does not exist, NULL is returned.
  - `w` opens the file for writing only
  - `w+` opens the file for both writing and reading. If the file exist, the previous contents are overwritten by new one.
  - `a` opens the file for appending (or adding) data to file.
  - `a+` opens the file for reading and appending. If the file does not exist, a new file is created.

Operations on a File in 'C' (contd…)

- When trying to open a file, one of the following things may happen:
  - When the mode is 'writing', a file with the specified name is created if the file does not exist. The contents are deleted, if the file already exists.
  - When the purpose is 'appending', the file is opened with the current contents safe. A file with the specified name is created if the file does not exist.
  - If the purpose is 'reading', and if it file exists, then it is opened with the current contents safe otherwise an error occurs.

Input/Output Operations on a File

- Using `getc()` and `putc()`

```
putc(ch, fp);
ch = getc(fp);
```

  - `putc(ch, fp);` is used to write the character contained in the character variable **ch** to the file associated with **FILE** pointer **fp**.
  - `c = getc(fp);` is used to read a character from a file that has been opened in read mode.

Input/Output Operations on a File (contd…)

- Using `getw()` and `putw()`

```
putw(in, fp);
in = getw(fp);
```

  - `getw()` and `putw()` are integer-oriented functions.
  - These are similar to the `getc()` and `putc()` functions and are used to read and write integer values.
  - `putw(in, fp);` is used to write the integer contained in the character variable **in** to the file associated with **FILE** pointer **fp**.
  - `in = getw(fp);` is used to read a character from a file that has been opened in read mode.

Input/Output Operations on a File (contd…)

- Using `fprintf()` and `fscanf()`

```
fprintf(fp, "control string", list);
fscanf(fp, "control string", list);
```

  - `fprintf()` and `fscanf()` are identical to the printf() and scanf() functions, except of course that they work on files.
  - `fprintf()` and `fscanf()` can handle a group of mixed data simultaneously.

Input/Output Operations on a File

- Using `getc()` and `putc()`

```
putc(ch, fp);
ch = getc(fp);
```

- Using `getw()` and `putw()`

```
putw(in, fp);
in = getw(fp);
```

- Using `fprintf()` and `fscanf()`

```
fprintf(fp, "control string", list);
Example: fprintf(fp, "%s %d %f", item, number, price);
When the end of line is reached, fscanf() returns EOF.
```

Errors during I/O Operations

- During I/O operations, an error may occur due to the following reasons:
  - Trying to read beyond the end-of-file mark.
  - Trying to use a file that has not been opened.
  - Trying to perform an operation on a file, when the file is opened for another type of operation.
  - Opening a file with an invalid file name.
  - Attempting to write to a write-protected file.

Error Handling during I/O Operations

- `feof()` function can be used to test for an end of file condition.
  - It takes a FILE pointer as its only argument and returns a nonzero integer value if all of the data from the specified file has been read, and returns zero otherwise.

```
if(feof(fp) != 0)
  printf("End of data.\n");
```

- `ferror()` function reports the status of the file indicated.
  - It takes a FILE pointer as its argument and returns a nonzero integer if an error has been detected up to that point, during processing. It returns zero otherwise.

```
if(ferror(fp) != 0)
  printf("An error has occurred.\n");
```


Error Handling during I/O Operations (contd…)

- Whenever a file is opened using `fopen()` function, a file pointer is returned.
  - If the file cannot be opened for some reason, then the function returns a NULL pointer.
  - This facility can be used to test whether a file has been opened or not.

```
if(fp == NULL)
  printf("File could not be opened.\n");
```

Random Access to Files

- `ftell()` takes a file pointer and return a number of type long, that corresponds to the current position.
- This function is useful in saving the current position of a file, which can be used later in the program.
- It takes the following form: `n = ftell(fp);`
  - n would give the relative offset (in bytes) of the current position, which means that n bytes have already been read (or written).

Random Access to Files (contd…)

- `rewind()` takes a file pointer and resets the position to the start of the file.
  - The statement `rewind(fp); n = ftell(fp);` will assign **0** to **n** because the file position has been set to the start of the file by **rewind**.
- This function helps us in reading a file more than once, without having to close and open the file.
  - Whenever a file is opened for reading or writing, a rewind is done implicitly.
- Note: The first byte in the file is numbered as 0, second as 1, and so on.

Random Access to Files (contd…)

- `fseek()` function is used to move the file position to a desired location within the file.
- It takes the following form: fseek(fileptr, offset, position);
  - **fileptr** is a pointer to the file concerned; **offset** is a number or variable of type long; **position** is an integer number.
  - The offset specifies the number of positions (bytes) to be moved from the location specified by position.
  - The position can take one of the following three values: 0 (beginning of file), 1 (current position) and 3 (end of file).
  - The offset may be positive (move forwards) or negative (move backwards).

Examples of Operations of the fseek()

Statement	Meaning
fseek(fp,0L,0);	Go to the beginning. (Similar to rewind)
fseek(fp,0L,1);	Stay at the current position. (Rarely used)
fseek(fp,0L,2);	Go to the end of the file, past the last character of the file.
fseek(fp,m,0);	Move to (m+1)th byte in the file.
fseek(fp,m,1);	Go forward by m bytes.
fseek(fp,-m,1);	Go backward by m bytes from the current position.
fseek(fp,-m,2);	Go backward by m bytes from the end. (Positions the file to the mth character from the end.)

Input/Output Operations in Binary File

- `fread()` and `fwrite()` functions are commonly used to read and write binary data to and from the file respectively.
- **`fwrite`**`(void *ptr, int size, int n, FILE *fp);`
- **`fread`**`(void *ptr, int size, int n, FILE *fp);`
  - `ptr` points to the block of memory which contains the data items to be written.
  - `size` specifies the number of bytes of each item to be written.
  - `n` is the number of items to be written.
  - `fp` is a pointer to the file where data items will be written.
  - **On success, these functions returns the number of items successfully written/read to/from the file.**

Thank You